Copying Garbage Collection

- Recall: use reachability as conservative approx. of when it is safe to reclaim objs.

- Copying is a different take
  - allocate objs until heap is full
  - find all reachable objects
    and copy them from current heap to new heap
  - reclaim all of old heap (reachable + unreachable)
    → actually, keep old heap around
      to serve as the new new heap at next GC
    → but, this "bulk reuse" of whole heap
      is faster than reclaiming individual objs
      (trading space for time)
Copying GC: Example (Simple)

- **from space** ("old heap")
- **Stack**
- **to space** ("new heap")
Copying GC: Example (Simple)

from space ("old heap")

Stack

to space ("new heap")
Copying GC: Details

- Copying into a contiguous prefix of to-space leaves a contiguous suffix of to-space available
  - Use "bump pointer" allocation:
    - At end of copying, alloc ptr at start of available region
    - At allocation, increment alloc ptr by requested size
      (no fragmentation or irregular obj size issues)
    - Much faster than free-list
Copying Collection : Details

- Two issues
  - a recursive copying function would use stack space (don't want to use mem. to reclaim mem.)
  - must preserve sharing

\[
\begin{align*}
\begin{array}{c}
1 \\
1 \\
1
\end{array}
\end{align*}
\]

should not be copied as

\[
\begin{align*}
\begin{array}{c}
1 \\
1 \\
12
\end{array}
\end{align*}
\]

(would use more space and, with mutation, would change semantics)

- related to both issues, beware of cycles in object graph
Copying Collection: Details

- Solutions (Cheney's Alg)
  - use to-space as a queue of "to be copied" objs.
  - when obj. is copied, install forwarding pointer;
  - when obj. would be copied again, use forwarding ptr.

- Algorithm
  1) **Forward roots**
     - copy objs pointed to and update pointers
  2) Scan to-space for objs w/ fields pointing to from-space;
     - forward such fields
  3) GC done when all objs in to-space scanned
Copying GC: Example

From:
A B C D E F G H I J

To:
N O P Q R S T U V W

↑ alloc
↑ scan
Copying GC: Example

from A B C D E F G H I J T

from N O P Q R S T U V W

↑alloc

↑scan
Copying GC : Details

ptr alloc(sz) {  
    res = alloc;
    alloc = alloc + sz;
    return res;
}

ptr forward(p) {  
    if (p->fwd != NULL) {  
        return (p->fwd);
    }
    q = alloc(p->sz);
    memcpy(q, p, p->sz);
    p->fwd = q;
    return q;
}
Copying GC: Details

```plaintext
scan() { while (scan < alloc) {
  p = scan
  foreach f in fields(p) {
    p->f = forward(p->f);
  }
  scan = scan + p->sz;
}
}
```
Copying: Analysis

- Per object overhead
- Allocation cost
- Pause time
  - higher constant factors due to writes/copying
- Notes
  - ½ of heap unused when not GCing
  - BFS graph traversal
    may affect locality
  - Objects with long lifetimes may be copied many times
    (but most objects die young)
  - motivation for generational copying collection
Heap Resizing

• What happens if a GC does not reclaim many objs?
  \( \Rightarrow \) Will need to GC again very soon.

• What happens if a GC does not reclaim any objs?
  \( \Rightarrow \) Will need to grow heap.

• Recall:
  \( H \) - size of heap
  \( L \) - amount of live data
  \( \gamma = \frac{H}{L} \) - ratio of heap size to live data
  \( \frac{1}{\gamma} \) - fraction of heap occupied by live objs

• If \( \gamma \) too small
  \( \Rightarrow \) frequent GCs (and \% of program time in GC very high)
Heap Resizing

- Start with a small heap, and grow in response to growth of live data.
- After each collection, compute $L$ and $\gamma$.
  
  If $\gamma$ too small, then increase $H$ (by requesting mem. from OS).
  If $\gamma$ too large, then shrink $H$ (by returning mem. to OS).

  - Why give mem. back? “play nice” w/ other programs fit into physical memory

  $\Rightarrow$ Maintain a roughly constant $\gamma$

- Mark-Sweep performs well w/ $\gamma > 2$
- Copying performs well w/ $\gamma > 3$
- Production systems often target $\gamma \approx 8$
Advanced Features of GCs

• The mark-sweep and copying GCs are very basic. Pause times are a major concern.

• Incremental: do a few steps of GC after every few steps of program execution or at each allocation by program.
  + if enough GC steps done often enough, then no long pauses (just very many very short ones).
  - more complicated relationship b/w GC and mutator.
    ⇒ GC must be prepared for changes to heap made by mutator.

• Real-time: stronger version of incremental, guarantee max pause time in any time window.
  + req’d for certain applications.
    - usually very conservative (reserve much more time than typically needed to defend against worst case).
Advanced Features of GCs

- **Concurrent**: GC executes at the same time as mutator (in a separate thread)
  - if GC runs fast enough, then low/no pause times
  - complex coordination b/w GC and mutator

- **Parallel**: GC executes using multiple threads (but mutator paused)
  - lower pause times (b/c GC work split b/w threads)
  - complex coordination b/w GC threads
  - for good performance, want to balance work among threads
    (but can be hard: how to parallelize marking a long list?)

- **Concurrent + Parallel**: GC executes using multiple threads at the same time as mutator
  - better pause times
  - complex coordination among GC and mutator threads
Advanced Features of GCs

- Many of these features work best w/ mark-sweep → non-moving makes coordination w/ mutator easier
- Many, many variants and novel GCs
- Remains an active area of research
  - Prog. Lang. Design and Implementation (PLDI)
  - International Symposium on Mem. Mgmt. (ISMM)
  - International Conf. on Functional Prog. (ICFP)
  - The Garbage Collection Handbook
    by Hosking, Moss, and Jones