Garbage Collection

Idea: Compiler/runtime automatically frees dead objects

- no dangling pointers
- memory reclaimed automatically
- can have faster memory allocation & better memory locality

Two main styles:

- reference counting
- tracing
  - mark/compact, mark/sweep
  - copying
  - lots of kinds: incremental, conservative, generational, distributed

Reference Counting

For each allocated object,
maintain count of # of pointers to object

- count initialized to 0 when object created
- on creating a new reference to object, increment count
- on deleting a reference to an object, decrement count
  - if count = 0 then delete object

Example:

```
x = malloc()
y = malloc()
y = x
x->next = malloc()
x->next->next = y
x = y = malloc()
```

```
X:


Y:
```


Pros and Cons

+ local, incremental work
+ little or no language/runtime support required
+ reclaims memory immediately

– cannot reclaim cyclic data structures
– high run time overhead (10-20%)
  - deferred reference counting [Deutsch & Bobrow 76]
– free memory is fragmented

Tracing Collectors

Start with set of **root** pointers (pointers from outside the heap to objects in the heap)

- global variables
- contents of stack & registers

Traverse objects transitively, starting from roots

- “live” objects are those visited on this traversal
- all other objects are dead

Key issues:

- how to identify pointers?
- how to know when an object has been visited?
- how to free unvisited objects?
- how to synchronize collector and program (**mutator**)?

Identifying pointers

Two approaches:

"accurate":

- static type info + compiler support
- run-time tagging scheme
- conventions about where pointers can be

Conservative

- "*if it looks like a pointer, it must be a pointer*"

Mark/sweep collectors

stop-the-world tracing collector [McCarthy 60]

When heap fills:

1. stop the application
2. trace reachable objects
   - set mark bit in each object
   - traversal control via depth-first-search
3. sweep through all of memory
   - add unmarked objects to free list
   - clear marks of marked objects

Pros and cons

+ collects cyclic data structures
+ simple to implement

– "embarassing pause" problem
– work proportional to total memory, not live memory
– poor memory locality during marking and sweeping phases
– free memory is fragmented

Collectors that move objects

Why?

+ better memory locality
+ fast allocation
– more complex
– requires accurate information about pointers

Fast allocation:

```
x = allocate_bytes(12);
```

If free memory is contiguous, allocation can be very fast:

```
if (freePtr + 12 > endOfHeap)
  doGarbageCollect(12);
x = freePtr;
freePtr += 12;
```

(6 instructions if we dedicate a register to hold `freeptr`)

If free memory is fragmented, allocation more complicated:

- must manage free lists of available memory
- allocation must search free lists for available space
- fast malloc implementations: around 40 instructions

Mark/compact collectors

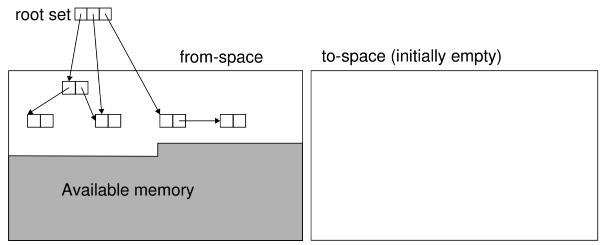
When sweeping through memory, compact rather than free

+ free memory is all contiguous ⇒ fast allocation

– slower to sweep
– more complex
– work still proportional to total memory, not live memory
– requires changing pointers
  ⇒ must have accurate pointer info

Copying collectors [Cheney 70]

Divide heap in two equal-sized **semi-spaces**:



When from-space fills, do a GC:

- visit objects referenced by root set
- on visiting an object:
  - copy to to-space
  - leave forwarding pointer in from-space version
  - if visit object again, just redirect pointer to to-space copy
- scan to-space linearly to visit reachable objects
  - when done, all live objects have been copied to to-space
- when done scanning to-space
  - empty from-space
  - **flip**: reverse roles of to-space and from-space
- restart mutator

Pros and cons

+ collects cyclic structures
+ supports compaction & fast allocation automatically
+ no separate stack required (to-space acts as queue)
+ work proportional to live memory, not all of memory

– requires twice the (virtual) memory
  physical memory moves back and forth between GCs
– still has "embarassing pause" during GC
– changes pointers
– copying can be slow

An improvement

Add a large object space [Caudill & Wirfs-Brock 86]:

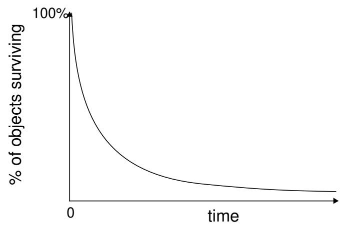
- big objects slow to copy, so allocate them in separate area
- use mark/sweep in large object space

+ no copying of large objects
– more complex

Generational GC

Observation: most objects die soon after allocation

- e.g. closures, cons cells, stack frames, numbers



Idea: concentrate GC effort on young objects

- divide heap into multiple generations
- gc each generation with different frequencies/algorithms

Original idea: Peter Deutsch

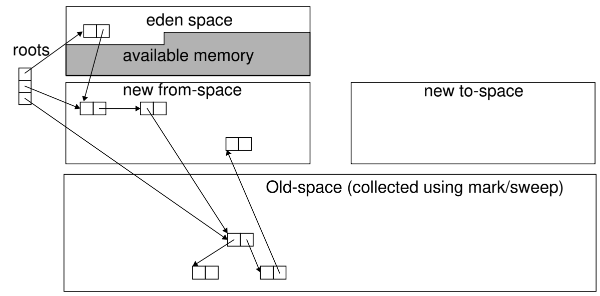
Generational mark/sweep: [Lieberman & Hewitt 83]

Generational copying GC: [Ungar 84]

Generation scavenging [Ungar 84]

2 generations: **new-space** and **old-space**

- new-space managed using 3-space copying collector
- old-space managed using mark/sweep
- new-space much smaller than old-space



New objects allocated into eden space

When eden fills up, do copying GC of eden + new from-space to new to-space ("**scavenge**")

- If object survives many scavenges, then copy it to old-space
  - maintain age field in each object

If old-space (nearly) full, do a full GC

- old space managed using mark/sweep

Roots for generational GC scavenges

Must include pointers from old-space to new-space as roots when scavenging new-space

How can we find these?

Option 1: scan old-space at each scavenge

Option 2: keep track of pointers from old-space to new-space

What granularity to remember these old-to-new pointers?

- individual words containing pointers [Hosking & Moss 92]
- **remembered set** of objects possibly containing pointers [Ungar 84]
- **card marking** [Wilson 89, SELF system]

How should creation of old-to-new pointers be detected?

- functional languages: easy! (can't happen)
- imperative languages: need a **write barrier**
  - specialized hardware [Berkeley SOAR, some Lisp machines]
  - standard page protection hardware (OS folks: we want fast VM traps!)
  - in software, inserting extra checking code at stores

Pros and cons

+ scavenges are short: fraction of a second
+ low run-time overhead
  - 2-3% in Smalltalk interpreter
  - 5-15% in optimized SELF code
+ uses less VM space than pure copying
+ better memory locality than pure mark/sweep

– more complex
– requires write barrier
– "embarassing pauses" still occur during full GCs
– need space for age field

Extensions

Multiple generations
- may eliminate need for age field

Feedback-mediated tenuring policy [Ungar & Jackson 88]

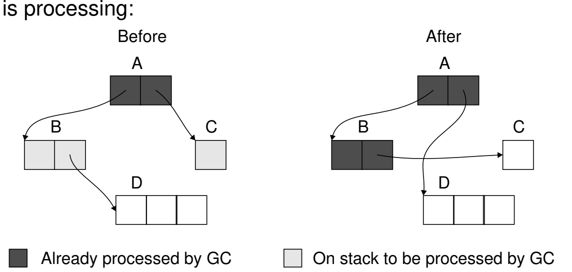
Large object space

Incremental & parallel GC

Want to avoid long "embarassing pauses"
- for interactive applications
- for real-time applications

Idea: run mutator and collector in parallel
- real parallelism (multiprocessor)
- simulated parallelism: mutator and GC as separate threads

Problem: mutator may change reachable object graph while GC is processing:



GC must see changes to ■ objects!

Need to synchronize collector and mutator

Two approaches: read barrier and write barrier

Read barrier [Baker 78, Moon 84]:
- standard Cheney-style copying collector
- mutator is never allowed to see pointers into from-space
- whenever object is accessed, check to see if is in from-space
  - If so, copy it to "to-space"

– requires check on every load of heap pointer ⇒ can be slow

Write barrier methods:

can be used for non-copying incremental GC

- make sure GC system sees writes to already visited objects

Snapshot-at-beginning [Yuasa 90]:

- on pointer write to heap, save overwritten value and push onto GC marking stack
- ensures no object will become unreachable during GC traversal

– overly conservative: some objects might be garbage

Incremental update write-barriers [Dijkstra 78; Appel, Lewis & Li 88]

- on pointer write to already processed object, push object onto GC traversal stack again

Write-barrier requirements almost identical to those for generational GC ⇒

+ making a generational GC system incremental not too hard

Conservative GC systems

[Barlett 88, Boehm & Weiser 88, Boehm 93]

Pointers may not be known:

- unsafe languages (e.g. C)
- polymorphic languages like ML
- language independent intermediate languages

Can still do GC by being **conservative**

- don't free anything that might be referenced by a pointer

  "*if it looks like a pointer, assume it is*"

Can't compact or copy:

- Can't update something that only *might* be a pointer

What might be a pointer?

Make assumptions about what might be a pointer

- don't be wrong, but...
- don't retain too much unnecessary garbage

Approaches differ in optimism about application's behavior:

Optimistic:

- assume pointers are word aligned
- pointers point to beginning of heap-allocated objects

Less optimistic:

- pointers might point into the middle of an object

Even less optimistic:

- pointers might point just before or just after an object

Really unoptimistic:

- don't require alignment of target pointer
- or maybe even of pointer?

Can be thwarted by compiler optimizations

Compiler can hide pointers in mysterious ways:

```
int i, sum;
a = (int *)malloc(100000 * sizeof (int));
b = (int *)malloc(100000 * sizeof (int));
for (i = 0; i < 100000; i++) {
    sum += a[i] + b[i];
}
```

Compiler might do the following:

```
diff = b-a;
for(aptr = a; aptr < a +100000; a++) {
    sum += *aptr + *(aptr+diff);
}
```

Uh oh. Where's the pointer to `B`?

Somewhat good news: Almost never happens in real programs

Longer-term solution: gc-safe mode for C compilers to prevent certain optimization that hide pointers

Pros and cons

+ Supports GC even in systems that weren't designed for it
+ relatively easy to include

```
#include "gc.h"
#define malloc GC_malloc
```

– still not completely conservative in the face of some opts
– for some programs, may retain lots of garbage
  - being careful about where you allocate objects can alleviate this: see [Boehm 93]
– must forgo advantages of compaction, copying
  - allocation is slow, especially for high-allocation rates

Mostly copying collectors

[Barlett 88, Bartlett 89]

Assumptions:

- Pointers in root set are not known
- Within heap-allocated objects, pointers are known:
  - compiler support, or
  - user registers pointers explicitly

Solution:

- treat roots conservatively
  $\Rightarrow$ can't move an object referred to by a root "pointer"
- can move objects referred to only by other heap objects

Pros and cons:

+ gives most of the benefits of copying/compaction
  - better locality
  - allows most free storage to be contiguous $\Rightarrow$ fast allocation
– more difficult to use without compiler support
– not completely conservative
– more complex