CSEP 590 – Programming Systems
University of Washington

Lecture 5: Garbage Collection

Michael Ringenburg
Spring 2017

Course News

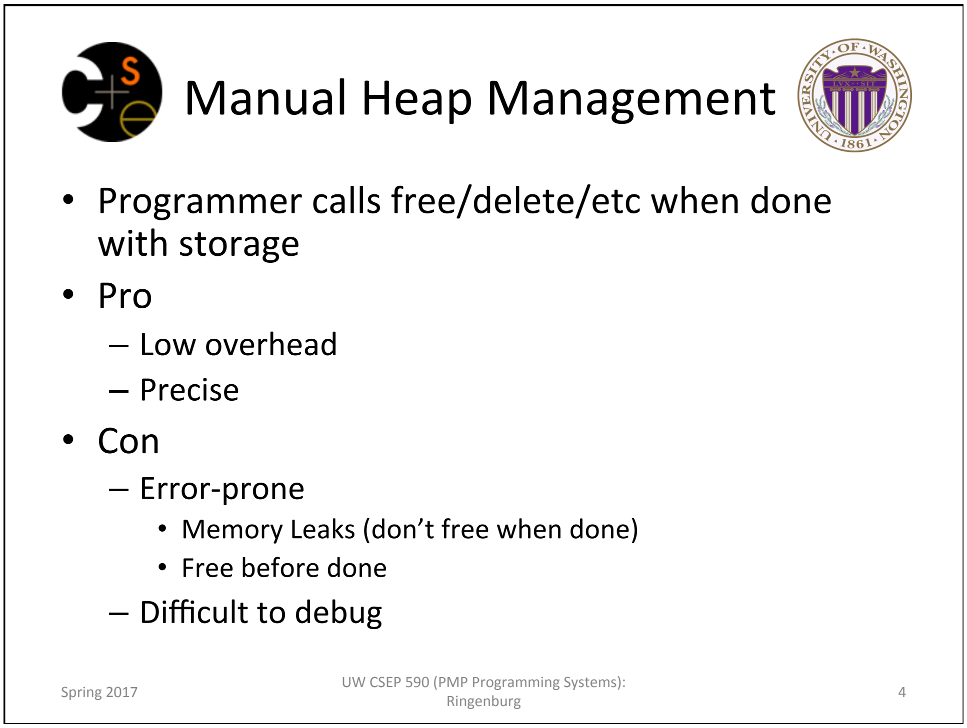
- Presentations
  - Schedule Posted on course web
- Reminder: No class on May 2
- Will try to start catching up on HW grading … sorry (no TA)
- Today: Garbage collection
- May 9: Potpourri of suggested topics. So far I've had suggestions for …
  - Just-In-Time (JIT) compilation
  - Query optimization
  - Type Theory/Type Checking intro – *slightly* outside the charter of this class … but may try to squeeze it in with more of a focus on checking

Garbage Collection References

- First topic we've covered that I haven't implemented (just studied) … but some of you may have?
- Some great references:
  - *Uniprocessor Garbage Collection Techniques*
    Wilson, IWMM 1992 (longish survey)
  - *The Garbage Collection Handbook*
    Jones, Hosking, Moss, 2012 (book)
- Today's slides adapted from Hal Perkins, CSE 401 and 501
  - In turn adapted from slides by Vijay Menon, CSE 501, Sp09
  - Plus additions from other sources as noted within

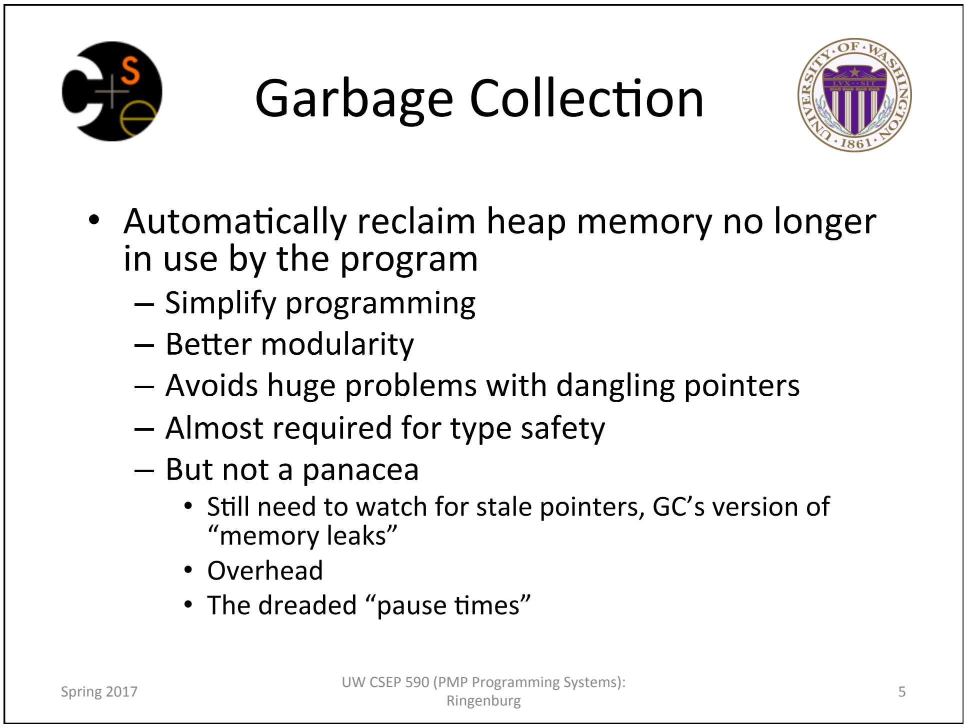
Program Memory

- Typically divided into 3 regions:
  - Global / Static: fixed-size at compile time; exists throughout program lifetime
  - Stack / Automatic: per function, automatically allocated and released (local variables)
  - Heap: Explicitly allocated by programmer
    - Need to recover storage for reuse when no longer needed: Manually or automatically



Manual Heap Management

- Programmer calls free/delete/etc when done with storage
- Pro
  - Low overhead
  - Precise
- Con
  - Error-prone
    - Memory Leaks (don’t free when done)
    - Free before done
  - Difficult to debug



Garbage Collection

- Automatically reclaim heap memory no longer in use by the program
  - Simplify programming
  - Better modularity
  - Avoids huge problems with dangling pointers
  - Almost required for type safety
  - But not a panacea
    - Still need to watch for stale pointers, GC’s version of “memory leaks”
    - Overhead
    - The dreaded “pause times”

Heap Characteristics

- Most objects are small-ish (often < 128 bytes)
- Object-oriented and functional code allocates a huge number of short-lived objects
- Want allocation, recycling to be fast and low overhead
  - Serious engineering required

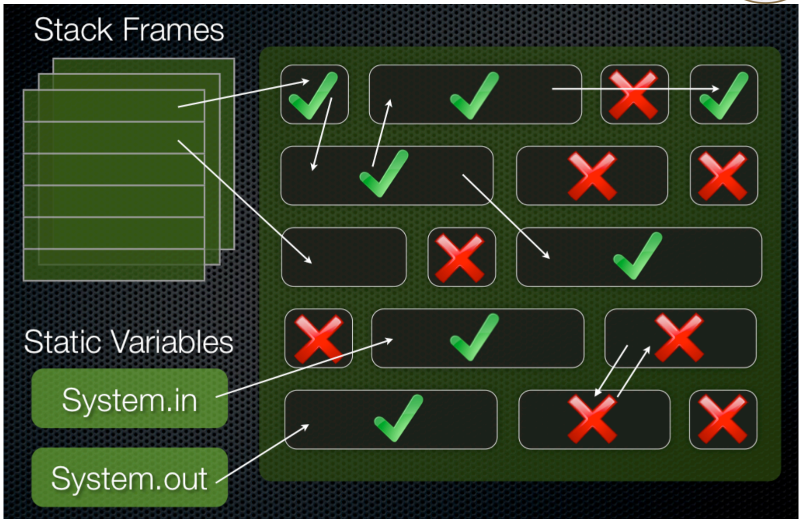
What is Garbage?

- An object is *live* if it is still in use
- Need to be conservative
  - OK to keep memory no longer in use
  - Not ok to reclaim something that is live
- An object is *garbage* if it is not live

Reachability

- *Root set* : the set of global and local (stack/register) variables visible to active procedures
- Heap objects are *reachable* if:
  - They are directly accessible from the root set
  - They are accessible from another reachable heap object (pointers/references)
- Liveness implies reachability (conservative approximation)
- Not reachable implies garbage

Reachability, illustrated



From Douglas Q Hawkins, https://www.slideshare.net/dougqh/understanding-garbage-collection

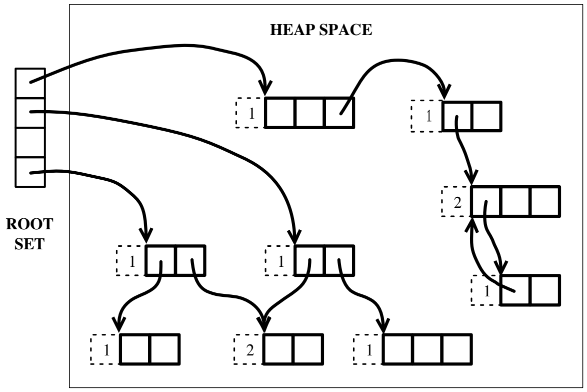
Reachability

- Compiler produces:
  - A *stack-map* at *GC safe points*
    - *Stack map:* enumerates all GC roots (e.g., global variables, stack variables, live registers)
    - *GC safe points:* Points in execution where we are guaranteed to know all roots, and have a consistent heap (e.g., new(), method entry, method exit, etc).
- When a thread reaches a safe point, check if the safe point is needed (e.g., a GC has been scheduled). If so, block.
  - Once all threads blocked at safe point, GC can proceed.

Reference Counting Collectors

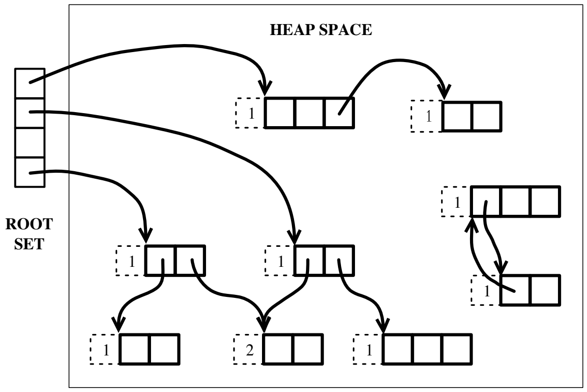
- Keep extra integer associated with every heap object
  - Set to 1 when object allocated
  - Increment when new reference established
  - Decrement when reference disappears (e.g., pointers stack frame/scope goes away; pointer assigned different value)
  - When reference count == 0, can be freed

Reference Counting Example



From “Uniprocessor Garbage Collection Techniques”, Paul R Wilson
1992 International Workshop on Memory Management

Reference Counting: The Cycle Problem



From “Uniprocessor Garbage Collection Techniques”, Paul R Wilson
1992 International Workshop on Memory Management

Reference Counting: Evaluation

- Pros
  - Simple to understand
  - No large pauses to clean: just free anything when its RC gets to 0. Important for real-time applications.
- Cons
  - Cycles!
  - Space inefficient: extra integer per object
  - Time inefficient: operations on every pointer change/allocation/deallocation. Can get rid of some (e.g., local pointer adjustments), but costs still generally higher than tracing collectors.

Tracing Collectors

- Mark the objects reachable from the root set, then perform a transitive closure to find all reachable objects
- All unmarked objects are dead and can be reclaimed
- Various algorithms: mark-sweep, copying, generational…

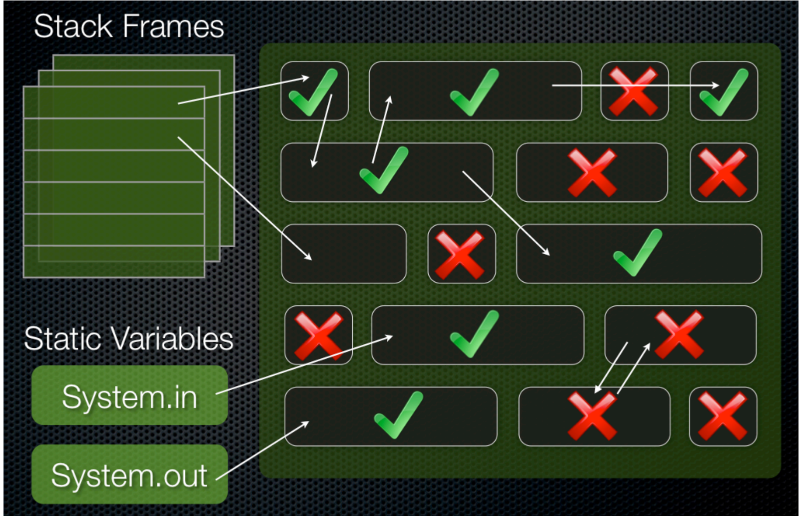
Mark-Sweep Allocation

- Multiple free lists organized by size for small objects (e.g., 8, 16, 24, 32 bytes); additional list for large blocks
  - Regular malloc does exactly the same
- Allocation
  - Grab a free object from the right free list
  - No more memory of the right size triggers a collection

Mark-Sweep Collection

- Mark phase – find the live objects
  - Transitive closure from root set marking all live objects
- Sweep phase
  - Sweep memory for unmarked objects and return to appropriate free list(s)

Mark Phase



Stack Frames

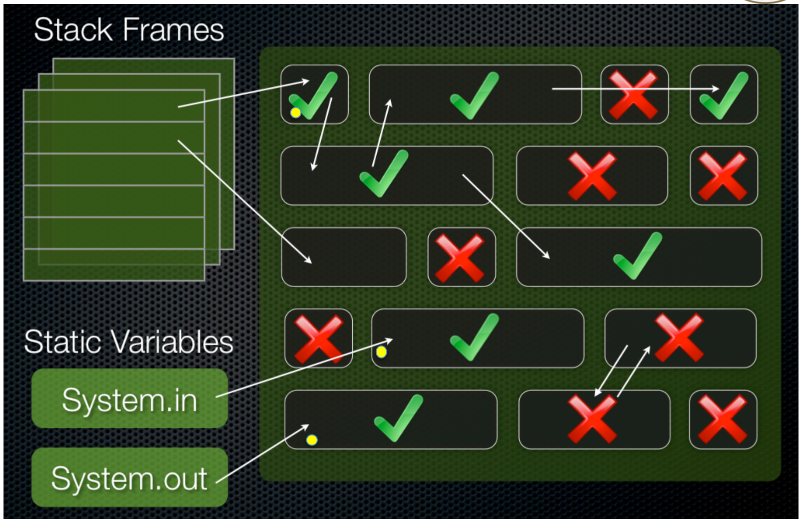
Static Variables

System.in

System.out

From Douglas Q Hawkins, https://www.slideshare.net/dougqh/understanding-garbage-collection
licensed under Creative Commons: Attribution-ShareAlike License (https://creativecommons.org/licenses/by-sa/4.0/)

Mark Phase



Stack Frames

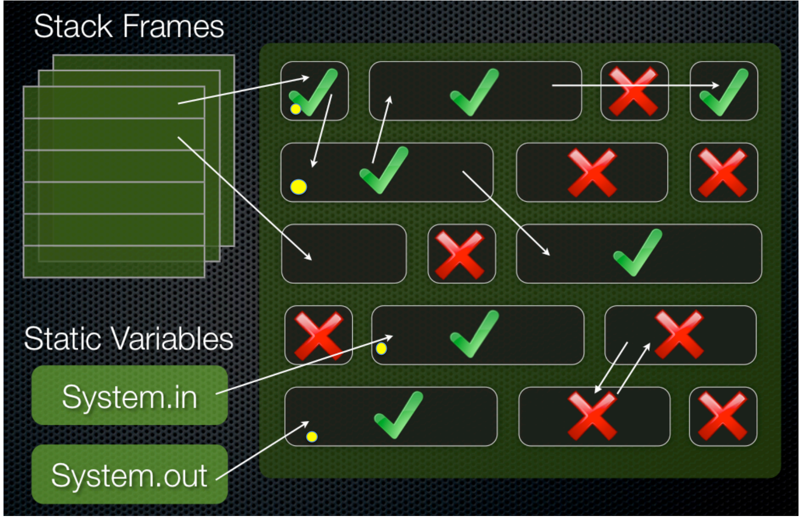
Static Variables

System.in

System.out

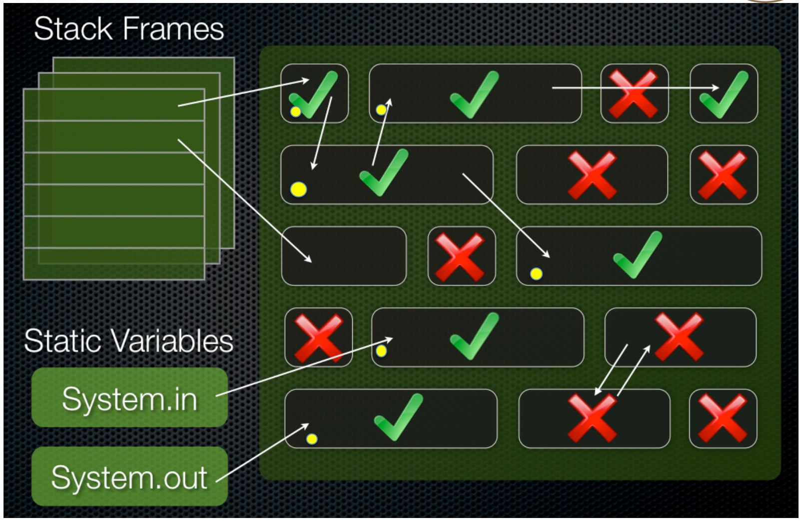
From Douglas Q Hawkins, https://www.slideshare.net/dougqh/understanding-garbage-collection
licensed under Creative Commons: Attribution-ShareAlike License (https://creativecommons.org/licenses/by-sa/4.0/)

Mark Phase

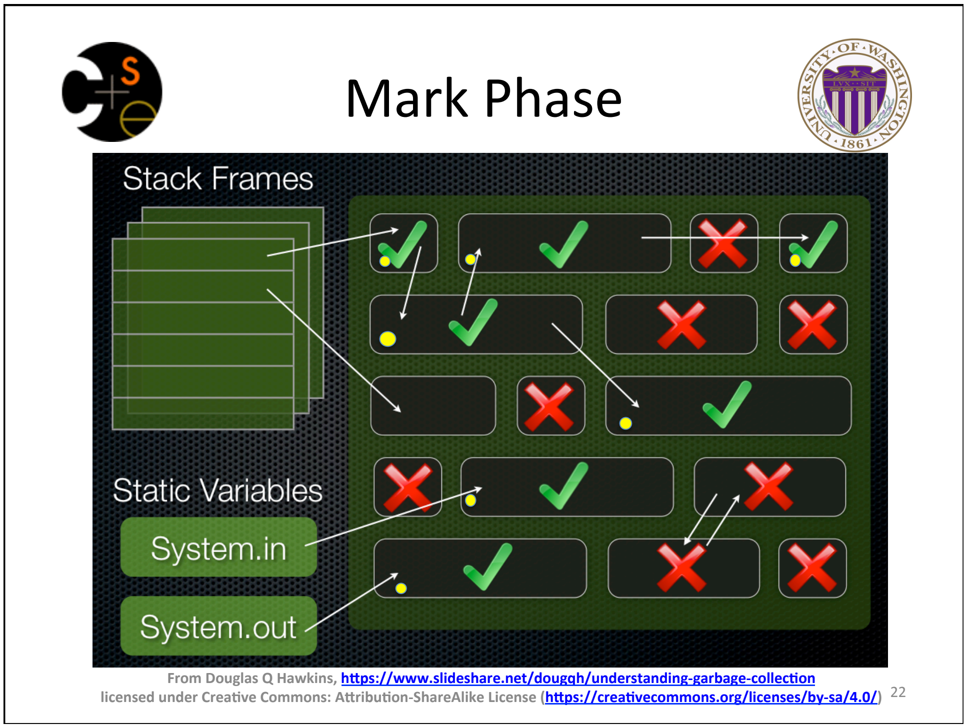


From Douglas Q Hawkins, https://www.slideshare.net/dougqh/understanding-garbage-collection
licensed under Creative Commons: Attribution-ShareAlike License (https://creativecommons.org/licenses/by-sa/4.0/)

Mark Phase

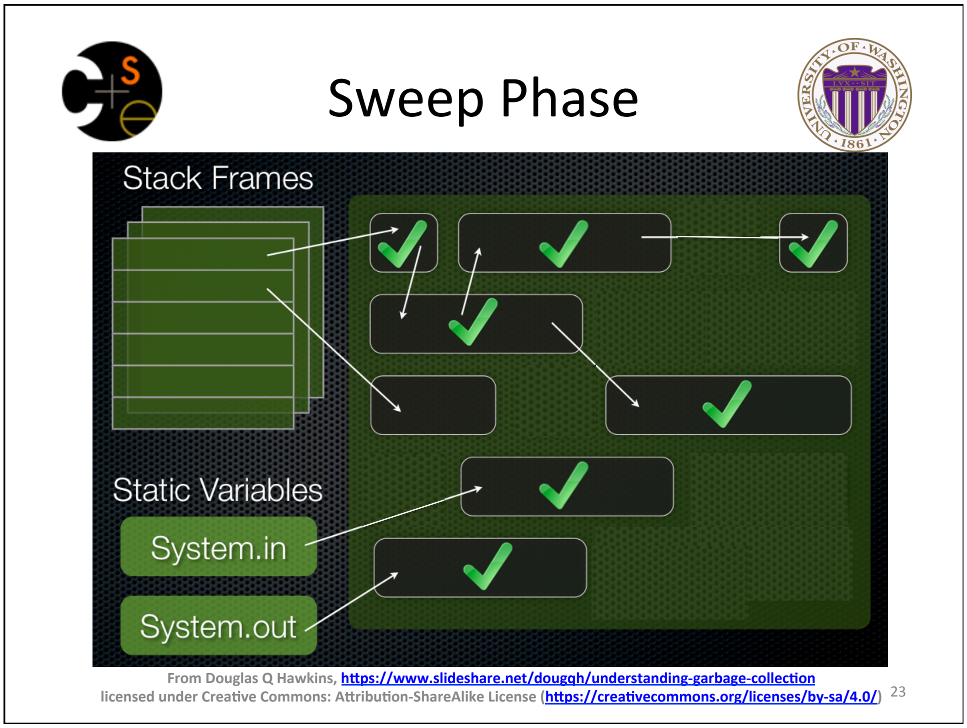


From Douglas Q Hawkins, https://www.slideshare.net/dougqh/understanding-garbage-collection
licensed under Creative Commons: Attribution-ShareAlike License (https://creativecommons.org/licenses/by-sa/4.0/)



Mark Phase

From Douglas Q Hawkins, https://www.slideshare.net/dougqh/understanding-garbage-collection
licensed under Creative Commons: Attribution-ShareAlike License (https://creativecommons.org/licenses/by-sa/4.0/)



Sweep Phase

From Douglas Q Hawkins, https://www.slideshare.net/dougqh/understanding-garbage-collection
licensed under Creative Commons: Attribution-ShareAlike License (https://creativecommons.org/licenses/by-sa/4.0/)

Mark-Sweep Evaluation

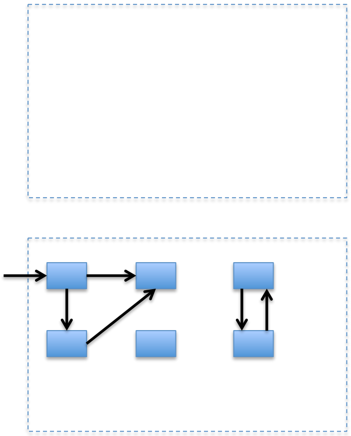
- Pro
  - Space efficiency
  - Incremental object reclamation
- Con
  - Relatively slower allocation time
  - Poor locality of objects allocated at around the same time
  - Redundant work rescanning long-lived objects
  - May lead to fragmentation
    - Sometimes add compaction
  - Long pauses: “Stop the world I want to collect”

Semispace Copying Collector

- Idea: Divide memory in half
  - Storage allocated from one half of memory
  - When full, copy live objects from old half (“from space”) to unused half (“to space”) & swap semispaces (“from” becomes “to”, “to” becomes “from”)
- Fast allocation – next chunk of to-space
- Requires copying collection of entire heap when collection needed

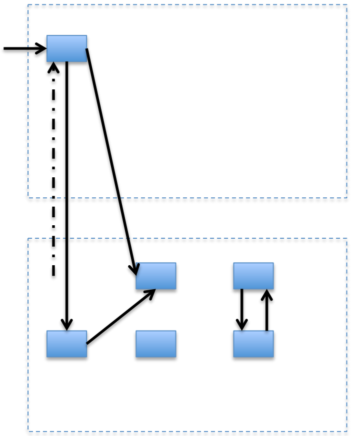
Semispace collection

- Same notion of root set and reachable
- Copy each object when first encountered
- Install forwarding pointers in from-space referring to new copy in to-space
- Transitive closure: follow pointers, copy, and update as it scans
- Reclaims entire “from space” in one shot
  - Swap from- and to-space when copy done



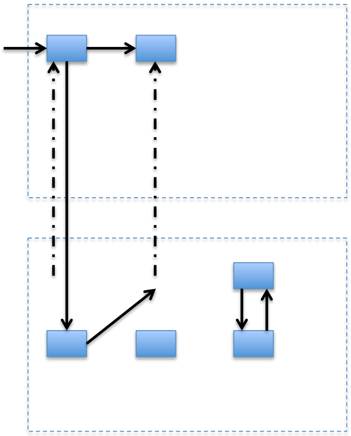
Semispace collection

- Same notion of root set and reachable
- Copy each object when first encountered
- Install forwarding pointers in from-space referring to new copy in to-space
- Transitive closure: follow pointers, copy, and update as it scans
- Reclaims entire “from space” in one shot
  - Swap from- and to-space when copy done



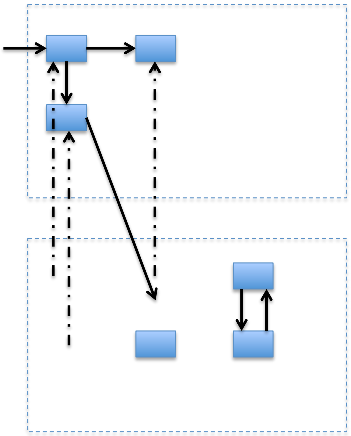
Semispace collection

- Same notion of root set and reachable
- Copy each object when first encountered
- Install forwarding pointers in from-space referring to new copy in to-space
- Transitive closure: follow pointers, copy, and update as it scans
- Reclaims entire “from space” in one shot
  - Swap from- and to-space when copy done



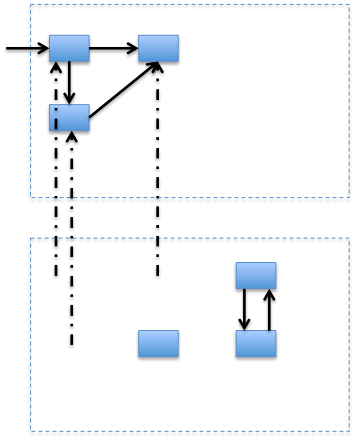
Semispace collection

- Same notion of root set and reachable
- Copy each object when first encountered
- Install forwarding pointers in from-space referring to new copy in to-space
- Transitive closure: follow pointers, copy, and update as it scans
- Reclaims entire “from space” in one shot
  - Swap from- and to-space when copy done



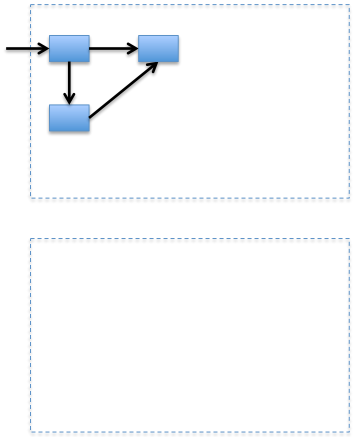
Semispace collection

- Same notion of root set and reachable
- Copy each object when first encountered
- Install forwarding pointers in from-space referring to new copy in to-space
- Transitive closure: follow pointers, copy, and update as it scans
- Reclaims entire “from space” in one shot
    - Swap from- and to-space when copy done



Semispace collection

- Same notion of root set and reachable
- Copy each object when first encountered
- Install forwarding pointers in from-space referring to new copy in to-space
- Transitive closure: follow pointers, copy, and update as it scans
- Reclaims entire “from space” in one shot
    - Swap from- and to-space when copy done



Semispace Copying Collector Evaluation

- Pro
  - Fast allocation
  - Locality of objects allocated at same time
  - Locality of objects connected by pointers (can use depth-first or other strategies during the mark-copy phase)
- Con
  - Wastes half of memory
  - Redundant work rescanning long-lived objects
  - Long pauses: “Stop the world I want to collect”

Generational Collectors

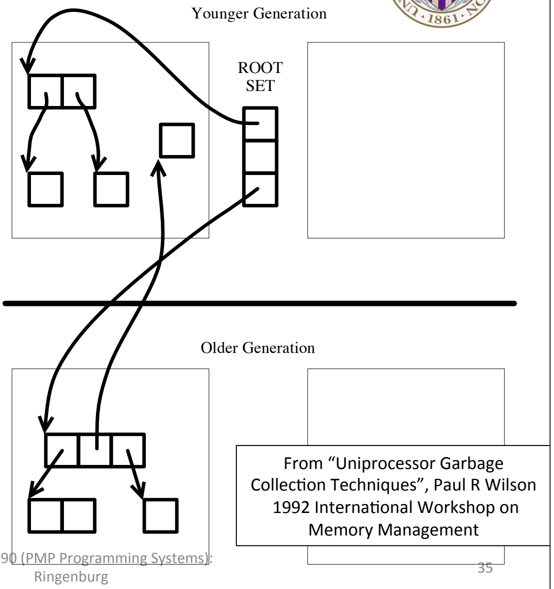
- Generational hypothesis: young objects die more quickly than older ones (Lieberman & Hewitt ‘83, Ungar ‘84)
  - Bimodal distribution – most object have a short life span, but the rest tend to live a very long time
- Most pointers are from younger to older objects (Appel ‘89, Zorn ‘90)
- So, organize heap into young and old regions, collect young space more often

Generational Collectors

- Divide heap into two spaces: young, old
- Allocate new objects in young space
- When young space fills up, collect it and copy surviving objects to old space
  - Refinement: require objects to survive at least a few collections before copying
  - Generally using copying collector for young generation, since small (not too much wasted memory)
- When old space fills, collect both
  - Old space may use different technique, e.g. mark-sweep
- Can generalize to multiple generations

Pointers from Old to New

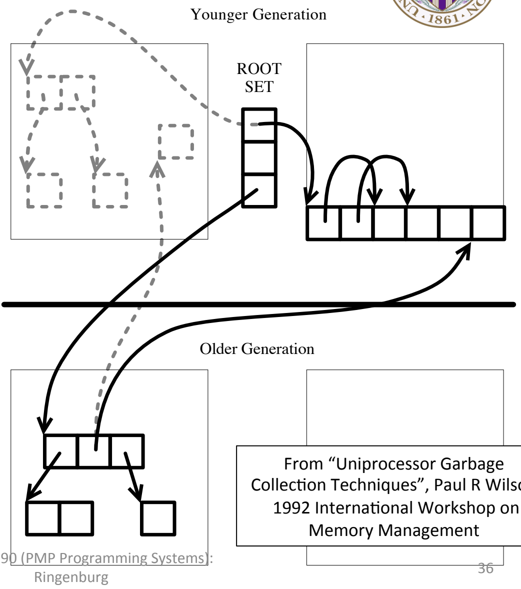
- Pointers from old to new are rare, but do occur
- What do we do during minor GC (young collection)?
  - Must treat these pointers as roots
  - Can use indirection table
  - Or, mark pointers that were changed in old generation



From “Uniprocessor Garbage Collection Techniques”, Paul R Wilson 1992 International Workshop on Memory Management

Pointers from Old to New

- Pointers from old to new are rare, but do occur
- What do we do during minor GC (young collection)?
  - Must treat these pointers as roots
  - Can use indirection table
  - Or, mark pointers that were changed in old generation



From “Uniprocessor Garbage Collection Techniques”, Paul R Wilson 1992 International Workshop on Memory Management

GC Tradeoffs

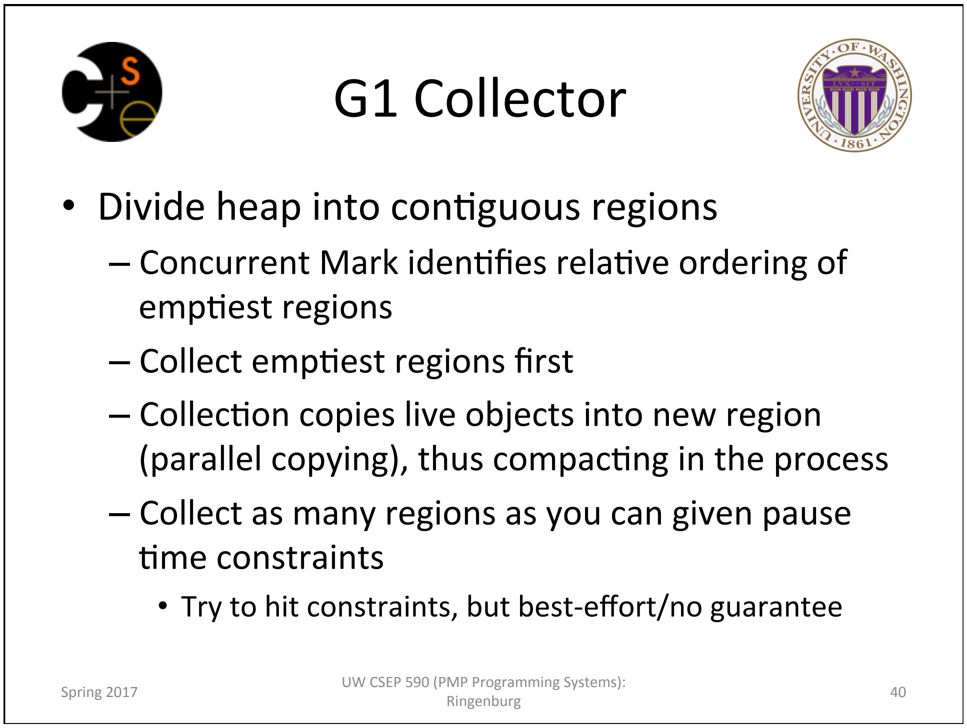
- Performance
  - Mark-sweep often faster than semispace
  - Generational better than both
- Mutator performance
  - Semispace is often fastest
  - Generational is better than mark-sweep
- Overall: generational is a good balance
- But: we still “stop the world” to collect

Enhancements

- Parallel copying collector
  - Multiple threads tracing roots/copying objects. Each thread responsible for a subset of the roots and a segment of the object table
  - Original parallel collector in Hotspot JVM used this for young generation (old generation serial)
- Parallel mark-sweep
  - Same idea, except marking rather than copying. Threads assigned regions of heap
  - To compact: Identify low occupency regions to move objects to. Thread responsible for destination region does copy
  - New parallel collector ("parallel compacting") in Hotspot JVM uses this for old generation (young still uses parallel copying).

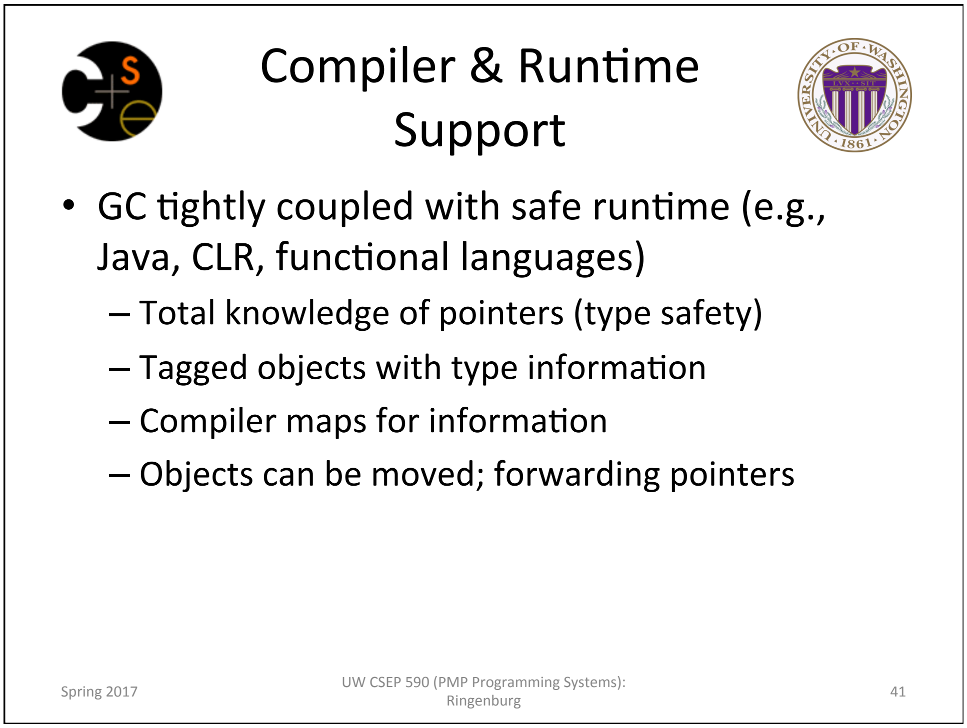
Enhancements, cont

- Concurrent Mark-Sweep (e.g., in Hotspot)
  - Goal: Minimize stop-the-world long pauses. Increased responsiveness.
  - Young Generation: Parallel Copying Collector (young is quick)
  - Old Generation has three phases:
    - **Initial mark**: Short pause to identify object directly reachable from roots
    - **Concurrent mark**: A thread or threads continue to trace and mark *while application continues running*. May miss some objects since heap is changing.
    - **Remark**: Pause while parallel mark visits anything that has changed while concurrent mark was running
    - **Concurrent sweep**: Collect all unmarked objects *while rest of application continues to run*. No compaction.
    - Concurrent phases can also be done incrementally.



G1 Collector

- Divide heap into contiguous regions
  - Concurrent Mark identifies relative ordering of emptiest regions
  - Collect emptiest regions first
  - Collection copies live objects into new region (parallel copying), thus compacting in the process
  - Collect as many regions as you can given pause time constraints
    - Try to hit constraints, but best-effort/no guarantee



Compiler & Runtime Support

- GC tightly coupled with safe runtime (e.g., Java, CLR, functional languages)
  - Total knowledge of pointers (type safety)
  - Tagged objects with type information
  - Compiler maps for information
  - Objects can be moved; forwarding pointers

What about unsafe languages? (e.g., C/C++)

- Boehm/Weiser collector: GC still possible *without* compiler/runtime cooperation(!)
  - If it looks like a pointer, it's a pointer
  - Mark-sweep only – GC doesn't move anything
  - Allows GC in C/C++ but constraints on pointer bit-twiddling

And a bit of perspective…

- Automatic GC has been around since LISP I in 1958
- Ubiquitous in functional and object-oriented programming communities for decades
- Mainstream since Java (mid-90s)
- Now conventional wisdom?

Discussion

- Tracing and Reference Counting … algorithmic duals?
  - (They had to slightly modify the formulation of reference counting)
- Argued that any optimized collector can be viewed as a hybrid
- What are the implications of this Duality?
- What does this imply about the design space?
- Can you think of algorithms that don't fit this model?