Generation Org and Age, inter-generational pointers

How do we organize generations or record age?

Generation organization

- One [semi-]space per generation
  - Simplest promotion policy:
    - Advance all live objects at once
    - No need to record object ages
    - Use older generation as to_space OR recycle youngest gen.
    - Requires multiple generations to filter tenured garbage
    - Promotion rate is high

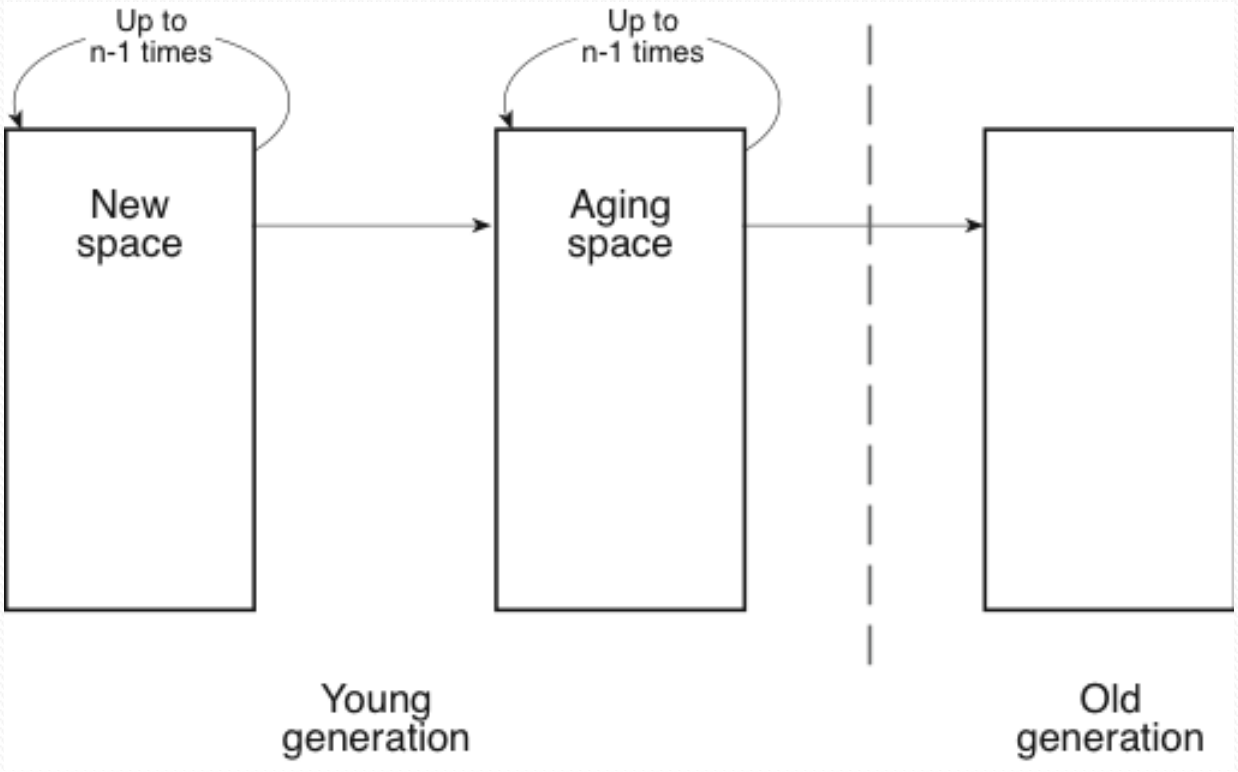
Generation organization

- Creation space
  - Divide generation into **creation space** and **aging space**
  - Allocate objects in creation space
  - Aging space stores survivors from creation space
  - # of survivors of each scavenge expected to be low, both spaces can be small
  - For good performance
    - Hold creation space in memory
    - Do not swap it out

Age recording

- Not necessary for *en masse* promotion schemes
  - All survivors are promoted
- Methods requiring object's age could
  - Record object's age in its header
    - Cost? Manipulated? Copied?
  - Segregate objects of different ages within a generation
    - Shaw uses buckets
    - New bucket → aging bucket → next generation
    - Buckets != generation

Buckets vs Generations



What about large objects?

- Use large object space
  - Use as in copy collector
  - Save pause time
- How large is large?
  - Absolute measure: e.g., > 1024 bytes
  - Relative measure: e.g., occupy > 10% of to-space
- Should headers be promoted?
  - Some algorithms do
  - Some algorithms do not want to risk letting headers become tenured garbage

Inter-generational ptrs

- Generational collectors reduce pause time:
  - They collecting only a region of the heap
- The only reference to an object in this region may be
  - A ptr from outside the region
  - An inter-generational ptr.
    - Must be identified and treated as part of root set
    - Whose responsibility?

Identifying inter-generational ptrs

- Simplest approach:
  - Scan older-generation at collection time
    - Introduces no cost to mutator, but
    - Requires more scanning
    - Has terrible locality of reference
    - Start at roots and follow their objects for inter-generational ptrs
- More precise methods for recording them?

Write-barrier usage

- Recall
  - Inter-generational ptrs arise in two ways
    - Promotion of objects to old generation - detected by collector
    - Pointer stores – How do we detect and record?
      - Scan older generations?
      - Use a write-barrier
- Write-barrier:
  - Interception of writes to certain memory locations by mutator
  - Implemented in
    - Hardware; Software; OS support

Hardware write-barrier

- Requires no additional instructions
- Advantageous in presence of uncooperative compilers
- Requires special-purpose hardware
- Modifications to virtual-memory systems not always readily available

Software write-barrier

- Have compiler emit a few instructions before each ptr read or write
- Implementers must consider these factors:
  - Minimization of cost to mutator
  - Space overhead of recording ptr writes
  - Efficiency of identifying old-young ptrs at collection time
  - Do we inline the write-barrier code?

OS supported write-barrier

- Uses virtual memory protection mechanism
  - To trap access to protected pages
- Use virtual memory page modification dirty bits
  - As map of locations of objects with updated ptr fields
- Advantage of using virtual memory
  - It is portable
  - Requires no changes to compiler

Which pointer writes to trap

- Writes that may not need to be trapped
  - Stores to registers
  - Stores to stack
  - Initializing stores
    - Not easy to detect in many languages
  - These stores form the majority of pointer stores
- Stores that should be trapped
  - Non-initializing stores
    - ~ 1 % of instructions generated by Lisp or ML compilers

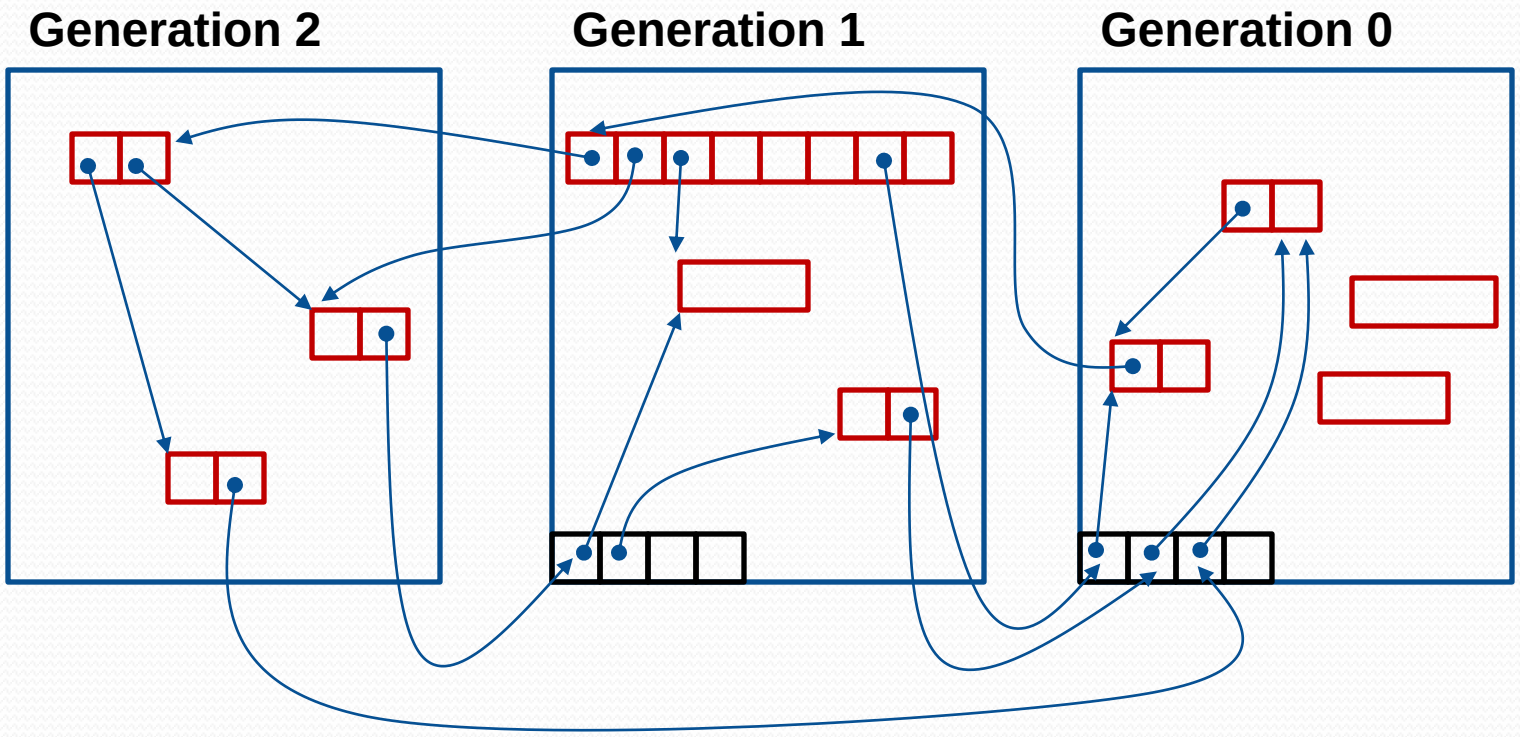
Trapping & recording inter-gen ptrs

- Entry tables
- Remembered sets
- Sequential store buffer
- Page marking with hardware support
- Page marking with virtual memory support
- Card marking

Entry tables

- First generational collector
  - [implemented by Liberman & Hewitt , 1983]
  - Made objects from old gen to point *indirectly* to objects in young gen
  - Each gen had an *entry table of refs* from older gen to younger gen
    - Storing pointer to young gen object from old gen object resulted in adding new entry to young gen's entry table
    - If old object already contains ref to item in entry table, that entry was removed

Entry table example



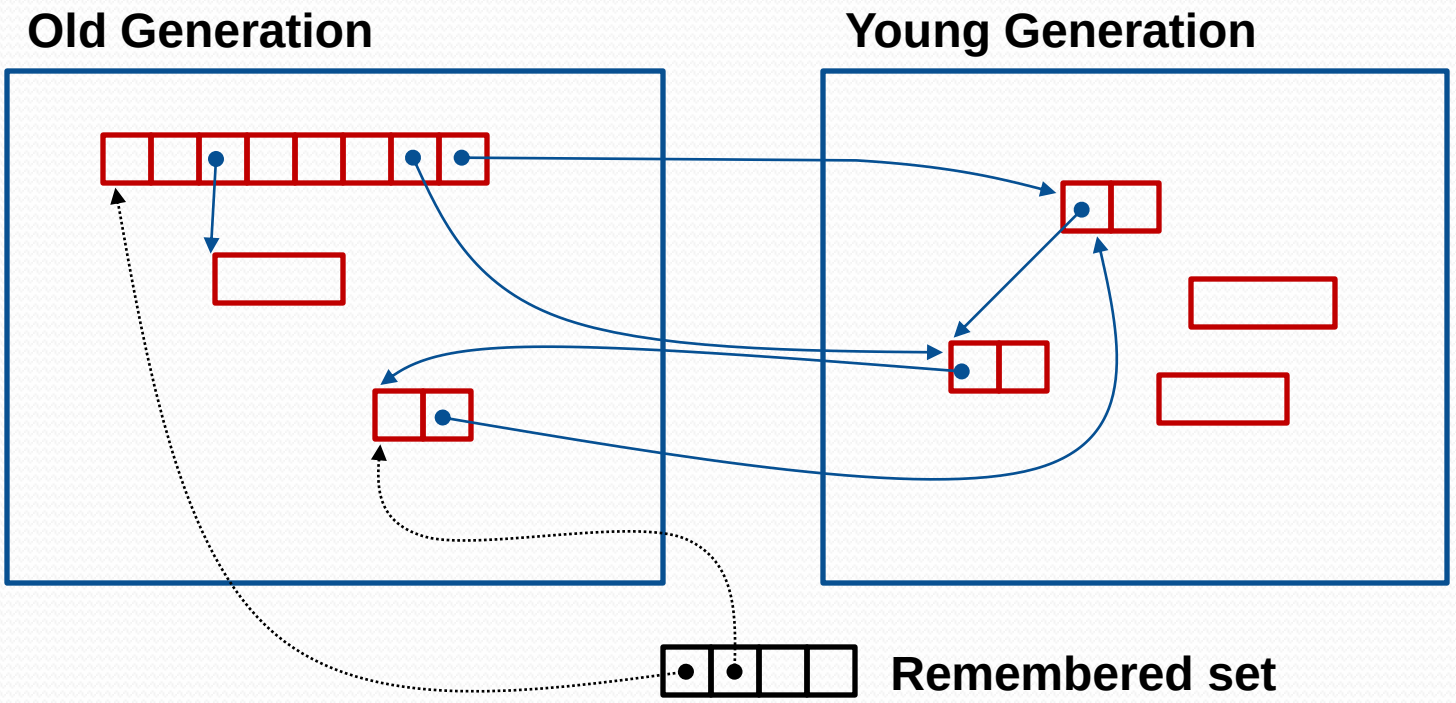
Effects of using entry table

- Advantages:
  - When collecting young generation,
    - No need to search every older generation
    - Only scan its entry table
- Disadvantages:
  - Indirection scheme
  - Table may contain duplicate references to a single object
    - Cost α # store operations vs # inter-gen ptrs
  - Most modern gen collectors avoid indirection
    - They record location of ptrs instead

Remembered sets

- Ungar's Generational Scavenging Collector
  - Records objects that point to younger generations
  - Write-barrier implemented in software
  - It intercepts stores to check
    - Whether pointer was being stored
    - Whether ref to young object stored in old object
      - Address of such object added to remembered set
      - Object has bit in header to indicate if already in remembered set

Remembered set example



Effects of using remembered set

- Advantages:
  - Scanning costs at collection time dependent on # of remembered set objects, not number of pointers
- Disadvantages:
  - Cost of store checking is high
    - If multiple ptrs written into old object between collections , checks would be repeated
  - Object would be scanned in its entirety at collection time
    - Why?

Reducing scanning cost

- Remember location of ptr in object in remembered set
- Problems:
  - Increase size of remembered set
  - Multiple entries for large objects if large objects have multiple ptrs modified
- Approaches to reduce write-barrier costs while limiting space & time overhead
  - Use vector of address for stored-into-object
    - If overflow add more space or run GC
    - GC filters the list of objects that meet certain requirements