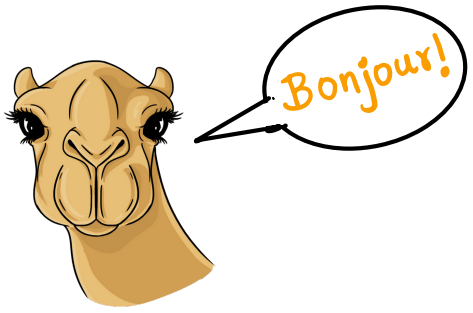
How Multicore GC Works



Meet our old fren Camel

He got cooler! OCaml is finally multicore \O/

OCaml came into existance in 1996 (older than me, what?! :O)

OCaml stood the test of time as a stable and performant FP language

Alas! OCaml didn't support multicore :(

For the longest time, OCaml didn't support native parallelism

The challenge?

Building a multicore capable Garbage Collector (GC) that's backwards compatible and performant

In came a group of hackers who took up this formidable task

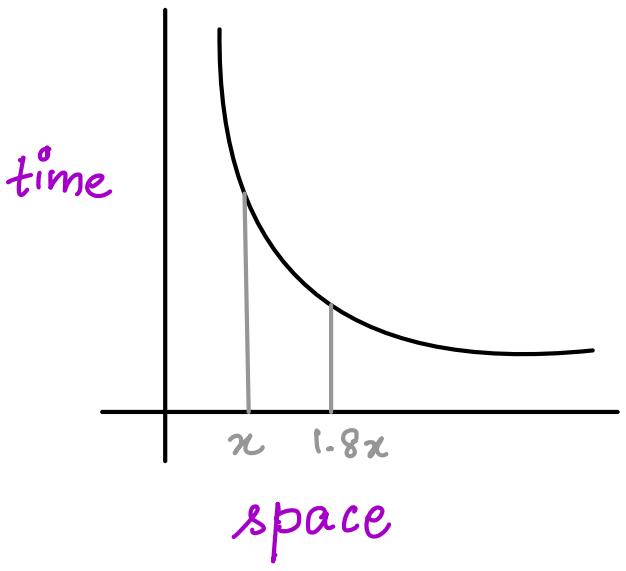
They set on this long and arduous journey of building a multicore GC

Multicore logo

It finally got merged and released as OCaml 5.0 in December 2022

Let's take a deep dive on the multicore GC

Space-time Trade off



* operating a GC is essentially choosing a trade-off b/w space and time
* x is the theoretical min amount of space required
* As the available space increases the program gets faster

* Of course, one doesn't have the luxury of infinte space

$\Rightarrow$ 1.8x is usually the sweet-spot *

* depends on a multitude of factors

But, first things first - how are OCaml values represented in the memory?

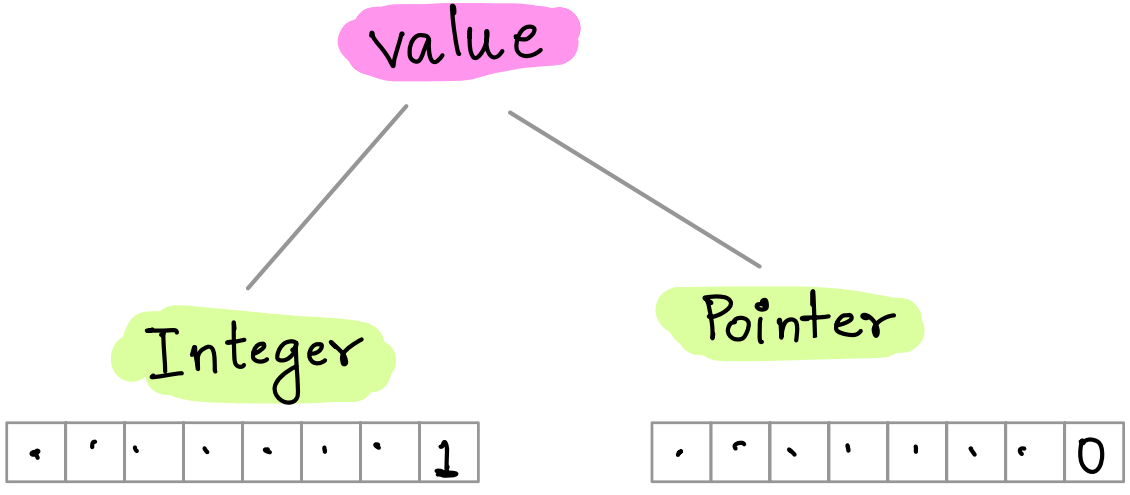
MEMORY REPRESENTATION OF VALUES

blocks of memory

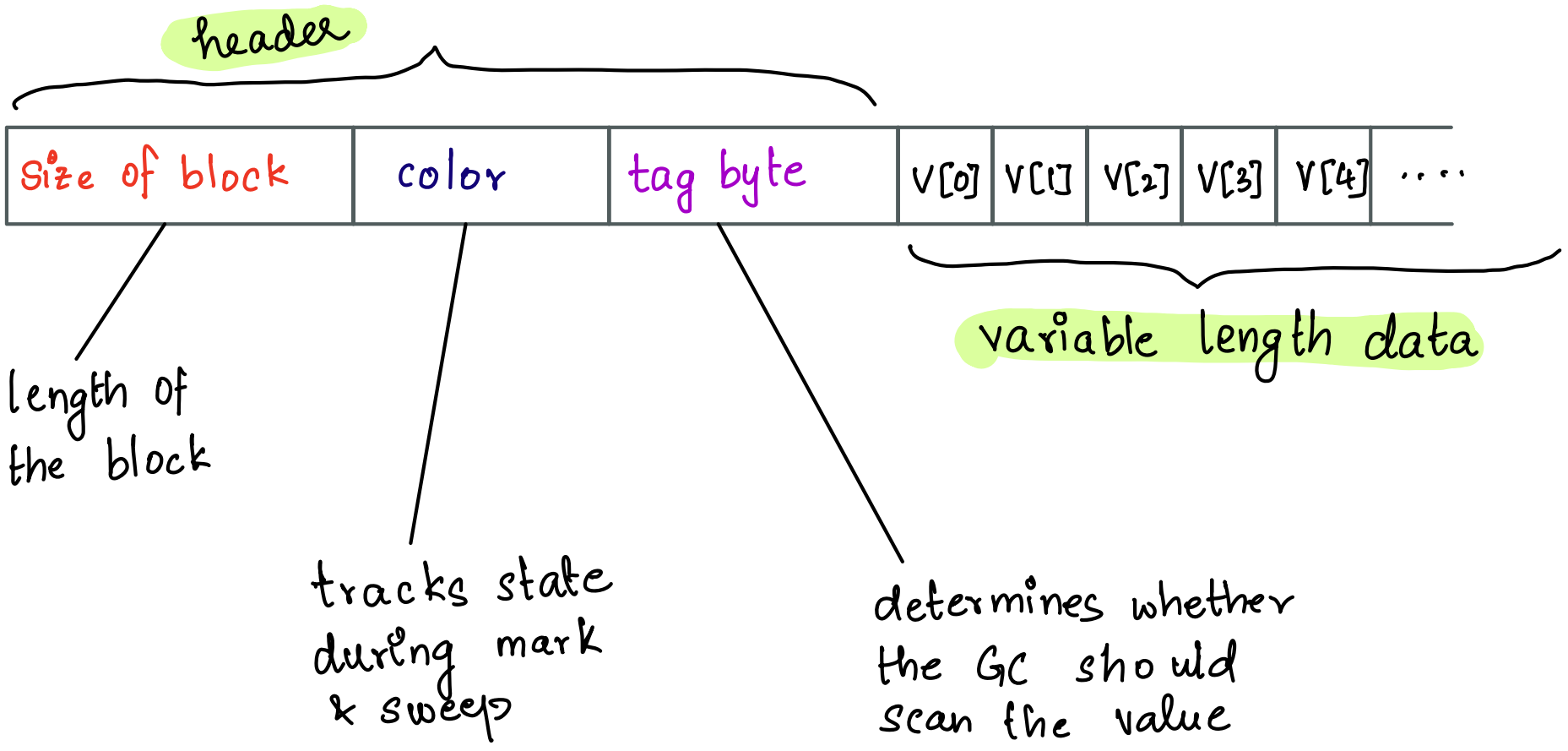
OCaml follows uniform memory layout ~ the layout is consistent regardless of type

Values are either integers (unboxed) or pointers (boxed)

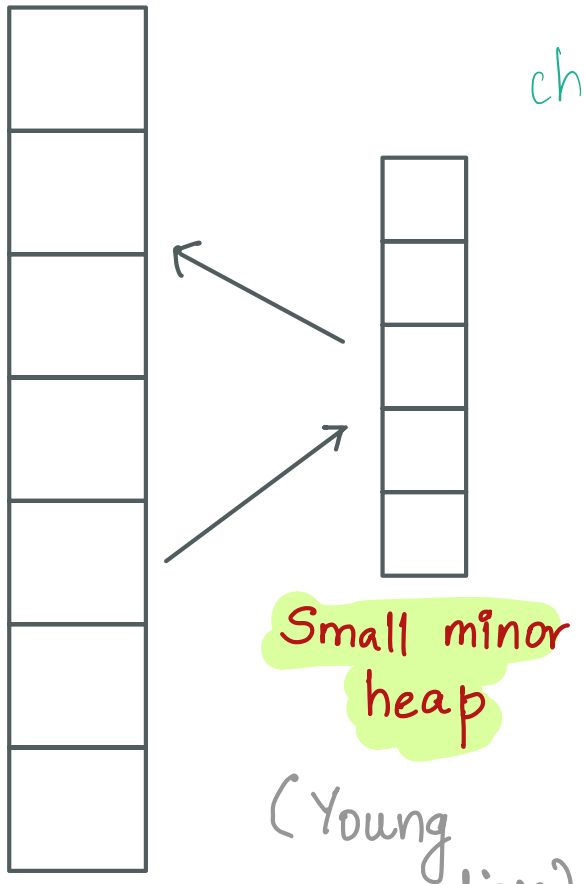
Boxed values have meta-data attached to it



Let's take a closer look at the constituents of a block:



GENERATIONAL HYPOTHESIS



- young blocks tend to die young
- old blocks tend to stay longer
- live blocks from minor heap are promoted to major heap

MAJOR HEAP: mark and sweep collection

MINOR HEAP: copying collection

KEY TAKEAWAYS

Allocations are fast: ~80% of allocations happen in minor heap

Pause times are small: single digit ms pausetimes at max GC memory is not exhausted

Survival rate is low: The survival rate of allocated objects is roughly 10% on the minor heap. Meaning, most allocated objects are collected. Small objects die young

To speedup your program: Allocate as little as possible!

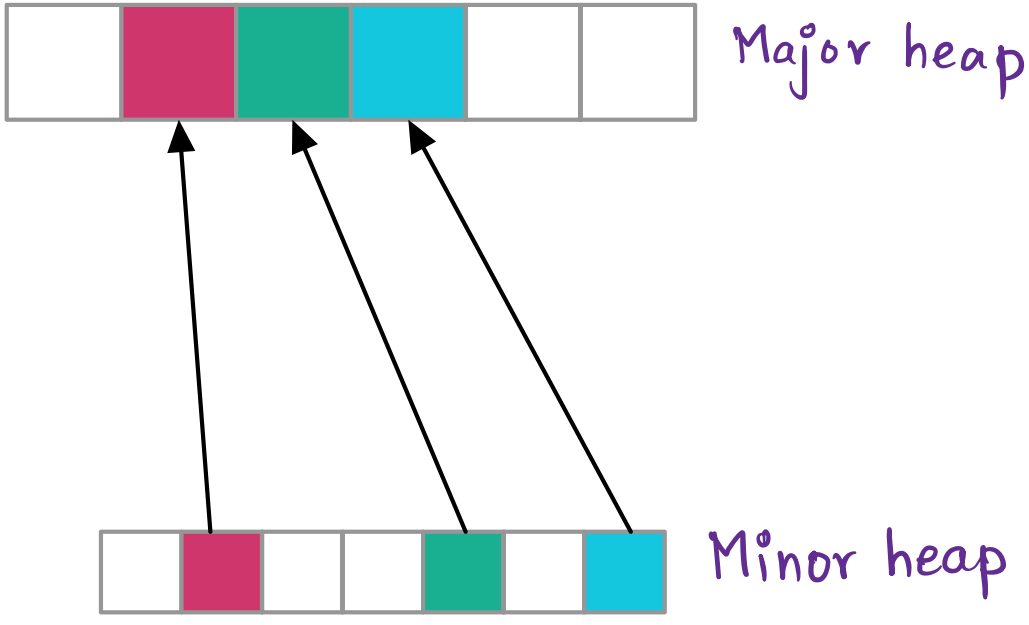
MINOR HEAP

The minor heap is one single blob of memory

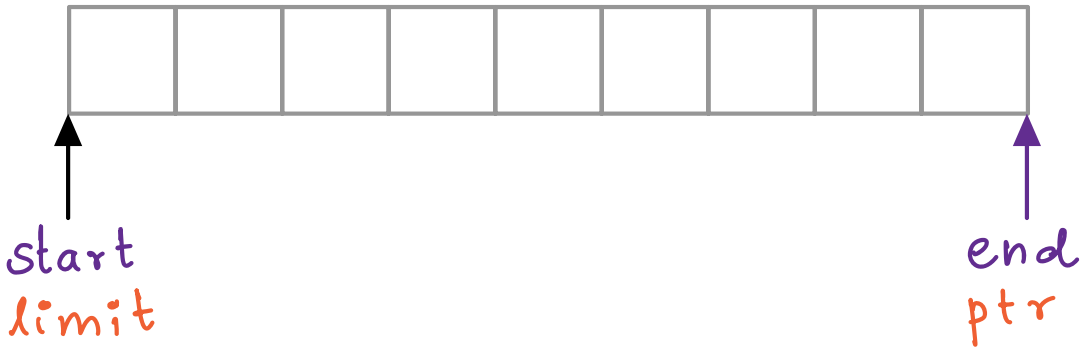
Allocation is constant time and fast

Collection is non-incremental: entire collection happens in one go

Live objects from minor heap are promoted to the major heap



Allocation



Note: all the pointers have a 'caml_young' prefix. They're dropped for brievity

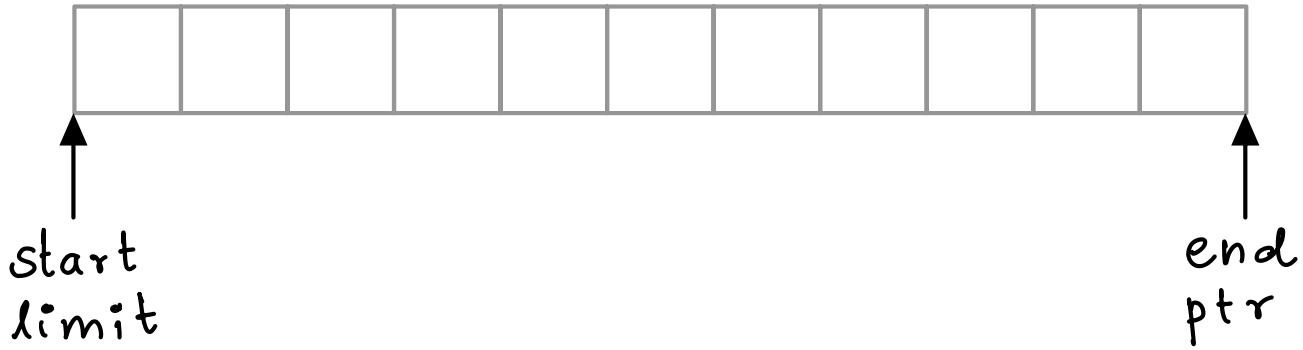
start: starting address of the minor heap

end: finishing address of the minor heap

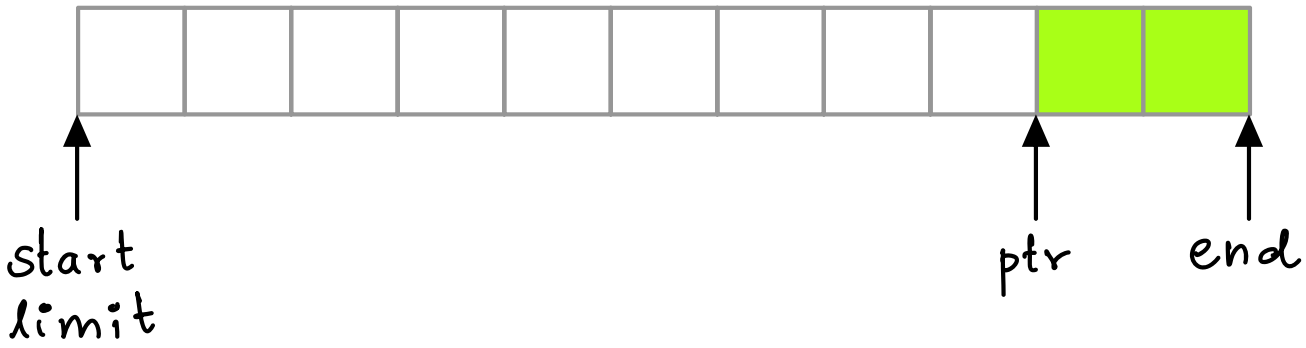
ptr: current location of the allocation pointer

limit: variable to define the end of minor heap. making limit = end triggers a collection

Initial state:

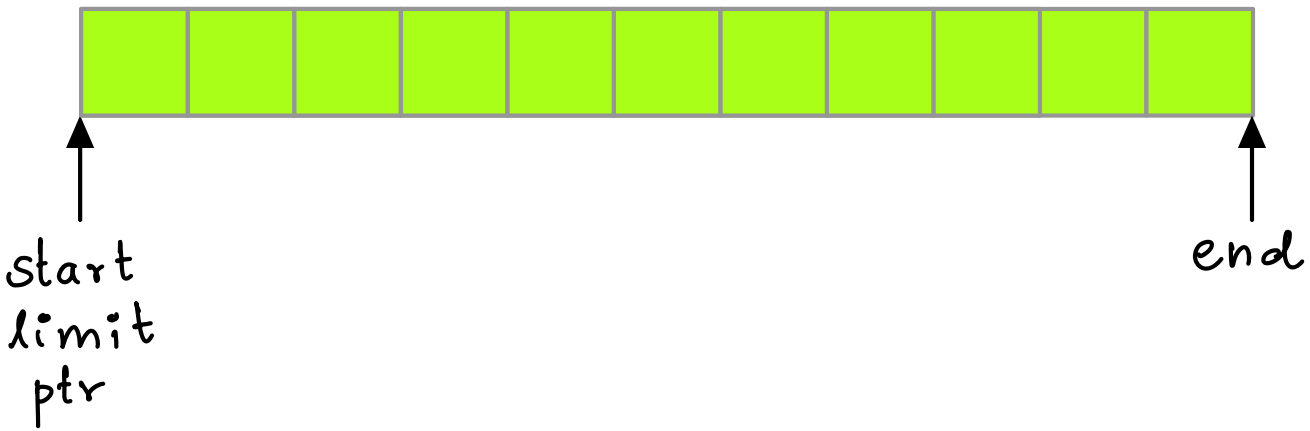


Allocate two blocks:



the pointer is bumped by two addresses

The minor heap is full:

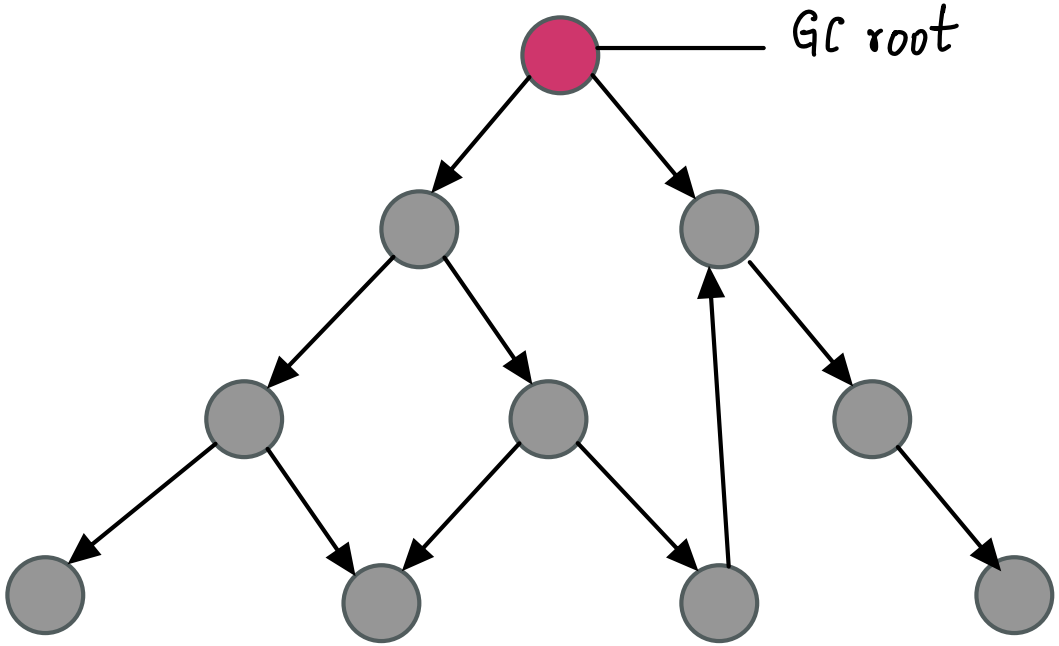


minor collection is triggered

Minor Collection

The collector traces live objects starting from a set of roots

Includes remembered set: retains pointers from major heap to minor heap



All reachable values from a root are traced via a breadth-first search

The live values are copied to the major heap (promotion)

Garbage values are untouched and the entire heap is cleared at the end of a minor cycle

In comes multicore

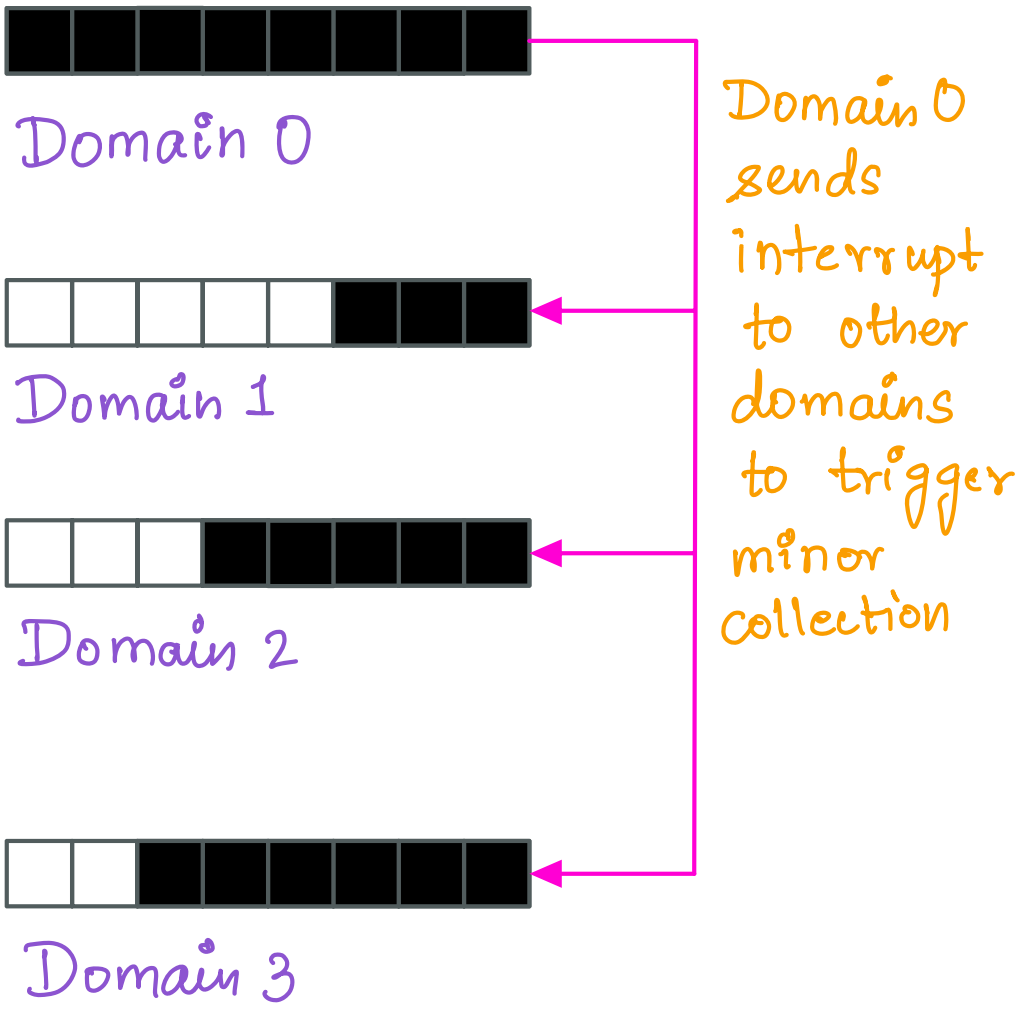
Shared major heap

Domain 0 Domain 1 Domain 2

Domain local minor heaps

- "Domain" is the basic unit of parallelism.
- The major heap is shared between domains.
- Every domain has its own minor heap
- A domain allocates in its own minor heap or the major heap
- Minor collection is stop-the-world and is performed by all the domains parallely.

Stop-the-world Parallel Minor Collection



Parallel promotion:

All domains promote live objects in parallel. Two domains attempting to promote the same object is serialized.

Static work-sharing:

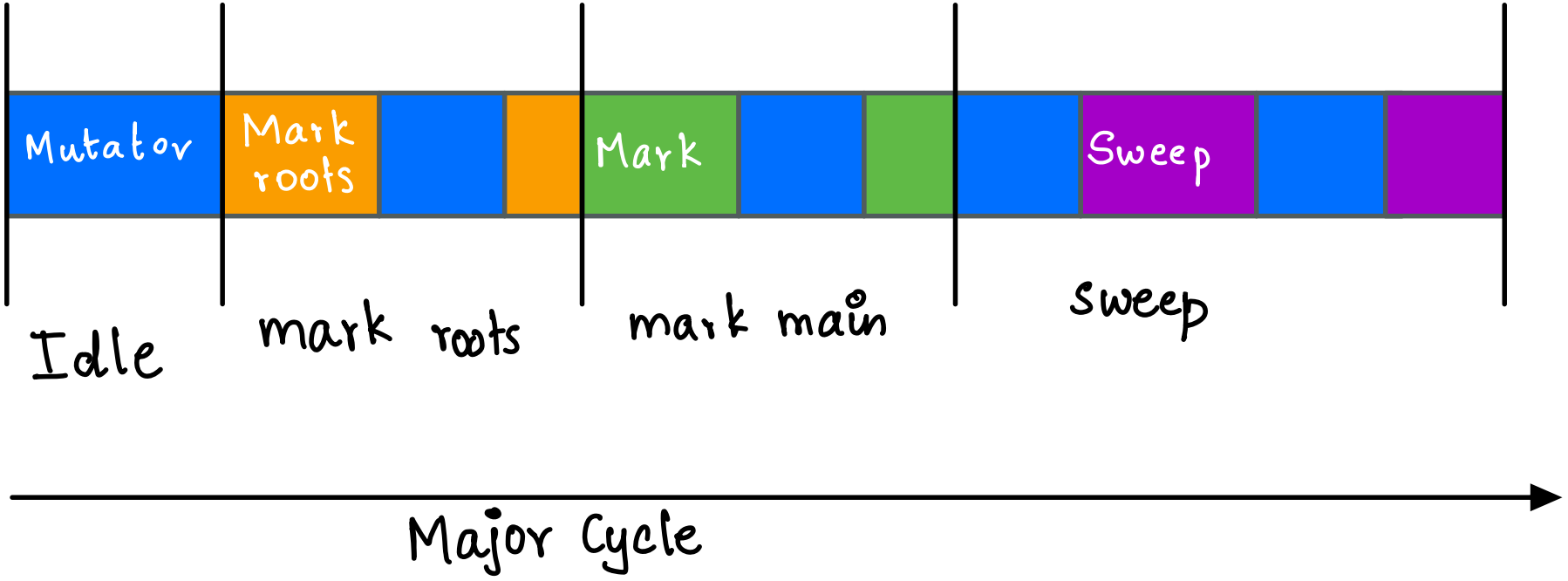
The number of roots to scan is evenly divided amongst the domains. One domain could end up with more work.

MAJOR HEAP

Mark-and-sweep: Marking determines allocations still in use.
Sweeping collects unused allocations.

Non-moving: Objects are in the same memory address.

Incremental: Program stops in small slices for low pausetimes



Marking

Consider a sequential program in OCaml 5

GC Colours:

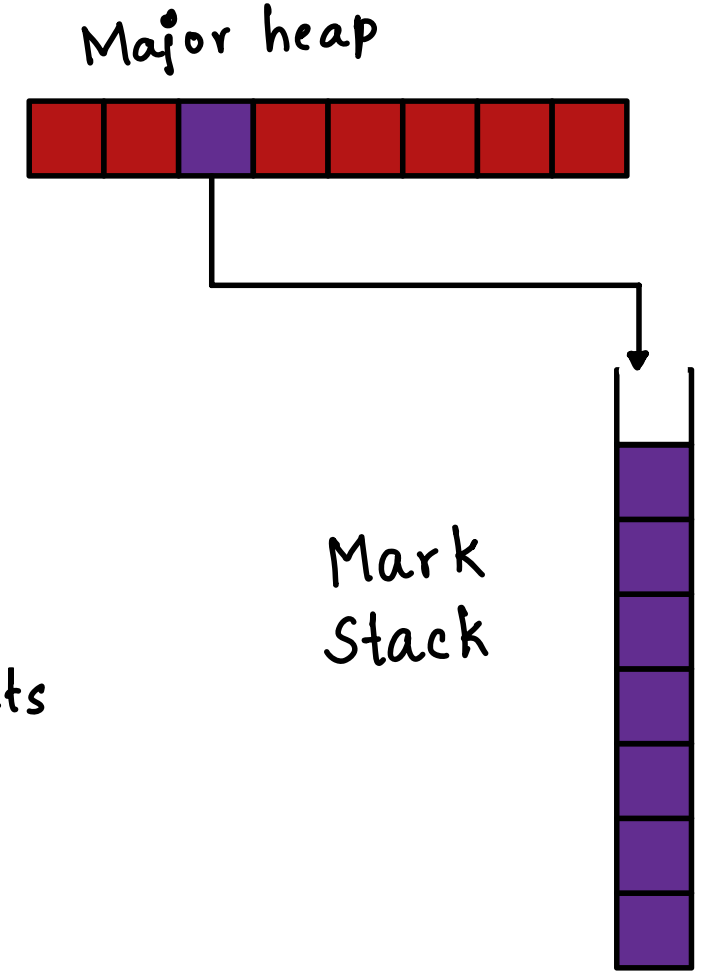
MARKED

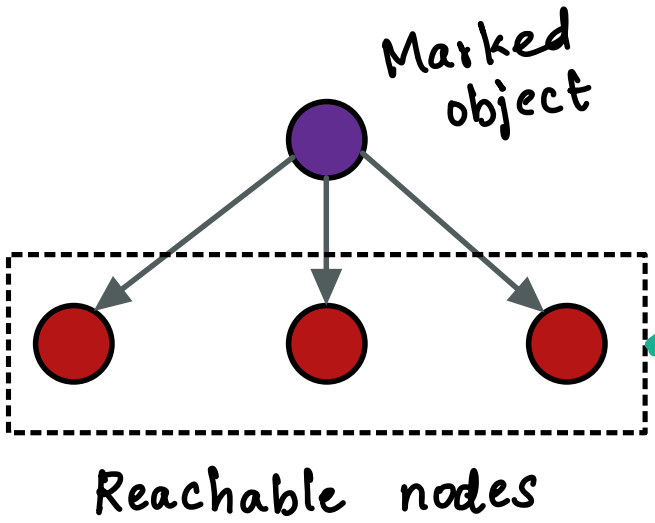
UNMARKED

GARBAGE

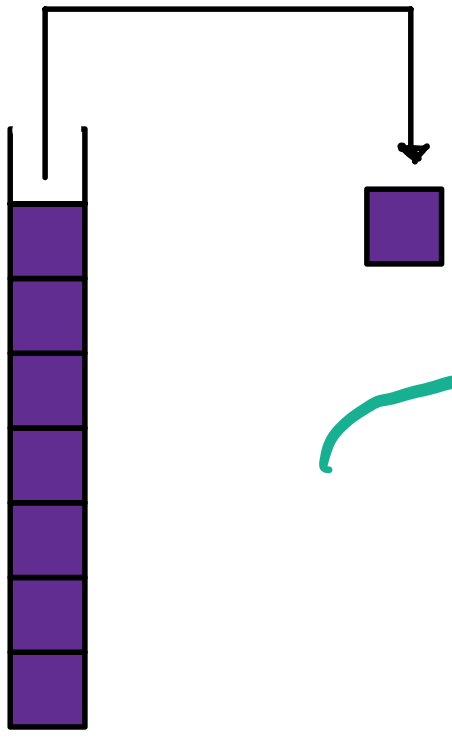
FREE

During the marking phase all live objects are marked and put into a mark stack





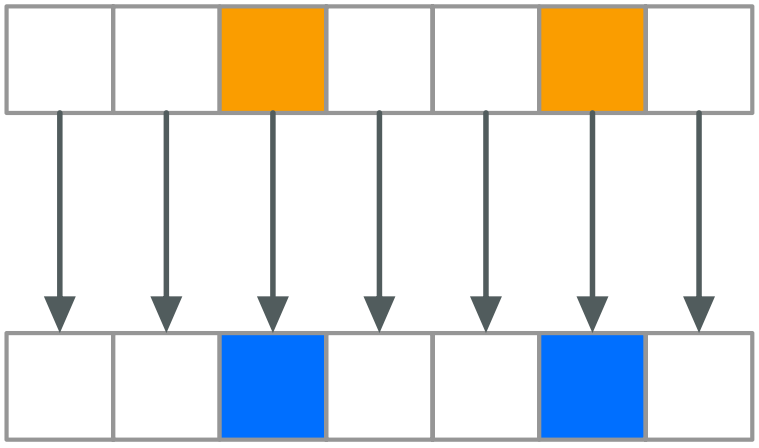
The objects reachable from a live object are also marked and pushed into a mark stack



Items from mark stack are popped and all its children are marked

This process is repeated until the mark stack is empty.

Sweeping



Set of objects affected by marking and sweeping is disjoint. No synchronization is necessary.

Sweeping pass traverses the whole heap incrementally and flips the meaning of the state

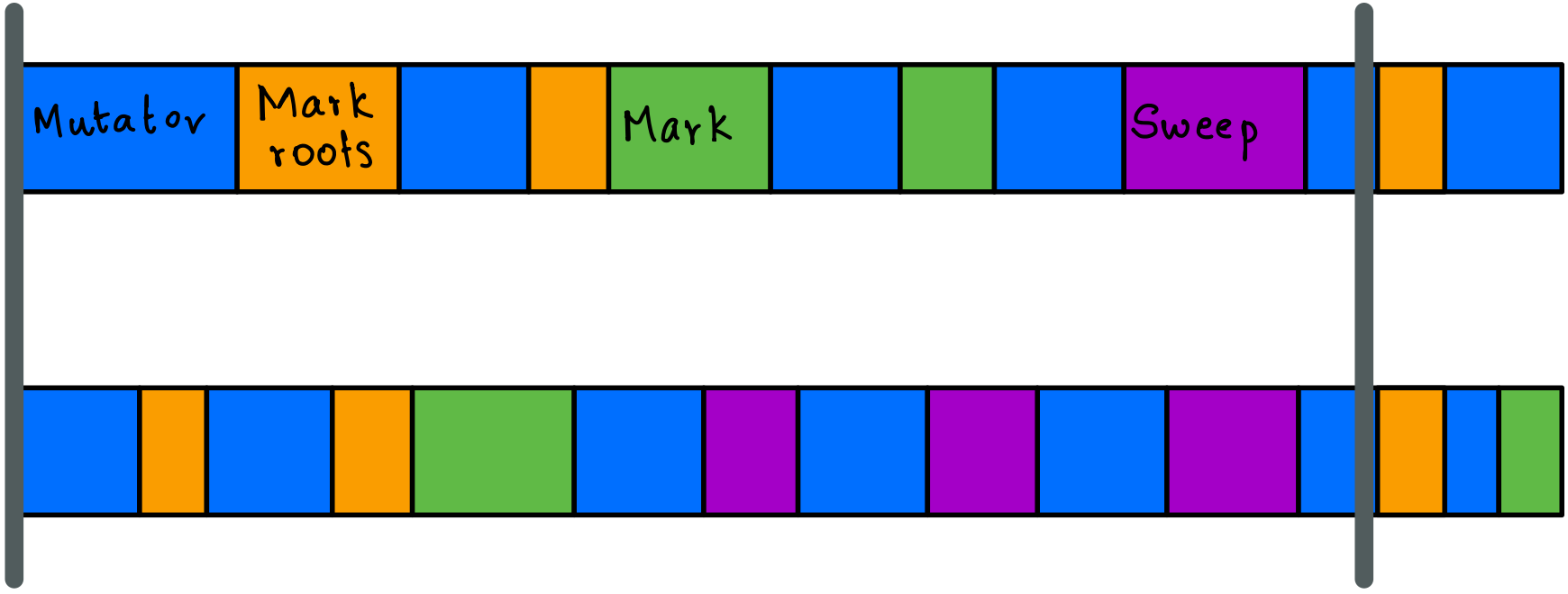
MARKING

Unmarked ⇒ Marked

SWEEPING

Garbage ⇒ Free

Multicore Major Heap

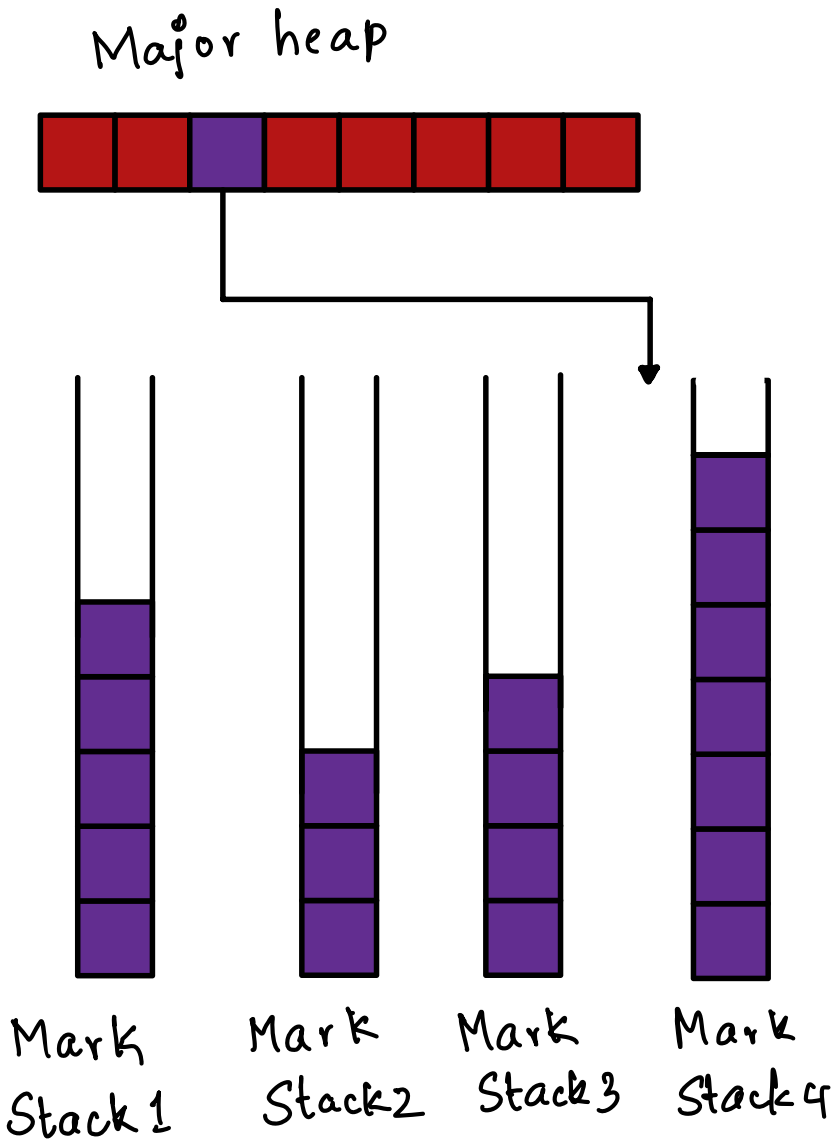


Start of major cycle

End of major cycle

The major collector's design allows overlap of mark & sweep phase, thus removing the need for synchronization.

There's a small STW section at the end of a major cy



- Every domain has its own mark stack
- Marking is idempotent: okay to have the same object on more than one mark stack.
- Sweeping is disjoint: Every domain only sweeps the memory it allocated.