Garbage collection

Strategies for automatic memory management

Memory management

- Explicit memory management:
    - i.e., malloc(..) and free(..) are both explicit.
    - A dangerous source of bugs (dangling pointers, leaks).
        - Performance still varies greatly by implementation.
        - Unsuitable for functional programming.
- Garbage collection: fully automatic memory management.
    - Complex/many strategies; non-deterministic; pauses.

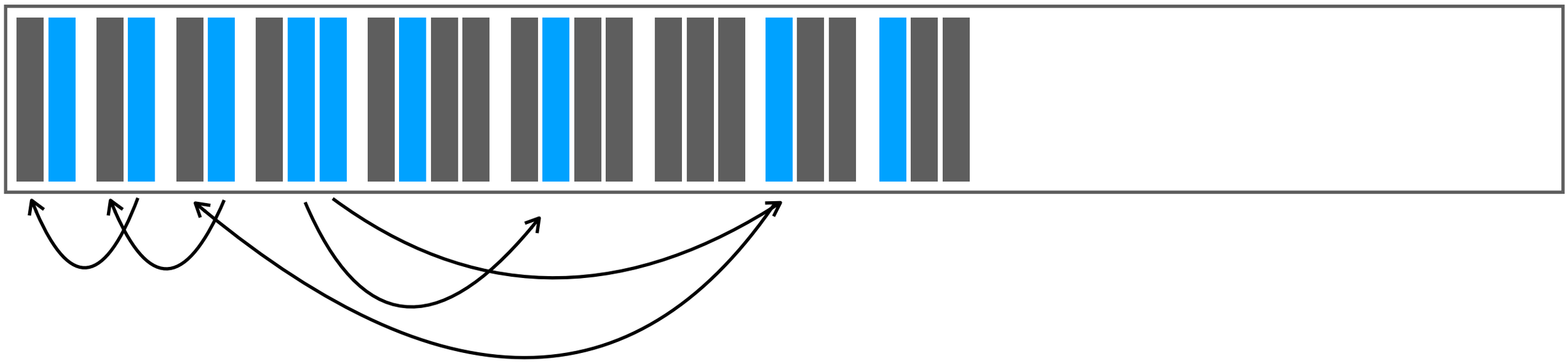
Reference counting

- Add a field `ref-count` to every heap-allocated object.
- When allocated, set `ref-count` to 1.
- Whenever a new (non-transient) reference is established, increment the reference count by 1.
- Whenever a reference disappears (e.g., when a referring object is freed/reclaimed) decrement the count and check if it’s zero; when it’s zero, reclaim the object.
- Problem: reference counting is precise but incomplete.

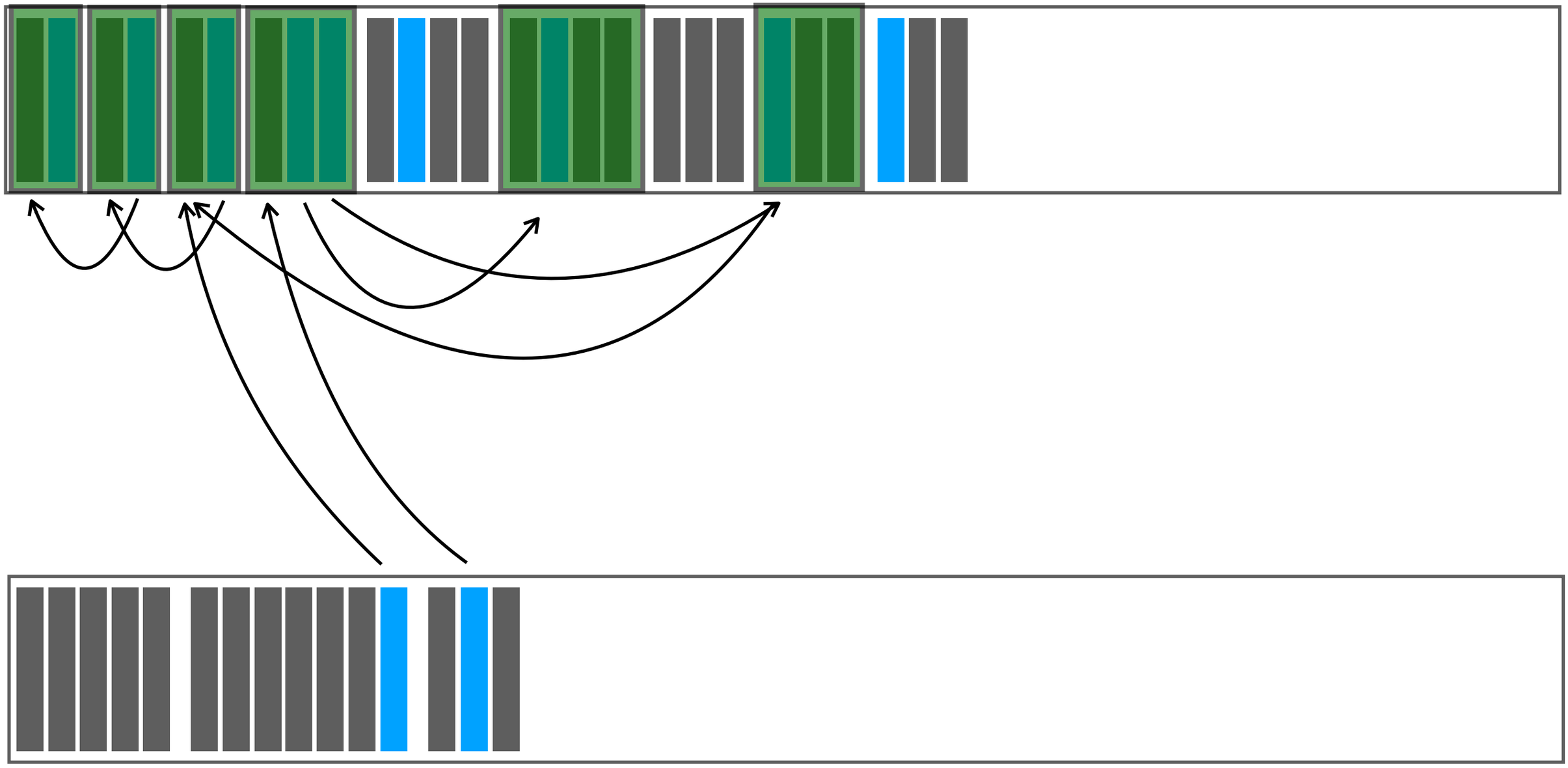
Mark and sweep

- Invented by McCarthy for LISP in 1960 (1 page)
- Also called a *tracing* collector because it follows pointers to *mark* all reachable objects and then *sweeps* all others.
- First identifies a *root set*. Usually all values on the stack.
- Then a *marking* phase traverses the reachable obj graph.
- Finally a *sweeping* phase frees all unmarked objects.

Available heap space

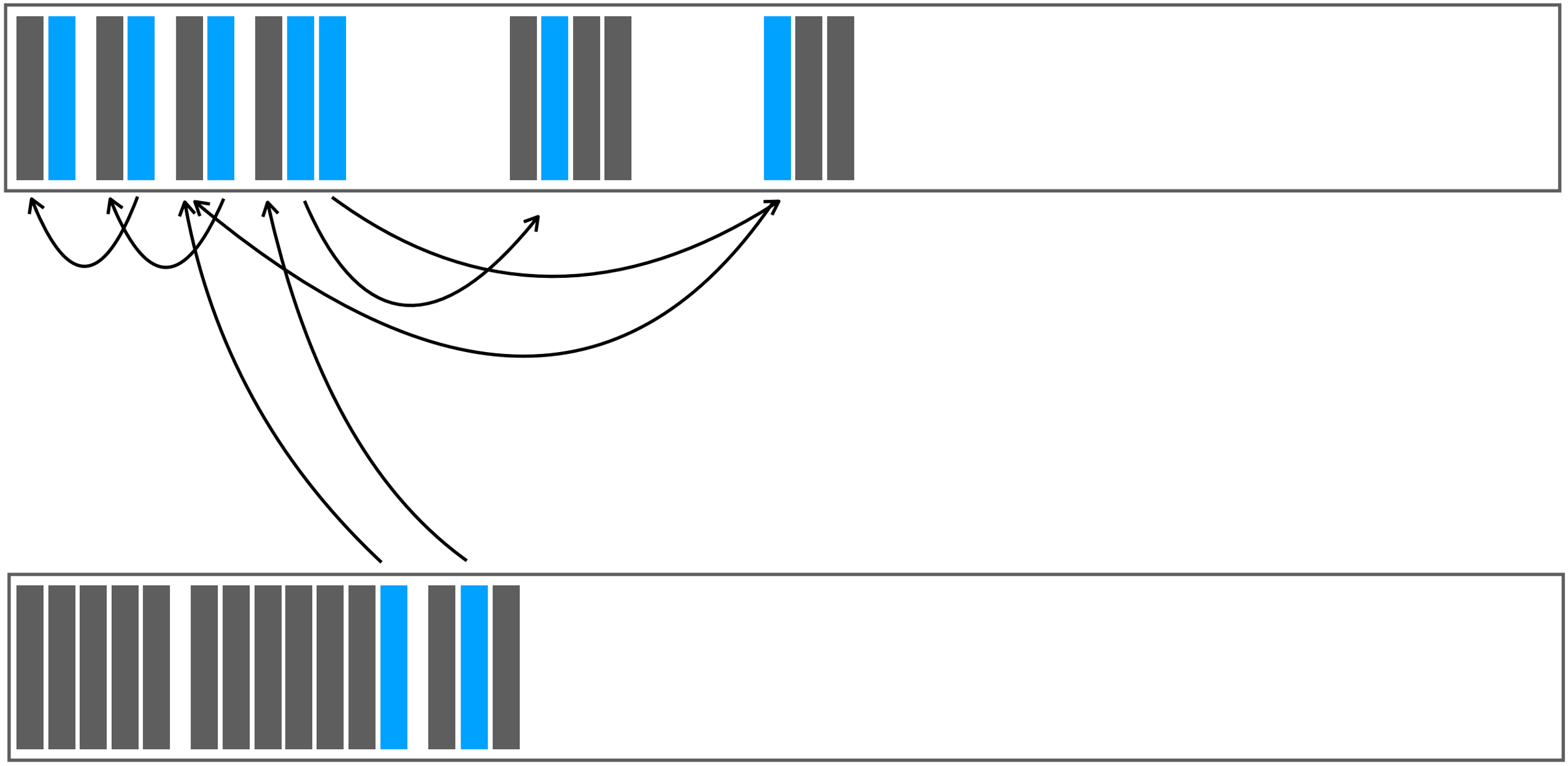


Available heap space



Call stack (root set)

Available heap space



Call stack (root set)

Tagged vs tagless collection

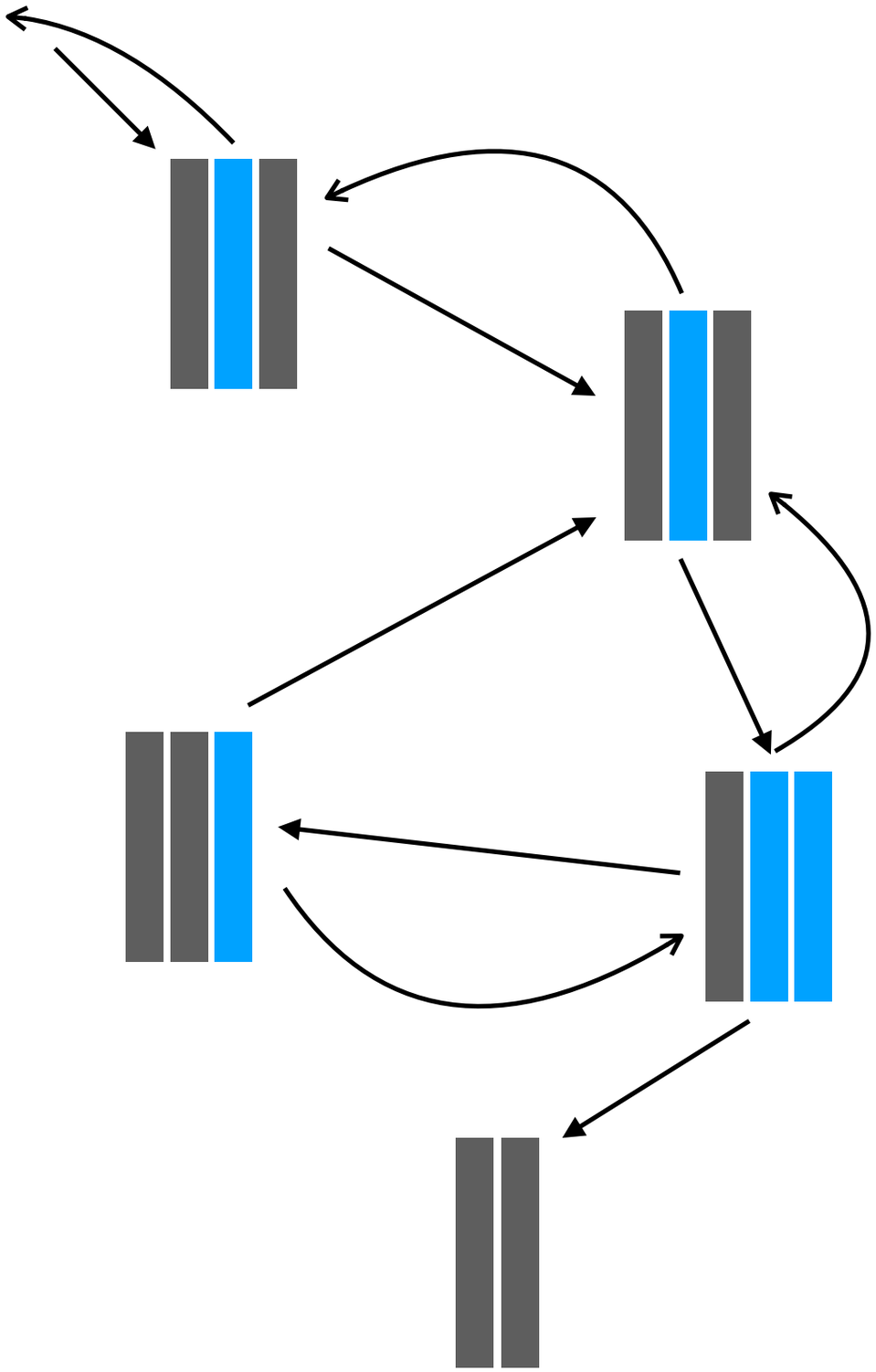
- How do we know which words of memory are pointers?
- Some collectors tag the bottom bit of pointers. (Ocaml)
    - Chez Scheme uses runtime type tagging and segregates these types into their own regions.
- Some collectors exploit static typing to compile linked specialized traversal routines with a function for each type.
- Some compilers produce a traversal per class (e.g. JVM).
- Some collectors are imprecise (but conservative).

Moving and copying

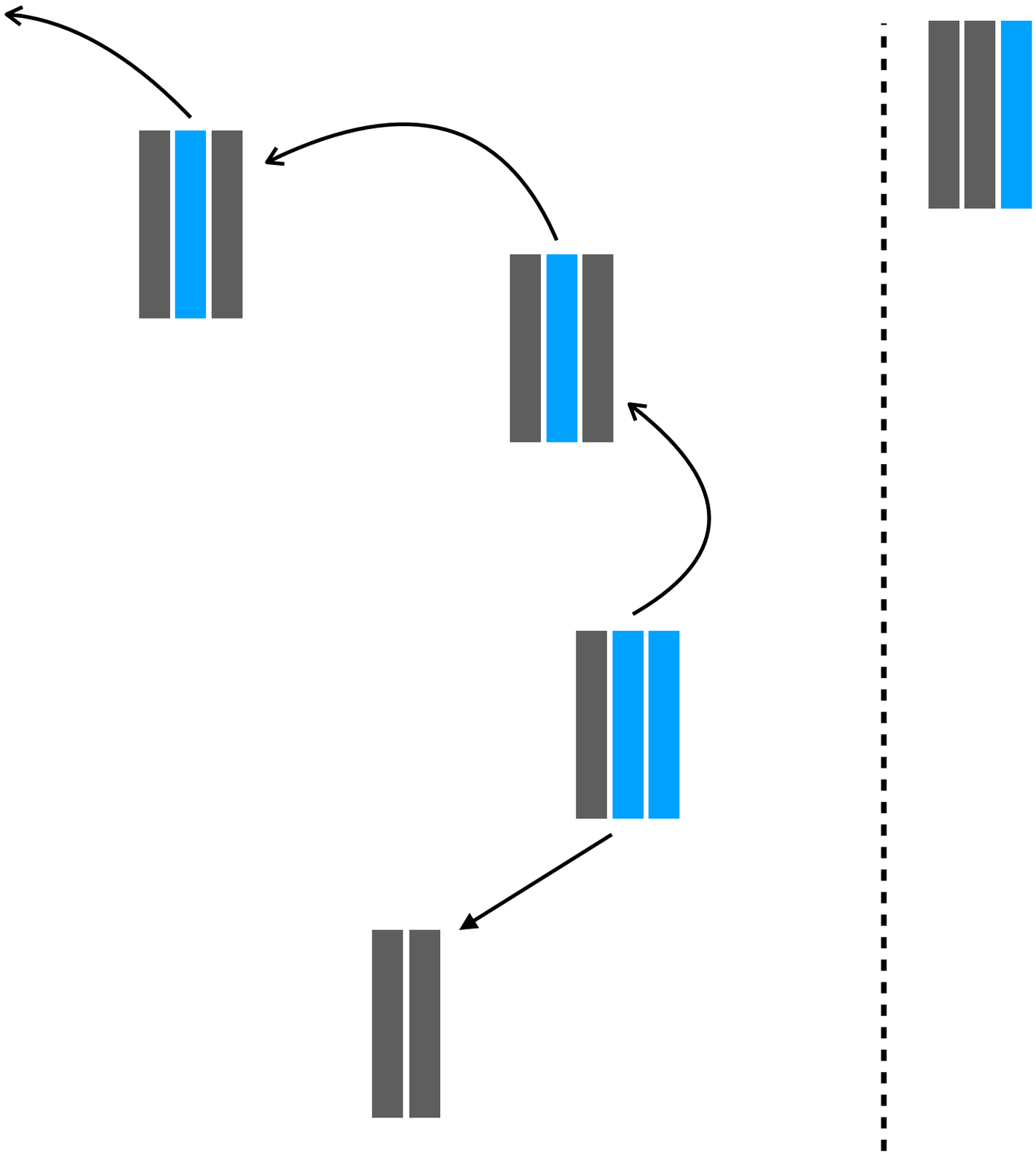
- *Moving/compacting* collectors perform a phase of global defragmentation after each sweep phase.

- *Copying* collection maintains two regions: New and Old.
  - As objects are marked, they are copied into New region.
    - All pointers are forwarded. New is defragmented.
    - No need for a free list; New and Old are swapped.

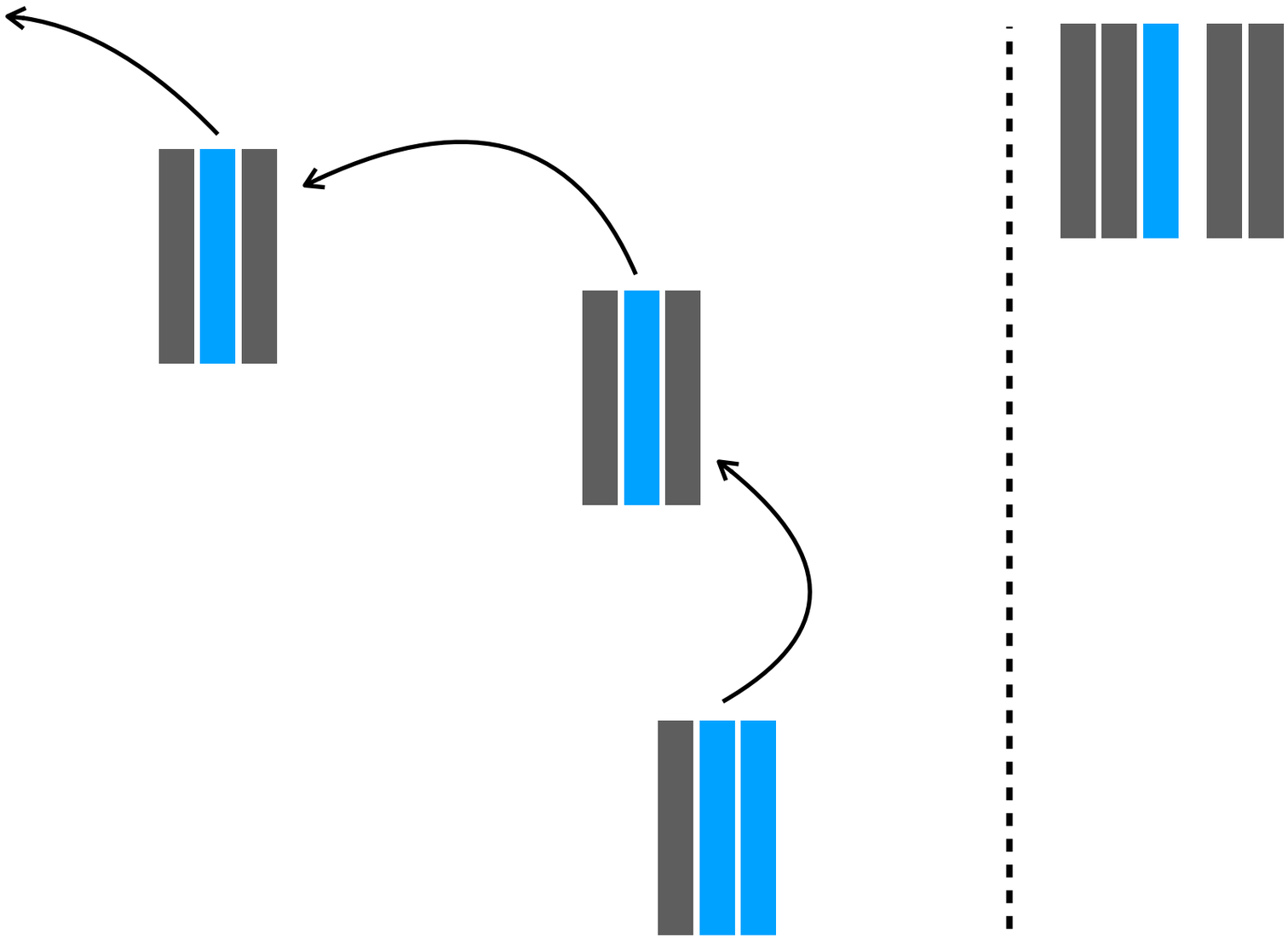
Pointer reversal with copying



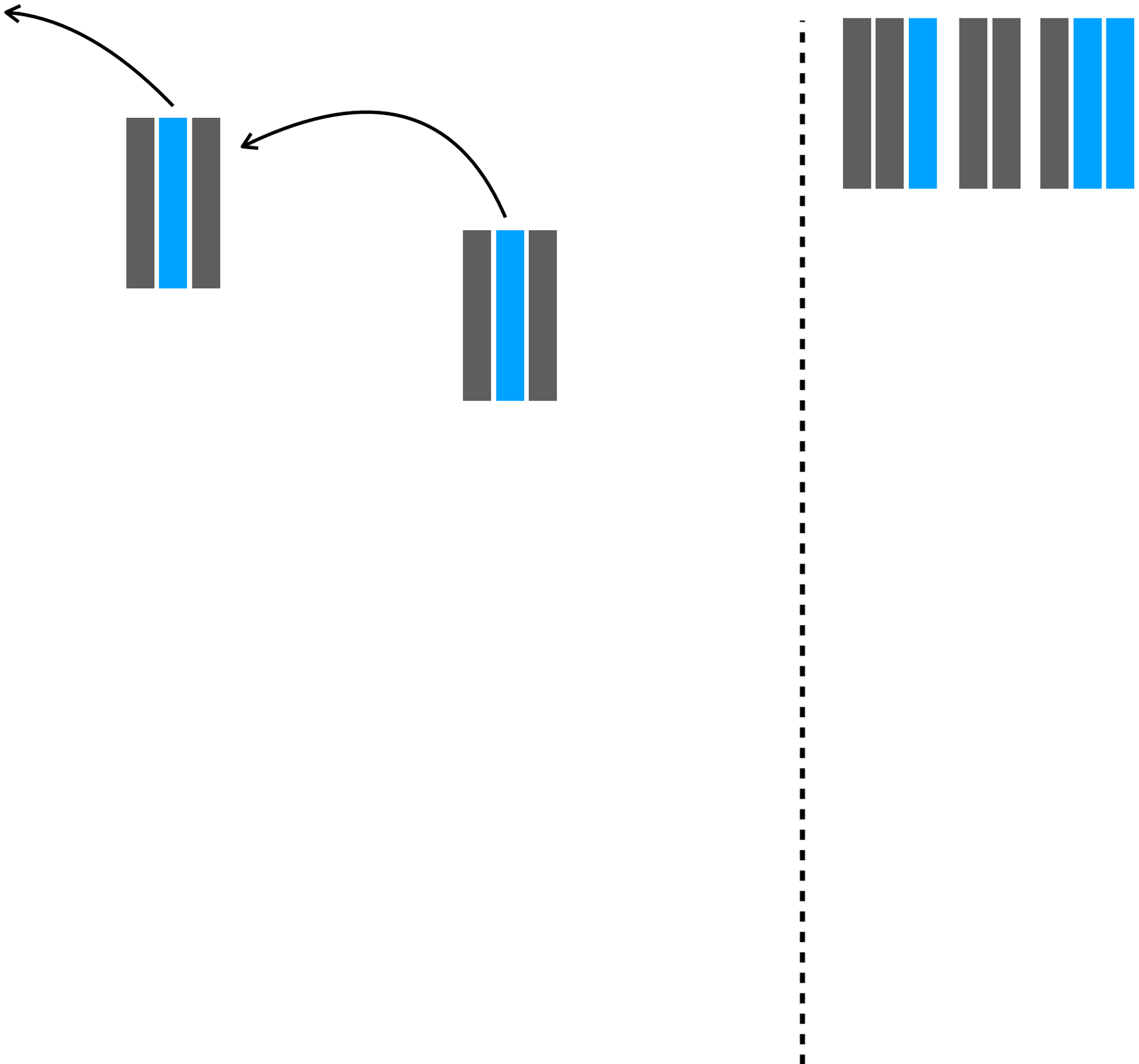
Pointer reversal with copying



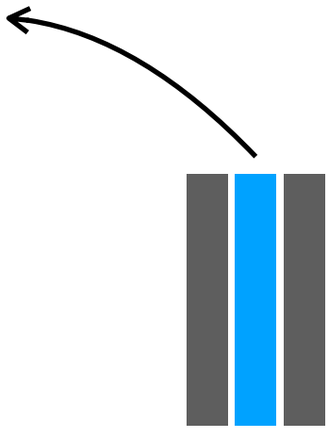
Pointer reversal with copying

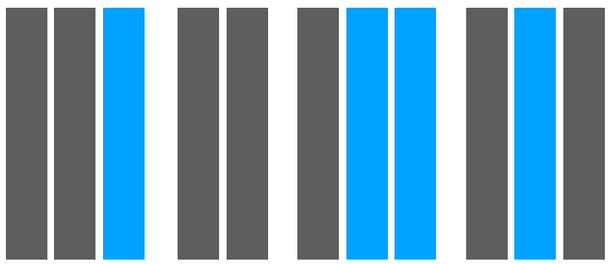


Pointer reversal with copying



Pointer reversal with copying





Pointer reversal with copying

Now this becomes the New space.

…and this becomes the Old space.

Generational Collection

- Most objects are short-lived; previous object lifespan is a good predictor of future object lifespan.

- GC maintains multiple generations; e.g., G0 or nursery space (“eden” in JVM), G1 (“survivor”), G2 (“tenured”), …

- *Minor* collections and rarer *major* collections. For the most part, new objects are only pointed to by other new objects.

    - Where this isn’t true, a store list must be maintained with all mutated references in tenured memory.

Concurrent collection

- Baker (1981) proposed a pause-free O(1) copying collector.
    - At every allocation of N bytes, the algorithm copies O(N) worth of reachable objects to the New space.
        - Adds significant overhead to read instructions.
        - Concurrent&real-time, but with very poor throughput.
- VCGC, very concurrent GC (Huelsbergen et al., 1998):
    - Associates objects with epochs, uses them to pipeline mutation/allocation, marking, sweeping.

Conservative collection

- Boehm–Demers–Weiser GC (or just Boehm GC)
    - “Garbage collection in an uncooperative environment” (Boehm, et al., 1988)
- Avoids allocating the lowest part of virtual address space.
- Doesn’t require tagging; aligned values are all pointers.
- Requires a free list, no moving/copying
- Go try it out! https://github.com/ivmai/bdwgc