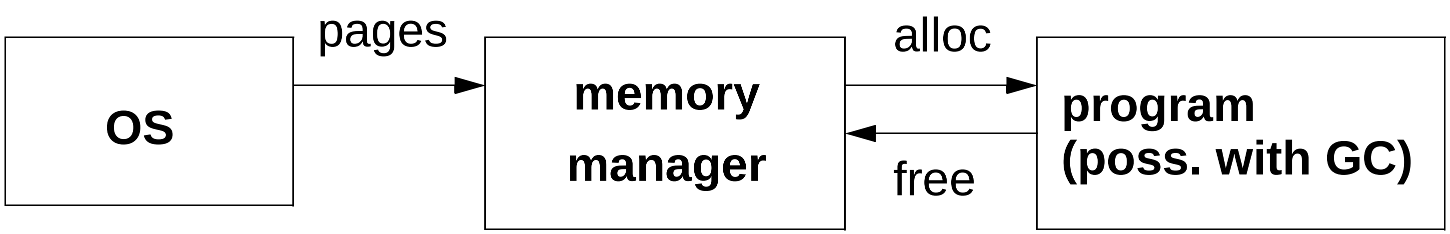
Lecture 14

Introduction to Garbage Collection

Readings: Chapter 7.4-7.6.4

I. Basic Memory Management



- **Tasks of memory manager**
  keep track of free space and respond to malloc and free

- Starting point: One large contiguous space

- **alloc:**
  - find big enough hole, get space from OS if neccessary
  - decide where to allocate, creates small holes if not a perfect fit

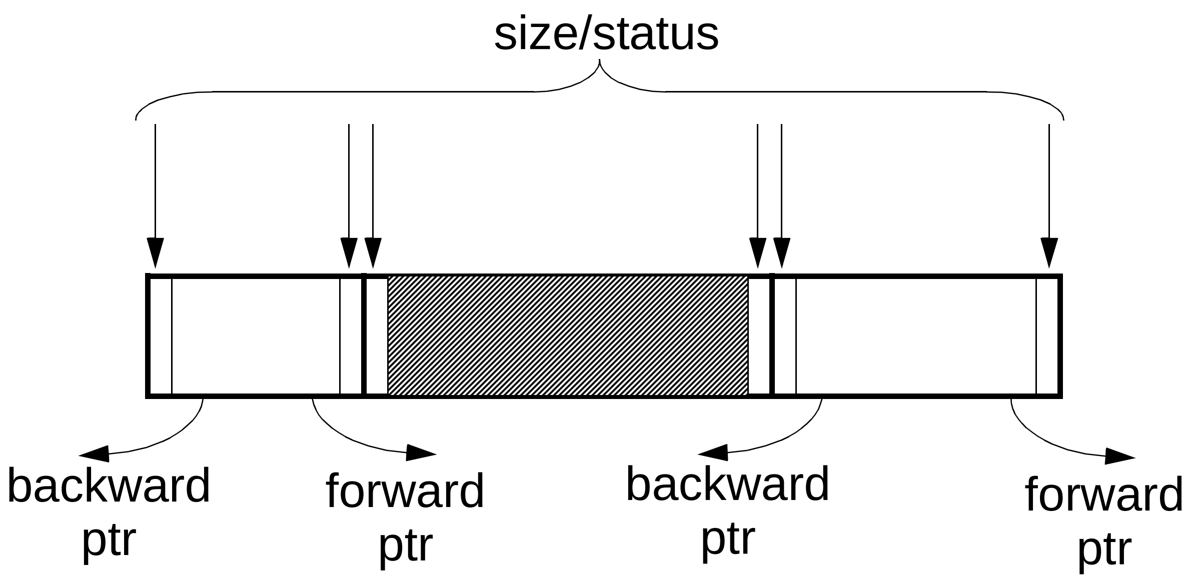
- **free:**
  - coalesce adjacent free space to create larger space

Allocation Algorithm

- **Performance consideration**
  - Space efficiency
    - too many small holes that cannot be used
  - Time
    - dominated by allocations of small chunks
      (cost of large chunk amortized by more computation)
  - Spatial locality
    - more compact is better (best if all space is coalesced)
- **Algorithms**
  - best fit: best utilization
  - next fit:
    - improves spatial locality because
      consecutively allocated data tend to have similar life times

Two Data Structures

- **Boundary tags (for coalescing)**
- **Doubly-linked embedded list (for keeping space in ascending-size order)**



II. Why Automatic Memory Management?

- **Perfect**

	live	dead
not deleted	✓	–––
deleted	–––	✓

- **Manual management**

	live	dead
not deleted		
deleted		

- **Assume for now the target language is Java**

What is Garbage?

When is an Object not Reachable?

- **Mutator (the program)**
  - New / malloc: (creates objects)
  - Store p in a pointer variable or field in an object

  - Load
  - Procedure calls

- **Important property**
  - once an object becomes unreachable, stays unreachable!

How to Find Unreachable Objects?

III. Reference Counting

- **Free objects as they transition from "reachable" to "unreachable"**
- **Keep a count of pointers to each object**
- **Zero reference -> not reachable**
  - When the reference count of an object = 0
    - delete object
    - subtract reference counts of objects it points to
    - recurse if necessary
- **Not reachable -> zero reference?**
- **Cost**
  - overhead for each statement that changes ref. counts

IV. Why is Trace-Based GC Hard?

- **Reasons**
  - Requires complementing the reachability set - that's a large set
  - Interacts with resource management: memory

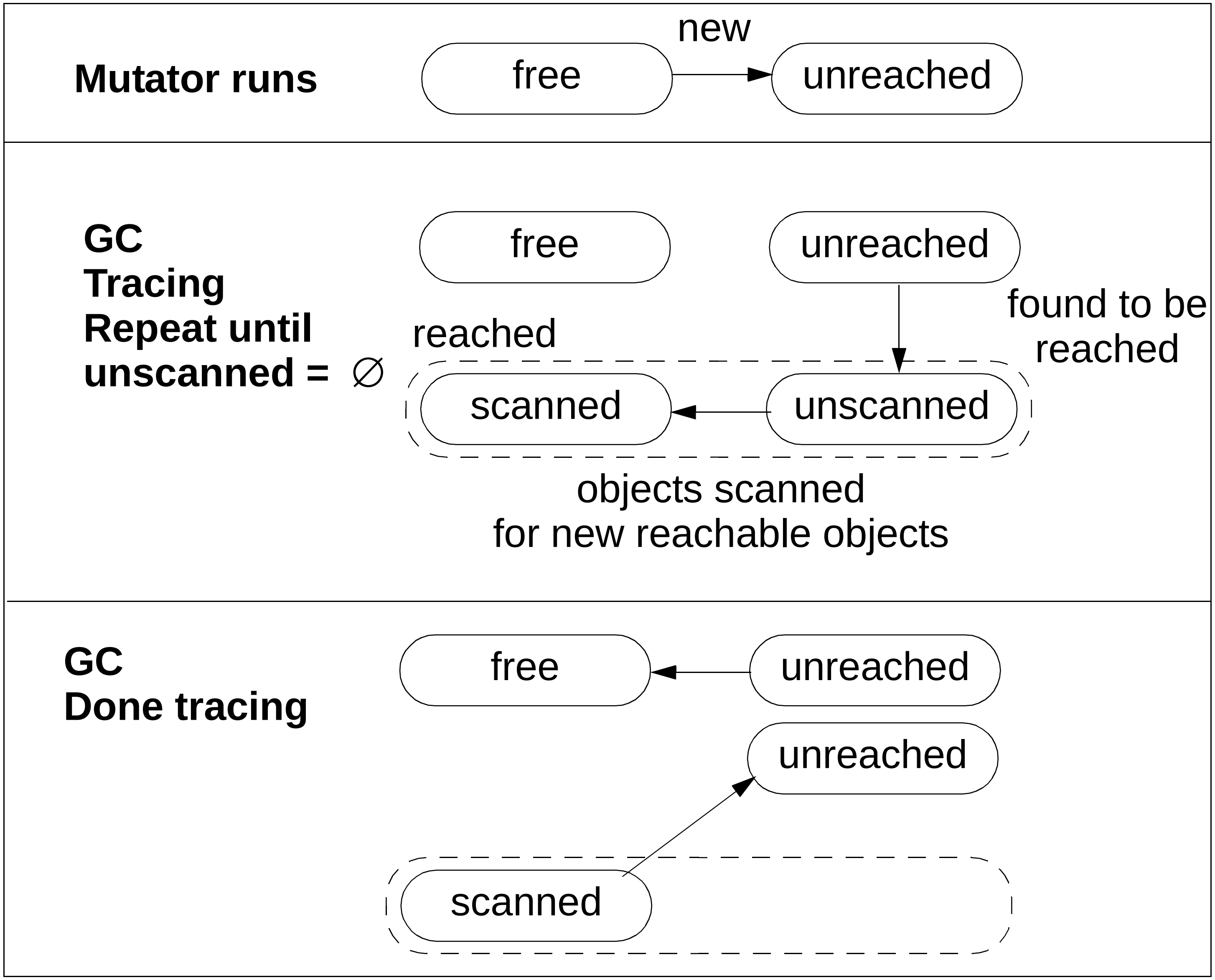
Trace-based GC

- **Reachable objects**
  - Root set: (directly accessible by prog. without deref'ing pointers)
    - objects on the stack, globals, static field members
  - + objects reached transitively from ptrs in the root set.
- **Complication due to compiler optimizations**
  - Registers may hold pointers
  - Optimizations (e.g. strength reduction, common subexpressions) may generate pointers to the middle of an object
  - Solutions
    - ensure that a "base pointer" is available in the root set
    - compiler writes out information
      to decipher registers and compiler-generated variables
      (may restrict the program points where GC is allowed)

Baker's Algorithm

- **Data structures**
  - Free: a list of free space
  - Unscanned: a work list
  - Unreached: a list of allocated objects
  - Scanned: a list of scanned objects
- **Algorithm**
  - Scanned = $\emptyset$
  - Move objects in root set from Unreached to Unscanned
  - While Unscanned $\neq \emptyset$
    - move object o from Unscanned to Scanned
    - scan o, move newly reached objects from Unreached to Unscanned
  - Free = Free $\cup$ Unreached
  - Unreached = Scanned

Trace-Based GC: Memory Life-Cycle



When Should We GC?

Frequency of GC

- **How many objects?**
  - Language dependent, for example, Java:
    - all non-primitive objects are allocated on the heap
    - all elements in an array are individually allocated
    - “Escape” analysis is useful
      -- object escapes if it is visible to caller
      -- allocate object on the stack if it does not escape
- **How long do objects live?**
  - Objects die young
- **Cost of reachability analysis: depends on reachable objects**
  - Less frequent: faster overall, requires more memory

Performance Metric

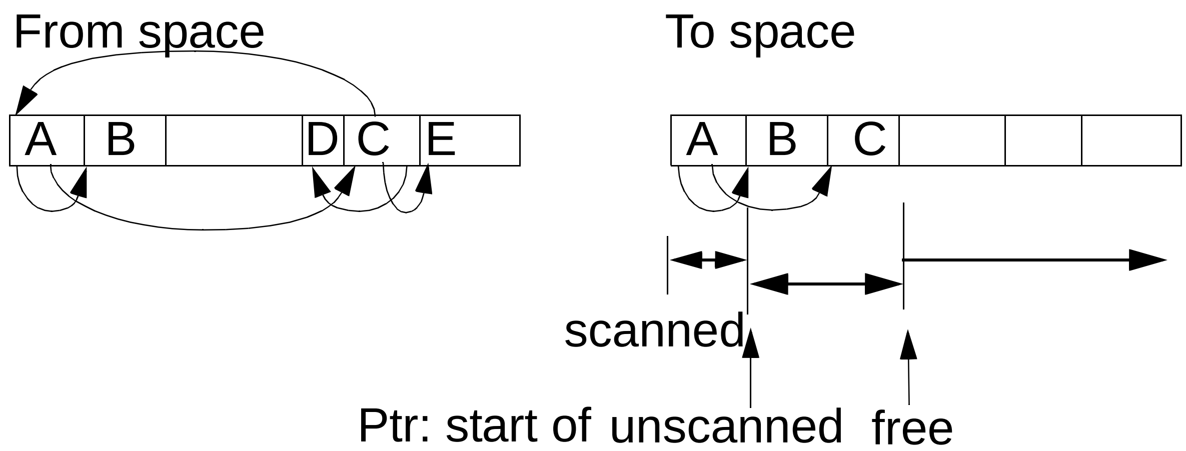
	Reference counting	Trace Based
Space Reclaimed		
Overall execution time		
Space usage		
Pause time		
Program locality		

Copying Collector

- **To improve data locality**
  - place all live objects in contiguous locations
- **Memory separated into 2 (semi-)spaces: From and To**
  - Allocate objects in one
  - When (nearly) full, invoke GC,
    which copies reachable objects to the other space.
  - Swap the roles of semi-spaces and repeat

Copying Collector (cont)

- **Algorithm**



- UnScanned = Free = Start of To space
- Copy root set of objects space after Free, update Free;
- While UnScanned ≠ Free
  - scan o, object at UnScanned
  - copy all newly reached objects to space after Free, update Free
  - update pointers in o
  - update UnScanned

Conclusions

- **Manual GC is error-prone**
  - Memory leaks & dangling pointers
- **Automatic GC: eliminate unreachable objects, not dead objects**
  - May still leak memory, if pointers to unused data exist
- **Reference counting**
  - Delete objects when their reference counts go to 0
  - Expensive
  - Cannot collect circular data structures
- **Trace-based GC**
  - Find all reachable objects, complement to get unreachable
  - 4 states: free, unreached unscanned, scanned
  - Stop-the-world GC: Baker’s algorithm has a long pause time
  - Copying collector improves data locality