Garbage Collection

- Storage Regeneration
  - no need to free()
  - solves dangling pointer problem
  - partially solves memory leak.

Live objects — will be referenced again
Dead objects — not
Reachable objects — accessible by closure (root set)
Unreachable objects — not

▷ all unreachable are dead
▷ not all reachable are live (mem leak)

Java: reachability

strong — normal reference
softly Rbl — via ~~soft ref~~ SoftReference()
- may be collected, gc discretion
  mem low
- all soft refs cleared before
  out of mem error.
- caching behavior (browser)

LRU

weakly Rbl — via ~~weak ref~~ WeakReference()
- will be reclaimed
- usually: annotations about object
- time consuming to compute but
  don't outlive object

finalizer reachable — finalize() not run yet
phantom reachable — has been finalized ∴ dead
- reachable via phantom ref
- informed abt state change.

unreachable

Liveness = global property
alloc = local, outlives creator frame
leaks & dangles = largest pgmr investment

[dbx / check -all.]

hardest to diagnose

break basic abstractions.

prob with refs — referential transparency

x = 7 } same or diff 7
y = 7

x = 9 } — does this change the 7 of y?
y = x } — y same 9 as x?
x = 2 } — change y's 9?

expensive?

⇒ Appel - Gcol can be faster than stack alloc.
⇒ good gcol slows down pgm ≈ 10%
⇒ Moore's law: CPU doubles in 18 months
10% = 2½ months

⇒ reliability

Two phase

1. distinguish live (reachable) vs dead (unrch)
2. ~~recycle~~ reclaim dead obj

- reachable $\triangleq$ accessible from rootset
  | from other reachable obj

rootset = { ptrs in static + ptrs on stack }
— must flush regwin

Obj repr

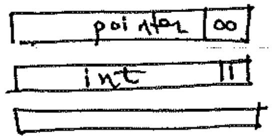
- need RTTI - what type are heap objs?
- self ident
- C - can't:

```
p = i << 3
Q = j >> 6
m = p + Q
*(int*) m = 6
```

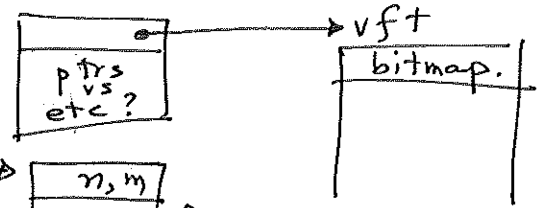
tags

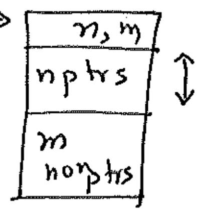
104B
Gcol-3

Obj repr

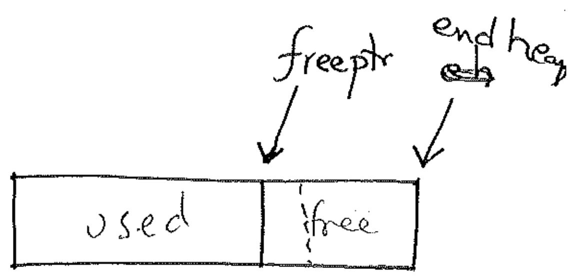
– tagging: 

etc.

– object header (00) 

– gc fast
∴ reorg ptrs. 

algs: ref count
mark & sweep
mark – compact
copying with semispaces.
non-copying implicit
conservative
~~incremental~~
~~generational~~
~~po~~ concurrent




```
fastmalloc(n) {
    n = (n + 7) & ~7 //roundup.
    if( freeptr + n > endheap) call gcol()
    r = freeptr
    freeptr += n
    return r
}
fastfree (p) {∅}
```

104B
Gcol-4

Ref counting

overhead: one count / node.

$p = Q \longrightarrow op = \{$

1. $t_1 = p$
2. $t_2 = t_1.count$
3. $t_3 = t_2 - 1$
4. if ($t_3 == 0$) {
   ~~call free(t_1)~~
5. $arg\emptyset = t_1$
6. call free }
7. else {
8. $t_1.count = t_3$ }
9. $t_4 = Q$
10. $t_5 = t_4.count$
11. $t_6 = t_5 + 1$
12. $t_4.count = t_6$
13. $p = t_4$

free: decr ptr fields
∴ recursive.

incremental
∴ no stop the world.

Perl uses it

interp ⇒ not signif slow

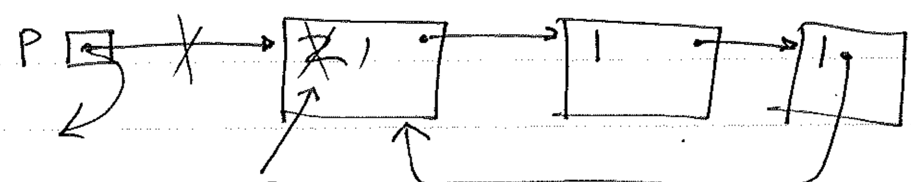
compile ⇒ ld/st 2 insn ⇒ 13.

Overhead 6 x slower.

not satisfy real time requirements

progs
- ▷ batch
- ▷ interactive
- ▷ real time
  - ▷ soft
  - ▷ hard.

Cycles:



when 2 → 1, p = ? dead cycle.

Efficiency

- local vars → defer ref counting
- for (p = ...) clunk down list
  - don't register p as a ptr.
  - extra work for compiler
- ref counts adj only for heap → heap objs.

smart ptrs

104B

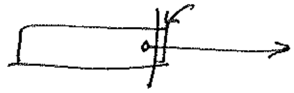
Gcol -5

Mark & Sweep

```
mark: foreach p (all static ptrs, all stack ptrs ≡ Rootset) {
                mark(p);
      }
mark (p) { if (p -> mark == false) {
                 p -> mark = true
                 foreach Q (p -> someptr) {
                       mark(Q)
                 }
         }

sweep: p = begin heap;
       while (p < end heap) {
             if (p -> mark = true) p -> mark = false
             else { put p on free list
                    p += sizeof p -> object
             }
       }
```

problems
- memory frags like malloc/free
- alloc large obj difficult
- cost = O (size of heap)
- locality of reference => not page friendly
  - large working set
- bit map overhead = 1 bit/cell
  - during collect
  - could finesse a bit in header.



Small vs Large Object areas

Hash consing of small objects

- fixed size lists

gcol 5

Mark - Compact

- remedy frag problem.
  1. mark (closure (rootset))
  2. scan heap (entirely)
     - ident live objs & set forward ptrs
  3. scan root set and heap
     - adjust ptrs to objects
  4. scan heap and slide all objs
  5. maybe sbrk() to adjust heap size.

```
fastalloc (n) {
        p = newp;
        newp = (newp + n + 7) & ~7 ;
        if(newp > break) {
                break = (newp + reserve + pgsiz - 1)
                        & ~ (pgsiz - 1) ;
                sbrk(break)
                assert (ok)
        }
        return p
}.
```

pgsize $= 2^K$.
Sparc = 8K
4K possible

- elim frag
- still pisseson working set.
- even worse: 3 scans of whole heap.

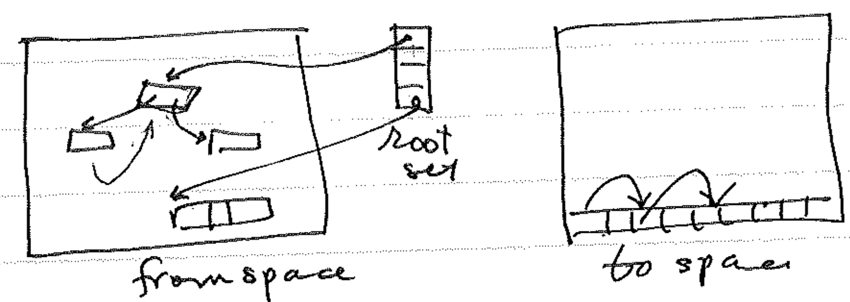
gcol 7

Copying Collection

- does not collect "garbage"
- collects live objects
- speed $O(n)$, n = # live objs
  indep of heap size.

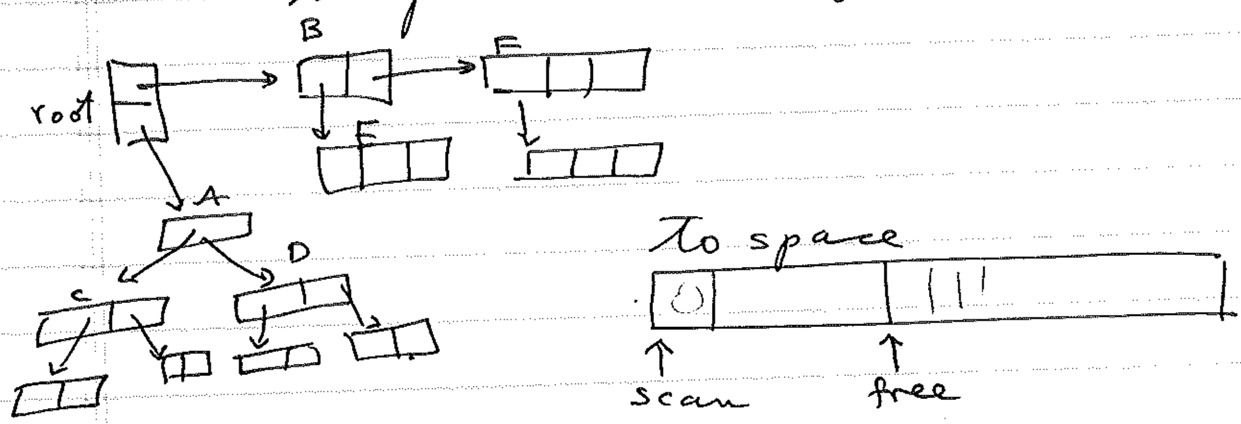
Stop and Copy with Semi Spaces

- semispace: divide heap into 2 pises
  use only one,
  except during gc, other is swapped out



Cheney alg

simple breadth-first search



gcol 8

```
~~cheneygc() {~~
    ~~foreach (obj in root set)  copy to (o)~~
cheney gc() {
    scan = free = & start of tospace
    foreachp(ptr in root set) { ~~copy (p)~~ p = copy (p) }
    while (scan < free) {
        foreach (p ∈ scan -> object) { p = copy(p) }
        scan += scan -> sizeof
    }
}
    ~~copy (p) {~~
        ~~if (p ∈ tospace) return p;~~
        ~~np = fastalloc (p);~~
        ~~copy p -> object to np -> object.~~
copy (p) {
        if (p -> fwdadr ∈ tospace) return p;
        np = fastalloc (p) ;
        copy( p -> object, np -> object);
        p -> fwdadr = np;
}
```

---

note : overhead of one ptr per obj
- oo need it anyway
- use classptr as fwd ptr.
- classptr ∈ sdata seg
- fwdptr ∈ to space .

gcol #9

Efficiency

- arbitrarily, given lots of mem.
- classic time vs space tradeoff
- obj become garb before coll not collected

ex: 20MB alloc, 1MB live.

have 2 × 3MB semis → 10 ×

1/3 full after coll. → 2MB for alloc.

have 2 × 6MB semis

→ 5 MB free, ea coll.

3 → 4 × gcol.

Appel

Gcol faster stack alloc.

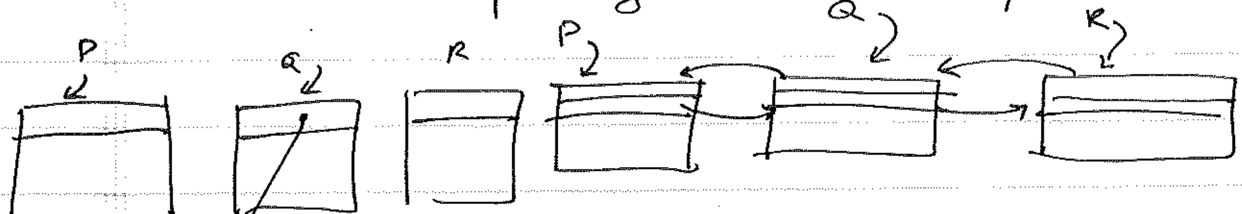
Non copying collection

3 sets:

live | reachable | garbage

non copying uses 2 ptr fields and a color.

- doubly linked list as a set
- can optimize into one ptr.



→ store Q→link = p ^ R;

- can still frag.
- doesn't need compiler cooperation
- copying collects req fascist ptr mgmt.
  - native methods must register their ptrs.

gcol # 10

Costs

alloc, init, free (gcol)

~~#~~ # of insns for each.

- differ in c.

$O(n)$ is really $T(cn)$

if cost (sweep) * 10 = cost (copy)

then marksweep ≈ copy.

(assume 10% survives gc)

- use Large object area
  - copy small objs.
  - sweep large objects.

Conservative Collectors (Boehm)

- used by gcj
- mark sweep
- if it looks like a ptr it is a ptr.
- small amt of leak.
- smaller w/ larg V. adr space

Simple Tracing

asymptotic cost of copying $\xrightarrow{\lim} 0$

as mem $\to \infty$

- large mem: expensive
- VM → poor locality of reference.
- RAM speed = 1000 to 10000 * disk speed

- collection itself messes wk. set

gcol ||

Incremental

- req in RT apps.
- fine grained incr gc.

- ref counting is.
- parallel threads: mutator + collector.
- similar to readers/writers problem.

tricolor marking

black - to be retained
white - obj subject to gcol
gray - obj is reachable but not scanned
black → gray → white.

Generational Collection

- most obj live very short time
- small % long lived.
- pgm speed measured by heap alloc.
- large frac of obj that survive one survive many
- collecting youngest generation quick

Multiple Subheaps

- must be able to collect young gen w/o older one.

- need a write barrier
- scan dirty bits.

gcol #12

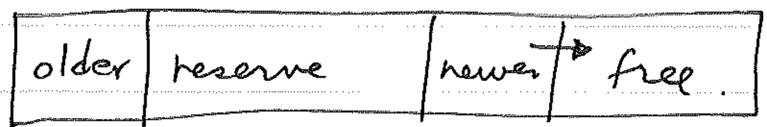
great if few assignments
gc speed so fast overhead
of alloc dominates o/H of g.

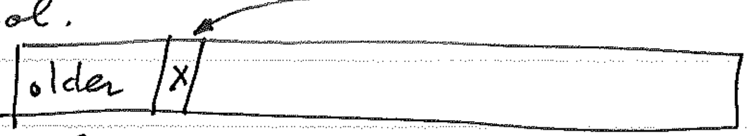
- not so great if lots of store opns.

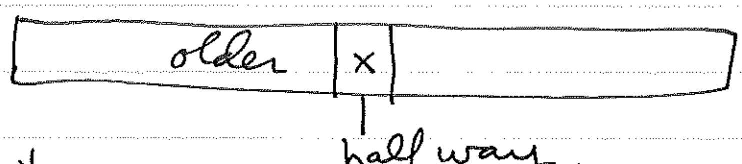
<u>2 Gen</u>

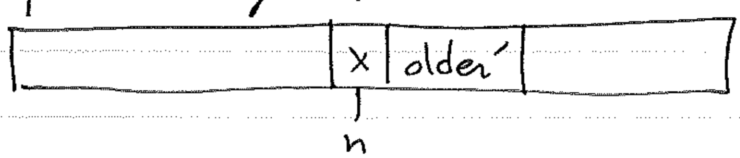
M = mem size
A = live data

$M > 2A$.

bss


gcol.


eventually


collect ~~copy~~ older region


blockmove


gcol 13

Tracking intergen refs

indirection tables
pagemarking (write barrier via hardware)
store lists - bags.
bitmap $\frac{1\text{bit}}{16\text{bytes}} = 0.78\%$ of mem.
slow

Locality

- programming style.
- direct gc effects.
- indirect effects — reallocation