Object Lifetimes

- Most objects live for a very short time, while a small fraction of them live much longer.
  - This has been empirically measured in a variety of programs in a variety of programming languages.
  - 80-98% of all newly-allocated objects die before another megabyte has been allocated.
  - The majority of die even more quickly – within tens of kilobytes of allocation.

Object Lifetimes

- Most objects live for a very short time, while a small fraction of them live much longer.
  - This has been empirically measured in a variety of programs in a variety of programming languages.
  - 80-98% of all newly-allocated objects die before another megabyte has been allocated.
  - The majority of die even more quickly – within tens of kilobytes of allocation.
- This leads to the major source of inefficiency in simple garbage collectors.
  - Even if garbage collections are fairly close, separated by a few kilobytes of allocation, most objects die before a collection and never need to be copied.
  - However, a large fraction of objects that survive a single collection live through multiple collections. These objects are repeatedly copied at multiple scavenge points.

Generational Collection

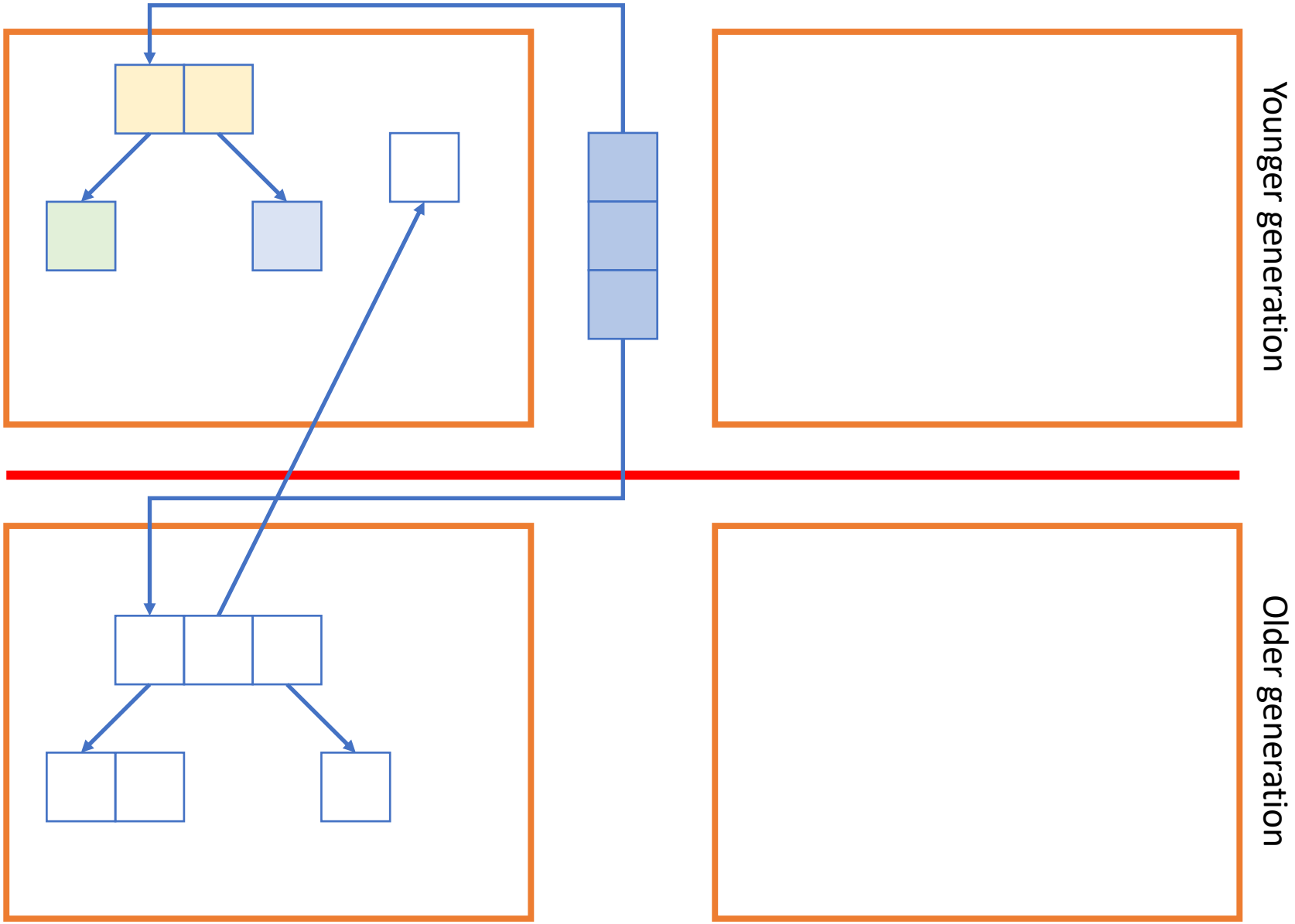
- Key idea
  - Partition heap into multiple bins (“generations”) for objects of well-separated age ranges.
  - Scavenge older generations less frequently than younger ones.
  - Advance objects to old generation space as they age (survive several scavenges of their current generation).

Generational Collection

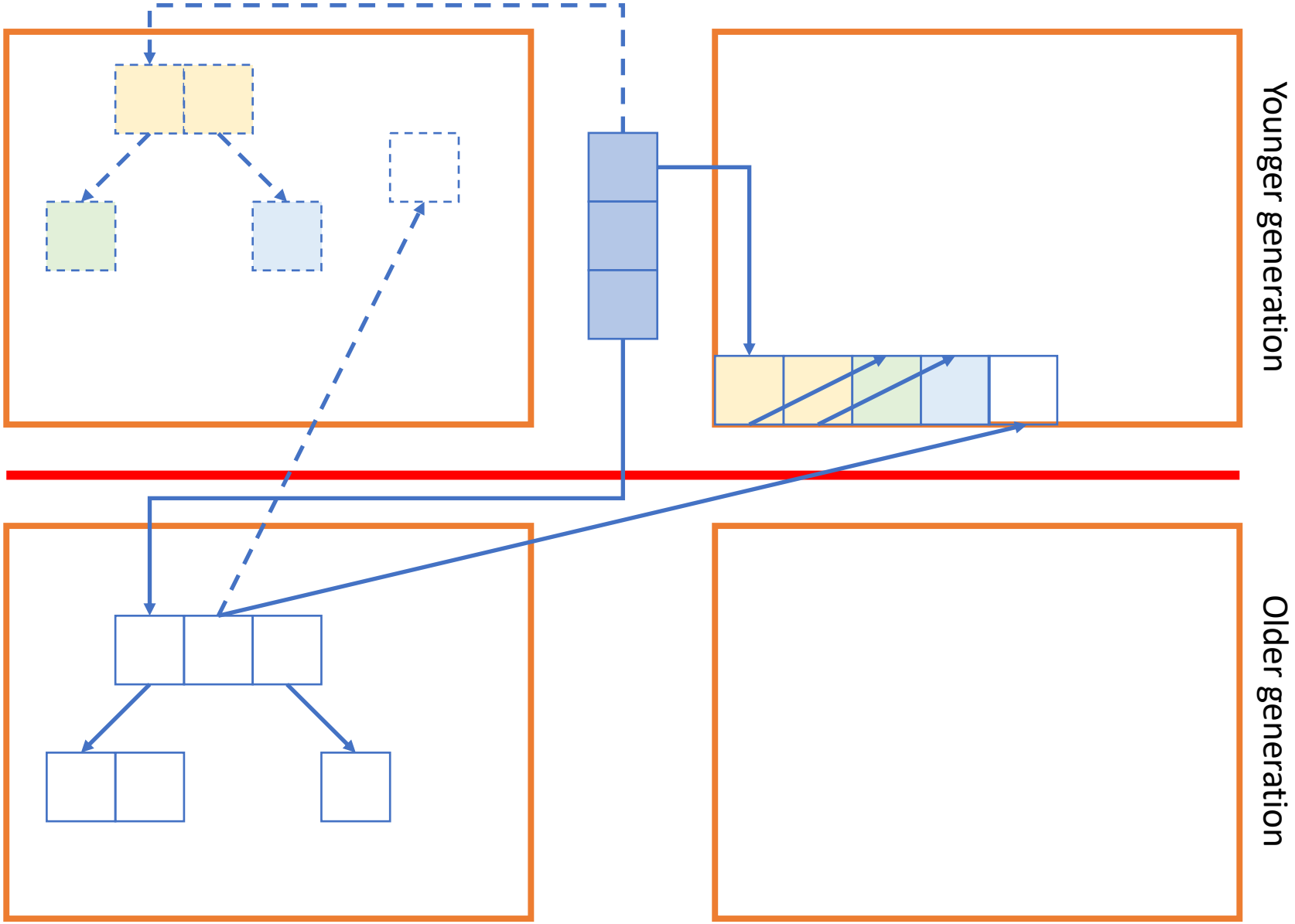
- Key idea
  - Partition heap into multiple bins (“generations”) for objects of well-separated age ranges.
  - Scavenge older generations less frequently than younger ones.
  - Advance objects to old generation space as they age (survive several scavenges of their current generation).

- The generational idea pertains to the tracing question.
  - Whether reclamation happens by copying or marking is orthogonal.
  - For concreteness and simplicity, we will assume a “stop-and-copy” collector using semispaces and with two generations.

Initial State



State After Scavenging One Generation



Detecting Intergenerational References

- It must be possible to scavenge the younger generation(s) without scavenging older ones.
  - But object liveness is a global property.
  - Any pointers from an older generation to a newer one must be found at scavenge time and used as a root of the traversal.

Detecting Intergenerational References

- It must be possible to scavenge the younger generation(s) without scavenging older ones.
  - But object liveness is a global property.
  - Any pointers from an older generation to a newer one must be found at scavenge time and used as a root of the traversal.
- Possible solutions.
  - No pointer can point directly from an older generation to a newer generation, but must point to a cell in an *indirection table*. This table is used as part of the root set.
    - Needs specialized hardware support.

Detecting Intergenerational References

- It must be possible to scavenge the younger generation(s) without scavenging older ones.
  - But object liveness is a global property.
  - Any pointers from an older generation to a newer one must be found at scavenge time and used as a root of the traversal.
- Possible solutions.
  - No pointer can point directly from an older generation to a newer generation, but must point to a cell in an *indirection table*. This table is used as part of the root set.
    - Needs specialized hardware support.
  - Use a *pointer recording* technique, i.e., keep track of such pointers so that they can be found at scavenge time.
    - This requires something like a write barrier.
    - Results in a conservative approximation of true liveness.

Detecting Intergenerational References

- It must be possible to scavenge the younger generation(s) without scavenging older ones.
  - But object liveness is a global property.
  - Any pointers from an older generation to a newer one must be found at scavenge time and used as a root of the traversal.
- Possible solutions.
  - No pointer can point directly from an older generation to a newer generation, but must point to a cell in an *indirection table*. This table is used as part of the root set.
    - Needs specialized hardware support.
  - Use a *pointer recording* technique, i.e., keep track of such pointers so that they can be found at scavenge time.
    - This requires something like a write barrier.
    - Results in a conservative approximation of true liveness.
- Also possible to track pointers the other way, allowing independent scavenging of an older generation.
  - Generally not cost-effective. Easier to just consider all data in the newer generation as possible roots.

Design Decisions

- Advancement policy
  - How long must an object survive in one generation before it is advanced to the next?
- Heap organization
  - How should storage space be divided and used between generations, and within a generation?
  - How does this affect locality in virtual memory and cache?
- Traversal algorithms
  - What effect does the traversal algorithm have on locality, and what traversal yields the best result?
- Collection scheduling
  - For a non-incremental collector, how might we avoid or mitigate the effect of disruptive pauses, especially in interactive applications?
  - Can we improve efficiency by careful “opportunistic” scheduling?
  - Can this be adapted to incremental schemes to reduce floating garbage?
- Intergenerational references
  - What is the best way to find the live pointers from older generations into the one(s) we are scavenging?