CSE P 501 – Compilers

Memory Management & Garbage Collection

Hal Perkins

Winter 2008

© 2002-08 Hal Perkins & UW CSE

Agenda

- Dynamic memory – heap storage
- Manual storage management: malloc/free
- Reference counting
- Automatic garbage collection
  - Classic mark/sweep collectors
  - Copying and compacting collectors
  - Generational garbage collection
  - Incremental collection
  - Garbage collection in hostile environments (C++)

© 2002-08 Hal Perkins & UW CSE

References

- Appel, ch. 13
- Dragon book 2nd ed, sec. 7.4-7.8
- *Garbage Collection* by Jones & Lins, Wiley, 1996

Oh, Garbage! Garbage!

They're filling the heap with garbage!

(with apologies to Bill Steele and Pete Seeger)

© 2002-08 Hal Perkins & UW CSE

Storage Classes (Review)

Most languages provide the following:

- Static
  - Single copy; lifetime = program execution
- Automatic
  - Allocated on procedure entry, released on exit; lifetimes nest with procedure calls; can usually be implemented with stacks
- Dynamic
  - Allocated and freed at arbitrary times under program control

Manual Storage Allocation

- malloc(size), new <type>
  - Find a block of storage of (at least) the requested size and return a pointer to it
- free(p), delete p
  - Release the block of storage designated by p – which must have been acquired with malloc/new
  - Presumably this block of storage will be reused later by malloc/new if needed

 © 2002-08 Hal Perkins & UW CSE

Some Implications

- Allocated blocks must hold some (meta-) information describing their size or type
  - (Otherwise free/delete doesn't know what its got)
- Memory manager maintains a list of free storage
  - Requests satisfied from this list
  - free/delete returns storage here
  - Overall dynamic storage pool size increased by memory requests from OS as needed

© 2002-08 Hal Perkins & UW CSE

Performance Issues

- malloc/new search strategies:
  - First-fit
  - Best-fit
- free/delete:
  - Should combine newly released blocks with adjacent free blocks to avoid having lots of small, mostly useless chunks (fragmentation)
  - Can use tags at both ends of free blocks to coalesce adjacent blocks in constant time

© 2002-08 Hal Perkins & UW CSE

Multiple Free Lists

- Even if we coalesce free blocks, fragmentation & free-list search is a performance problem
- One widely used solution – keep multiple free lists with different size blocks
  - Generally lots of fixed-size bins (~100 sizes) and one very large bin for other requests
  - Satisfy requests from appropriate list, or split a block from the next larger list if needed (smallest-first, best-fit)
  - Best known example: Doug Lea's malloc in glibc (http://g.oswego.edu/dl/html/malloc.html)

© 2002-08 Hal Perkins & UW CSE

But…

- Manual memory management is horribly error-prone
  - Memory leaks
  - Dangling pointers
- Huge costs for debugging
- So, can we automate it?
  - Yes – and we have been for 50 years!

© 2002-08 Hal Perkins & UW CSE

Reference Counting

- Simple idea: add a field to each block of storage keeping track of number of live references to that block
- When executing `p=q;`
  - Decrease reference count of *p
    - ☞ If reference count is now 0, free the block!
  - Increase reference count of *q

© 2002-08 Hal Perkins & UW CSE

Reference Counting Evaluated

- Two serious problems as a general allocator
  - Very high overhead on pointer assignment (relative to cost of assignment)
  - Circular structures will never have reference counts of 0, even if no external references exist
    - Solution is to break manually, but that's bug-prone
- So not used as a general memory manager
  - But is used in applications where these are not drawbacks – e.g., reclaiming files in file systems

© 2002-08 Hal Perkins & UW CSE

Automatic Garbage Collection

- Idea: any storage that is not reachable by a chain of pointers from program variables is garbage and should be reclaimed
- General strategy
  - Scan storage to find all live data
  - Place any heap data not reached during the scan on the free list (using the usual coalescing strategies, etc.)

© 2002-08 Hal Perkins & UW CSE

Liveness and Reachability

- Conservative approximation to liveness: *reachability*
- Definition:
  - All variables in the *root set* are reachable
    - Root set = all pointers contained in: registers + active stack frames + static variables
  - All data that can be reached transitively from some reachable variable is also reachable

© 2002-08 Hal Perkins & UW CSE

Mark-Sweep Garbage Collector

- Steps. Stop program execution, then
  1. (Mark) Starting at the root set, find all reachable data
  2. (Sweep) Scan the heap sequentially and place any data that is not marked as reachable on the free list
     - During this phase, reset the mark bits on all marked data to prepare for the next collection

© 2002-08 Hal Perkins & UW CSE

Mark-Sweep Implementation

- Mark phase

```
for each root r, dfs(r),
where: dfs(r) =
if r points into the heap
  if record r is not marked
      mark r
      for each field f in r,
         dfs(r.f)
```

- Sweep phase

```
p := beginning of heap
while p < end of heap
  if record p is marked
     unmark p
  else
     add record p to
        freelist
   p += size of record p
```


© 2002-08 Hal Perkins & UW CSE

What the Compiler Must Tell the Garbage Collector (1)

- Implicit is that, given a heap pointer, the garbage collector can know the type (& therefore size) of the referenced object, and the offsets and types of its fields
- Often almost free – in object-oriented systems, every object has a reference to a class vtable anyway, so include type information in that data structure

© 2002-08 Hal Perkins & UW CSE

What the Compiler Must Tell the Garbage Collector (2)

- Harder: the GC must be able to identify every register, local variable, and temporary that contains a heap reference – regardless of where/when the program is stopped for collection(!)
- ∴ Need a pointer map for each point of the program where a GC might happen
  - For sure, every point where allocation is requested
  - But also need to worry about finding pointers on the stack if a GC happens in the middle of a function call (including pointers in registers saved on the stack)

Storage for Mark Phase

- As described, mark phase uses a DFS of the heap to find reachable storage
  - But depth of recursion is potentially bounded by size of the heap(!)
    - And we’re out of storage – which is why we’re doing a GC in the first place (!!)
      - oops!!!

© 2002-08 Hal Perkins & UW CSE

Pointer Reversal

- Idea: Once we follow a pointer, we don't need it again during the mark phase
  - So reverse each pointer as we encounter it
    - Keeps track of return path in the heap graph
  - Then as DFS function returns, flip the pointers back to their original state
- Tricky to get right, but allows a mark phase in (basically) constant space

Problems with Mark-Sweep

- Storage fragmentation
  - Over time, active storage in the heap becomes fragmented and spread out
- Pauses
  - "Stop the world I want to collect" is not great for animation, user interaction, real-time
- Overhead
  - Lots of redundant work rescanning long-lived objects

© 2002-08 Hal Perkins & UW CSE

Copying Collectors

- Over time active storage becomes fragmented
  - Not great for virtual memory systems, cache
- Idea: During a GC, copy active objects to contiguous storage
  - Need to fix up pointers as we go
- Two versions: compress in place, or semispaces – we'll look at the later

Semi-Space Copying Collector

- Idea: Divide heap into two halves
  - *from-space* contains the data to be collected/compacted
  - *to-space* is initially empty
- Collection goes through from-space moving all reachable objects to to-space
  - When an object is moved, leave a *forwarding pointer* in its location in from-space
  - When we encounter a pointer p, if it references a forwarding pointer, just update p, otherwise recursively copy the referenced from-space object
- When finished, flip roles of from-space and to-space
  - All the data is now in the newly copied/compressed from-space, and to-space (the old from-space) is empty for the next collection

© 2002-08 Hal Perkins & UW CSE

Copying Collector Variables

- Root set (as before)
- GC pointer referencing to-space:
  - scan – address of next object moved to to-space but not yet scanned for pointers to other objects
  - next – address of next available location in to-space for newly moved objects
- During the collection, scan chases next until it catches up when the last reachable object has been copied and processed

© 2002-08 Hal Perkins & UW CSE

Cheney's Algorithm (informal)

```
scan := next :=
          start of to-space
for each root r
    r := forward(r)
while scan < next
    for each field f in
             object at scan
        scan.f :=
             forward(scan.f)
    scan += size of
              record at scan
```

```
forward(p) ≡
  if p points to to-space
    then return p
    else if *p is a forwarding
             pointer to to-space
      then return *p
      else // copy record p.
        for each field f in
                    record p
             next.f := p.f
        // store in from-space
        //   forward ptr to copy.
        p := next
        next += size of record p
        return p
```


© 2002-08 Hal Perkins & UW CSE

Would an Example Help?

© 2002-08 Hal Perkins & UW CSE

Locality of Reference

- Cheny's algorithm makes a breadth-first copy, which tends to have poor locality
  - (Think about what happens to a linked-list or tree when it is copied)
- Depth-first copying would be great, but is a mess (pointer reversal)
- Reasonable compromise: use breadth-first, but if possible place a child of each copied object near the object (semi-depth-first)

© 2002-08 Hal Perkins & UW CSE

Now What?

- We’ve done a fair amount about fragmentation, but still haven’t addressed overhead or pauses
- Solutions
  - Overhead: Generational Collection
  - Pauses: Incremental Collection

© 2002-08 Hal Perkins & UW CSE

Object Lifetimes

- Functional and object-oriented programs, in particular, allocate lots of short-lived and often small objects
- So if we can concentrate our GC efforts on recently allocated objects, we're likely to reclaim a larger percentage of what we scan

© 2002-08 Hal Perkins & UW CSE

Generational Garbage Collection

- Idea: divide the heap into "generations" G_0, G_1, … (typically no more than 3 or 4 total).
- All objects in G_1 are older than any objects in G_0; same is true for G_{i+1} and G_i
- New objects are created in G_0, often called the *nursery*.
- Collect G_0 frequently; other generations less so
- Objects in G_0 that survive several collections should be promoted to G_1 (and so forth)

© 2002-08 Hal Perkins & UW CSE

Generational GC

- Pretty much the same as mark-sweep or copying collector
- Difference: when collecting G_0, root set also includes all objects in G_1, G_2, … .
- In general, when we collect G_i:
  - Root set includes G_{i+1}, G_{i+2}, …
  - Collect G_i and all younger generations back to G_0 at the same time

But That’s a Huge Root Set!

- Yes and no
  - Yes, we need to worry about all references from older objects to new ones
  - No, there aren’t many of these
- So need an efficient strategy to detect references to new objects stored in old objects
  - Preferably without having to scan the old generations (which would loose most of the efficiency)

© 2002-08 Hal Perkins & UW CSE

Remembered Sets (1)

- To avoid searching old generations, compiler must arrange for program to remember pointers from old objects to new ones
- Basic idea is for compiler to generate code to flag objects or parts of storage that might contain old objects with pointers to new space

© 2002-08 Hal Perkins & UW CSE

Remembered Sets (2)

- Common strategies:
  - Compiler generates code to set a per-object flag bit whenever it stores a pointer that might point to a newer object; flagged objects are in the root set
  - Compiled code sets a flag bit whenever an object in some region of memory is changed (i.e., use some higher-order bits of the object address); all objects in that region are part of the root set
  - Use paging hardware to mark pages with old objects "read-only"; if a write is intercepted, mark that page as part of the root set before letting the write proceed

© 2002-08 Hal Perkins & UW CSE

Incremental Collection

- Still haven't solved the "stop the world I want to collect" problem
- Solution: an exercise in concurrent programming. Actors:
  - Mutator – the user program that is altering memory and creating garbage
  - Collector – the GC algorithms
- These run in separate threads
- Basic idea is to be sure the mutator can proceed even while the GC is doing work
  - See the literature & don't try to debug this stuff without proving your theorems first

Garbage Collection for Unsafe Languages

- What about C, C++, and others?
- Basic problem: program can compute addresses
  - A program can fabricate addresses from arbitrary collections of bits: `(int*)1234 = 17;`
  - ∴ we have no guarantees over where the pointers are stored or what kinds of things they point to – so GC can't do a precise job

Conservative GC (1)

- But most C/C++ programs are not that nasty, so we can do (a lot) better than nothing at all
- Idea: Conservative GC assumes anything that looks like a pointer to an address in the heap might be one
- Memory manager keeps track of types of objects it has allocated

© 2002-08 Hal Perkins & UW CSE

Conservative GC (2)

- Root set is scanned to find any bit pattern that looks like a pointer to the heap
- Data map is used to find starting address of corresponding chunk of heap storage
- This is scanned under the assumption we know its type
- This should find all reachable storage (under reasonable sanity assumptions) but also gets more
  - Yet another conservative analysis
- Best known example: Boehm/Wieser collector

© 2002-08 Hal Perkins & UW CSE

A Bit of Perspective

- Automatic garbage collection has been around since LISP I in 1958
- Ubiquitous in the functional programming community ever since
- Some appearance in mainstream languages over the years (e.g., Ada in the 80s)
- Widely used in object-oriented languages (e.g., Smalltalk, self, many others)
- Finally hit the mainstream with Java, mid-90s
- Now conventional wisdom in many settings

© 2002-08 Hal Perkins & UW CSE