

Garbage Collection Techniques

CS386L - Programming Languages

Dr. Greg Lavender
Department of Computer Sciences
The University of Texas at Austin



Some Terminology

- garbage collection (gc) - automatic management of dynamically allocated storage
- allocator - operation to allocate storage (e.g., new)
- mutator - the program from the perspective of the gc
- collector - the garbage collector
- comprehensive collection - no garbage is left uncollected

Cells, Reachability, Liveness

- A data item in the heap is called a cell
 - cells are "pointed to" by pointers held in registers, the stack, global/static memory, or in other heap cells
- The "roots" hold references to cells
 - registers, stack locations, global/static variables
 - a cell is reachable if it can be reached from a root
- A cell is "live" if its address is held in a root
 - or held by another live cell in the heap
- liveness may be determined directly or indirectly
 - direct collectors use a reference count to keep track of live cells
 - indirect collectors use "tracing" to compute liveness each time a request for memory fails, causing non-live cells (i.e., garbage) to be collected

4/26/04 22:17

CS386L - Programming Languages - Lecture 14

3

Why Garbage Collection?

- today's programs consume storage freely
 - large real memory systems
 - 1GB laptops now common, 1-4GB desktops
 - 8-512GB in servers, 2GB per processor is common in SMP systems
 - huge virtual address spaces
 - 2^{64} addresses in SPARC, PA-RISC, Itanium, Opteron
 - that's 18,446,744,073,709,551,616 bytes
 - $2^{60} = 1$ Exabyte, see <http://physics.nist.gov/cuu/Units/binary.html>
- today's programs often mismanage storage
 - space leaks, dangling reference, double free, heap fragmentation
 - null pointer de-reference (segmentation violation or GPF)
 - misaligned address reference (bus error)
 - poor use of reference locality, resulting in high cache miss rates and/or excessive demand paging

4/26/04 22:17

CS386L - Programming Languages - Lecture 14

4



What is Garbage Collection?

- gc is not a language feature per se, but a pragmatic concern
 - automatic reclamation of heap-allocated storage
 - garbage implies a cell is no longer "live" in the program
- the heap can be automatically and efficiently managed
 - cooperative languages
 - Lisp, Scheme, Prolog, Smalltalk, Eiffel, Modula-3, Java, ML, Haskell, etc.
 - uncooperative languages
 - Pascal, C, C++
 - but gc libraries have been built for C/C++
- "new" languages sparked a revival in gc techniques
 - object-oriented: Self, Modula-3 and Java
 - functional: ML and Haskell

4/26/04 22:17

CS386L - Programming Languages - Lecture 14

5



The Perfect Garbage Collector

- Incremental
 - no visible impact on program execution
- Works with any program and its data structure
 - e.g., handles cyclic data structures
- Always collects garbage cells quickly
 - no latent garbage and only collects garbage
- Has excellent spatial locality of reference
 - no excessive paging, no negative cache effects
- Manages the heap efficiently
 - always satisfies an allocation request and does not fragment
- Can meet real-time constraints

4/26/04 22:17

CS386L - Programming Languages - Lecture 14

6



Summary of GC Techniques

- Classical
 - Reference Counting GC
 - Tracing
 - Mark-Sweep Collection
 - Copying Collection
- Modern
 - Generational GC

4/26/04 22:17

CS386L - Programming Languages - Lecture 14

7



Reference Counting

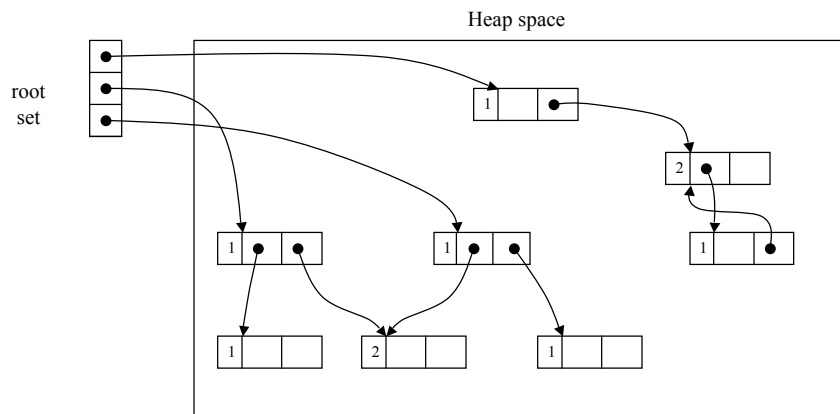
- simply count the number of references to a cell
 - requires space & time overhead to store count and increment/decrement each time a reference is added/removed
 - incremental cost over the running time of the program
 - no stop-and-gag effect since counts maintained in real-time
 - Unix file system uses a ref count for files
 - C++ "smart pointer" (e.g., `auto_ptr`) use ref counts

4/26/04 22:17

CS386L - Programming Languages - Lecture 14

8

Reference Counting Example



4/26/04 22:17

CS386L - Programming Languages - Lecture 14

9

Reference counting costs

- Strengths
 - incremental overhead as management of cells is interleaved with execution of the mutator
 - good for interactive or real-time computation
 - relatively easy to implement
 - can coexist with manual memory management
 - spatial locality of reference is good
 - access pattern to vm pages no worse than the program, so would not expect excessive paging to occur
 - freed cells can be reused as soon as their ref count is zero
 - immediately put back onto the free list
 - fewer page faults, better cache hit ratios
 - finalizer actions, if any, are deterministic
 - e.g., close a file immediately when $rc == 0$

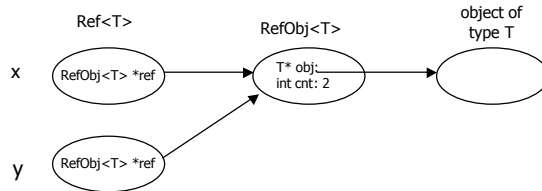
4/26/04 22:17

CS386L - Programming Languages - Lecture 14

10

Example: C++ "smart pointer"

- similar to ANSI C++ `std::auto_ptr<T>`



`sizeof(RefObj<T>) = 8 bytes of overhead per reference counted object`

`sizeof(Ref<T>) = 4 bytes`, which fits in a register, so easily passed by value as an argument or result of a procedure/function. I.e., no more costly than a pointer in terms of space efficiency, but much "safer."

4/26/04 22:17

CS386L - Programming Languages - Lecture 14

13

Example: C++ smart pointer

```

template<class T> class Ref {
    RefObj<T>* ref;
    Ref<T>* operator&() {} // hide reference operator
public:
    Ref() : ref(0) {}
    Ref(T* p) : ref(new RefObj<T>(p)) { ref->inc(); }
    Ref(const Ref<T>& r) : ref(r.ref) { ref->inc(); }
    ~Ref() { if (ref->dec() == 0) delete ref; }

    Ref<T>& operator=(const Ref<T>& that) {
        if (this != &that) {
            if (ref->dec() == 0) delete ref;
            ref = that.ref;
            ref->inc();
        }
        return *this;
    }

    T* operator->() { return *ref; }
    T& operator*() { return *ref; }
};

template<class T> class RefObj {
    T* obj;
    int cnt;
public:
    RefObj(T* t) : obj(t), cnt(0) {}
    ~RefObj() { delete obj; }

    int inc() { return ++cnt; }
    int dec() { return --cnt; }

    operator T*() { return obj; }
    operator T&() { return *obj; }
    T& operator *() { return *obj; }
};
  
```

4/26/04 22:17

CS386L - Programming Languages - Lecture 14

14

Example: C++ smart pointer

```
Ref<string> proc() {
    Ref<string> s = new string("Hello, world"); // ref count set to 1
    ...
    int x = s->length(); // s.operator->() returns string object ptr
    ...
    return s;
} // ref count goes to 2 on copy out, then 1 when s is auto-destructed

int main()
{
    ...
    Ref<string> a = proc(); // ref count is 1 again
    ...
} // ref count goes to zero and string is destructed, along with Ref and RefObj objects
```

4/26/04 22:17

CS386L - Programming Languages - Lecture 14

15

Mark-Sweep Collection

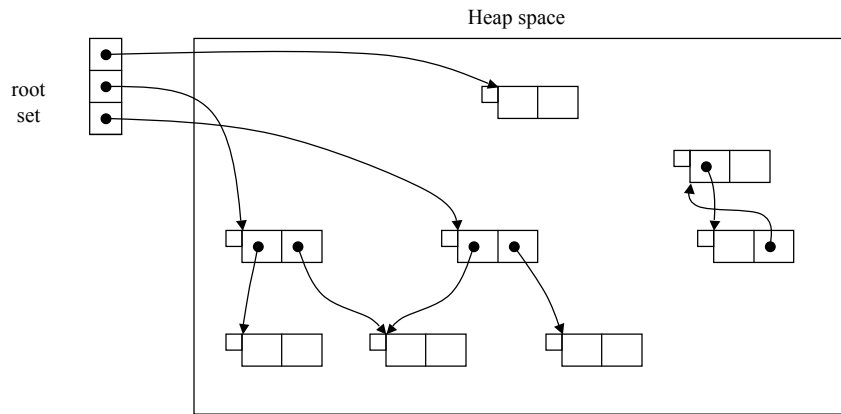
- cells are reclaimed in mass
 - remain unreachable and undetected until heap is used up
 - when a new cell is requested, processing is suspended while the collector goes to work identifying and collecting garbage cells
 - marking phase marks all live cells, starting from the roots
 - each cell has a "mark-bit" that is set if reachable from root set
 - sweep phase returns garbage cells to the free list
 - linear sweep of all cells, returning unmarked cells to the free list and resetting the mark-bit for the next collection cycle

4/26/04 22:17

CS386L - Programming Languages - Lecture 14

16

Mark-Sweep example

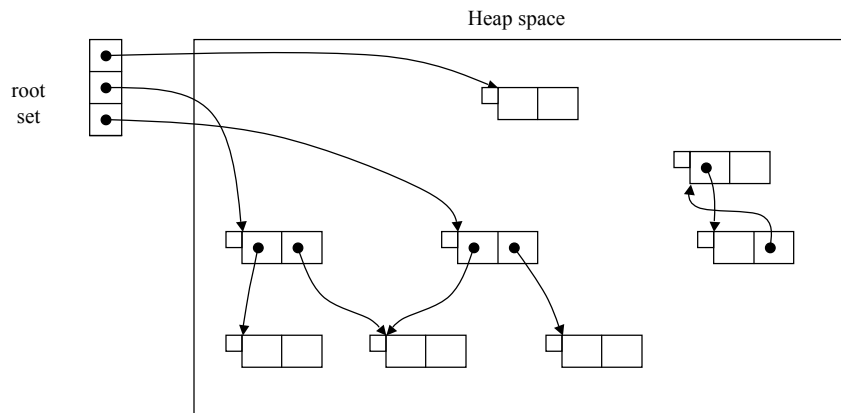


4/26/04 22:17

CS386L - Programming Languages - Lecture 14

17

Mark-Sweep example (cont.)

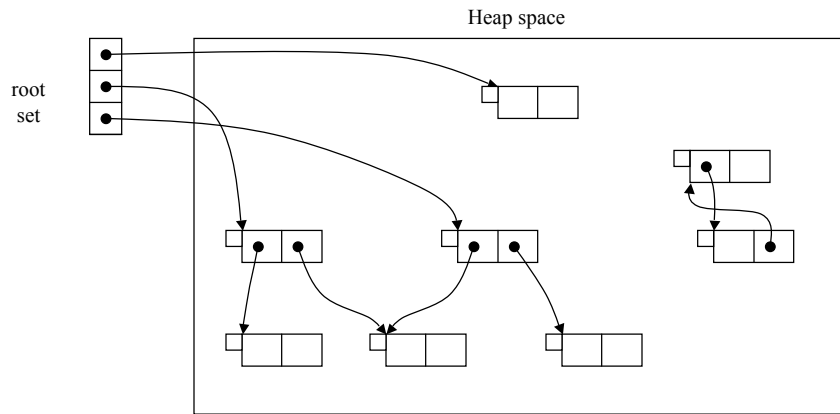


4/26/04 22:17

CS386L - Programming Languages - Lecture 14

18

Mark-Sweep example (cont.)

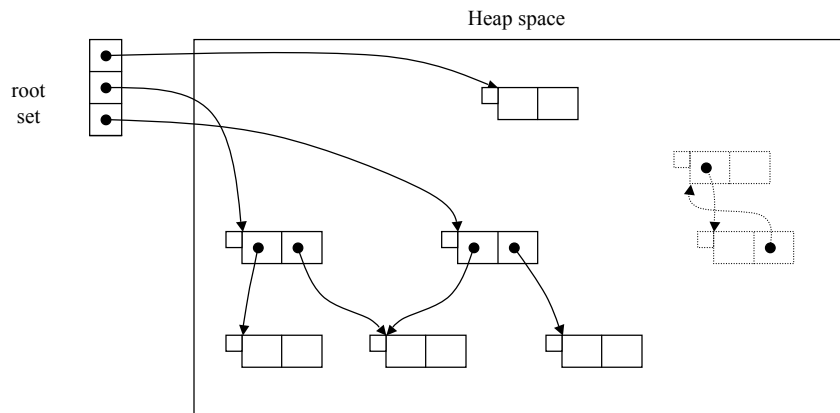


4/26/04 22:17

CS386L - Programming Languages - Lecture 14

19

Mark-Sweep example (concl.)



4/26/04 22:17

CS386L - Programming Languages - Lecture 14

20

Mark-Sweep costs

- strengths
 - improves on RC by handling cycles and no space overhead for counting
 - 1 bit used for marking cells may limit max values (e.g., integers)
- weaknesses
 - computation stops while mark-sweep phases underway
 - earlier versions of emacs had this behavior, sometimes delaying for a few seconds
 - mark-sweep may touch all vm pages, resulting in excessive paging if the working-set size is small and the heap is not all in physical memory
 - heap may fragment causing poor spatial locality of reference
 - cache misses and page thrashing
 - fragmentation also affects complexity of allocation

4/26/04 22:17

CS386L - Programming Languages - Lecture 14

21

Copying Collector

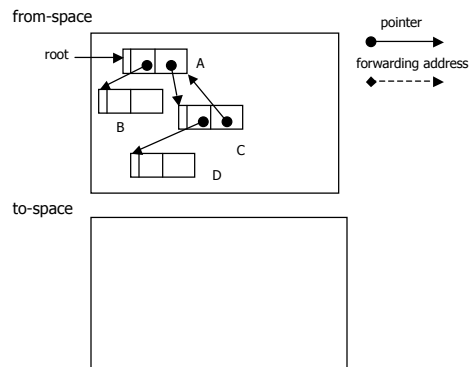
- Divide the heap into two semi-spaces
 - "from-space" and "to-space"
 - during gc, cells in from-space are traced and live cells are copied into to-space using forwarding pointers to keep things linked
 - this process is called "scavenging" since live cells are scavenged into to-space and garbage is left in from-space
 - copying into to-space allows the gc to compact space
 - the roles flip when to-space fills up
 - the old to-space becomes the from-space and the old from-space becomes the to-space, use
 - since we are copying (possibly linked) data structures, pointers have to be updated for roots and cells that point into to-space
 - references in Java and other languages are not pointers for this reason, but are indirect abstractions for pointers

4/26/04 22:17

CS386L - Programming Languages - Lecture 14

22

Example: copying a linked list (Cheney's algorithm)

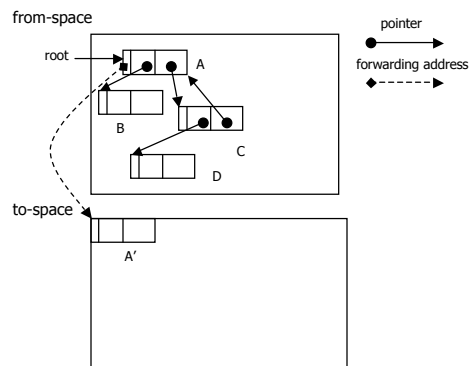


4/26/04 22:17

CS386L - Programming Languages - Lecture 14

23

Example: copying a linked list (cont.)

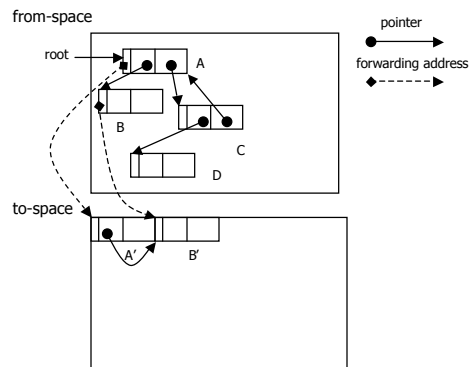


4/26/04 22:17

CS386L - Programming Languages - Lecture 14

24

Example: copying a linked list (cont.)

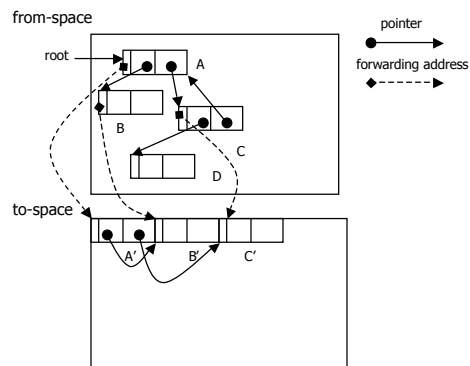


4/26/04 22:17

CS386L - Programming Languages - Lecture 14

25

Example: copying a linked list (cont.)

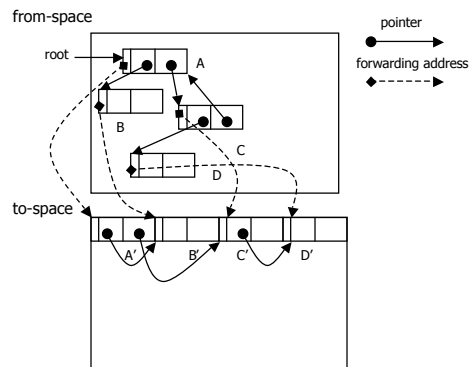


4/26/04 22:17

CS386L - Programming Languages - Lecture 14

26

Example: copying a linked list (cont.)

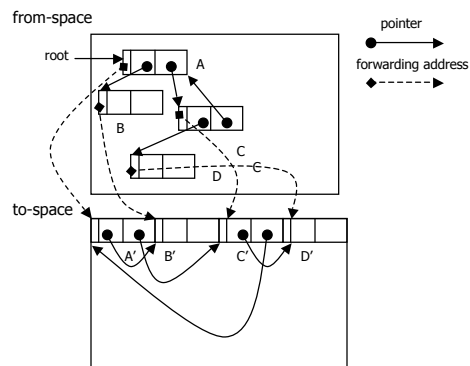


4/26/04 22:17

CS386L - Programming Languages - Lecture 14

27

Example: copying a linked list (cont.)

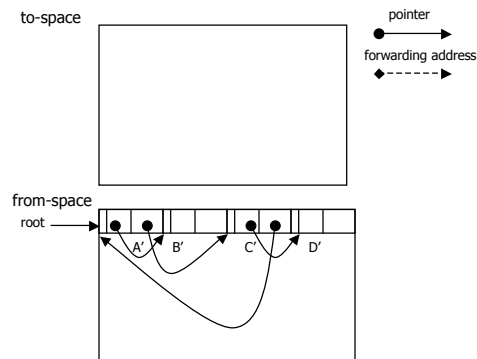


4/26/04 22:17

CS386L - Programming Languages - Lecture 14

28

Example: copying a linked list (concl.)



4/26/04 22:17

CS386L - Programming Languages - Lecture 14

29

Copying collector costs

- strengths
 - much better properties than ref counting or mark-sweep
 - cell allocation overhead very low
 - out-of-space check just requires an address comparison
 - can efficiently allocate variable-sized cells
 - compacting eliminates fragmentation
 - good locality of reference because of compacting
- weaknesses
 - twice the memory footprint over non-copying collectors
 - with 2^{64} addresses, probably not a big deal, except for paging costs
 - when copying, pages of both spaces need to be swapped into memory
 - for programs with large memory footprints (e.g., application in-memory caching) this could be lots of page faults for very little garbage collected
 - ideally, want large physical memory if this is the case

4/26/04 22:17

CS386L - Programming Languages - Lecture 14

30

Generational

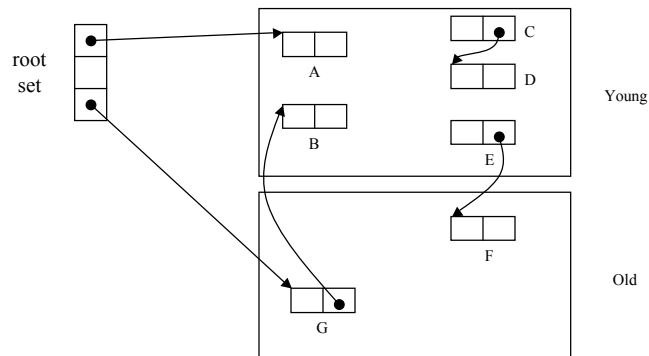
- based on the "weak generational hypothesis" - observation in many language systems that most cells that die, die young
 - divide the heap into generations, and gc the younger cells more frequently since they are statistically more likely to be garbage
 - avoids having to trace all cells during a gc cycle
 - can periodically reap the "older generations"
 - amortize the cost across generations (social security garbage collection :-)
 - e.g., in Scheme inner expressions are younger than outer expressions, so they become garbage sooner
 - (define x (+ 1 2 3 4 5))
 - nested scopes are entered and exited more frequently
 - so temporary objects in a nested scope are born and die close together in time

4/26/04 22:17

CS386L - Programming Languages - Lecture 14

31

Simple generational example (immediate 'aging')

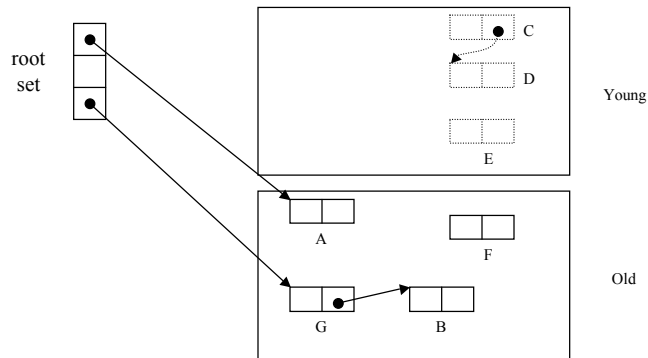


4/26/04 22:17

CS386L - Programming Languages - Lecture 14

32

Simple generational example (cont.)

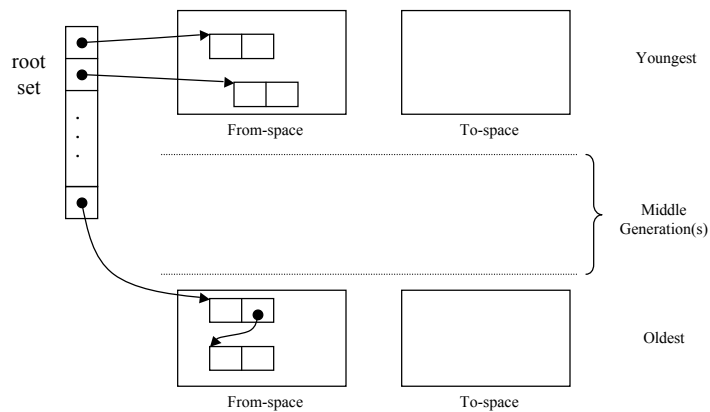


4/26/04 22:17

CS386L - Programming Languages - Lecture 14

33

Generational example with semispaces, >2 generations



4/26/04 22:17

CS386L - Programming Languages - Lecture 14

34



Generational Observations

- Can measure "youth" by time or by growth rate
 - 50-90% of Common Lisp objects die before they are 10KB old
 - 75-95% of Glasgow Haskell objects die within 10KB
 - no more than 5% survive beyond 1MB
 - SML/NJ reclaims over 98% of objects of any given generation during a collection
 - in C programs, one study showed that over 1/2 of the heap was garbage within 10KB and <10% lived for longer than 32KB
 - similar results for smalltalk, self, and other OO languages



References

- *On-the-fly Garbage Collection: An exercise in cooperation*, by E. W. Dijkstra, et al. *CACM*, 21(11):965-975, Nov. 1978
- Art of Computer Programming, Vol 1 - Fundamental Algorithms, by Donald Knuth
- *Uniprocessor Garbage Collection Techniques*, (see course web site), by Paul R. Wilson
- Garbage Collection: Algorithms for Automatic Dynamic Memory Management, by Richard Jones and Rafael Lins
- Papers by Henry Baker
 - <http://home.pipeline.com/~hbaker1/>
- Bohm-Weiser collector for C/C++
 - http://www.hpl.hp.com/personal/Hans_Boehm/gc/