# CS 412
# Introduction to Compilers

Andrew Myers
Cornell University

Lecture 33: Memory management
23 Apr 01

## Administration

- Programming Assignment 5 due Friday



## Schedule

Topics for remainder of course:

- Run-time support
  - Garbage collection
  - Linking and loading
  - Meta-objects
  - JITs and interpreters
- Advanced language support
  - First-class functions
  - Exceptions
  - Parametric polymorphism

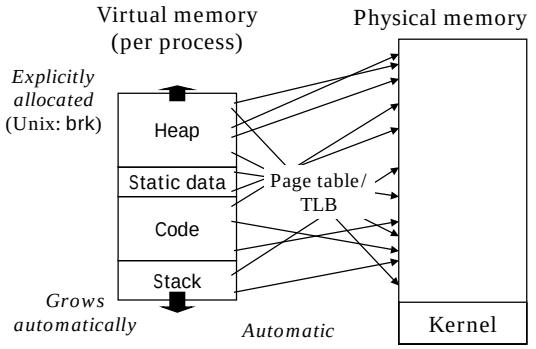


## Outline

- Overview of memory management, garbage collection techniques and impact on compiled code:
  - Storage heaps
  - Mark and sweep garbage collection
  - Reference counting GC
  - Copying GC
    - concurrent/incremental garbage collection
  - Generational GC



## Memory






## Explicit Memory Management

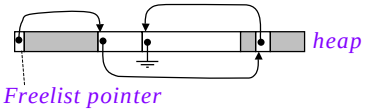
- Unix (libc) interface:

  `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




*Freelist pointer*

- Simple, but...
- Fragmentation ruins the heap
- malloc may be slow!



## 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
  - adjacent free blocks promoted to next freelist
- Trades *external fragmentation* for *internal fragmentation*
- Wasted space: ~30%



## Problem

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



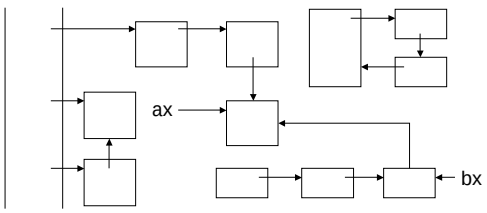
## Automatic garbage collection

- Usually most complex part of the run-time environment
- Want to delete objects automatically if they won't be used again: undecidable
- 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 can be reclaimed efficiently? Compiler can help

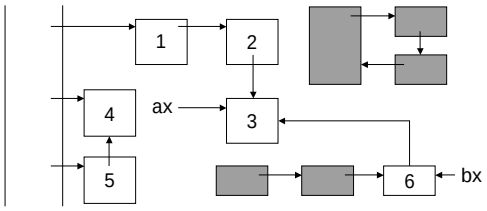





## Mark and sweep collection

- 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 (need double indirection via object table)

## Traversing the object graph





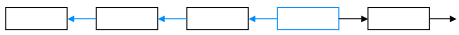

## Implementing mark phase

- 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?



## Pointer reversal

- *Idea:* during DFS, each pointer only followed once. Can *reverse pointers* after following them -- no stack needed! (Deutsch-Waite-Schorr alg.)



- Implication: objects are broken while being traversed; all computation over objects must be halted during mark phase (oops)



## Cost of mark and sweep

- Mark and sweep algorithm reads all memory in use by program
- Run time proportional to total amount of data (live *and* garbage)
- Can pause program for long periods!



## Conservative Mark & Sweep

- Allocated storage contains both pointers and non-pointers; integers may look like pointers
- Treating a pointer as a non-pointer: objects may be garbage-collected even though they are still reachable and in use
- Treating a non-pointer as a pointer: objects are not garbage collected even though they are not pointed to (safe)
- *Conservative collection*: assumes things are pointers unless they can't be; requires no language support (works for C!)

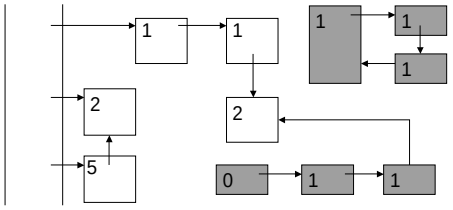


## Reference counting

- Old algorithm for automatic garbage collection: associate with every object a *reference count* that is the number of incoming pointers
- When number of incoming pointers is zero, object is unreachable: garbage

## Reference counts

- Reference counting doesn't detect cycles!






## Performance problems

- Consider assignment x.f = y
- Without ref-counts: mov [tx + f_off], ty
- With ref-counts:
  t1 = M[tx + f_off]; c = M[t1 + refcnt]; c = c - 1; M[t1 + refcnt] = c; if (c == 0) goto L1 else goto L2; L1: call release_Y_object(t1); L2: M[tx + f_off] = ty; c = M[ty + refcnt]; c = c + 1; M[ty + refcnt] = c;
- Data-flow analysis can be used to avoid unnecessary increments & decrements
- Can pause program, overrun stack!
- Result: reference counting not used much by real language implementations



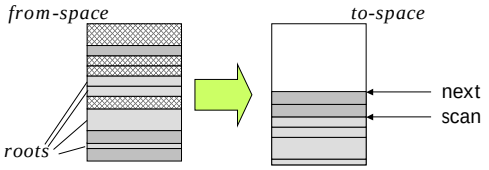
## Copying collection

- Like mark & sweep: collects all garbage
- Basic idea: two memory heaps
  - one heap in use by program
  - other sits idle until GC requires it
- GC:
  - copy all live objects from active heap (from-space) to the other (to-space)
  - dead objects discarded en masse
  - heaps then switch roles



## Copying collection (Cheney's)

- Copying starts by moving 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.





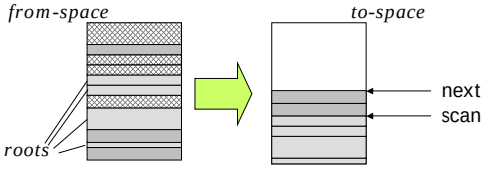

## Benefits of copying collection

- 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 # live objects
  - Automatically eliminates fragmentation by compacting memory
  - malloc(n) implemented as (top = top + n)
- **Bad:**
  - Precise pointer information *required*
  - Twice as much memory used



## Baker's 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 to from-space, copy object and fix pointer (extra fetch code: 20%)
- 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 objects to different *generations* $G_0$, $G_1$, $G_2$,...
- Generation $G_0$ contains newest objects, most likely to 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 $G_0$ 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
- Iota, Iota$^+$: Boehm/Demers/Weiser collector
  http://reality.sgi.com/boehm/gcdescr.html
  - conservative: no compiler support needed
  - generational: avoids touching lots of memory
  - incremental: avoids long pauses
  - true concurrent (multi-processor) extension exists
- GC is here to stay! (thanks to Java)