A .NET Object from allocation to collection

Maoni Stephens

09/28/2021

Our very first object

```
public static int Main()
{
    // class MaoniType has 2 int fields
    MaoniType o = new MaoniType(128, 256);
    // more stuff
    return 0;
}
```

We want to see our very first object in memory

How do we do that?

```
public static int Main()
{
    MaoniType o = new MaoniType(128, 256);
    Console.ReadLine();
    // more stuff
    return 0;
}
```

I'll use the SoS extension in windbg

We break into the debugger when ReadLine is waiting

```
0:008> !dumpheap -stat
Statistics:
              MT    Count    TotalSize Class Name
00007ff9f3cc3478        1           24 System.Threading.Tasks.Task+<>c
00007ff9f3cad2d8        1           24 System.IO.Stream+NullStream
00007ff9f3ca7b60        1           24 System.IO.SyncTextReader
00007ff9f3ca3598        1           24 MaoniType
[omitted]
00007ff9f3bbb590        7        18056 System.Object[]
00007ff9f3cc4128        1        32792 System.Char[]
00007ff9f3c6d448       56        69466 System.String
Total 168 objects
```

Finding our object in memory

```
0:008> !dumpheap -type MaoniType
         Address               MT                     Size
000002b1447e3848     00007ff9f3ca3598       24

Statistics:
              MT          Count    TotalSize          Class Name
00007ff9f3ca3598            1           24          MaoniType
Total 1 objects
```

How are objects laid out in memory

```
0:008> !eeheap -gc
Number of GC Heaps: 1
generation 0 starts at 0x000002b1447d1030
generation 1 starts at 0x000002b1447d1018
generation 2 starts at 0x000002b1447d1000
ephemeral segment allocation context: none
         segment            begin        allocated        committed  allocated size    committed size
000002b1447d0000 000002b1447d1000 000002b1447edfb0 000002b1447f2000  0x1cfb0(118704)  0x21000(135168)
[omitted]
```

Our object is at 000002b1447e3848 (000002b1447e3848-generation 0 start = 12818)

(000002b1447edfb0 - 000002b1447e3848 = 42856)

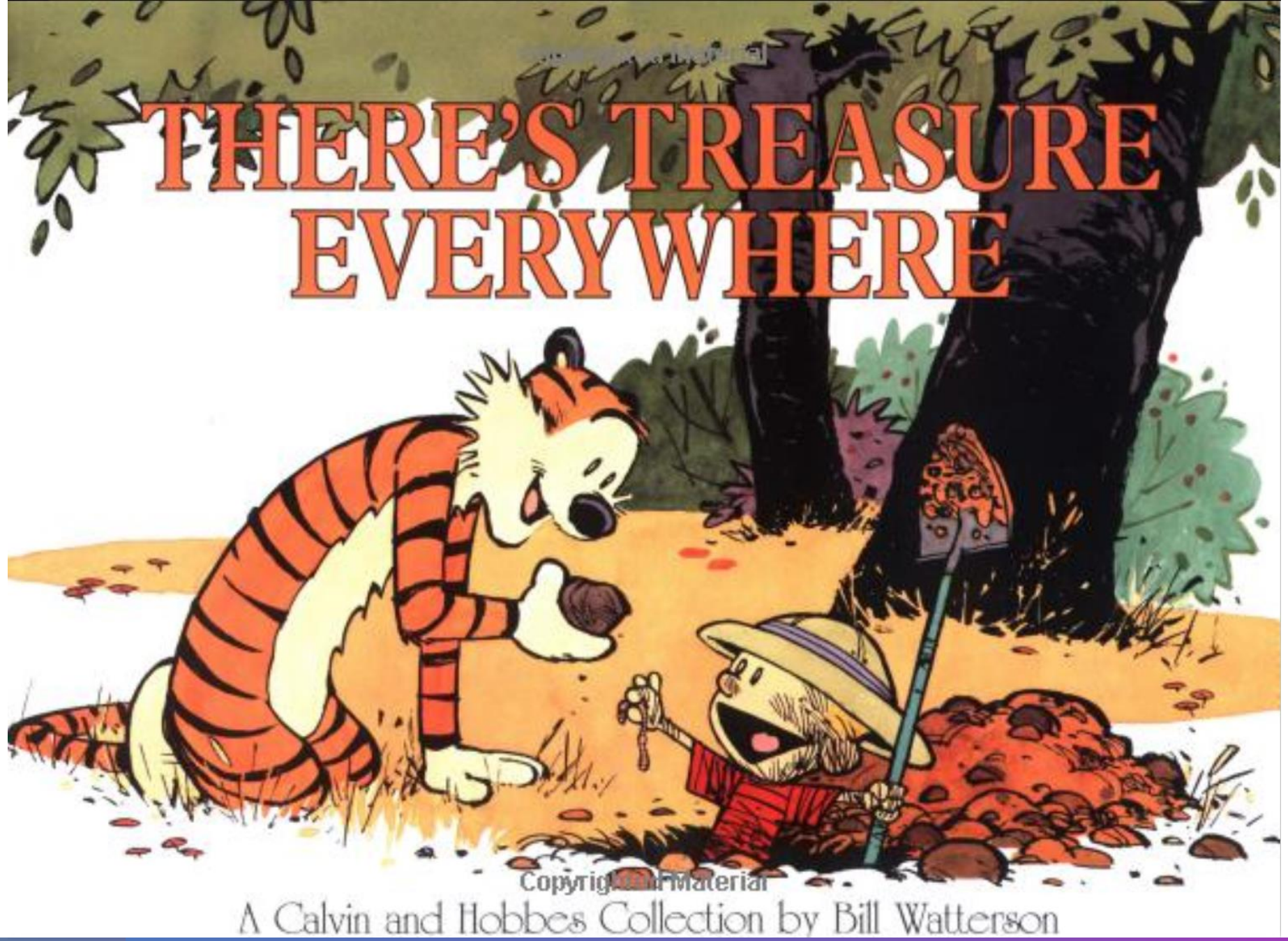
How are segments created?

- What is a segment?
  - It's a unit of memory the GC acquires from the OS
  - What does "acquire" mean? How to acquire?
  - How does the OS acquire memory to give to us?

Let's start from the beginning

- What I talk about here is mostly the simplified views
- I will explain the relevant parts, how they fix together and demystify some terms
- Links throughout the slide deck so you can get more details
- You can safely assume that there's optimization everywhere

- Parallelism
- Speculative execution
- On demand
- Caching



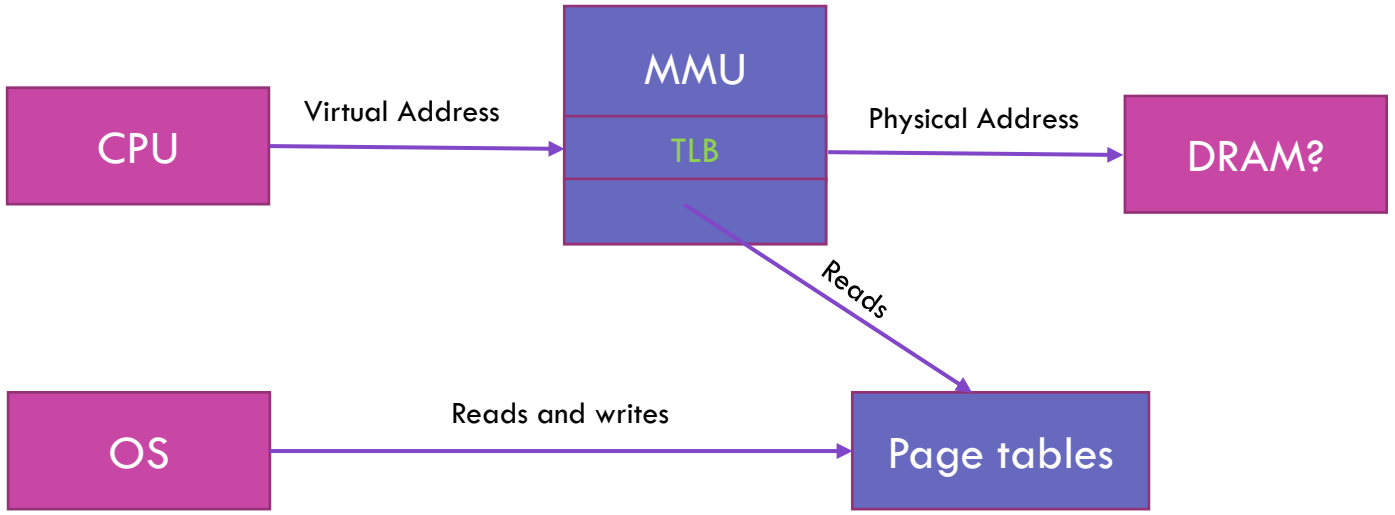
How do we use DRAM?

- Also referred to as physical memory or main memory
- We don't use physical memory directly
- We use VM (Virtual Memory)
  - Provided by the VMM (Virtual Memory Manager, or MM) in the OS
- Being able to use VM is really useful!
  - Every process thinks it has its own memory space
    - Enables both isolation and sharing
  - You can ask for more than what you use
    - Only pay for what you use with physical memory
  - You can have more virtual memory than your physical memory
  - A contiguous VM range doesn't need the physical memory to be contiguous

How does the OS implement VM?

- Hardware implements a mechanism called *paging* which allows the OS to implement virtual memory
  - Provided by the MMU (Memory Management Unit)
  - Memory divided in pages (usually 4k)
  - Virtual to physical page mapping in page tables (that the MMU walks)
  - When a virtual page doesn't have a valid corresponding physical page, a page fault occurs
  - Control is transferred to the OS to provide a physical page (or an AV)
    - Demand paging
- How to make it fast?
  - TLB (Translation Lookaside Buffer) and page table caching
- Many videos on youtube, eg, this series
- Intel Software Developer Manual Vol 1 Chap 3.3

Interacting with the MMU

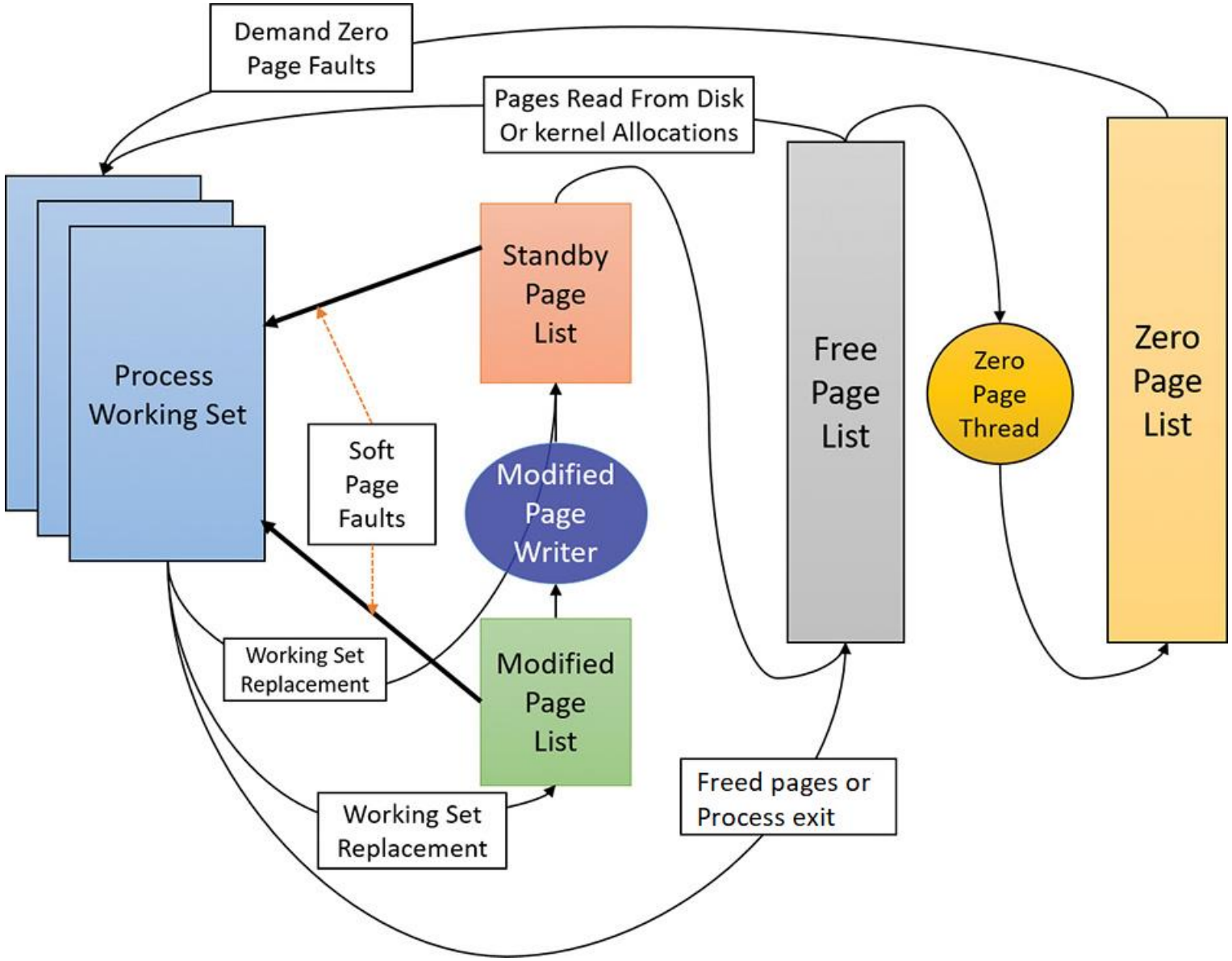


How are physical pages organized

- When the machine is booted, the OS puts all physical pages from DRAM on a list
- When a physical page is allocated for a process to use, it becomes part of its WS (Working Set)
- When a physical page is removed from WS, it can be returned either via a soft page fault or a hard page fault
- Hard page faults are much more expensive so we try to avoid them!
  - This is what folks refer to as “started paging”
- Avoiding them means we do NOT want to grow the heaps larger than the physical memory
  - Watch out for the physical memory load (physical memory in use / total)

Page transition

(From the "Windows Internals" book)



How does the GC acquire VM?

- Because of demand paging, we can ask for a range of address that we might use later
  - This is called reserve (VirtualAlloc with MEM_RESERVE)
  - At this point you cannot store any data yet
- When we need to store data in pages, tell the OS about them
  - This is called commit (VirtualAlloc with MEM_COMMIT)
  - Commit guarantees you will not get OOM when you need to use this space
- Reserve is fast (relatively – you still incur a user <->kernel mode transition)
- Commit is fast (relatively) at the time you call it… till you actually store data
  - A page fault occurs and physical memory is allocated at this time
- Finally, we can store some data on this page!

Quick recap

- Reserved
- Committed
- WS/physical memory (could be much smaller than Committed)

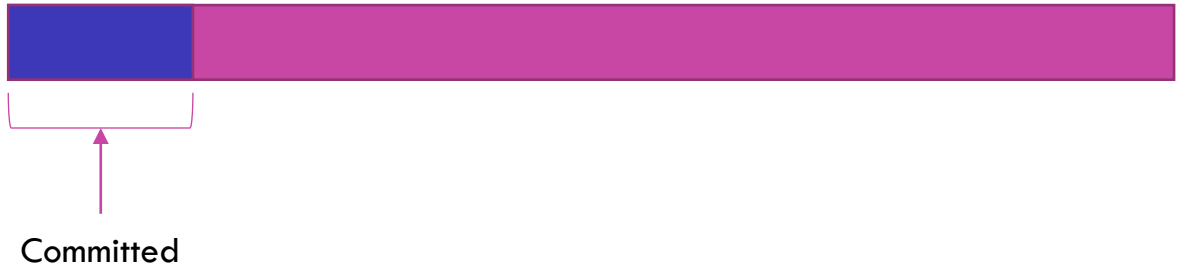
Windows Internals, Memory Management Chapter

Going back to our segment

- Reserve memory for a segment
  - A large amount of space
  - No objects are stored yet

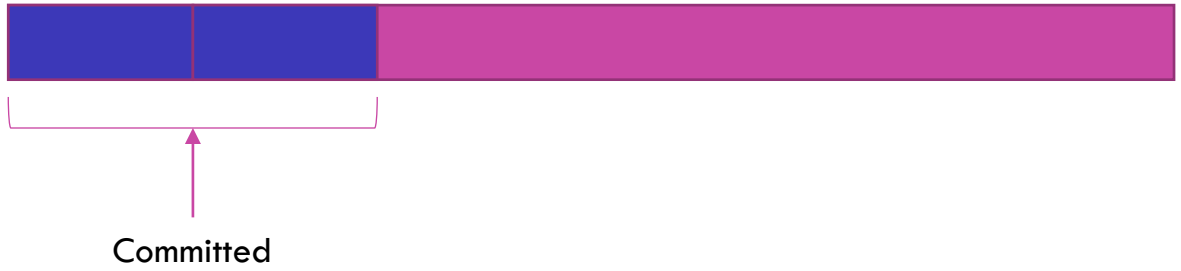
Going back to our segment

- Need to store data on the segment! So we commit
  - Always commit the first 1 page since we need to store segment info
  - Commit as needed (usually 64k), decommit as needed



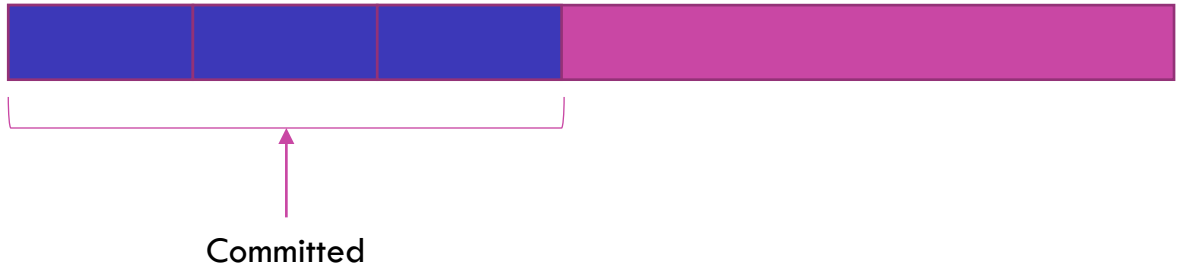
Going back to our segment

- Need to store data on the segment! So we commit
  - Always commit the first 1 page since we need to store segment info
  - Commit as needed for objects (usually 64k)



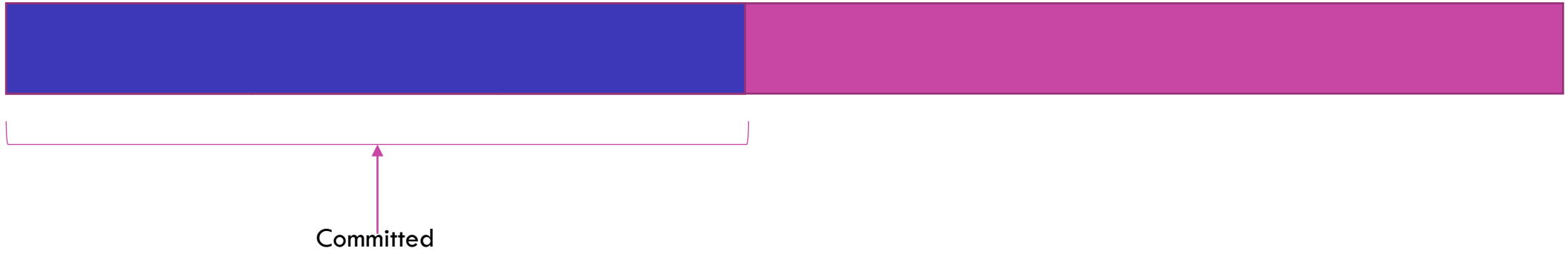
Going back to our segment

- Need to store data on the segment! So we commit
  - Always commit the first 1 page since we need to store segment info
  - Commit as needed for objects (usually 64k)



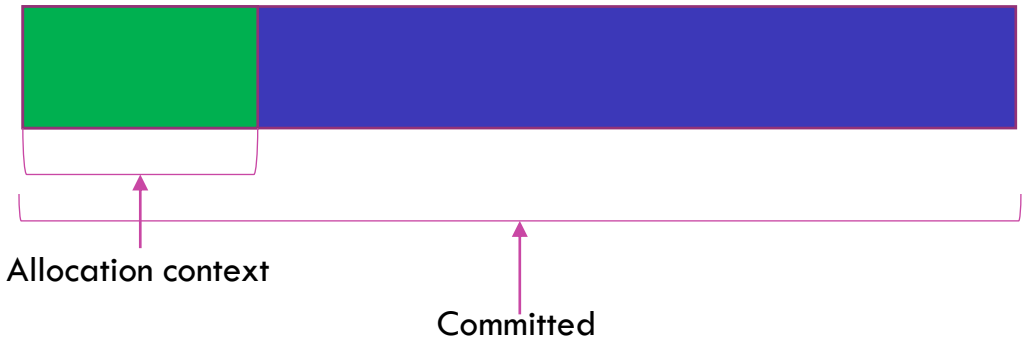
Constructing the MaoniType instance

- GC does not give out memory for just one object (not always true)
  - It gives out an allocation context (a few KB)
  - No objects are constructed yet



Constructing the MaoniType instance

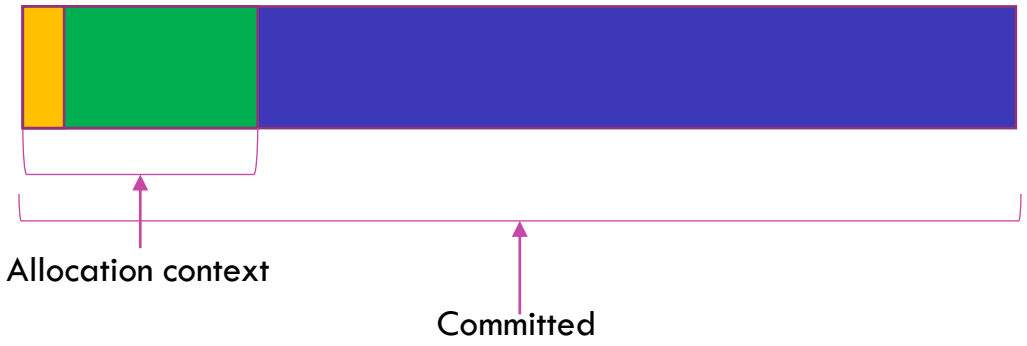
- GC does not give out memory for just one object (not always true)
  - It gives out an allocation context (a few KB)
  - No objects are constructed yet



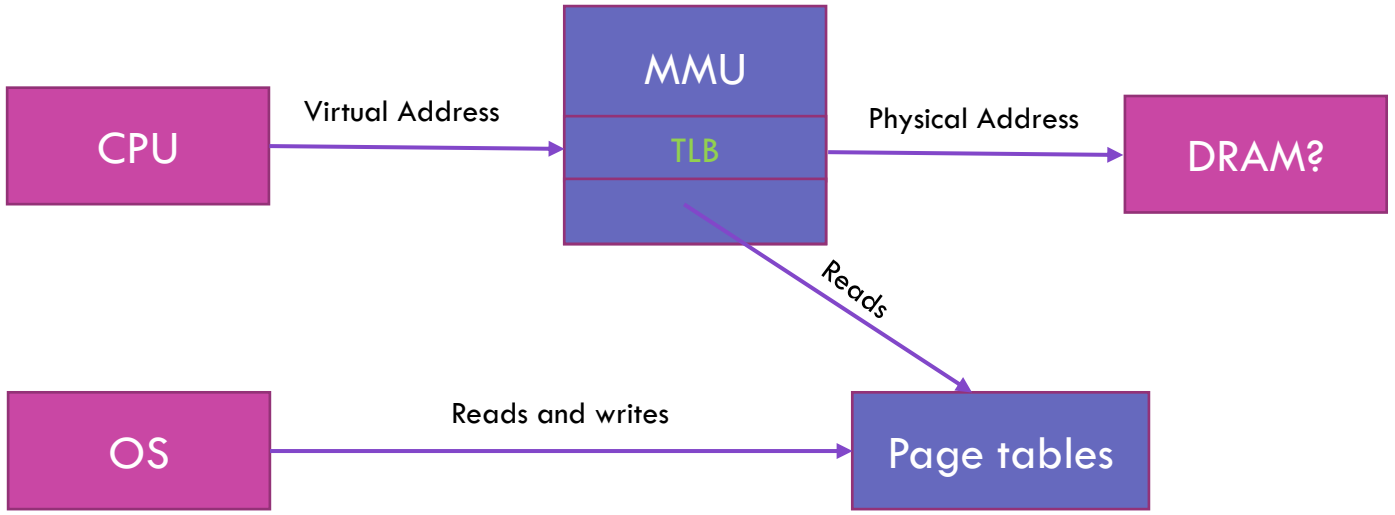
Constructing the MaoniType instance

- Touch memory to write
  - A physical page is allocated and added to this process's WS

```
0:008> dq 000002b1447e3848-8 l3
000002b1`447e3840  00000000`00000000 00007ff9`f3ca3598
000002b1`447e3850  00000100`00000080
```

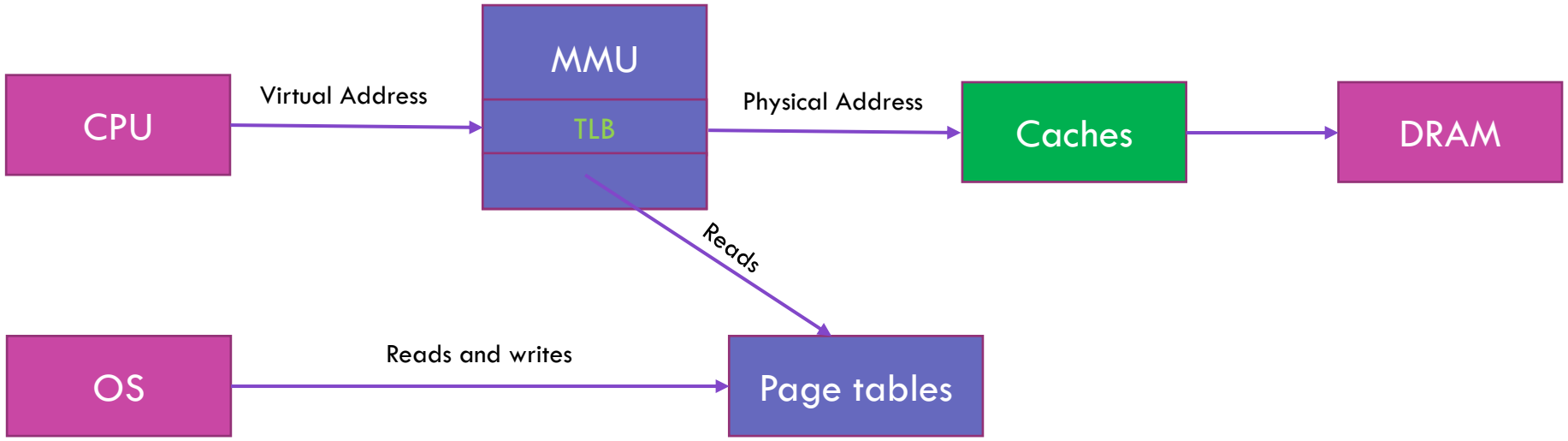


Interacting with the MMU



Speeding things up

- Note that this is an **extremely** simplified view



Caches are much faster

- Skylake cache latency (best case!)
  - L1 4 cycles
  - L2 12 cycles
  - L3 44 cycles (LLC – Last Level Cache)

(From Intel's Optimization Reference Manual)

- DRAM 60ns – 100ns

Why is cache latency not const

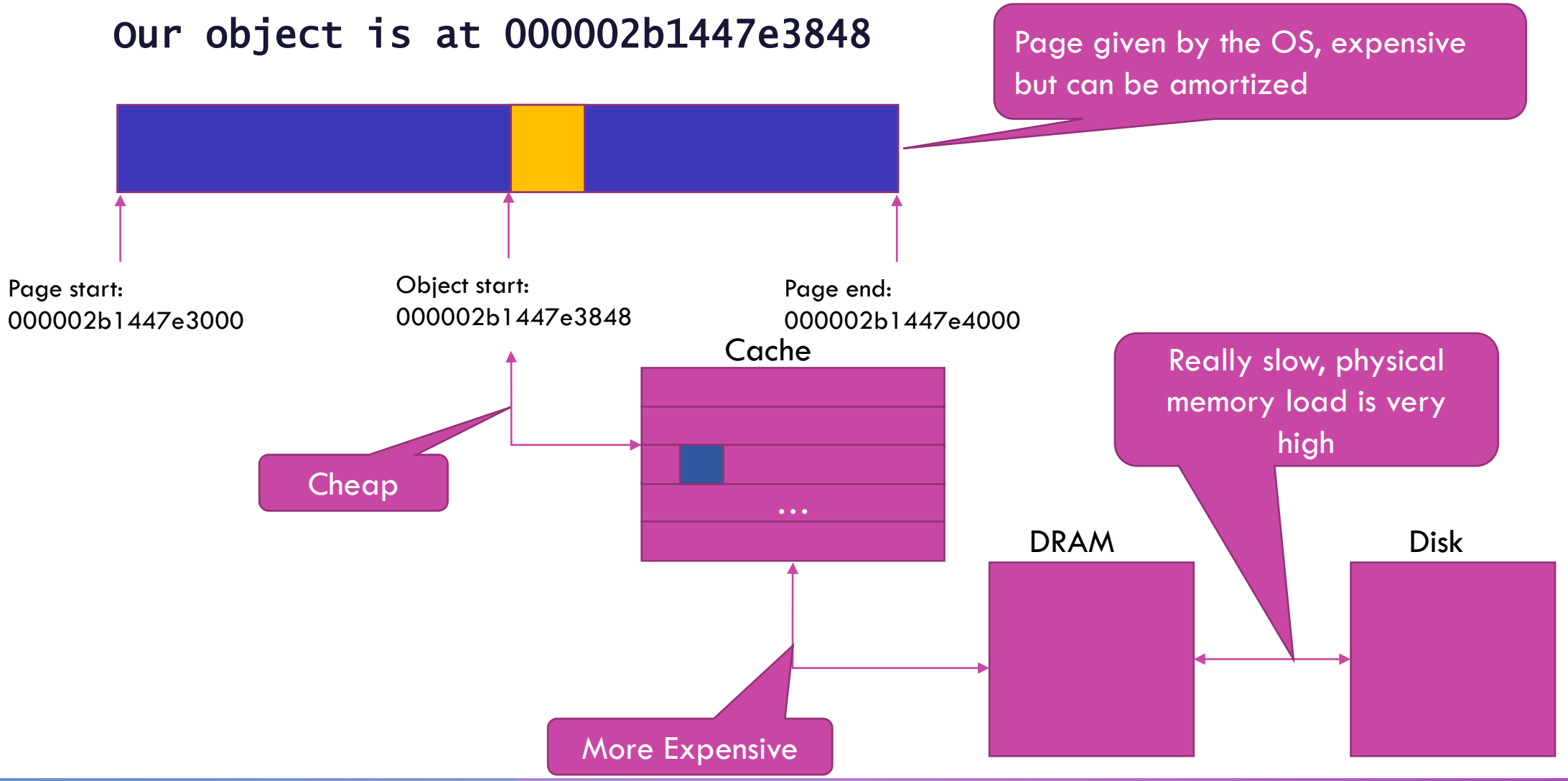
- Caches for different cores need to be coherent
- If a cache line only exists in one cache, it's an easy case (relatively)
- If it exists in multiple caches, how do you update the value?
- Some cache coherency protocol needs to be implemented
  - MESI is a very common one
- It's expensive to update a cache line when shared by multiple caches!

When will the cache line be written to main memory?

- If a cache line is modified, it will need to be written to main memory (at some point)
- As long as it's in a cache, the CPU can proceed
- It's up to the cache when/how to write the line back (as long as it does not lose the write)
- Write policy: write back (more common) vs write through
- Finally, we stored data on our page!

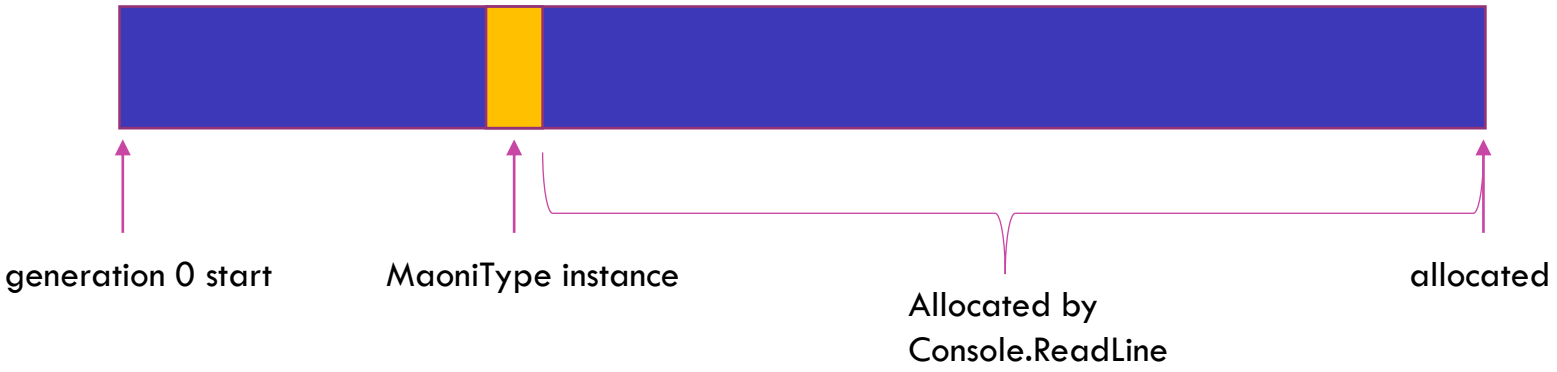
Back to the MaoniType instance

Our object is at 000002b1447e3848



Back to the MaoniType instance

```
0:008> !eeheap -gc
Number of GC Heaps: 1
generation 0 starts at 0x000002b1447d1030
generation 1 starts at 0x000002b1447d1018
generation 2 starts at 0x000002b1447d1000
ephemeral segment allocation context: none
         segment            begin         allocated         committed    allocated size     committed size
000002b1447d0000  000002b1447d1000  000002b1447edfb0  000002b1447f2000   0x1cfb0(118704)   0x21000(135168)
Our object is at 000002b1447e3848 (000002b1447e3848-generation 0 start = 12818)
(000002b1447edfb0 - 000002b1447e3848 = 42856)
```



Back to GC land

- You allocate…
- And allocate some more…
- After you've allocated enough

Yay, I collect!!!

How does GC determine if it should collect our object?

```
public static int Main()
{
    MaoniType o = new MaoniType(128, 256);
    GCHandle h = GCHandle.Alloc(o, GCHandleType.Weak);
    GC.Collect();
    Console.WriteLine("Collect called, h.Target is {0}",
                (h.Target == null) ? "collected" : "not collected");
    return 0;
}
```

Output - Collect called, h.Target is not collected

“Why is GC not collecting my object???”

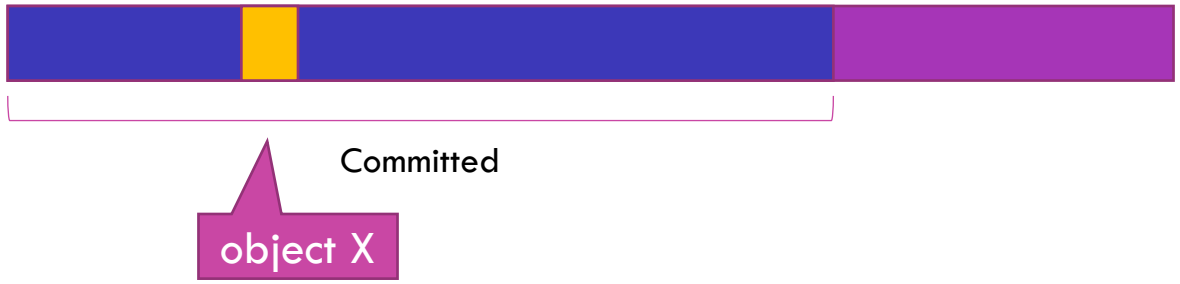
- Generational GC
- GC.Collect() collects the whole heap
  - This means GC does NOT decide the objects’ lifetime
  - GC only gets told by others if an object is live
  - In this case JIT tells the GC that o is still live
    - User roots
  - JIT is free to lengthen the object lifetime till the end of method and has always been

```
    [MethodImpl(MethodImplOptions.NoInlining)]
    public static void TestLifeTime()
    {
        MaoniType o = new MaoniType(128, 256);
        h = GCHandle.Alloc(o, GCHandleType.Weak);
    }

    public static int Main()
    {
        TestLifeTime();
        GC.Collect();
        Console.WriteLine("Collect called, h.Target is {0}",
                (h.Target == null) ? "collected" : "not collected");

        return 0;
    }
```

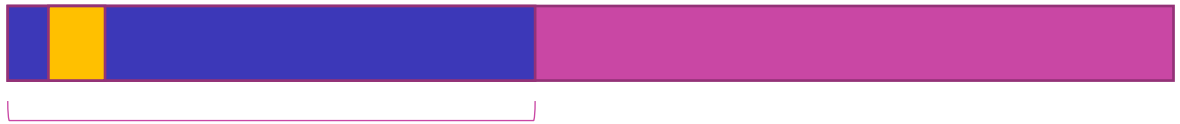
Output: Collect called, h.Target is collected

What the heap looks like

- GC#1 - start



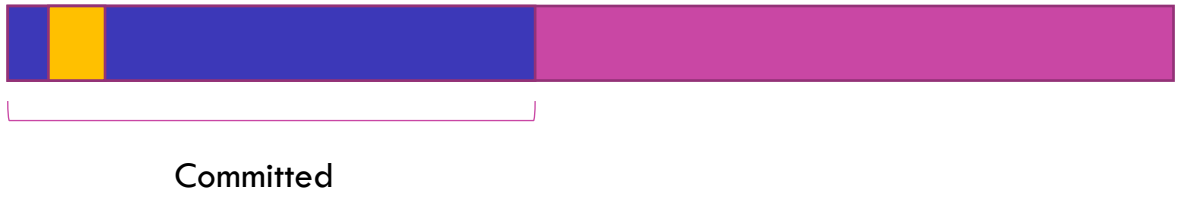
What the heap looks like



Committed

What the heap looks like

- GC#1 - end

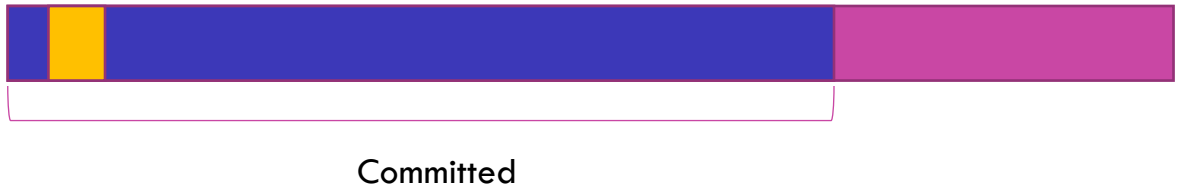


What the heap looks like

Committed

What the heap looks like

- GC#2 - start



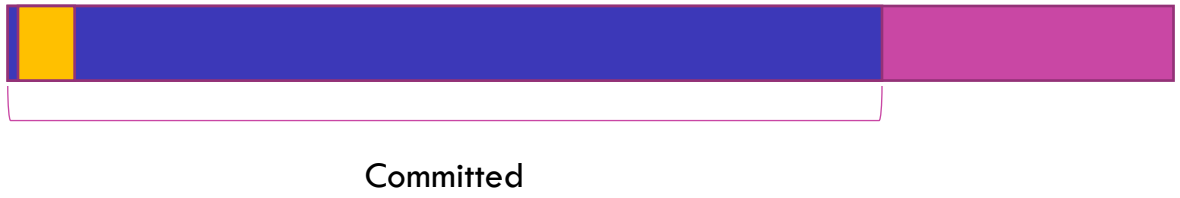
What the heap looks like

What the heap looks like

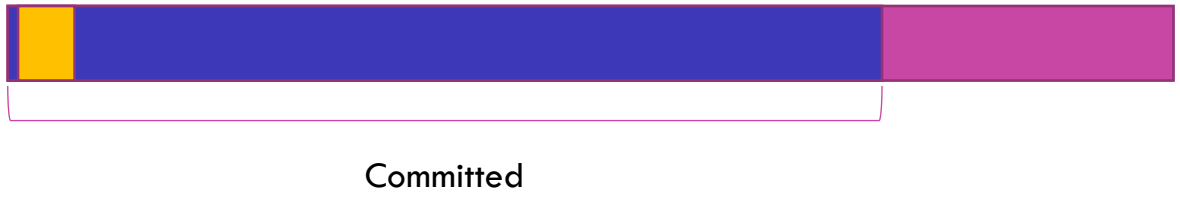
- GC#2 - end

Some GCs later...

- GC#N - start



Discovers the object is no longer rooted



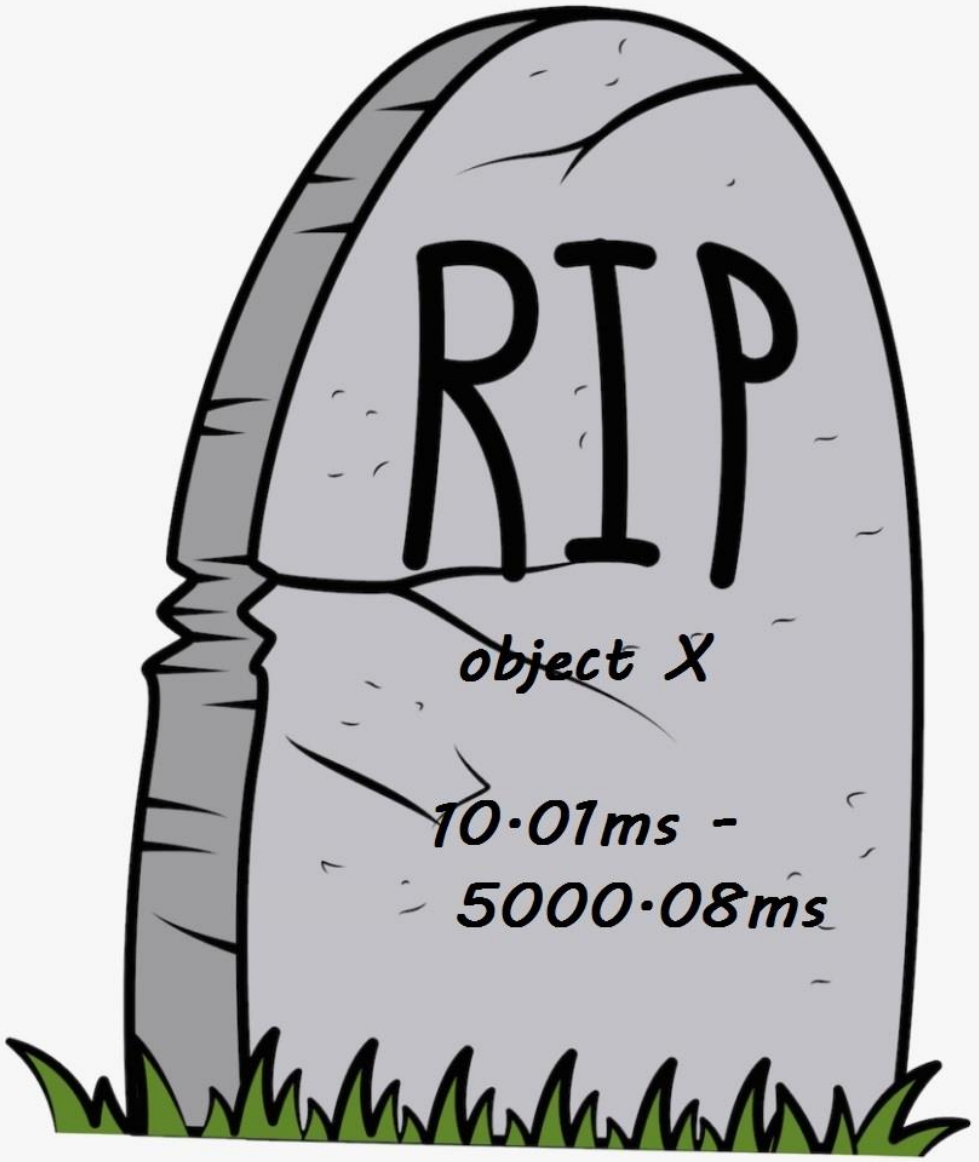
Object is collected

Committed

Object is collected

- GC#N - end

Committed



Question – when do you measure?

- Depending on when you measure, you can get VERY different results!
- Does it matter? That depends on the problem
  - The more obvious the problem is, the less it matters
  - Less obvious problems mean more careful/methodical measurement is required
- For less obvious problems, always start with a top level GC trace
  - It shows you the sizes at the start and the end of each GC
  - And a lot of other info about each GC

Question – do we need to change committed?

- We established that we want to amortize cost of getting pages we from the OS
- We don't actually keep committing and decommitting!
  - We don't decommit space we know we'll put objects in right away.
  - The top level GC trace will show you the heap size
  - Committed *is related to* the amount of physical memory we use

Where to get help?

- mem-doc
- File an issue on our GH repo and ask there!
- How to be efficient when you ask for help
  - Indicate the version of the runtime you are using
  - Describe the problem, impact and analysis you already did
  - Attach any perf data you can share