Generational and Incremental Garbage Collection

- Object Lifetimes
- Copying Generational Collectors
- Incremental Garbage Collection

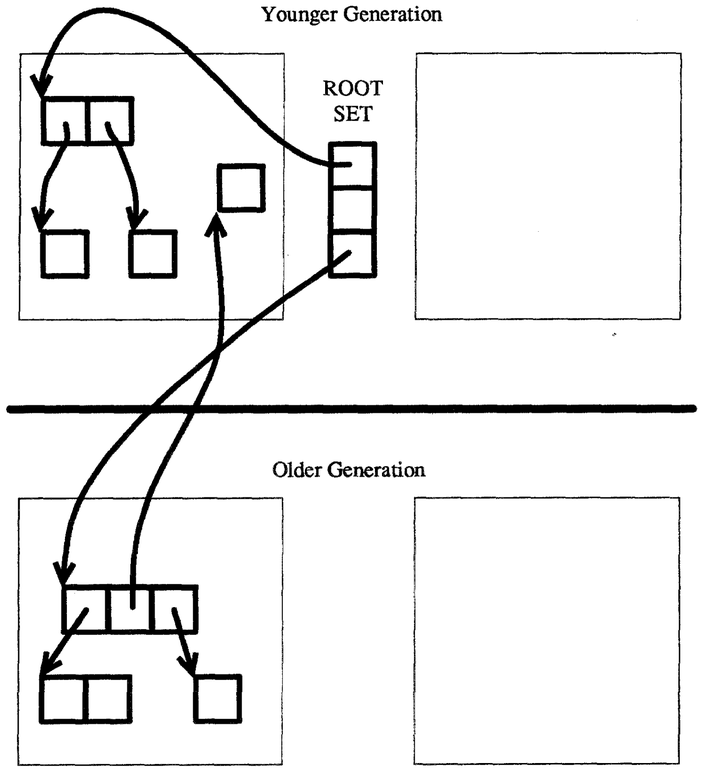
Object Lifetimes

- Most objects have short time; a small percentage live much longer
  - This seems to be generally true, no matter what programming language is considered, across numerous studies
  - Although, obviously, different programs and different languages produce varying amount of garbage

- Implications:
  - When the garbage collector runs, live objects will be in a minority
  - Statistically, the longer an object has lived, the longer it is likely to live
  - Can we design a garbage collector to take advantage?

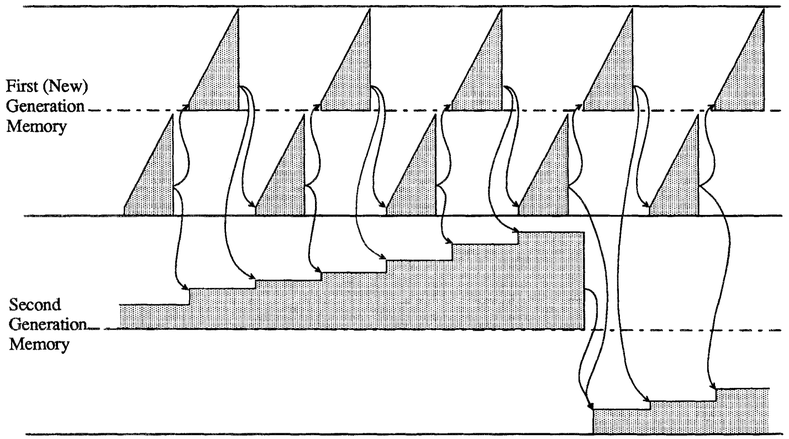
Copying Generational Collectors (1/4)

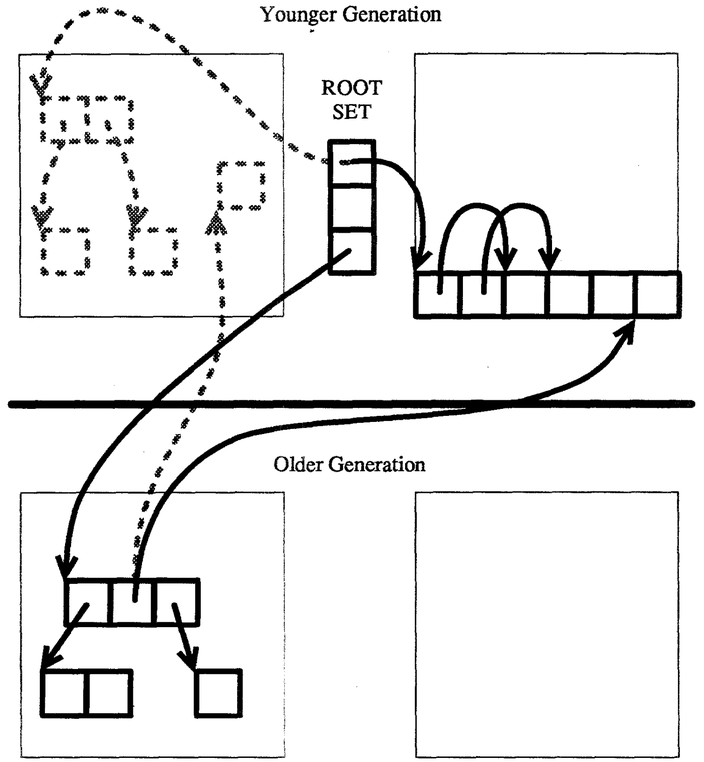
- In a generational garbage collector, the heap is split into regions for long-lived and young objects
  - Regions holding young objects are garbage collected more frequently
  - Objects are moved to the region for long-lived objects if they're still alive after several collections
  - More sophisticated approaches may have multiple generations, although the gains diminish rapidly with increasing numbers of generations
- Example: stop-and-copy using semispaces with two generations
  - All allocations occurs in the younger generation's region of the heap
  - When that region is full, collection occurs as normal
  - …



Copying Generational Collectors (2/4)

- …
- Objects are tagged with the number of collections of the younger generation they have survived; if they're alive after some threshold, they're copied to the older generation's space during collection
- Eventually, the older generation's space is full, and is collected as normal





- Note: not to scale: older generations are generally much larger than the younger, as they're collected much less often

Copying Generational Collectors (3/4)

- Young generation must collected independent of long-lived generation
- But – there may be references between generations
  - References from young objects to long-lived objects
    - Straight-forward – most young objects die before the long-lived objects are collected
    - Treat the younger generation objects as part of the root set for the long-lived generation, when collection of the long-lived generation is needed
  - References from long-lived objects to young objects:
    - Problematic, since requires scan of long-lived generation to detect
    - Maybe use indirection table (“pointers-to-pointers”) for references from long-lived generation to young generation
      - The indirection table forms part of the root set of the younger generation
      - Moving objects in younger generation requires updating indirection table, but not long-lived objects
      - Long-lived objects are collected infrequently, and may keep younger objects alive longer than expected

Copying Generational Collectors (4/4)

- Variations on copying generational collectors are widely used
  - E.g., the HotSpot JVM uses a generational garbage collector

- Copying generational collectors are efficient:
  - Cost of collection is generally proportional to number of live objects
  - Most objects don't live long enough to be collected; those that do are moved to a more rarely collected generation
  - Longer-lived generation must eventually be collected; this can be very slow

Incremental Garbage Collection (1/5)

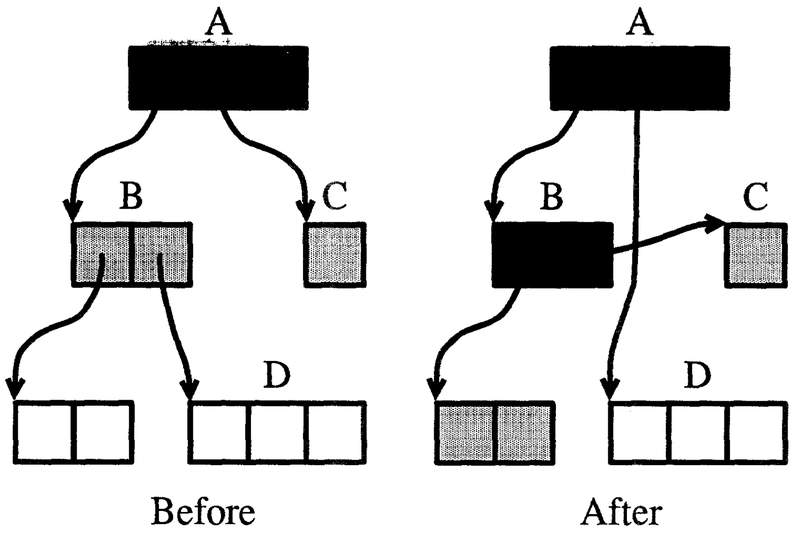
- Preceding discussion has assumed the collector “stops-the-world” when it runs
  - Problematic for interactive or real-time applications
- Desire a collector that can operate incrementally
  - Interleave small amounts of garbage collection with small runs of program execution
  - Implication: the garbage collector can’t scan the entire heap when it runs; must scan a fragment of the heap each time
  - Problem: the program (the “mutator”) can change the heap between runs of the garbage collector
  - Need to track changes made to the heap between garbage collector runs; be conservative and don’t collect objects that might be referenced – can always collect on the next complete scan

Incremental Garbage Collection (2/5)

- Tricolour marking: each object is labelled with a colour:
  - White – not yet checked
  - Grey – live, but some direct children not yet checked
  - Black – live
- Basic incremental collector operation:
  - Garbage collection proceeds with a wavefront of grey objects, where the collector is checking them, or objects they reference, for liveness
  - Black objects behind are behind the wavefront, and are known to be live
  - Objects ahead of the wavefront, not yet reached by the collection, are white; anything still white once all objects have been traced is garbage
  - No direct pointers from black objects to white – any program operation that will create such a pointer requires coordination with the collector

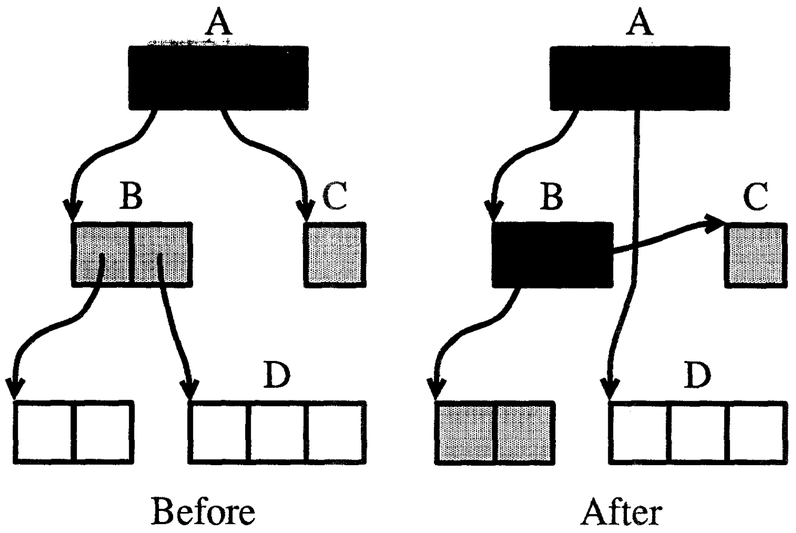
Incremental Garbage Collection (3/5)

- Program and collector must coordinate
  - Garbage collector runs
    - Object A scanned, known to be live → black
    - Objects B and C are reachable via A, and are live, but some of their children have not been scanned → grey
    - Object D not checked → white
  - Program runs, and swaps the pointers from A→C and B→D such that A→D and B→C
  - This creates a pointer from black to white
  - Program must now coordinate with the collector, else collection will continue, marking object B black and its children grey, but D will not be reached since children of A have already been scanned



Incremental Garbage Collection (4/5)

- Coordination strategies:
  - Read barrier: trap attempts by the program to read pointers to white objects, colour those objects grey, and then let program continue
    - Makes it impossible for the program to get a pointer to a white object, so it cannot make a black object point to a white
  - Write barrier: trap attempts to change pointers from black objects to point to white objects
    - Either then re-colour the black object as grey, or re-colour the white object being referenced as grey
    - The object coloured grey is moved onto the list of objects whose children must be checked



Incremental Garbage Collection (5/5)

- Many variants on read- and write-barrier tricolour algorithms
  - Performance trade-off differs depending on hardware characteristics, and on the way pointers are represented
  - Write barrier generally cheaper to implement than read barrier, as writes are less common in most code
- There is a balance between collector operation and program operation
  - If the program tries to create too many new references from black to white objects, requiring coordination with the collector, the collection may never complete
  - Resolve by forcing a complete stop-the-world collection if free memory is exhausted, or after a certain amount of time

Generational and Incremental Garbage Collection

- Object Lifetimes
- Copying Generational Collectors
- Incremental Garbage Collection