Windows Kernel Heap

Part 1: Segment heap in windows kernel

Angelboy

angelboy@chroot.org

@scwuaptx

Whoami

- Angelboy
  - Researcher at DEVCORE
  - CTF Player
    - HITCON / 217
  - Chroot
  - Co-founder of pwnable.tw
  - Speaker
    - HITB GSEC 2018/AVTokyo 2018/VXCON

PWNABLE.TW

DEVCORE SECURITY CONSULTING

Kernel Pool

- Non Paged Pool
  - Memory that will not be page out
- Paged Pool
  - Memory that will be page out
- Session Paged Pool
  - Memory that will be page out
  - Each session is independent

Memory Allocator in kernel

- ExAllocatePool/ExAllocatePoolWithTag
- ExFreePool/ExFreePoolWithTag
- Nt Heap
  - RtlpAllocateHeap
  - RtlpFreeHeap

Memory Allocator in kernel

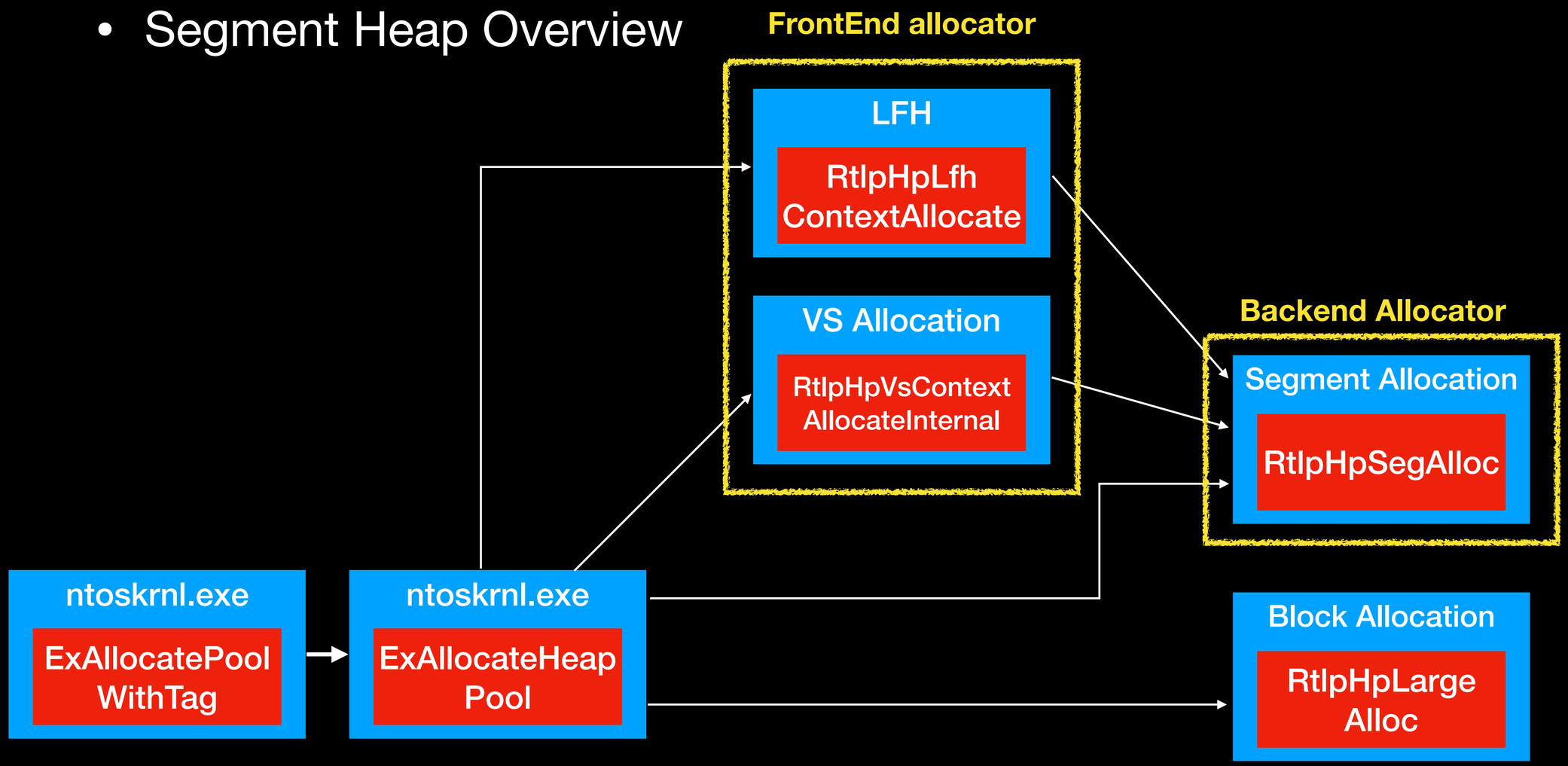
- ExAllocatePoolWithTag/ExFreePoolWithTag
  - Before 19H1 (1903)
    - Kernel Pool Allocator
  - From 19H1 ~ Now
    - Segment heap
  - Note
    - Segment heap in kernel is very similar to segment heap in userland
      - Some size and config are difference
      - We will focus on kernel in this slide
        - Windows 10 20H2

Memory Allocator in kernel

- Segment Heap Overview
  - Frontend Allocation
    - Low FragmentationHeap
    - Variable Size Allocation
  - Backend Allocation
    - Segment Allocation
  - Large Block Allocation

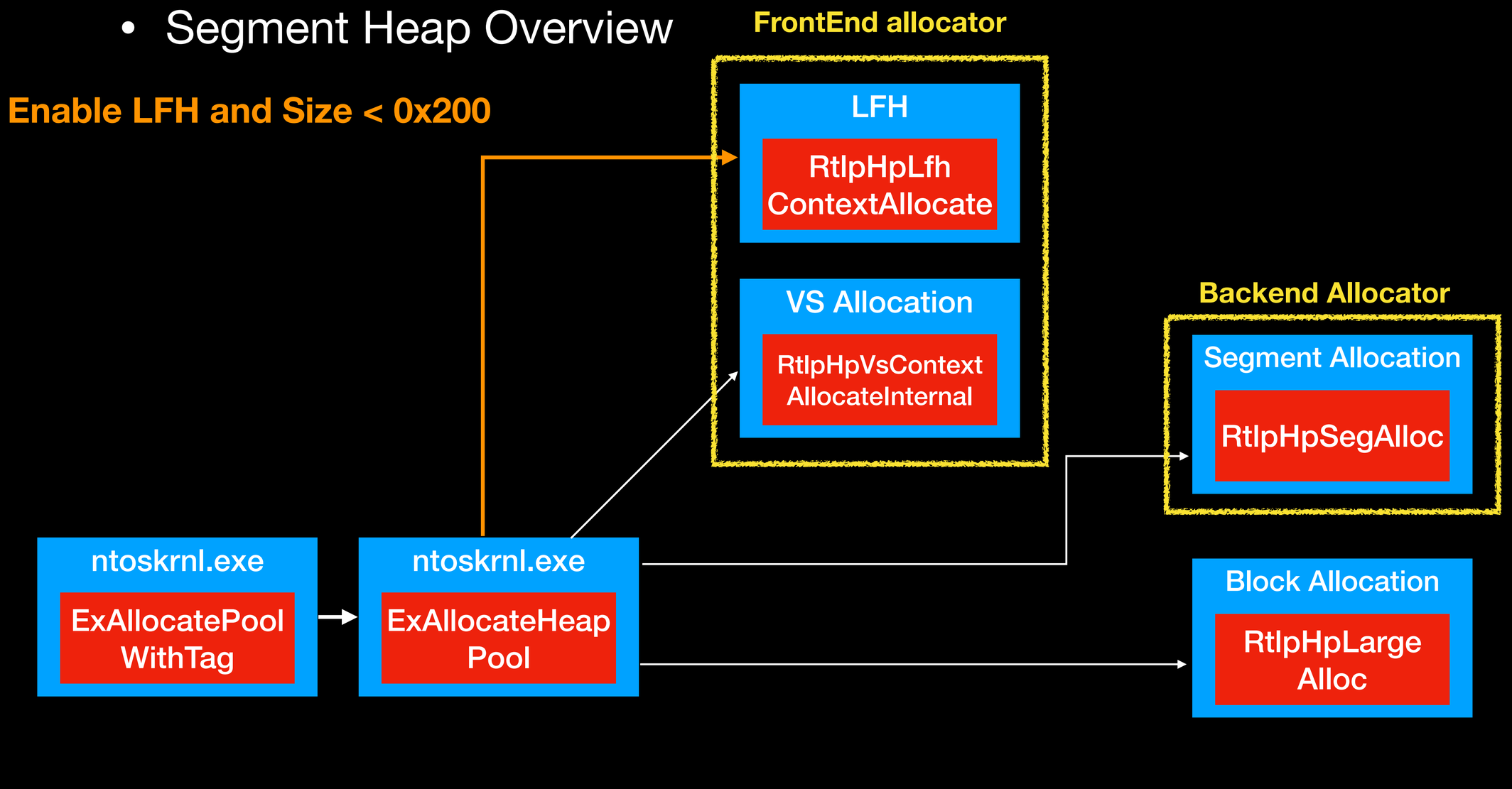
Memory Allocator in kernel

- Segment Heap Overview



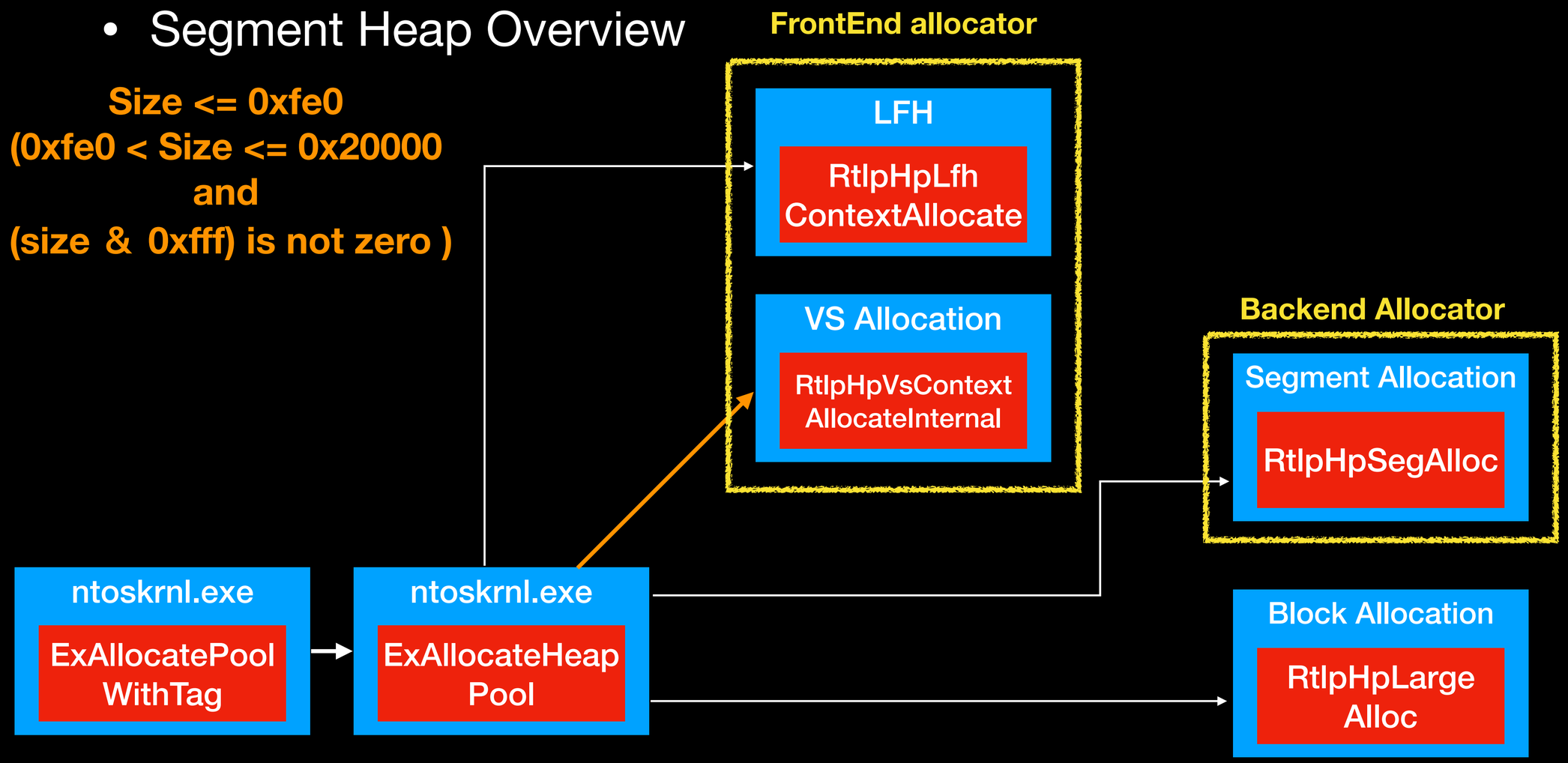
Memory Allocator in kernel

- Segment Heap Overview



Memory Allocator in kernel

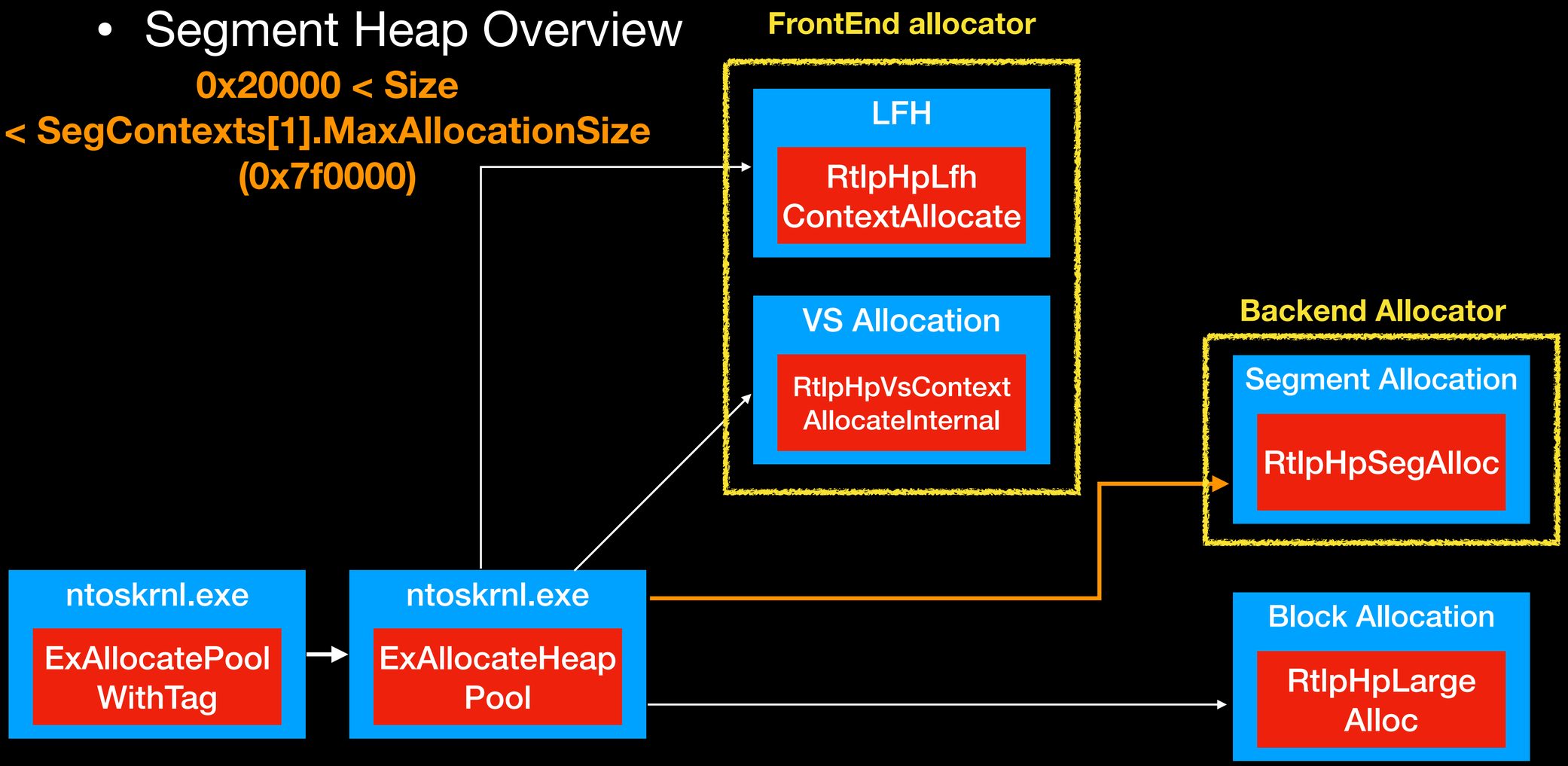
- Segment Heap Overview



Memory Allocator in kernel

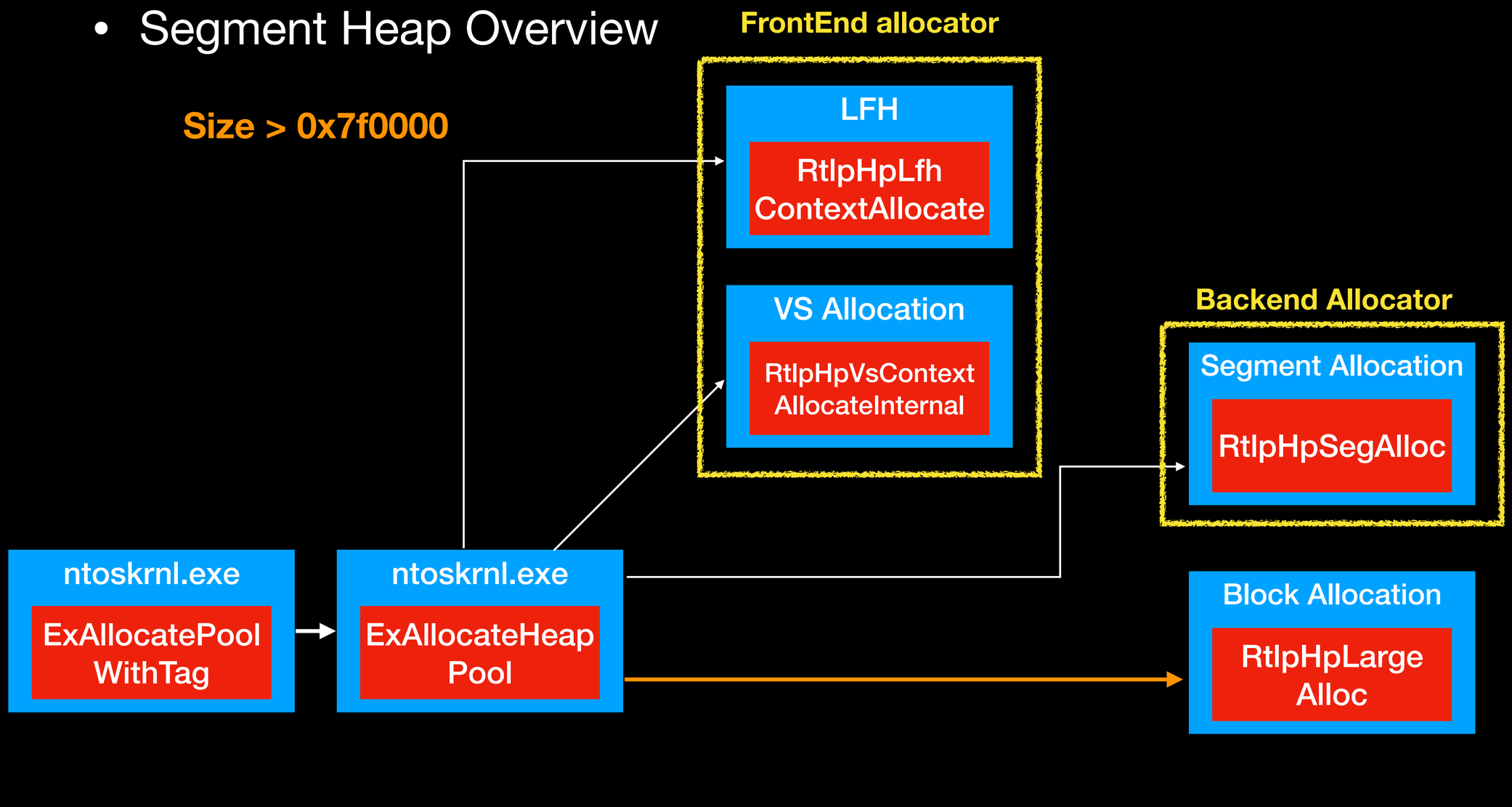
- Segment Heap Overview

0x20000 < Size < SegContexts[1].MaxAllocationSize (0x7f0000)



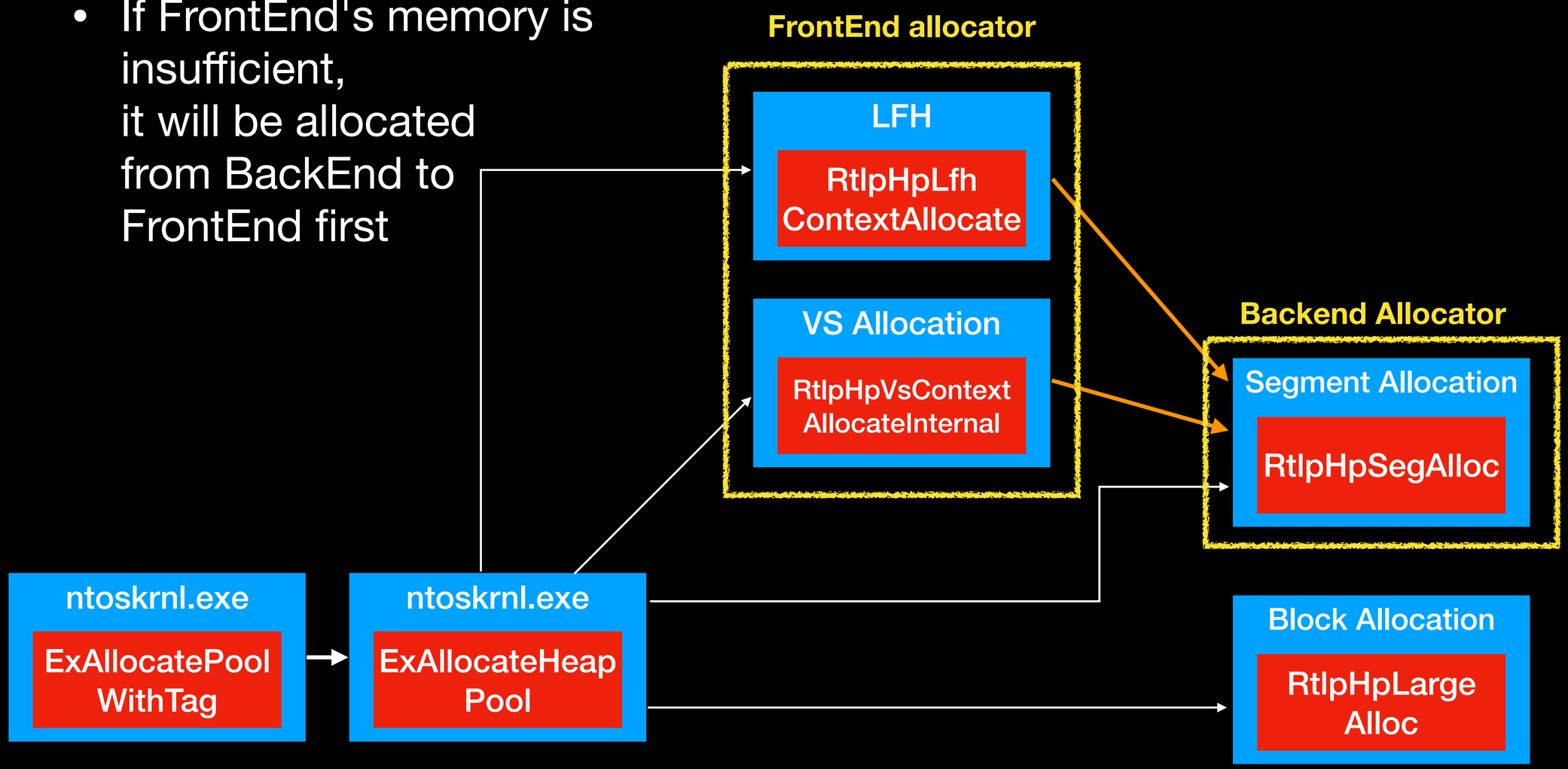
Memory Allocator in kernel

- Segment Heap Overview



Memory Allocator in kernel

- If FrontEnd's memory is insufficient,
  it will be allocated from BackEnd to FrontEnd first



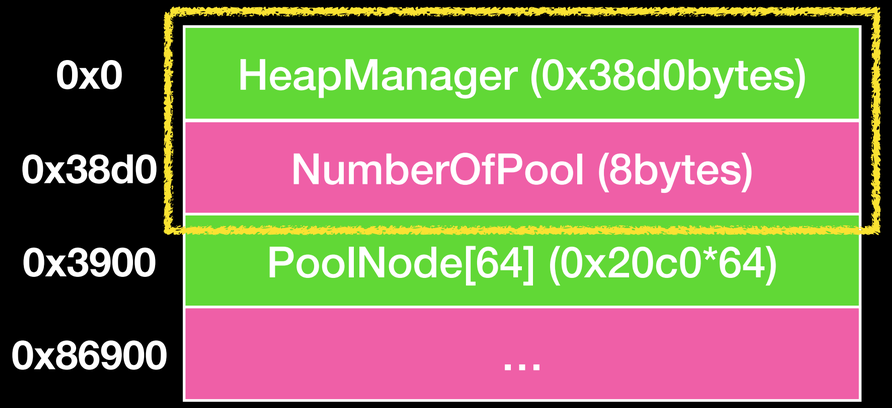
Memory Allocator in kernel

- Important structure
  - _EX_POOL_HEAP_MANAGER_STATE
  - _SEGMENT_HEAP
  - _RTLP_HP_HEAP_GLOBALS

Memory Allocator in kernel

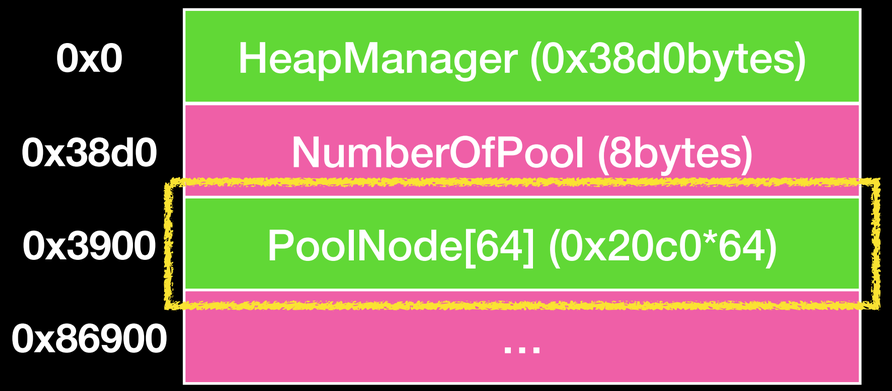
- nt!ExPoolState (_EX_POOL_HEAP_MANAGER_STATE)
  - The core structure in the Kernel pool memory allocator
  - Whether it is Page Pool or Non page pool, etc. are managed by this structure

Memory Allocator in kernel



- ExPoolState (_EX_POOL_HEAP_MANAGER_STATE)
  - HeapManager (_RTLP_HP_HEAP_MANAGER)
    - Store some Global variables, some Metadata, etc.
  - NumberOfPool
    - Number of Pool Node
      - Default 1

Memory Allocator in kernel



- ExPoolState (_EX_POOL_HEAP_MANAGER_STATE)
  - PoolNode (_EX_HEAP_POOL_NODE)
    - Each node will have four Heaps corresponding to different segment heaps, such as Paged/ Nonpaged pool, etc.

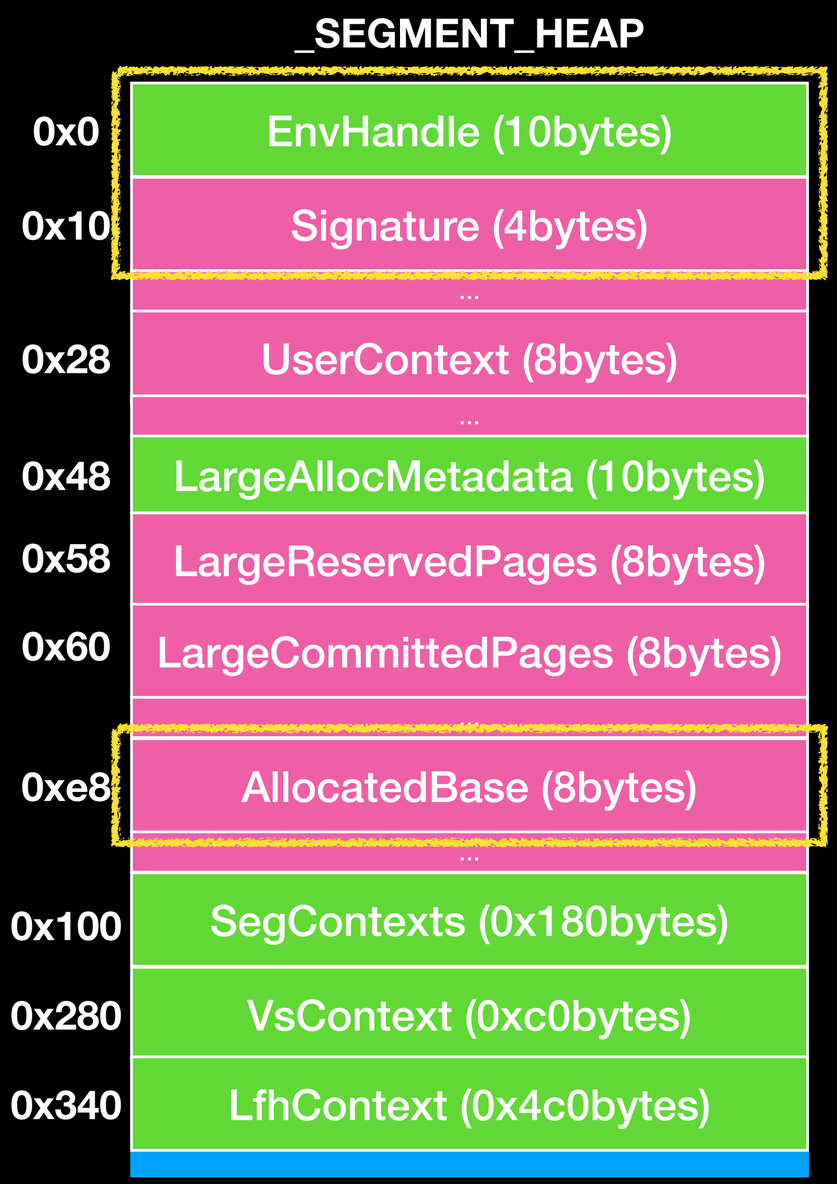
Memory Allocator in kernel

- _SEGMENT_HEAP
  - The core structure in the segment heap
  - Each Pool will have a _SEGMENT_HEAP structure. When allocating memory, it will decide which HEAP to allocate according to the pool type.
  - In ExPoolState
    - These are four segment heap will be created and initialized when the system is turned on
    - ExPoolState.PoolNode[0].Heap[0-4]

Memory Allocator in kernel

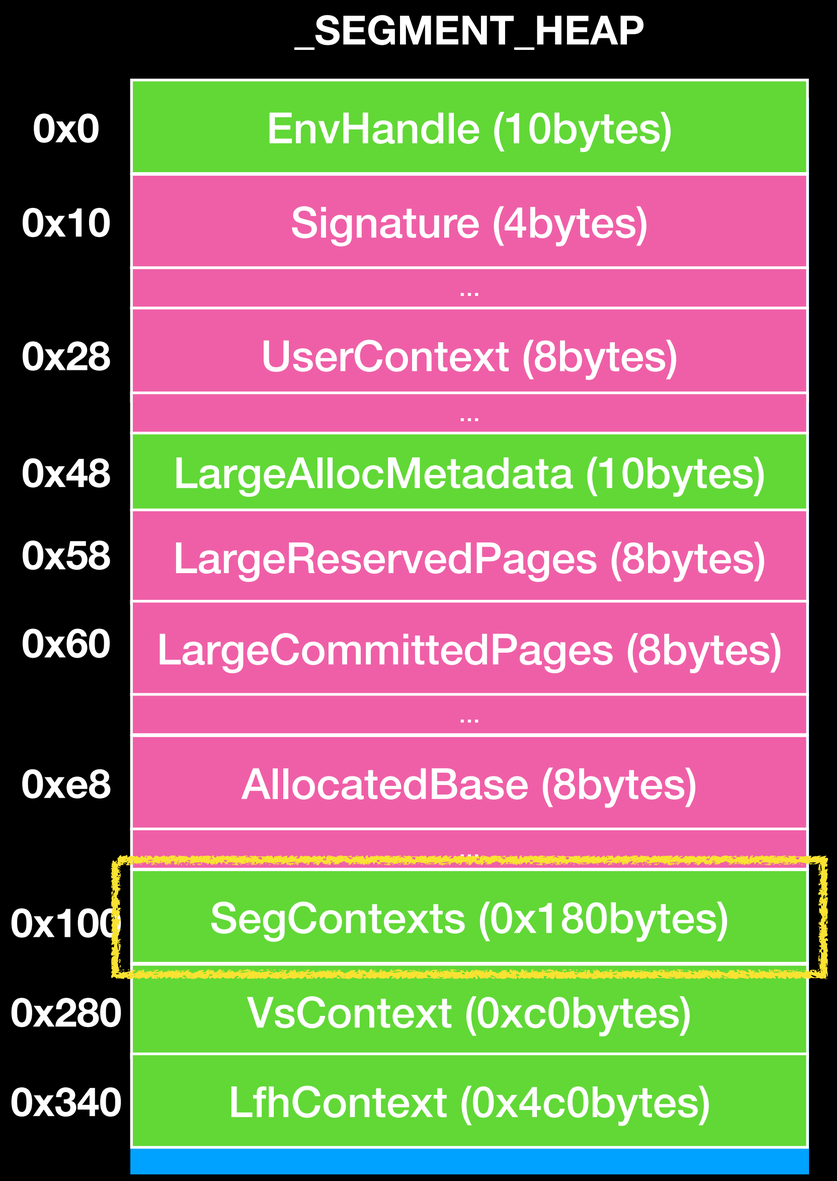
- _SEGMENT_HEAP
  - ExPoolState.PoolNode[0].Heap[0] -> NonPagedPool
  - ExPoolState.PoolNode[0].Heap[1] -> NonPagedPoolNx
  - ExPoolState.PoolNode[0].Heap[2] -> PagedPool

Memory Allocator in kernel



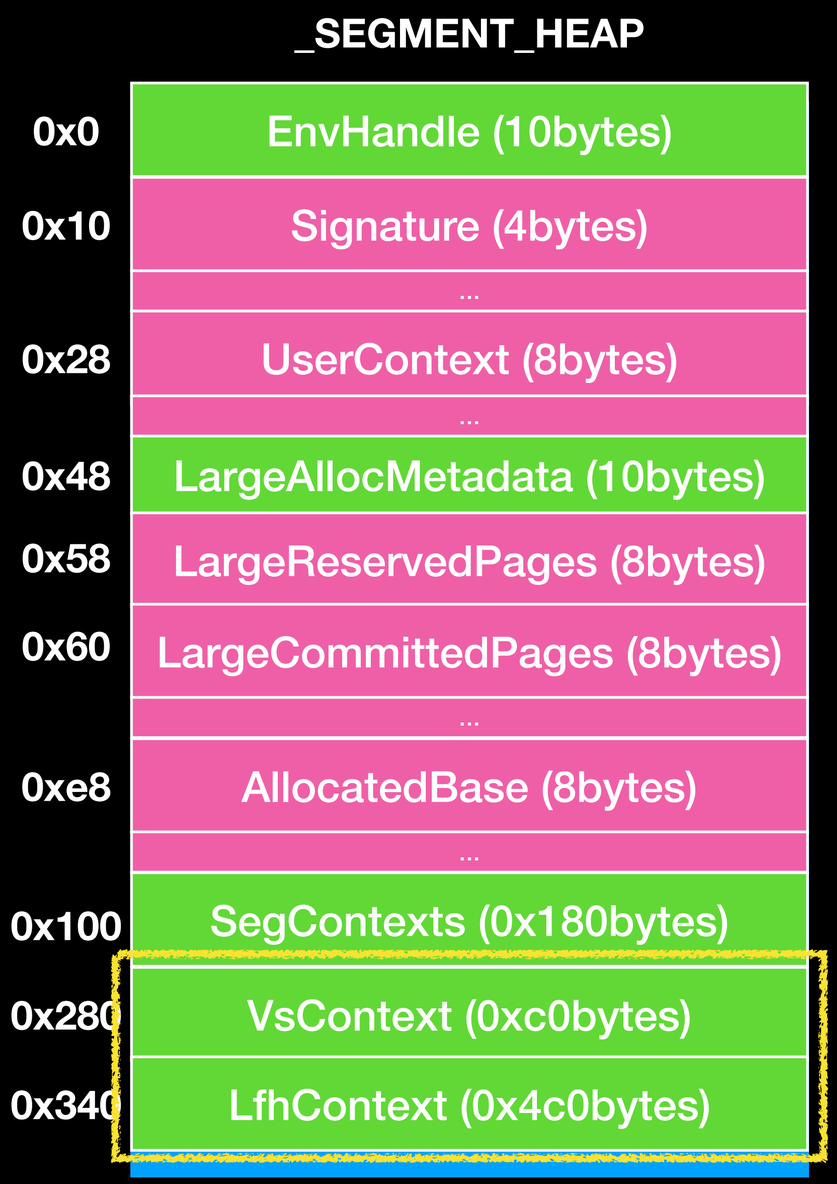
- _SEGMENT_HEAP
  - EnvHandle (RTL_HP_ENV_HANDLE)
    - environment handle of segment heap
  - Signature
    - Signature of segment heap
      - It always 0xddeeddee
  - AllocatedBase
    - Point to the end of the entire structure of _SEGMENT_HEAP, used to allocate the structure required by LFH Allocator, after allocation, it will point to the end of the allocated structure

Memory Allocator in kernel



- _SEGMENT_HEAP
  - SegContexts (_HEAP_SEG_CONTEXT)
    - The core structure of the back-end manager
    - Divided into two segcontexts according to size
      - 0x20000 < Size <= 0x7f000
      - 0x7f000 < Size <= 0x7f0000

Memory Allocator in kernel

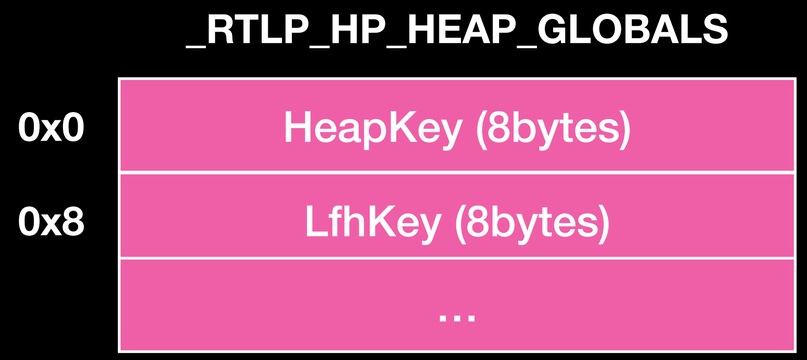


- _SEGMENT_HEAP
  - VsContext (_HEAP_VS_CONTEXT)
    - The core structure of front-end VS Allocator
  - LfhContext (_HEAP_LFH_CONTEXT)
    - The core structure of the front-end LowFragmentationHeap Allocator

Memory Allocator in kernel

- nt!RtlpHpHeapGlobals (_RTLP_HP_HEAP_GLOBALS)
  - In Segment Heap, many fields, values, and functions are mostly encoded
  - This structure will be used to store related key and other information

Memory Allocator in kernel



- RtlpHpHeapGlobals (_RTLP_HP_HEAP_GLOBALS)
  - HeapKey
    - Used for VS Allocator, Segment Allocator
  - LfhKey
    - Used for LowFragmentationHeap
  - Both are 8 byte random value

Memory Allocator in kernel

- Segment Heap
  - Frontend Allocation
    - Low FragmentationHeap
    - Variable Size Allocation
  - Backend Allocation
    - Segment Allocation
  - Large Block Allocation

Low FragmentationHeap

- Size
  - In the case of default
    - Size <= 0x200 and LFH is enabled for this size
  - The activation timing is very similar to that of NtHeap. It will be activated after 18 consecutive allocations of the same size block

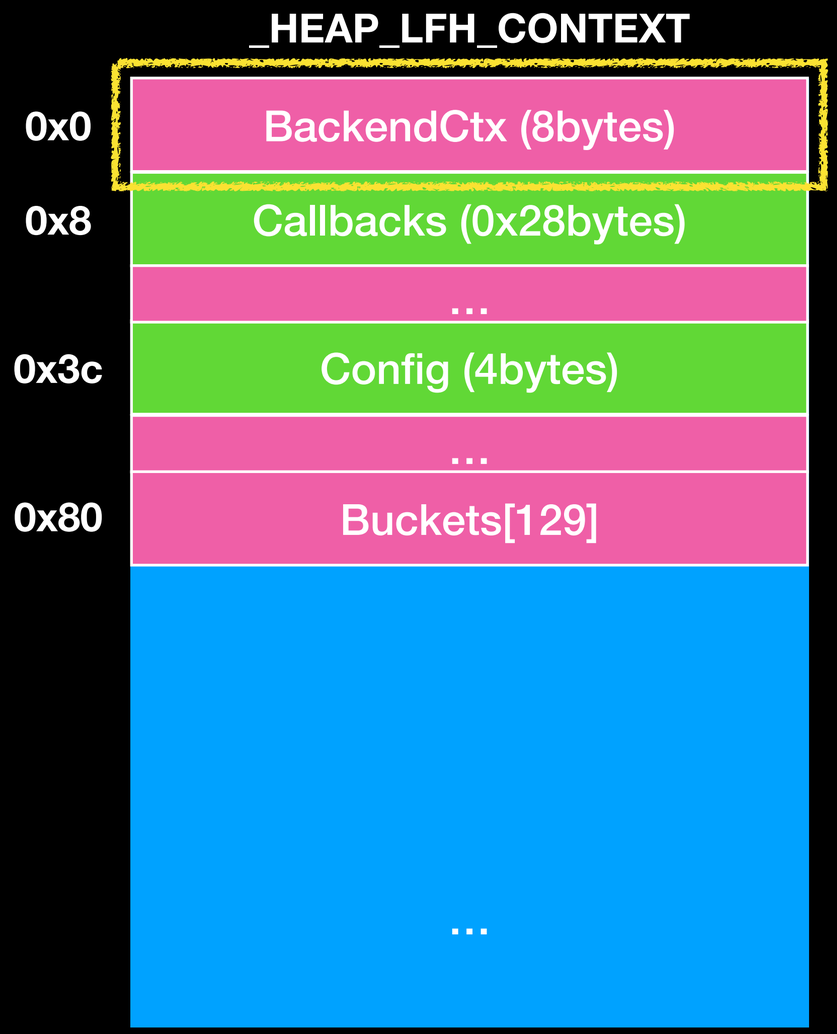
Low FragmentationHeap

- Data Structure
- Memory allocation mechanism

Low FragmentationHeap

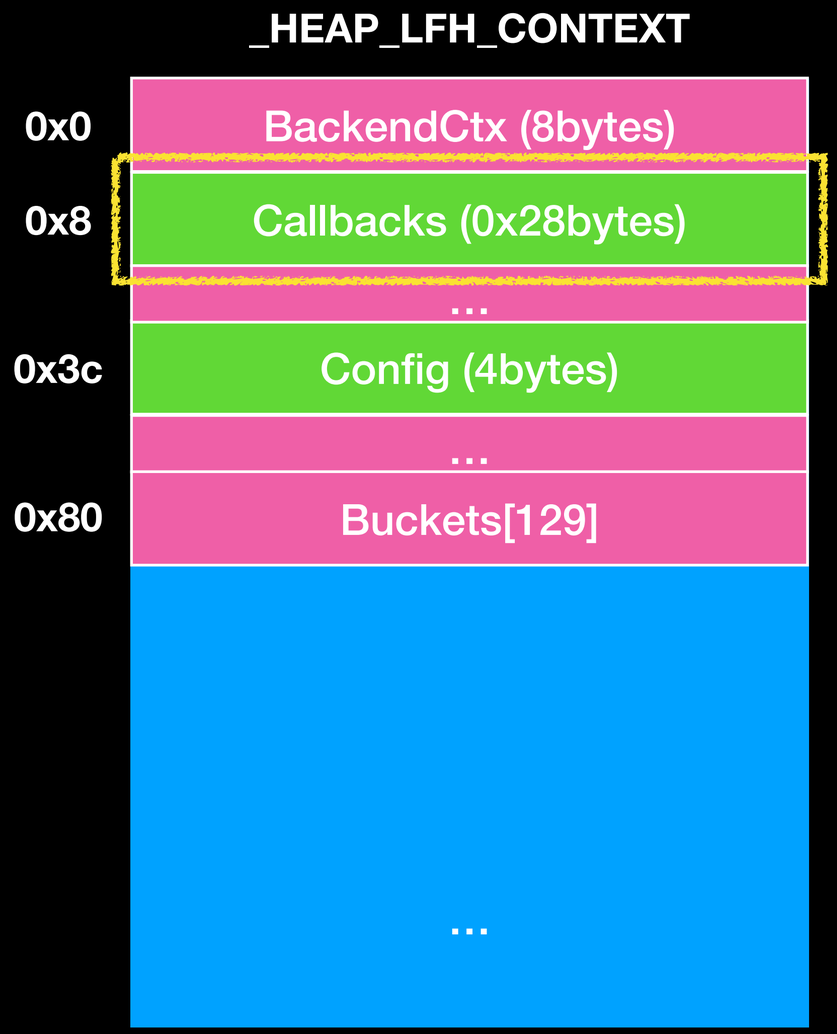
- LfhContext (_HEAP_LFH_CONTEXT)
  - The core structure of the LFH Allocator
  - Used to manage the memory allocated by LFH, record all the information and structure of LFH in the heap

Low FragmentationHeap



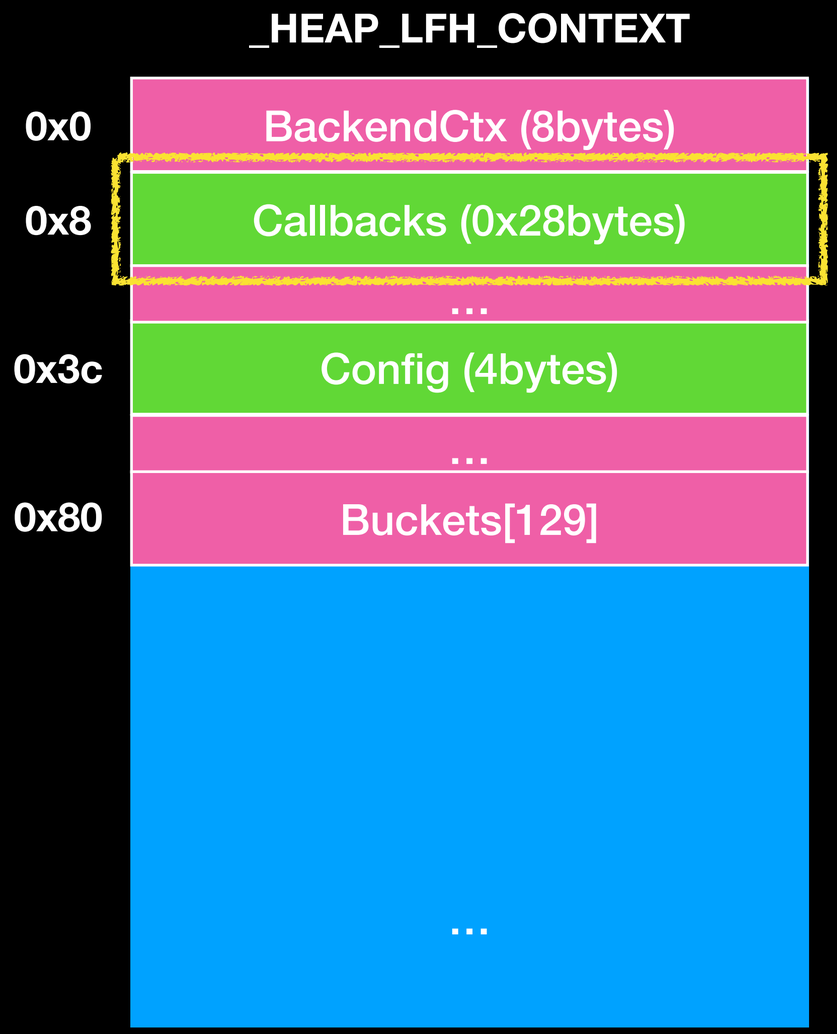
- LfhContext (_HEAP_LFH_CONTEXT)
  - BackendCtx (_HEAP_SEG_CONTEXT)
    - Point to the Backend allocator used by the LFH

Low FragmentationHeap



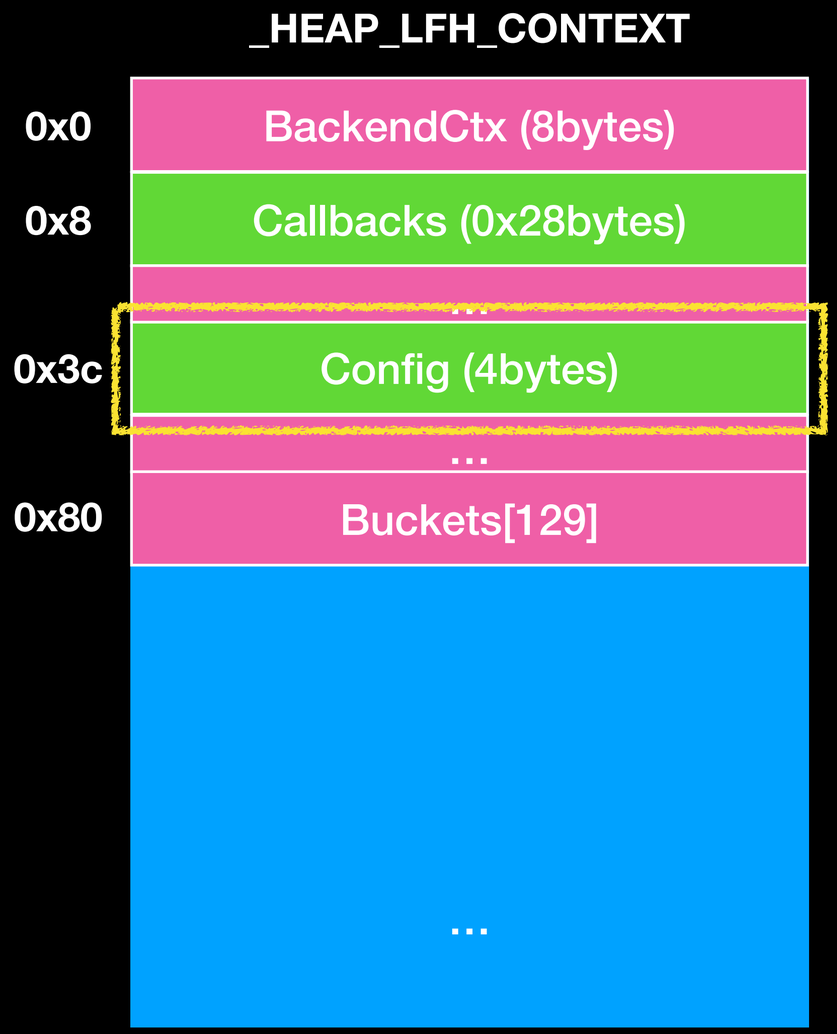
- LfhContext (_HEAP_LFH_CONTEXT)
  - Callbacks (_HEAP_SUBALLOCATOR_CALLBACKS)
    - Call back function table (use to allocate/ release subsegments)
      - Allocate
      - Free
      - Commit
      - Decommit
      - ExtendContext

Low FragmentationHeap



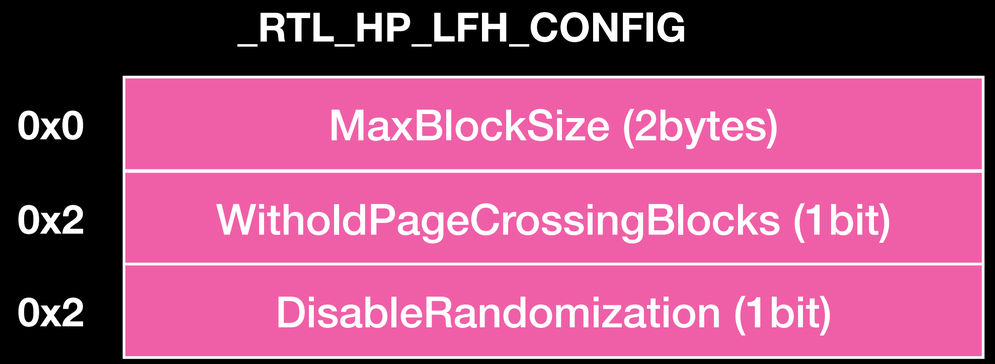
- LfhContext (_HEAP_LFH_CONTEXT)
  - Callbacks (_HEAP_SUBALLOCATOR_CALLBACKS)
    - Function pointer 會被 encode 過
      - The value of callbacks will be xored by
        - RtlpHpHeapGlobals.HeapKey
        - Address of LfhContext
        - Function pointer

Low FragmentationHeap



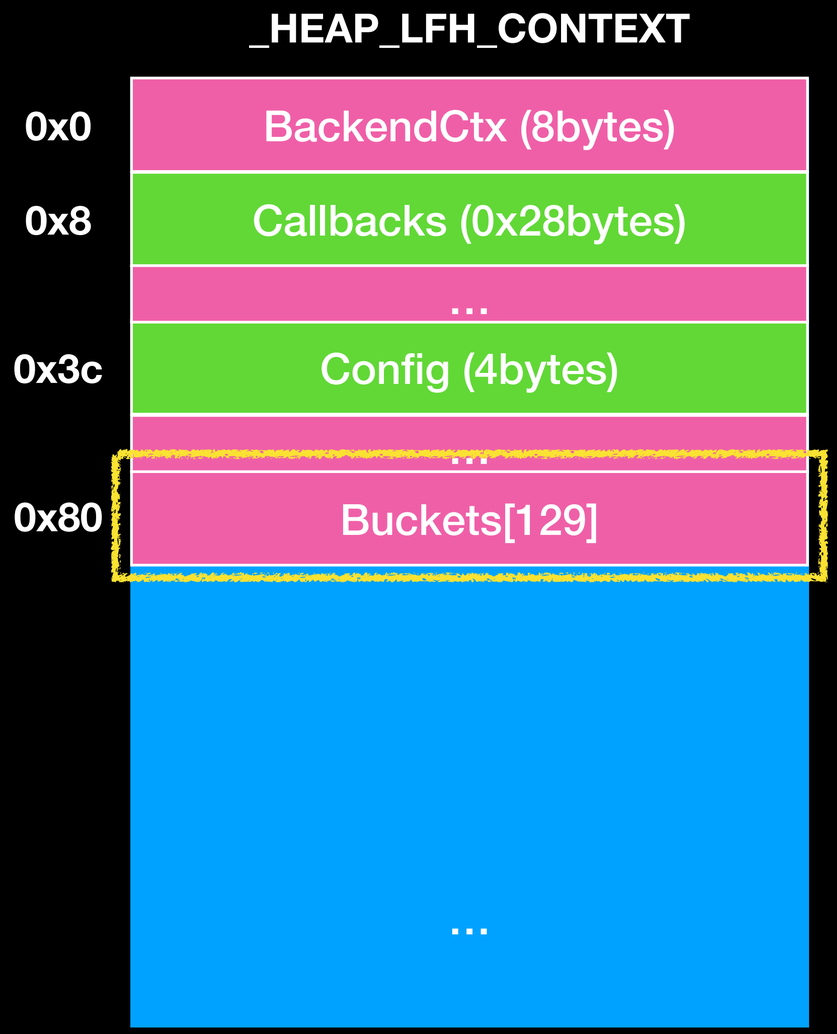
- LfhContext (_HEAP_LFH_CONTEXT)
  - Config (_RTL_HP_LFH_CONFIG)
    - Used to indicate the attributes of the LFH Allocator
    - It will be used to determine whether the size of allocation is within the scope of LFH allocator

Low FragmentationHeap



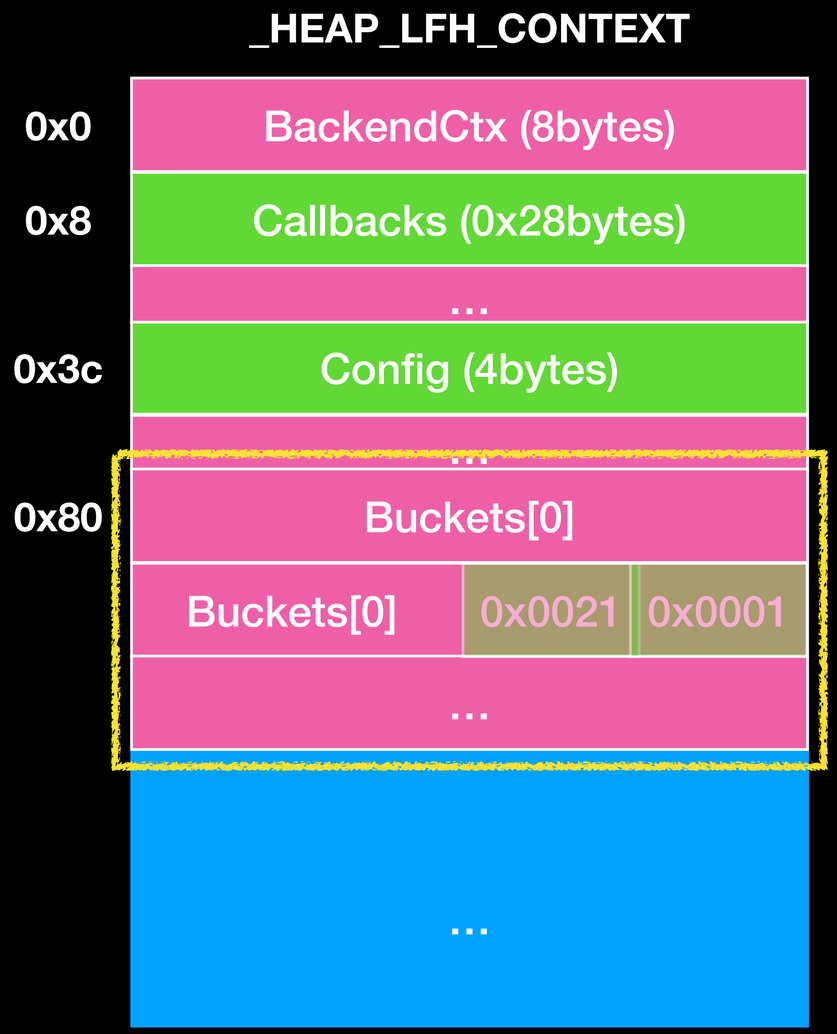
- LfhContext (_HEAP_LFH_CONTEXT)
  - Config (_RTL_HP_LFH_CONFIG)
    - MaxBlockSize
      - The size of the max block in LFH
    - WitholdPageCrossingBlocks
      - Are there any cross-page blocks
    - DisableRandomization
      - Whether to disable randomization of LFH

Low FragmentationHeap



- LfhContext (_HEAP_LFH_CONTEXT)
  - Buckets (_HEAP_LFH_BUCKET)
    - Bucket array, each buckets corresponds to blocks in a specific size range
    - When LFH is enabled, it will point to the _HEAP_LFH_BUCKET structure

Low FragmentationHeap

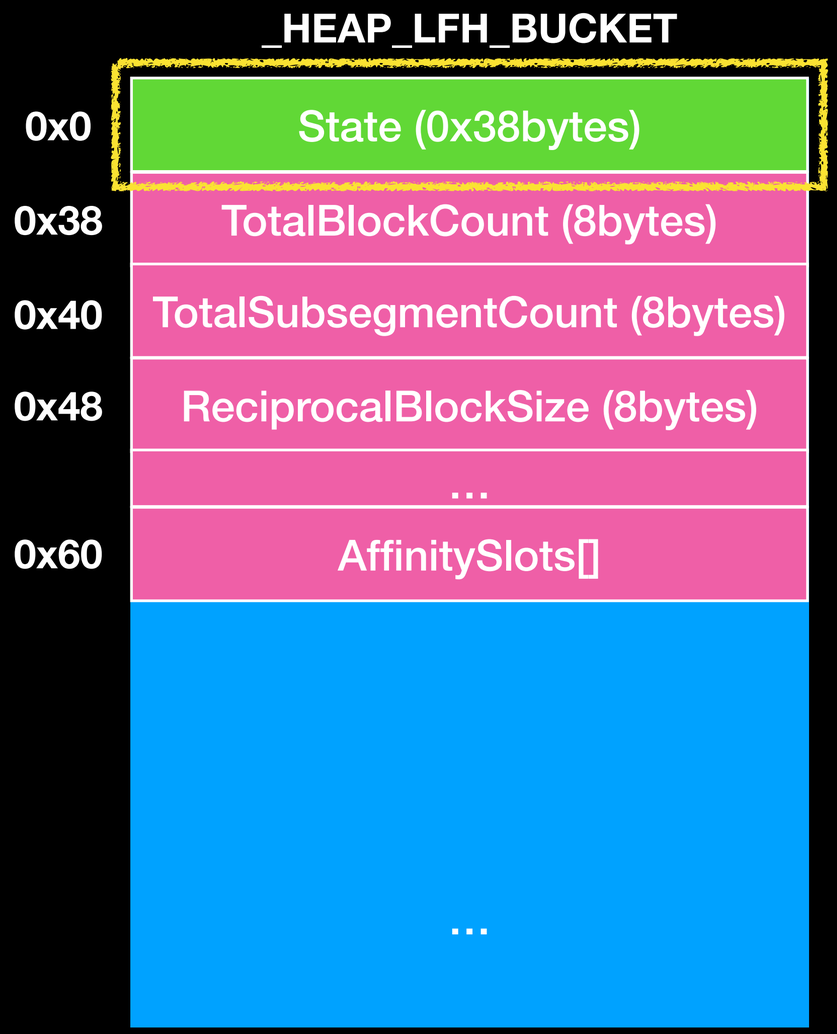


- LfhContext (_HEAP_LFH_CONTEXT)
  - Buckets (_HEAP_LFH_BUCKET)
    - If it is not enabled, it will be used to indicate the number of times the block of the size is allocated.
      - The last 2 bytes are always 1
      - The next 2 byte is the number of allocations
        - Each allocation will add 0x21

Low FragmentationHeap

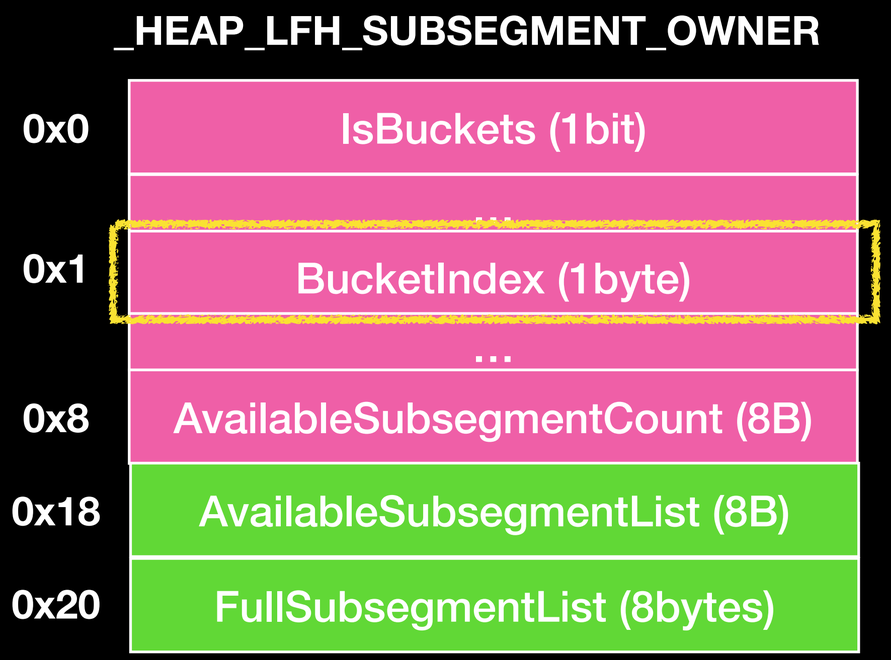
- Buckets (_HEAP_LFH_BUCKET)
  - The bucket and its related structure will be allocated only when the LFH is enabled
  - LFH Allocator is use the structure to manage the block corresponding to the Size

Low FragmentationHeap



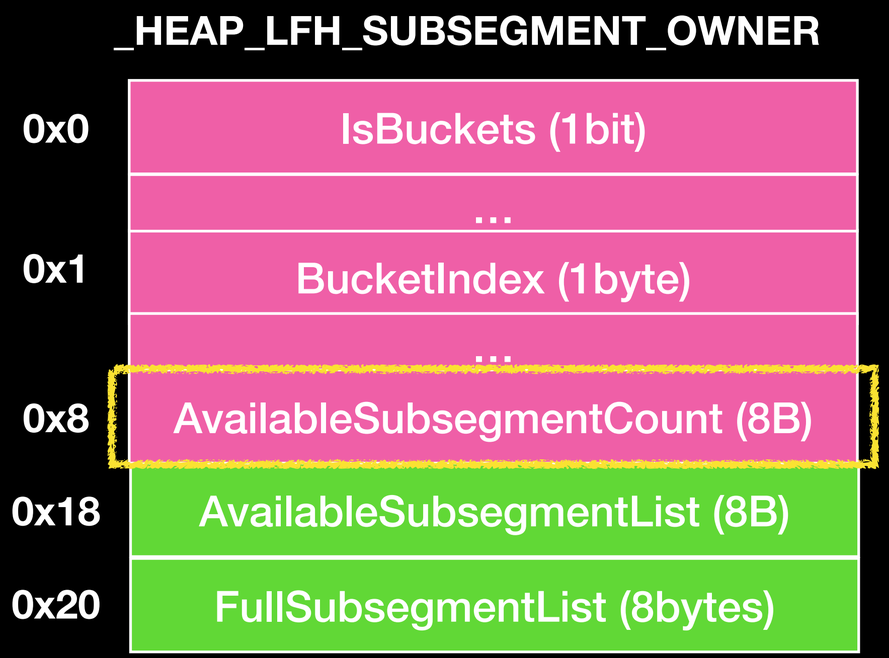
- Buckets (_HEAP_LFH_BUCKET)
  - State (_HEAP_LFH_SUBSEGMENT_OWNER)
    - Used to indicate the status of the Buckets
    - This structure is used to manage the memory pool of LFH

Low FragmentationHeap



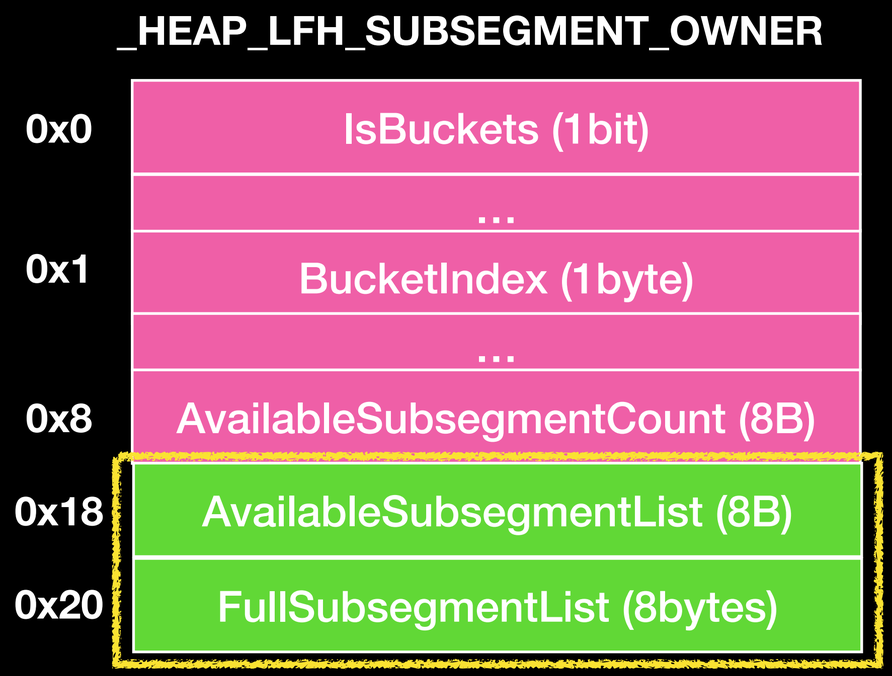
- Buckets (_HEAP_LFH_BUCKET)
  - State (_HEAP_LFH_SUBSEGMENT_OWNER)
    - BucketIndex
      - The index of the bucket

Low FragmentationHeap



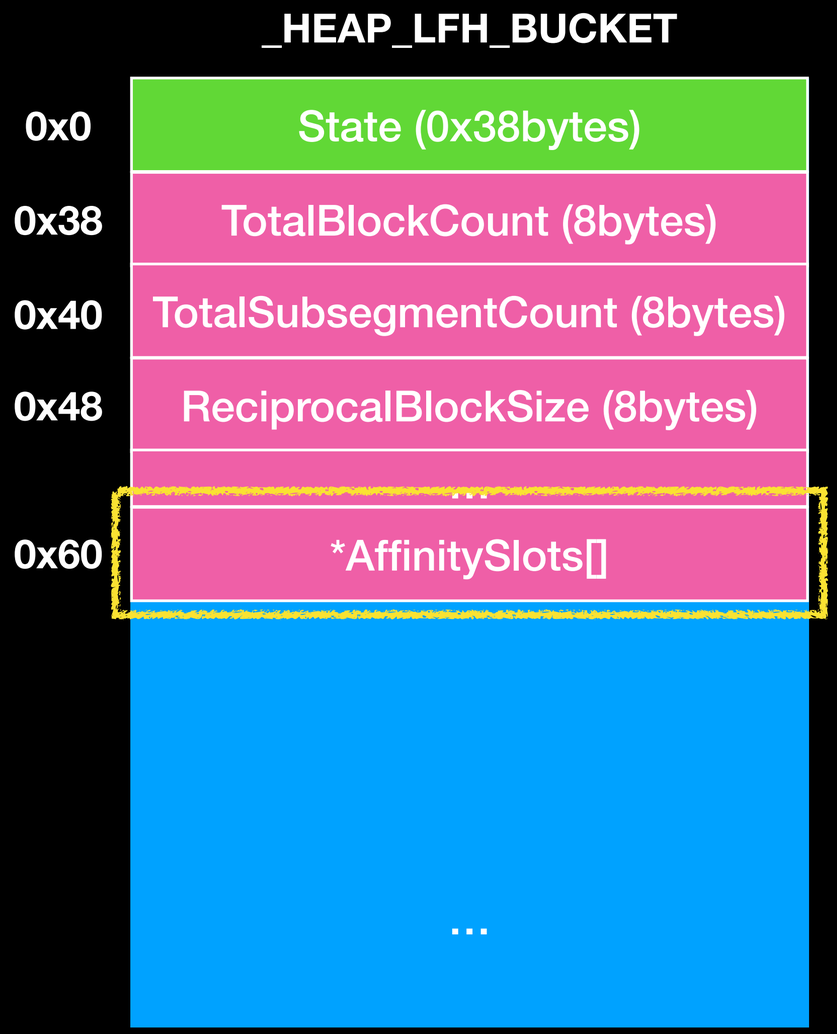
- Buckets (_HEAP_LFH_BUCKET)
  - State (_HEAP_LFH_SUBSEGMENT_OWNER)
    - AvailableSubsegmentCount
      - Number of available subsegments

Low FragmentationHeap



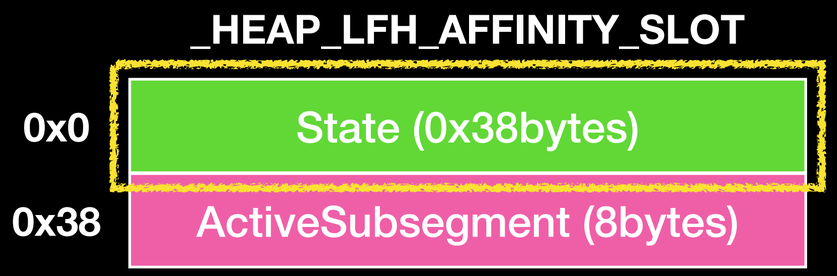
- Buckets (_HEAP_LFH_BUCKET)
  - State (_HEAP_LFH_SUBSEGMENT_OWNER)
    - AvailableSubsegmentList (_LIST_ENTRY)
      - Point to the next available subsegment in the bucket
    - FullSubsegmentList (_LIST_ENTRY)
      - Point to the next used up subsegment in the bucket

Low FragmentationHeap



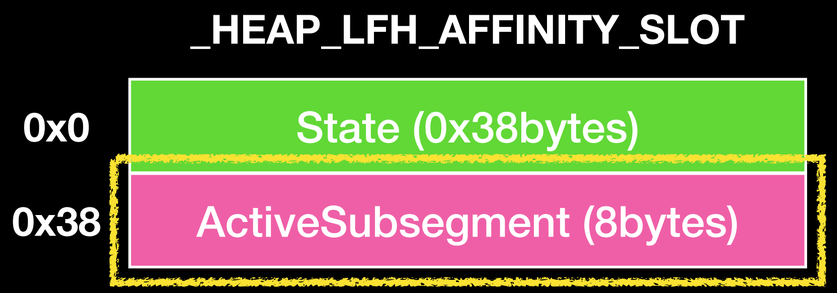
- Buckets (_HEAP_LFH_BUCKET)
  - AffinitySlots (_HEAP_LFH_AFFINITY_SLOT)
    - The main structure used to manage the memory pool using by LFH
    - There is only one by default

Low FragmentationHeap



- Buckets (_HEAP_LFH_BUCKET)
  - AffinitySlots (_HEAP_LFH_AFFINITY_SLOT)
    - State (_HEAP_LFH_SUBSEGMENT_OWNER)
      - Same structure as State in Bucket, but this one is mainly used to manage subsegment

Low FragmentationHeap

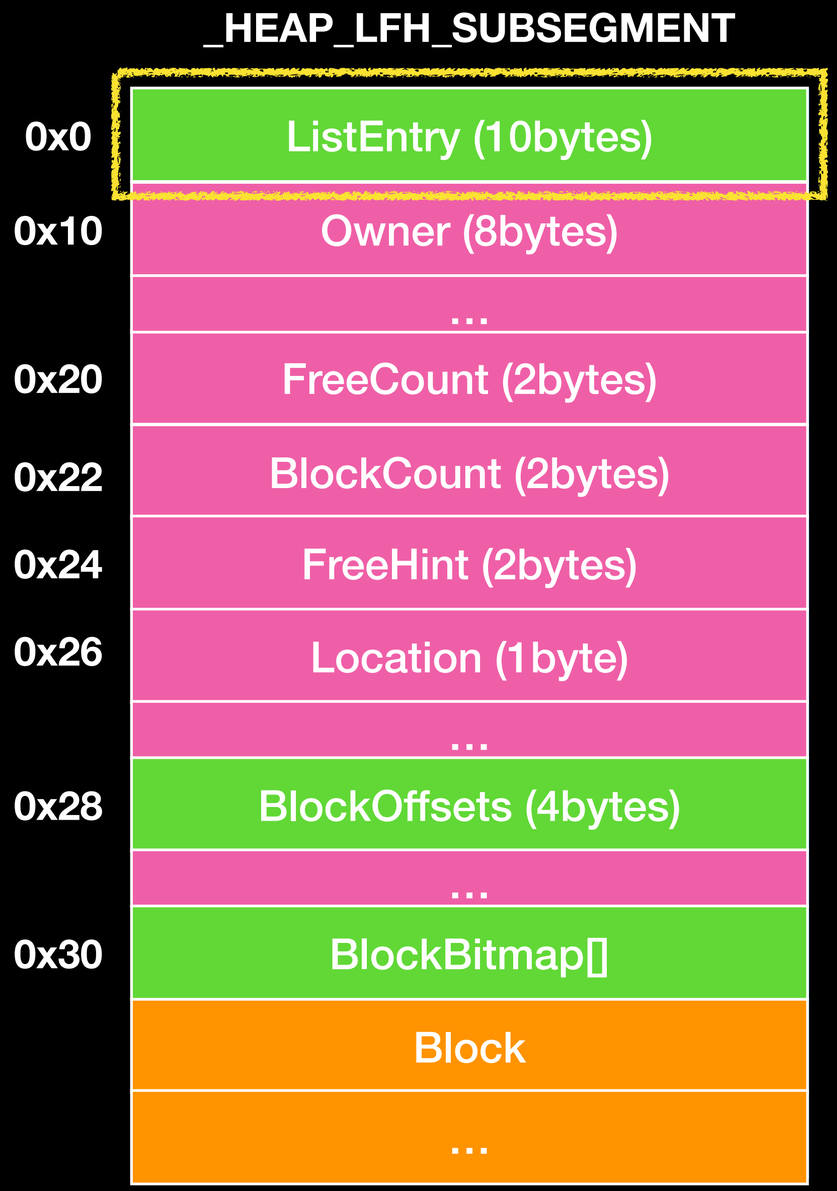


- Buckets (_HEAP_LFH_BUCKET)
  - AffinitySlots (_HEAP_LFH_AFFINITY_SLOT)
    - ActiveSubsegment (_HEAP_LFH_FAST_REF)
      - Point to the subsegment being used
      - The lowest 12 bits indicate how many blocks are available in the subsegment

Low FragmentationHeap

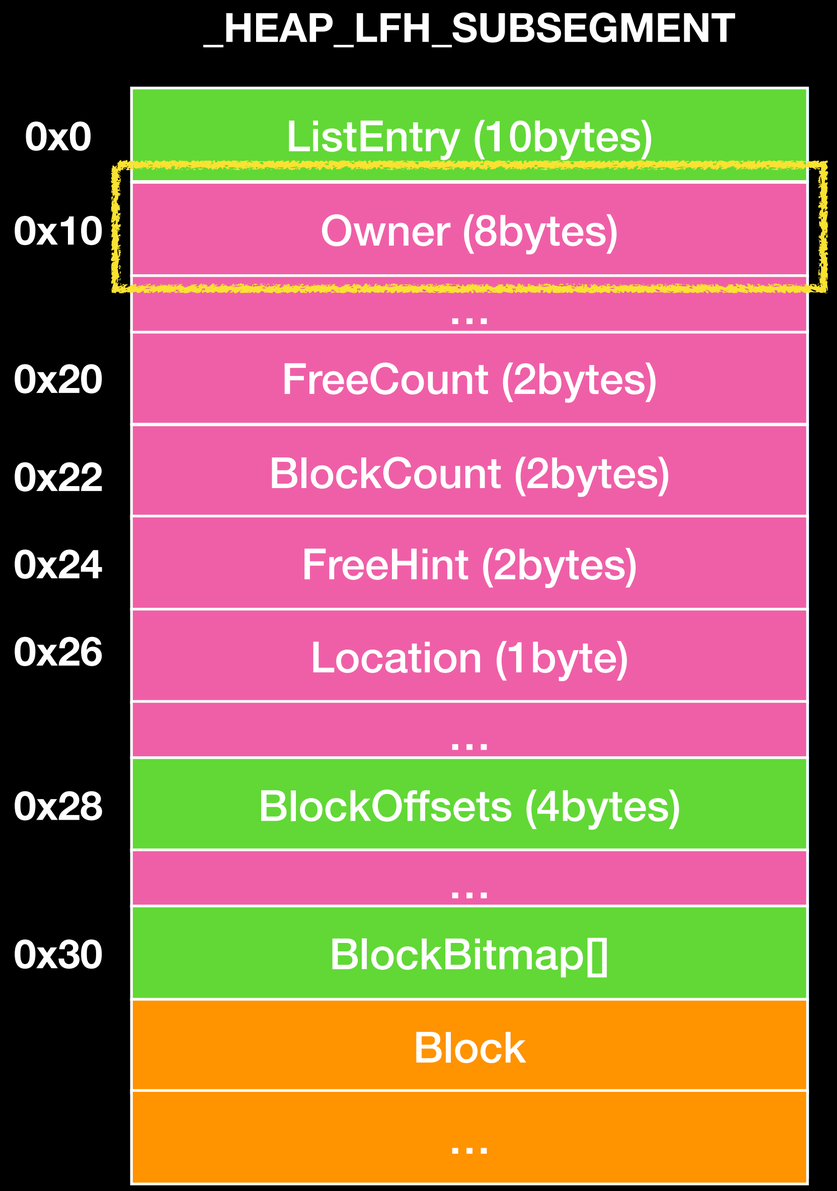
- LFH Subsegments (_HEAP_LFH_SUBSEGMENT)
  - The memory pool of LFH is very similar in structure to UserBlock in NtHeap, but each block does not have header and other metadata.
  - Once there is not enough memory, it will take subsegment from Buckets->State.availableSubsegmentList first, if it has no available subsegment , it will allocate from backend allocator for a new subsegment
  - Mainly managed by buckets->AffinitySlots

Low FragmentationHeap



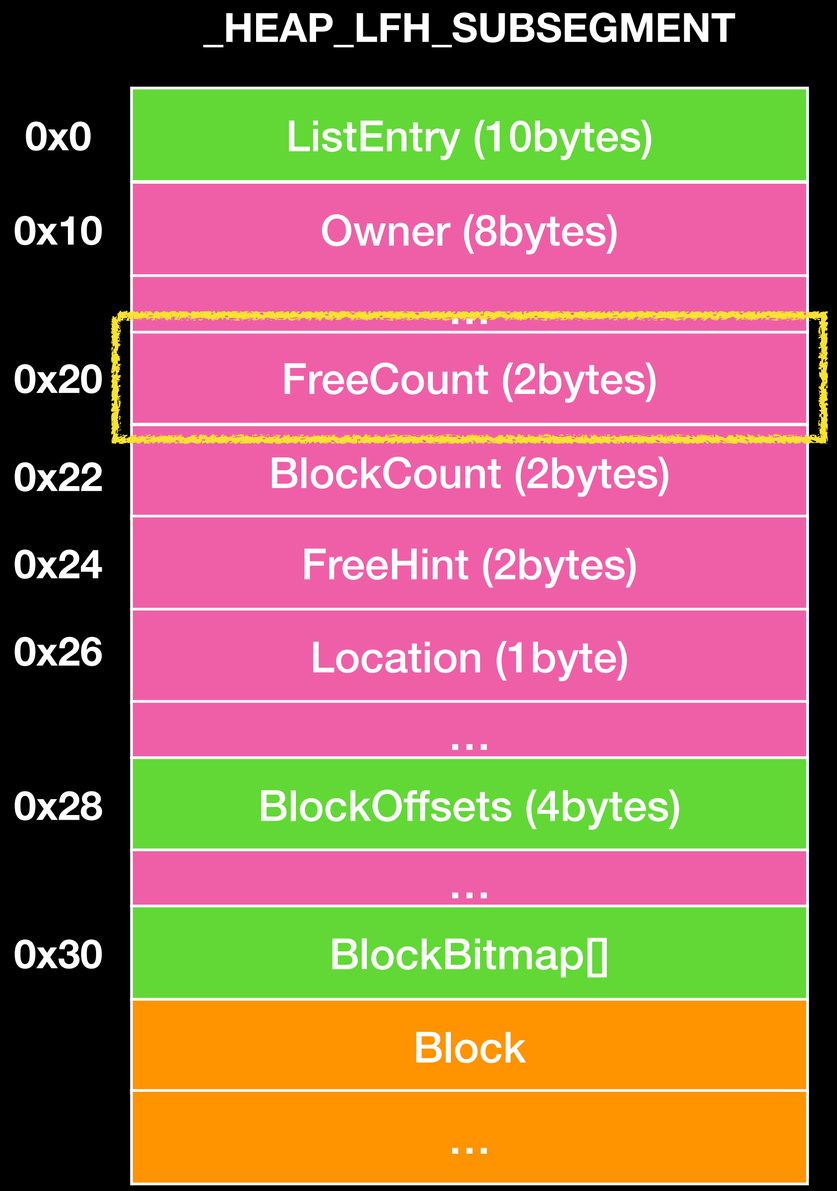
- LFH Subsegments (_HEAP_LFH_SUBSEGMENT)
  - ListEntry(_LIST_ENTRY)
    - Flink
      - Point to the next full/available LFH subsegment
    - Blink
      - Point to the previous full/available LFH subsegment

Low FragmentationHeap



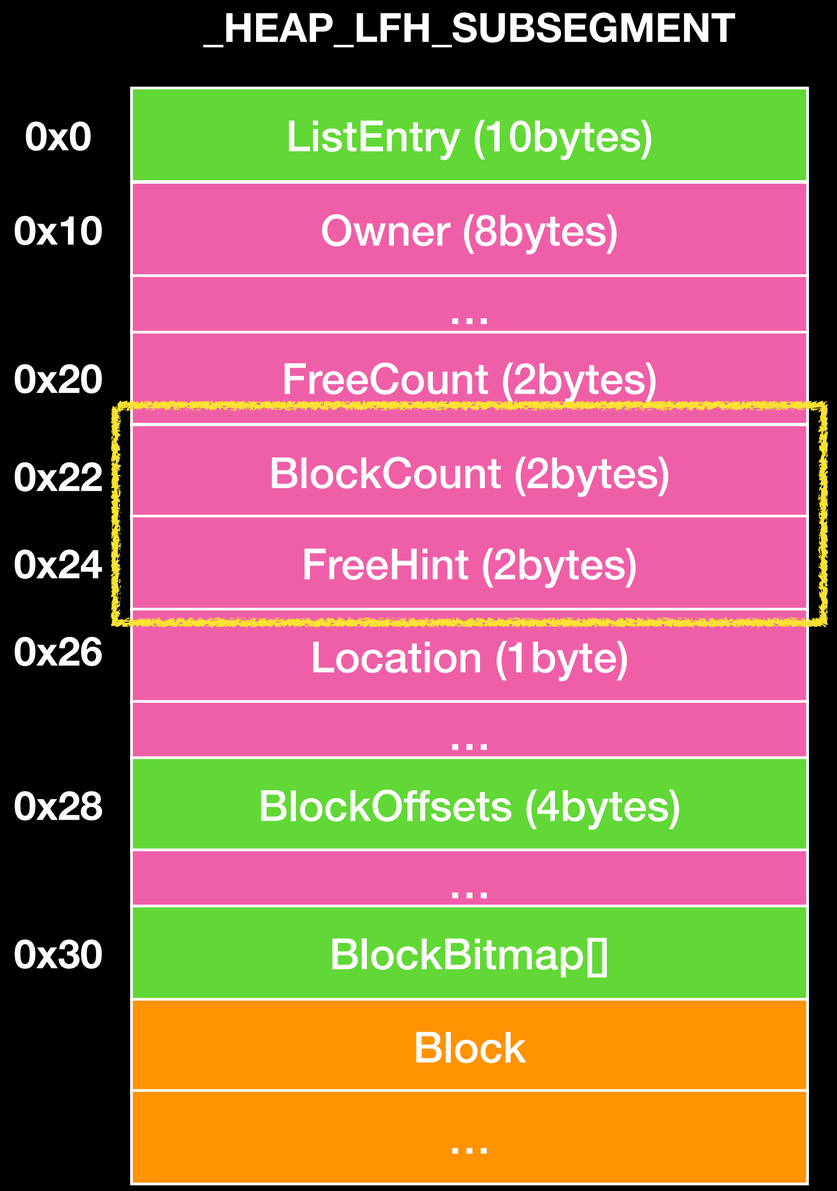
- LFH Subsegments (_HEAP_LFH_SUBSEGMENT)
  - Owner (_HEAP_LFH_SUBSEGMENT_OWNER)
    - Point to the structure that manages the subsegment
      - Point back to the AffinitySlots.State of the bucket to which it belongs

Low FragmentationHeap



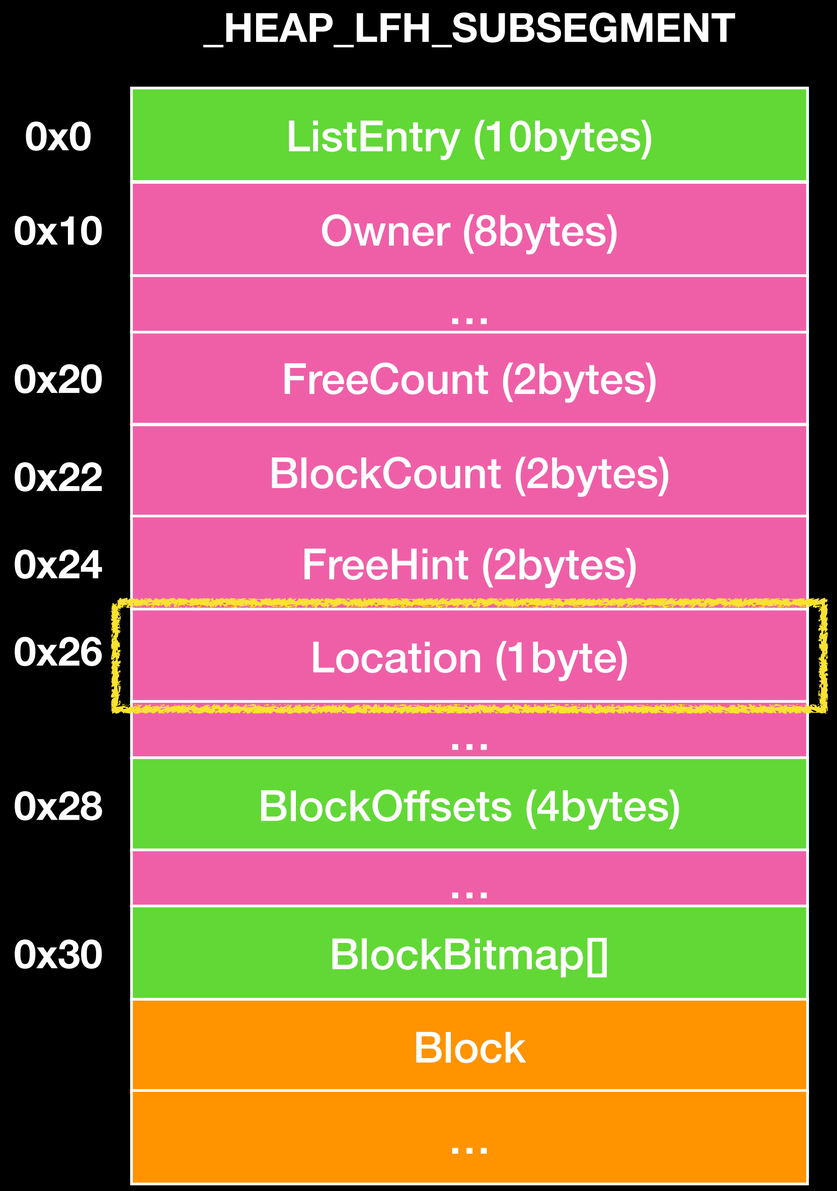
- LFH Subsegments (_HEAP_LFH_SUBSEGMENT)
  - FreeCount
    - The number of times the block is released
    - Not the number of freed blocks of the subsegment

Low FragmentationHeap



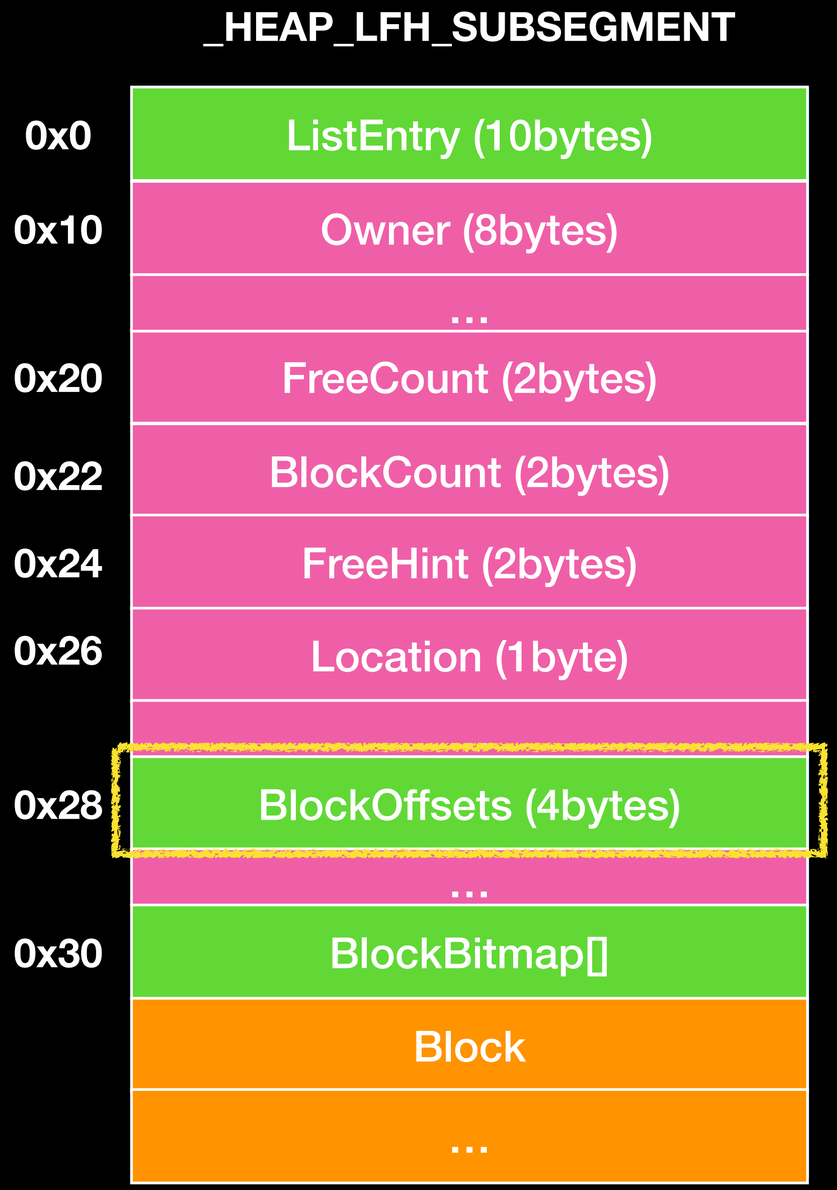
- LFH Subsegments (_HEAP_LFH_SUBSEGMENT)
  - BlockCount
    - the total number of block in the subsegment
  - FreeHint
    - The index of block that the last allocated
    - If the index of block that the last freed larger than freehint. It will be updated to the index
    - Used in allocated algorithm

Low FragmentationHeap



- LFH Subsegments (_HEAP_LFH_SUBSEGMENT)
  - Location
    - Indicates the location of the subsegment
      - 0 : AvailableSubsegmentList
      - 1 : FullSubsegmentList
      - 2 : The entire subsegment will be returned to back-end soon

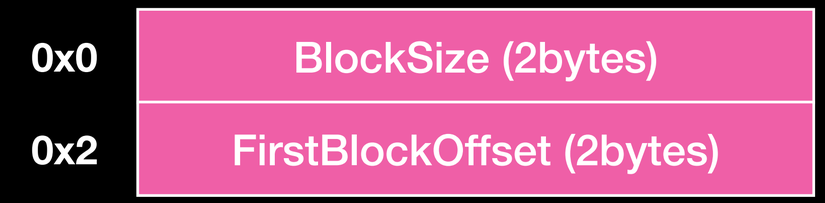
Low FragmentationHeap



- LFH Subsegments (_HEAP_LFH_SUBSEGMENT)
  - BlockOffsets (_HEAP_LFH_SUBSEGMENT_ENCODED_OFFSETS)
    - Used to indicate the block size of the subsegment and the offset of the first block in the subsegment

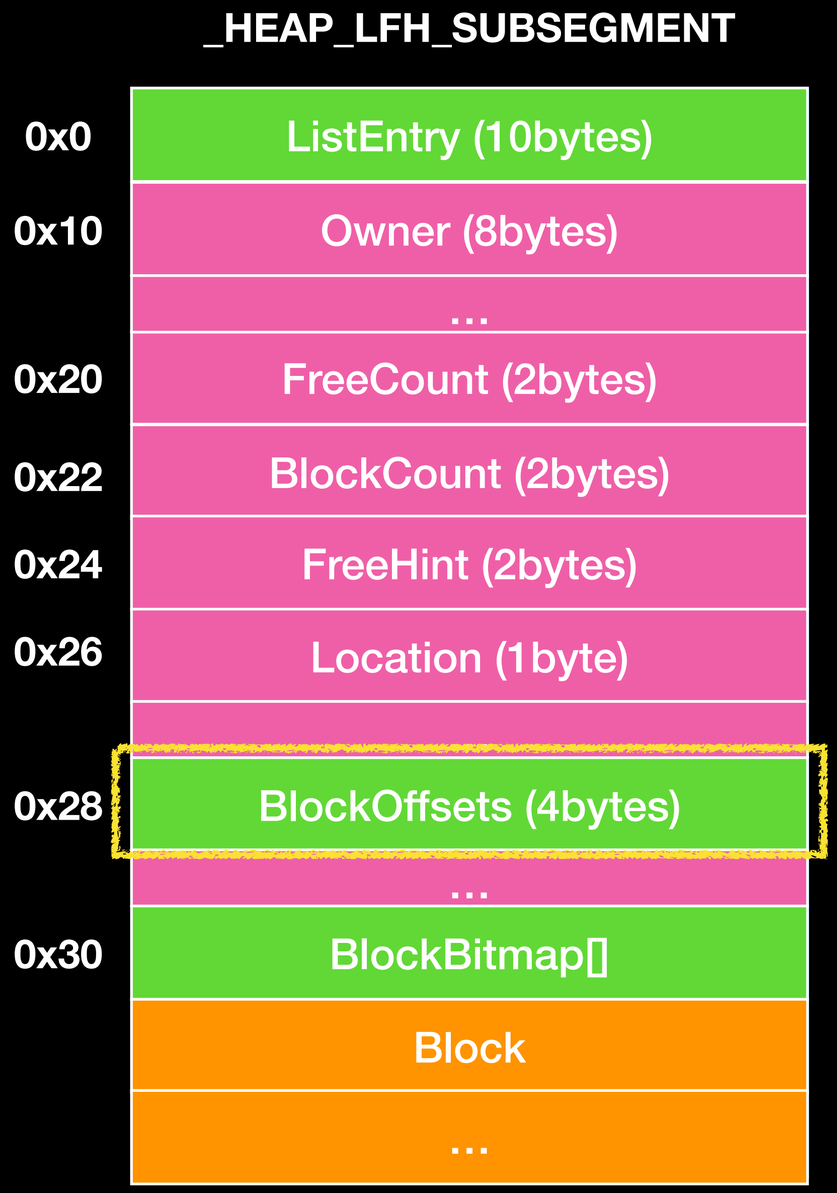
Low FragmentationHeap

_HEAP_LFH_SUBSEGMENT_ENCODED_OFFSETS



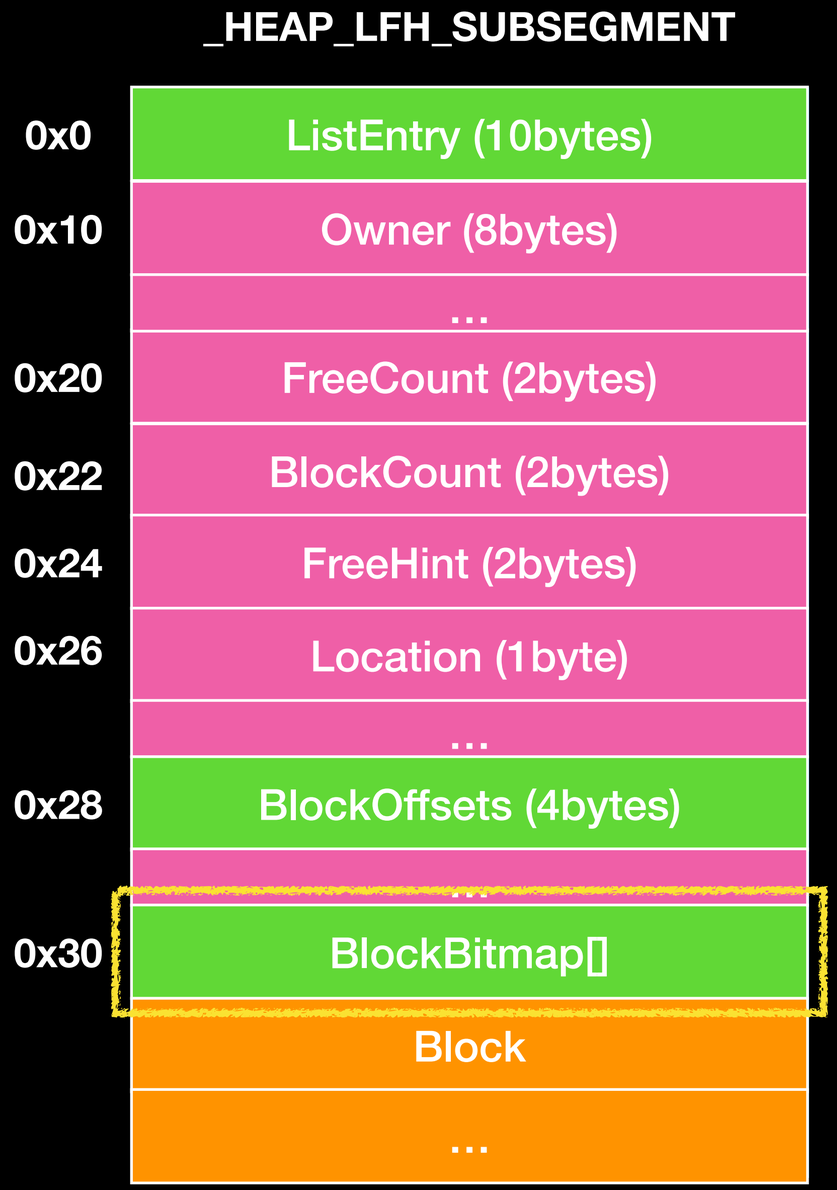
- LFH Subsegments (_HEAP_LFH_SUBSEGMENT)
  - BlockSize
    - The size of block in the subsegment
    - The size is the original size
  - FirstBlockOffset
    - The offset of the first block
    - FirstBlock = subsegment + FirstBlockOffset

Low FragmentationHeap



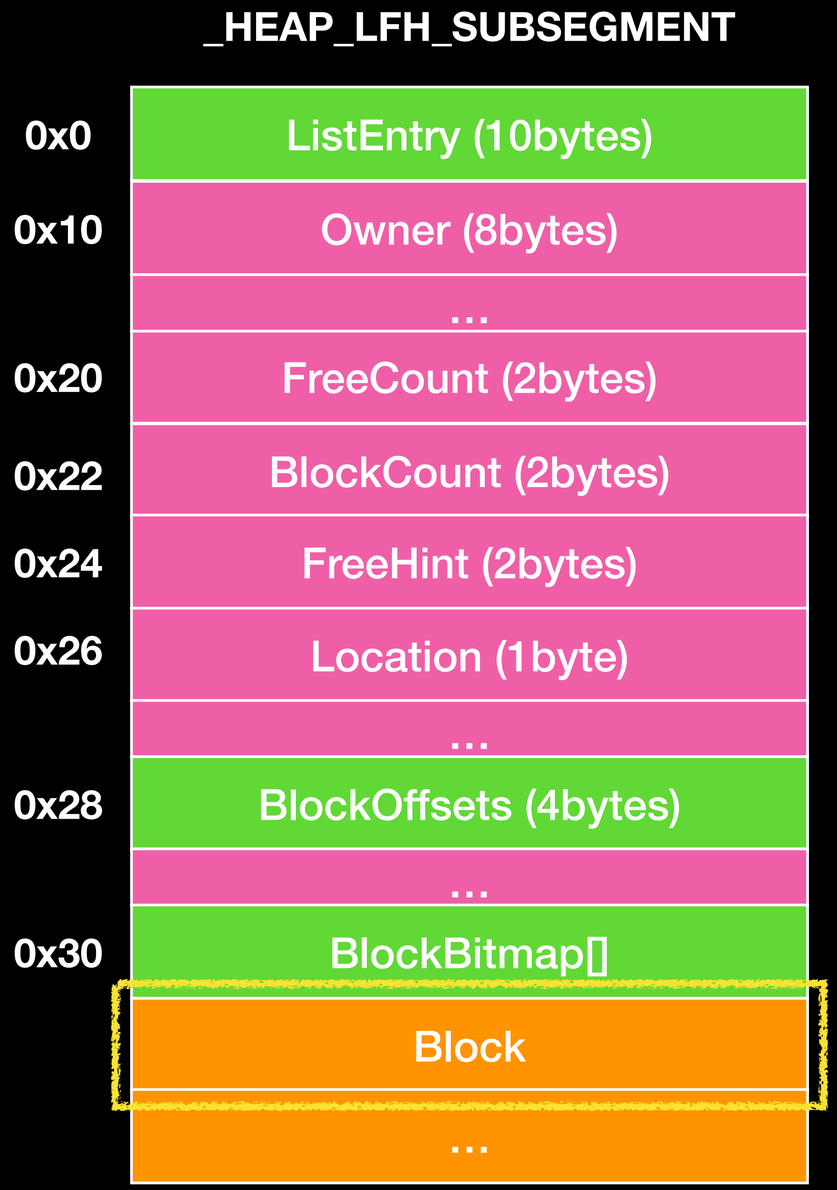
- LFH Subsegments (_HEAP_LFH_SUBSEGMENT)
  - BlockOffsets (_HEAP_LFH_SUBSEGMENT_ENCODED_OFFSETS)
  - The value will be encoded
    - EncodedData = RtlpHpHeapGlobals.LfhKey^BlockOffsets^(subsegment >> 12)

Low FragmentationHeap



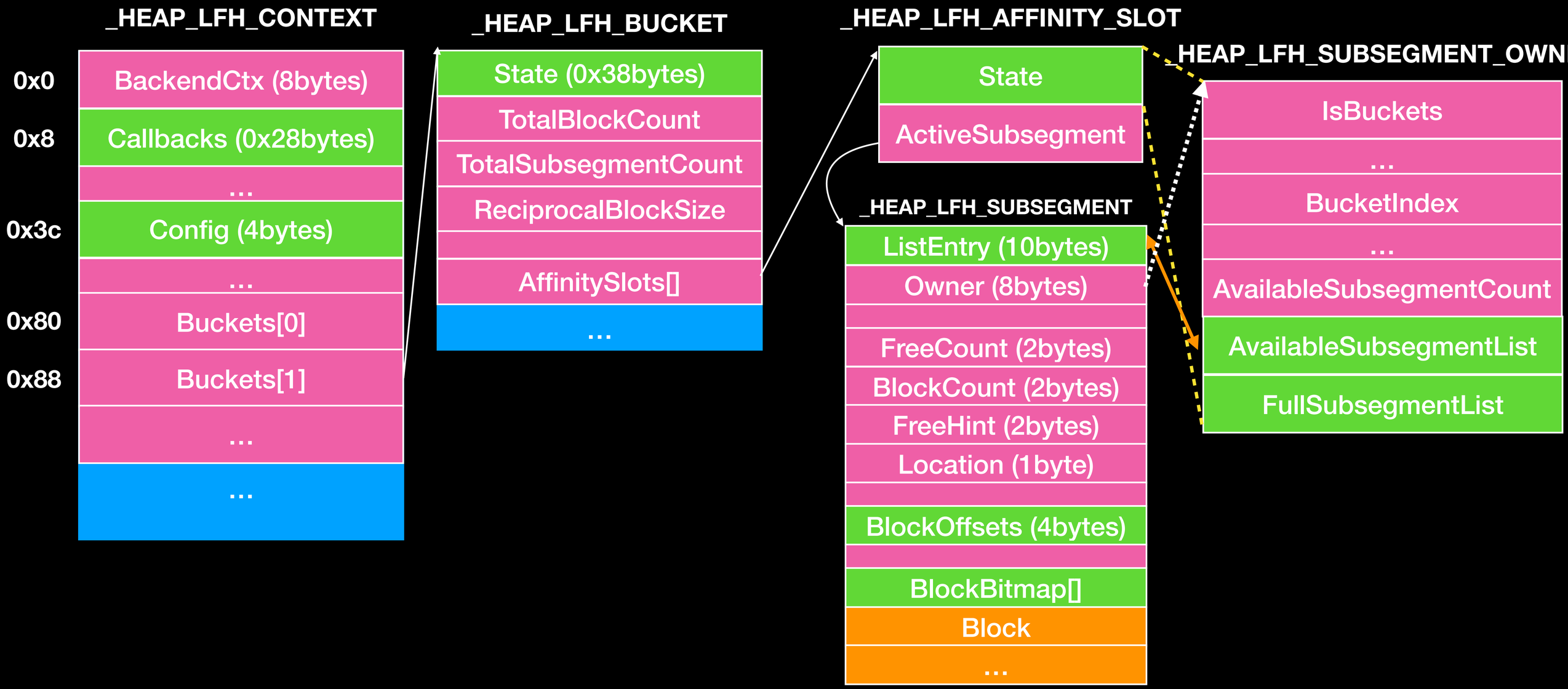
- LFH Subsegments (_HEAP_LFH_SUBSEGMENT)
  - BlockBitmap
    - Indicates the usage status of the block in the subsegment, and every two bits corresponds to a block
      - bit 0 : is_busy bit
      - bit 1 : unused bytes

Low FragmentationHeap



- LFH Subsegments (_HEAP_LFH_SUBSEGMENT)
  - Block
    - The allocated memory return to user.
    - If the unused byte in the corresponding BlockBitmap is 1, then the last two bytes of the block are used to represent unused bytes
    - If there is only 1 byte, it will be recorded as 0x8000

Low FragmentationHeap



Low FragmentationHeap

- Data Structure
- Memory allocation mechanism

Low FragmentationHeap

- Allocate
  - When allocating memory smaller than LfhContext->Config.MaxBlockSize, it will first check whether the corresponding bucket is LFH enabled
    - bucket index = RtlpLfhBucketIndexMap[needbytes+0xf]
    - Check : Buckets[index]->State & 1 == 1 (disable)
    - If it’s not enabled, it will update(RtlpHpLfhBucketUpdateStats) the usage times corresponding bucket.

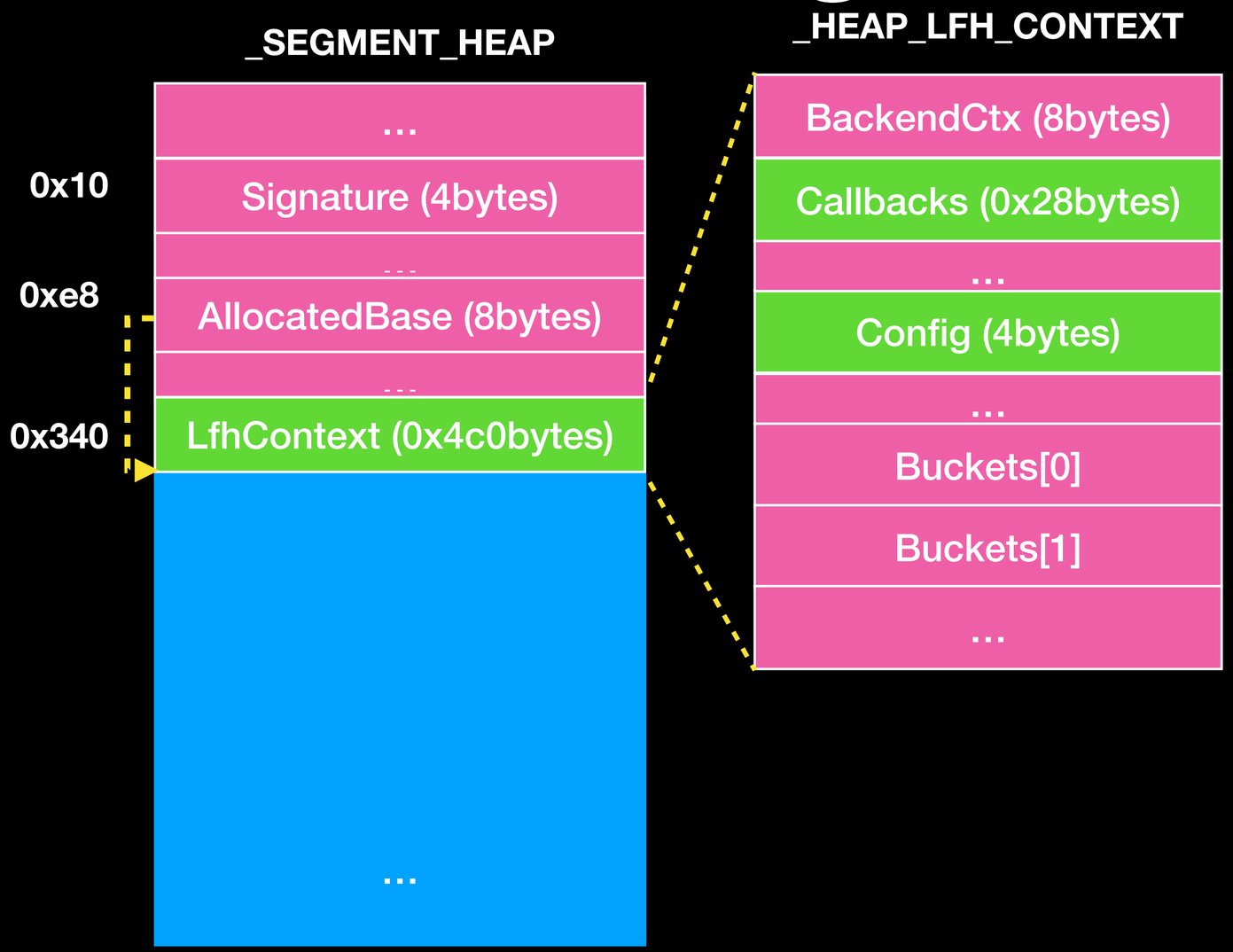
Low FragmentationHeap

- Allocate
  - When allocating memory smaller than LfhContext->Config.MaxBlockSize, it will first check whether the corresponding bucket is LFH enabled
    - The buckets[idx] will record the number of times when it is not enabled
      - When we allocated a chunk : the corresponding buckets[idx] += 0x210000
      - Once (buckets[idx] >> 16) & 0x1f> 0x10 or (buckets[idx] >> 16)> 0xff00 it will enable LFH (RtlpHpLfhBucketActivate)

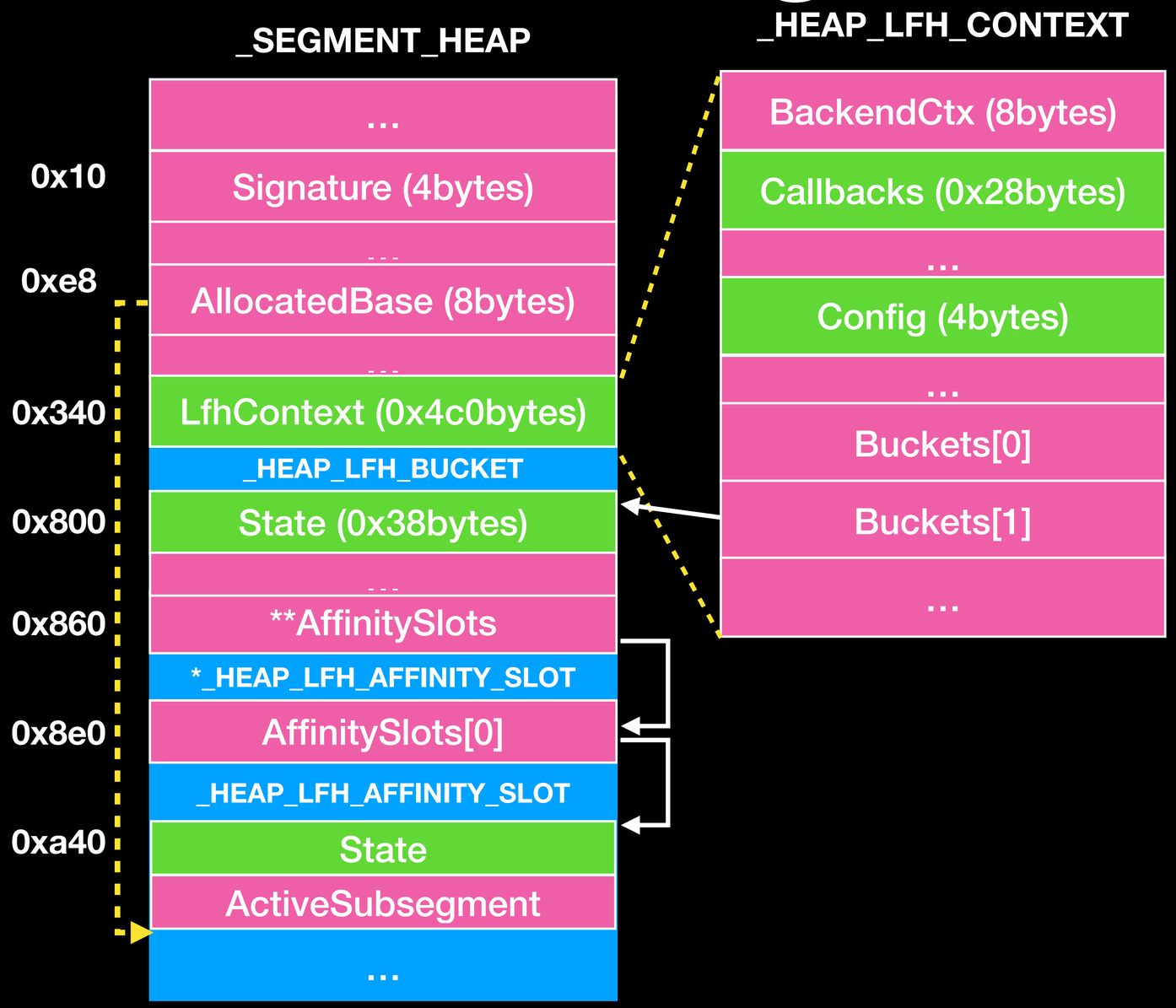
Low FragmentationHeap

- Allocate
  - About enable LFH (RtlpHpLfhBucketActivate)
    - At the beginning, it will allocate the required structure of the bucket (RtlpHpHeapExtendContext)
      - Bucket, owner, affinityslot …
    - The allocation method is directly allocated from _SEGMENG_HEAP->AllocatedBase (usually pointing to the end of the segment heap structure)
    - After allocation, the bucket related structure will be initialized by RtlpHpLfhBucketInitialize

Low FragmentationHeap



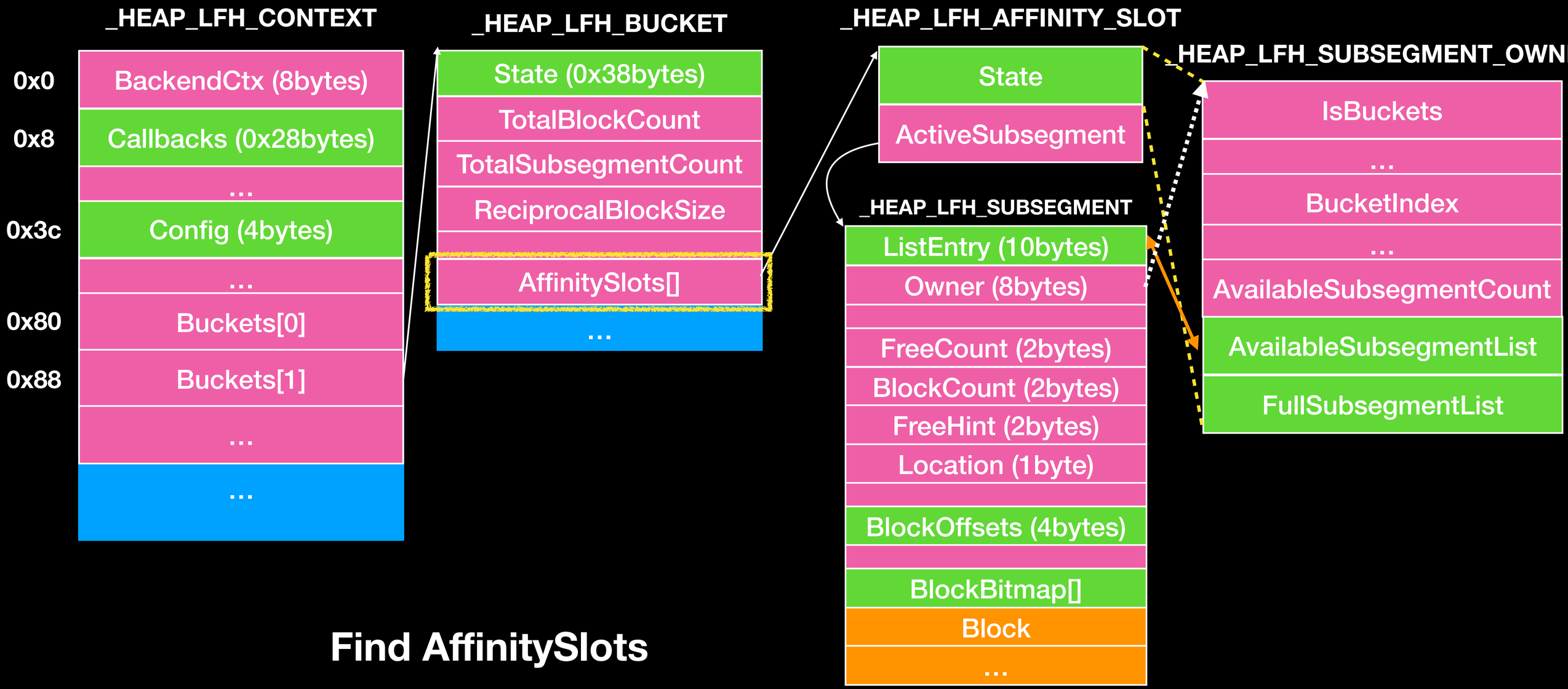
Low FragmentationHeap



Low FragmentationHeap

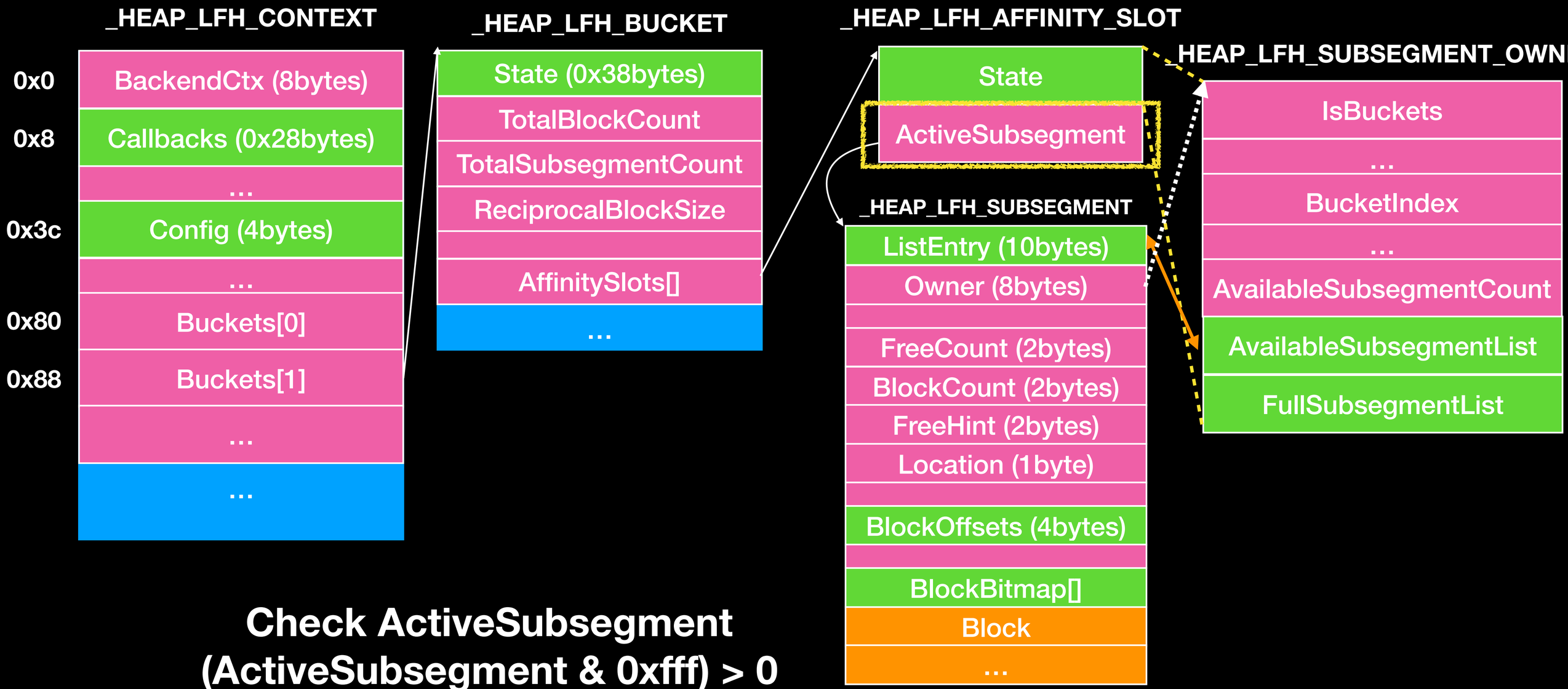
- Allocate
  - After LFH is enabled, LFH will be used as long as the block of the size is allocated
  - Implementation function is nt!RtlpHpLfhSlotAllocate
    - Next, it will check if there is an available block in ActiveSubsegment
      - The check is to take the lowest 12 bits of ActiveSubsegment (representing the number of blocks that can be allocated), and if there are available block, it will allocate from the subsegment
    - If not, it will confirm whether the subsegment really has no block that can be allocated

Low FragmentationHeap



Find AffinitySlots

Low FragmentationHeap



Check ActiveSubsegment
(ActiveSubsegment & 0xfff) > 0

Low FragmentationHeap

- Allocate
  - Which block will be taken?
    - In the case that the subsegment have available blocks, it will be similar to the LFH block in NtHeap, it would take the value of RtlpLowFragHeapRandomData[x] first.
      - Next time it will retrieve the value from RtlpLowFragHeapRandomData[x+1]
      - x is 1 byte ，x = rand() % 256 after 256 rounds
      - RtlpLowFragHeapRandomData is a 256-bytes array filled with random value
        - The range of random value is 0x0 - 0x7f

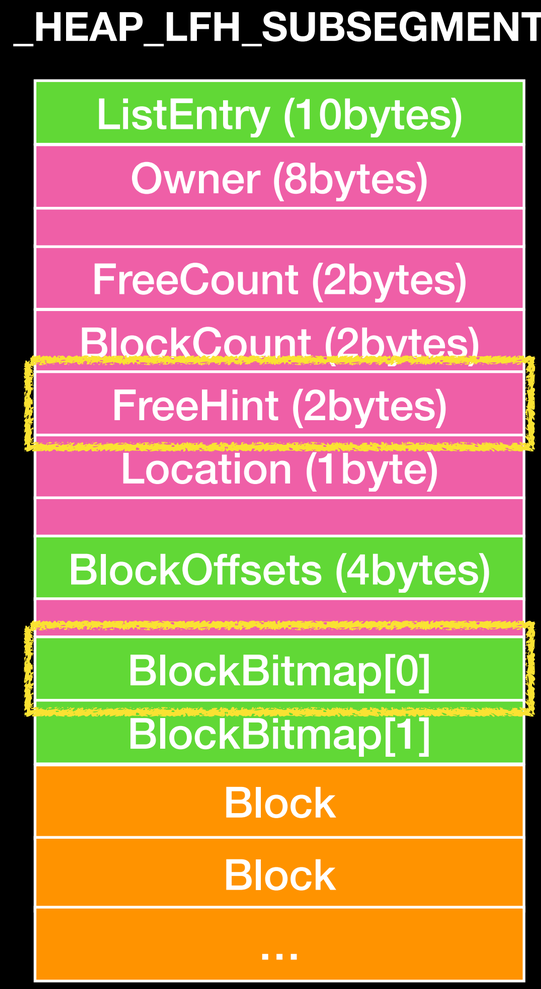
Low FragmentationHeap

- Allocate
    - Next, it will retrieve value of Bitmap[Index] according to the value of subsegment->FreeHint
        - Index = (2*freehint) >> 6
    - After retrieving the value of the Bitmap, it will look for the bit corresponding first allocated block
    - Then adding the random index, it is the block to be allocated

Low FragmentationHeap

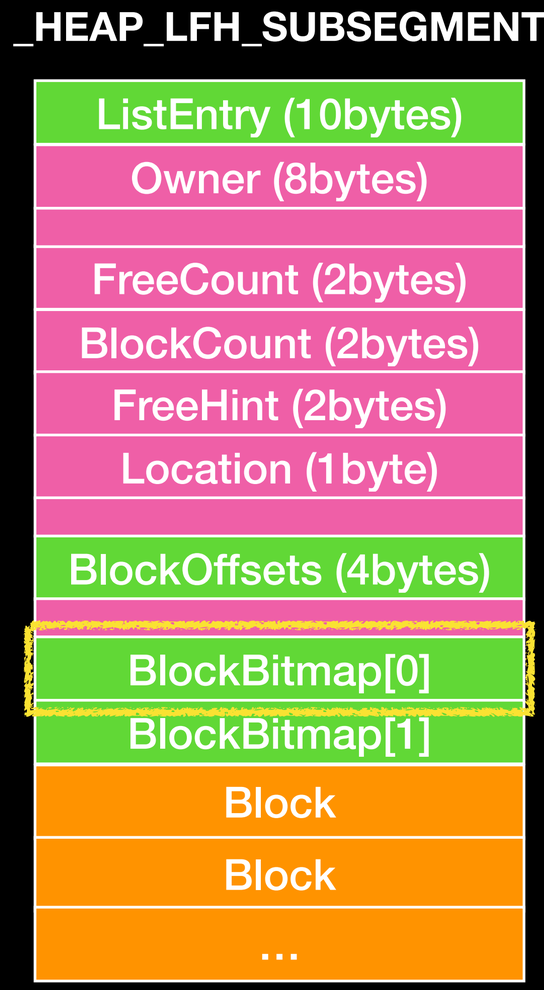
- Allocate
    - If the block is allocated, it will tack the next nearest block
    - Finally, after updating the bit map and unused byte, it will return the memory block

Low FragmentationHeap



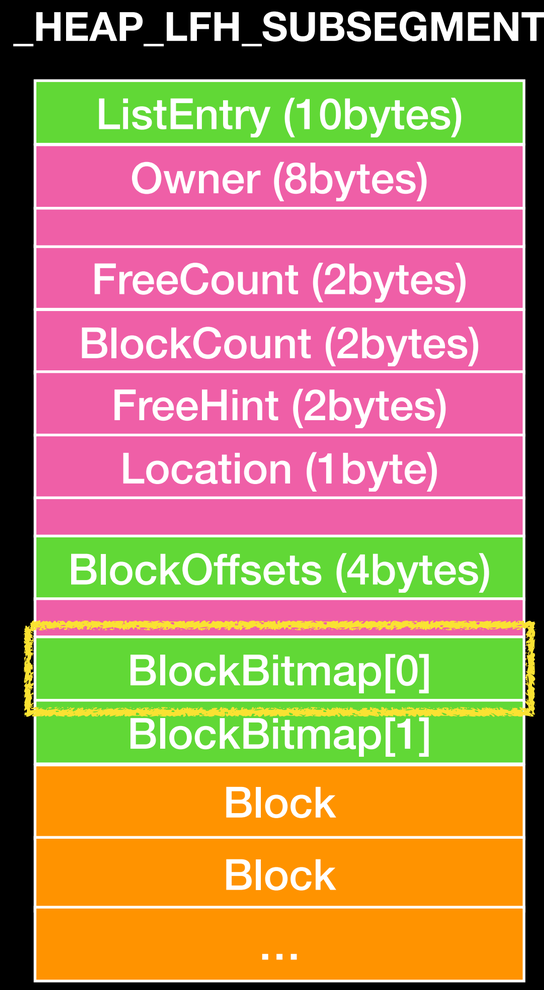
- Get an index
  - Searchwidth = RtlpSearchWidth[Bucketindex];
  - randval = RtlpLowFragHeapRandomData[x]
  - Bitmap = BlockBitmap[(2*freehint) >> 6]

Low FragmentationHeap



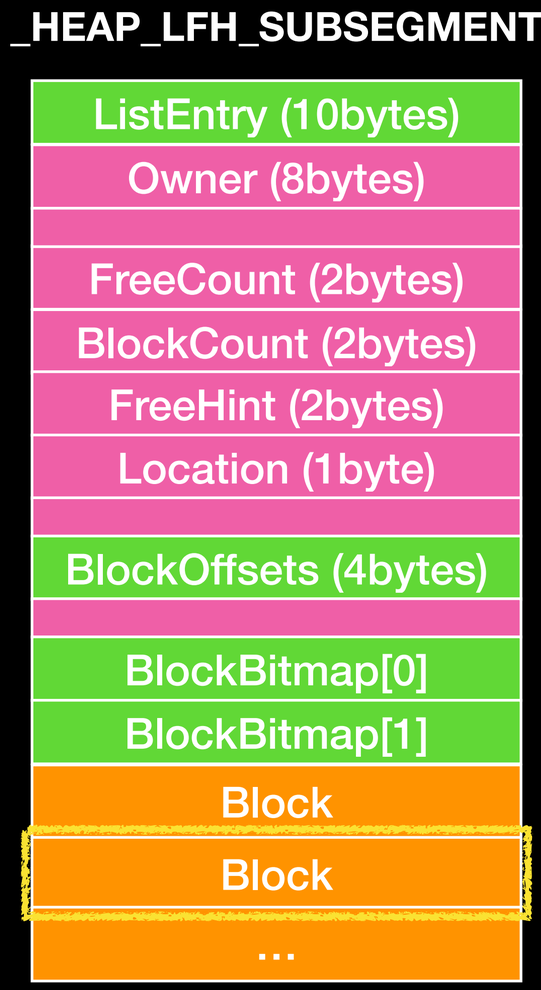
- Get an index
  - Firstnotfreeidx = first not free block in Bitmap
  - val = (searchwidth *randval >> 7) & 0x1FFFFFE
  - blockindex = (firstnotfreeidx + val) & 0x3f

Low FragmentationHeap



- Check if the BusyBitmap correspond to index is 0
  - If it is not zero, it will take the next nearest block
  - If it is zero, it will set the corresponding BlockBitmap, and confirm whether there is unused byte

Low FragmentationHeap



- If there are unused byte, set unusedbyte at the end of the block
- Return the block

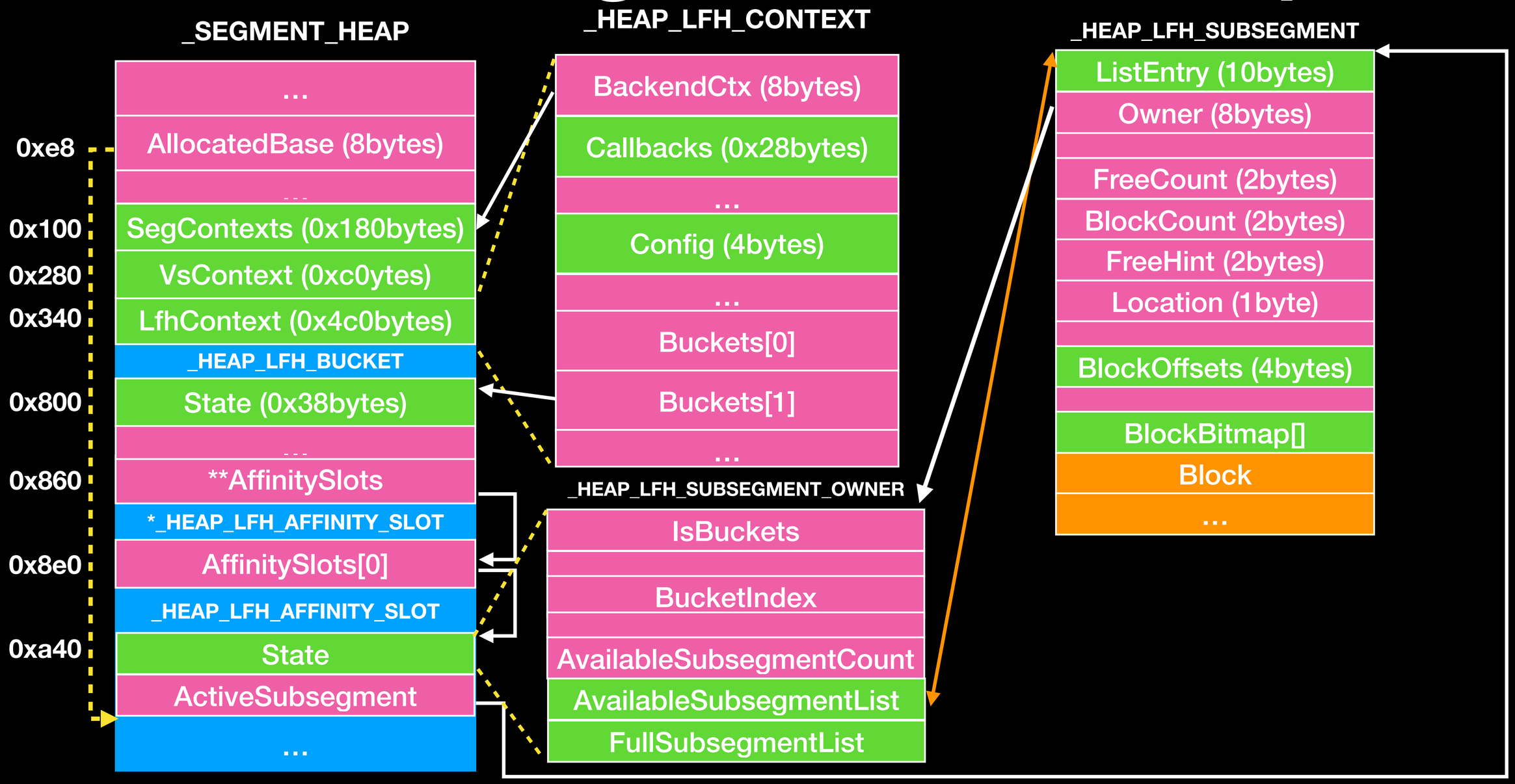
Low FragmentationHeap

- Allocate
    - If the lowest 12 bit of the ActiveSubsegment is 0, it would check whether subsegment->FreeCount is greater than 1
        - if the value is greater than 1 (indicating that the subsegment still have allocatable blocks), it will update the lowest 12 bit at the ActiveSubsegment
    - If the subsegment has no block that can be allocated, it will be filled from Buckets[idx]->State.AvailableSubsegmentList

Low FragmentationHeap

- Allocate
    - In no subsegment available, It would allocate a new subsegment from the back-end allocator and initialize the subsegment (RtlpHpLfhSubsegmentCreate)
        - It will initialize BlockOffsets, BitMap, etc.
    - After initialization, it will add to AffinitySlots->State.AvailableSubsegmentList, and point subsegment->Owner back to AffinitySlots->State
        - AffinitySlots->ActiveSubsegment is also set to this subsegment and the lowest 12 bit is initialized

Low FragmentationHeap



Low FragmentationHeap

- Free
  - Main implementation function is nt!RtlpHpLfhSubsegmentFreeBlock
    - At the beginning, it will take subsegment.BlockOffsets first and decode it
    - According to the content of BlockOffsets, it will retrieve block size and firstblockoffset
    - According to the previous info, the index of block can be calculated
    - Clear the corresponding bitmap and set FreeCount = FreeCount + 1

Low FragmentationHeap

- Free
    - If FreeCount == BlockCount-1 means that all the blocks of the subsegment are released, the subsegment will be removed from the AvaliableSubsegmentList
        - There will also be double linked list checks when removing here
        - It will release subsegment to backend Allocator

Memory Allocator in kernel

- Segment Heap
  - Frontend Allocation
    - Low FragmentationHeap
    - Variable Size Allocation
  - Backend Allocation
    - Segment Allocation
  - Large Block Allocation

Variable Size Allocation

- Size
  - Size <= 0x200 and not enable LFH
  - 0x200 < Size <= 0xfe0
  - 0xfe0 < Size <= 0x20000 and (Size & 0xfff) != 0

Variable Size Allocation

- Data Structure
- Memory allocation mechanism

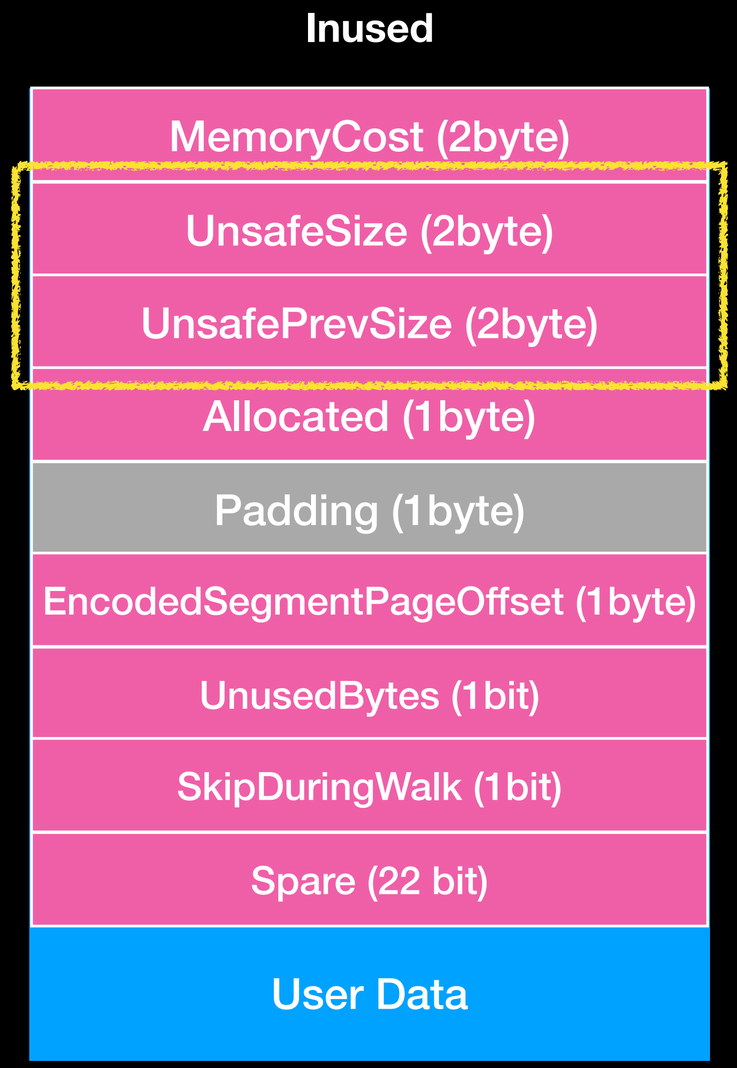
Variable Size Allocation

- Chunk
  - The basic structure of memory allocator
  - In front of the chunk in VS allocation, there are some related metadata that record information of the chunk
    - _HEAP_VS_CHUNK_HEADER
    - _HEAP_VS_CHUNK_FREE_HEADER
  - Similar to back-end allocator in NtHeap

Variable Size Allocation

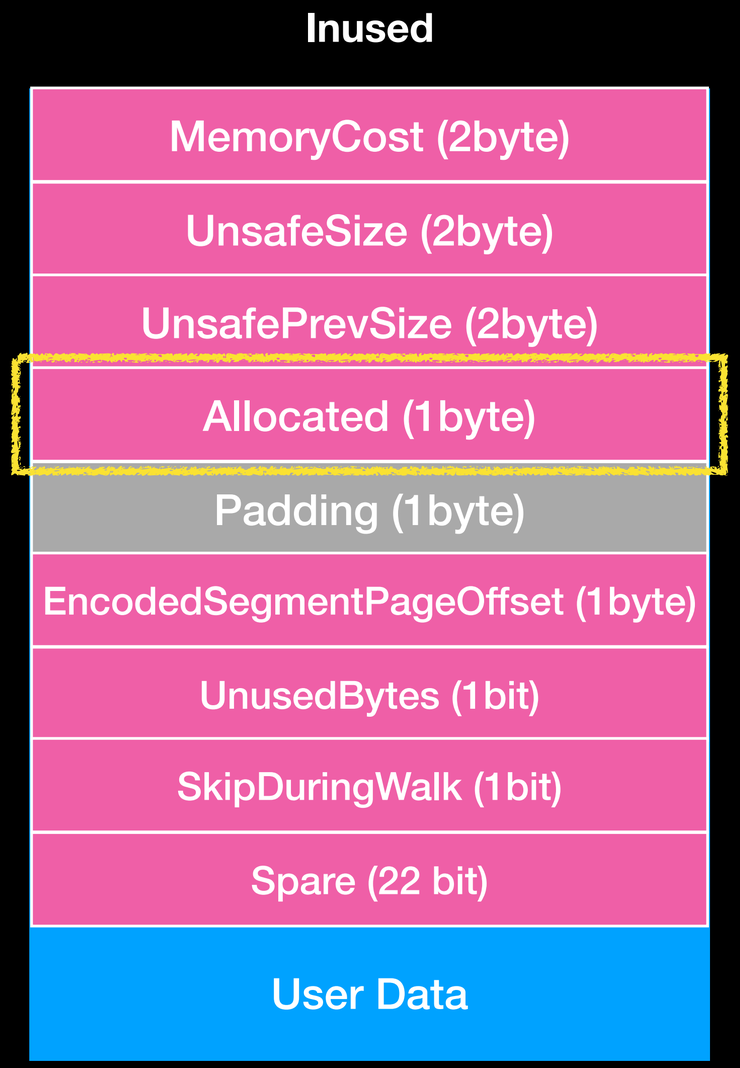
- _HEAP_VS_CHUNK_HEADER
  - MemoryCost
    - Only used when freed

Variable Size Allocation



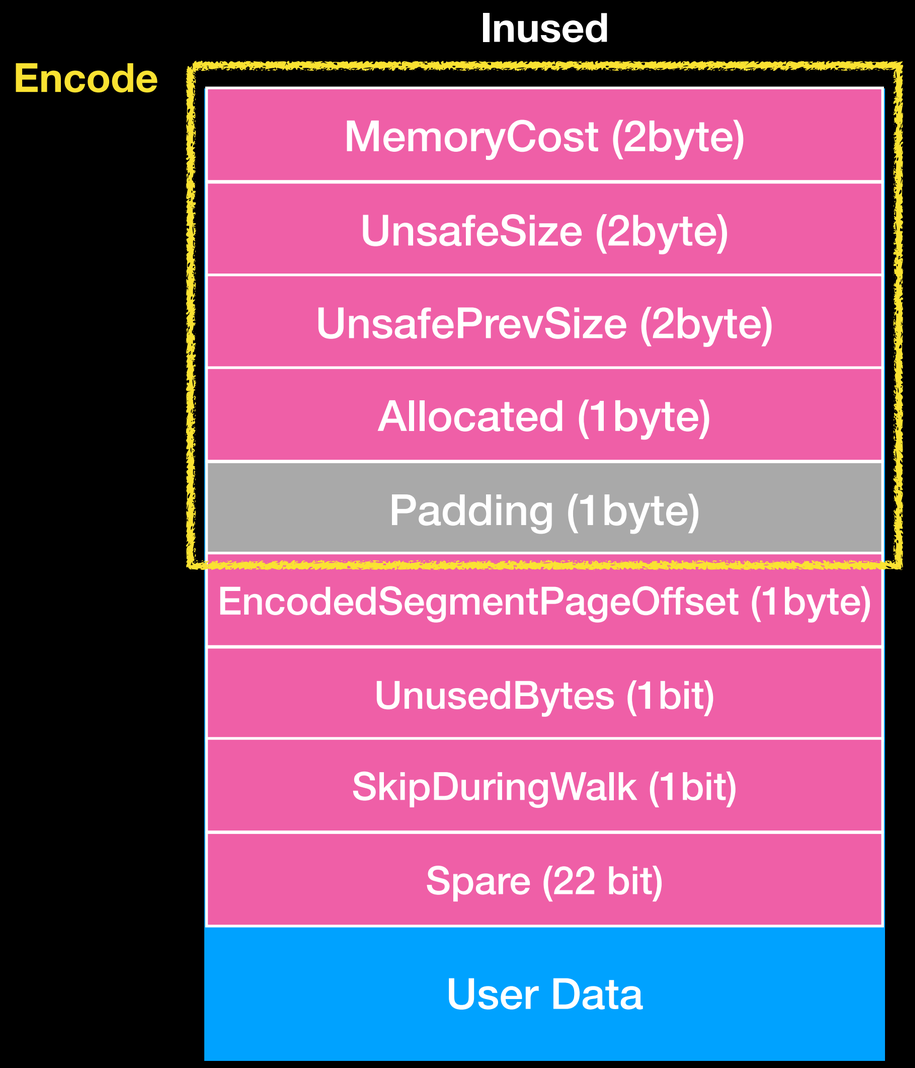
- _HEAP_VS_CHUNK_HEADER
  - UnsafeSize
    - The size of chunk
    - The value is right shift by 4 bits
  - UnasfePrevSize
    - The size of previous chunk
    - The value is right shift by 4 bits

Variable Size Allocation



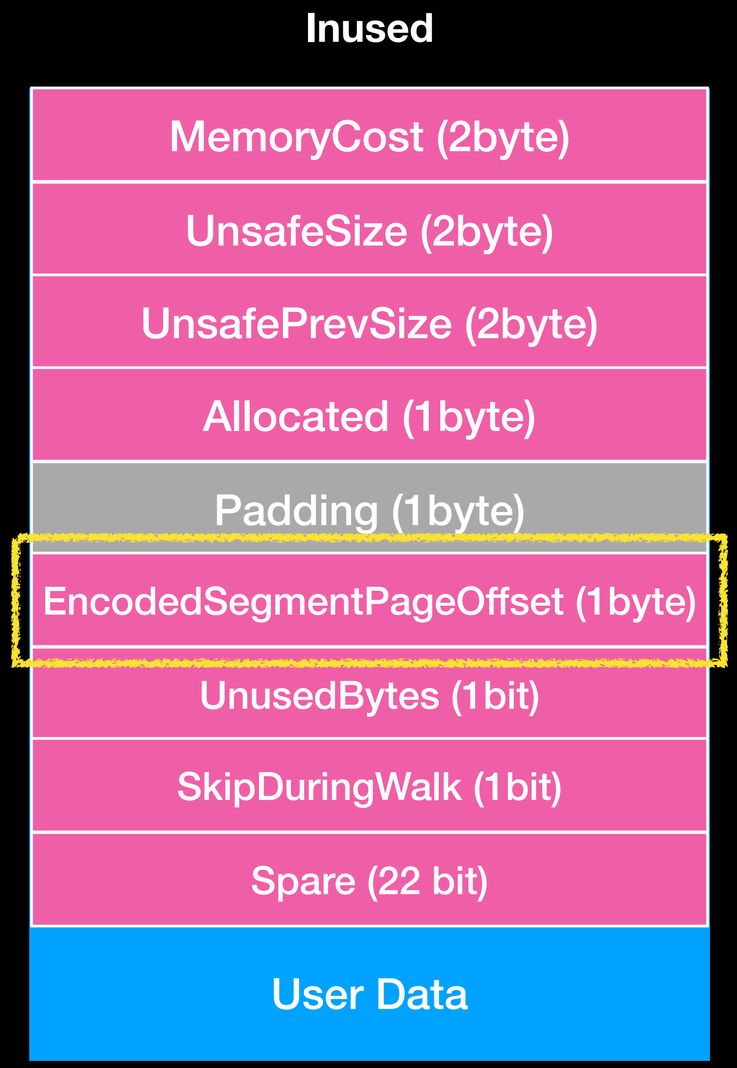
- _HEAP_VS_CHUNK_HEADER
  - Allocated
    - Indicates whether the chunk is allocated
    - The value is 1, if it is allocated

Variable Size Allocation



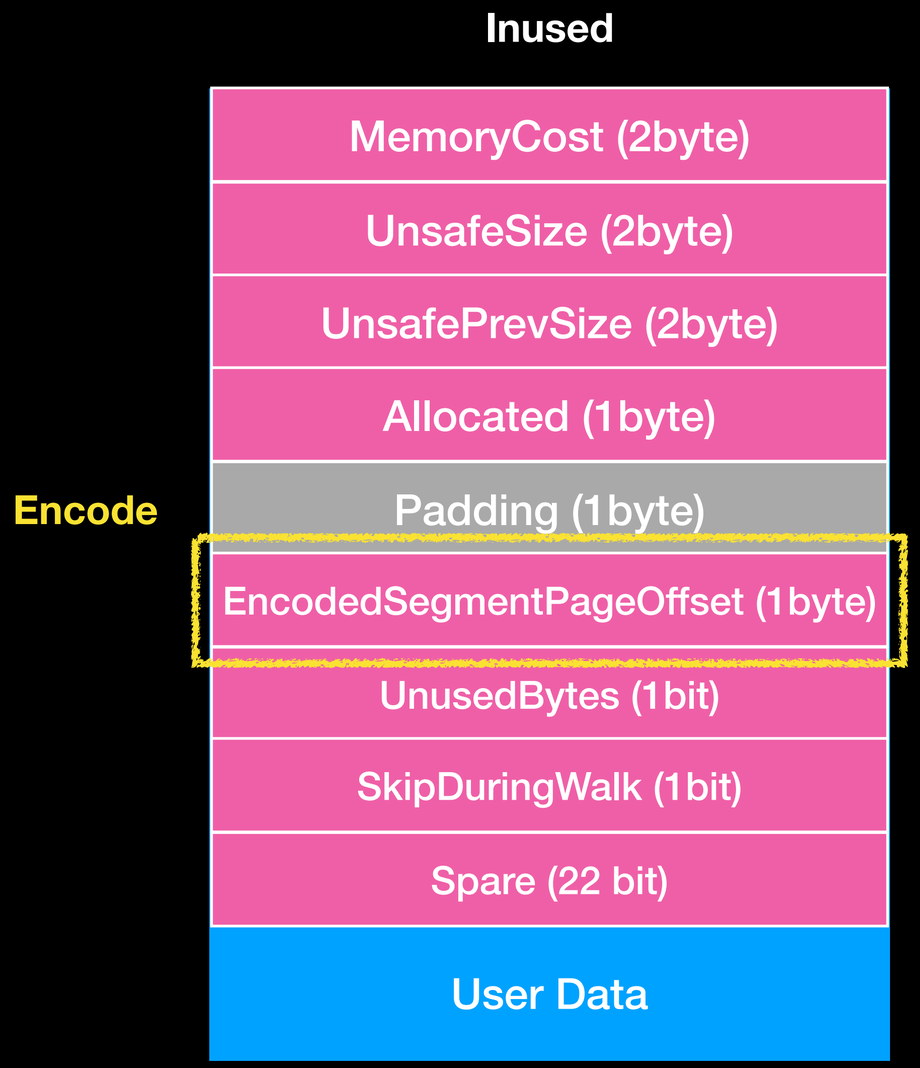
- _HEAP_VS_CHUNK_HEADER
  - Chunk header will be encoded
  - Encoded header is xor with
    - Chunk header
    - Chunk address
    - RtlpHpHeapGlobals.HeapKey

Variable Size Allocation



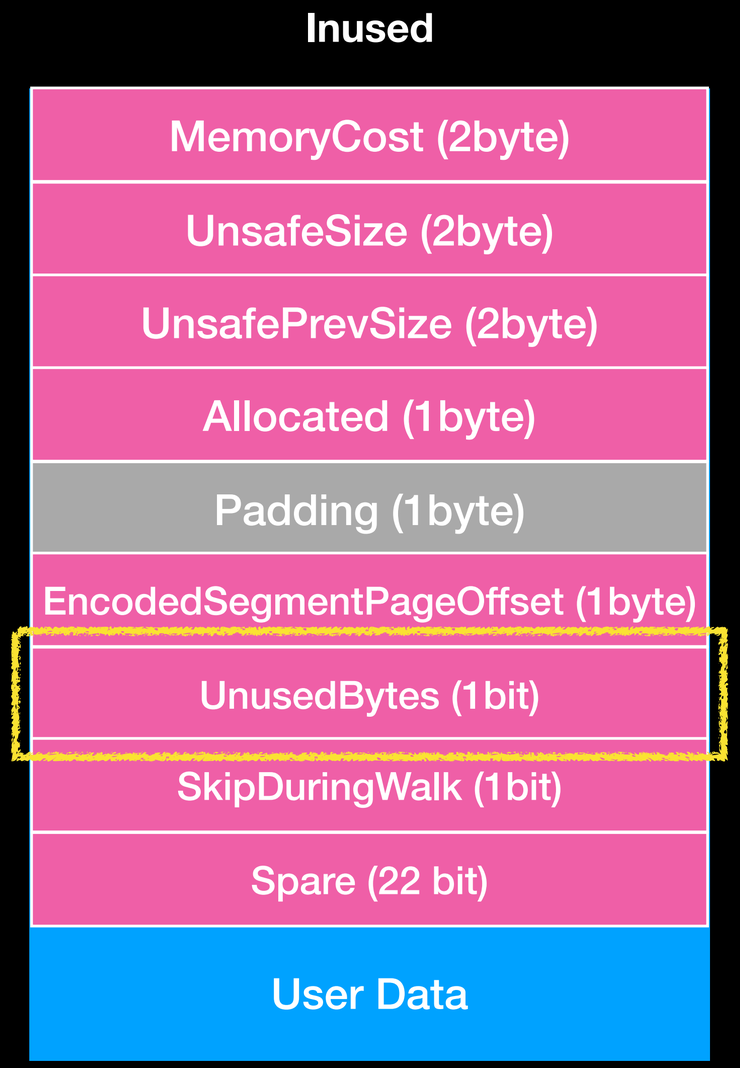
- _HEAP_VS_CHUNK_HEADER
  - EncodedSegmentPageOffset
    - Indicates the index of pages of the chunk in the VS subsegment
    - It is used to find the VS subsegment
    - It will also be encoded.

Variable Size Allocation



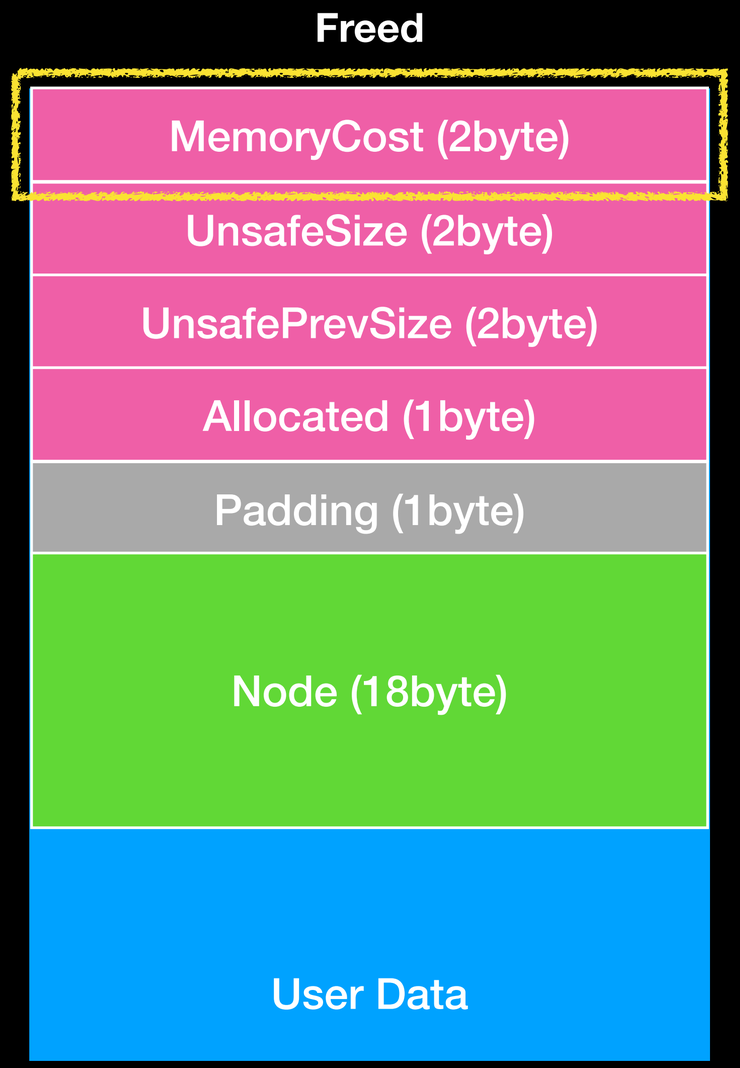
- _HEAP_VS_CHUNK_HEADER
  - EncodedSegmentPageOffset is xor with
    - (int8)Chunk address
    - (int8)RtlpHpHeapGlobals.HeapKey
    - SegmentPageOffset

Variable Size Allocation



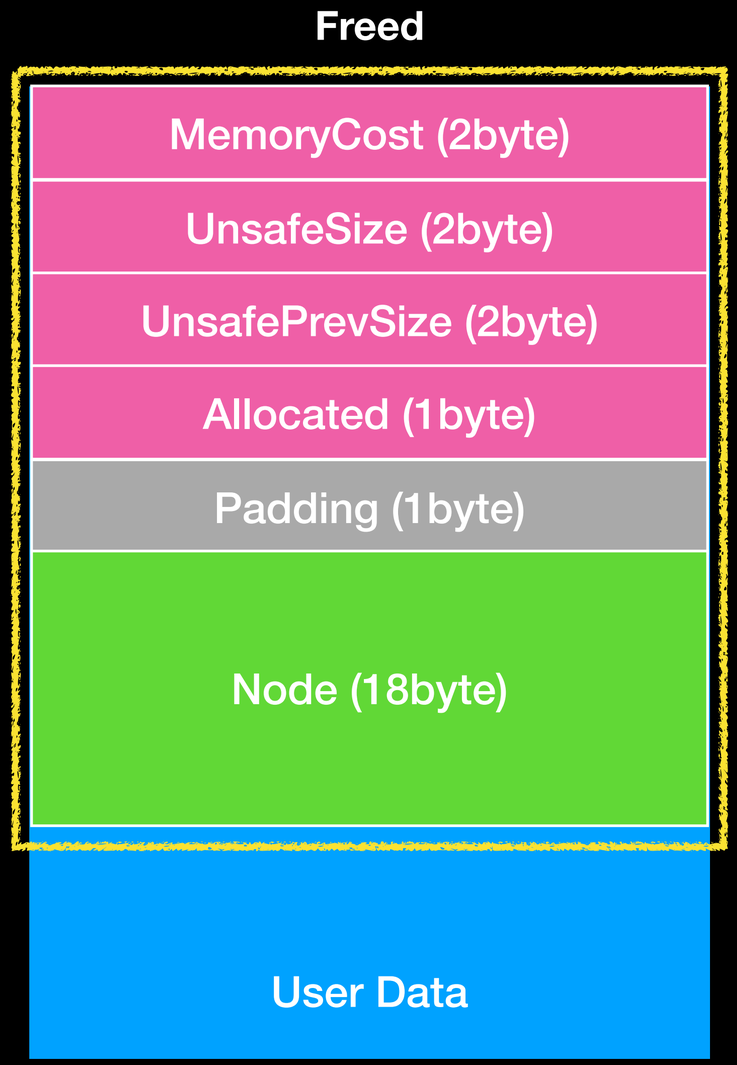
- _HEAP_VS_CHUNK_HEADER
  - UnusedBytes
    - Indicates whether the allocated chunk has unused memory

Variable Size Allocation



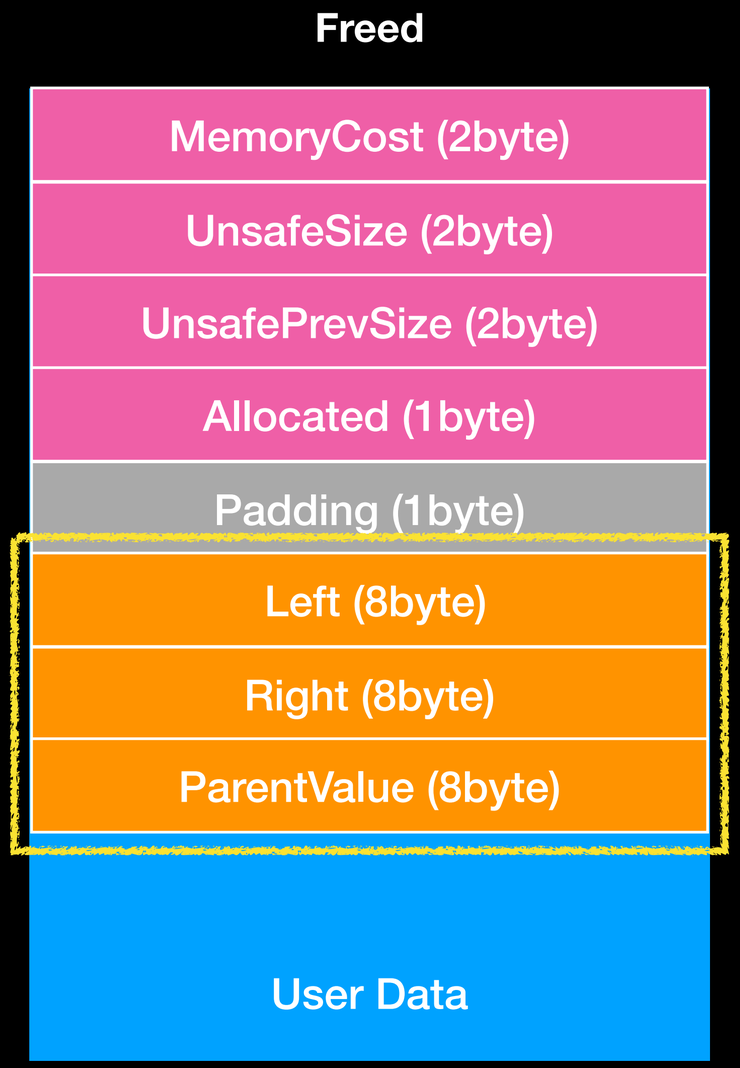
- _HEAP_VS_CHUNK_FREE_HEADER
  - MemoryCost
    - Indicates how many pages of memory need to be committed when the chunk is allocated

Variable Size Allocation



- _HEAP_VS_CHUNK_FREE_HEADER
  - 8 byte header is in the same as allocated chunk
  - Node (_RTL_BALANCED_NODE)
    - Node of rbtree and Freed chunk will be stored in a rbtree structure

Variable Size Allocation

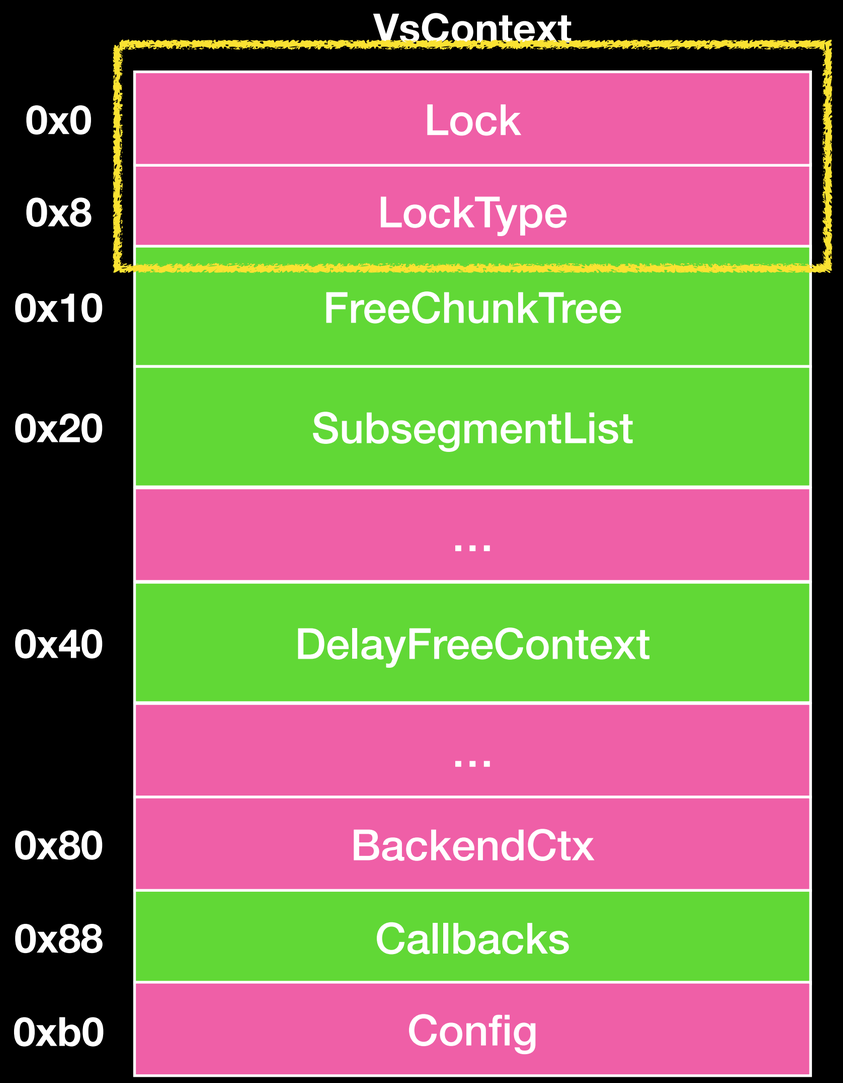


- _HEAP_VS_CHUNK_FREE_HEADER
  - Node (RTL_BALANCED_NODE)
    - Children
      - Left/Right
        - The left and right subtrees of the Node
    - ParentValue
      - Parent node of this node

Variable Size Allocation

- VsContext (_HEAP_VS_CONTEXT)
  - The core structure of the VS Allocator
  - Used to manage the memory allocated by VS allocator, and record all the information and structure of VS allocator in the heap

Variable Size Allocation

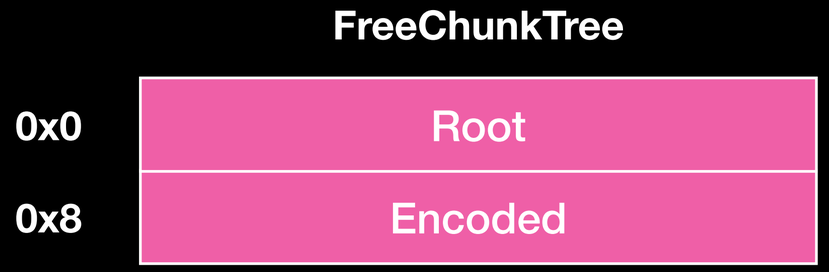


- VsContext (_HEAP_VS_CONTEXT)
  - Lock
    - Just for lock
  - LockType (_RTLP_HP_LOCK_TYPE)
    - The type of Lock can be divided into
      - Paged/NonPaged/TypeMax

Variable Size Allocation

- FreeChunkTree (_RTL_RB_TREE)
  - In VS Allocation, after free a chunk, the chunk will be placed in the FreeChunkTree of the heap, and the chunk will be inserted in the FreeChunkTree according to the size.
  - If the size of chunk is larger than the node, it will be placed in right subtree otherwise, will be placed in left subtree
  - If there is no larger chunk than the chunk, the right subtree is NULL, and the other side is also
  - There will be a node check when taken out of the tree

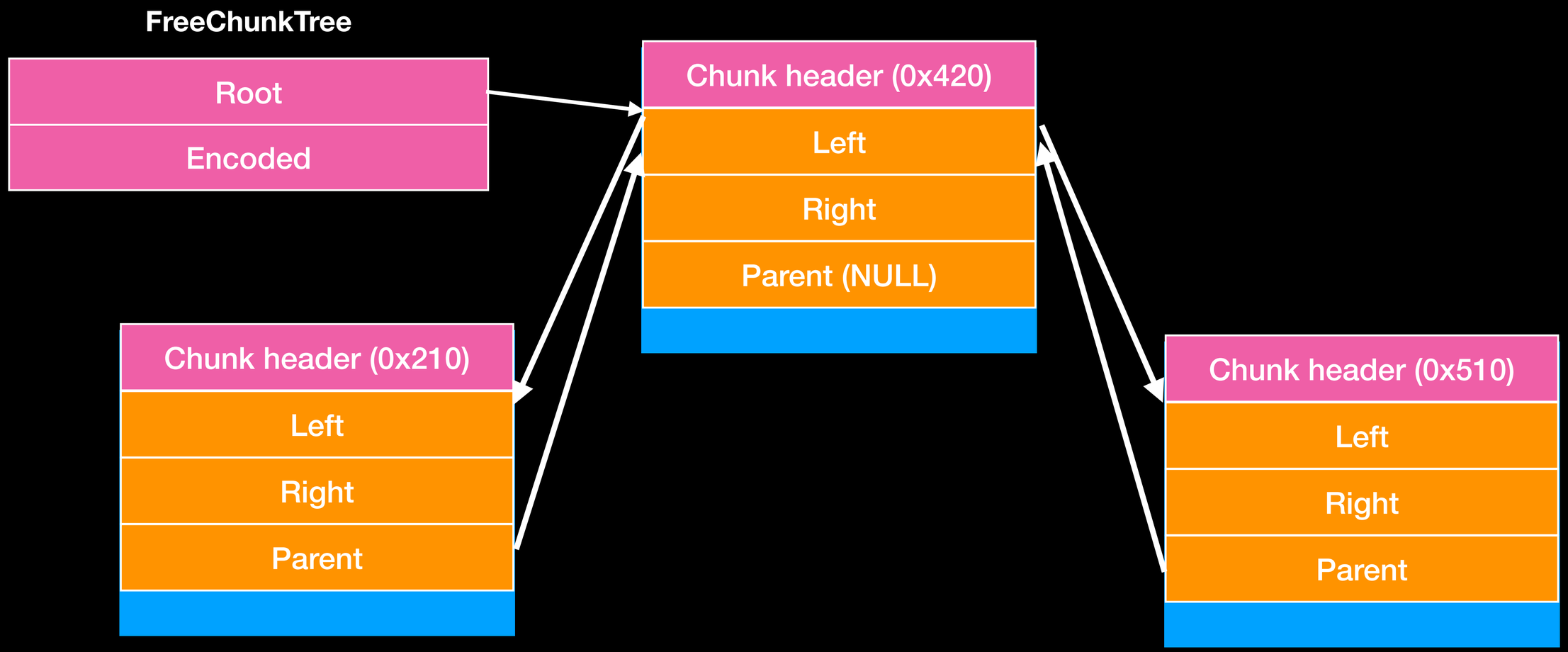
Variable Size Allocation



- FreeChunkTree (_RTL_RB_TREE)
  - Root
    - Point to the root of the rbtree
  - Encoded
    - Indicates whether the root has been encoded (default disable)
  - About encode :
    - EncodedRoot = Root ^ FreeChunkTree

Variable Size Allocation

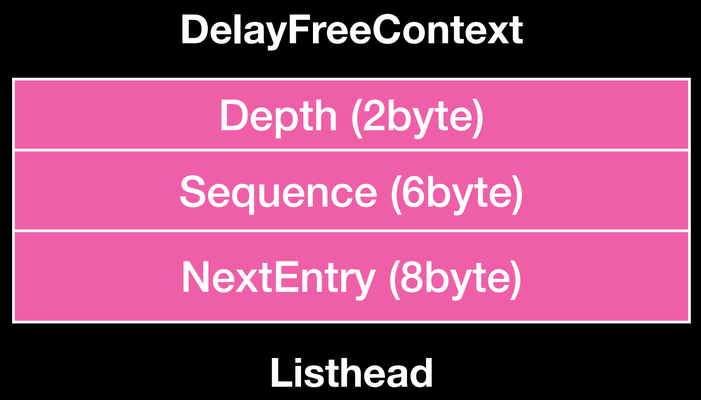
- FreeChunkTree (_RTL_RB_TREE)



Variable Size Allocation

- DelayFreeContext (_HEAP_VS_DELAY_FREE_CONTEXT)
  - When enable delay free (default in kernel but disable in usermode) and size of chunk < 0x1000, after Free a chunk, it will not be free immediately, but will be added to a singly linked list called DelayFreeContext, until the number of chunks in the linked list exceeds 0x20, the chunks in the linked list will be freed at one time
  - Next pointer will be put at the beginning of user data
  - FILO
  - If you want to check whether delay free is enable, you can check VsContext->Config

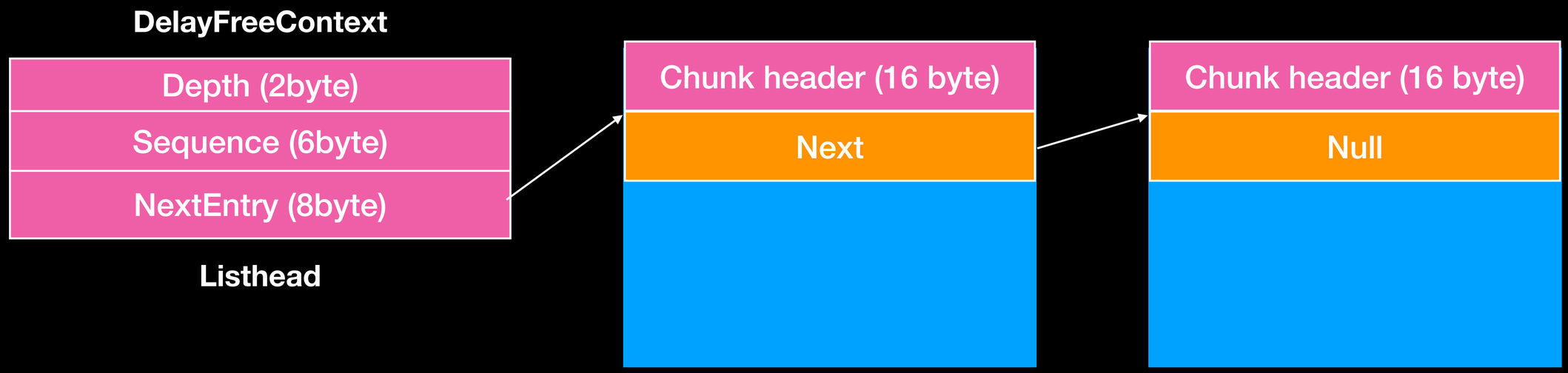
Variable Size Allocation



- DelayFreeContext (_HEAP_VS_DELAY_FREE_CONTEXT)
  - Depth
    - The number of chunks in the Linked list
  - NextEntry
    - Point to next chunk
    - At this time chunk is still Allocated

Variable Size Allocation

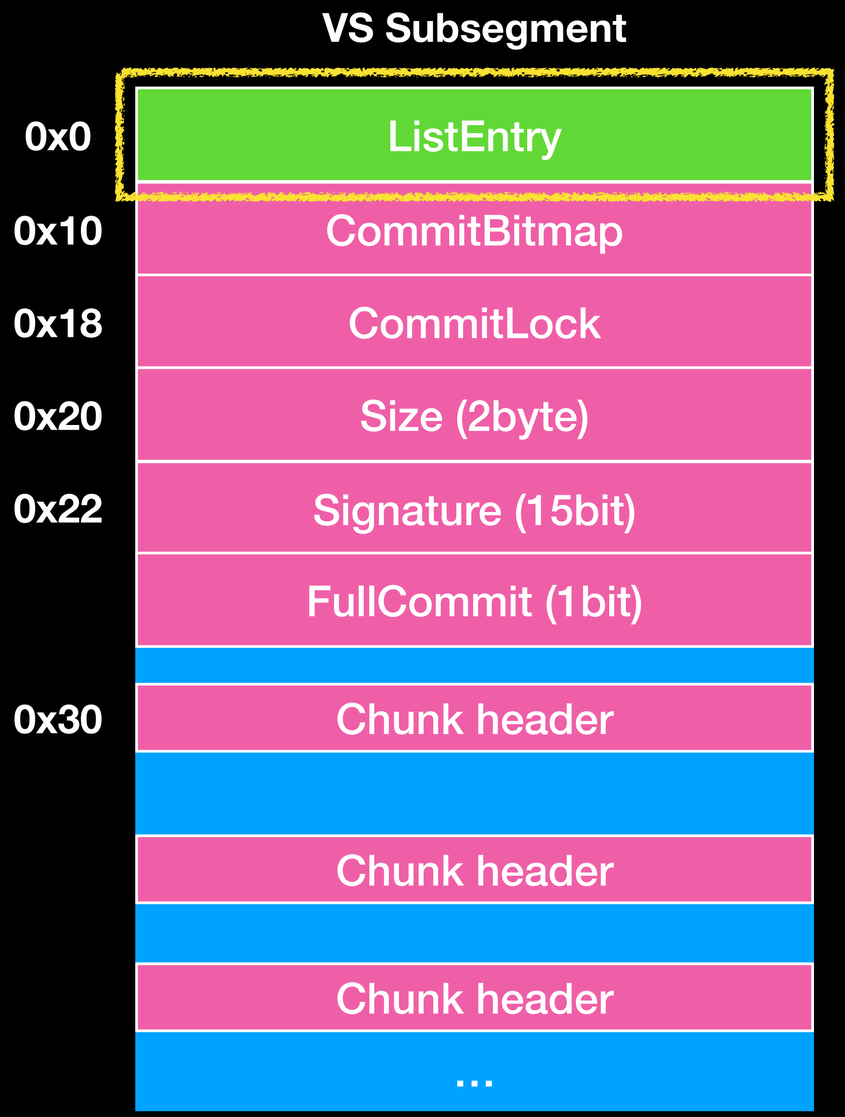
- DelayFreeContext (_HEAP_VS_DELAY_FREE_CONTEXT)



Variable Size Allocation

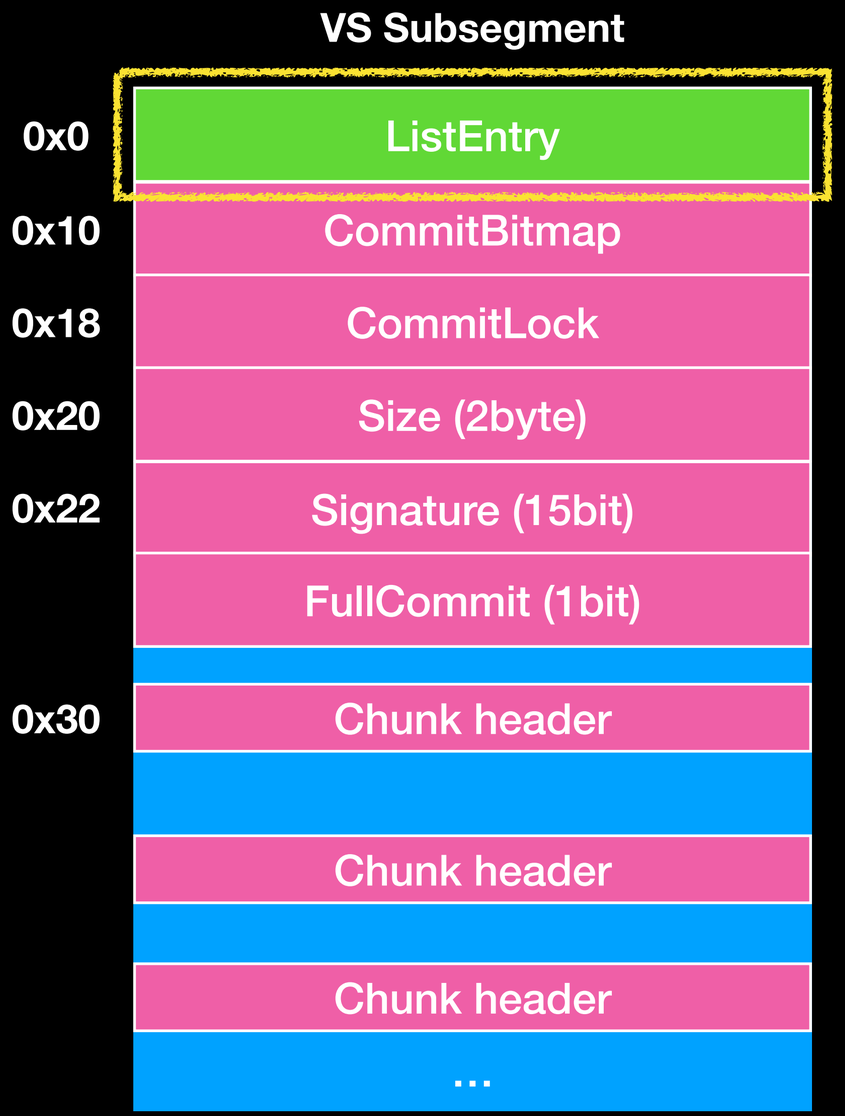
- VS Subsegment (_HEAP_VS_SUBSEGMENT)
  - The memory pool of VS allocation
  - Once there is not enough for allocation, it will allocated from the backend allocator for a new subsegment
  - Each subsegment will be linked in a linked list

Variable Size Allocation



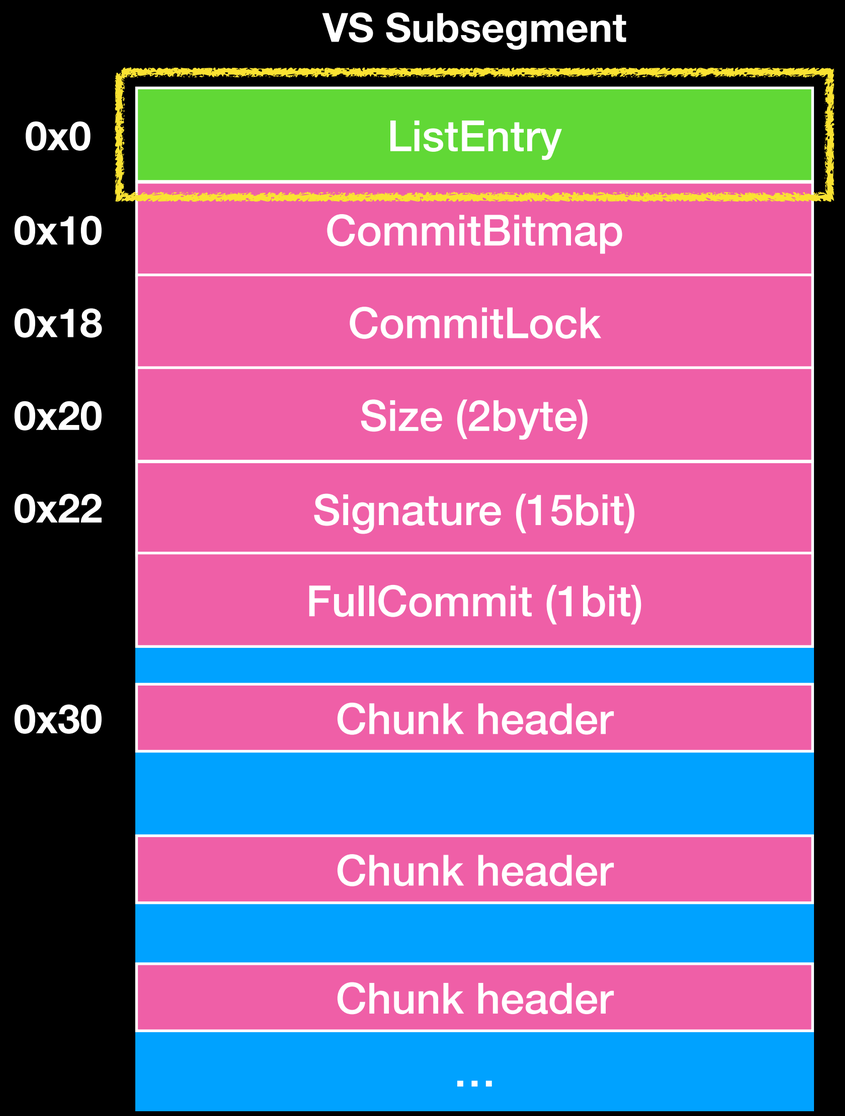
- VS Subsegment (_HEAP_VS_SUBSEGMENT)
  - ListEntry(_LIST_ENTRY)
    - Flink
      - Point to next subsegment
    - Blink
      - Point to previous subsegment

Variable Size Allocation



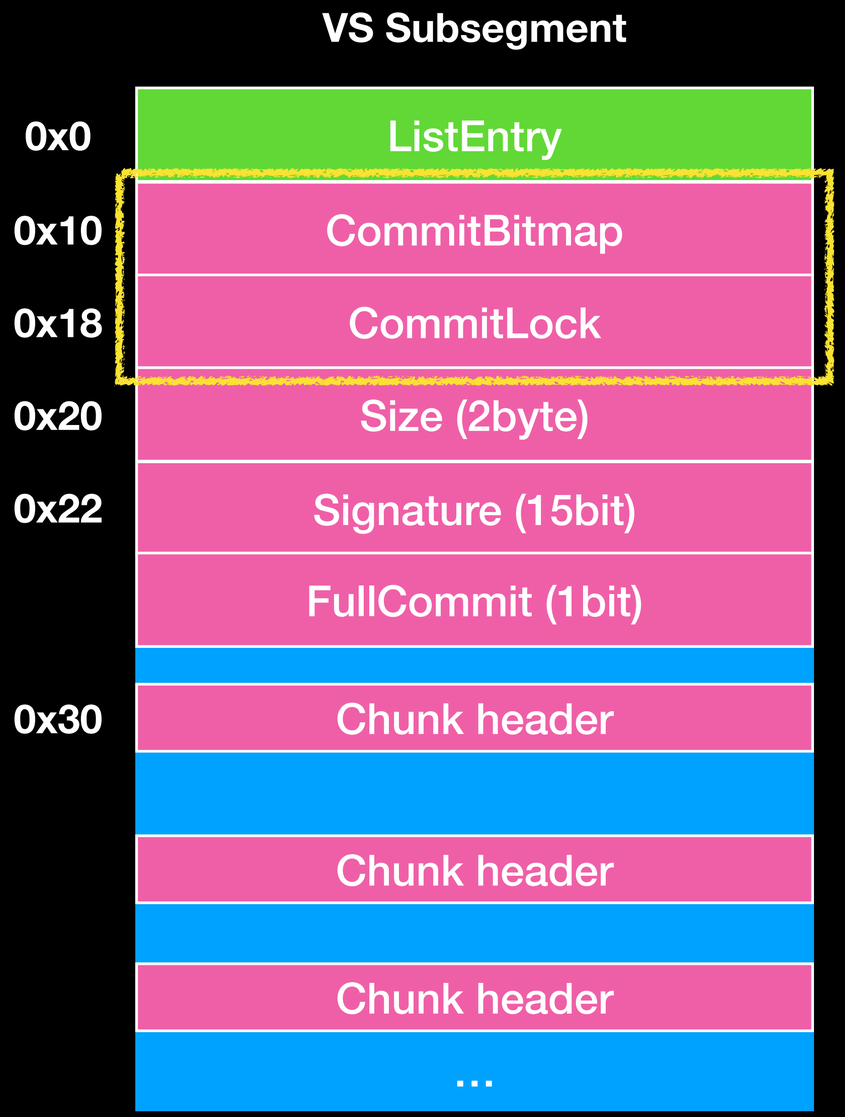
- VS Subsegment (_HEAP_VS_SUBSEGMENT)
  - ListEntry(_LIST_ENTRY)
    - The value would be encoded
      - Flink/Blink will xor with following value
        - Address of ListEntry
        - Address of next/prev subsegment

Variable Size Allocation



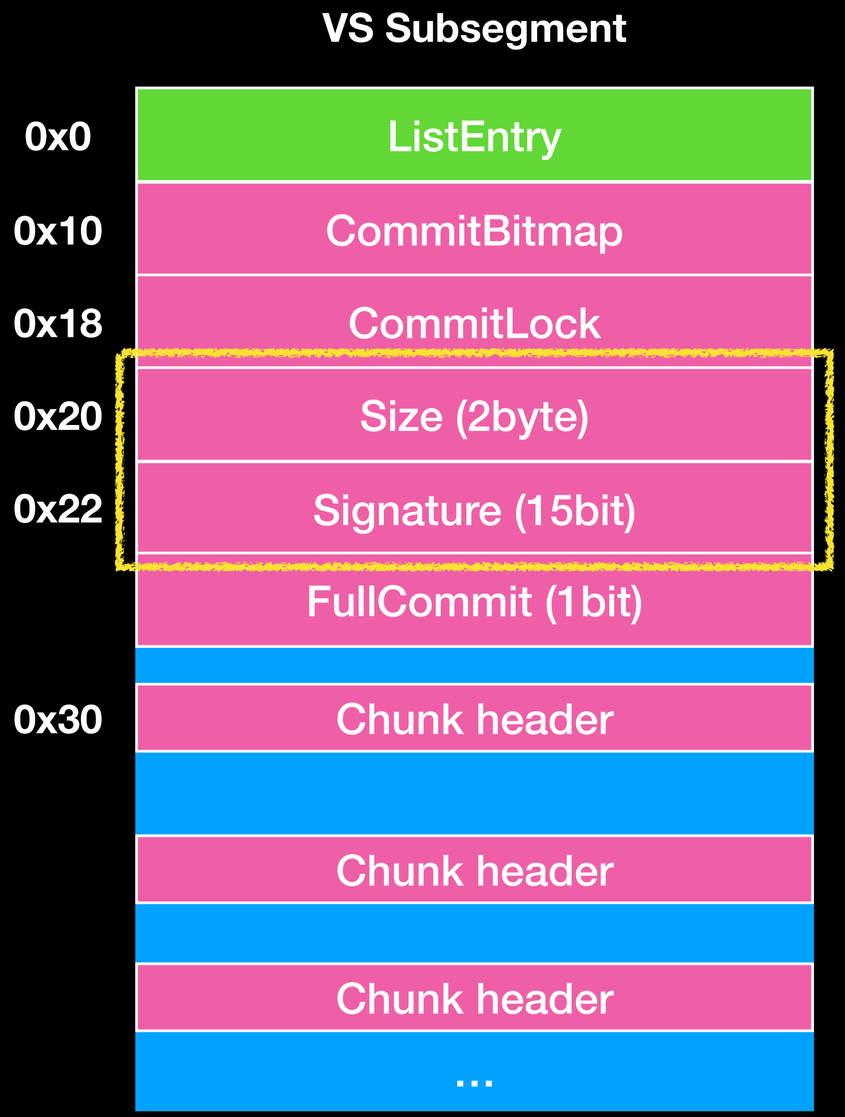
- VS Subsegment (_HEAP_VS_SUBSEGMENT)
  - ListEntry(_LIST_ENTRY)
    - There is also double linked list check here

Variable Size Allocation



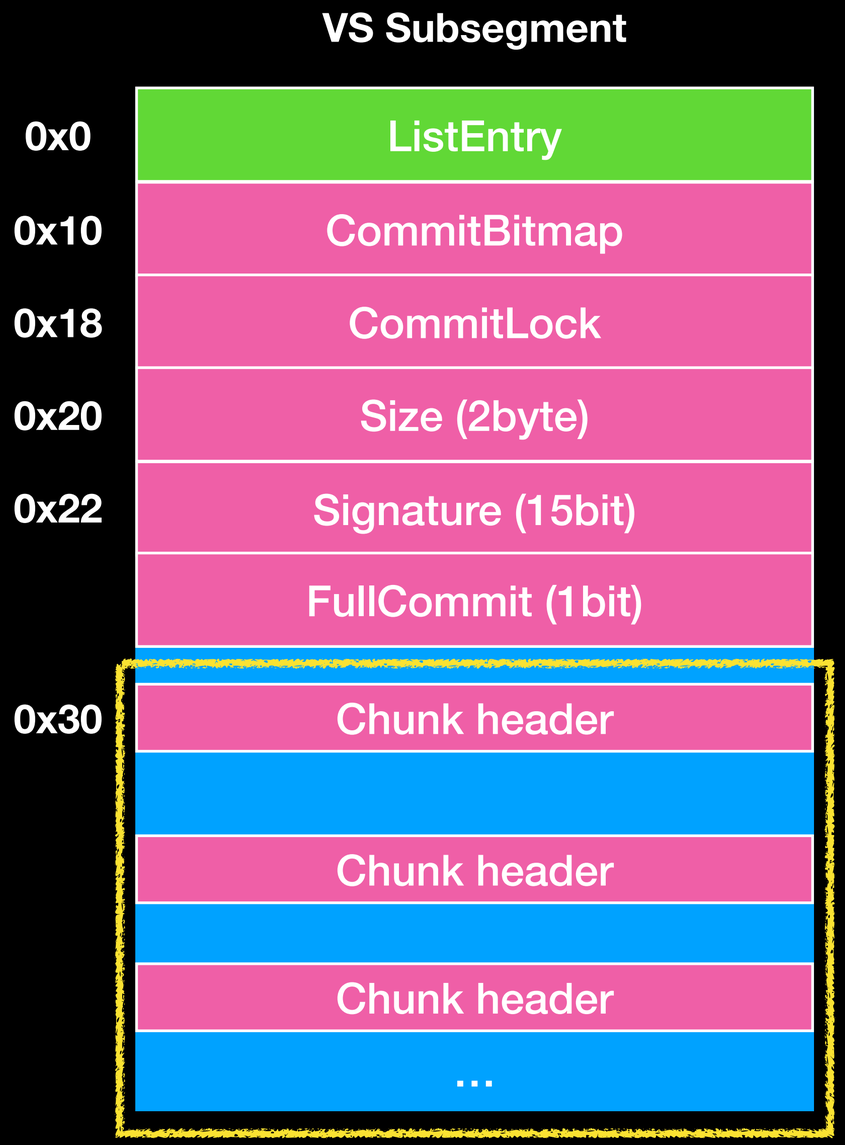
- VS Subsegment (_HEAP_VS_SUBSEGMENT)
  - CommitBitmap
    - Indicates the status of page commit in the subsegment, and the page starts from the beginning of the subsegment
  - CommitLock
    - Lock used when Commit

Variable Size Allocation



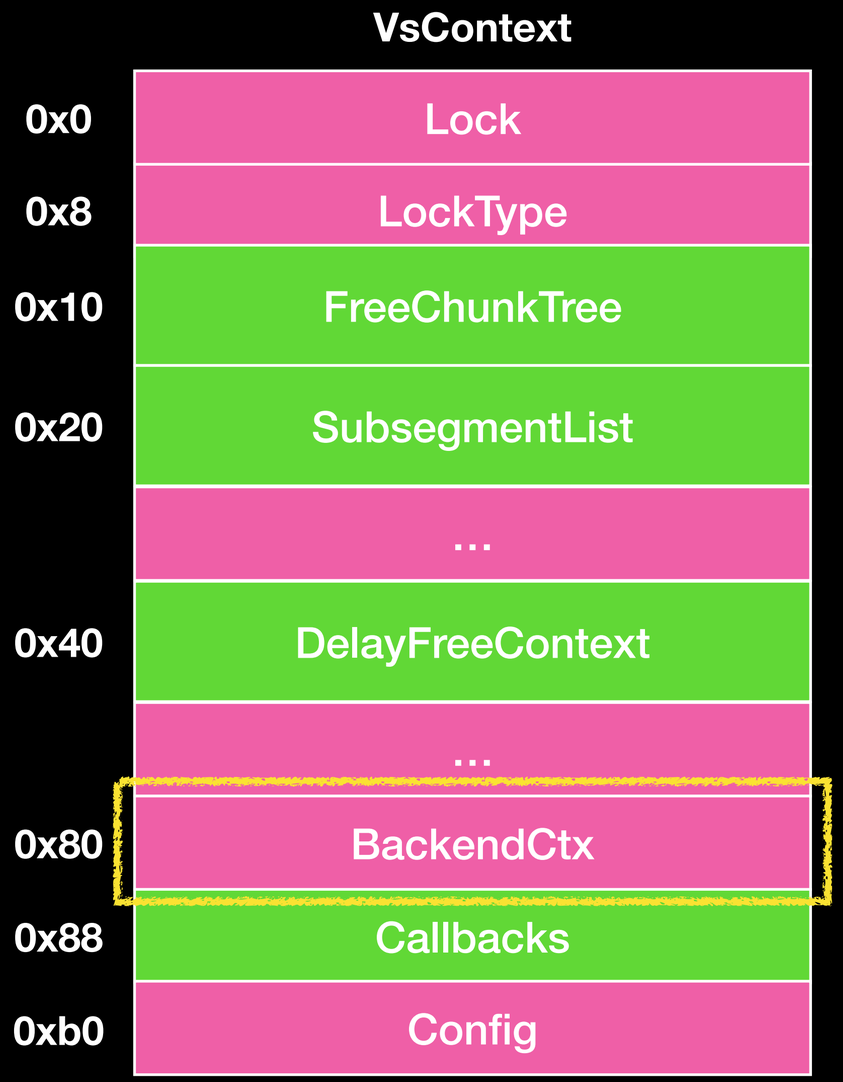
- VS Subsegment (_HEAP_VS_SUBSEGMENT)
  - Size
    - Size of VS subsegment
    - The value is right shift by 4 bits
  - Signature
    - Signature for verification, make sure that the subsegment is found when free

Variable Size Allocation



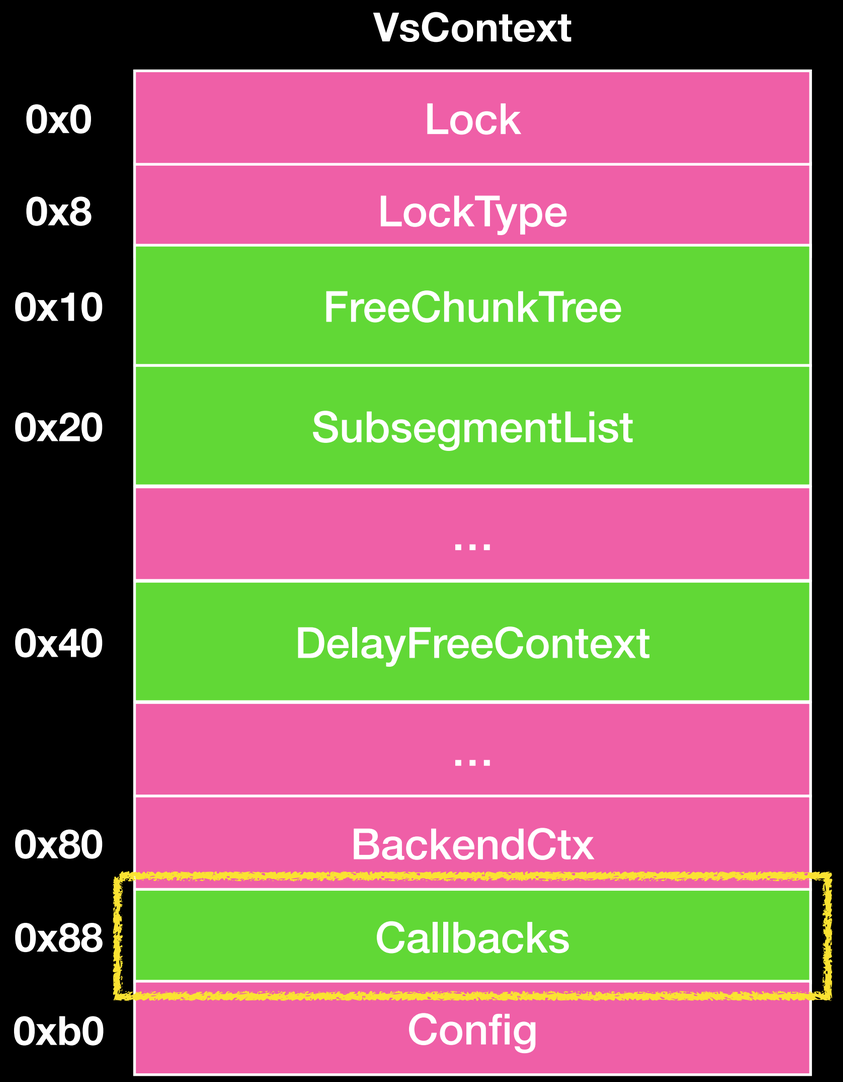
- VS Subsegment (_HEAP_VS_SUBSEGMENT)
  - Behind the VS subsegment header is the memory pool of VS Allocator. At the beginning, the memory pool of subsegment is a large chunk
  - Split when allocated
  - Coalesce to-be-freed block with neighbors

Variable Size Allocation



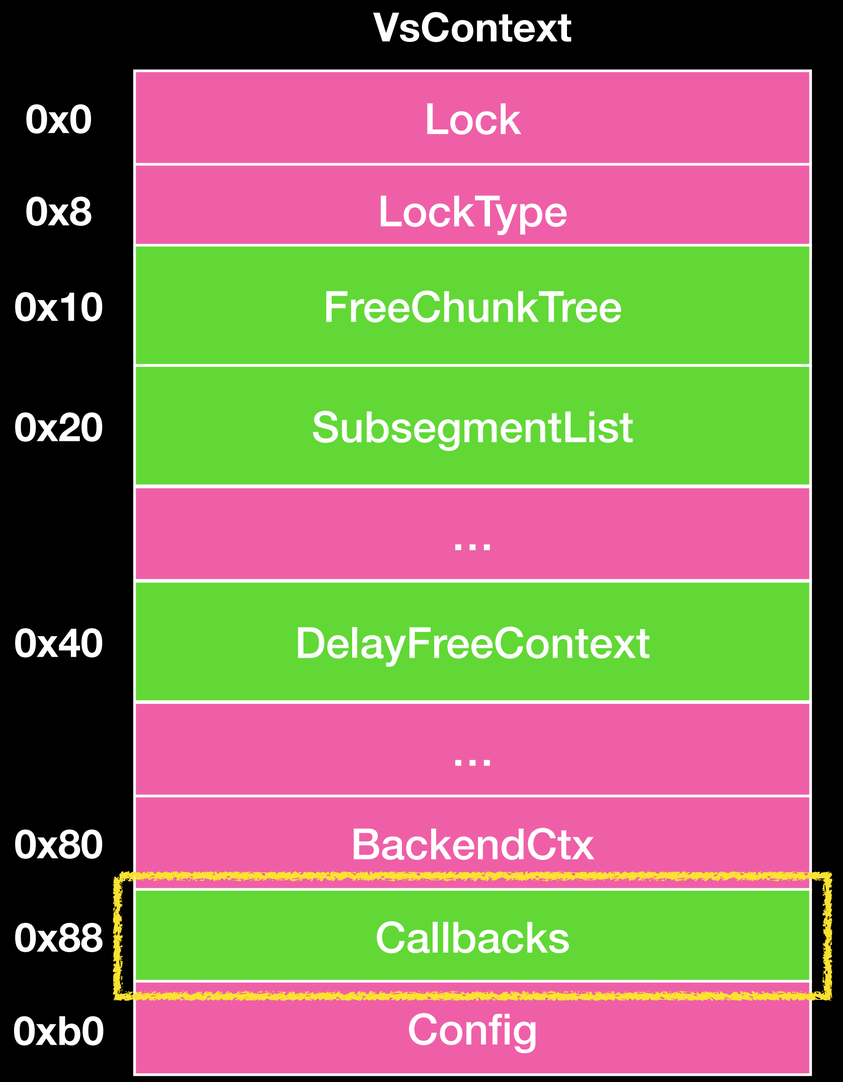
- VsContext (_HEAP_VS_CONTEXT)
  - BackendCtx
    - Point to the Backend allocator (_HEAP_SEG_CONTEXT) structure used by the VS Allocator

Variable Size Allocation



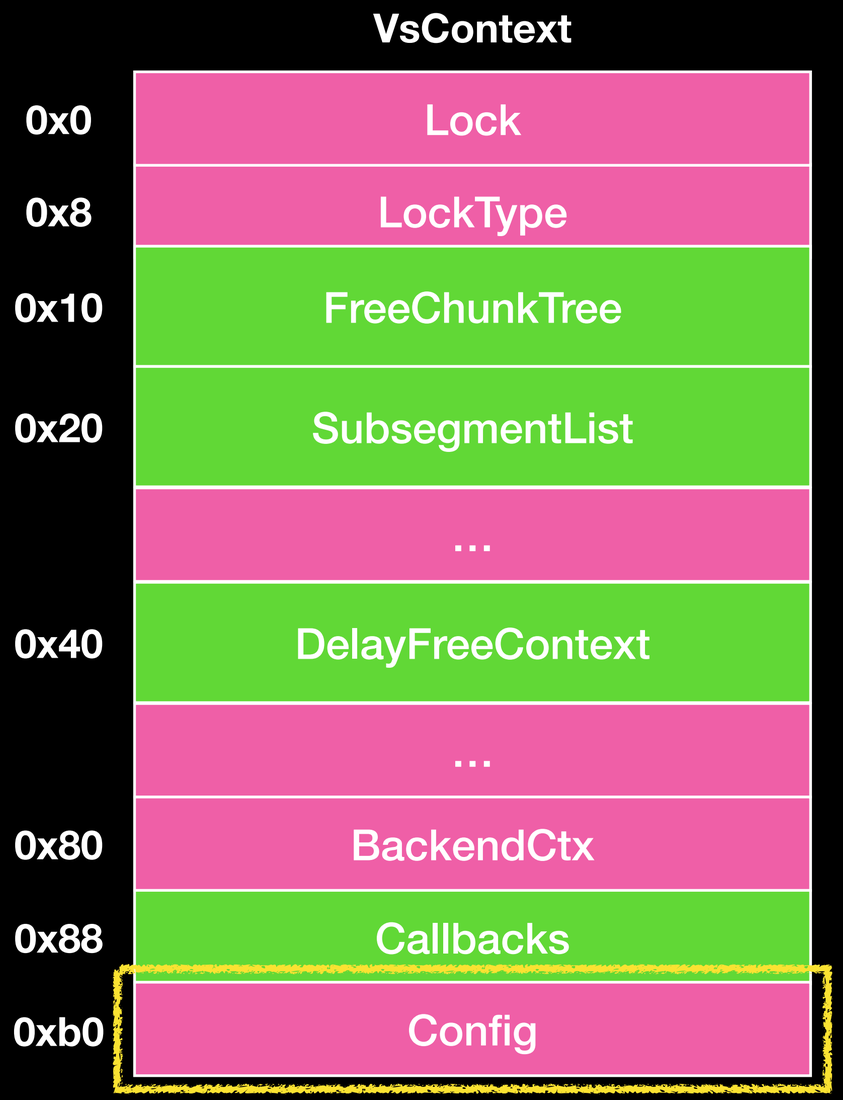
- VsContext (_HEAP_VS_CONTEXT)
  - Callbacks (_HEAP_SUBALLOCATOR_CALLBACKS)
    - Call back function table (Use to allocate/free subsegment)
      - Allocate
      - Free
      - Commit
      - Decommit
      - ExtendContext

Variable Size Allocation



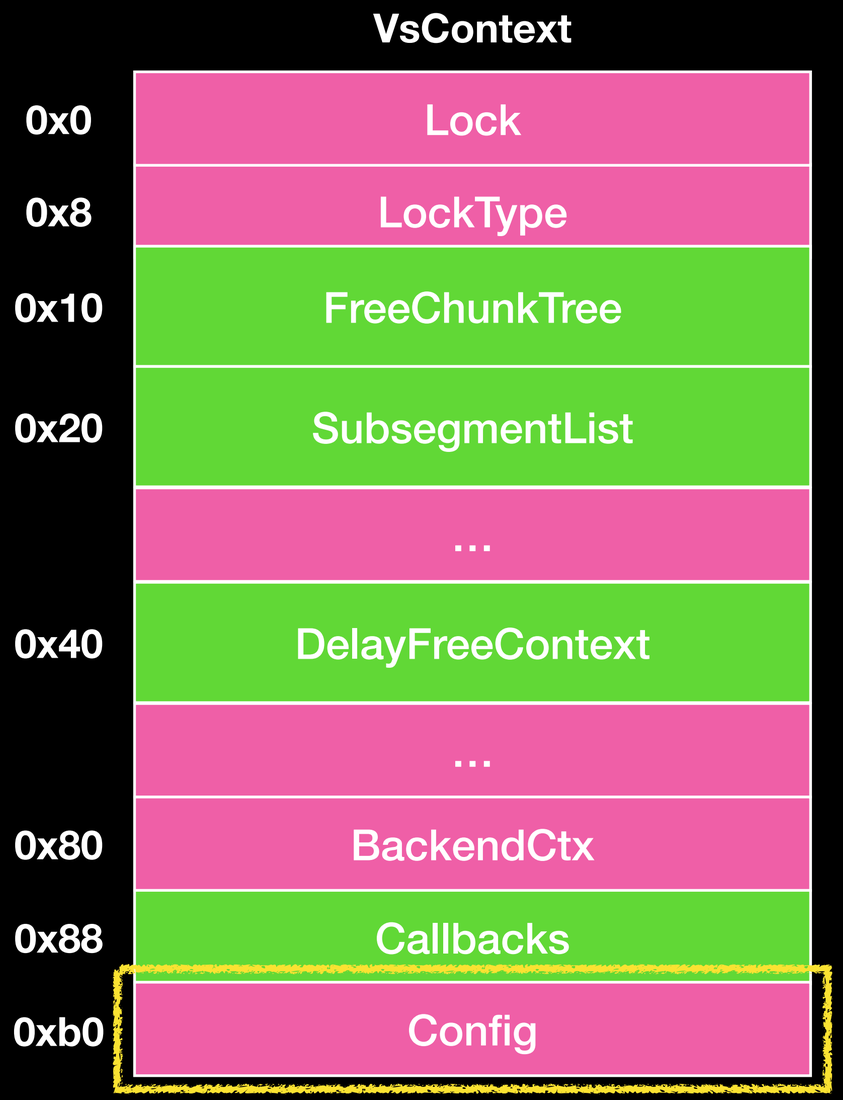
- VsContext (_HEAP_VS_CONTEXT)
  - Callbacks (_HEAP_SUBALLOCATOR_CALLBACKS)
    - Function pointer will be encoded
      - Callbacks will xor with following value
        - RtlpHpHeapGlobals.HeapKey
        - VsContext address
        - Function pointer

Variable Size Allocation



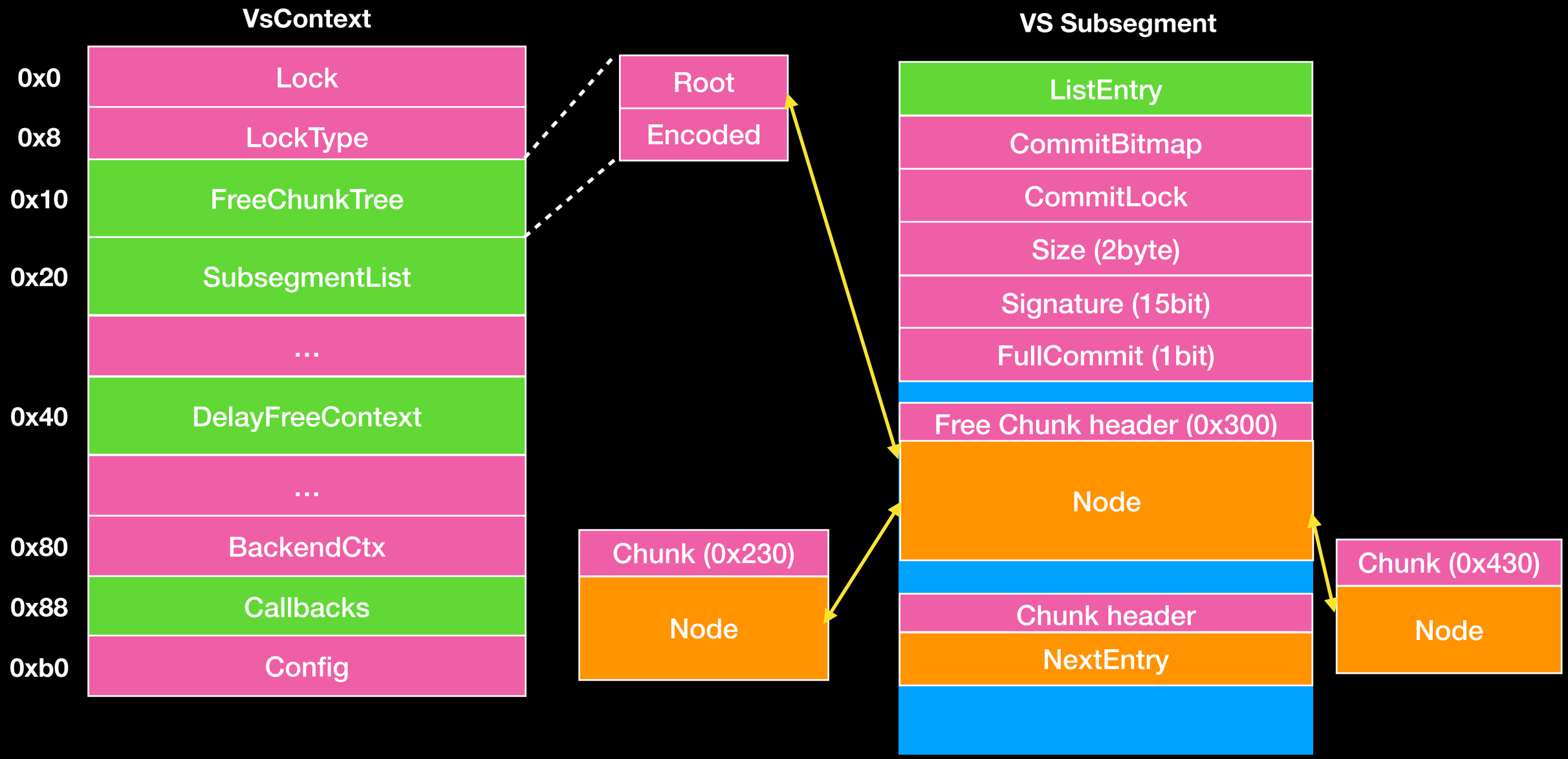
- VsContext (_HEAP_VS_CONTEXT)
  - Config (_RTL_HP_VS_CONFIG)
    - Used to represent the attributes of the Vs Allocator
      - PageAlignLargeAllocs (default 1 in kernel)
      - FullDecommit
      - EnableDelayFree (default 1 in kernel)

Variable Size Allocation

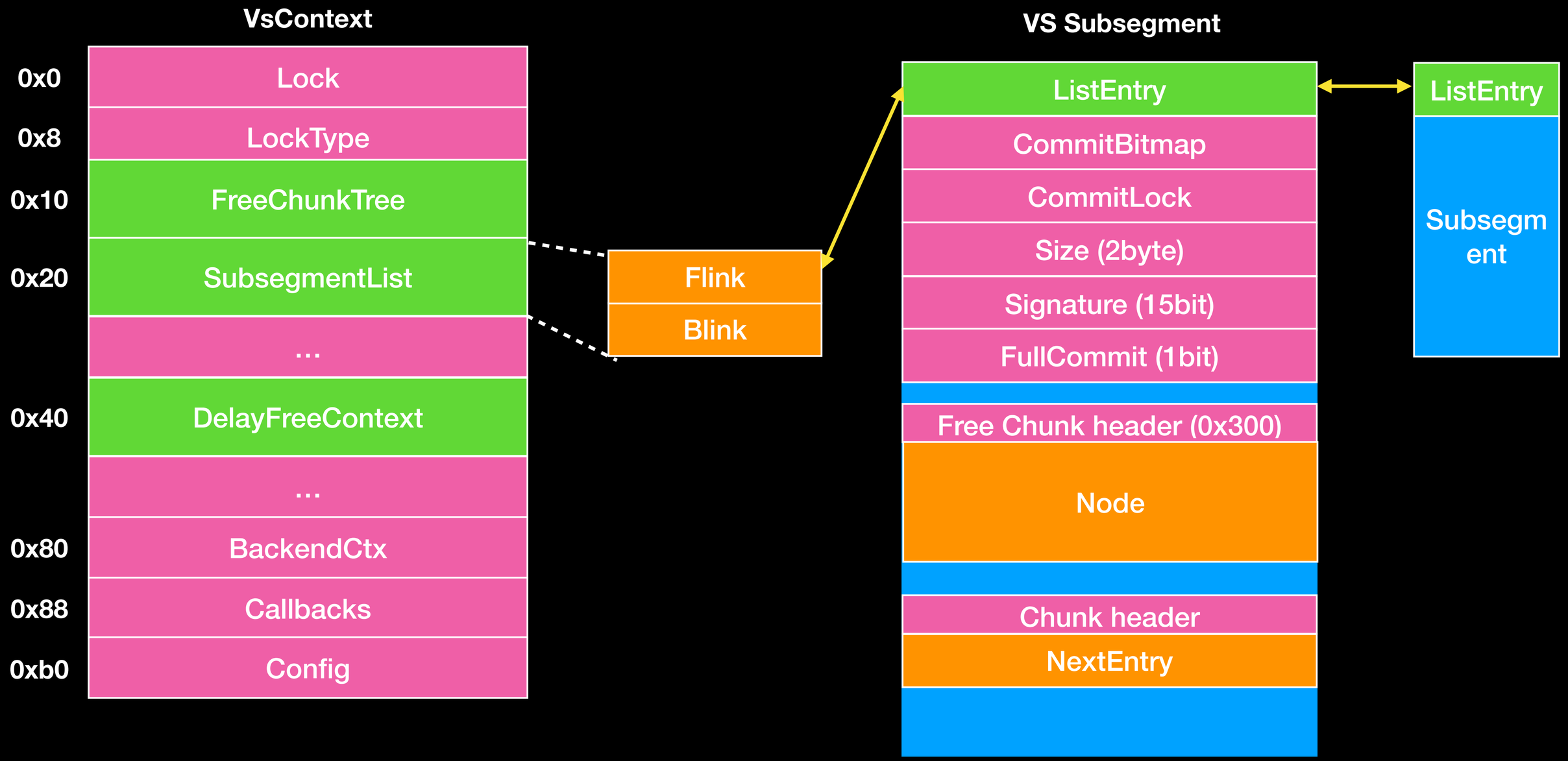


- VsContext (_HEAP_VS_CONTEXT)
  - Config (_RTL_HP_VS_CONFIG)
    - Used to represent the attributes of the Vs Allocator
      - PageAlignLargeAllocs (default 0 in user mode)
      - FullDecommit
      - EnableDelayFree (default 0 in usermode)

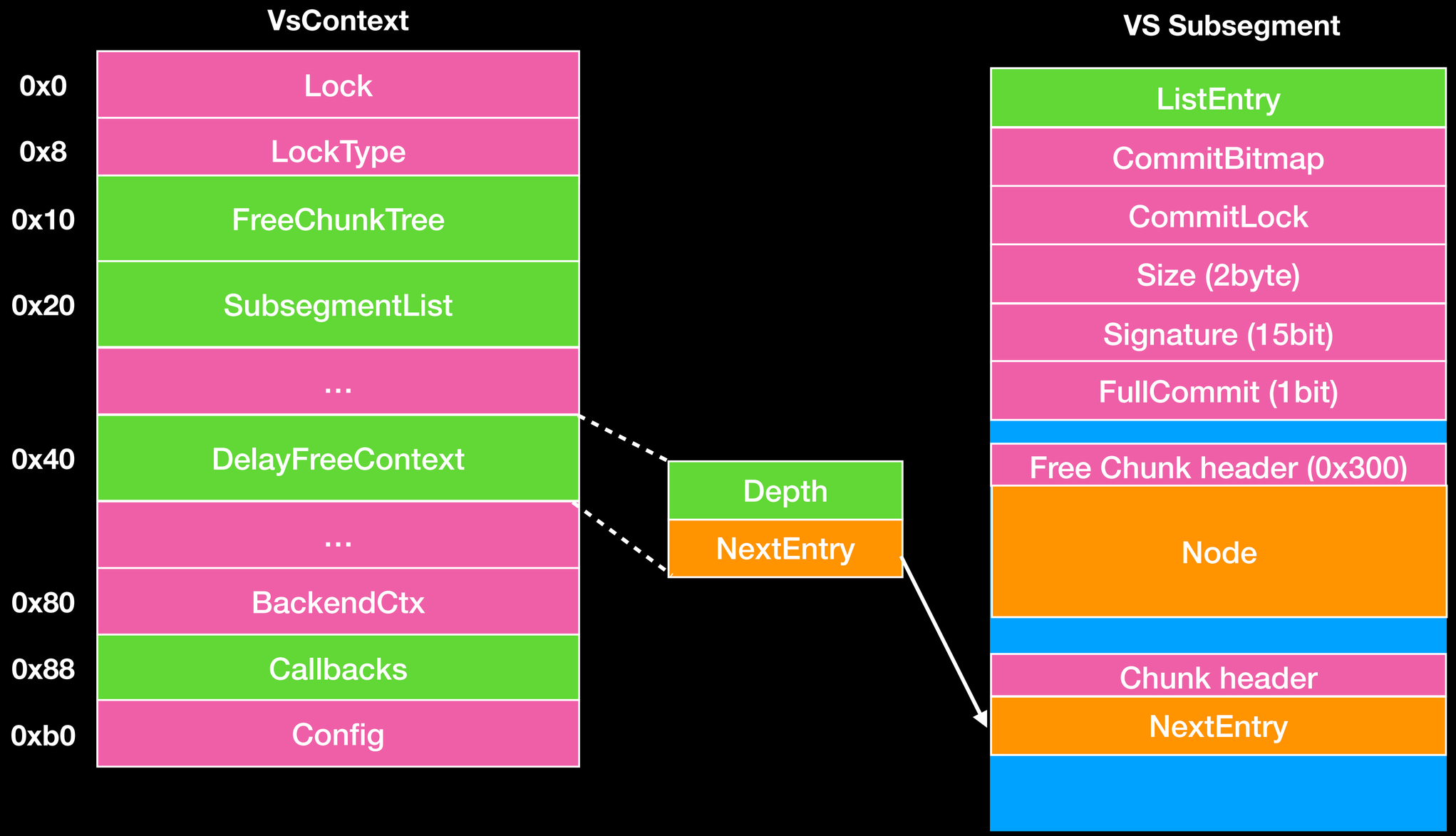
Variable Size Allocation



Variable Size Allocation



Variable Size Allocation



Variable Size Allocation

- Data Structure
- Memory allocation mechanism

Variable Size Allocation

- Allocate
  - Main implementation function is nt!RtlpHpVsContextAllocateInternal
    - It will calculate the required chunk size at the beginning
    - Then it will find a suitable chunk from FreeChunkTree in VsContext
      - Start searching from the root, when the required chunk is larger than the node, continue searching from the right subtree until it is found or is NULL

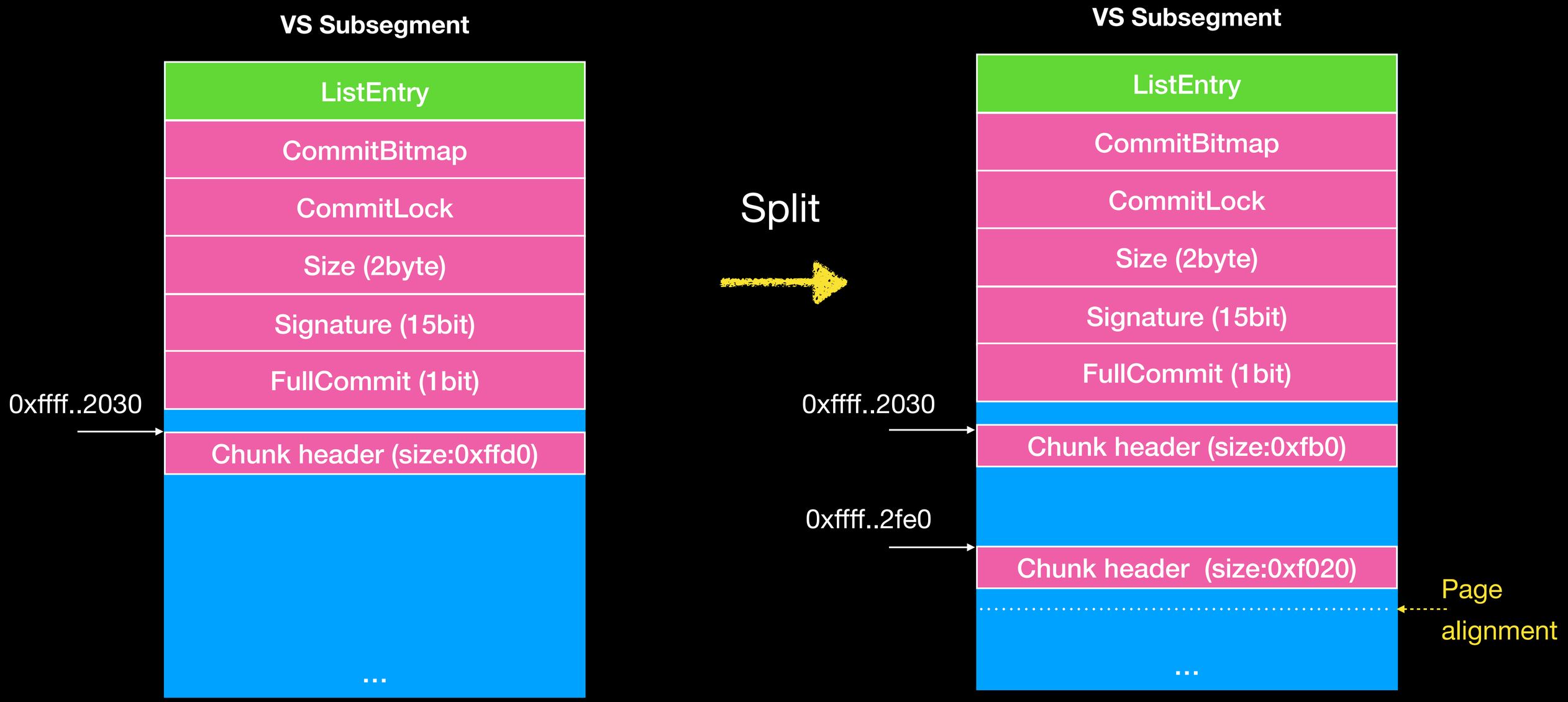
Variable Size Allocation

- Allocate
    - If the chunk that can be allocated is not found, a subsegment will be allocated and the subsegment will be added to the VsContext, and then start searching from FreeChunkTree again
        - It will create a large chunk when it initialize the subsegment.
        - RtlpHpVsSubsegmentCreate
            - Request memory (RtlpHpSegVsAllocate) from backend, and create a new subsegment, the minimum size is (0x10000)

Variable Size Allocation

- Allocate
    - RtlpHpVsContextAddSubsegment
        - Add subsegment to VsContext
            - It will determinate how to split the subsegment according to whether to enable PageAlignLargeAllocs
                - If PageAlignLargeAllocs is set, the subsegment will be split into two chunks, one is the first chunk behind the subsegment structure, and the second is the page alignment, the user data of the chunk is aligned, and both chunks are added to FreeChunkTree
                - If not, the entire subsegment will be treated as a large chunk and added to FreeChunkTree

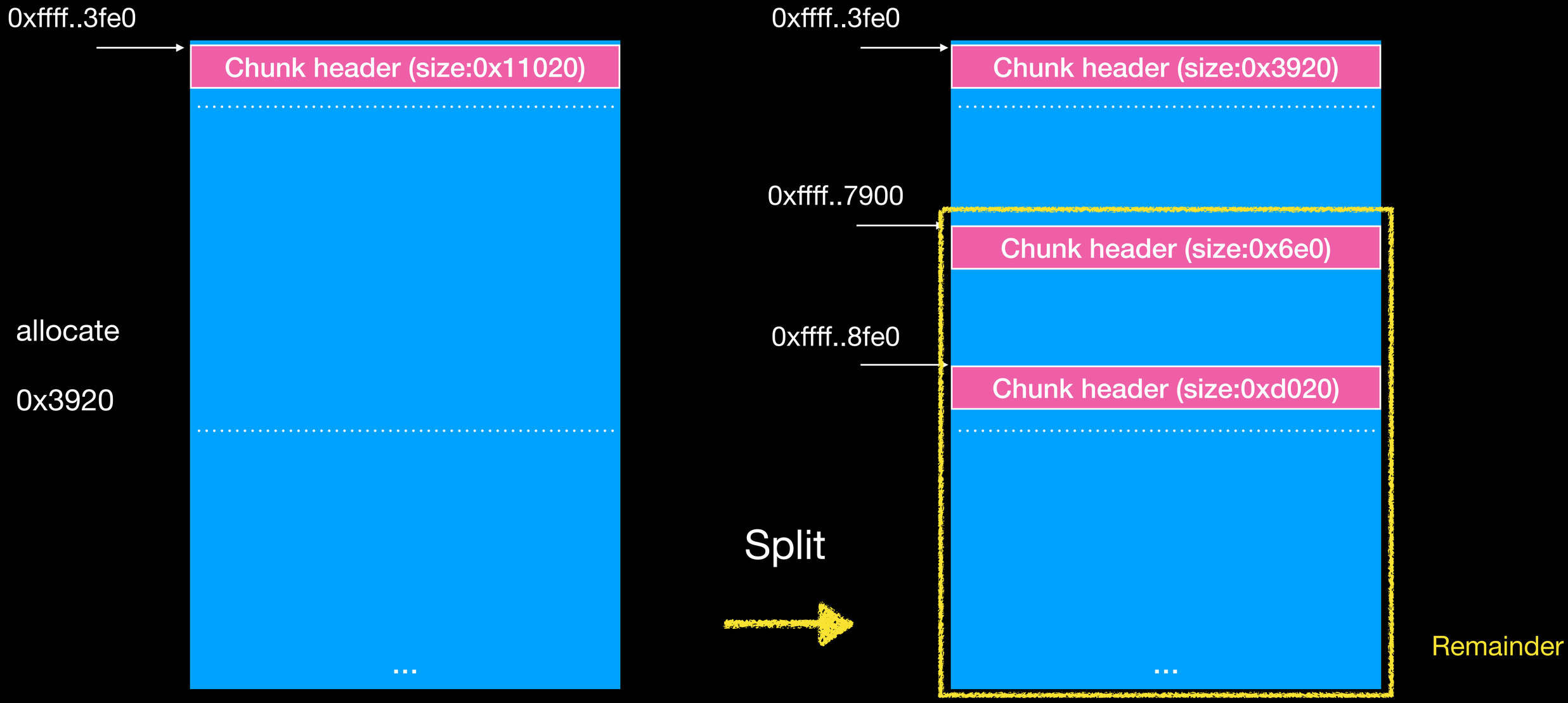
Variable Size Allocation



Variable Size Allocation

- Allocate
    - If the found size of chunk size larger then request size, the chunk will be split, and the remaining chunks will be re-added to FreeChunkTree
        - RtlpHpVsChunkSplit
            - It will remove the chunk out of FreeChunkTree, and split the chunk, and re-added re-added FreeChunkTree as a new Freed chunk.
            - This will also be split according to whether the page alignment is or not. If the size of the chunk to be split exceeds 1 page, it split remainder chunk into two pieces according to the page
        - If request size < chunk size, unused byte will be recorded at the last 2 bytes of chunk

Variable Size Allocation



Variable Size Allocation

- Free
  - Main implementation function is nt!RtlpHpVsContextFree
    - The signature and Allocated bytes of the subsegment will be verified at first
      - Subsegment->Size ^ Subsegment->Signature ^ 0x2BED == 0
      - BSOD if verification fails

Variable Size Allocation

- Free
  - Next, it will check if the number of chunks in VsContext->DelayFreeContext is greater than 0x20
    - VsContext->DelayFreeContext.Depth > 0x20
  - If it is less than 0x20, the chunk will be inserted at the beginning of the linked list and then return
    - VsContext->DelayFreeContext.NextEntry

Variable Size Allocation

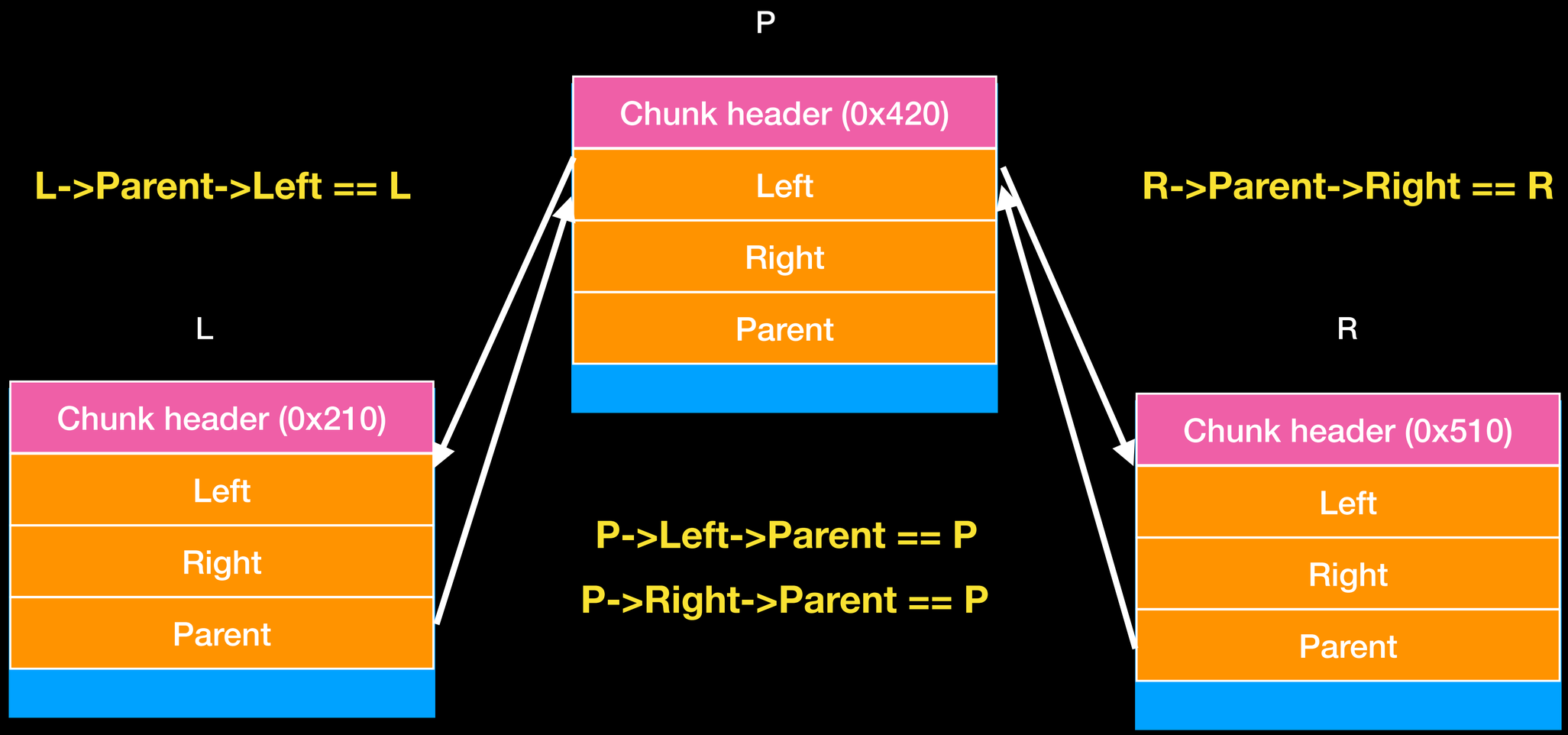
- Free
  - If DelayFreeContext> 0x20, then the chunks in the linked list will be freed one by one
    - When it free a chunk, EncodedSegmentPageOffset will be used to find the VS subsegment of the chunk, and verify the Allocated byte and segment signature

Variable Size Allocation

- Free
    - Next, it will check whether the front and next chunks can be merged. If VsContext->Config.flag has enable PageAlignLargeAllocs, then one more chunk will be check.
      The merged chunk will be moved out of FreeChunkTree first. There is a tree structure check. After the merge, update prev_size and Size
        - RtlpHpVsChunkCoalesce
    - After merging, if address of chunk+0x20 is the beginning of page, the chunk will be page aligned and split into two pieces

Variable Size Allocation

- FreeChunkTree check



Variable Size Allocation

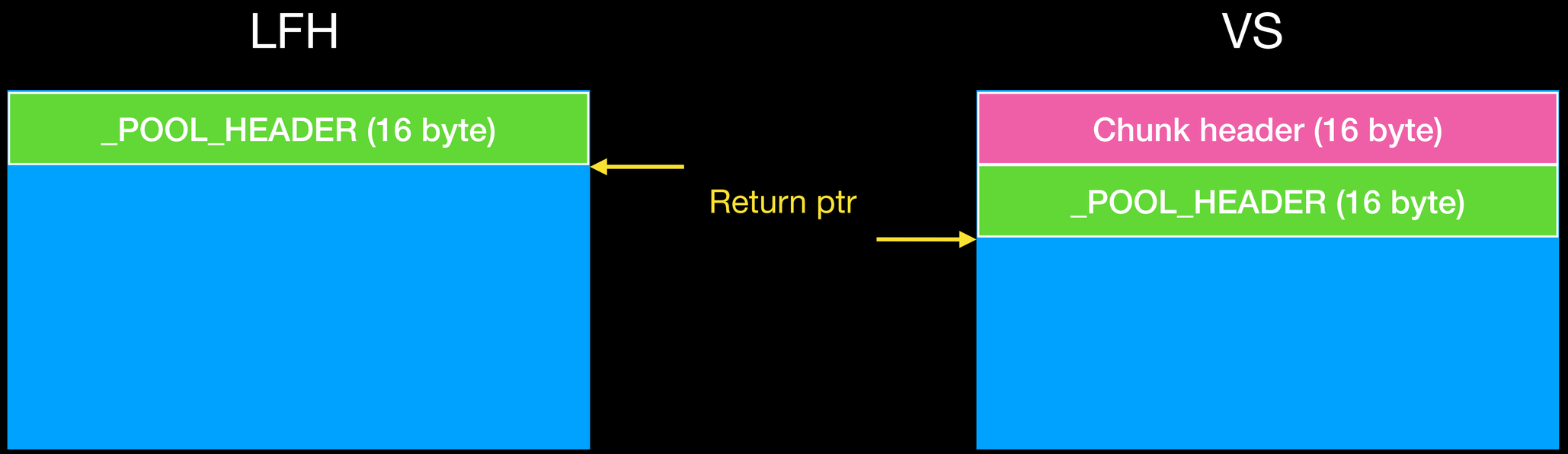
- Free
  - If the size of the merged chunk is exactly equal to the size of the subsegment, the entire subsegment will be moved out of the linked list.
  - There is also a double linked list check here and it will release the subsegment to the backend allocator
  - If it is not equal, it will calculate MemoryCost and Segmentpageoffset of the chunk and then encode it.
  - Finally, look for a suitable position in FreeChunkTree to insert

Pool Header

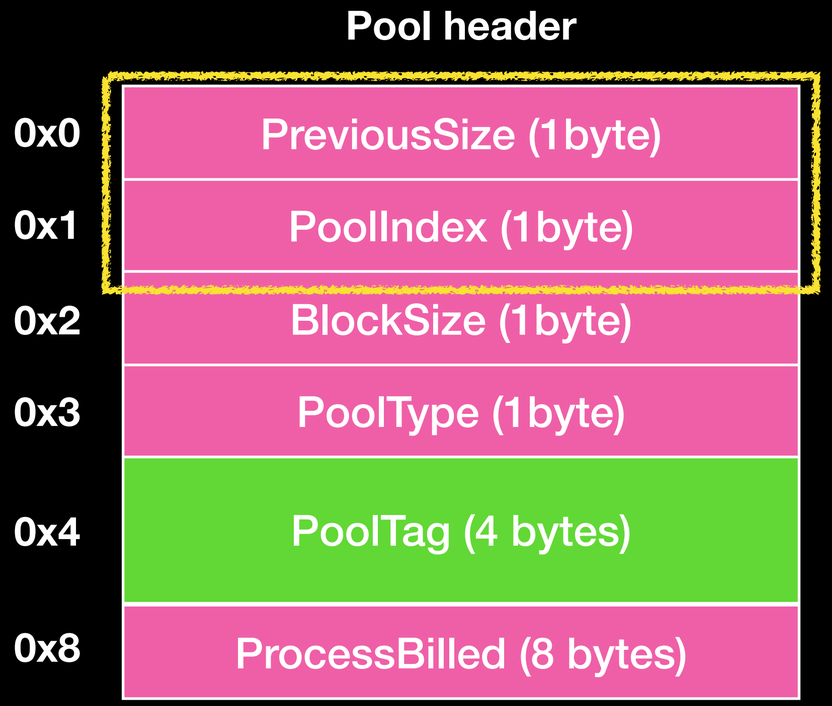
- When the size of allocation <= 0xfe0
  - Both LFH and VS Allocator will allocate additional 0x10 byte (64bit)
- It is used to store the Pool Header. Pool Header is used for Pool Allocator before 19H1 .
- Now it is basically used in a few cases such as CacheAligned, PoolQuota and PoolTrack
  - The header contains
    - Size
    - PreviousSize
    - Pool type
    - PoolIndex

Pool Header

- When the size of allocation <= 0xfe0

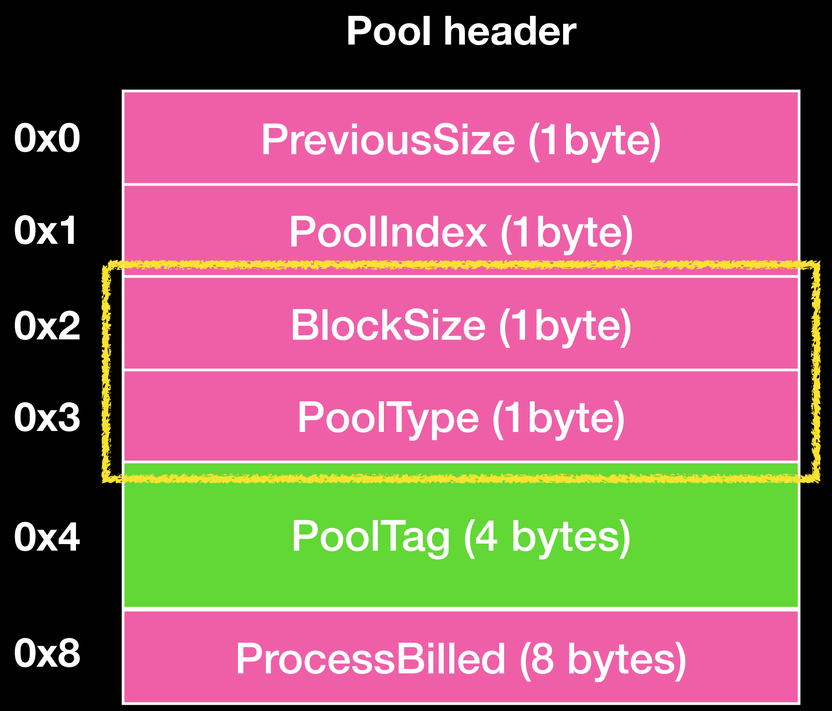


Pool Header



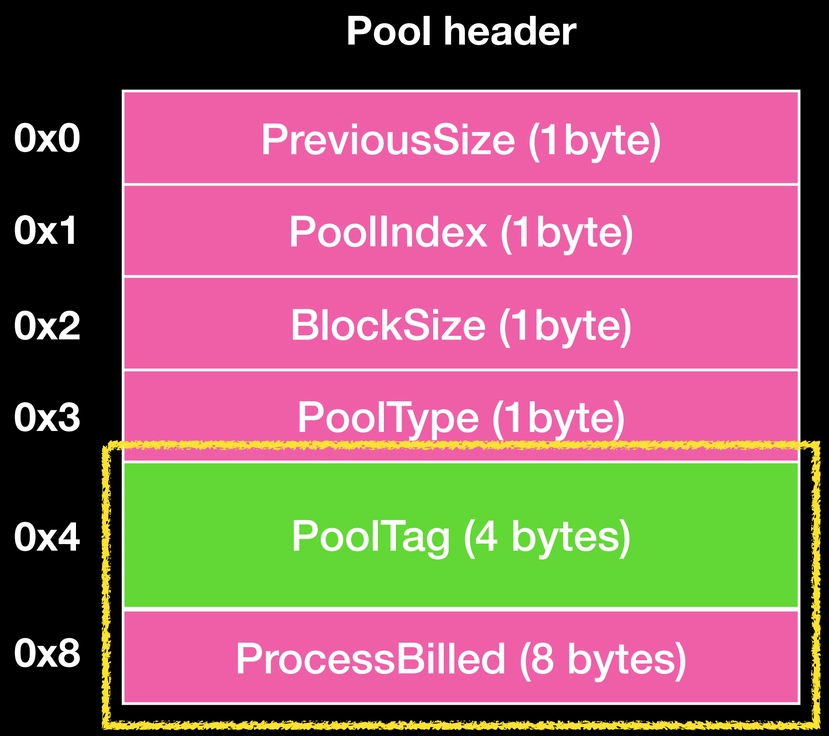
- Pool Header (_POOL_HEADER)
  - PreviousSize
    - Used in the CacheAligned case, indicating the offset between the previous Pool header and the header
  - Pool Index
    - Useless in Segment Heap

Pool Header



- Pool Header (_POOL_HEADER)
  - BlockSize
    - The size of the block
    - The value is right shift by 4 bits
  - PoolType
    - The pooltype of the block

Pool Header

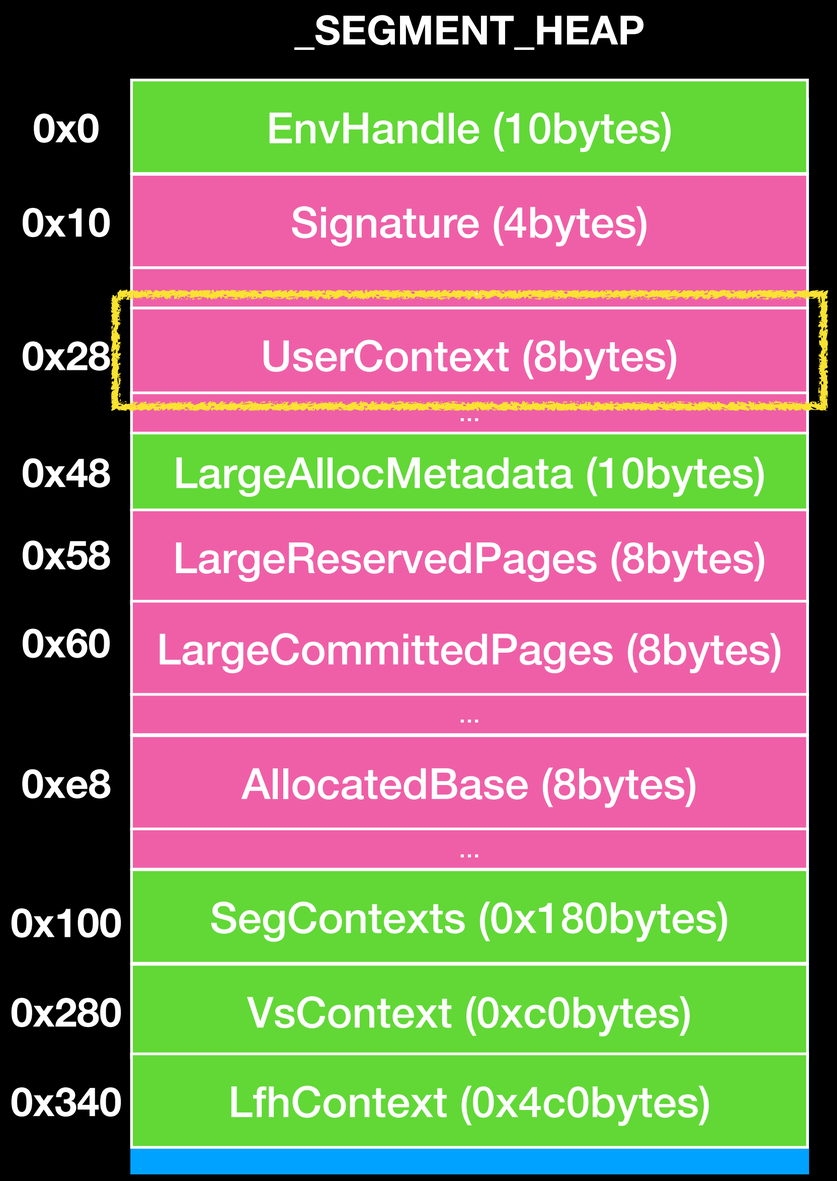


- Pool Header (_POOL_HEADER)
  - PoolTag
    - The tag string is filled in when the block is allocated
    - When you use ExAllocatePoolWithTag, you can specify the pooltag
  - ProcessBilled
    - Used for PoolQuota and CacheAligned

Dynamic Lookaside

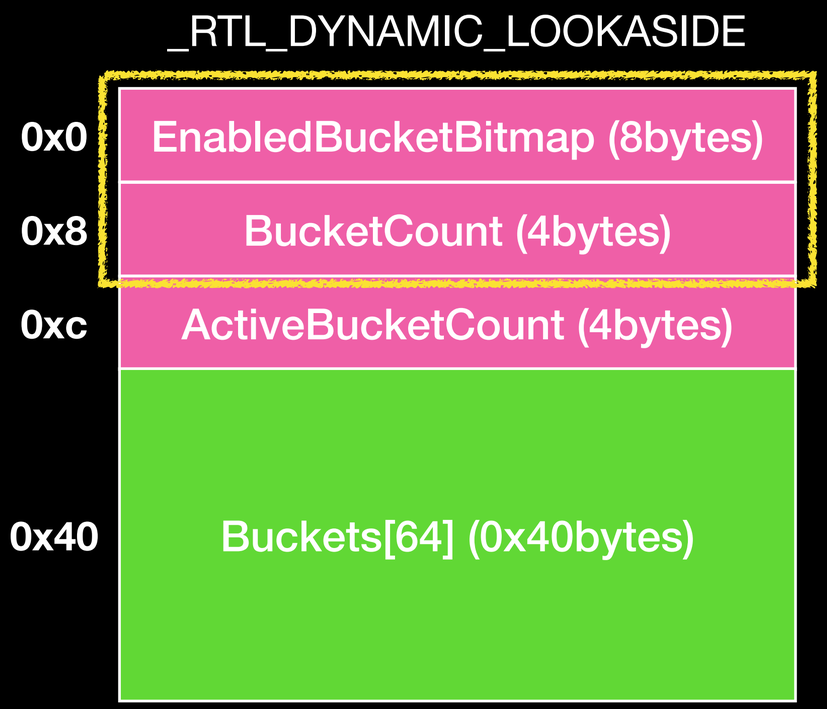
- If the size of allocation 0x201 <Size <0xfe0.
  - It will not be freed immediately after free. The block will be added to a Dynamic Lookaside
  - Depending on the size of the block, it will be added to different linked lists
  - The size is taken from Pool Header
- When it allocate, it will first take block from Dynamic Lookaside first
  - It will enter the normal allocation if it is not found

Dynamic Lookaside



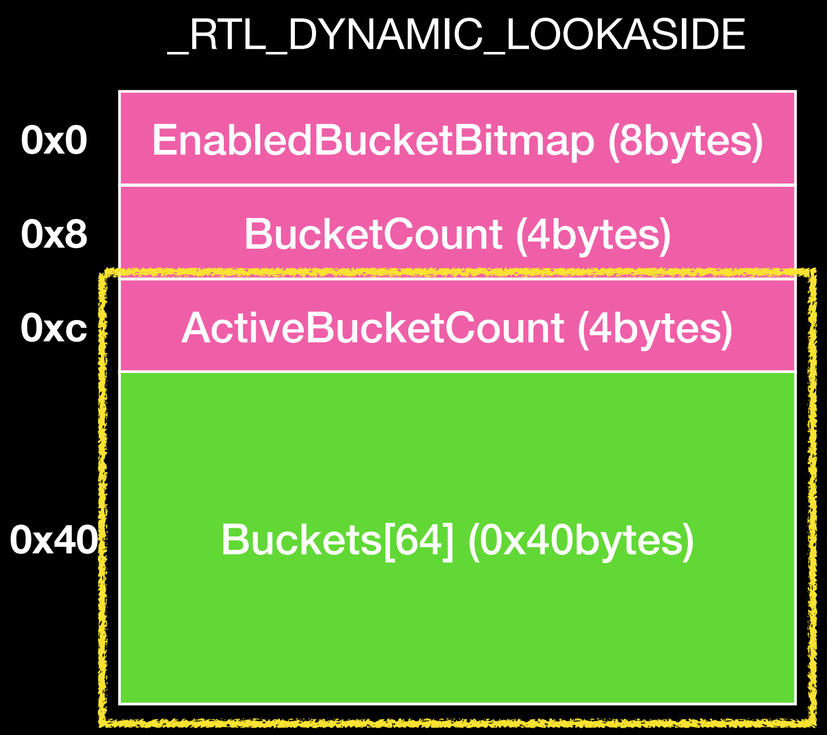
- _SEGMENT_HEAP
  - UserContext (_RTL_DYNAMIC_LOOKASIDE)
    - The core structure of the dynamic lookaside

Dynamic Lookaside



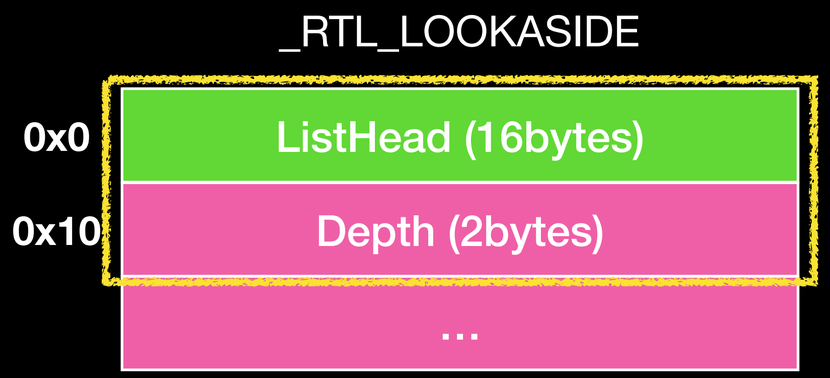
- UserContext (_RTL_DYNAMIC_LOOKASIDE)
  - EnabledBucketBitmap
    - A bitmap is used to indicate which buckets have enable lookaside
  - BucketCount
    - The total number of buckets in lookaside

Dynamic Lookaside



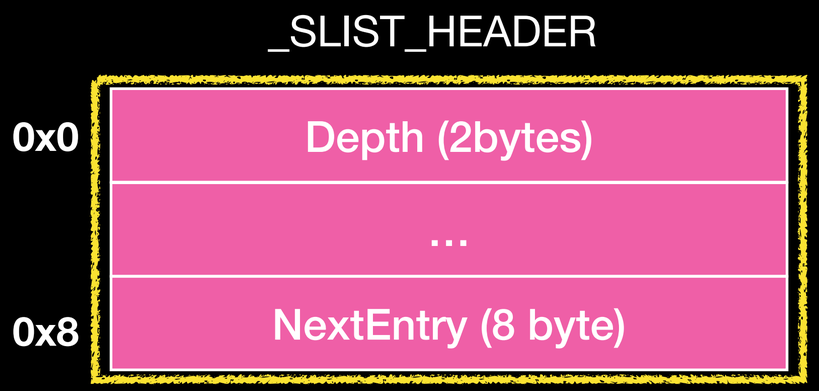
- UserContext (_RTL_DYNAMIC_LOOKASIDE)
  - ActiveBucketCount
    - The number of buckets with enable lookaside
  - Buckets (_RTL_LOOKASIDE)
    - Used to manage the structure of different sizes of lookaside, which is where the lookaside are

Dynamic Lookaside



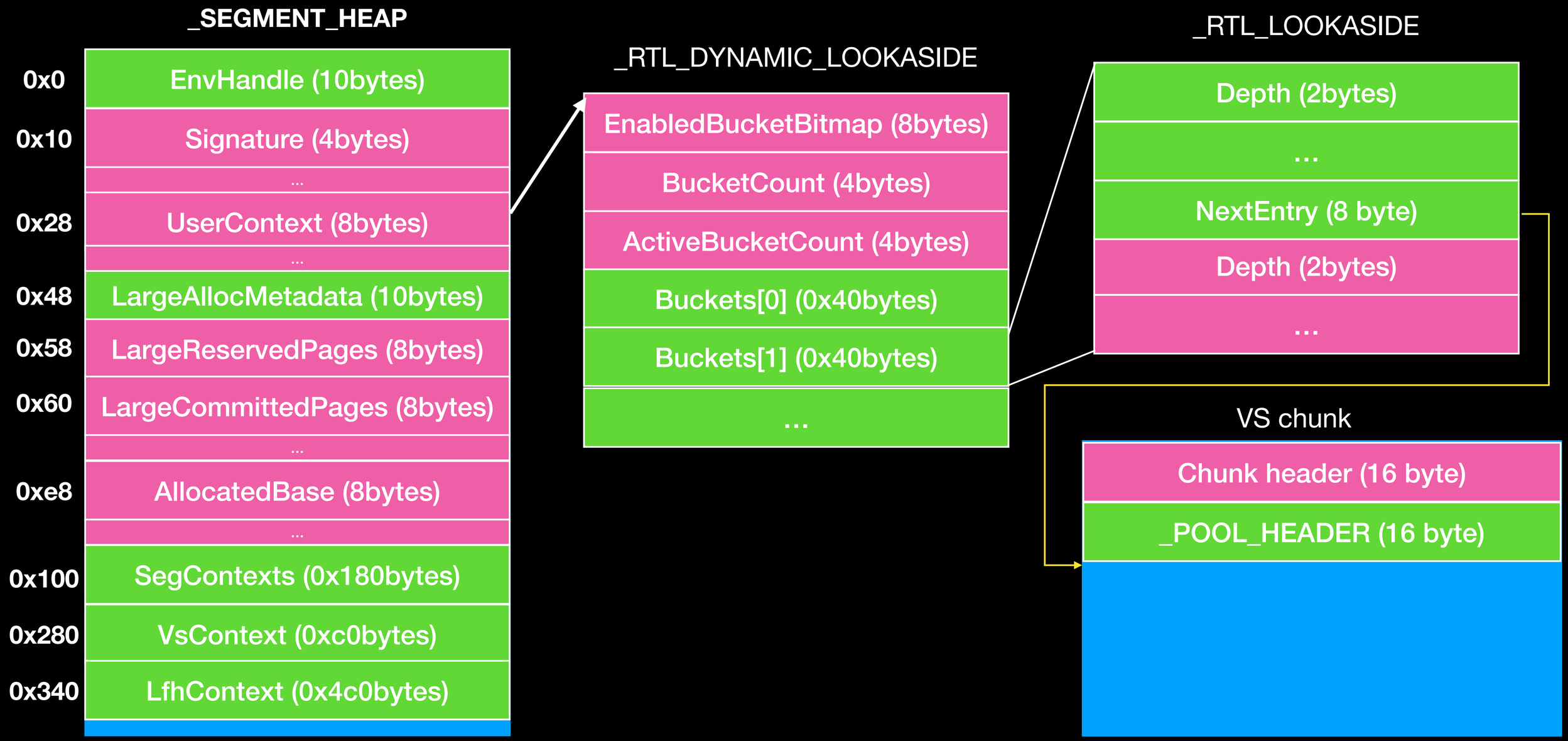
- Buckets (_RTL_LOOKASIDE)
  - ListHead (_SLIST_HEADER)
    - The structure of the head of the Singly linked list contains the length of the linked list, and the linked list itself, which is common in windows kernel
  - Depth
    - The number of chunks that can be stored in the bucket

Dynamic Lookaside



- _SLIST_HEADER
  - Depth
    - The number of nodes in the linked list
  - NextEntry
    - Point to next node
    - In the Dynamic Lookaside, it points to the Userdata of the freed chunk, behind the pool header

Dynamic Lookaside



Memory Allocator in kernel

- Segment Heap
  - Frontend Allocation
    - Low FragmentationHeap
    - Variable Size Allocation
  - Backend Allocation
    - Segment Allocation
  - Large Block Allocation

Segment Allocation

- Size (in the kernel)
  - Size is a multiple of page
    - Size <= 0x7f0000
  - Size is not a multiple of page
    - 0x20000 < Size <= 0x7f0000
- It’s a bit different in userland.

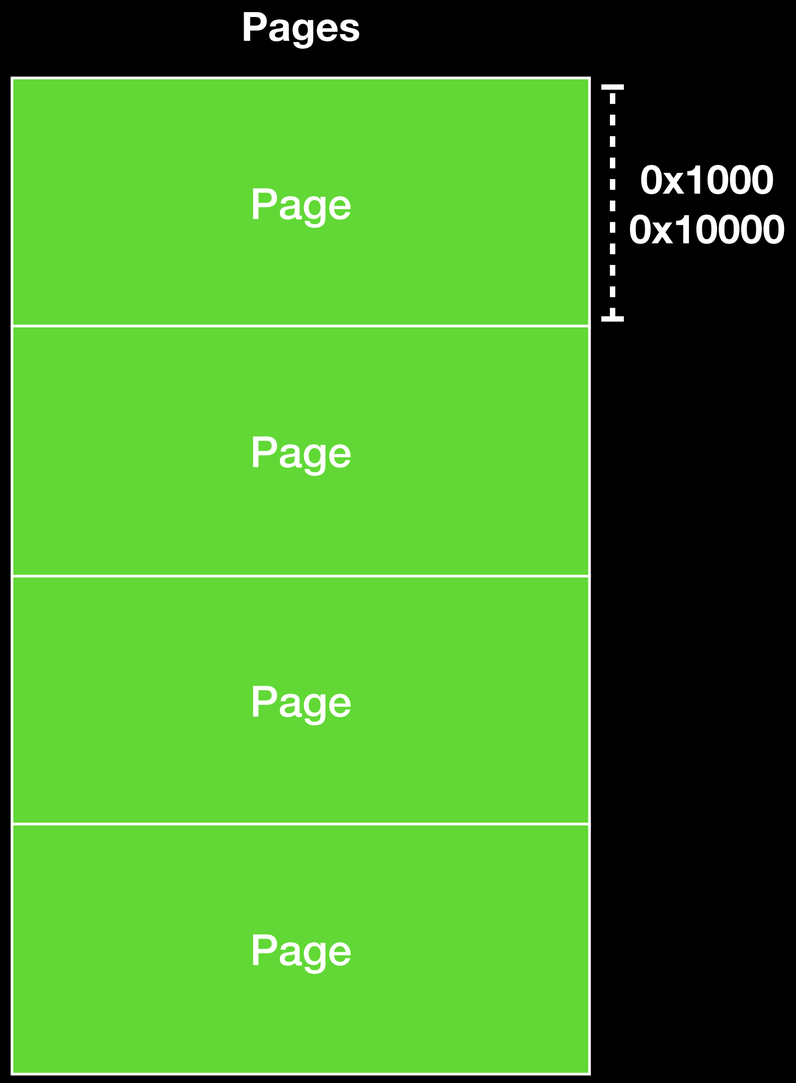
Segment Allocation

- it can be divided into two categories. The difference between the two is that the block units are different.
  - 0x20000 < Size <= 0x7f000
    - 0x1000 as the unit of block
  - 0x7f000 < Size <= 0x7f0000
    - 0x10000 as the unit of block
  - Both unit are called page here

Segment Allocation

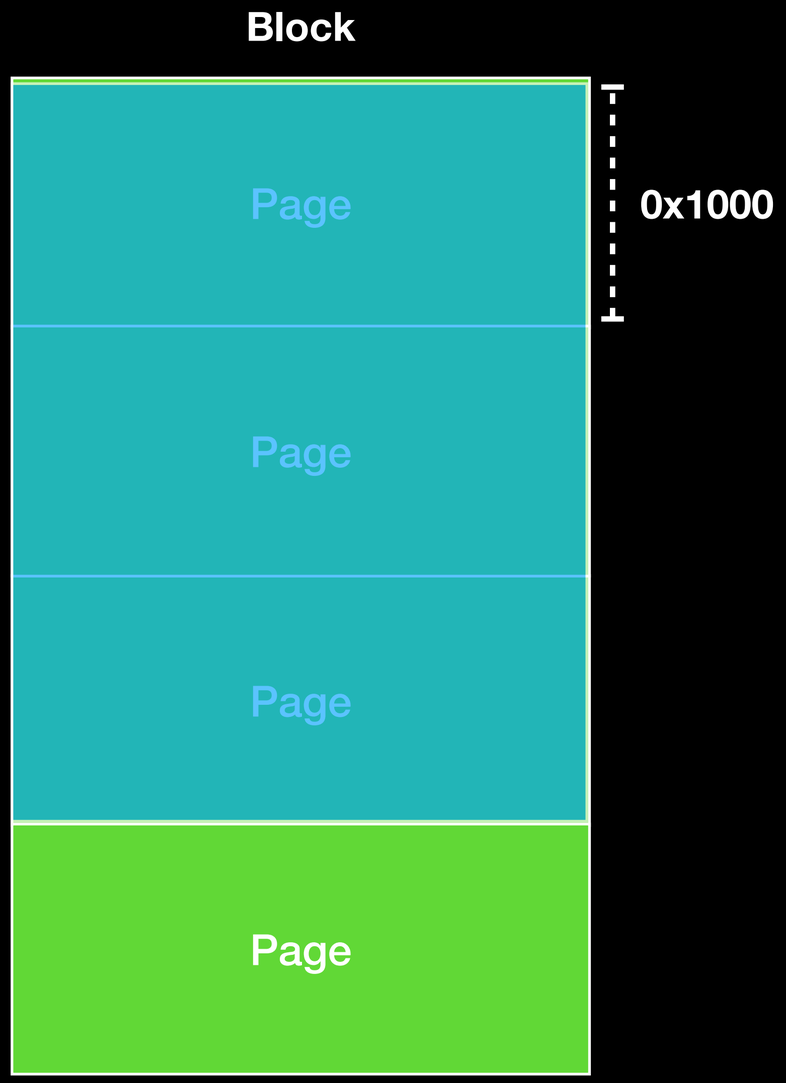
- Data Structure
- Memory allocation mechanism

Segment Allocation



- Page
  - In the case of size <= 0x7f000, the allocation unit of Segment Allocation
    - For example, if you allocate 0x1450 bytes, it would allocate 2 page
  - 0x7f000 < Size <= 0x7f0000
    - It will use 0x10000 as the allocation unit
  - In the following slides, I will use 0x1000 as an example

Segment Allocation

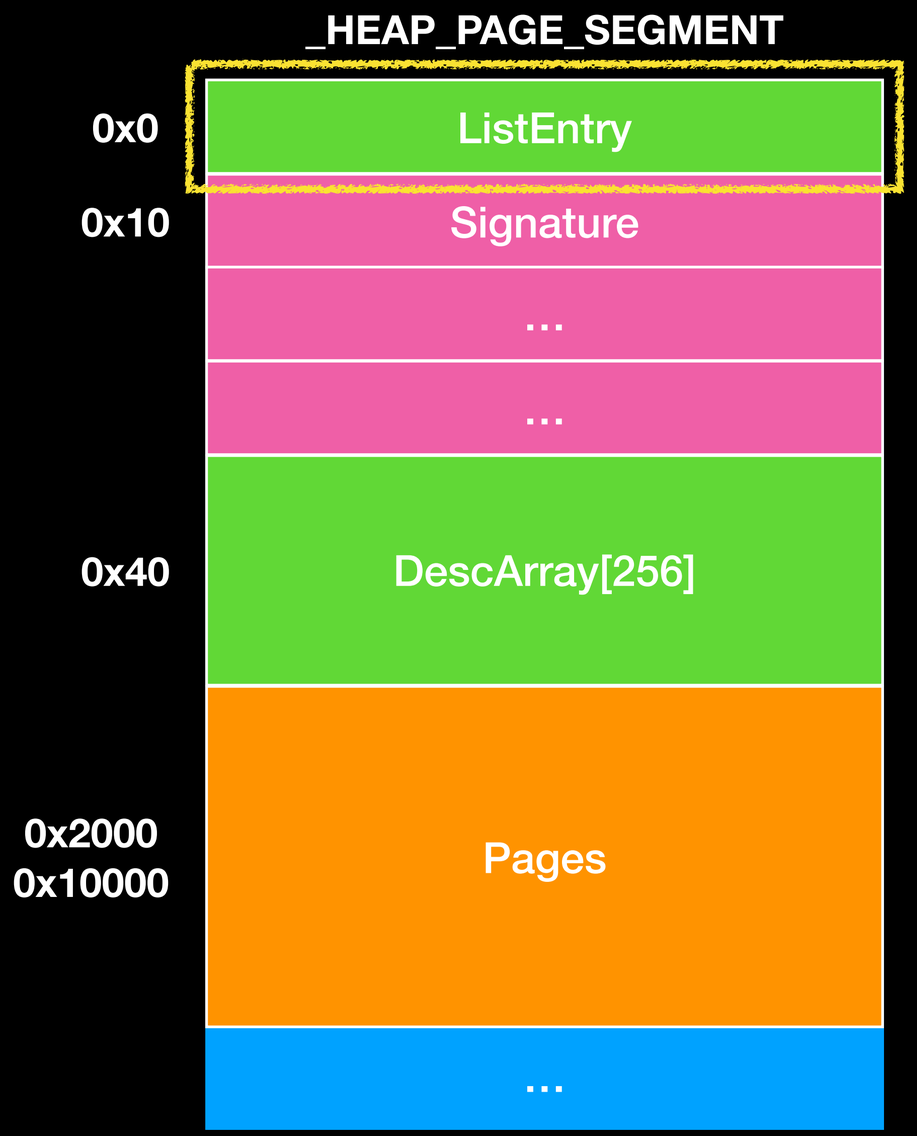


- Block (chunk)
  - Consists of single or multiple pages
  - The memory block allocated by Segment Allocatior
    - As shown in the figure, the block is composed of 3 pages

Segment Allocation

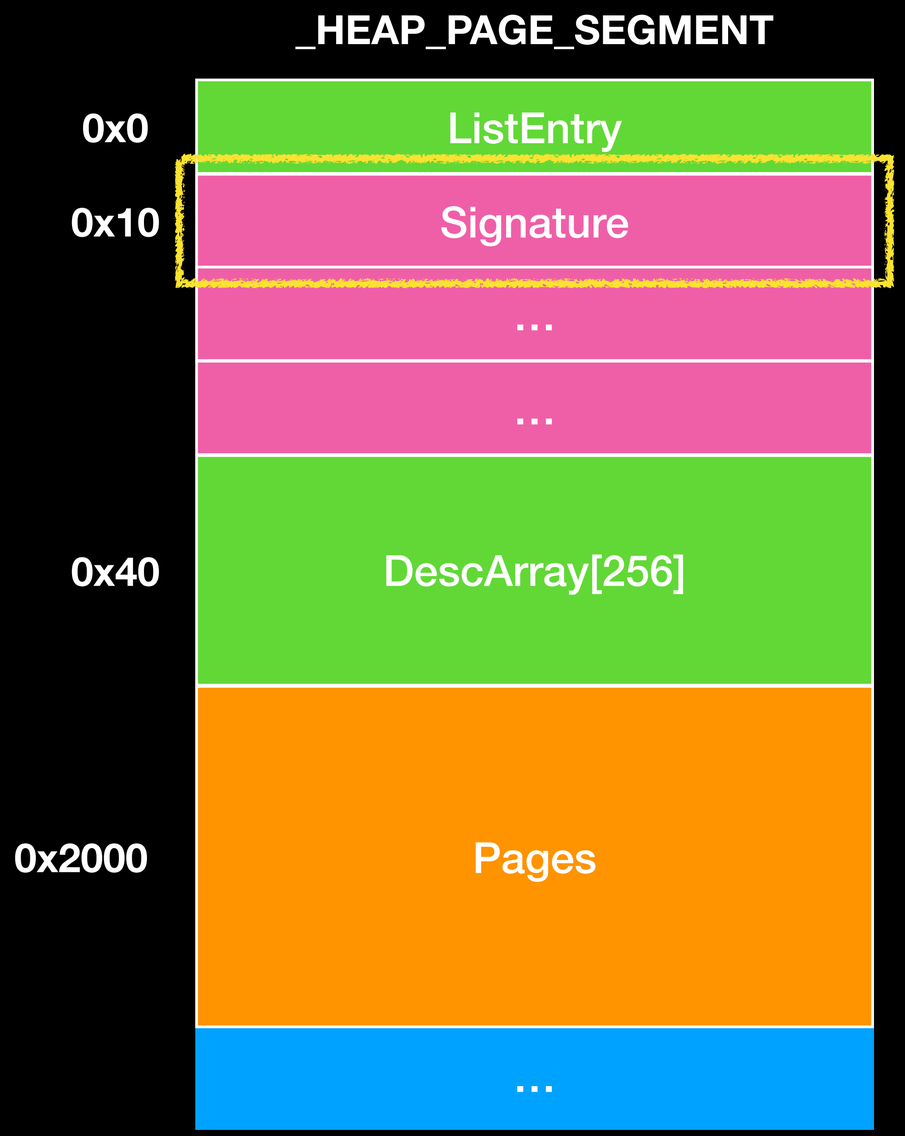
- Page Segment (_HEAP_PAGE_SEGMENT)
  - memory pool of segment allocation
  - Once there is no available page segment, a new page segment will be allocated from the system(MmAllocatePoolMemory) , but only the required structure will be allocated at the beginning
  - Each page segment will be inserted into a double linked list

Segment Allocation



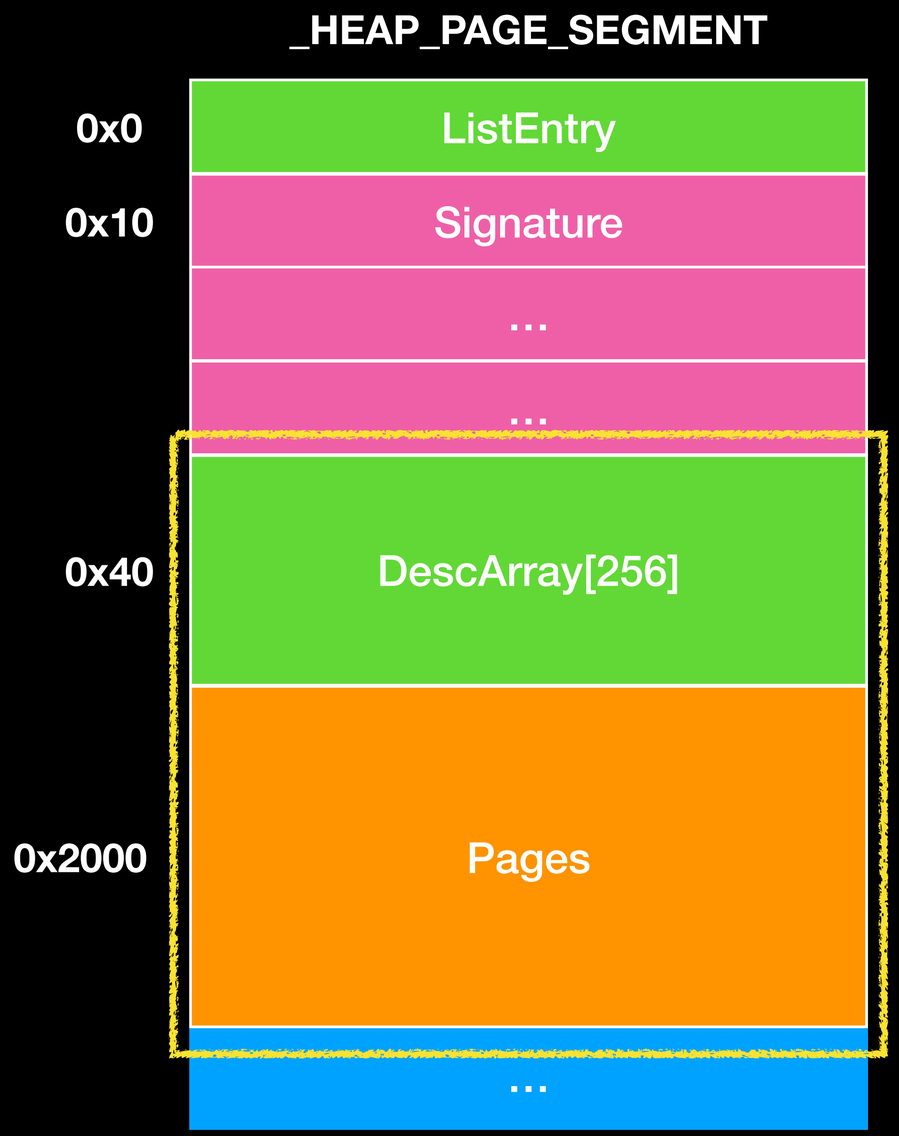
- _HEAP_PAGE_SEGMENT
  - ListEntry (_LIST_ENTRY)
    - Flink
      - Point to the next page segment in the Linked list
    - Blink
      - Point to the previous page segment in the Linked list

Segment Allocation



- _HEAP_PAGE_SEGMENT
  - Signature
    - The signature of the page segment is used to verify the page segment
    - Signature will xor with following value
      - Address of page segment
      - Address of Segcontext
      - RtlpHpHeapGlobals
      - 0xA2E64EADA2E64EAD

Segment Allocation

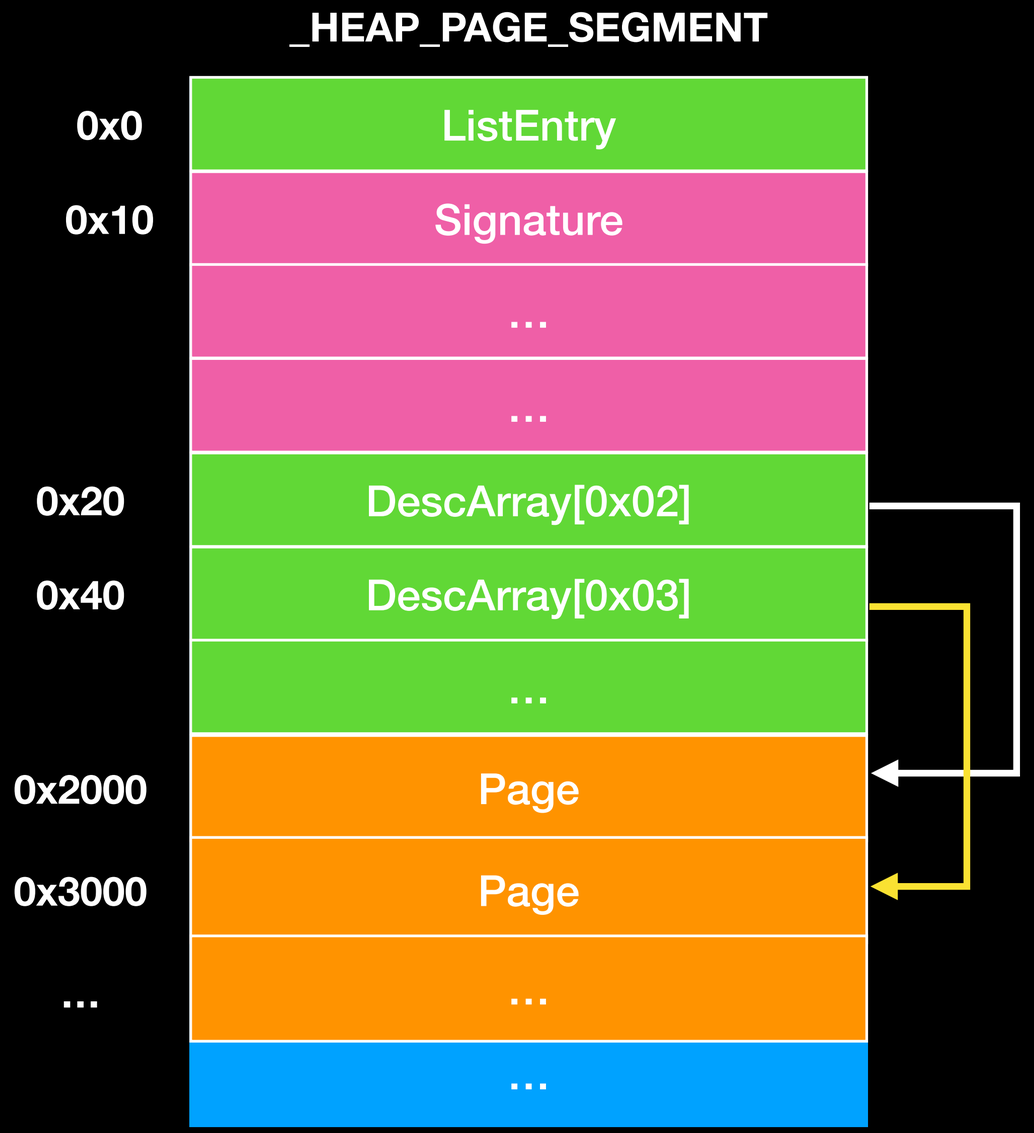


- _HEAP_PAGE_SEGMENT
  - DescArray (_HEAP_PAGE_RANGE_DESCRIPTOR)
    - Page range descriptor array
    - Each element corresponds to each page one by one
  - Pages
    - Memory pool of segment allocator

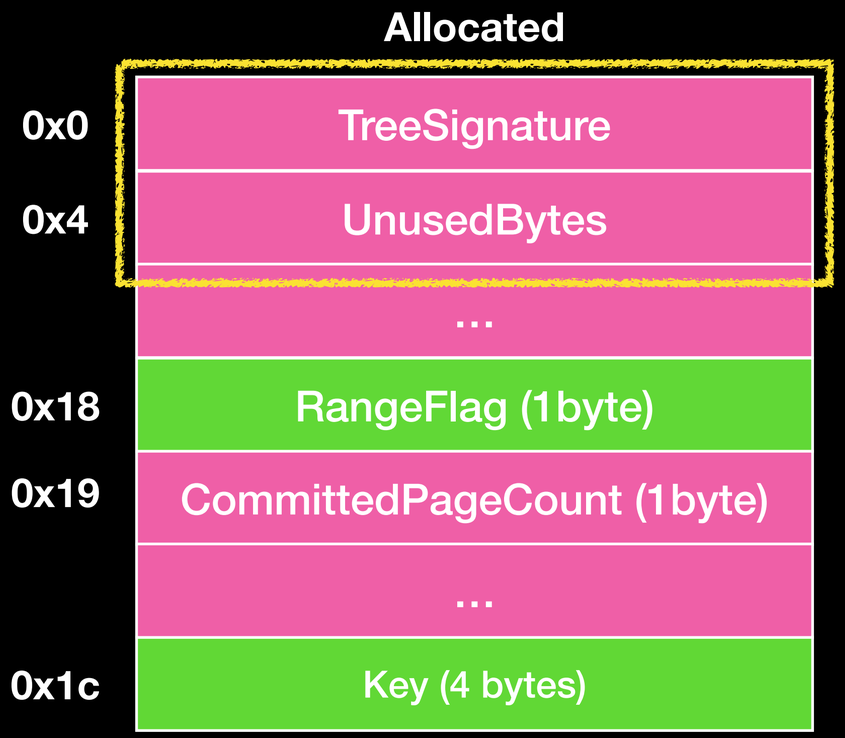
Segment Allocation

- Page range descriptor (_HEAP_PAGE_RANGE_DESCRIPTOR)
  - Descriptor for page
  - Indicates the status (Allocated or Freed) and information of each page in the page segment (whether the page is the beginning of a block, size of block, etc.)
  - It can be divided into allocated and freed
  - The page range descriptor in the freed state will be stored in FreePageRanges which is a rbtree structure

Segment Allocation

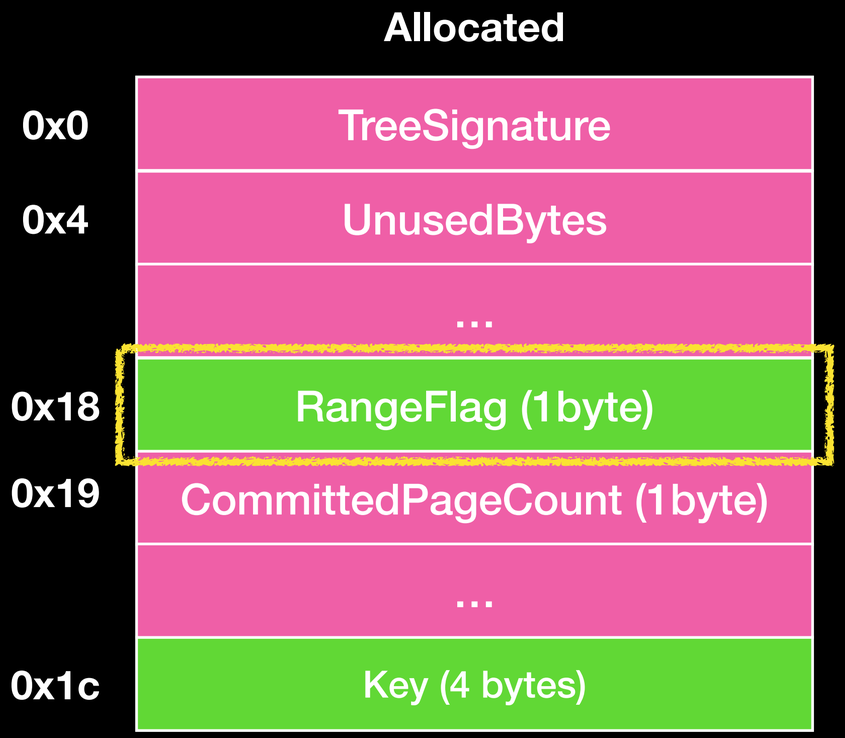


Segment Allocation



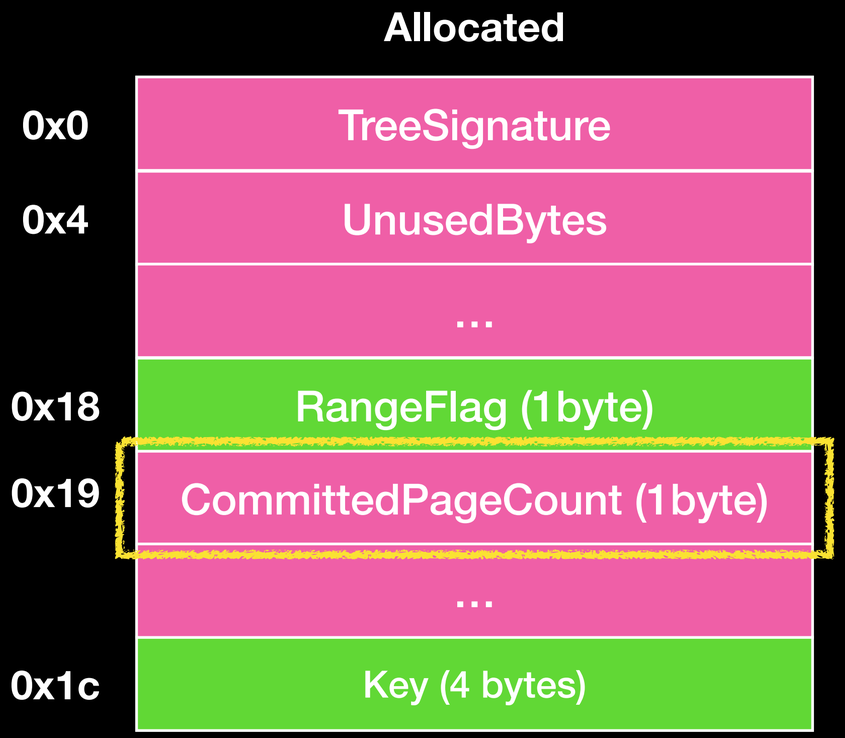
- _HEAP_PAGE_RANGE_DESCRIPTOR
  - TreeSignature
    - Signature of page range descriptor
    - The values is always 0xccddccdd
    - Only at the beginning of Block
  - UnusedBytes
    - Unusedbytes in an allocated block

Segment Allocation



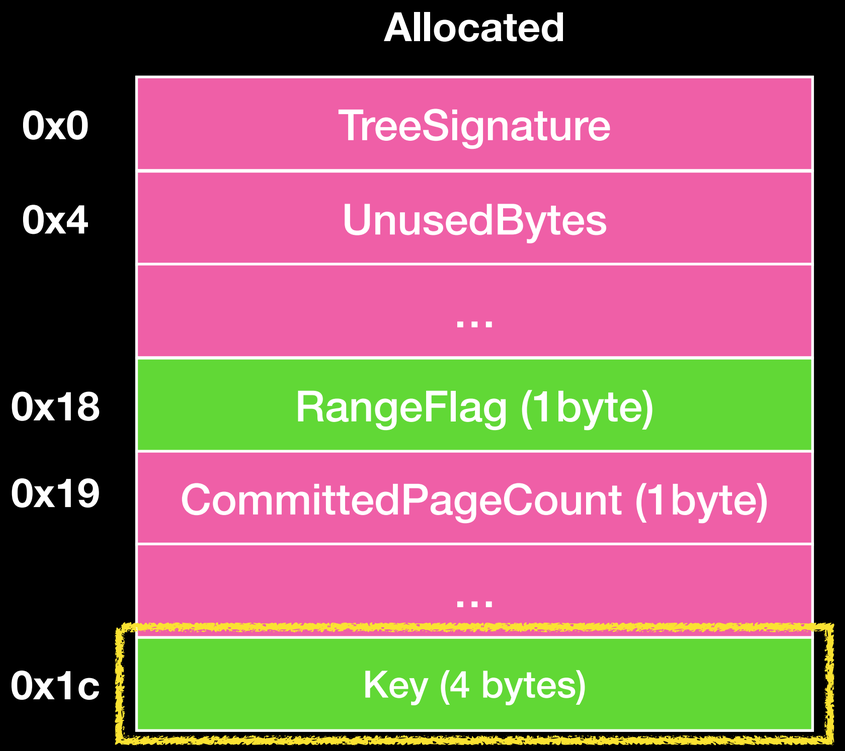
- _HEAP_PAGE_RANGE_DESCRIPTOR
  - RangeFlag
    - Used to indicate the page status
    - Bit 1 : allocated bit
    - Bit 2 : block header bit
    - Bit 3 : Committed
      - LFH : RangeFlag & 0xc = 8
      - VS : RangeFlag & 0xc = 0xc

Segment Allocation



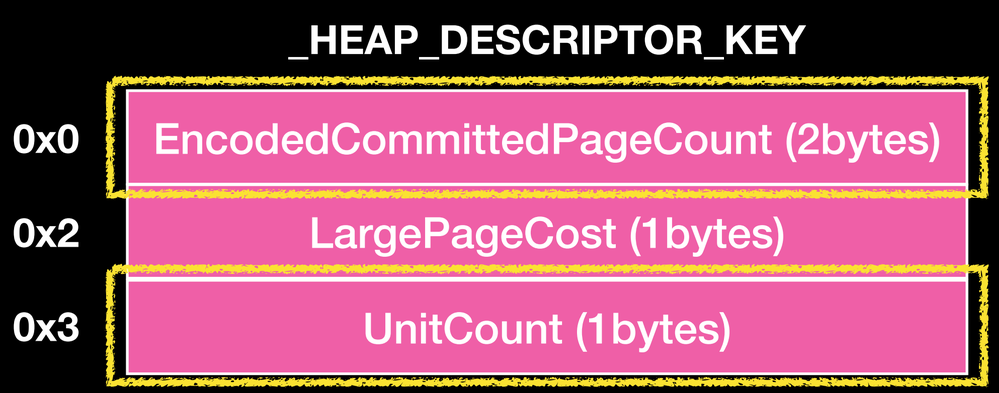
- _HEAP_PAGE_RANGE_DESCRIPTOR
  - CommitedPageCount
    - Indicates the number of pages committed in the corresponding page

Segment Allocation



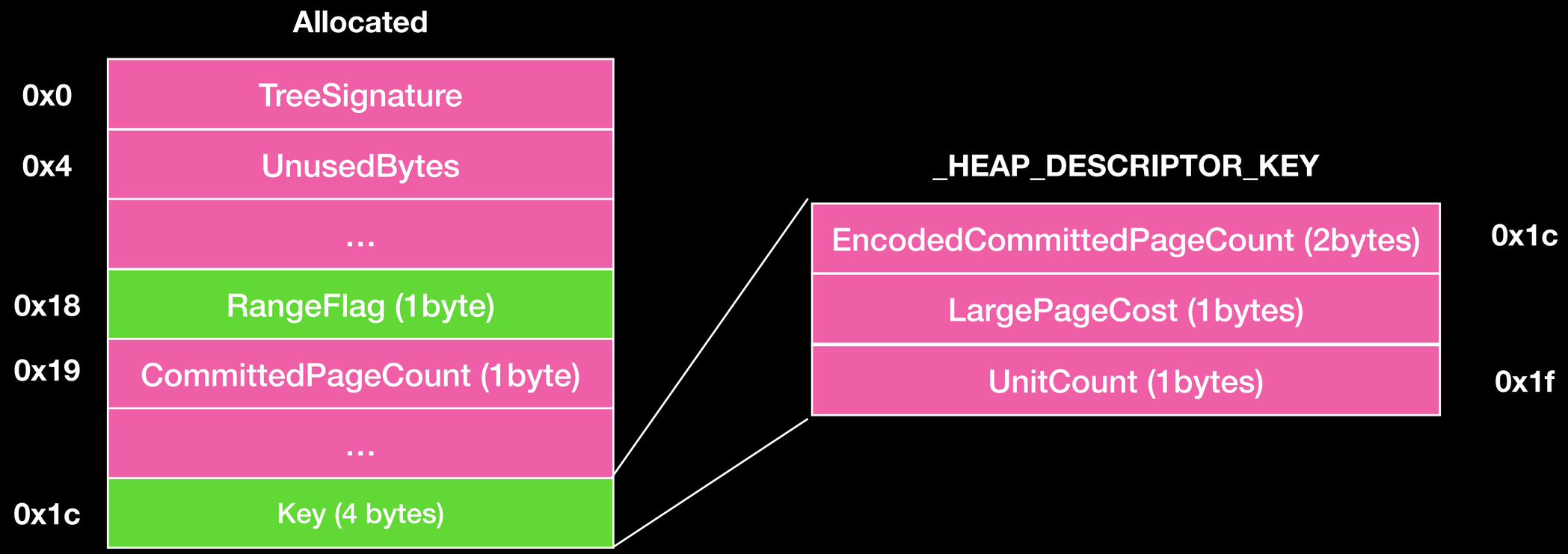
- _HEAP_PAGE_RANGE_DESCRIPTOR
  - Key(_HEAP_DESCRIPTOR_KEY)
    - Store the size of the page corresponding to the page descriptor and the number of committed pages

Segment Allocation

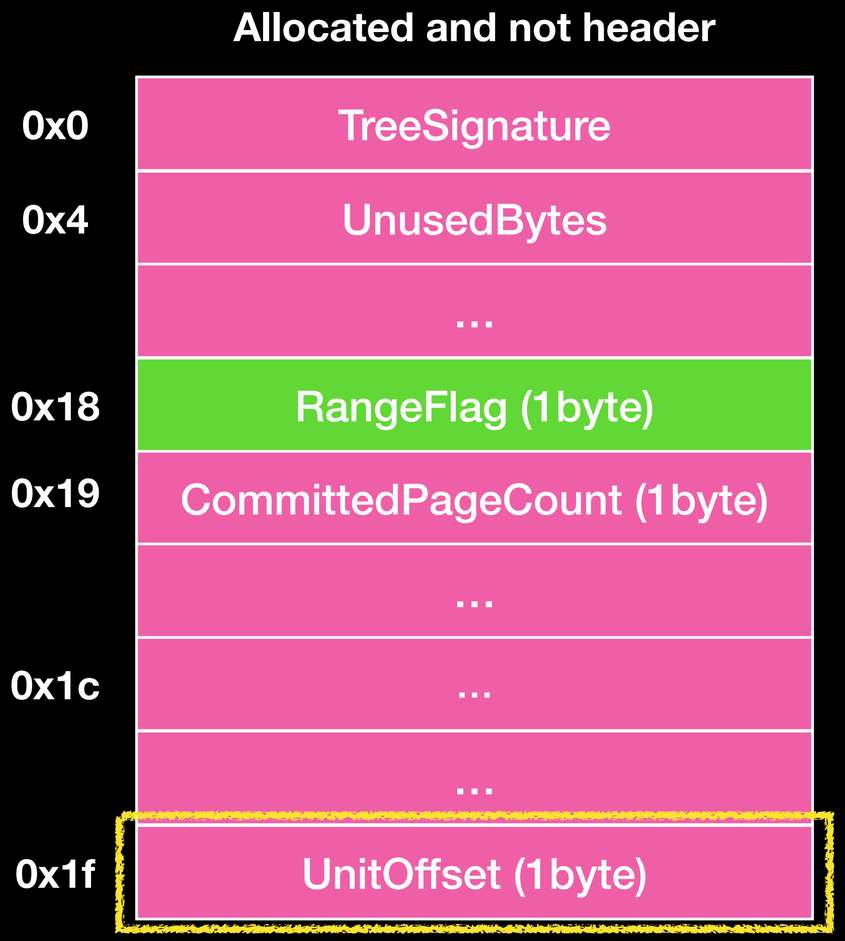


- _HEAP_DESCRIPTOR_KEY
  - EncodedCommittedPageCount
    - ~EncodedCommittedPageCount is the number of pages committed in the block
    - Only used in block header
  - UniCount
    - The size of Block
    - Value is page count

Segment Allocation

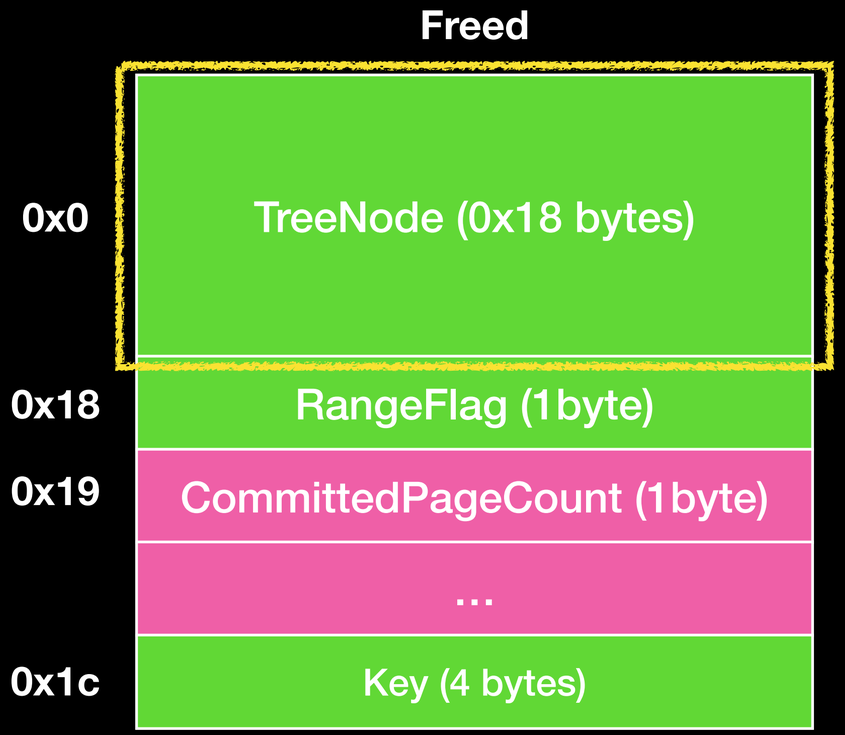


Segment Allocation



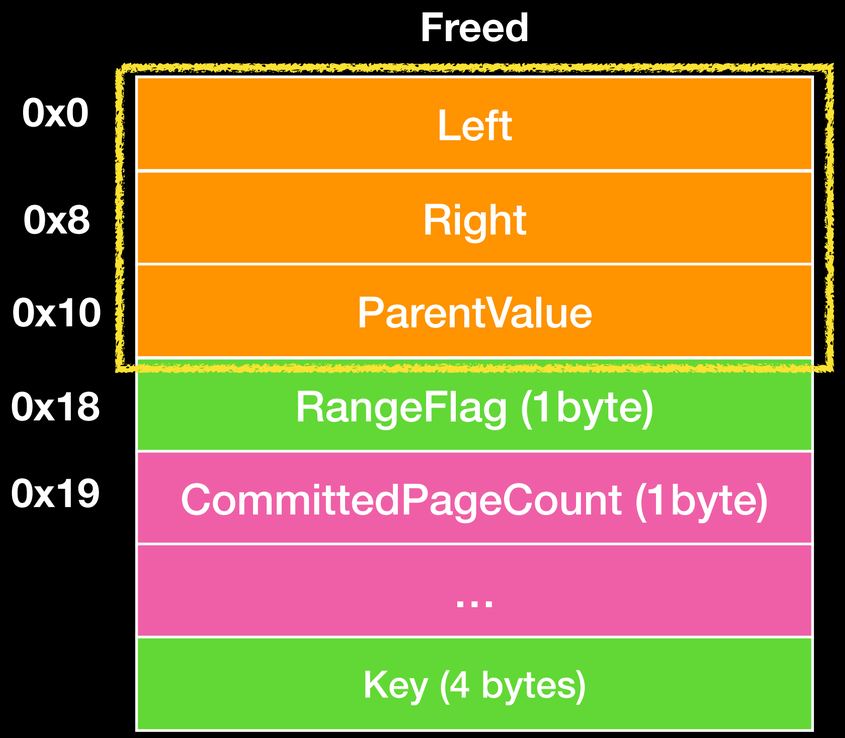
- _HEAP_PAGE_RANGE_DESCRIPTOR
  - UnitOffset
    - If page is not block header
      - The UnitCount field will be called UnitOffset
      - Indicates the offset of the page in the block

Segment Allocation



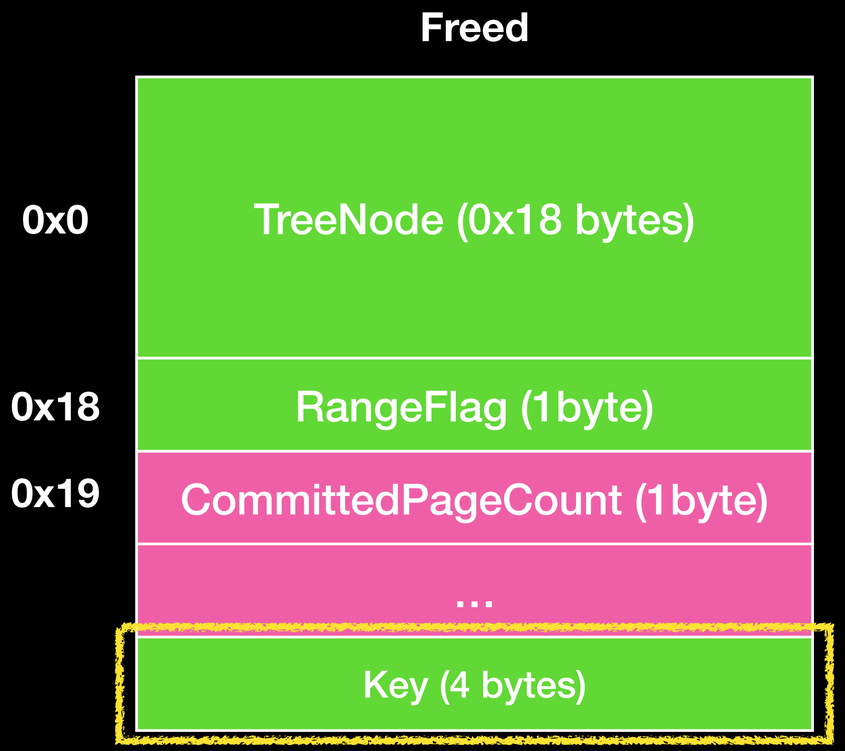
- _HEAP_PAGE_RANGE_DESCRIPTOR
  - TreeNode (_RTL_BALANCED_NODE)
    - The node in the FreePageRanges

Segment Allocation



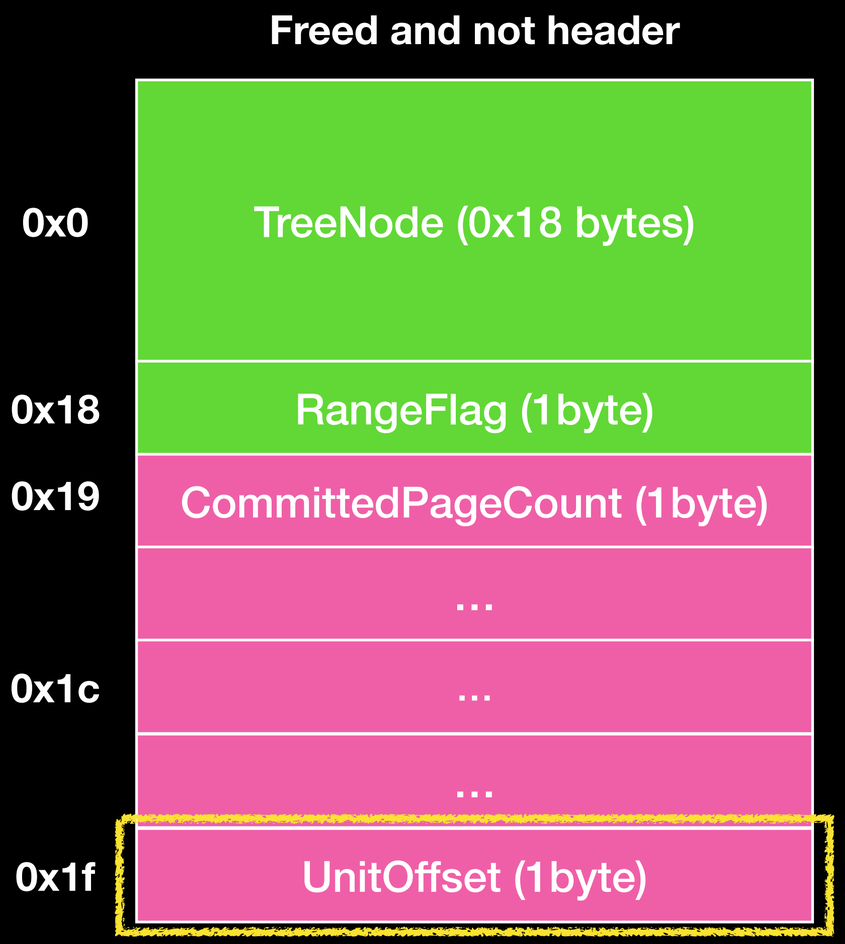
- _HEAP_PAGE_RANGE_DESCRIPTOR
  - TreeNode (_RTL_BALANCED_NODE)
    - Left
      - Point to a page descriptor that block size is smaller than its.
    - Right
      - Point to a page descriptor that block size is greater than its.
    - ParentValue
      - Point to parent node
      - The lowest 1 bit will determine whether to encode Parent

Segment Allocation



- _HEAP_PAGE_RANGE_DESCRIPTOR
  - Key(_HEAP_DESCRIPTOR_KEY)
    - Same as the case of allocated

Segment Allocation

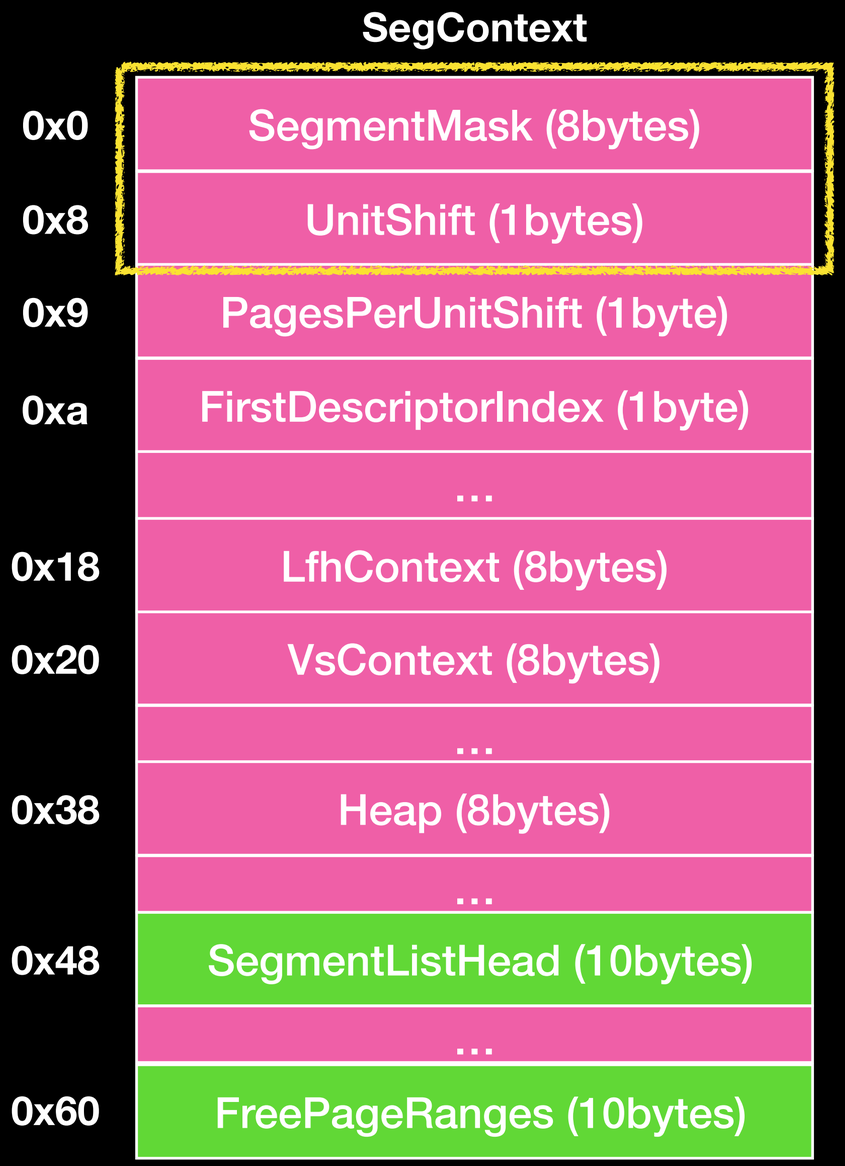


- _HEAP_PAGE_RANGE_DESCRIPTOR
  - UnitOffset
    - If page is not block header
    - The UnitCount field will be called UnitOffset
    - Indicates the offset of the page in the block

Segment Allocation

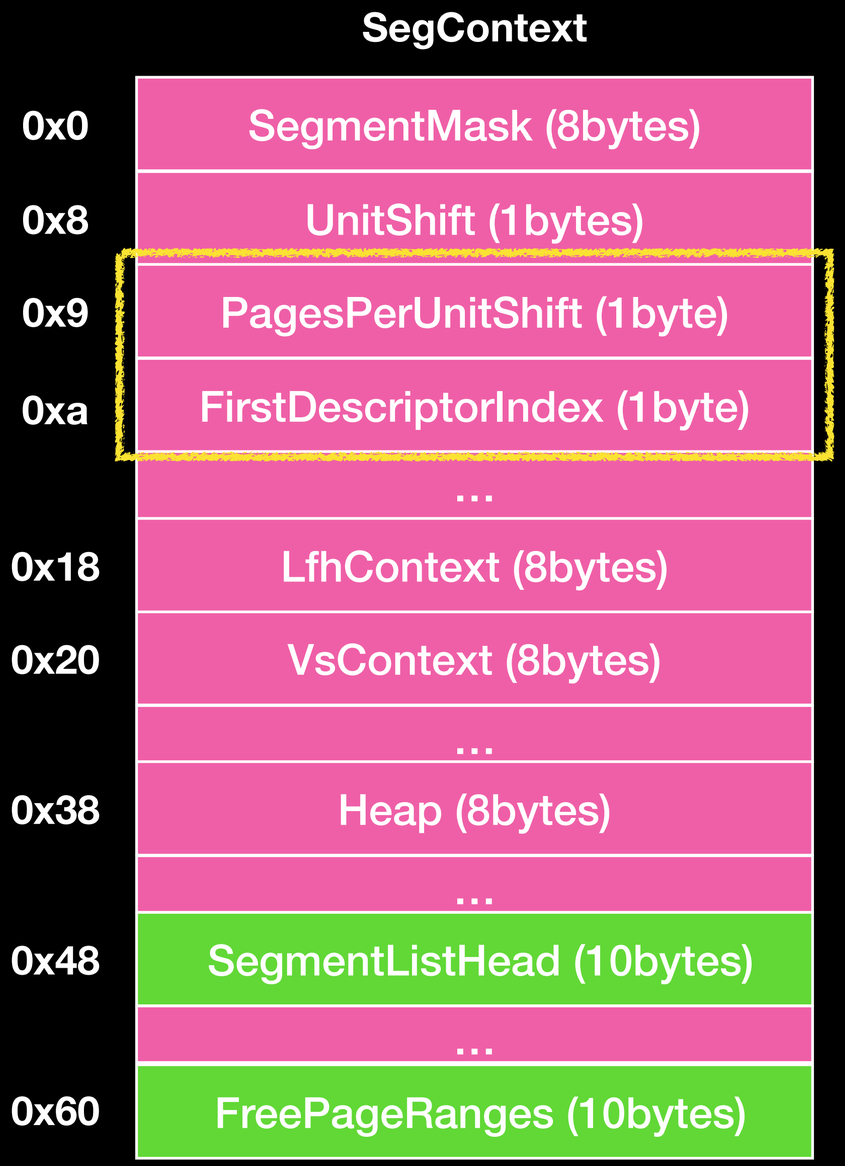
- SegContexts (_HEAP_SEG_CONTEXT)
  - The core structure of the Segment allocation
  - Used to manage the memory allocated by segment allocator, and record all the information and structure of segment allocator in the heap
  - There are two SegContexts in each Heap
    - Size <= 0x7f000
    - 0x7f000 < Size < 0x7f0000

Segment Allocation



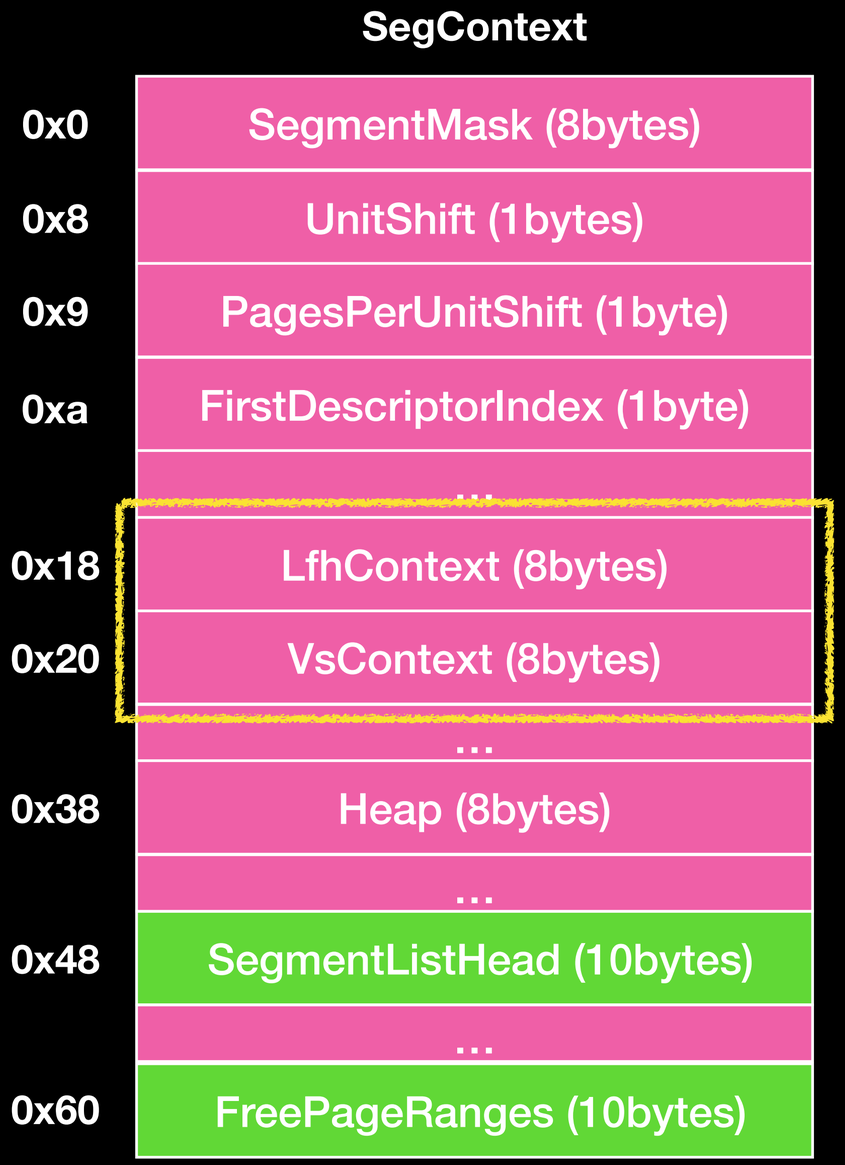
- SegContexts (_HEAP_SEG_CONTEXT)
  - SegmentMask
    - A mask used to find Page Segment
    - Page segment = block ptr & SegmentMask
  - UnitShift
    - Used to calculate the index of the page descriptor
    - Index = block ptr >> UnitShit

Segment Allocation



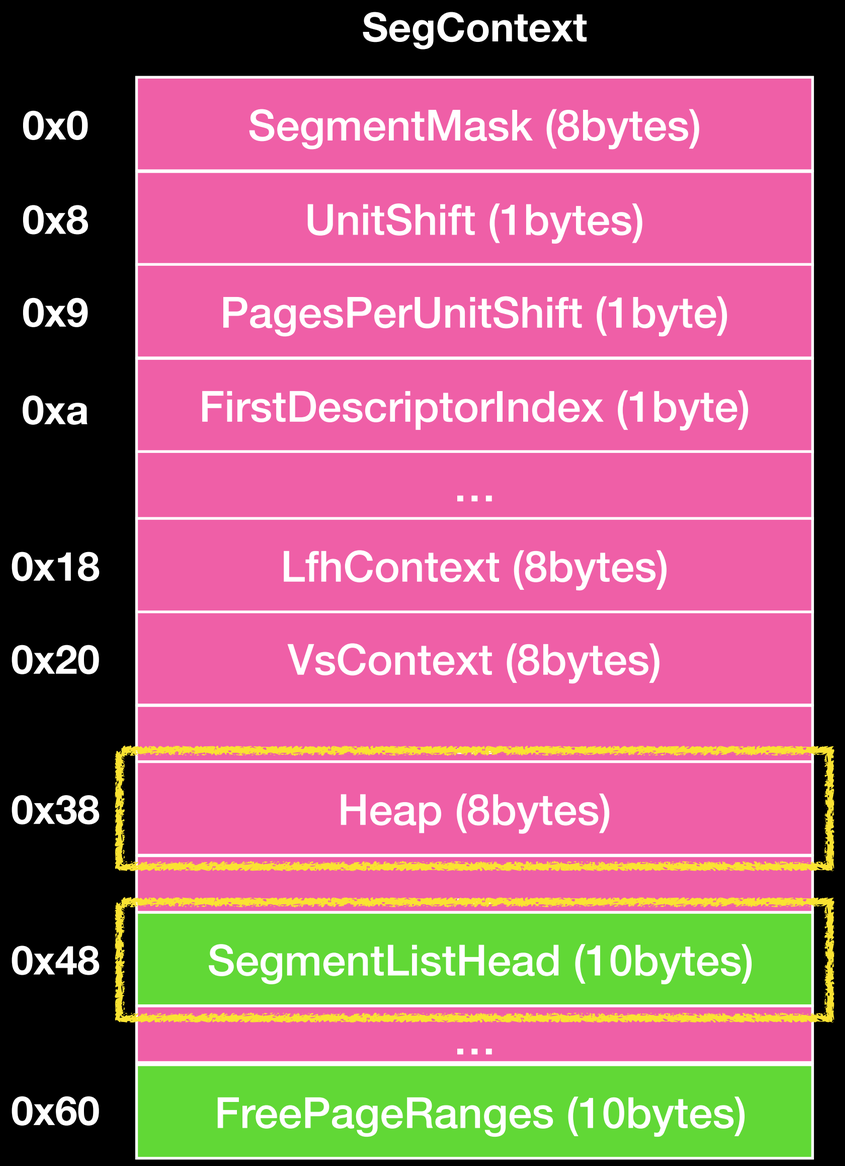
- SegContexts (_HEAP_SEG_CONTEXT)
  - PagePerUnitShift
    - (0x1 << PagePerUnitShift) indicates the size of a page in the SegContext
    - The size of page unit is (0x1 << PagePerUnitShift)*0x1000
      - If the value is zero
        - Page unit is 0x1000
  - FirstDescriptorIndex
    - The index of the first Page Descriptor in the SegContext

Segment Allocation



- SegContexts (_HEAP_SEG_CONTEXT)
  - LfhContext (_HEAP_LFH_CONTEXT)
    - Point to the LFH allocator in the segment heap
  - VsContext (_HEAP_VS_CONTEXT)
    - Point to the VS allocator in the segment heap

Segment Allocation

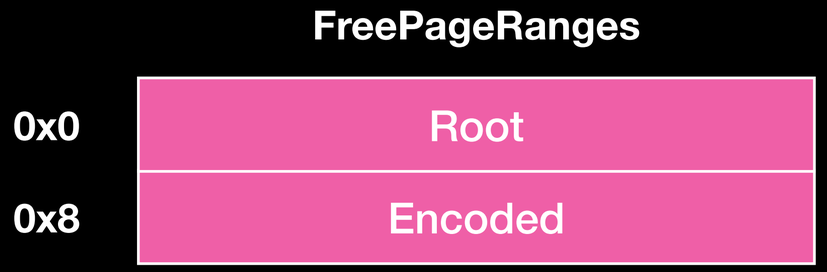


- SegContexts (_HEAP_SEG_CONTEXT)
  - Heap(_SEGMENT_HEAP)
    - Point to the segment heap to which it belongs
  - SegmentListHead (_LIST_ENTRY)
    - Point to the page segment in the segment allocator
    - The linked list is a double linked list with integrity check

Segment Allocation

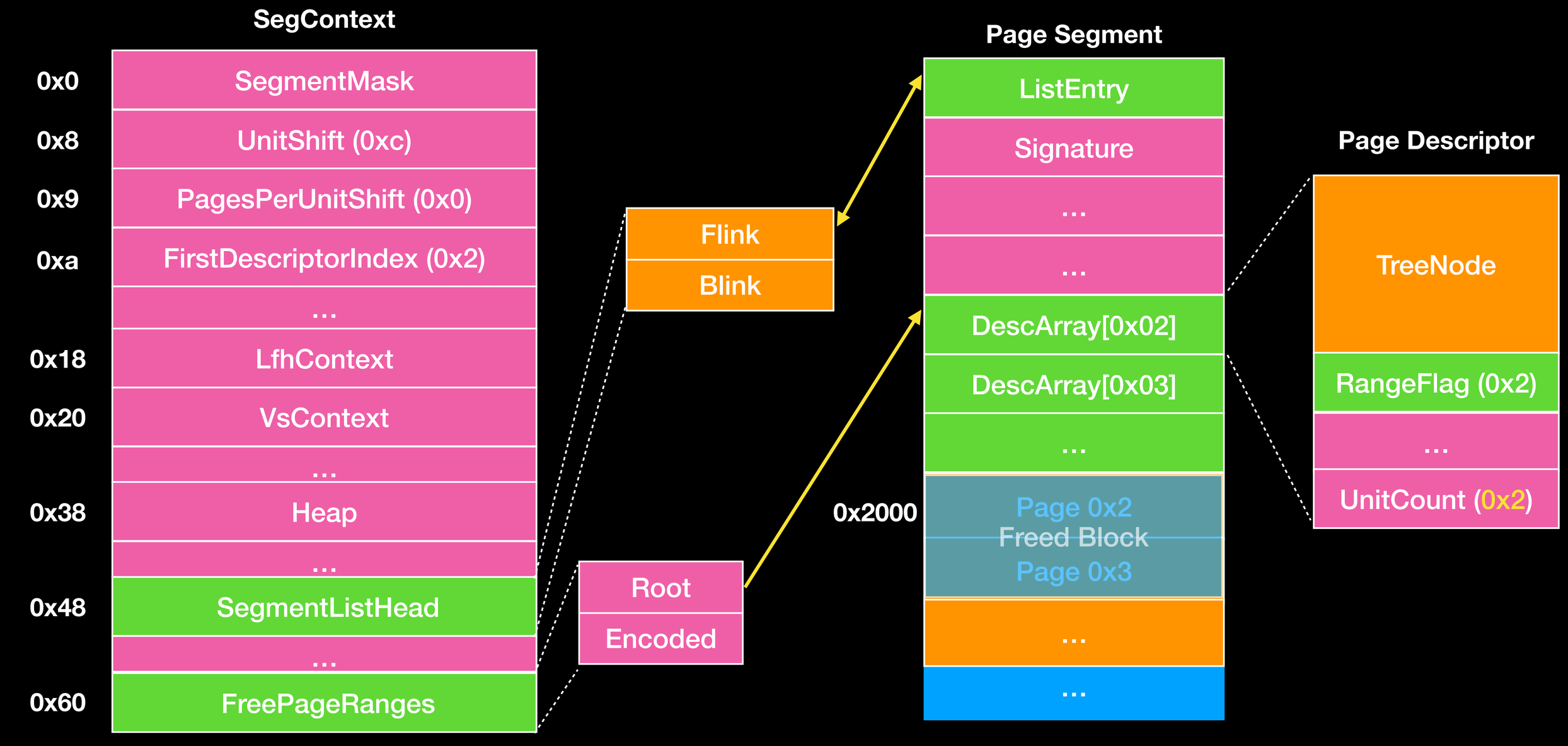
- FreePageRanges (_RTL_RB_TREE)
  - In Segment Allocation, after releasing a block, the page descriptor of the block will be inserted into the FreePageRanges of the SegContext according to the size
    - If the block size is greater than the node, the page descriptor will be inserted into the right subtree, otherwise will be will be inserted into left subtree.
  - If there is no greater than the page descriptor, the right subtree is NULL and the other side is also
  - There will be a node check when the node is taken out of the tree

Segment Allocation

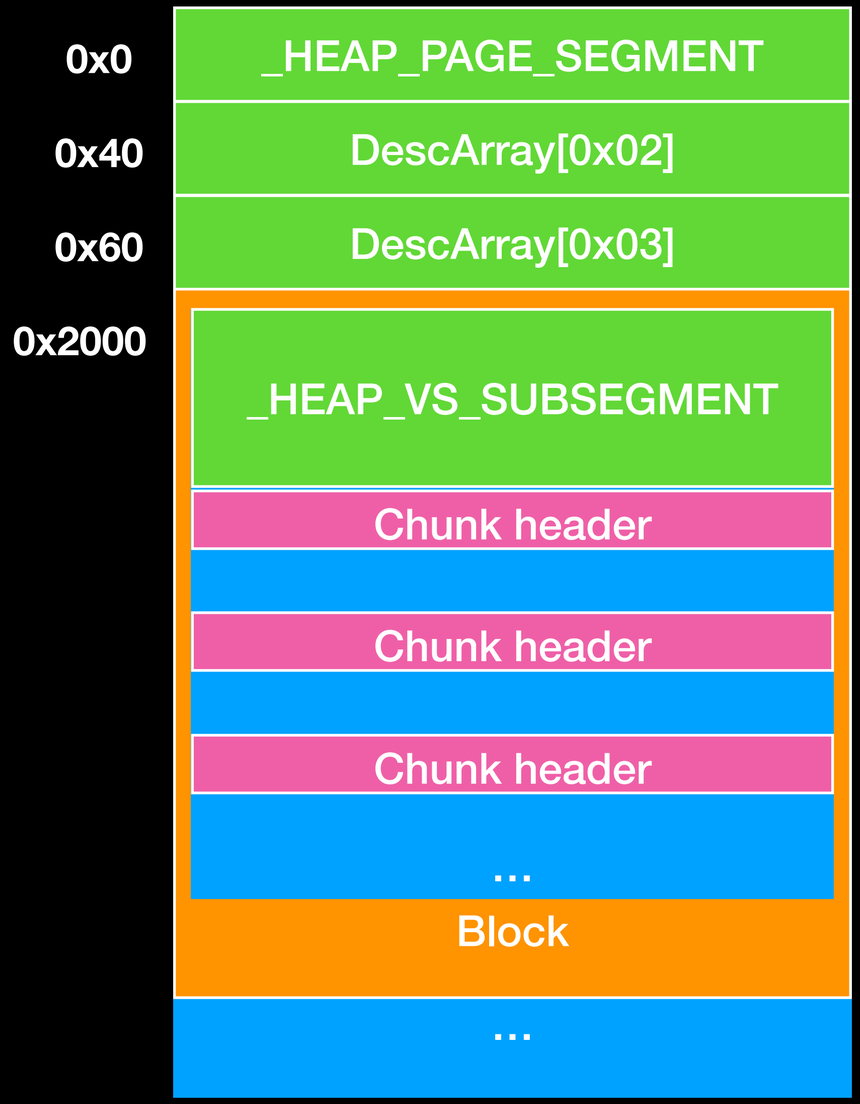


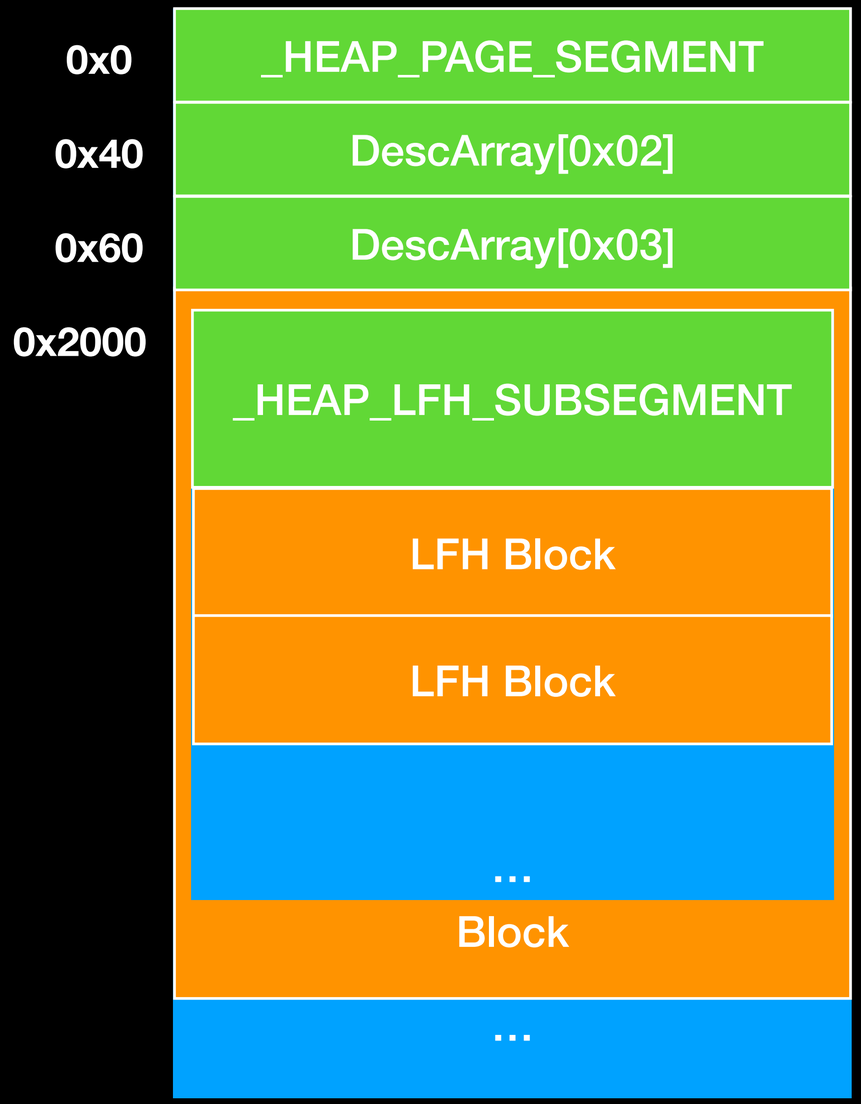
- SegContexts (_HEAP_SEG_CONTEXT)
  - FreePageRanges (_RTL_RB_TREE)
  - Root
    - Point to the root of the rbtree
  - Encoded
    - Indicates whether the root has been encoded (default disable)
  - About encode
    - EncodedRoot = Root ^ FreeChunkTree

Segment Allocation



Segment Allocation





Segment Allocation

- Data Structure
- Memory allocation mechanism

Segment Allocation

- Allocate
  - The allocation is based on page as the unit to allocate, and divided into
    - Size <= 0x7f000
      - Page uses 0x1000 bytes as a unit
    - 0x7f000 < Size < 0x7f0000
      - Page uses 0x10000 bytes as a unit

Segment Allocation

- Allocate
  - The allocation is based on page as the unit to allocate
    - For example, if it allocate 0x1337 bytes, segment allocation will allocate 0x2000, which is 2 page units, and the extra memory 0x2000-0x1337 will be recorded in unused byte

Segment Allocation

- Allocate
  - Main implementation function is nt!RtlpHpSegAlloc
    - It will use RtlpHpSegPageRangeAllocate to get freed page descriptor or create a new page descriptor
      - First, it will search from FreePageRanges.
        - Start searching from the root, when the required block is larger than the node, continue searching from the right subtree until it is found or is NULL

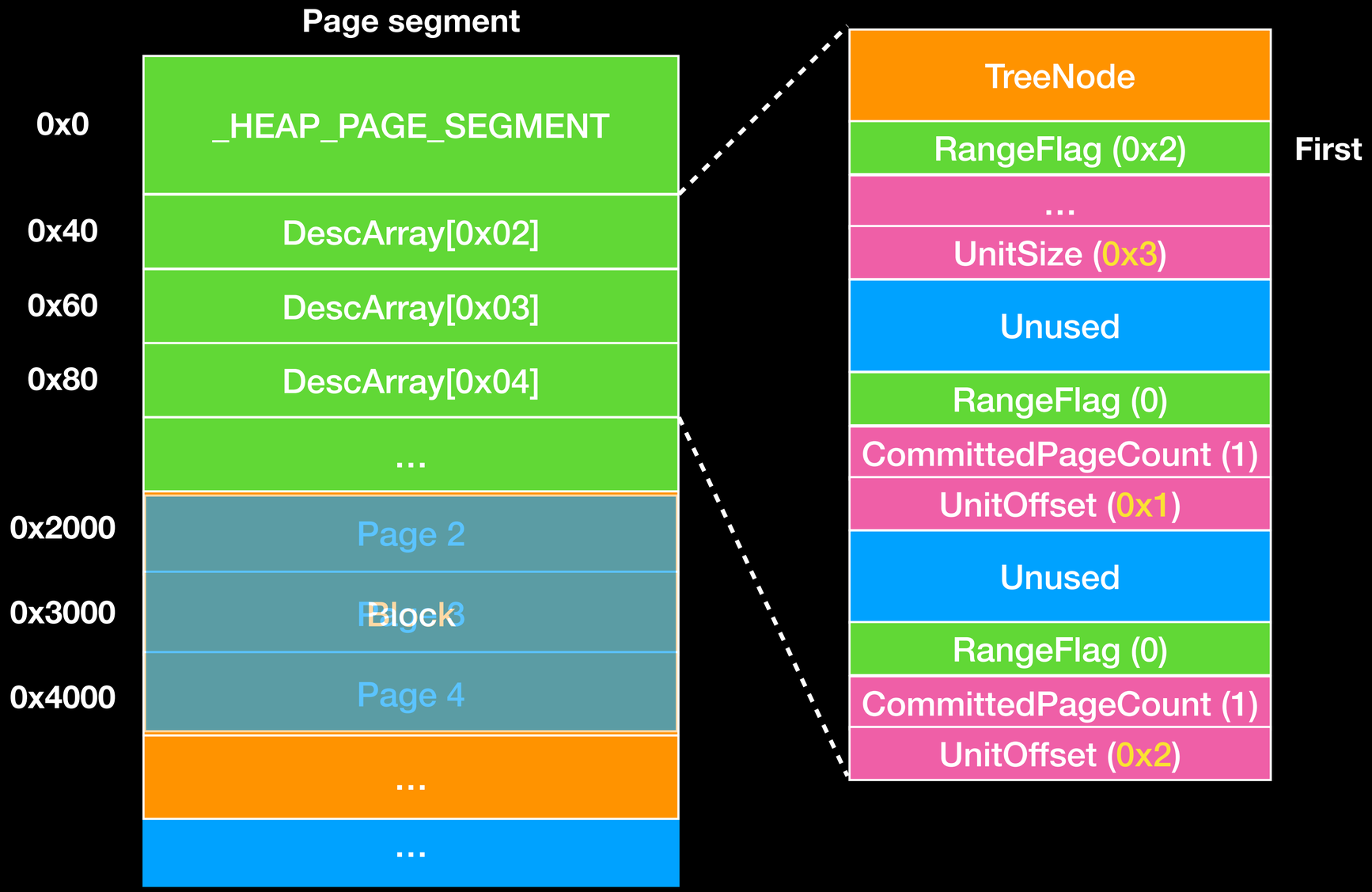
Segment Allocation

- Allocate
  - If no suitable page descriptor is found, a new page segment will be allocated and the first page descriptor of the page segment will be initialized, and then this page descriptor will be used for allocated. The page segment will be inserted into the SegmentListHead.
  - In fact, it only allocated the memory required for the page segment and page descriptor structure, and the block part is not allocated at first
    - RtlpHpSegSegmentAllocate : Allocate page segment
    - RtlpHpSegSegmentInitialize : Initialize the first page descriptor
    - RtlpHpSegHeapAddSegment : insert into SegmentListHead

Segment Allocation

- Allocate
  - If it found a suitable or created a page descriptor, the page descriptor will be removed from FreePageRanges
    - When the required number of pages is smaller than the block, splitting will be done.
    - The page descriptor corresponding to the remainder block will be inserted into FreePageRanges

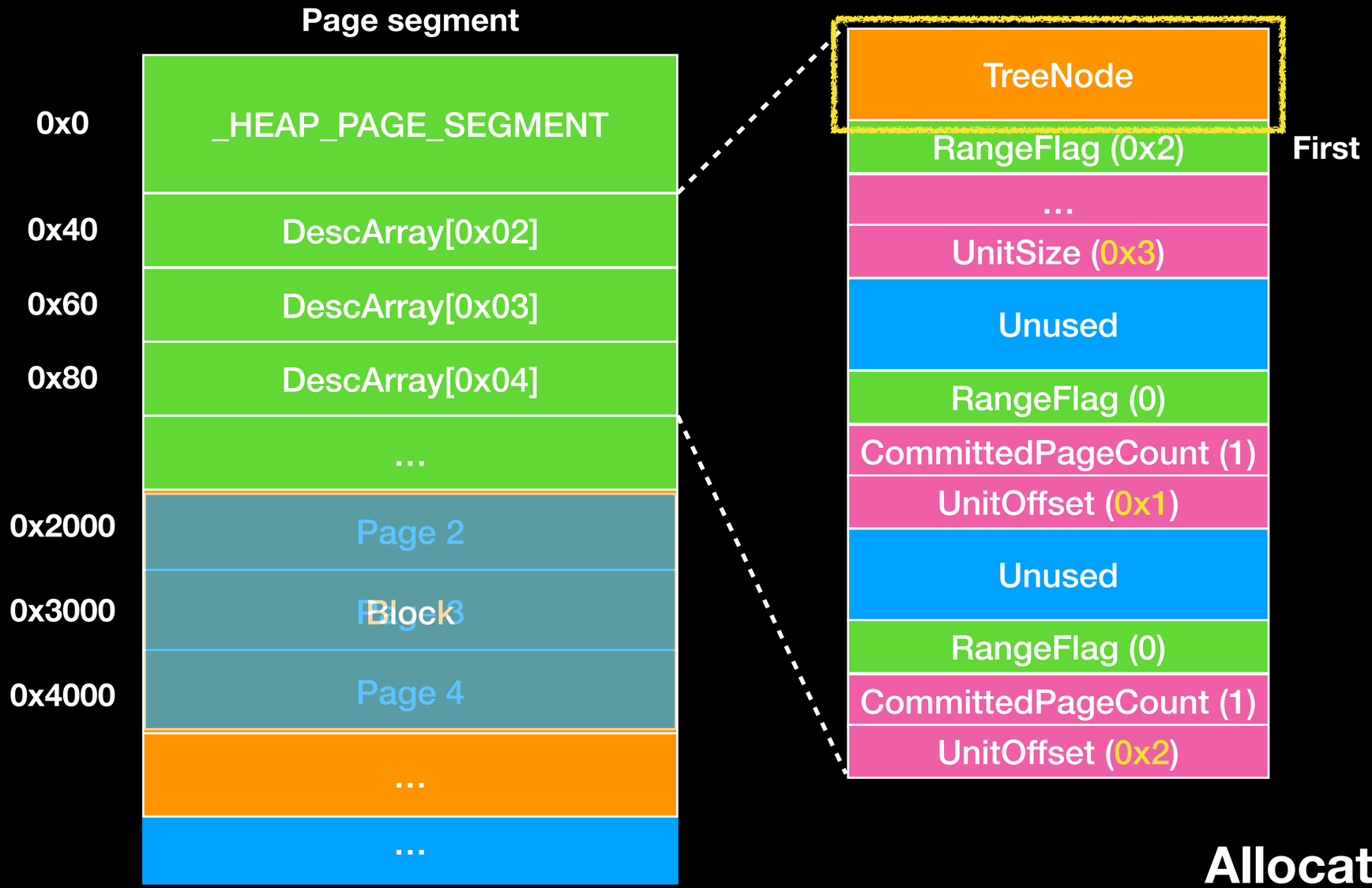
Segment Allocation



Segment Allocation

Found node

Page segment



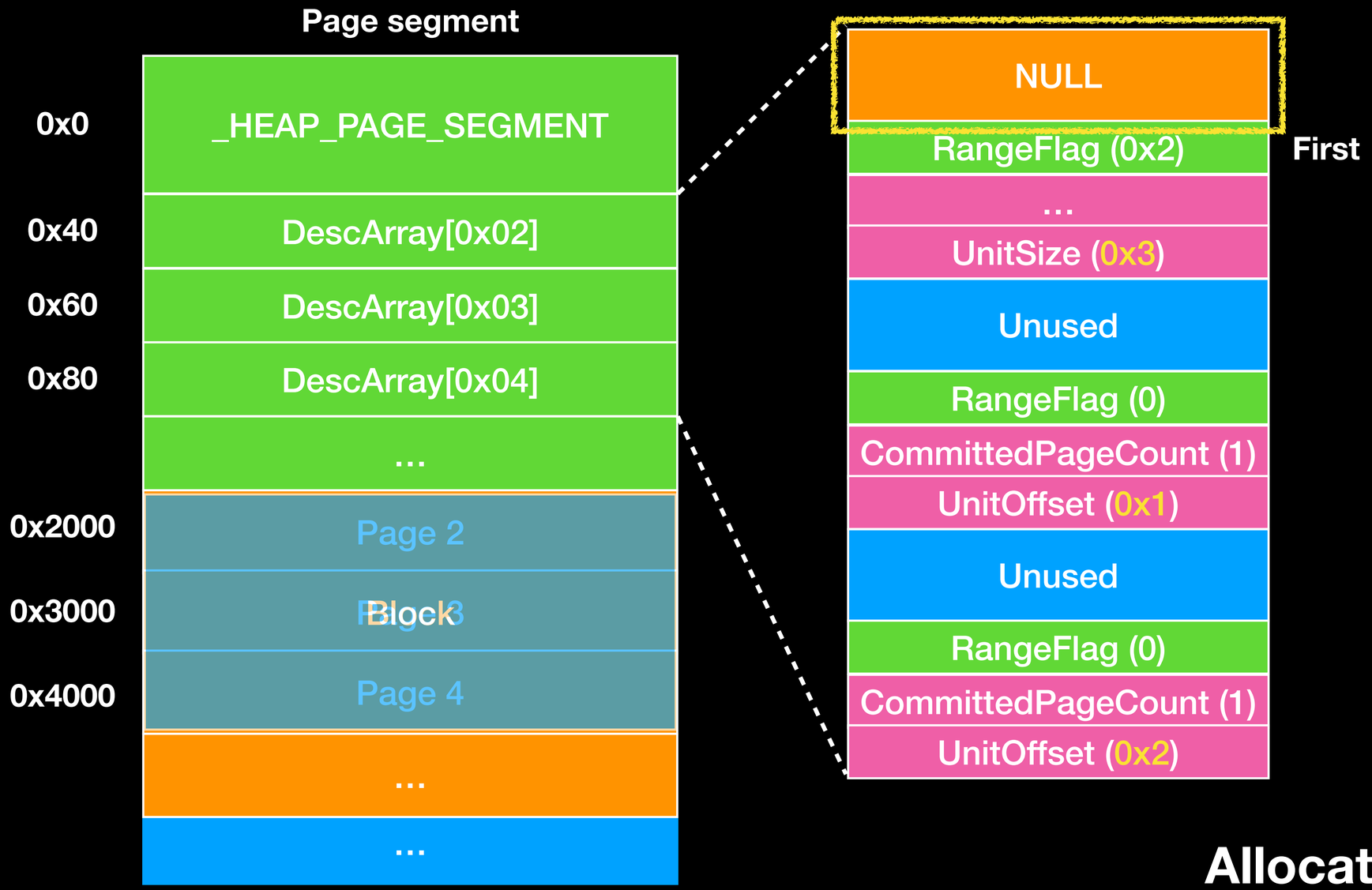
First

Allocate 0x1337

Segment Allocation

Remove node

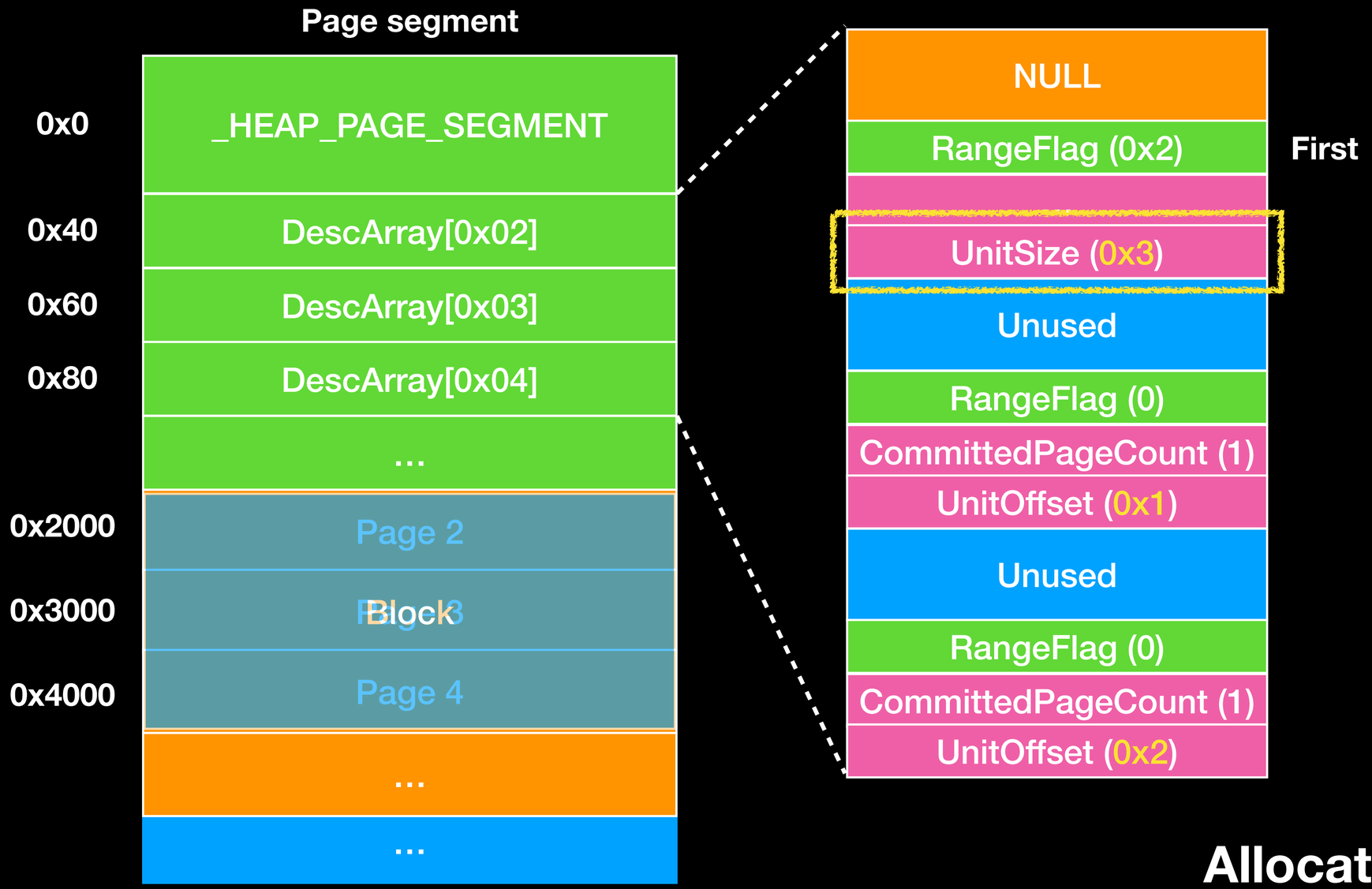
Page segment



Allocate 0x1337

Segment Allocation

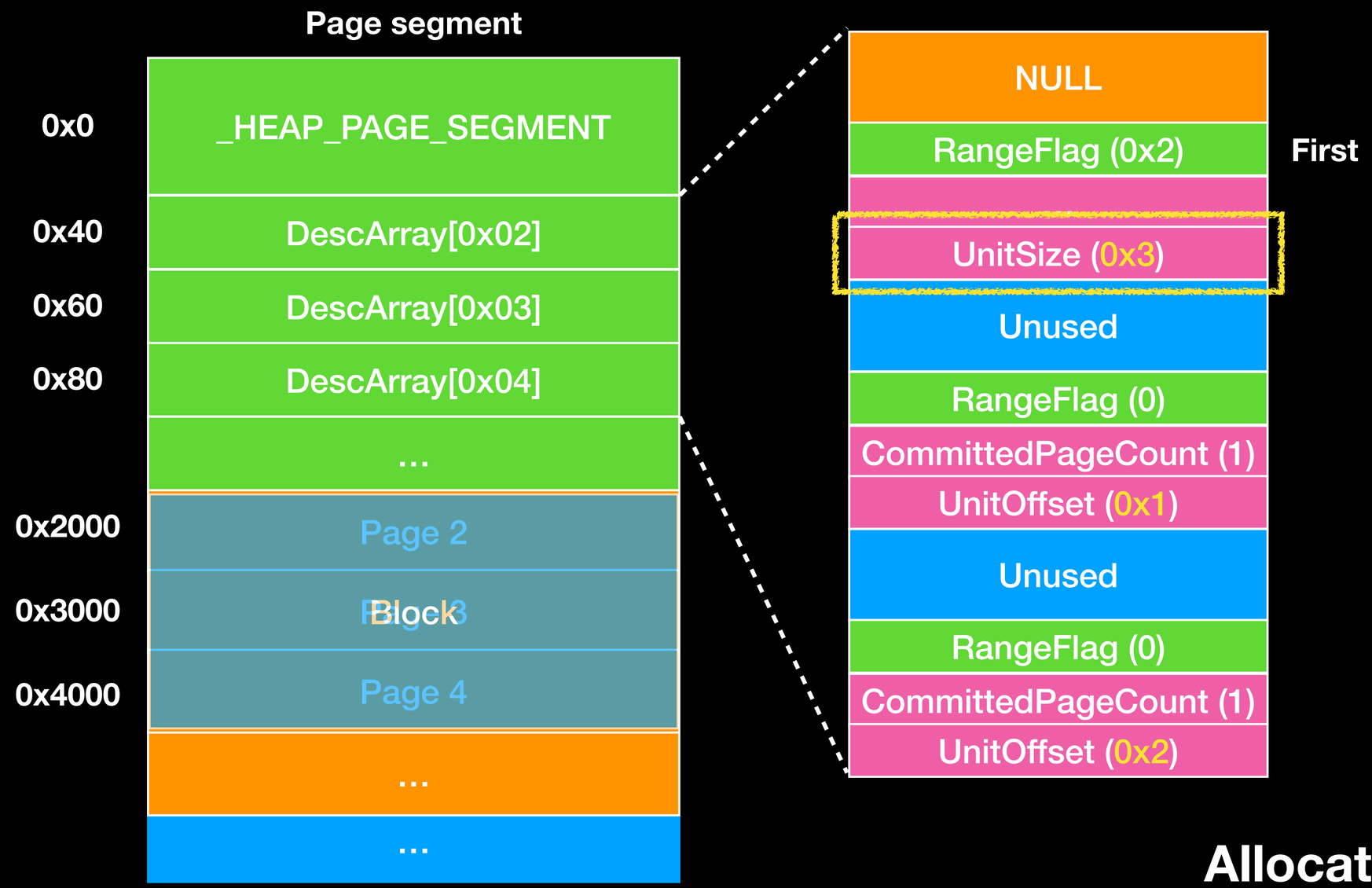
Request size
< block size



Allocate 0x1337

Segment Allocation

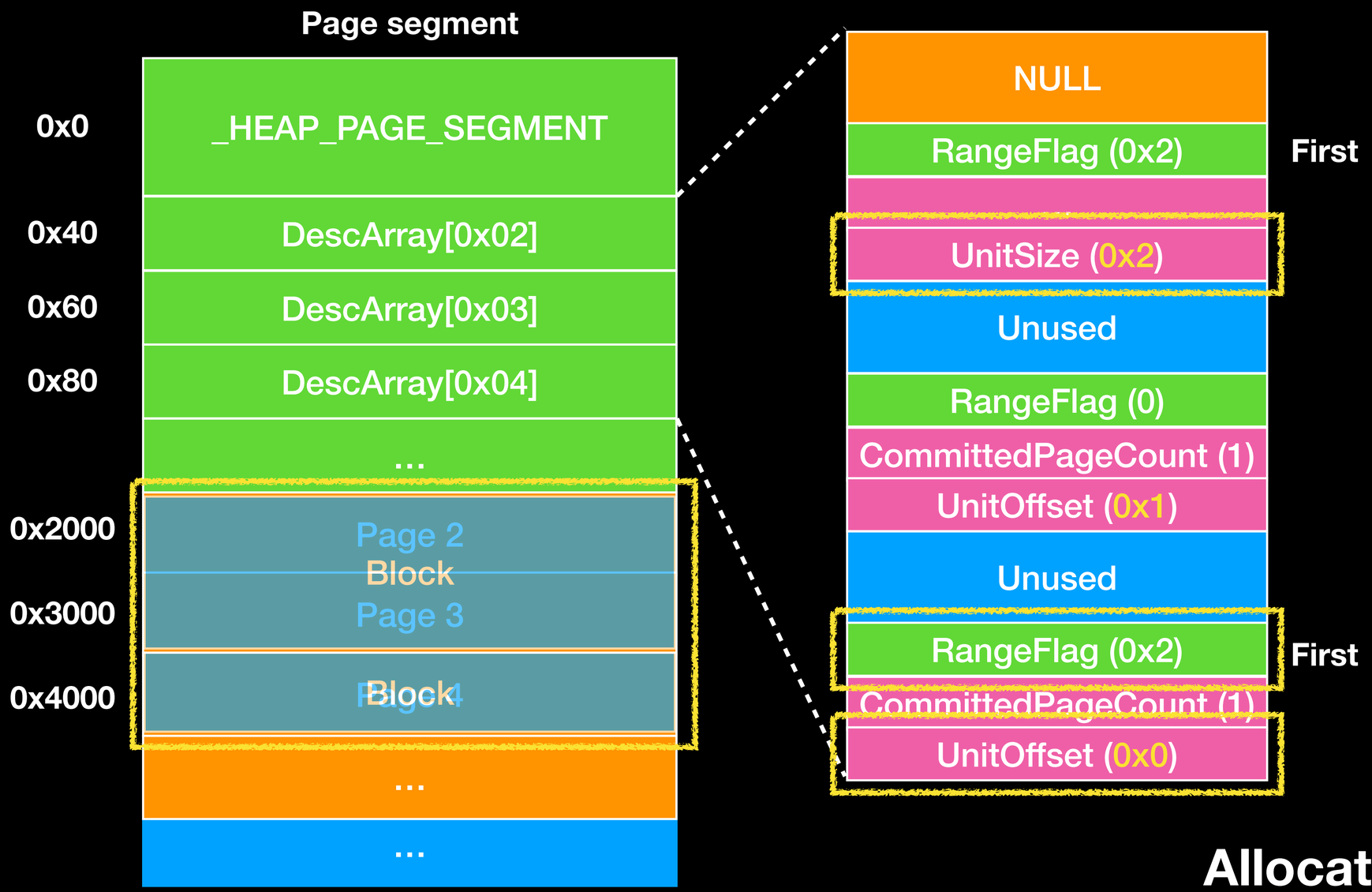
Split Block and update page descriptor of next block



Allocate 0x1337

Segment Allocation

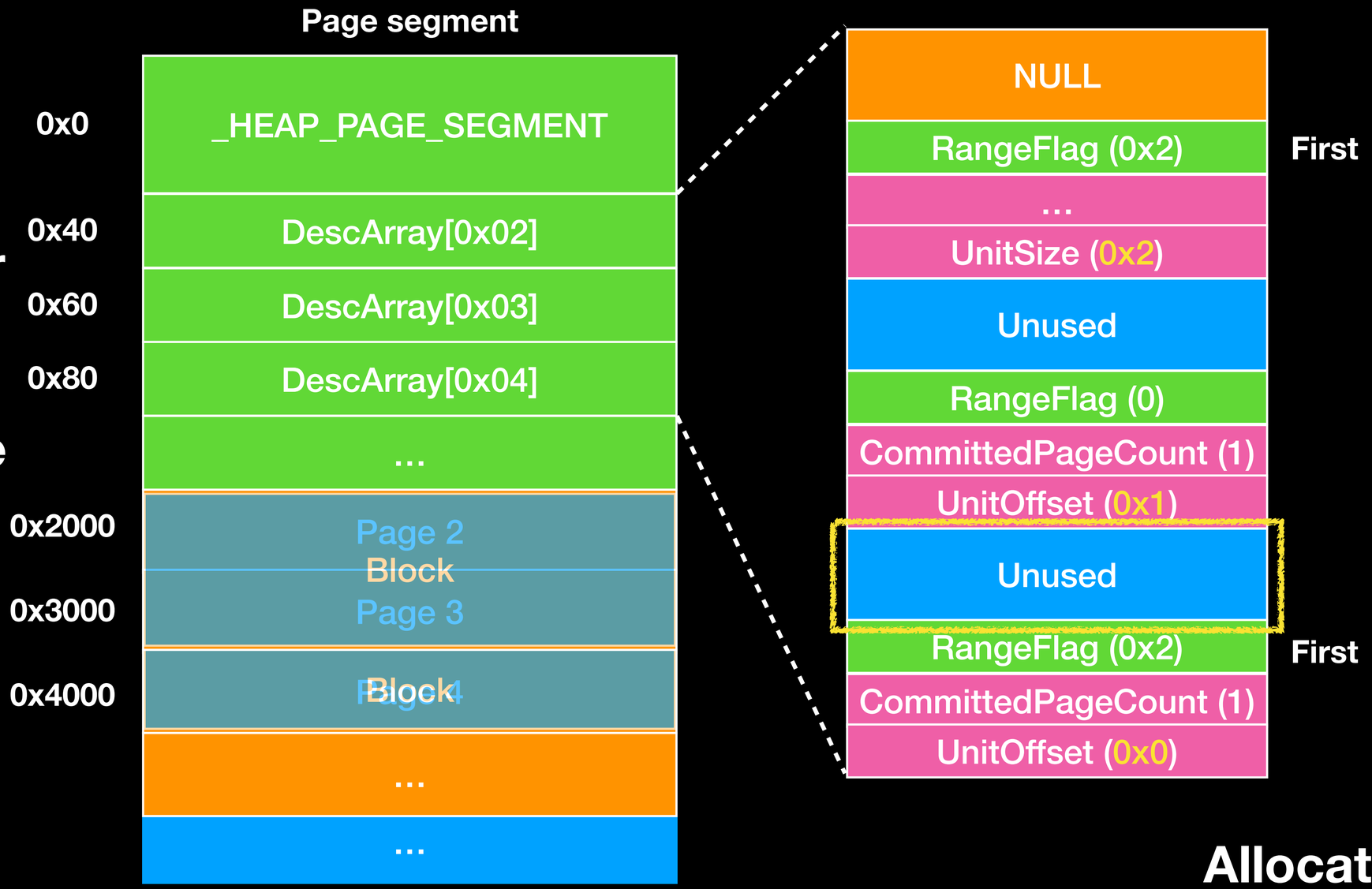
Split Block and update page descriptor of next block



Allocate 0x1337

Segment Allocation

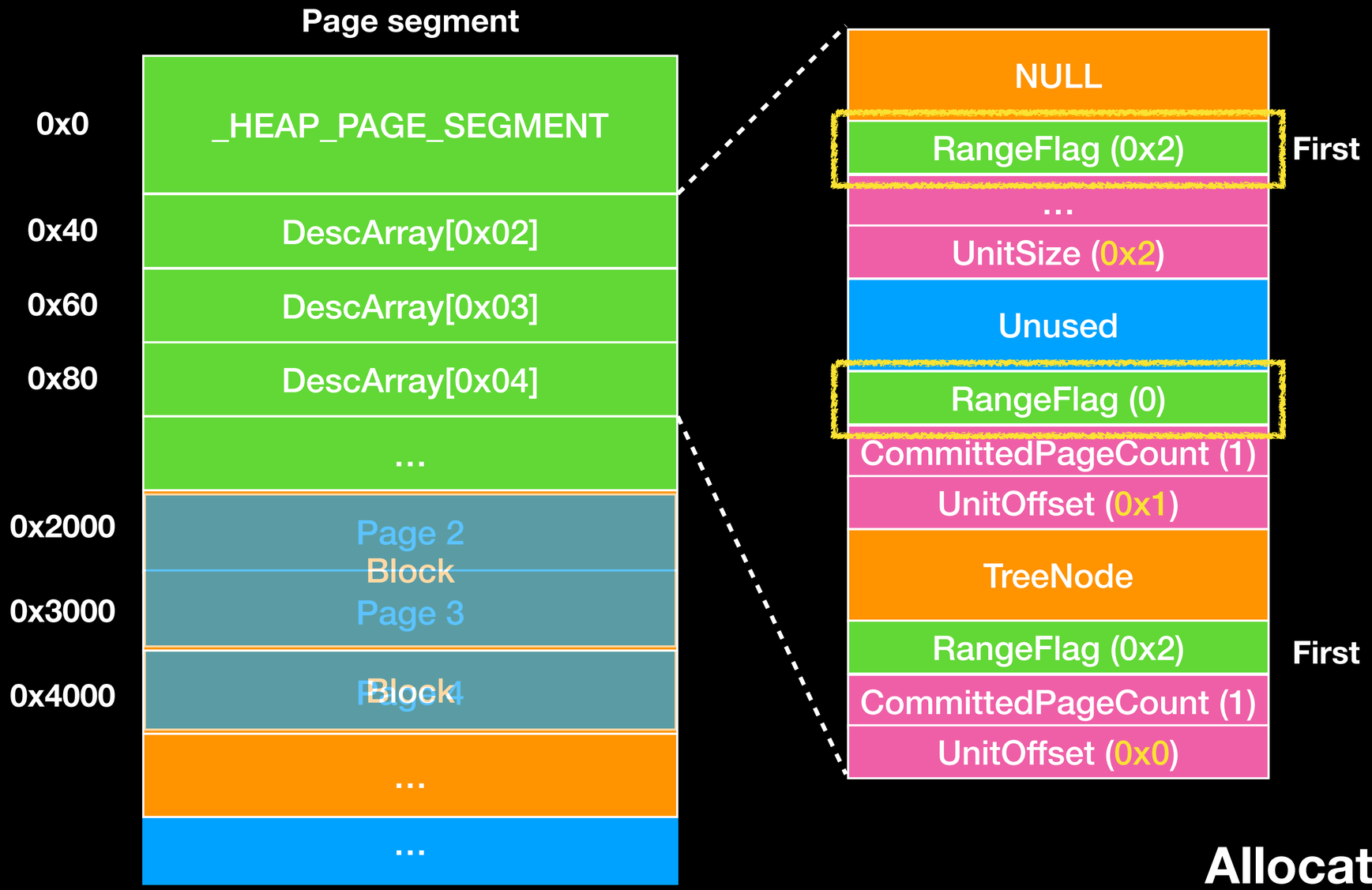
Insert page descriptor of next block to FreePageRange



Allocate 0x1337

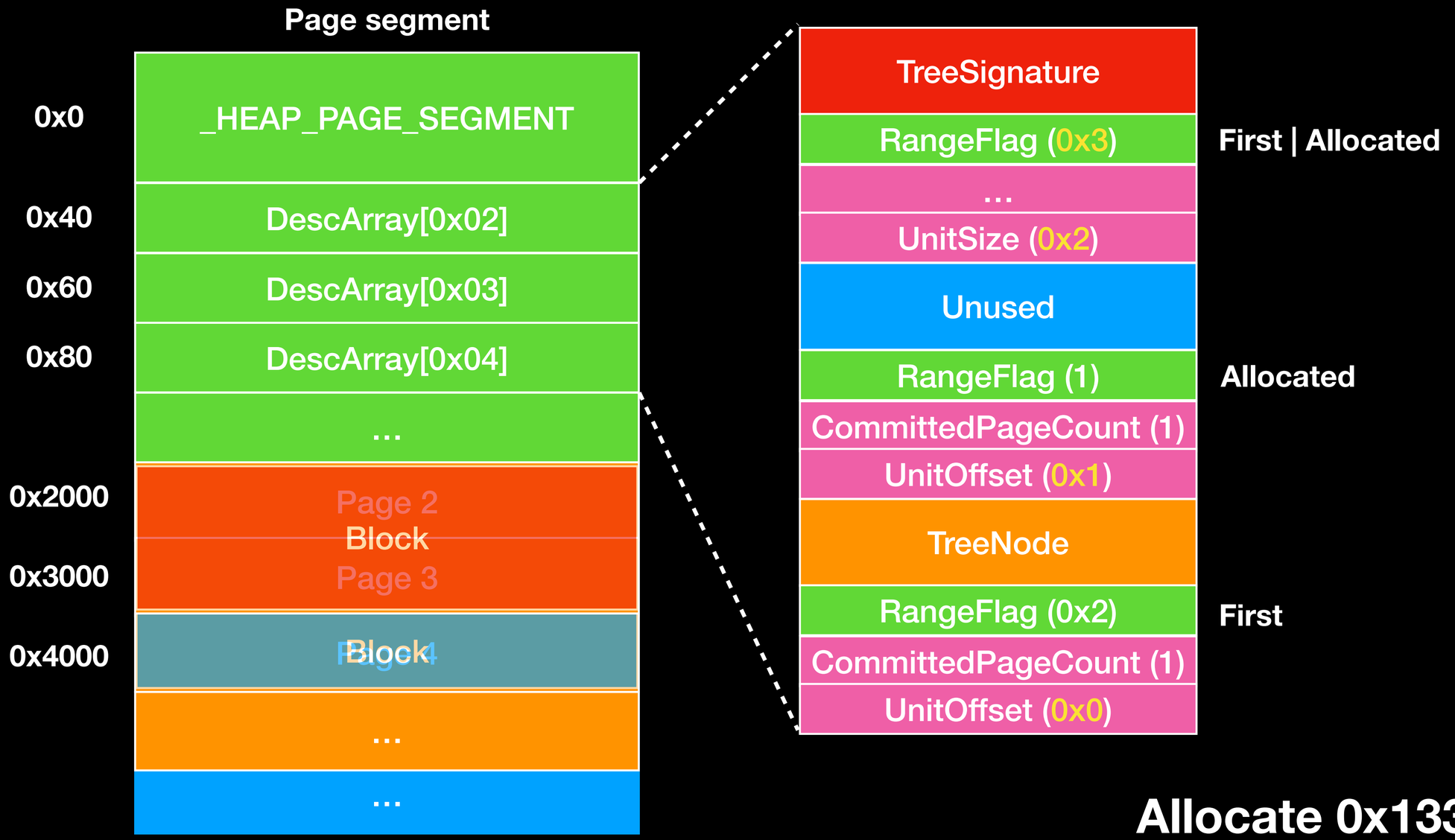
Segment Allocation

Update page descriptor of allocated block



Allocate 0x1337

Segment Allocation



Allocate 0x1337

Segment Allocation

- Allocate
  - After setting the page descriptor, it will check whether all pages in the block are committed
    - It will add up the CommittedPageCount of all page descriptors in the block
      - If the block needs committed, it will commit memory to the specified VA
        - RtlpHpSegMgrCommit ->RtlpHpAllocVA->MmAllocatePoolMemory
    - Then update the CommittedPagecount of all page descriptors corresponding to the block

Segment Allocation

- Allocate
  - Finally, it will return the block that correspond to the page descriptor.
    - Block = (Page descriptor & SegmentMask) + ((index of Page descriptor) << SegContext->Unitshift)

Segment Allocation

- Free
  - At the beginning, it will verify whether the page segment where the ptr is located is legal
    - RtlpHpSegDescriptorValidate
      - Verify: Page segment->signature
        - 0xA2E64EADA2E64EAD == Page segment^SegContext^ RtlpHpHeapGlobals^signature
      - Verify that the page descriptor of the page where the ptr is located is Allocated
        - Double Free check

Segment Allocation

- Free
  - Remark
    - from free pointer to page segment
      - Free pointer & segment mask
    - Page descriptor =
      page_segment + 0x20 * ((free pointer - page segment) >> segcontext->UnitShift)
    - _HEAP_SEG_CONTEXT =
      (page segment)^(page segment->Signature)^0xA2E64EADA2E64EAD^RtlpHpHeapGlobals.HeapKey

Segment Allocation

- Free
  - Next, we will see if the free pointer is at the beginning of block
    - If free pointer is not at the beginning of the block, it will check the RangeFlag of the Page descriptor to determine whether to use VS Allocator or Lfh Allocator to release the memory
    - If free pointer is not at the beginning of the block, it means that the free pointer is managed by segment allocation, and RtlpHpSegPageRangeShrink will be used

Segment Allocation

- Free
    - Then the Allocated bit of page descriptor correspond to block will be cleared, and it will check whether the previous and following blocks are Freed. If it is Freed, it will be merged
        - RtlpHpSegPageRangeCoalesce
            - The way to find the previous block is to check whether the page descriptor of previous page is at the beginning of the block. If it is not the beginning, it will use the UnitOffset of the page descriptor of previous page to calculate the page descriptor of the previous block.
            - The following is calculated using the UnitCount of the current page descriptor
            - Determine whether the RangeFlag of the Page descriptor at the beginning of the block is Allocated
            - It use page descriptor of the block at the beginning to check whether the block is allocated.

Segment Allocation

- Free
  - If the previous block is Free
    - Remove the Page descriptor of the previous block from FreePageRanges
    - Clear the first bit of the RangeFlag of the Page descriptor of the Block which we want to free
    - Update the UnitCount of the Page descriptor of the previous block
    - Update UnitOffset of the last page descriptor of Block after merge

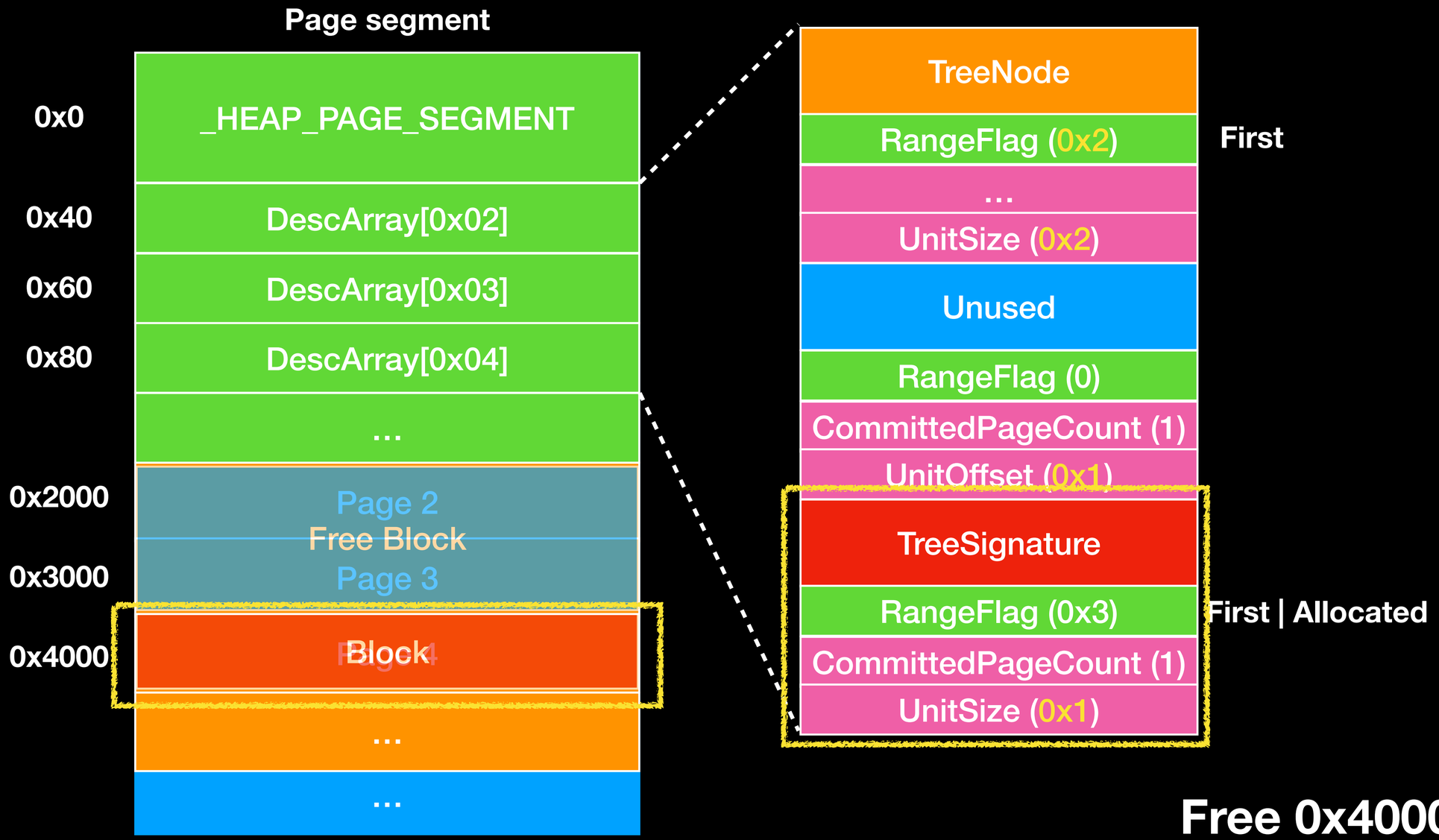
Segment Allocation

- Free
  - If the following block is Free
    - Remove the page descriptor of following block from FreePageRanges
    - Clear the first bit of the RangeFlag of the Page descriptor of the following block
    - Update the UnitCount of the Page descriptor of the block which we want to free
    - Update UnitOffset of the last page descriptor of Block after merge

Segment Allocation

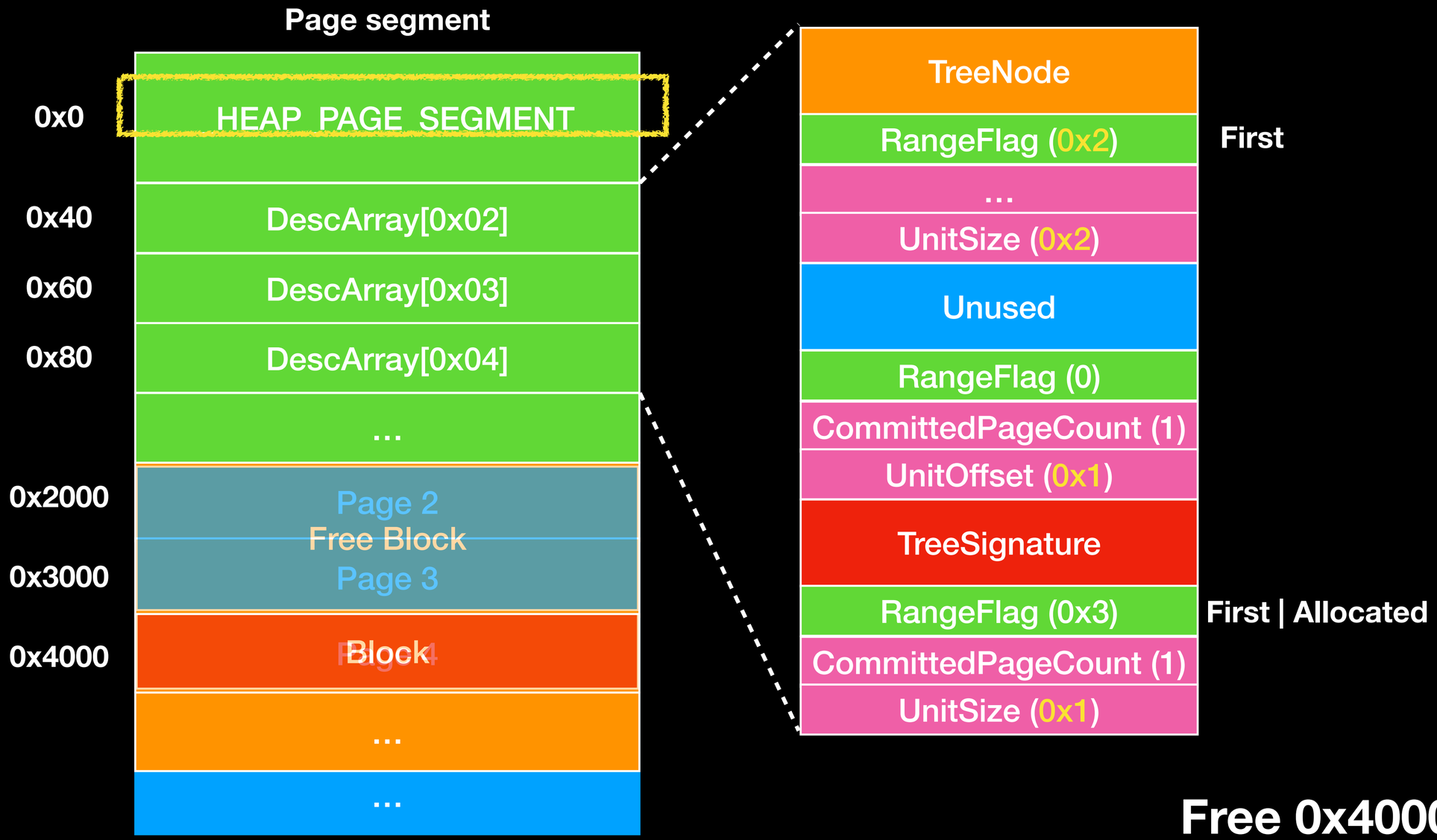
- Free
  - Finally, inserted it to FreePageRanges according to the block size

Segment Allocation



Segment Allocation

Check Signature of Page segment

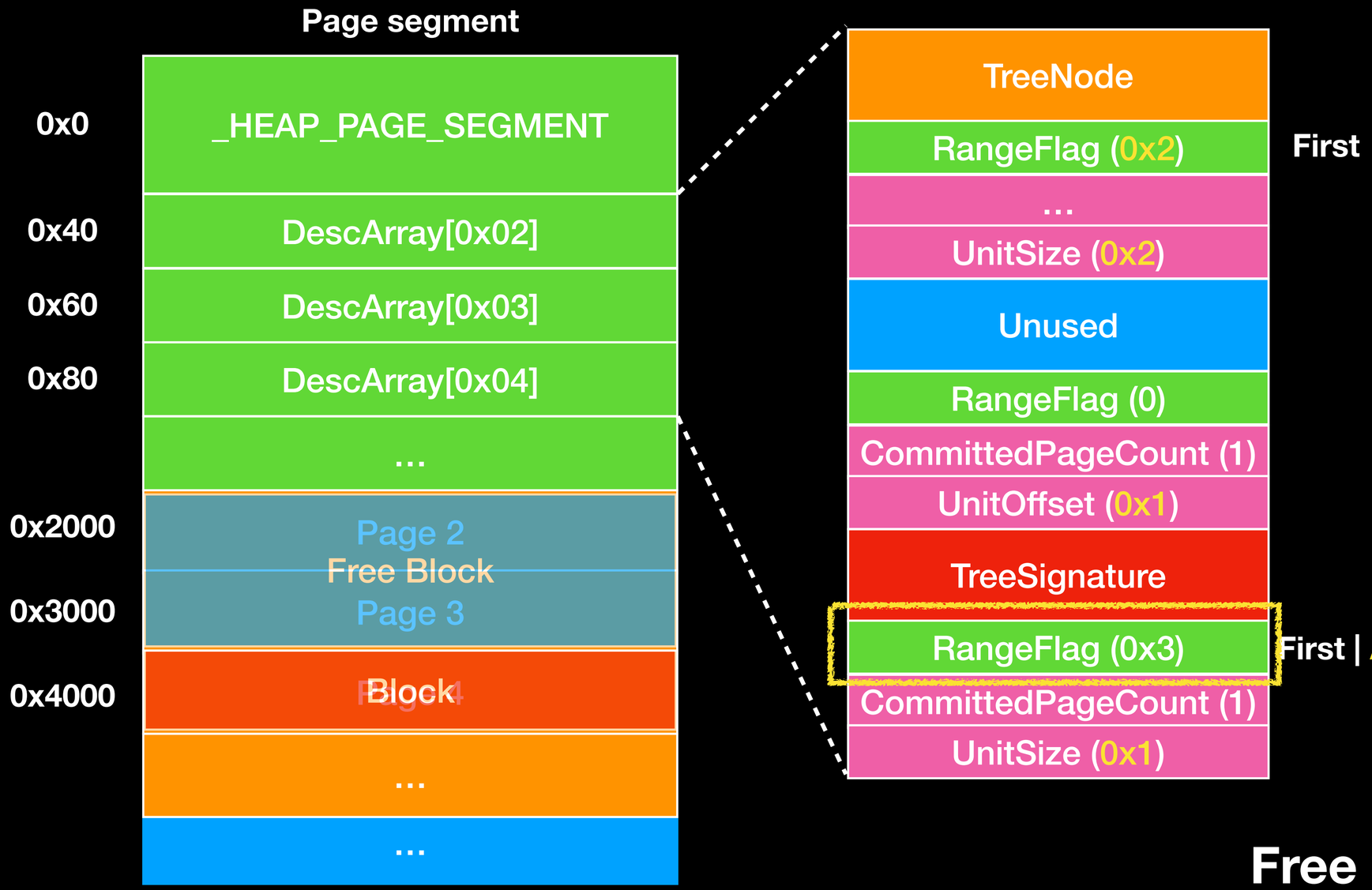


Free 0x4000

Segment Allocation

Check Allocated bit

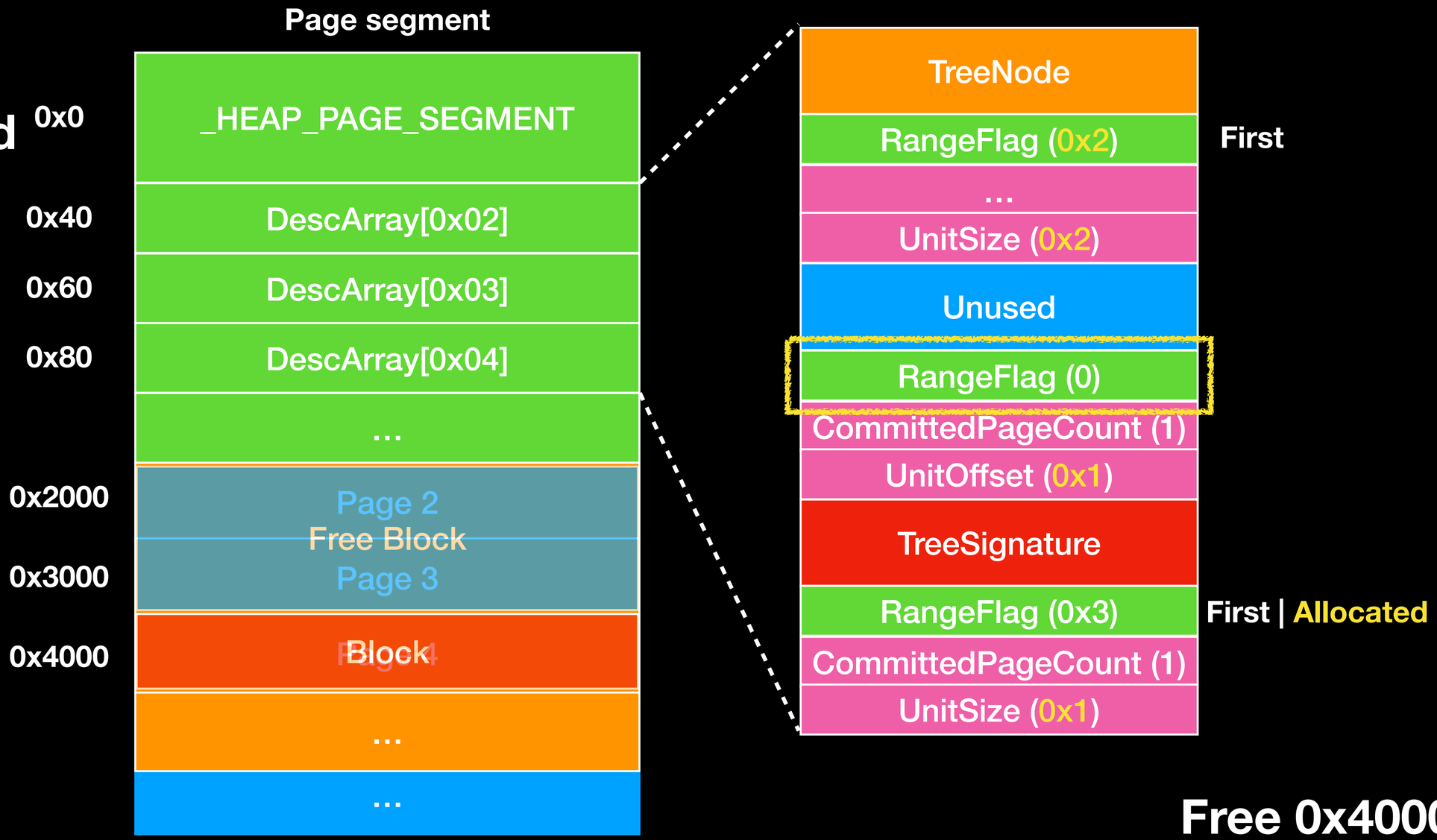
Page segment



Free 0x4000

Segment Allocation

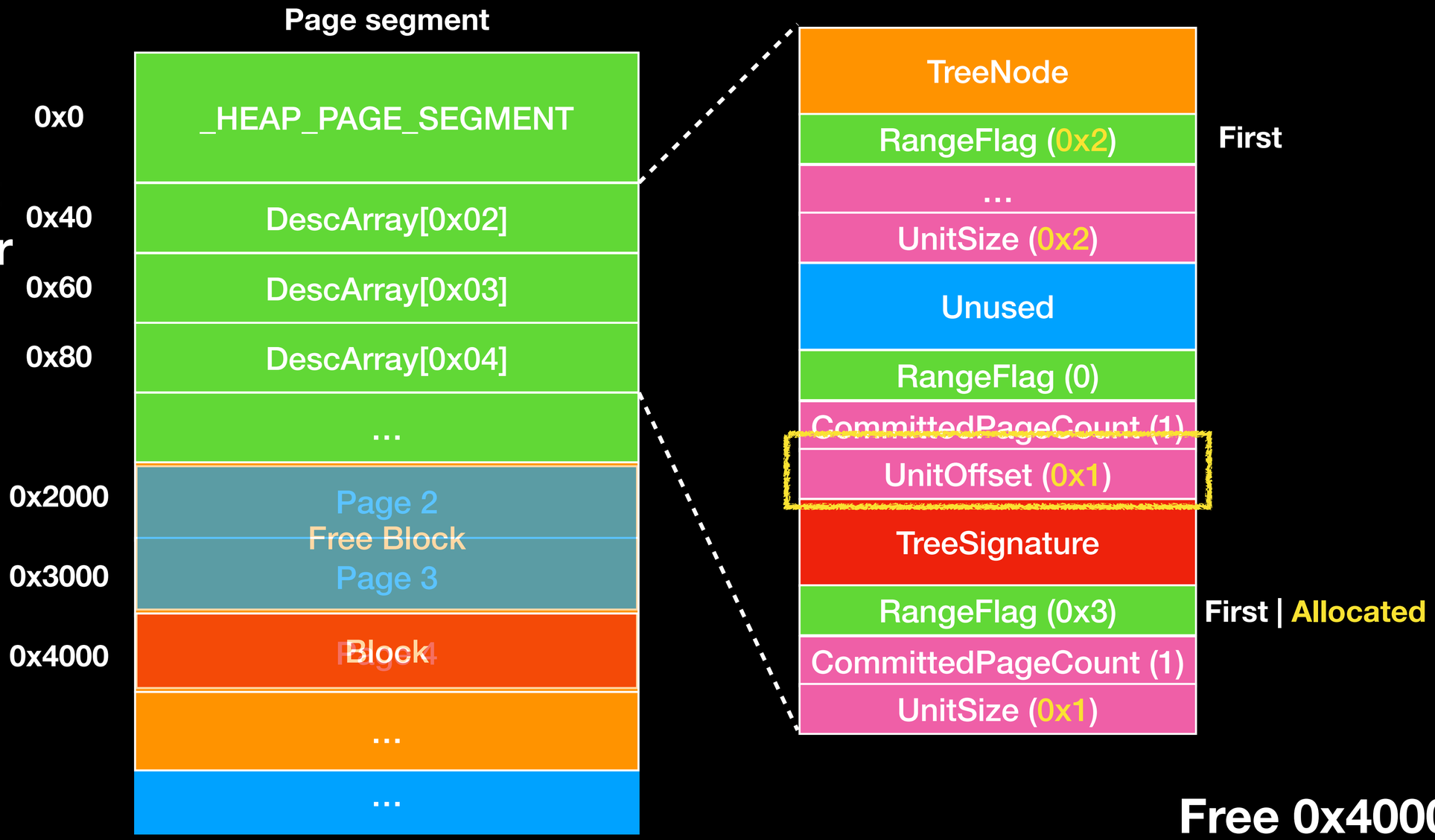
Check allocated and first bit of Prev page descriptor



Free 0x4000

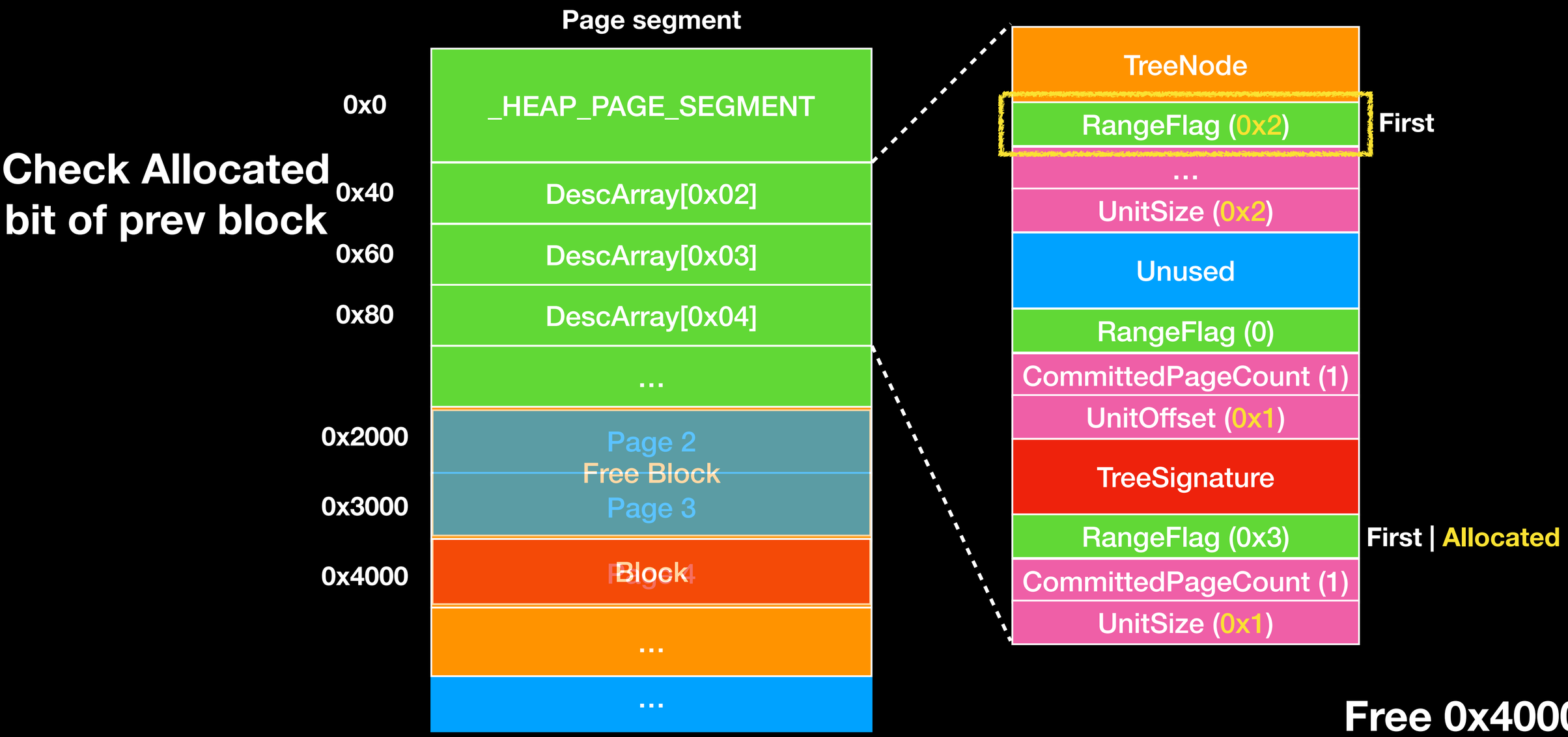
Segment Allocation

Use UnitOffset to find the first page descriptor of prev block

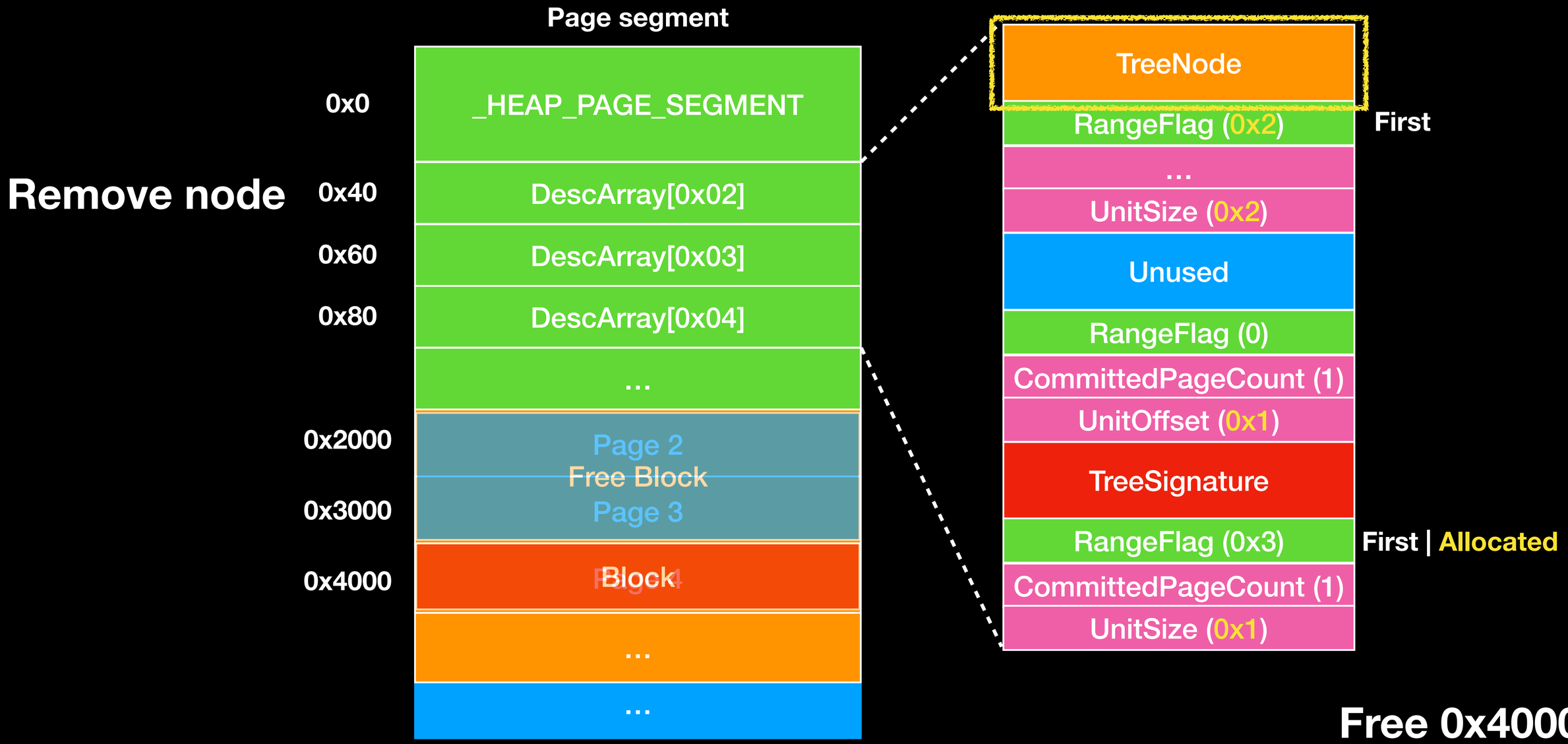


Free 0x4000

Segment Allocation

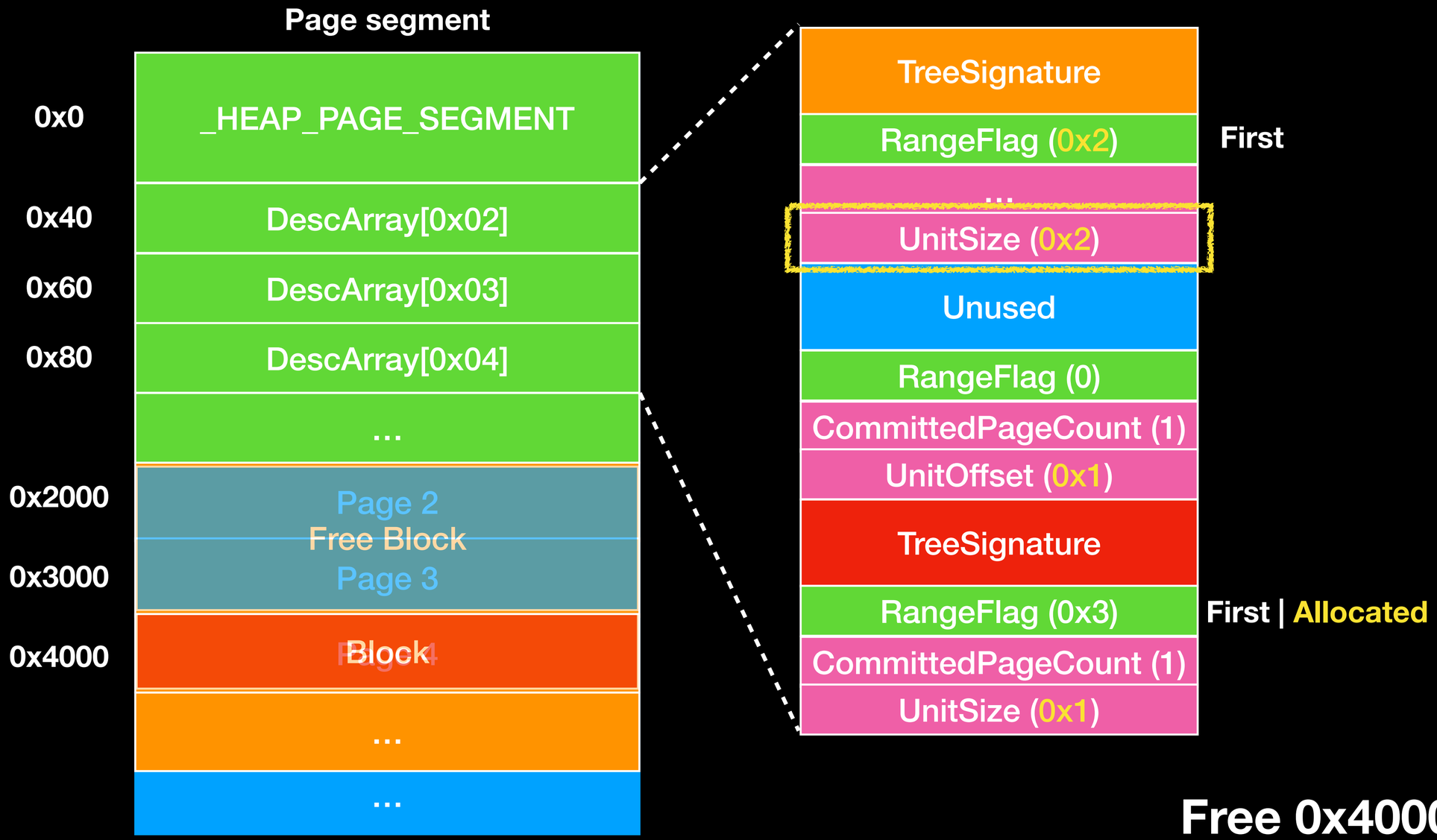


Segment Allocation



Segment Allocation

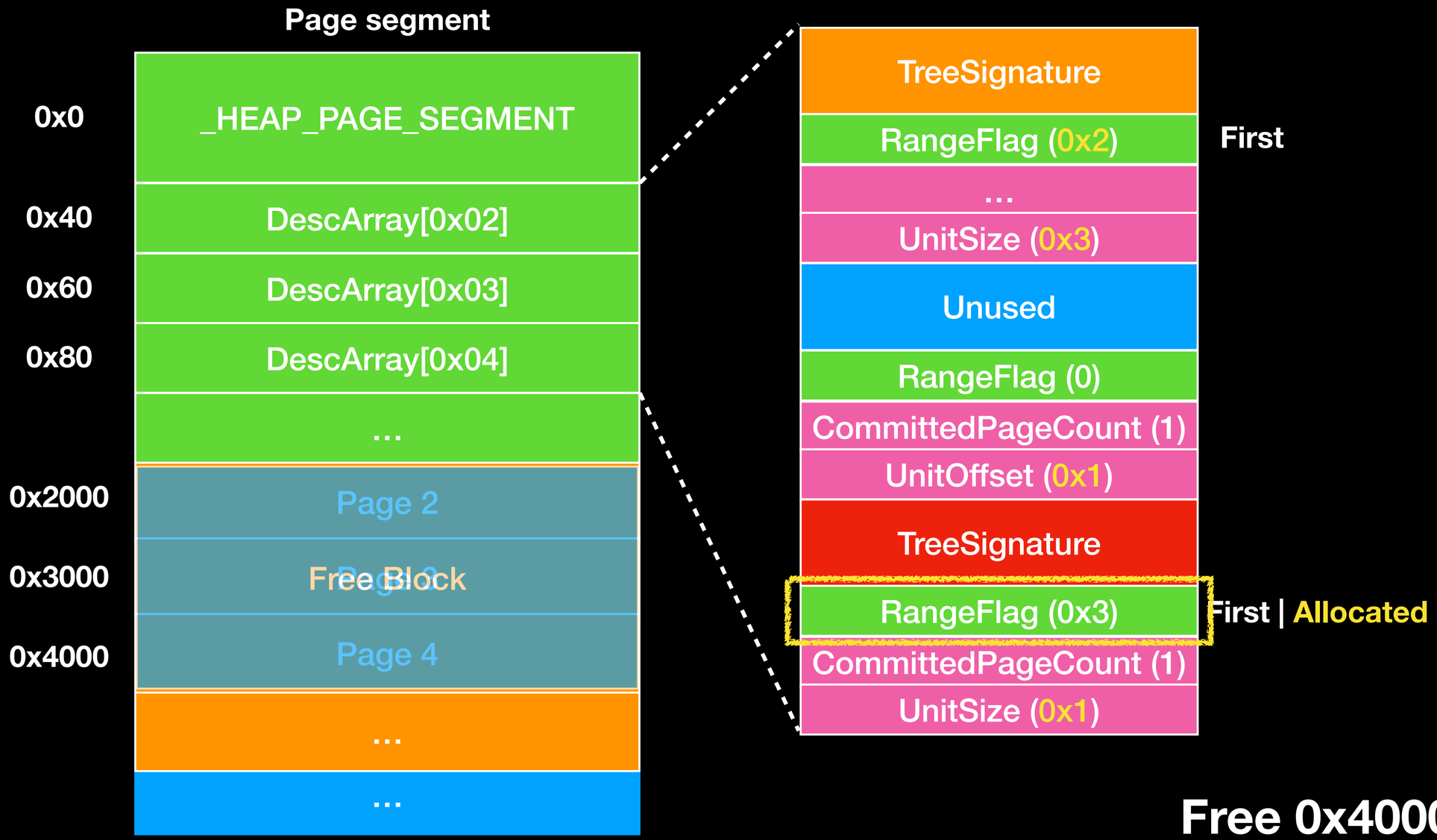
Update size of prev block



Free 0x4000

Segment Allocation

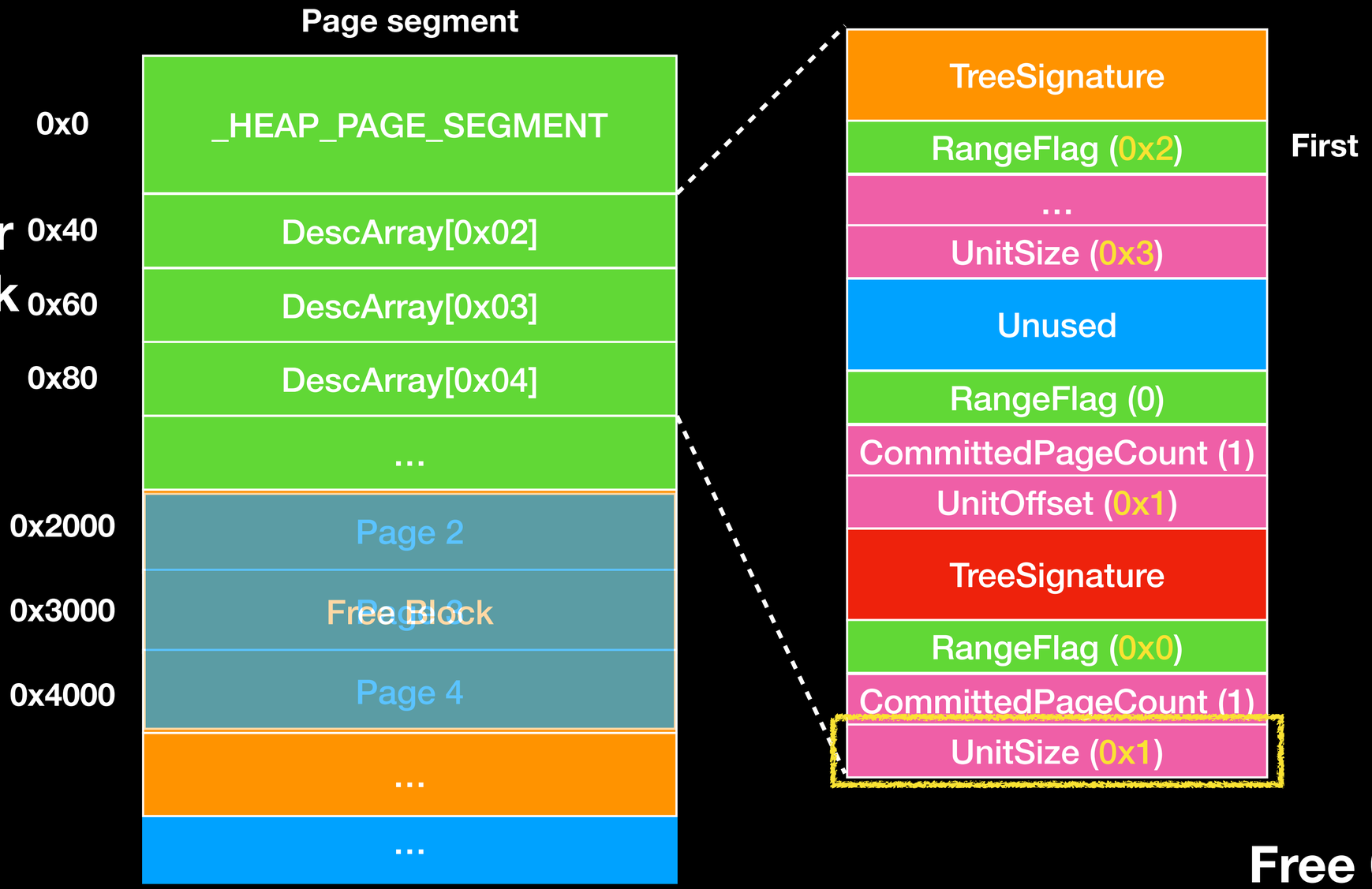
Clean Allcated and first bit of page descriptor



Free 0x4000

Segment Allocation

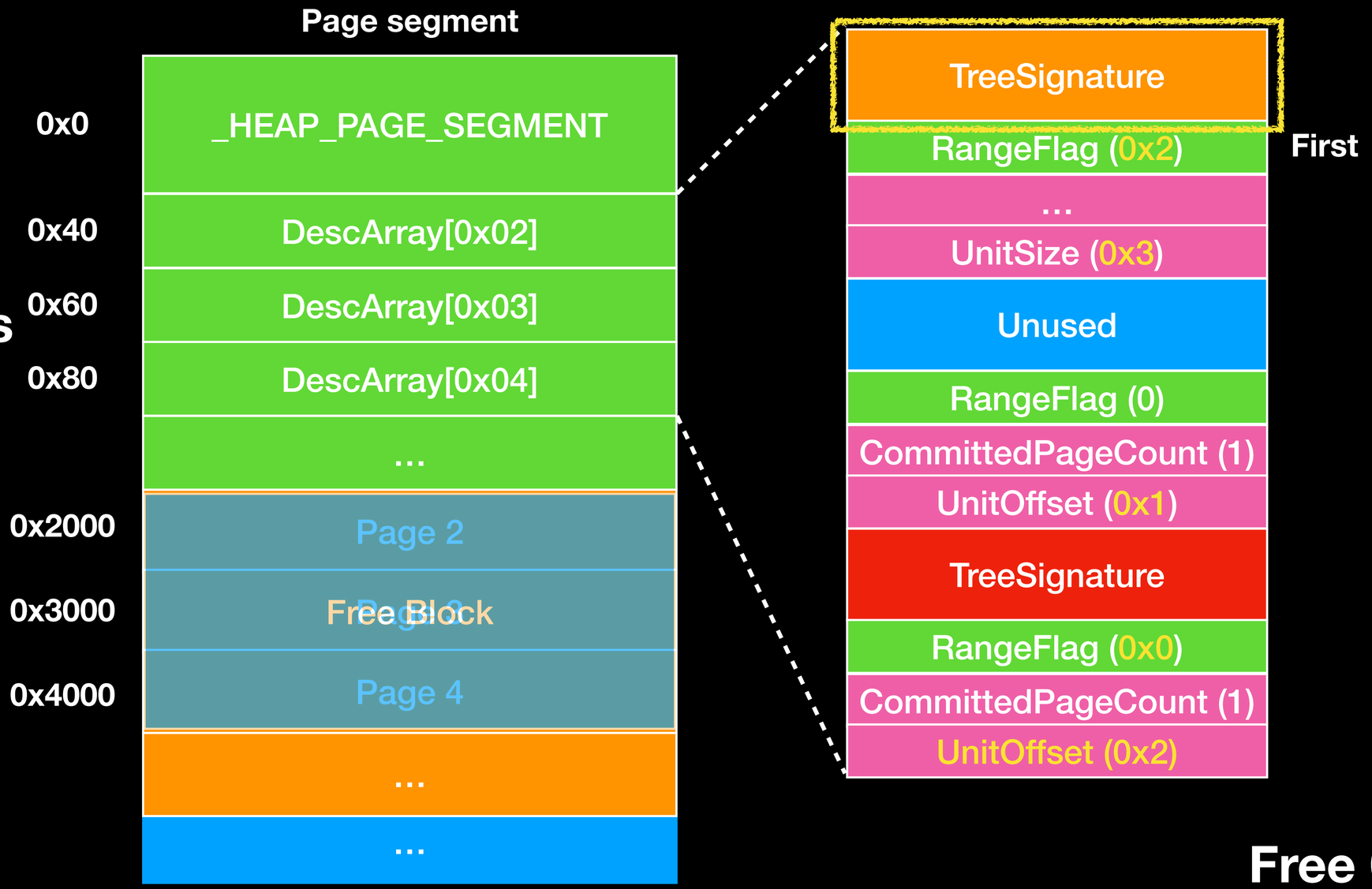
Set the last page descriptor of the free block



Free 0x4000

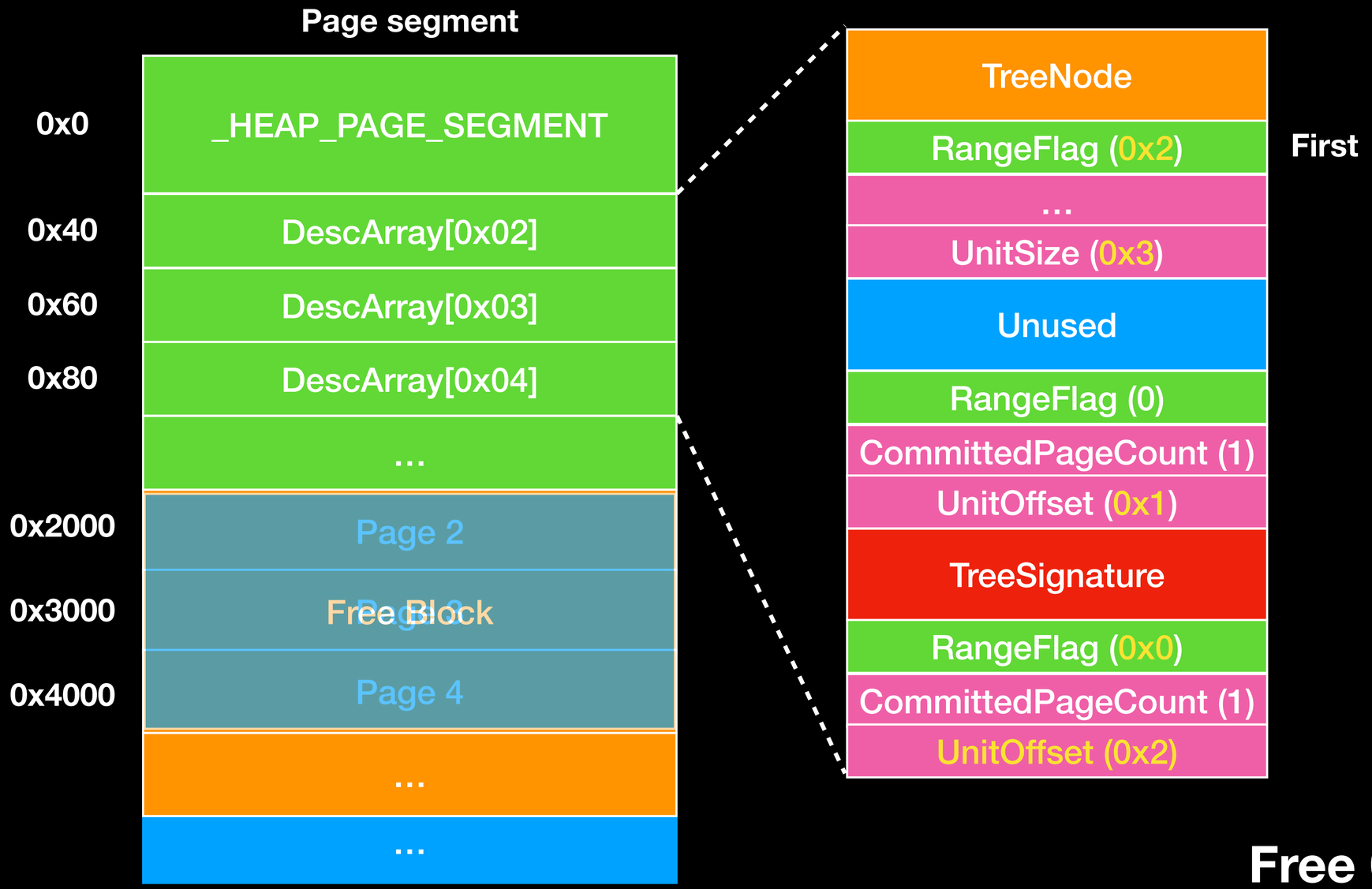
Segment Allocation

Insert the page descriptor to FreePageRages



Free 0x4000

Segment Allocation



Free 0x4000

Memory Allocator in kernel

- Segment Heap
  - Frontend Allocation
    - Low FragmentationHeap
    - Variable Size Allocation
  - Backend Allocation
    - Segment Allocation
  - Large Block Allocation

Large Block Allocation

- Size
  - Size > 0x7f0000

Large Block Allocation

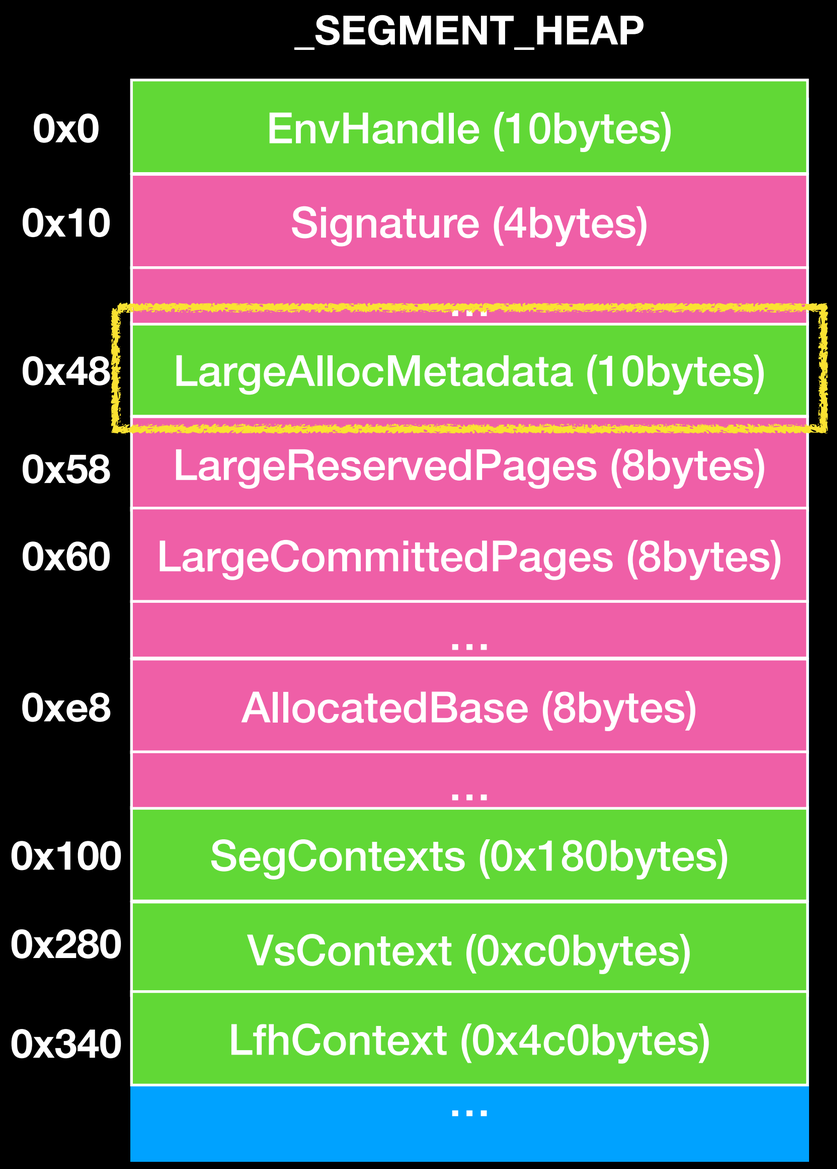
- Only use rbtree to manage. Compared to other Allocations, it is much simpler. In fact, it almost directly allocate for a large block of memory from the system and stores it with rbtree.

- Release is also removed from rbtree and returned to the system directly.

Large Block Allocation

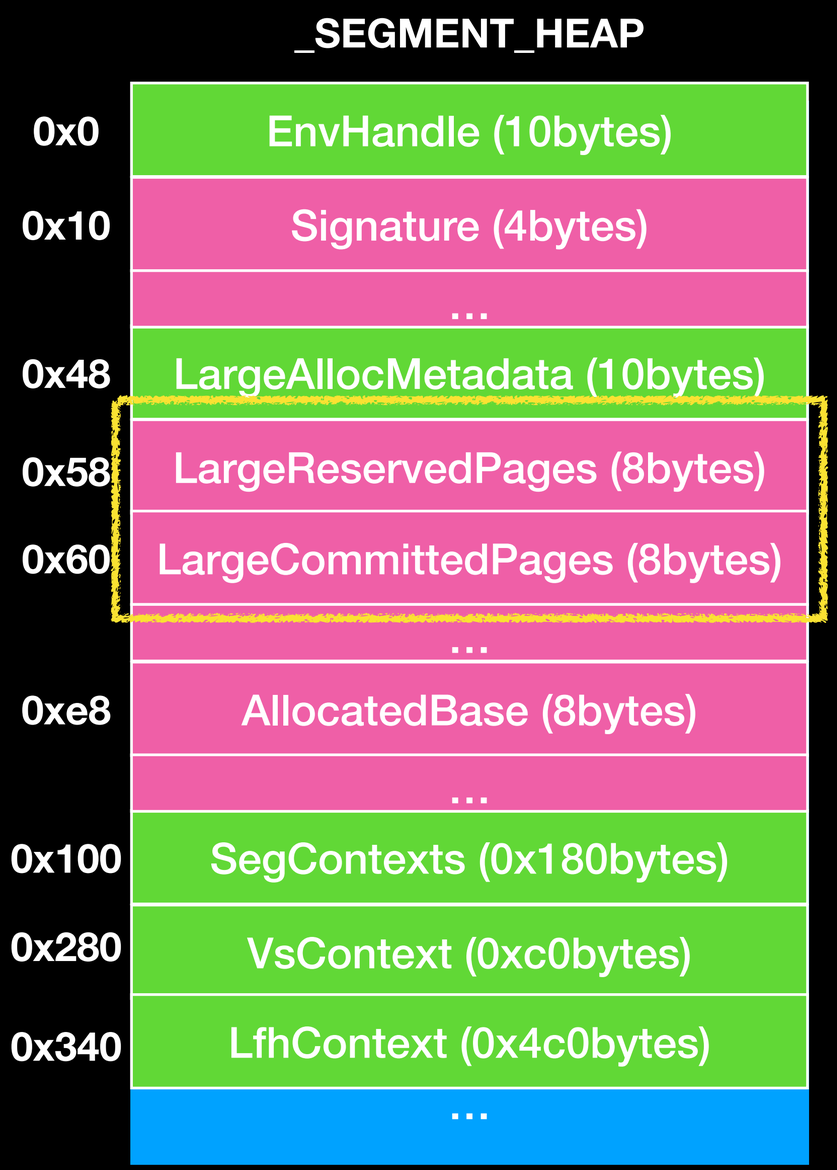
- Data Structure
- Memory allocation mechanism

Large Block Allocation



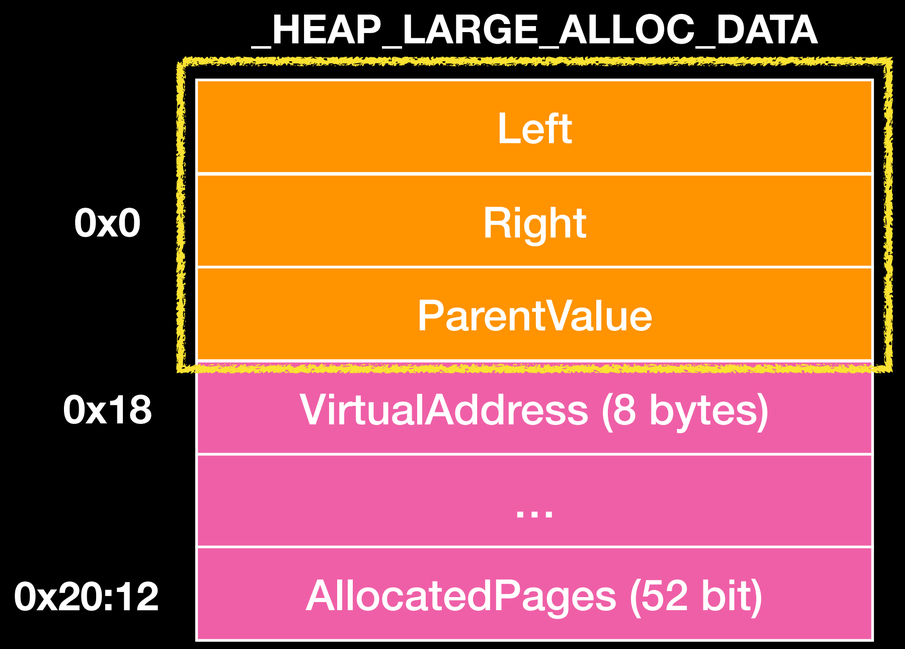
- _SEGMENT_HEAP
  - LargeAllocMetadata (_RTL_RB_TREE)
    - Used to manage the allocated memory by Large Allocation
    - It is also a rbree structure, and the address of allocated memory is used as KEY

Large Block Allocation



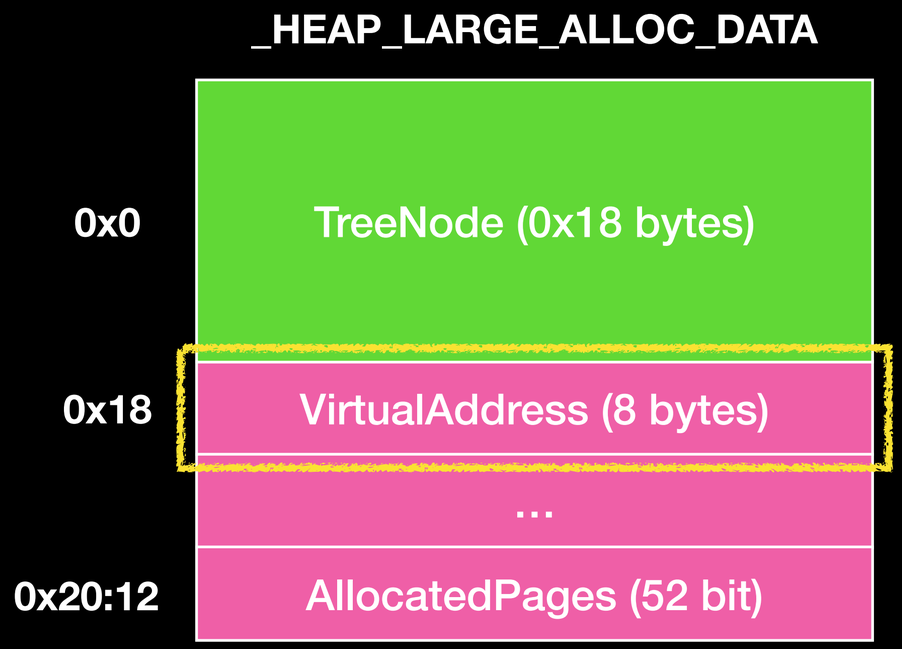
- _SEGMENT_HEAP
  - LargeReservedPages
    - Number of pages reserved for Large Block Allocation
  - LargeCommittedPages
    - Number of committed pages for Large Allocation

Large Block Allocation



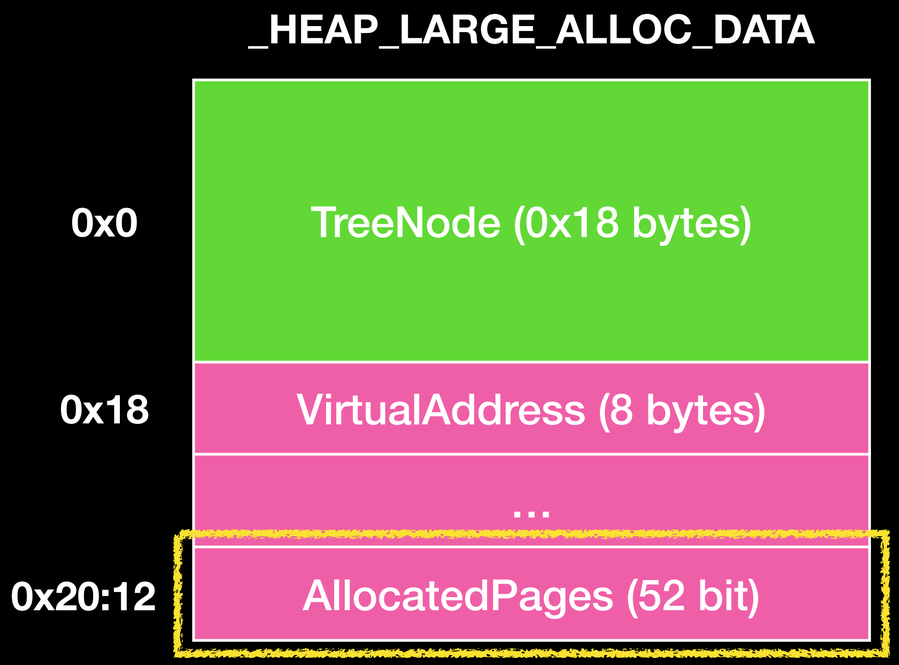
- _HEAP_LARGE_ALLOC_DATA
  - TreeNode (_RTL_BALANCED_NODE)
    - Left
      - Point to a node whose VirtualAddress is smaller than the node
    - Right
      - Point to a Node whose VirtualAddress is greater than the Node
    - ParentValue is pointed to parent node
      - The lowest 1 bit will determine whether to encode Parent node

Large Block Allocation



- _HEAP_LARGE_ALLOC_DATA
  - VirtualAddress
    - The lowest 16 bits as Unusedbytes
    - Address of large block

Large Block Allocation



- _HEAP_LARGE_ALLOC_DATA
  - AllocatedPages
    - The number of allocated pages

Large Block Allocation

- Data Structure
- Memory allocation mechanism

Large Block Allocation

- Allocate
  - The allocation is based on page as the unit.
  - Main implementation function is nt!RtlpHpMetadataAlloc
    - At the beginning, it will allocate memory for storing Large block Metadata (_HEAP_LARGE_ALLOC_DATA)
      - RtlpHpMetadataHeapCtxGet & RtlpHpMetadataHeapStart
        - It will determine which Heap to allocate from according to the value of Segment heap->EnvHandle
        - ExPoolState->HeapManager.MetadataHeaps[idx]

Large Block Allocation

- Allocate
  - The allocation is based on page as the unit.
  - Main implementation function is nt!RtlpHpMetadataAlloc
    - Next, it will use RtlpHpAllocVA to allocated memory, and then store the VirtualAddress in Metadata
    - Finally, insert the Metadtata into Segment heap->LargeAllocMetadata

Large Block Allocation

- Free
  - Main implementation function is (RtlpHpLargeFree)
    - First find the node correspond to the free ptr from Segment heap->LargeAllocMetadata, and remove the node
    - Use RtlpHpFreeVA to release memory
    - Finally release the memory storing Metadata (RtlpHpMetadataFree) of the released memory

Reference

- Segment Heap
  - https://www.blackhat.com/docs/us-16/materials/us-16-Yason-Windows-10-Segment-Heap-Internals.pdf
  - https://www.sstic.org/media/SSTIC2020/SSTIC-actes/pool_overflow_exploitation_since_windows_10_19h1/SSTIC2020-Article-pool_overflow_exploitation_since_windows_10_19h1-bayet_fariello.pdf

About Part 2

- We will talk about window kernel pool exploit
  - Way to spray and defragment in different heap
  - Metadata corruption in segment heap
  - …etc