CS42/413
Introduction to Compilers
Radu Rugina
Lecture 35: Garbage Collection
24 Apr 06

Outline
• Virtual memory
• Explicit memory management
• Garbage collection techniques
  — Reference counting
  — Mark and sweep
  — Copying GC
  — Concurrent/incremental GC
  — Generational GC
• Book: “Garbage Collection”, by R. Jones and R. Lins

Virtual Memory

Explicit Memory Management
• Unix (libc) interface:
  void* malloc(long n): allocate n bytes of storage on
  the heap and return its address
  void free(void *addr): release storage allocated by
    malloc at address addr
• User-level library manages heap, issues brk calls when
  necessary

Freelists
• Blocks of unused memory stored in freelist(s)
  malloc: find usable block on freelist
  free: put block onto head of freelist
• Simple, but fragmentation ruins the heap
• External fragmentation — small free blocks become scattered
  in the heap
• Cannot allocate a large block even if the sum of all free
  blocks is larger than the requested size

Buddy System
• Idea 1: freelists for different allocation sizes
  — malloc, free are O(1)
• Idea 2: freelist sizes are powers of two: 2, 4, 8, 16, ...
  — Blocks subdivided recursively: each has buddy
  — Round requested block size to the nearest power of 2
  — Allocate a free block if available
  — Otherwise, (recursively) split a larger block and put all the
    other blocks in the free list
  — Reverse operation: coalesce (with buddy, if free, not split)
• Internal fragmentation: allocate larger blocks because of rounding
• Trade external fragmentation for internal fragmentation
Explicit Garbage Collection

- Java, C, C++ have new operator / malloc call that allocates new memory
- How do we get memory back when the object is not needed any longer?
- Explicit garbage collection (C, C++)
  - delete operator / free calls destroys object, allows reuse of its memory: programmer decides how to collect garbage
  - makes modular programming difficult, because one has to know what code “owns” every object so that objects are deleted exactly once

Automatic Garbage Collection

- The other alternative: automatically collect garbage
- Usually most complex part of the run-time environment
- Conservative: delete only objects that definitely won’t be used again
- Reachability: objects definitely won’t be used again if there is no way to reach them from root references that are always accessible (globals, stack, registers)

Object Graph

- Stack, registers are treated as the roots of the object graph. Anything not reachable from roots is garbage
- How can non-reachable objects be reclaimed efficiently? Compiler can help

Algorithm 1: Reference Counting

- Idea: associate a reference count with each allocated block (reference count = the number of references (pointers) pointing to the block)
- Keep track of reference counts
  - For an assignment \( x = y \), increment the reference count of the new block \( x \) is pointing to
  - Also decrement the reference count of the block \( x \) was previously pointing to
- When number of incoming pointers is zero, object is unreachable: garbage

Reference Counts

- … how about cycles?

Reference Counts

- Reference counting doesn’t detect cycles!
Performance Problems

- Consider assignment \( x.f = y \)
- Without ref-counts: \( *\) (tx= off) = ty
- With ref-counts:
  \[ t1 = *\) (tx + f_off); c = \( *\) (t1 + refcnt); c = c - 1;
  \( *\) (t1 + refcnt) = c; if (c > 0) goto L2; free(t1);
  L2: c = \( *\) (ty + refcnt); c = c + 1; \( *\) (ty + refcnt)
  = c; \( *\) (tx + f_off) = ty;
- Data-flow analysis can be used to avoid unnecessary
  increments & decrements
- Large run-time overhead
- Result: reference counting not used much by real language
  implementations

Algorithm 2: Mark and Sweep

- Classic algorithm with two phases
- Phase 1: Mark all reachable objects
  - start from roots and traverse graph forward marking every
    object reached
- Phase 2: Sweep up the garbage
  - Walk over all allocated objects and check for marks
  - Unmarked objects are reclaimed
  - Marked objects have their marks cleared
  - Optional: compact all live objects in heap

---

Traversing the Object Graph

Implementing Mark Phase

- Mark and sweep generally implemented as depth-first
  traversal of object graph
- Has natural recursive implementation
- What happens when we try to mark a long linked list
  recursively?

Cost of Mark and Sweep

- Mark and sweep accesses all memory in use by program
  - Mark phase reads only live (reachable) data
  - Sweep phase reads all of the data (live + garbage)
- Hence, run time proportional to total amount of data
- Can pause program for long periods!
Conservative Mark and Sweep

- Allocated storage contains both pointers and non-pointers; integers may look like pointers
- Issues: precise versus conservative collection
- Treating a pointer as a non-pointer: objects may be garbage-collected even though they are still reachable and in use (unsafe)
- Treating a non-pointer as a pointer: objects are not garbage collected even though they are not pointed to (safe, but less precise)
- Conservative collection: assumes everything is a pointer; requires no language support (works for C!)

Algorithm 3: Copying Collection

- Like mark & sweep: collects all garbage
- Basic idea: use two memory heaps (two "spaces")
  - one heap in use by program
  - other sits idle until GC uses it
- GC mechanism:
  - Copy all live objects from active heap (from-space) to the other (to-space)
  - Dead objects discarded during the copy process
  - The two spaces then switch roles
- Issue: must rewrite referencing relations between objects

Copying Collection (Cheney)

- Copy = move all root objects from from-space to to-space
- From space traversed breadth-first from roots, objects encountered are copied to top of to-space.

Benefits of Copying Collection

- Once scan=next, all uncopied objects are garbage. Root pointers (registers, stack) are swung to point into to-space, making it active
- Good:
  - Simple, no stack space needed
  - Run time proportional to size of live objects
  - Automatically eliminates fragmentation by compacting memory
  - malloc() implemented as (top = top + n)
- Bad:
  - Precise pointer information required
  - Twice as much memory used

Incremental and Concurrent GC

- GC pauses avoided by doing GC incrementally; collector & program run at same time
- Program only holds pointers to to-space
- On field fetch, if pointer from from-space, copy object and fix pointer
- On swap, copy roots and fix stack/registers

Generational GC

- Observation: if an object has been reachable for a long time, it is likely to remain so
- In long-running system, mark & sweep, copying collection waste time, cache scanning/copying older objects
- Approach: assign heap objects to different generations G₀, G₁, G₂...
  - Generation G₀ contains newest objects, most likely to quickly become garbage (<10% live)
Generations

- Consider a two-generation system. \( G_0 \) = new objects, \( G_1 \) = tenured objects
- New generation is scanned for garbage much more often than tenured objects
- New objects eventually given tenure if they last long enough
- Roots of garbage collection for collecting \( G_2 \) include all objects in \( G_1 \) (as well as stack, registers)

Remembered Set

- How to avoid scanning all tenured objects?
- In practice, few tenured objects will point to new objects; unusual for an object to point to a newer object
- Can only happen if older object is modified long after creation to point to new object
- Compiler inserts extra code on object field pointer writes to catch modifications to older objects—older objects are remembered set for scanning during GC, tiny fraction of \( G_1 \)

Summary

- Garbage collection is an aspect of the program environment with implications for compilation
- Important language feature for writing modular code
- IC: Boehm/Demers/Weiser collector
  – conservative: no compiler support needed
  – generational: avoids touching lots of memory
  – incremental: avoids long pauses
  – true concurrent (multi-processor) extension exist