Mark-sweep GC

Optimization for marking and sweeping

Optimization for marking

- Use a marking stack
- Iterative marking
- Minimize stack depth to avoid stack overflow
  - Knuth: treat marking stark circularly
  - Kurokawa: remove items from stack that have fewer than 2 unmarked children
  - Pointer reversal: eliminate need for marking stack
  - Bitmap marking: store in memory if small enough

Pointer reversal variable sized nodes

- Each object had 2 additional fields
  - n-field: holds # of pointers in object
  - i-field: used for marking (large as a pointer)
    - Number of sub-trees fully marked
- i-field initialized to 0
- i > 0: Object is marked
- i == n: All children of object are marked

Pointer reversal: features

- Recycles 3 variables (current, previous, & next)
- Conceal marking stack in heap objects
  - Reduces space overhead
- Time overhead is significant
  - Visits each branch node n + 1 times
  - Each visit requires additional memory fetches
    - Memory fetches are expensive
  - Each visit recycles values and modify flags

Verdict on pointer reversal

- Use only as a last resort to address stack overflow
- Avoid otherwise

Bitmap marking

- Finding bits for bit mapping:
  - In object’s header
  - In object’s address
  - In a separate bitmap table

What is bitmap marking

- One bit represents start address of object in heap
- Bitmap size inversely proportional to size of smallest object
- Bit corresponding to object's address is found my shifting bits in object's address

Bitmap marking example

- Consider:
  - 32-bit architecture
  - Smallest object ~ 8 bytes
- Size of bitmap == 1.5 % of heap
- If *addr* is start address of object *obj*, then

```
mark_bit(addr) {
        return bitmap[addr >> 3]
}
```

Advantages of bitmap marking

- Space overhead is negligible
- Bitmap mostly like can be stored in RAM
- # of bitmaps decreases with larger objects
- Heap does not have to be contiguous
- Objects do not have to be touched when GC runs

Disadvantages of bitmap marking

- Mapping object’s address to bit in bitmap more expensive than if bitmap were stored in object

Optimization for sweeping

- Lazy sweeping
  - Problem:
    - Sweeping phase expensive
  - How do we solve it?
    - Pre-fetch pages or cache lines
    - Not likely to affect virtual memory behavior
  - Problem:
    - Sweep causes long delay in user program
  - How do we solve it?
    - Run sweep phase in parallel with mutator

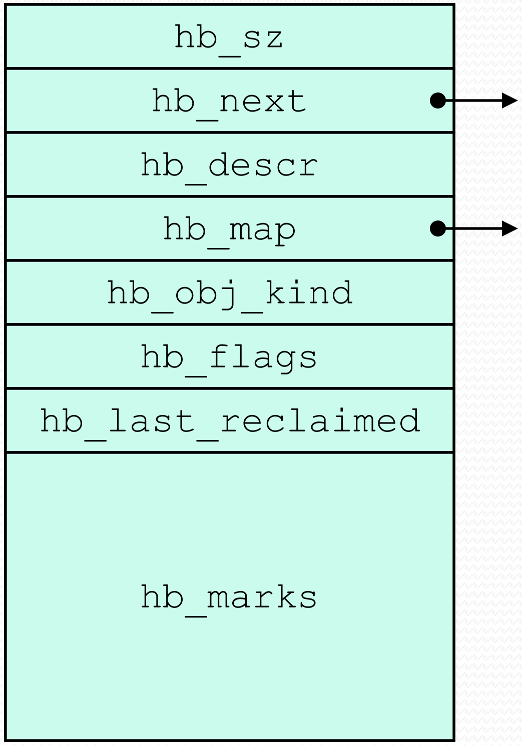
Hughe’s lazy sweep algorithm

- Executes sweeper and mutator in parallel
- Do a fixed amount of sweeping at each allocation
- Transfers cost of sweep phase to allocation
- No free-list manipulations necessary
- Performance reduced by bitmaps
  - Performs better when mark bit stored in object

Boehm-Demers-Weiser sweeper

- 2-level allocation:
  - low-level: acquire 4 KB blocks from OS for single sized objects
    - using malloc or other standard allocator
  - high-level: assign individual objects to the blocks
- free-list for each object size, threaded through blocks allocated for that size
- Each block has separate block header
  - Chained together in linked list
- Queues for reclaimable blocks maintained
  - Next unswepped block is dequeued and swepped

Block header



hb_sz	Size of objects in block
hb_next	next block header to be reclaimed
hb_descr	
hb_map	
hb_obj_kind	(atomic, normal)
hb_flags	
hb_last_reclaimed	
hb_marks	mark bits

Zorn’s lazy sweeper

- Allocates from a cache vector of n objects for each common object size
- Uses no free-lists
- When vector is empty, sweep to refill it
- Sweeps and allocates very rapidly

Mark-sweep (MSGC) vs RCGC

- MSGC places less overhead on user program
- RCGC reclaims garbage immediately
- RCGC causes user program shorter pause times
- MSGC reclaims cyclic structures naturally
- RCGC is naturally incremental
- RCGC has better locality
- MSGS only touches live objects once if separate bitmaps