CSE 401 – Compilers

Memory Management
and Garbage Collection

Hal Perkins

Winter 2017

References

- *Uniprocessor Garbage Collection Techniques* Wilson, IWMM 1992 (longish survey)
- *The Garbage Collection Handbook* Jones, Hosking, Moss, 2012 (book)
- Adapted from slides by Vijay Menon, CSE 501, Sp09

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 (malloc/new/cons)
  - Need to recover storage for reuse when no longer needed (manual or automatic)

Manual Heap Management

- Programmer calls free/delete when done with storage
- Pro
  - Cheap
  - Precise
- Con
  - How do we enumerate the ways?
  - Buggy, huge debugging costs, pain, …

Garbage Collection

- Automatically reclaim heap memory no longer in use by the program
  - Simplify programming
  - Better modularity, concurrency
  - 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"

Heap Characteristics

- Most objects are small (< 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

- Compiler produces:
  - A *stack-map* at *GC safe points*
    - *Stack map:* enumerate global variables, stack variables, live registers (tricky stuff! Why?)
    - *GC safe points:* new(), method entry, method exit, back edges (thread switch points)
  - *Type information blocks*
    - Identifies reference fields in objects (to trace the heap)

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 (8, 16, 24, 32, … depends on alignment); additional list for large blocks
  - Regular (manual) 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-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
  - "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
- 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 as in mark-sweep collector
- 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
  - “Stop the world I want to collect”

Generational Collectors

- Generational hypothesis: young objects die more quickly than older ones (Lieberman & Hewitt '83, Ungar '84)
- 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 Collector

- 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
  - Engineering: use barriers to avoid having to scan all of old space on quick collections
  - Refinement: require objects to survive at least a few collections before copying
- When old space fills, collect both
- Can generalize to multiple generations

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

Recent Research Areas

- Parallel/concurrent garbage collection
  - Found in some production collectors now
    - Tricky stuff – can't debug it into correctness – there be theorems here
- Locality issues
  - Object collocation
  - GC-time analysis
- Distributed GC

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

Boehm/Weiser Collector

- Useful for development/debugging
  - Less burden on compiler/runtime implementor
- Used in some Java and .net implementations
  - Particularly research projects
- Similar ideas for various tools to detect memory leaks, etc.

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?