Garbage Collection

April 2, 2001
CS 132: Compiler Design

Flavors of Memory Allocation

1. Static
   • Size, locations determined at compile-time
2. Stack
   • Memory allocated for a single procedure call
   • Size might vary depending on arguments
   • Space automatically freed when function exits
3. Heap
   • Arbitrary order of allocation/de-allocation

Heap Management

• Explicit
  - User specifies when objects are allocated and deallocated.
  - Potential for dangling pointers, memory leaks
  - User must keep track: who is responsible for deallocating memory?
• Implicit
  - User only specifies allocations
  - Objects automatically deallocated when determined to be safe.

Vocabulary

• We will call any data item on the heap an object.
• An object is live if it will be used later in the execution of the program.
• An object that is not live is said to be dead or to be garbage.
• A garbage collector detects and deallocates garbage.
  - Deallocating all garbage is undecidable
  - Thus collectors deallocate some subset of the dead objects
More Vocabulary

- The pointers into the heap that a program maintains are called **roots**
  - Roots found in registers, static memory, and the stack
- Data on the heap is said to be **reachable** if it can be accessed by following pointers through the heap, starting with one of the roots; **unreachable** otherwise.
- >99% of garbage collectors are based on the idea that **unreachable objects are garbage**.

Method 1: Reference Counting

- Idea: For each object in the heap, record how many pointers there are to this object
  - In registers
  - In static memory
  - On the stack
  - In other heap objects.
- Update this count when pointers are copied or overwritten or discarded.
- Deallocate any object whose count falls to zero, because it's not reachable.
Pointer Redirected

Program Resumes Allocating

Advantages

1. Conceptually simple
2. Memory can be re-used immediately
3. Can be applied only to pieces of data with ill-defined lifetimes.
4. Memory management is (sort-of) incremental
   • Pause time proportional to amount of data freed at once
5. Can run finalizers (cleanup actions) as soon as an object becomes unreachable.
Disadvantages

1. Very tricky and error-prone if maintaining reference counts by hand.
2. Cost of reference-count updates.
4. Cyclic data structures problematic.

Initial State

Pointer Redirected

Pointer Redirected
Method 2: Tracing GC

- Idea:
  - Allocate until memory exhausted
  - When more space is required, determine the reachable data, and deallocate the rest.

- Two classical variants
  - mark-sweep collectors
  - copying collectors

Mark-Sweep

- Each object has a "mark bit"
- Collection occurs in two steps
  1. Traverse the graph of reachable data, and set the mark bit on all reachable objects
  2. Sequentially examine all the objects in the heap
     - If mark bit was not set, deallocate the object
     - Otherwise clear the mark bit (for the next GC)
Program Continues Allocating

Copying Collector

• Divide memory into halves
  – called fromspace and tospace.
• Allocate objects in fromspace until full
• Then,
  – Copy all reachable data from the fromspace into the tospace.
  – Swap the names "fromspace" and "tospace".
  – Resume allocating in the fromspace (the old tospace)

Before and After Summary

Fromspace Full: Start Collection
Tracing Collectors: Comparison

• Mark-Sweep
  - Collection time proportional to number of objects (live and dead) in the heap.
• Copying
  - Collection time proportional to bytes of live data
  - Copying compacts (defragments) heap
  - Requires twice as much memory (at worst.)
  - Allocation extremely cheap
• Both
  - No problems with cycles in heap
  - Performance degrades as live data increases

How is Tracing Possible?

• It's time to do a garbage collection. Which registers and which stack locations contain pointers into the heap?
  - And of the pointers, which are live?
• What sort of objects are being pointed to?
  - How big?
  - Do they contain pointers? Where?

Finding Roots

• Guess [Conservative Collection]
  - Assume that any register or stack location containing a value that "looks like" a pointer is a pointer
• Tag bits
  - Take a bit (or two) from every word to denote pointer/non-pointer.
• Lookup tables
  - At compile time, generate information about stack frames and register sets at each point in program where GC could occur.

Parsing the Heap

• Bread crumbs
  - Tag heap objects with object's size
  - Tag objects with pointer locations
    • Or just guess, or assume tag bits
• Types
  - From the types of pointers, deduce the object's layout (and the type of every object it points to)
  - Much harder with polymorphism or abstract types
• Segregate objects by shape (BIBOP)
Generational GC

• Problem: long-lived data is copied repeatedly
  – From fromspace to tospace on every collection
• Generational idea:
  – Most data dies young; most young data dies
  – Therefore, create separate heaps for young and old data (the "generations")
  – Young data that survives collection(s) is promoted to an older generation.
  – Try to GC younger generations without touching older generations.

Is Young Independent of Old?

• What happens if old data points to young data?
  – Need to treat such pointers as roots.
  – When young objects moves, need to update the pointers to them in the older generations.
• How do we find these pointers without scanning the older generations (doing as much work as a full GC)?
  – The only way old can point to young is if somebody did an update to the old data
  – General idea: "write barrier". Remember information about all (relevant) updates.

Generational Collection

• Advantages
  – Much shorter pauses
  – Often dramatically less copying.
    • Actually, the generational idea can be used in mark-sweep collectors as well
• Disadvantages
  – Have to keep track of assignments
    • For languages that don’t do many assignments, generational GC can work really well.
  – Can increase memory requirements.

Large Objects

• For large objects, copying even once can be overly expensive.
  – And larger objects likely to be long-lived
• Some generational collectors contain a special heap for large objects
  – Anything large gets automatically allocated there
  – May be managed differently (e.g., mark-sweep)
More Embellishments

- Incremental collectors
  - Do small amounts of collection, more frequently
  - Real-time: hard bound on amount of work done by the
    collector in each step
  - Increases total collection time, decreases maximum pause
- Parallel collectors
  - Several processors cooperating together on collection
- Concurrent collectors
  - Main program and GC run simultaneously
- Hybrid collectors
  - Variants that combine copying/mark-sweep/ref counts

Which Collector is Best?

- Nobody knows; performance depends on
  - Client program behavior
    - Frequency of allocation
    - Sizes of objects
    - Number of reads / writes / pointer copies / etc.
  - Cache and virtual memory performance
  - Order of memory traversal (DFS, BFS, ...)
  - When garbage collections occur
- Lots of work on improving collectors