CS24: Introduction to Computing Systems

Spring 2015

Lecture 9

Last Time

- Finished covering most of C's abstraction capabilities
  - Dynamic memory allocation on heap, structs, unions
- Began to see something very disturbing:
  - It's *very easy* to write incorrect or unsafe programs in C!
- Unchecked array accesses and buffer overflows:
  - Allow an attacker to crash a program, modify its data in unintended ways, or even execute arbitrary code!
- Other memory management problems as well:
  - Programs don't free memory when they are done with it
  - Programs allocate memory and then access beyond its end
  - Programs access memory after they have freed it
- Idea: *Most people don't actually need all this power!*
  - Provide a simplified memory management abstraction that makes it much easier to implement correct programs

Higher-Level Language Facilities

- Starting to enter realm of higher-level languages
- Much safer programming models:
  - Easier to write correct programs
  - Fewer potential security holes!
- Much greater abstracting capabilities
  - Greater modularity, encapsulation, code reuse
- Also requires much larger run-time support for various facilities
  - Usually slower than C/C++ programs, but much safer!

Higher-Level Language Features (2)

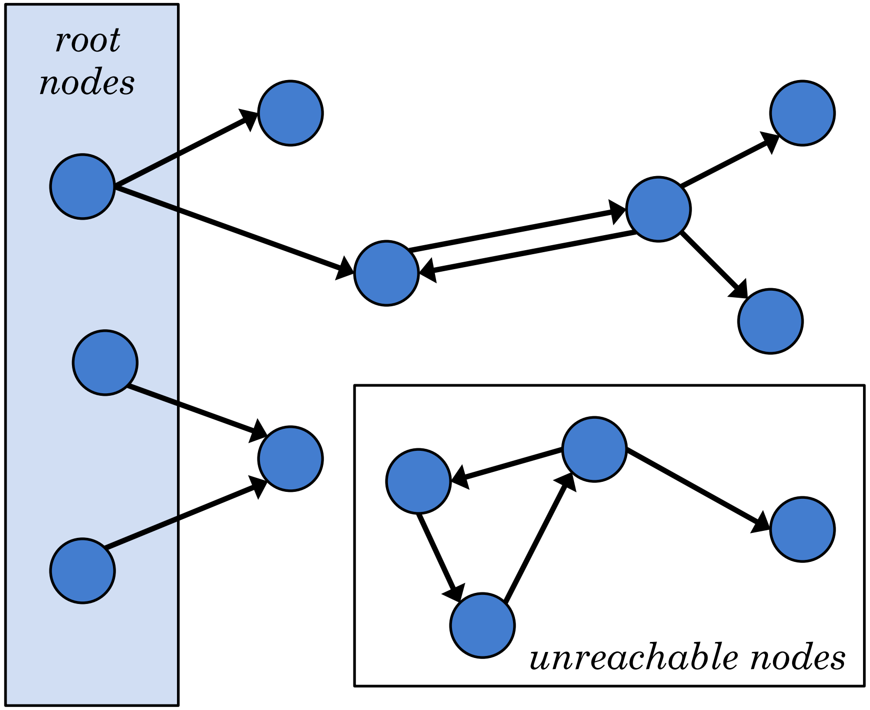
- In next few lectures, will explore three language features:
  - Implicit allocators and garbage collection
  - The object-oriented programming model
  - Exception handling
- Specifically, how to map these features to C, IA32
  - Taking another step up the abstraction hierarchy…
- Examples will draw from Java language features
  - A higher level language than C, with C-like syntax
  - (Not a *huge* step up the abstraction hierarchy…)
  - Includes all of the above language features
  - Itself implemented in C/C++ and assembly language

Implicit Heap Allocators

- Explicit heap allocators rely on the program to release memory when no longer in use
  - *…but programs are notoriously bad at this…*
- A first step towards better programs:
  - Implicit allocators assume the responsibility for identifying when a program is finished with memory
  - Employ a process called garbage collection to identify when a memory block is no longer used by a program
- Use of an implicit allocator eliminates *many* memory management issues for programs
  - Additional overhead for performing garbage collection
  - (A few other issues as well, all relatively minor)

Reachability Graph

- Garbage collectors are often built on the concept of a <u>reachability graph</u>
- Some allocated nodes are root nodes
  - Referenced from global environment, or stored on the stack
- All nodes reachable from the root nodes are live
- Unreachable nodes are garbage
  - May be reclaimed by allocator and reused for subsequent allocations
- How to determine this reachability graph?



Implicit Allocators and Pointers

- What about performing garbage collection in a language like C or C++?
- With such languages, garbage collection can be *very* difficult
- Unfortunately, C/C++ allows:
  - Pointers into the middle of memory blocks
  - Pointers into the middle of structs and arrays
  - Recasting pointers into pointers of other types
  - Recasting pointers into integers and vice-versa
  - All kinds of crazy pointer arithmetic!
- Makes it *very* difficult to build an accurate reachability graph in a C/C++ run-time system

Conservative Garbage Collectors

- In languages like C and C++:
- Garbage collector attempts to identify any value that "looks like" a pointer
  - References a memory location that is currently valid
  - Assume this is a pointer to memory that is in use
- If the value isn't actually a pointer?
  - Only real drawback is that the garbage-collector thinks memory is in use, when it really isn't in use
  - (Hopefully) doesn't happen often enough to be a problem
- <u>Conservative</u> garbage collection:
  - All reachable blocks are identified as reachable
  - Some unreachable blocks are also identified as reachable
  - **Not all garbage is reclaimed.** But we can live with that.

Precise Garbage Collectors

- Another approach is to *strictly control* how programs can use and manipulate pointers
  - Specify rules on casting pointers, and disallow casting to/from non-pointer types
  - Completely forbid pointer arithmetic!
- Allows <u>precise</u> garbage collection
  - Garbage collector can determine the reachability graph *exactly*, for all memory blocks in the heap
  - All garbage can be reclaimed by the allocator
- Given other issues with pointer manipulation (e.g. buffer overflows), *makes tons of sense* to constrain pointers this way!!!

Java References

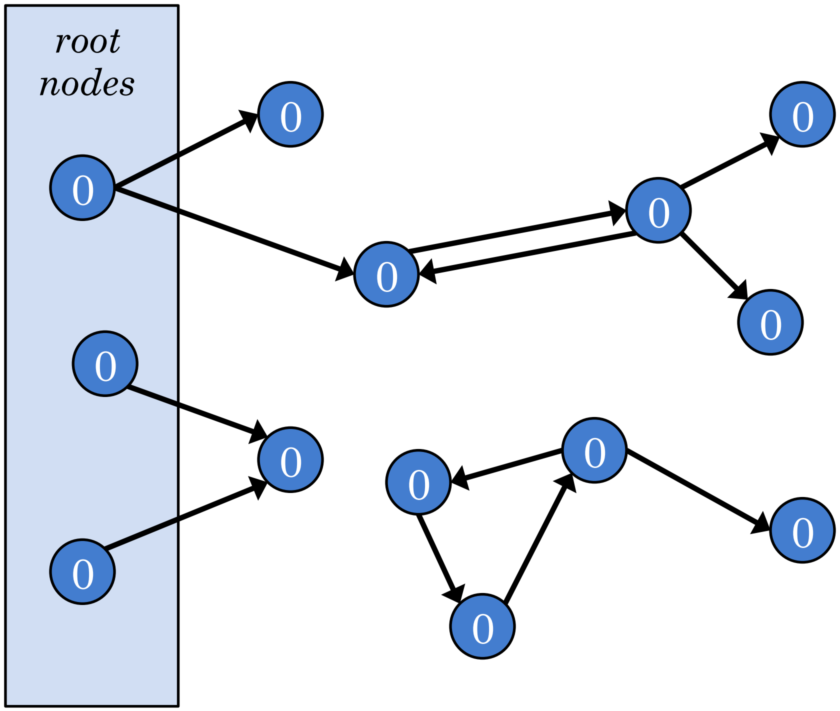
- Like many higher-level languages, Java includes references
- A "reference" is simply a means for looking up and accessing a particular object
  - A pointer is a very primitive kind of reference: it contains the exact memory address of the object
- Introduce a higher-level <u>reference</u> abstraction:
  - The reference is opaque to the program!
  - Programs can no longer directly access or manipulate the memory address associated with a reference
  - (The reference's type can also only be manipulated in very specific, controlled ways.)
  - Gives the run-time system much greater flexibility in managing memory (e.g. moving allocated blocks)

INDIRECTION

- **Indirection** is a very important technique used in computer system design
  - The ability to reference something using a name or other value that represents the *actual* value
- Have already seen this technique multiple times
  - e.g. "branch to register," labels as jump-targets in code, using pointers to access values, jump tables, …
- References allow programs to *indirectly* access objects, while the runtime directly accesses them
- "All problems in computer science can be solved by another level of indirection."
  - David Wheeler, inventor of the subroutine

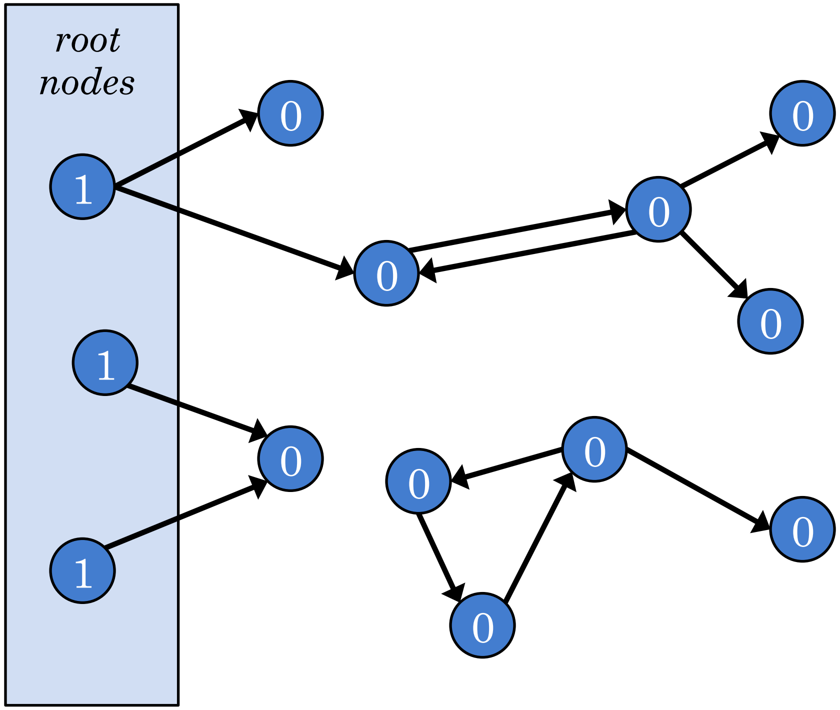
Garbage Collection Algorithms

- Variety of algorithms used for garbage collection
- Simplest algorithm is called “mark and sweep”
- Every object has a flag associated with it
- Initially: all flags are 0
- Two phases:
  - First phase involves traversing entire object graph, marking all reachable objects
  - Second phase involves removing all unreachable objects



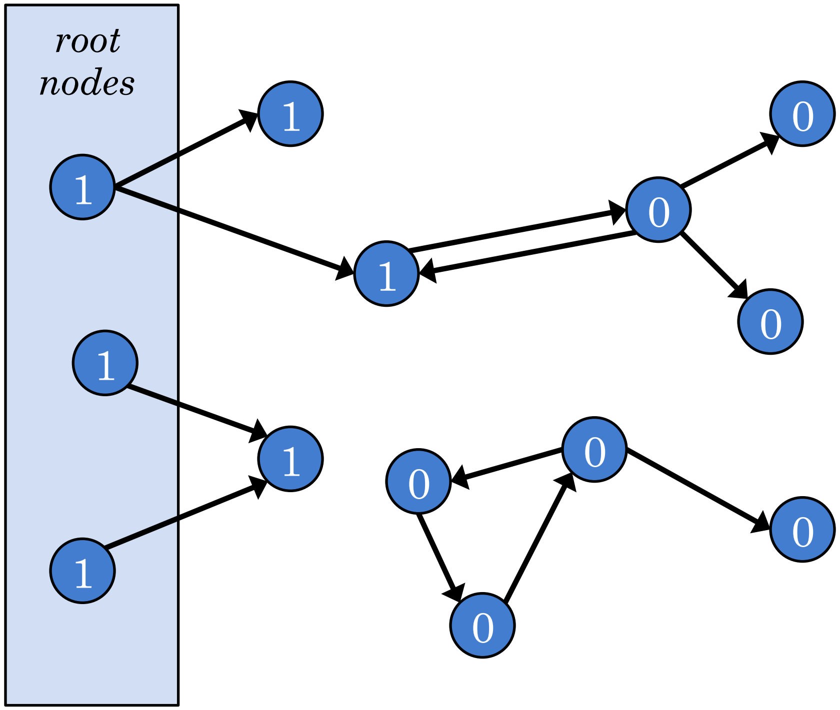
Mark-and-Sweep (1)

- Start by marking root nodes as reachable
- (Note: would implement marking phase as depth-first traversal, not breadth-first…)



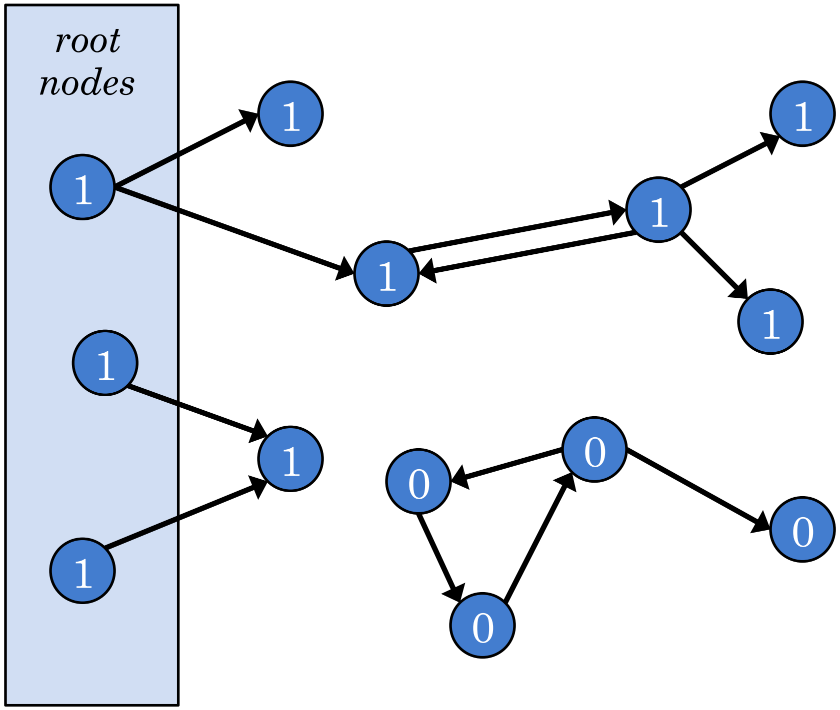
MARK-AND-SWEEP (2)

- Next, nodes reachable from root nodes



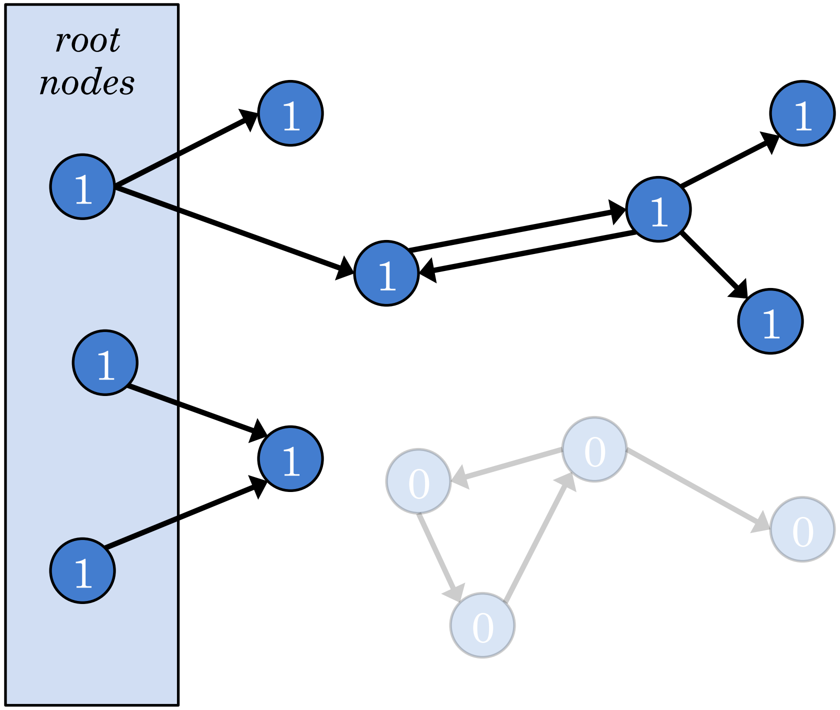
Mark-and-Sweep (3)

- Continue until all reachable nodes are marked.
- Now, any node with a 0 is unreachable, and may be reclaimed.



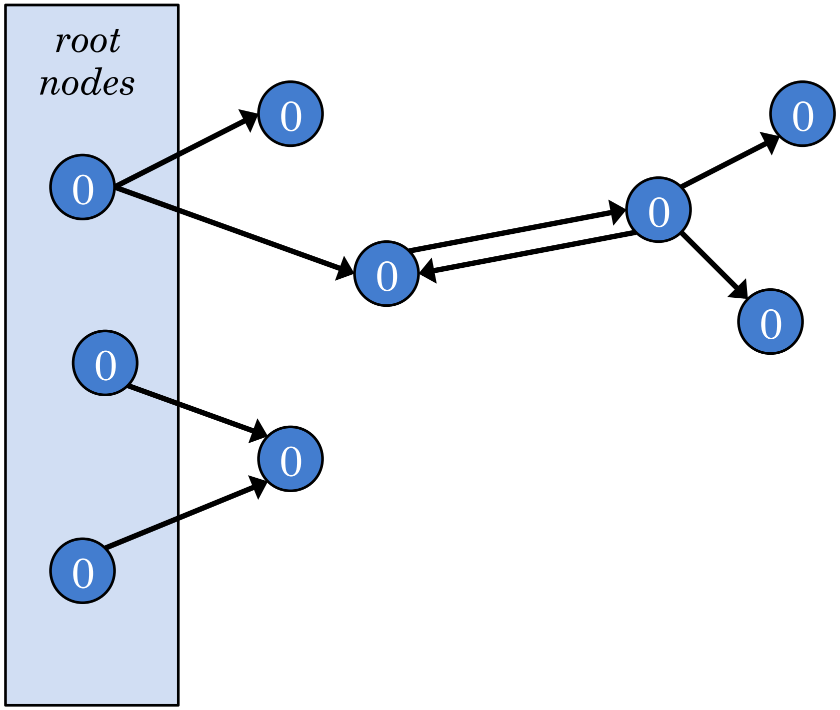
Mark-and-Sweep (4)

- Second phase:
  - If object is unreachable, reclaim it.
  - (If object is reachable, reset flag to 0 for next time.)



Mark-and-Sweep (5)

- Final result:



Mark-and-Sweep Characteristics

- This GC algorithm has several drawbacks
- Most important one:
  - To ensure that all garbage is identified, the program cannot run while the garbage collector is working!
  - Even on a multicore system, program and GC cannot run concurrently
- This is called a "stop-the-world" garbage collector
  - Entire program must be suspended while garbage collection is performed
- Clearly unacceptable for applications where response-time is critical
  - e.g. real-time applications, interactive applications

Garbage Collector Characteristics

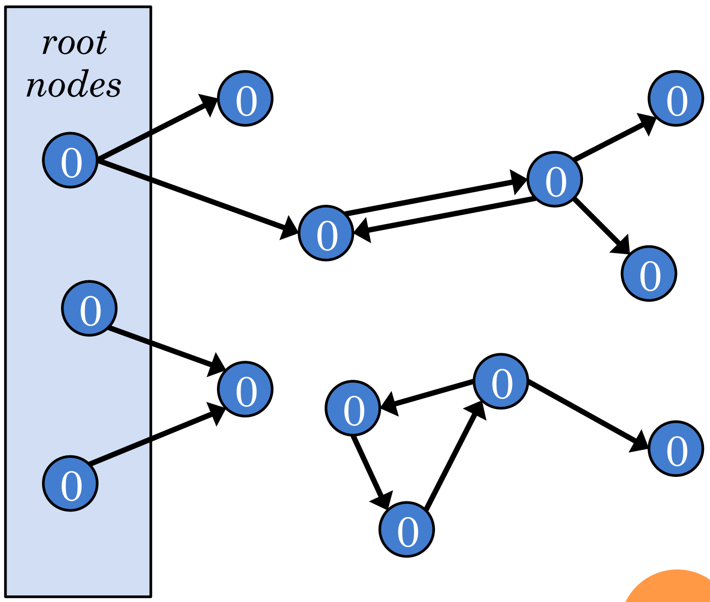
- Several kinds of garbage collector characteristics
- Stop-the-world GC vs. concurrent GC
  - Stop-the-world garbage collectors must suspend the program while performing garbage collection
  - Concurrent garbage collectors work in the background, while the program continues to run
    - Algorithm may also require "stop-the-world" phases, but they are kept as short as possible
    - Frequently, the "stop-the-world" phases are parallelized to minimize wait-time
- Serial GC vs. Parallel GC
  - Serial garbage collectors are not able to use multiple processors during GC phases
  - Parallel collectors are able to employ multiple processors to speed up collection phases

GC CHARACTERISTICS (2)

- Compacting GC vs. non-compacting GC
  - Non-compacting collectors do nothing with the remaining live objects
    - This is the only real option for languages that allow programs to use explicit pointers into memory
    - Memory fragmentation can become a big problem!
  - Compacting garbage collectors move remaining live objects together, maximizing size of free space
  - Only possible to compact memory if language doesn't expose explicit pointers to programs!
    - Another reason to only expose opaque references

Mark-and-Sweep Characteristics (2)

- In what situations is mark-and-sweep garbage collection the least expensive?
  - (Compacting mark-and-sweep, in particular?)
- If most objects are not reclaimed, mark-and-sweep will be relatively inexpensive
  - Mark phase always touches *all* objects, regardless…
  - Sweep phase will have to do less work if most objects don't need reclaimed
  - Particularly true if GC must also compact memory!



Other GC Strategies

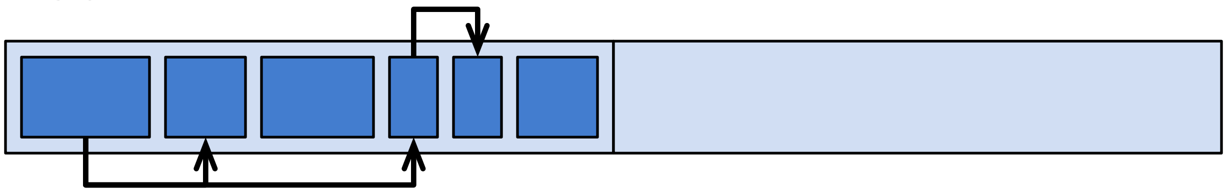
- Mark-and-sweep garbage collection must do a lot of work if many objects are deleted during sweep:
  - Must deallocate each garbage object individually (this overhead definitely adds up)
  - Must compact live objects to avoid fragmentation
- Mark-and-sweep prefers long-living objects ☺
- Another strategy: Copying garbage collectors
  - Instead of compacting live objects together, live objects are all copied ("evacuated") to another, contiguous region of memory
  - Compaction is performed automatically!
  - All dead objects can be deallocated in one operation!

Stop-and-Copy Garbage Collectors

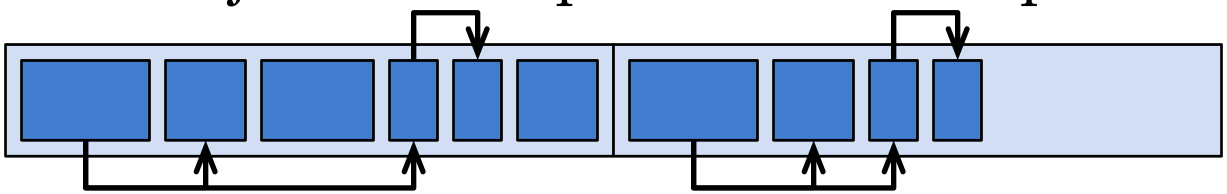
- Stop-and-copy garbage collectors divide memory into two regions: "from-space" and "to-space"
- New objects are allocated in the to-space
- When to-space becomes full, it becomes the "from-space," and vice versa:
  - Starting with root objects, all reachable objects are copied from from-space into to-space
  - At end, entire from-space can be reclaimed at once
- Program resumes execution, using new to-space
- Automatically compacts live objects, but also effectively halves memory available to programs

Stop-and-Copy Example

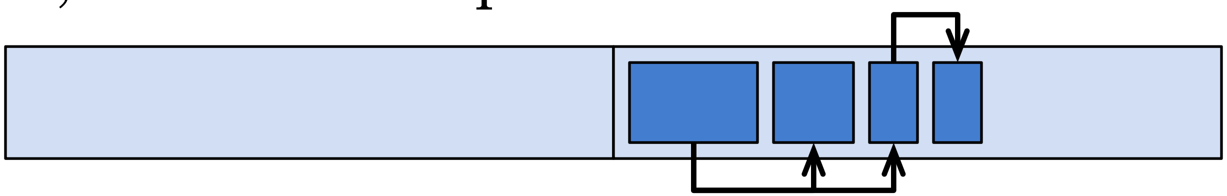
- Memory divided into from-space and to-space
- New objects are allocated in the to-space, until it becomes full



- Program is stopped; starting with root objects, reachable objects are copied to new to-space

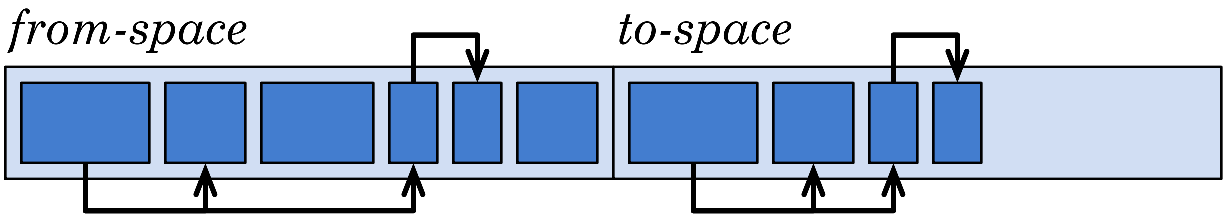


- At end, entire from-space is reclaimed



STOP-AND-COPY GC

- Stop-and-copy garbage collection is fast when:
  - Not a lot of objects live through the GC phase!



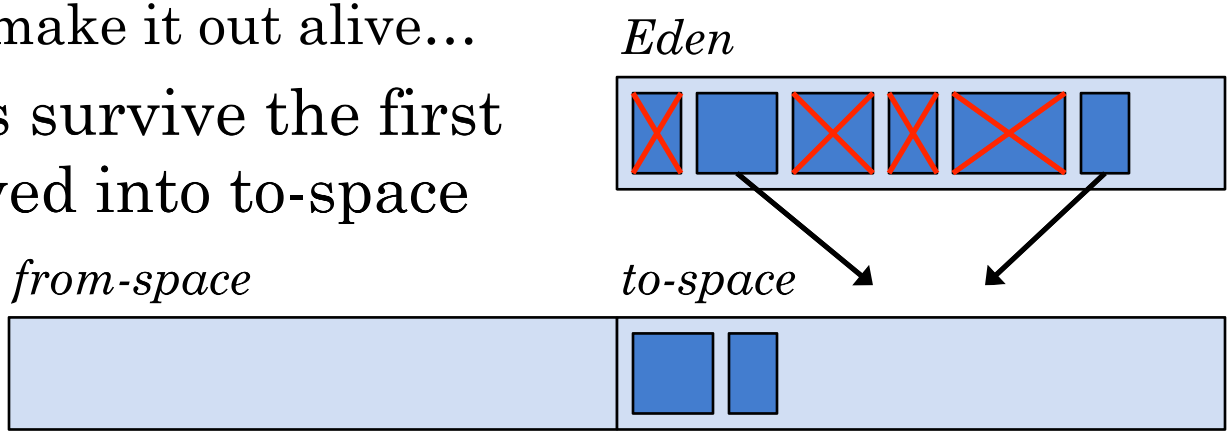
  - The fewer objects you have to copy, the better.
- Stop-and-copy prefers short-lived objects ☺
- Observation: Different garbage collection algorithms are best for different situations…
- Questions:
  - Do all heap-allocated objects behave "the same" with respect to garbage collection?
  - Are there differences that we can take advantage of?

Generational Garbage Collection

- Some empirical observations about programs:
- Most objects in a program are very short-lived
  - e.g. used for local variables, intermediate results, etc.
  - "Most objects die young."
- Longer-lived objects generally do not reference shorter-lived ones
- These ideas called the *generational hypothesis*
  - (Also "infant mortality," but that's just macabre.)
- Idea:
  - Design a garbage collector that takes advantage of this behavioral characteristic of OO program state
  - Called <u>generational garbage collection</u>
  - Optimization: "Make the common case fast."

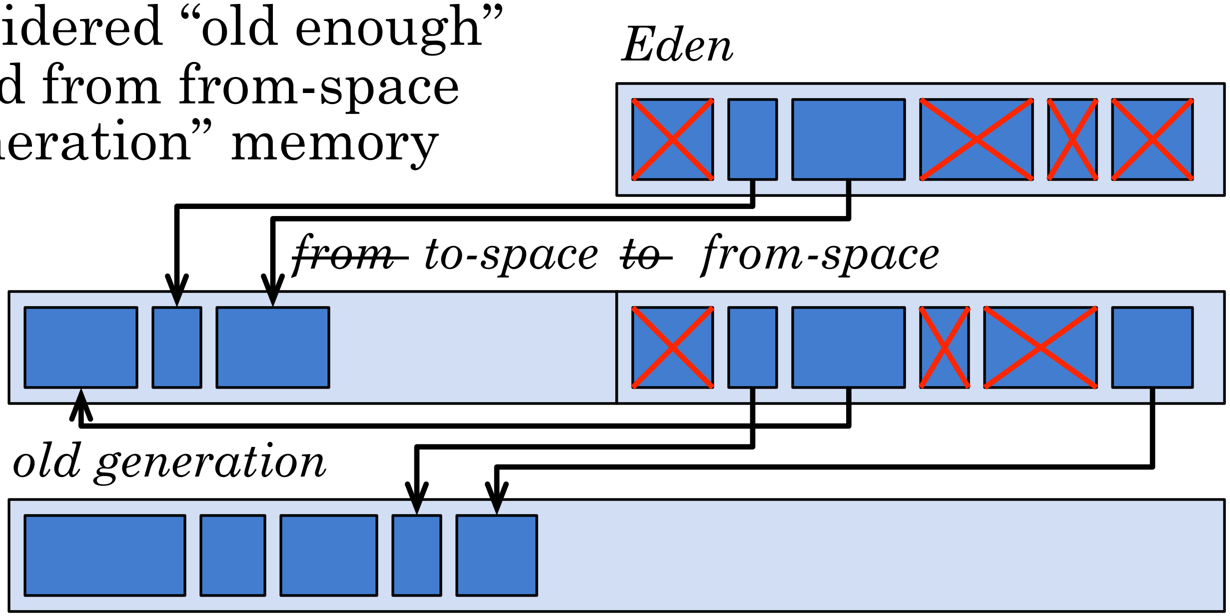
Sun Hot-Spot Generational GC

- Sun Java VMs implement generational garbage collection
- Divides objects into "young objects," "old objects"
- Young objects kept in a separate memory area
  - Uses stop-and-copy GC, since most will not live long
  - Three memory areas: Eden, and two survivor spaces
- Newest objects are allocated in Eden
  - Most never make it out alive…
- If new objects survive the first GC pass, moved into to-space



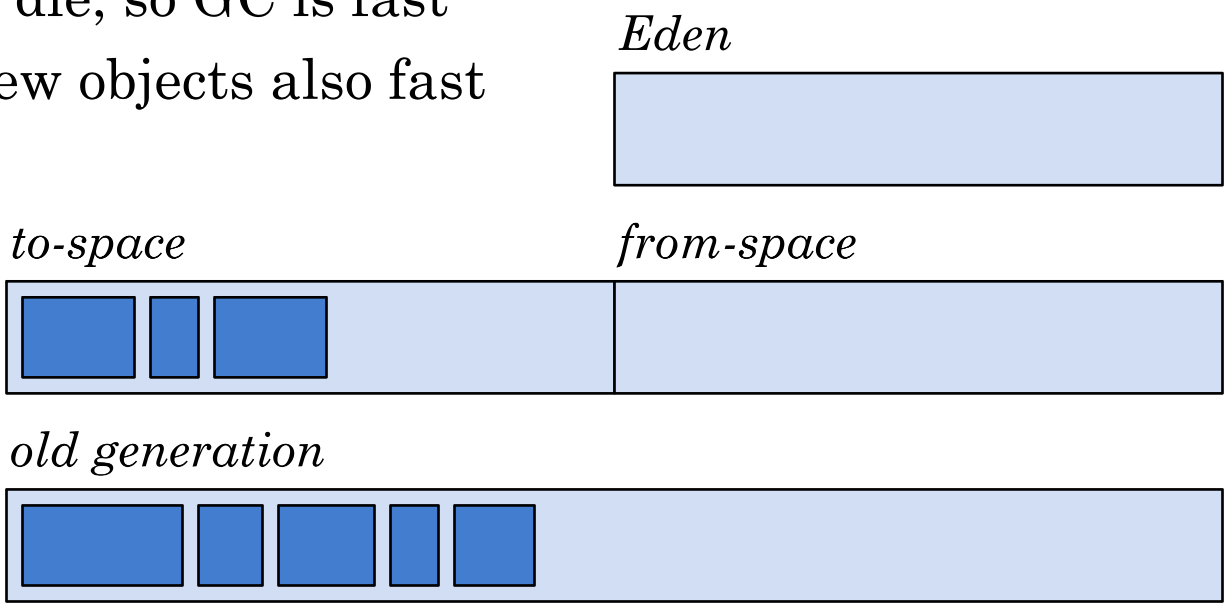
Sun Hot-Spot Generational GC (2)

- Survivor spaces for slightly older "young objects"
  - Give young objects "additional chances to die" before they are considered "old objects"
- As before, when to-space fills up, turn into from-space, and perform stop-and-copy GC
- An important difference:
  - Objects considered "old enough" are promoted from from-space into "old generation" memory



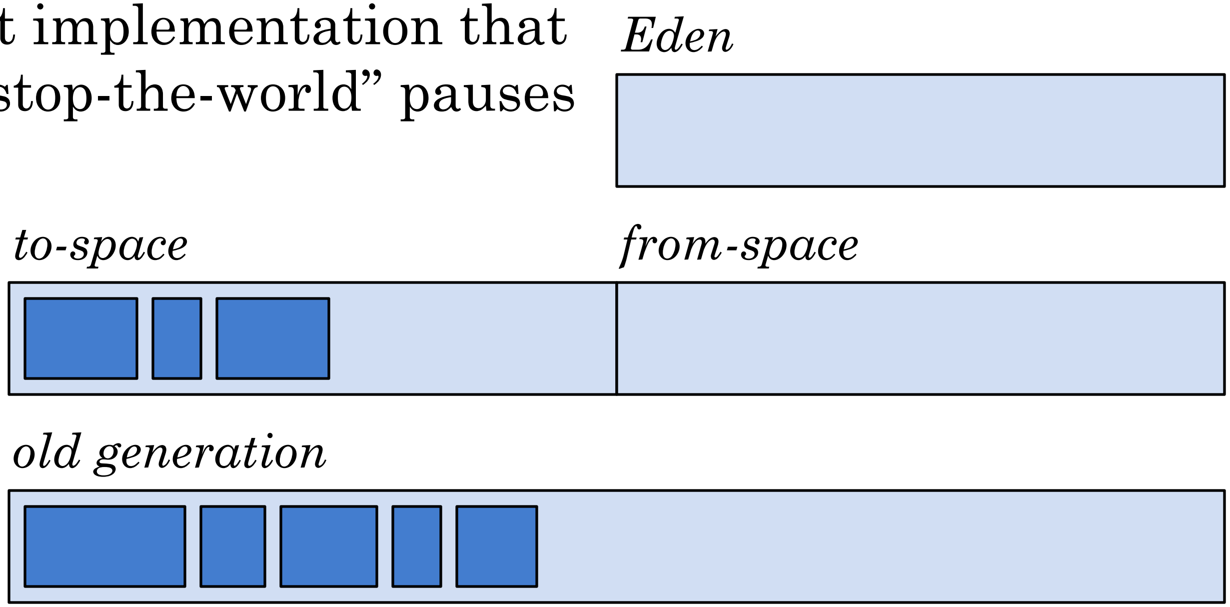
Sun Hot-Spot Generational GC (3)

- After young-object garbage collection, both Eden and the from-space are now empty
  - Objects have either been reclaimed, or copied to another memory area
- Two benefits of stop-and-copy GC here:
  - Most objects die, so GC is fast
  - Allocating new objects also fast



Sun Hot-Spot Generational GC (4)

- Finally, old generation also needs periodic GC
- Most of these objects are expected to survive…
  - Stop-and-copy garbage collection is not appropriate!
- Old generation is managed with a compacting mark-and-sweep algorithm
  - A concurrent implementation that minimizes "stop-the-world" pauses



Generational Garbage Collection

- Generational garbage collection is very complex!
- Takes advantage of two important details:
  - Different garbage collection algorithms are good in different situations
  - Program state tends to fall into two major categories: young, short-lived objects, and old, long-lived objects
- Provides a much more effective garbage collection system than the individual GC algorithms could possibly provide on their own!

General-Purpose Solutions

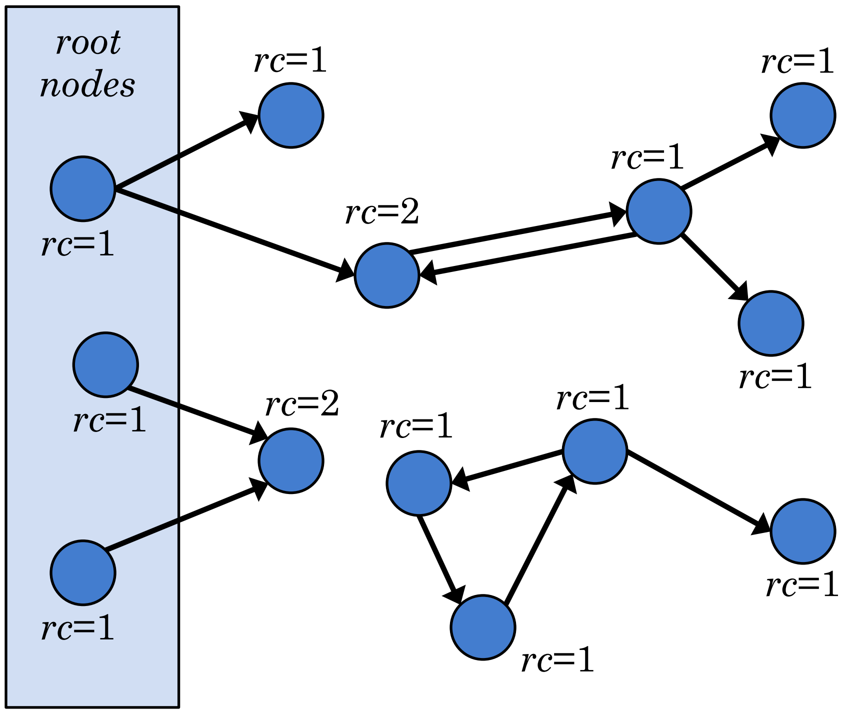
- Another very important system-design pattern: creating general solutions from specialized ones
- Can frequently solve a problem in multiple ways
  - e.g. mark-and-sweep GC vs. stop-and-copy GC
  - Each solution works well in different situations
- We want our computers to be general-purpose…
- We want our operating systems to support a wide range of program behaviors and usage scenarios
- The most powerful, generic solutions frequently blend multiple techniques in a very elegant way
  - Generational garbage collection is a great example!
  - This is a <u>common theme</u> in computer system design

Reference Counting

- Some implicit allocators are based on reference counting, instead of on a reachability graph
- Each object keeps a reference count
  - A simple integer count of how many other objects reference the object
- When code first references an object, its reference-count is automatically incremented
- When code finishes working with the object, its reference count is automatically decremented
- When an object's reference count hits zero, it is automatically reclaimed
  - Instead of periodic, complicated garbage-collection sweeps, objects are immediately reclaimed

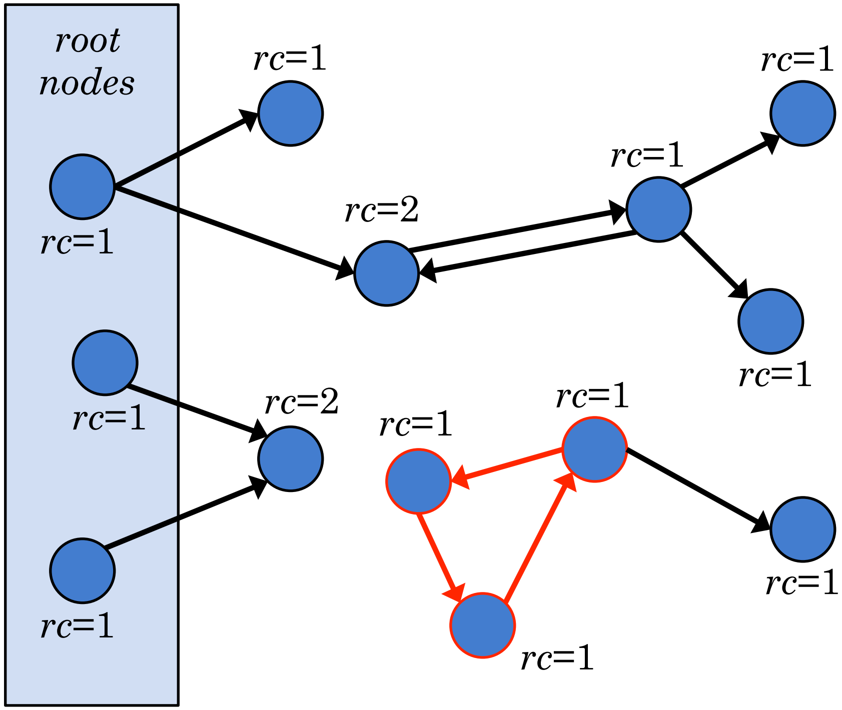
Reference Counting and Cycles

- Major drawback of reference counting:
  - Cannot properly release cycles of objects!
- Our earlier example:
  - All objects have a nonzero reference-count…
  - …but some of the objects are unreachable!
- Despite this limitation, many systems still use reference-counting for automatically freeing objects
  - e.g. the Python runtime



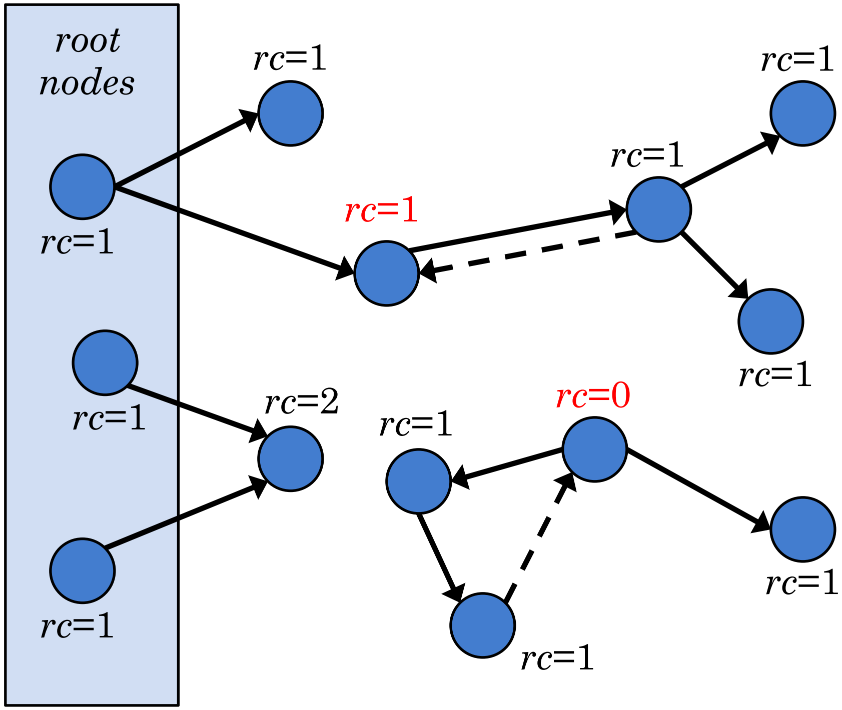
Reference Counting and Cycles (2)

- Several techniques for dealing with cycles
- Cycle detection algorithms
  - Use more standard reachability-graph techniques to detect and reclaim cycles
  - Since it's only needed for a subset of objects, won't have a heavy impact
  - Approaches usually focus on identifying objects that *could be* part of a cycle, and starting from there
- More complex allocator, but keeps the coder's life simpler!



Reference Counting and Cycles (3)

- Another approach: weak references
  - Simply don't allow cycles in reference graph
  - When objects need to refer to each other, one object uses a <u>weak reference</u>
    - Doesn't increment target object's reference count
    - Target of a weak reference may go away unexpectedly
- Properly breaks cycles…
- Programmer must design the program to use weak references properly
  - Can be *very* difficult to get this correct!



Ref-Counting, Garbage Collection

- Other drawbacks with reference counting
  - Cost of incrementing and decrementing reference-counts really adds up
  - Languages with garbage collection don't incur this overhead
  - Some ways to optimize away reference-count updates
- Nonetheless, still quite a common approach
  - Easy to implement!

Summary

- Implicit allocators allow us to eliminate *many* memory management issues
  - Provide a simplified abstraction to programs
- Programs allocate memory, but implicit allocator uses garbage collection to determine when to free
  - e.g. mark and sweep, stop and copy
  - More advanced allocators blend these techniques to create very powerful, general-purpose approaches, e.g. generational garbage collection
- Many languages replace pointers with references
  - Program indirectly references objects, while runtime can still directly access and manipulate them
  - Eliminates many other kinds of serious security bugs!