CS61: Systems Programming and Machine Organization
Harvard University, Fall 2009

Lecture 12:
Dynamic Memory Allocation 3

Rohan Murty

October 13, 2009

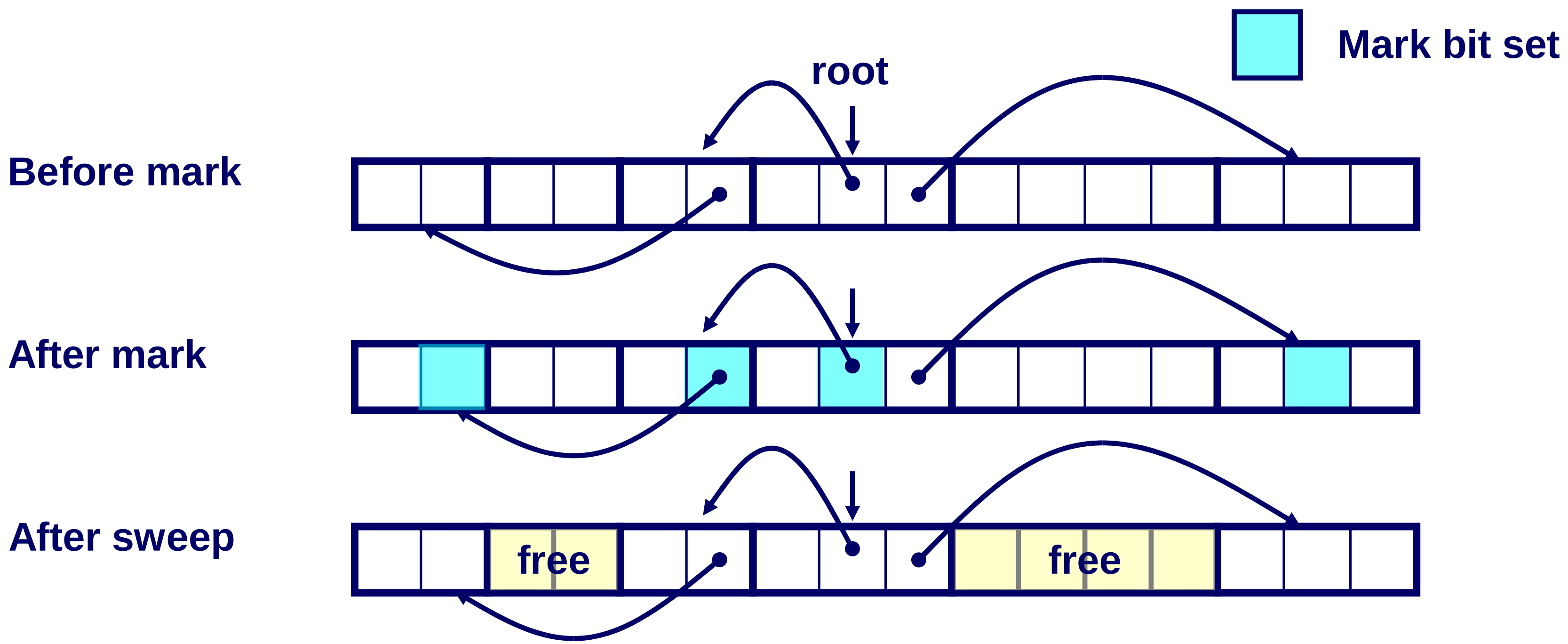
Road map for Today

- Mark and Sweep GC
- Generational Garbage Collection
- Reference Counting
- glibc malloc implementation
- Buddy allocation
- Measuring malloc performance

Mark and Sweep GC

- Idea: Use a **mark bit** in the header of each block
  - GC scans all memory objects (starting at roots), and sets mark bit on each reachable memory block.

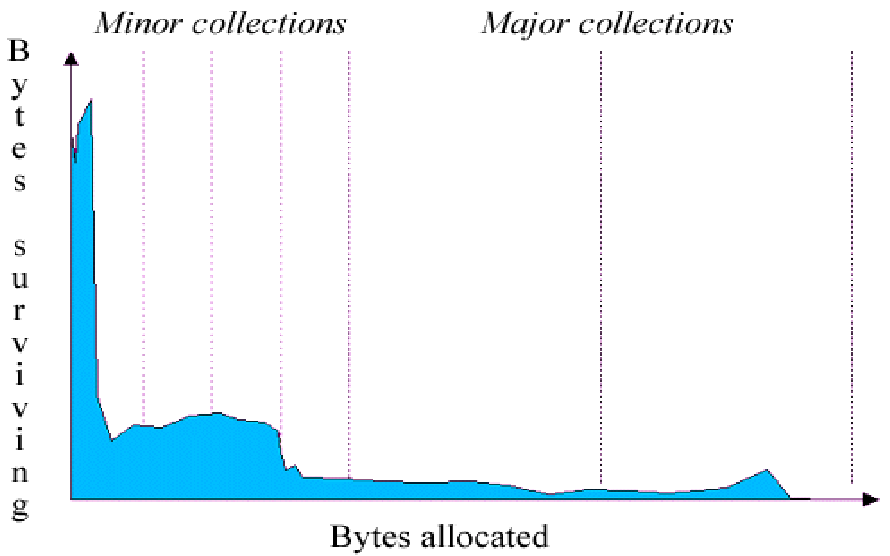
- Sweeping:
  - Scan over all memory blocks.
  - Any block without the mark bit set is freed



Reflecting on Mark and Sweep

- Advantages
  - Easily handles cycles
  - Low overhead

- Large object set
  - Scanning all objects is *slow*
  - **Expensive!**



Most objects have short lifespans

→ Most objects created post previous GC Cycle

An object that survives a few GC cycles → Likely to survive longer

Generational Garbage Collection

- Used in Java and .NET

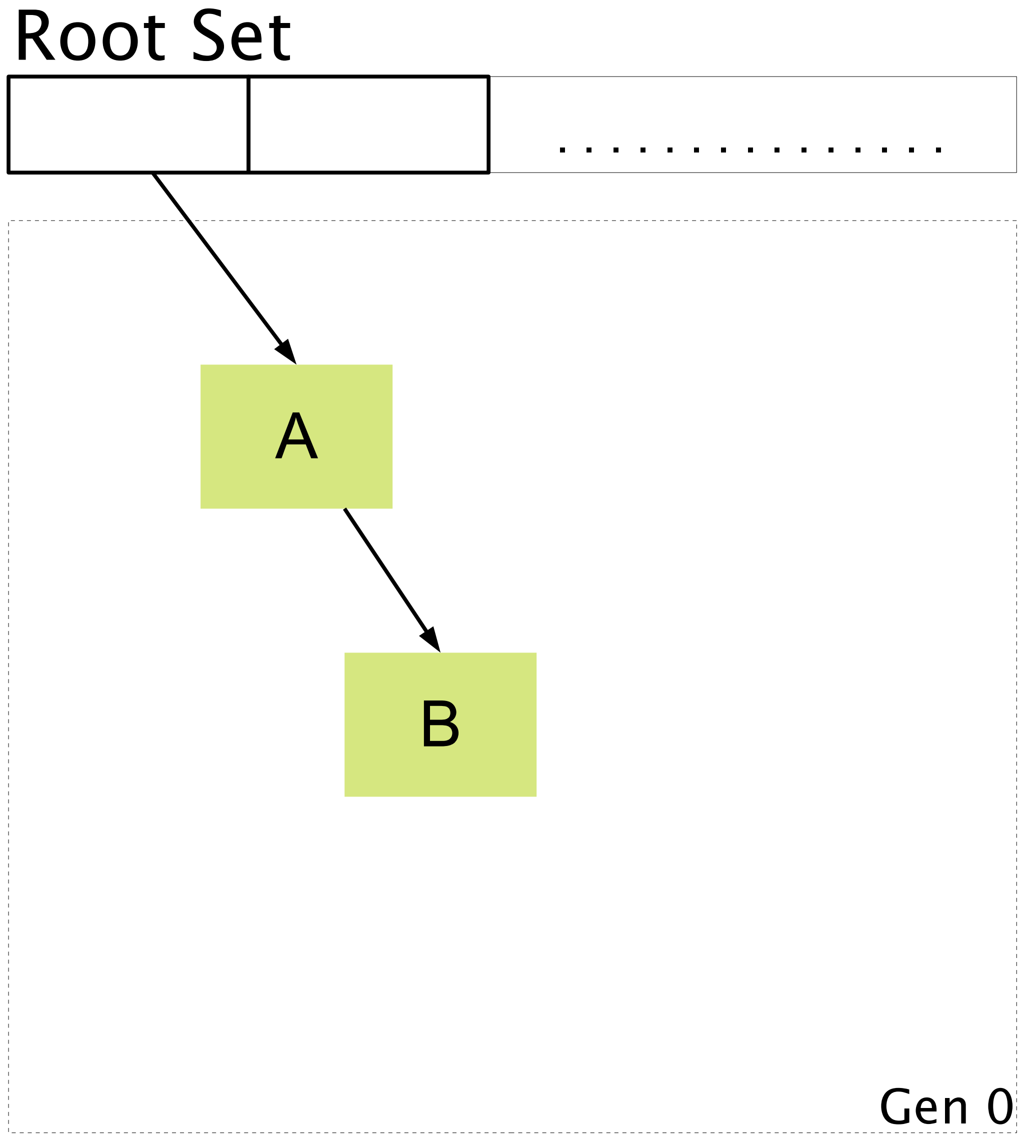
- Fast incremental GC
  - Based on mark and sweep
  - Classify objects into 'generations'
  - GC certain generations more aggressively than others

- Scan “newer” objects more often than “older” ones

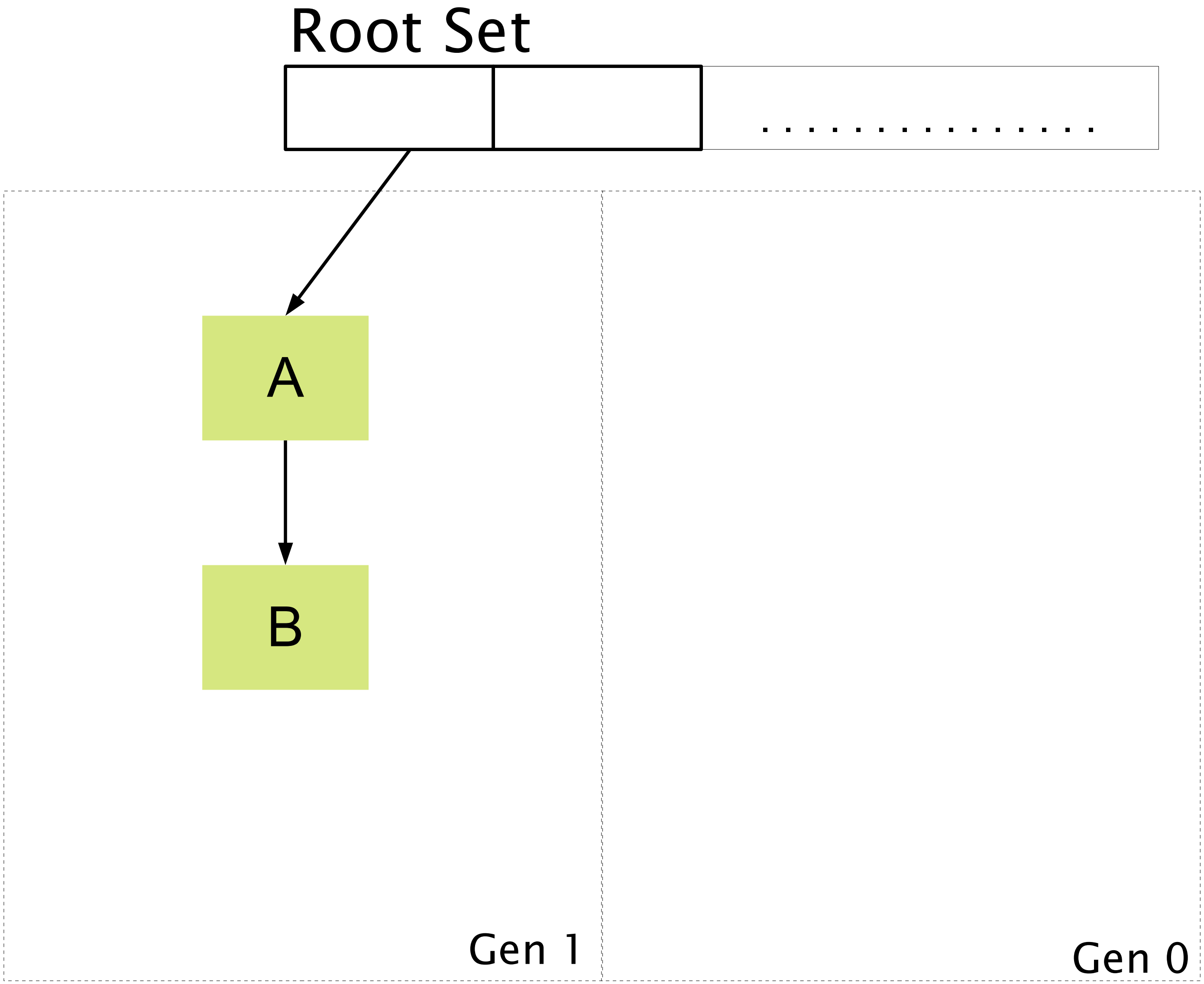
Generational Garbage Collection

- Algorithm:
- Divide objects into generations (Gen0, Gen1...)
- Aggressively GC lower generation objects
- If a region is filled with objects
  - Perform GC
  - For Objects of Gen(n) that survive GC
    - Promote to Gen(n+1)

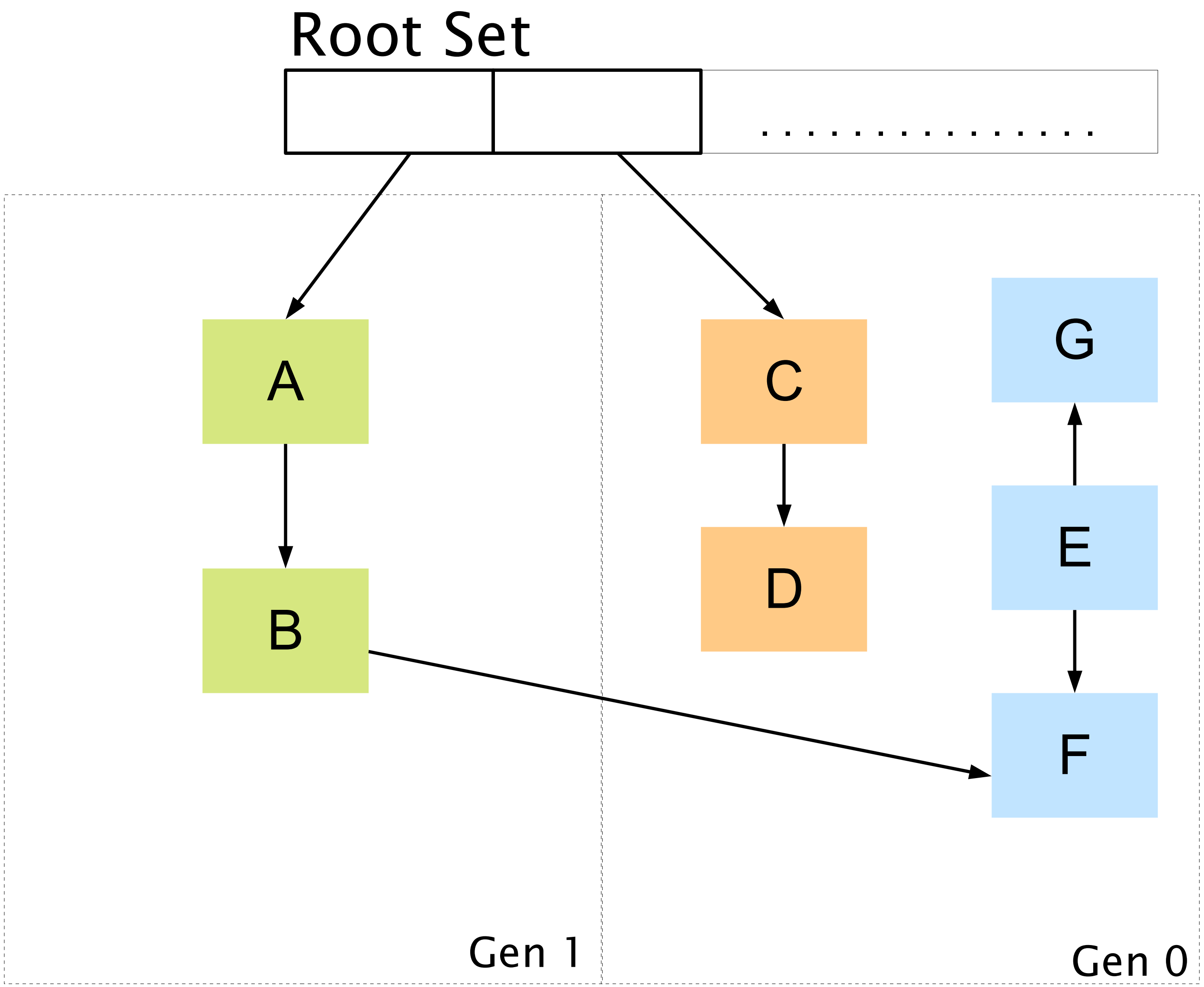
Generational Garbage Collection



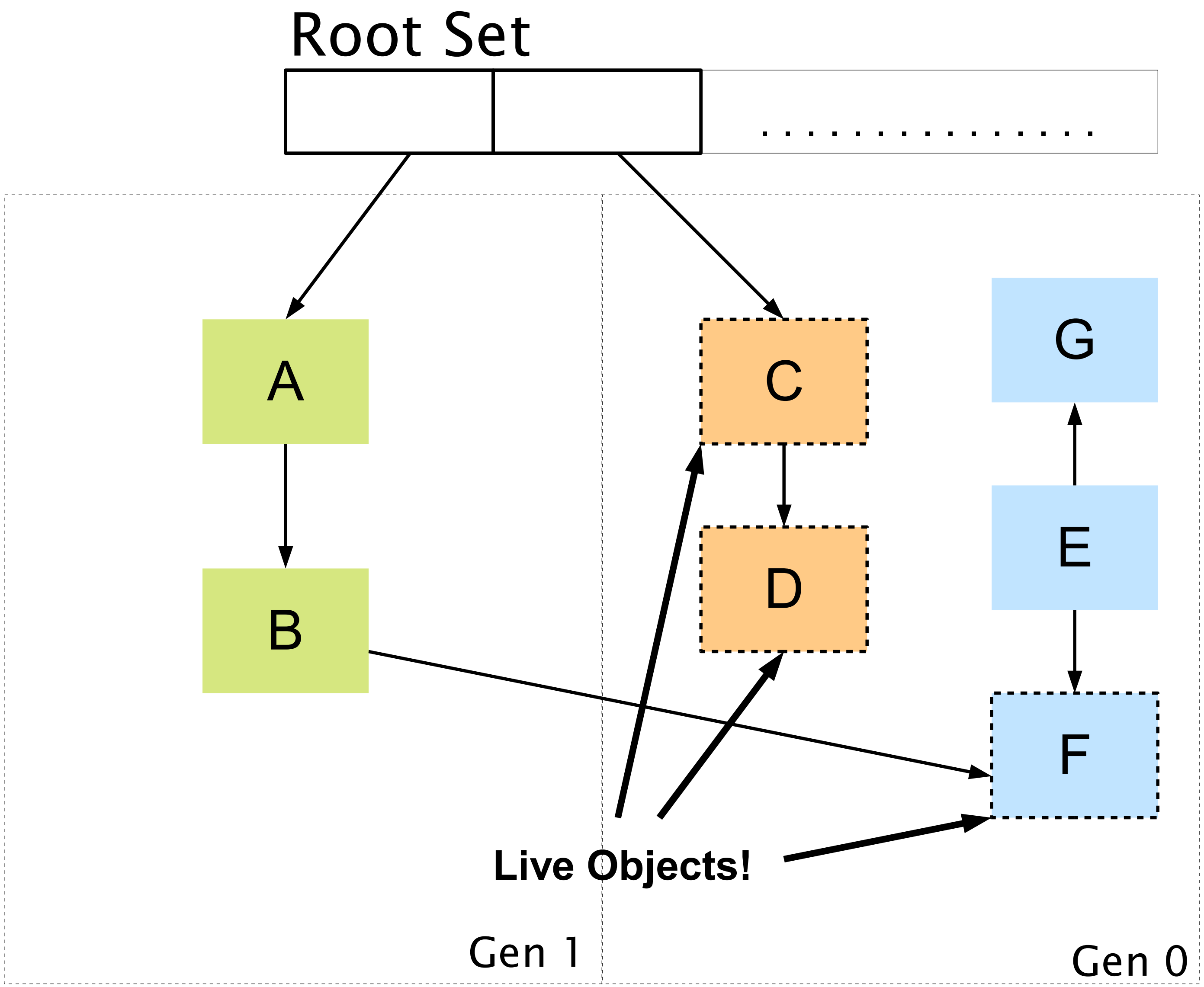
Generational Garbage Collection



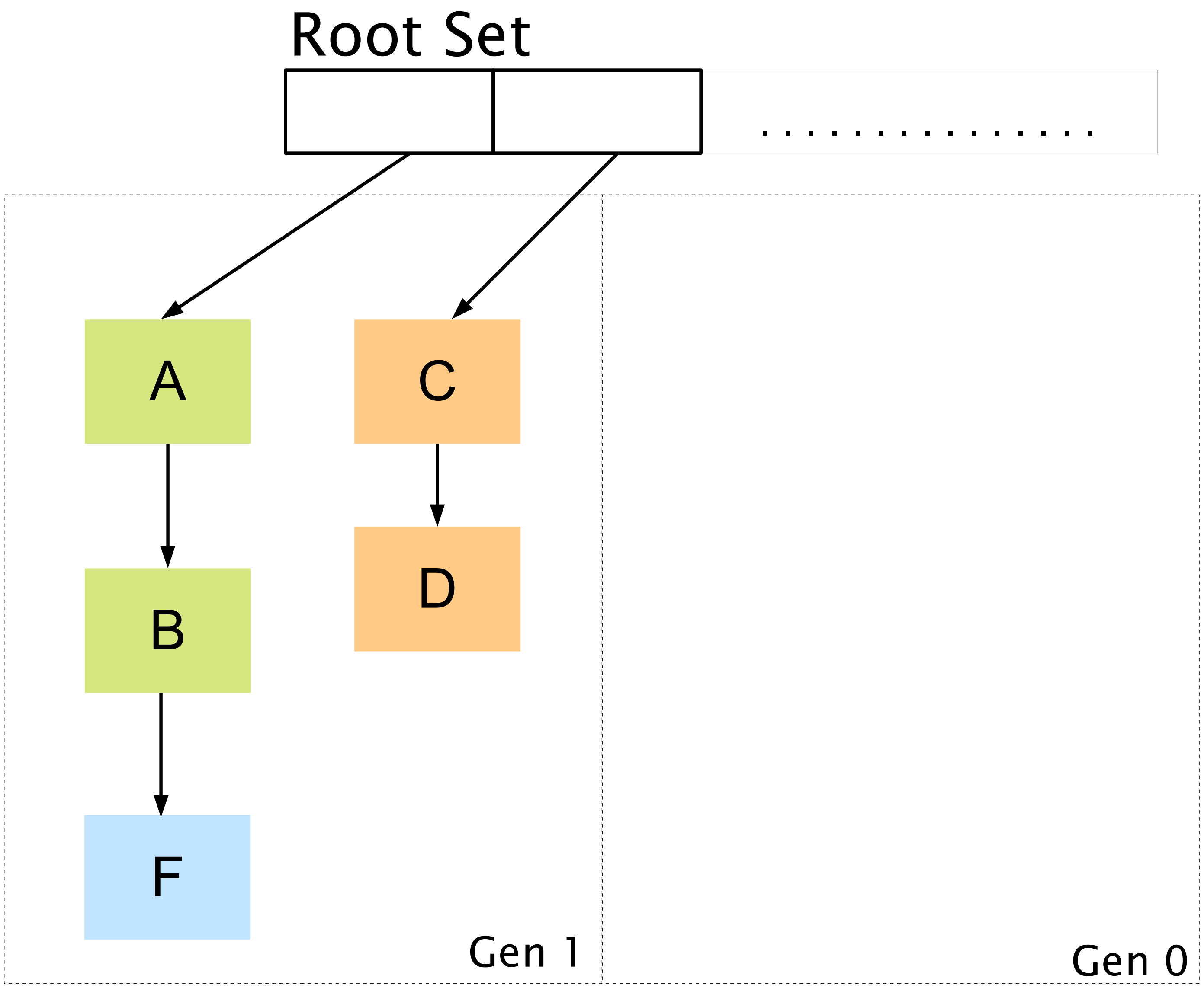
Generational Garbage Collection



Generational Garbage Collection



Generational Garbage Collection



Reference Counting

- Problem: How do we decide if an object is no longer referenced?

Reference Counting

- Problem: How do we decide if an object is no longer referenced?

- Solution: Keep track of number of *active* references to every object
  - As soon as reference hits 0, can immediately GC object
  - Tricky: How do we know an object is garbage, really?

- Used in: Python, Microsoft COM, Apple Cocoa, etc.
  - Many C hash table implementations use ref. counting to keep track of active entries

Summarizing GC

- Root set used to track live objects
- Possible strategies
  - Mark and sweep
  - Generational GC
  - Reference Count

- C, by default, does not support GC
- Now, real world implementations of malloc

GNU C Library (glibc) malloc

- Original implementation (dlmalloc)
- Current implementation (ptmalloc2)
  - Based on dlmalloc
  - Used in glibc 2.4 for linux based systems
  - Supports multiple heaps
    - But we won't worry about that for now

Doug Lea Allocator (dlmalloc)

- Fast and efficient implementation

- Heap allocated using sbrk()
- Memory allocated as “chunks”

- Uses a best-fit strategy
- Coalesce chunks during free
    - Reduces fragmentation
- Segregated lists (binning) to find free chunks quickly

Chunks

- Stored in bin
  - doubly-linked list
- Easy to coalesce adjacent chunks
  - Size of current and previous chunks stored in chunk header
- Minimum chunk size is 16 bytes

Size of previous chunk
Forward pointer to next chunk in the list
Backward pointer to previous chunk in the list
Unused Space (≥ 0 Bytes)
Size of chunk

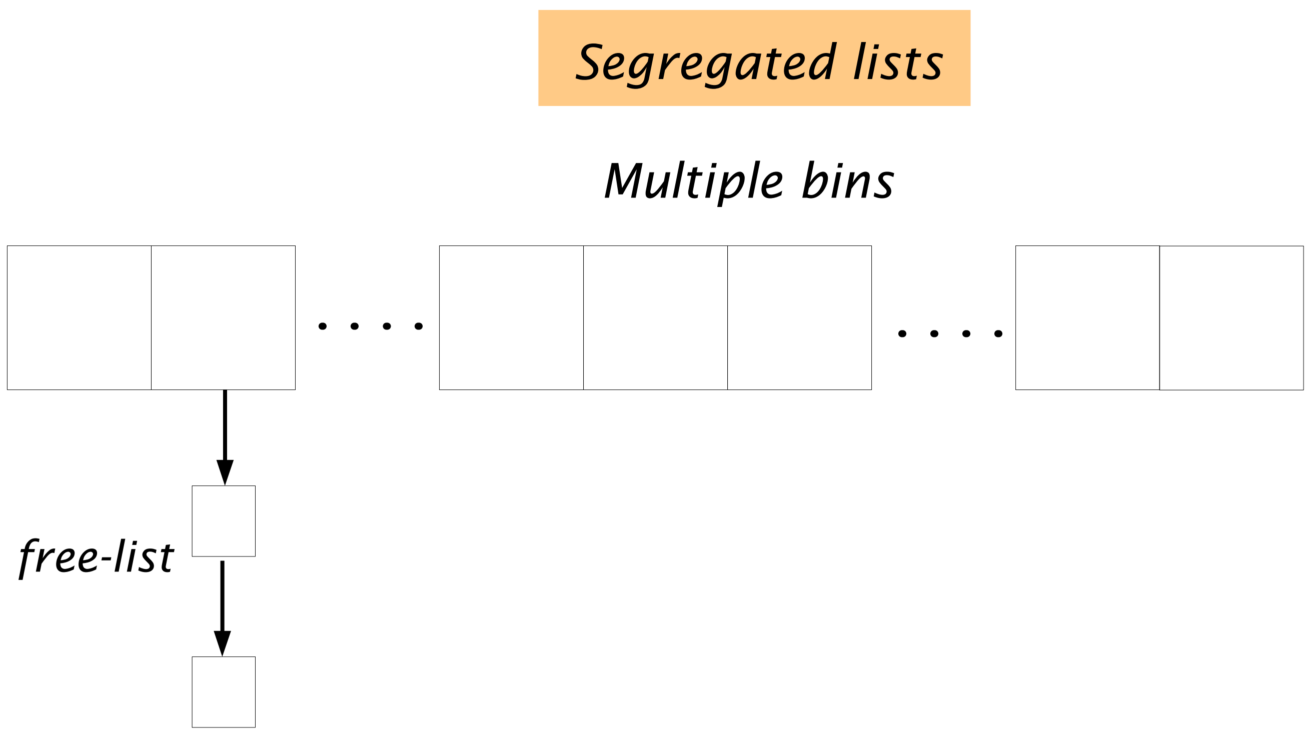
Free Chunk Format

Designing *dlmalloc*

- Goal: Find best-fitting *free* chunk list

Designing *dlmalloc*

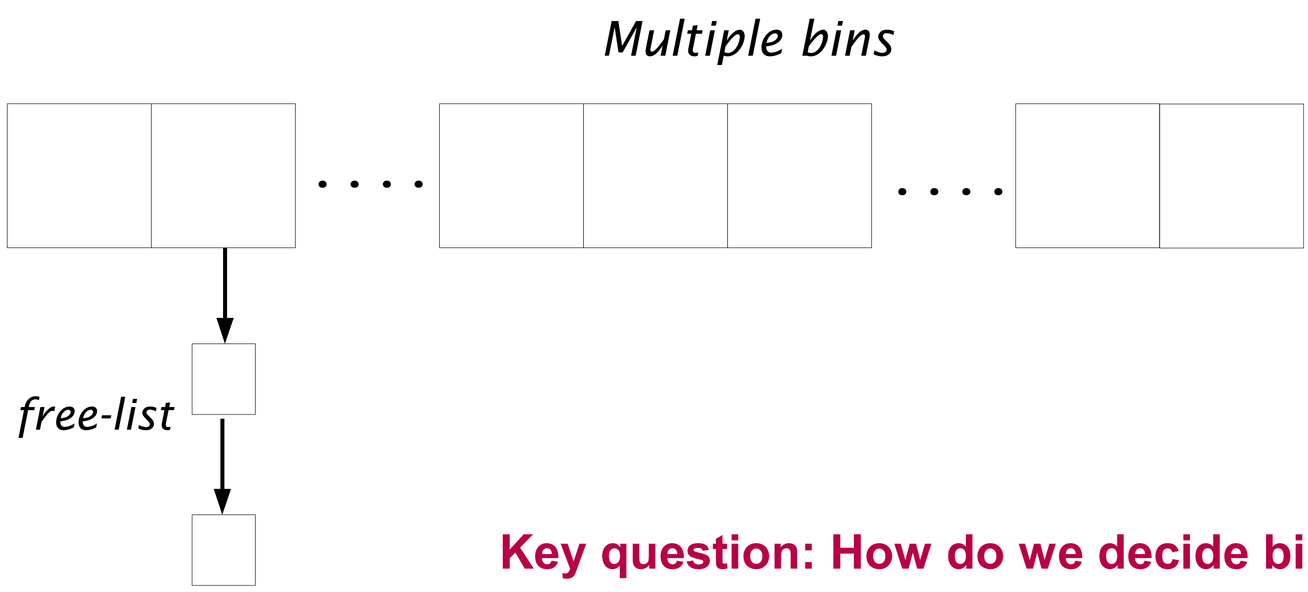
- Goal: Find best-fitting *free* chunk list



Designing *dlmalloc*

- Goal: Find best-fitting *free* chunk list
- *Observation: We receive many small (< 256 bytes) requests*

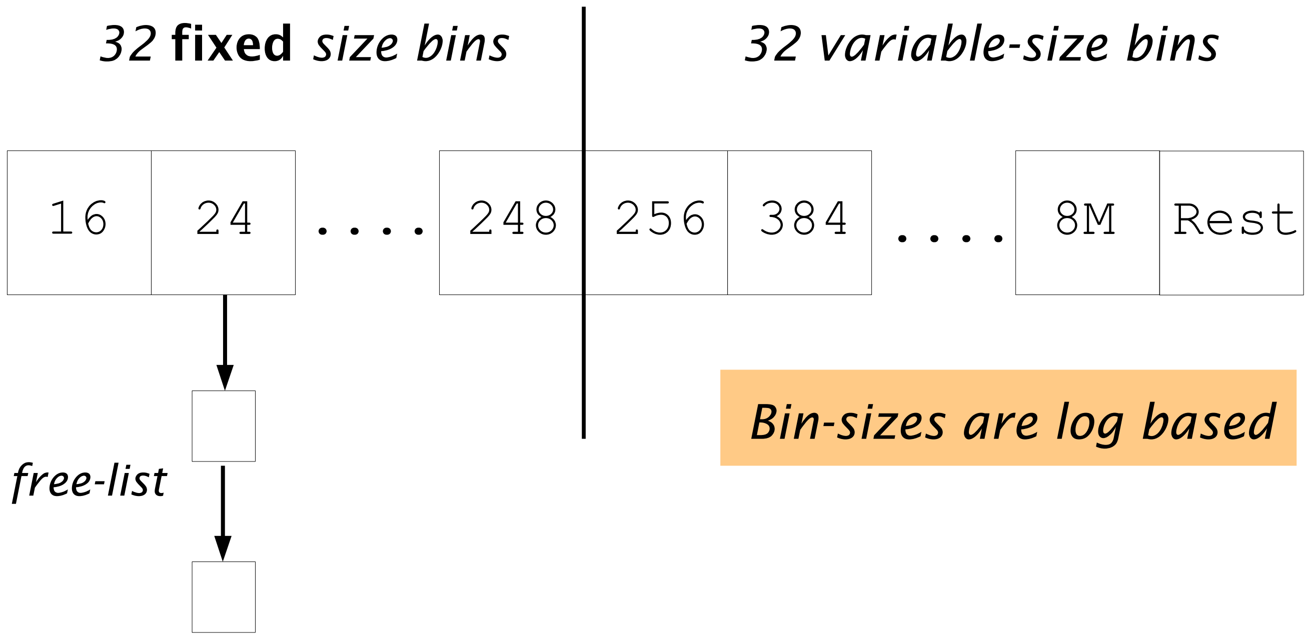
Segregated lists



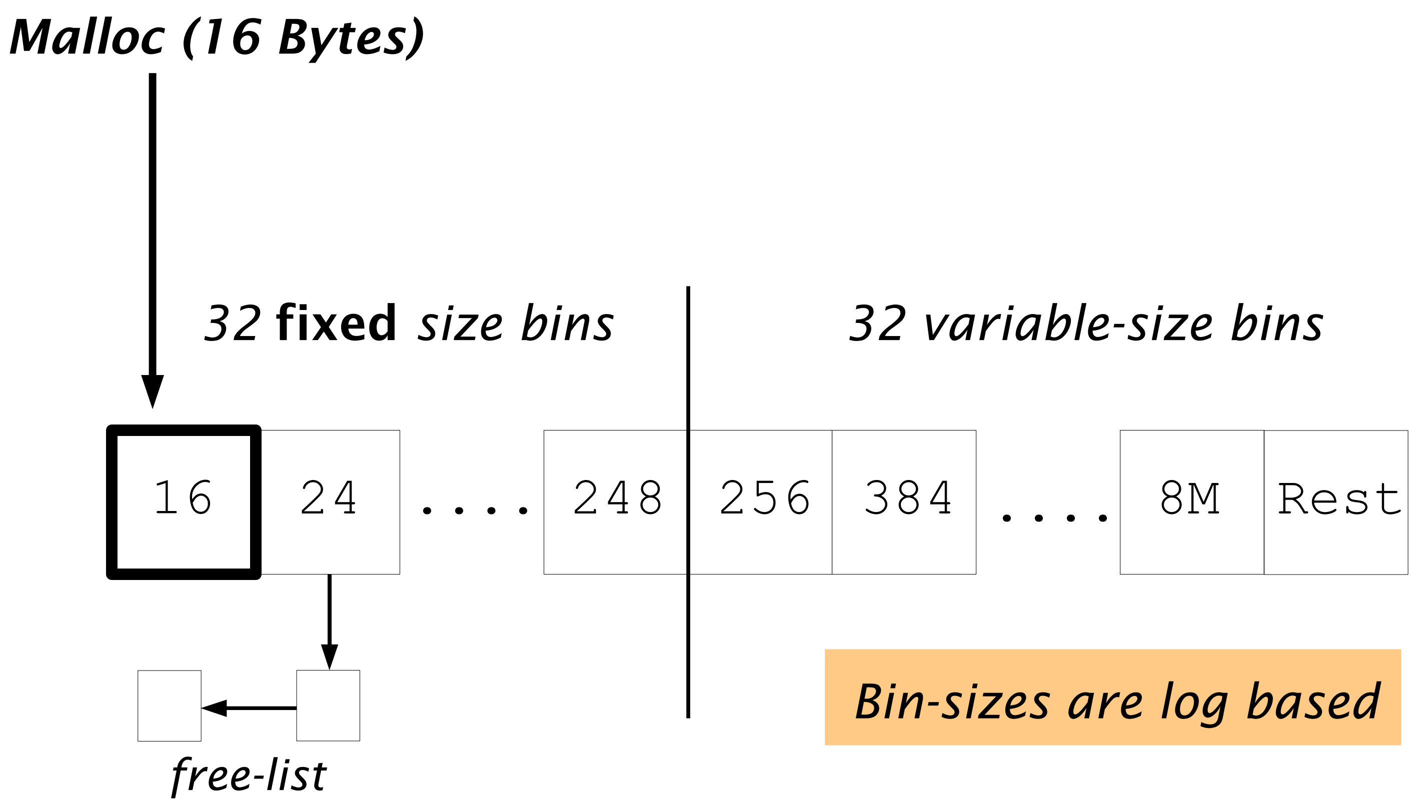
Key question: How do we decide bin-sizes?

Designing *dlmalloc*

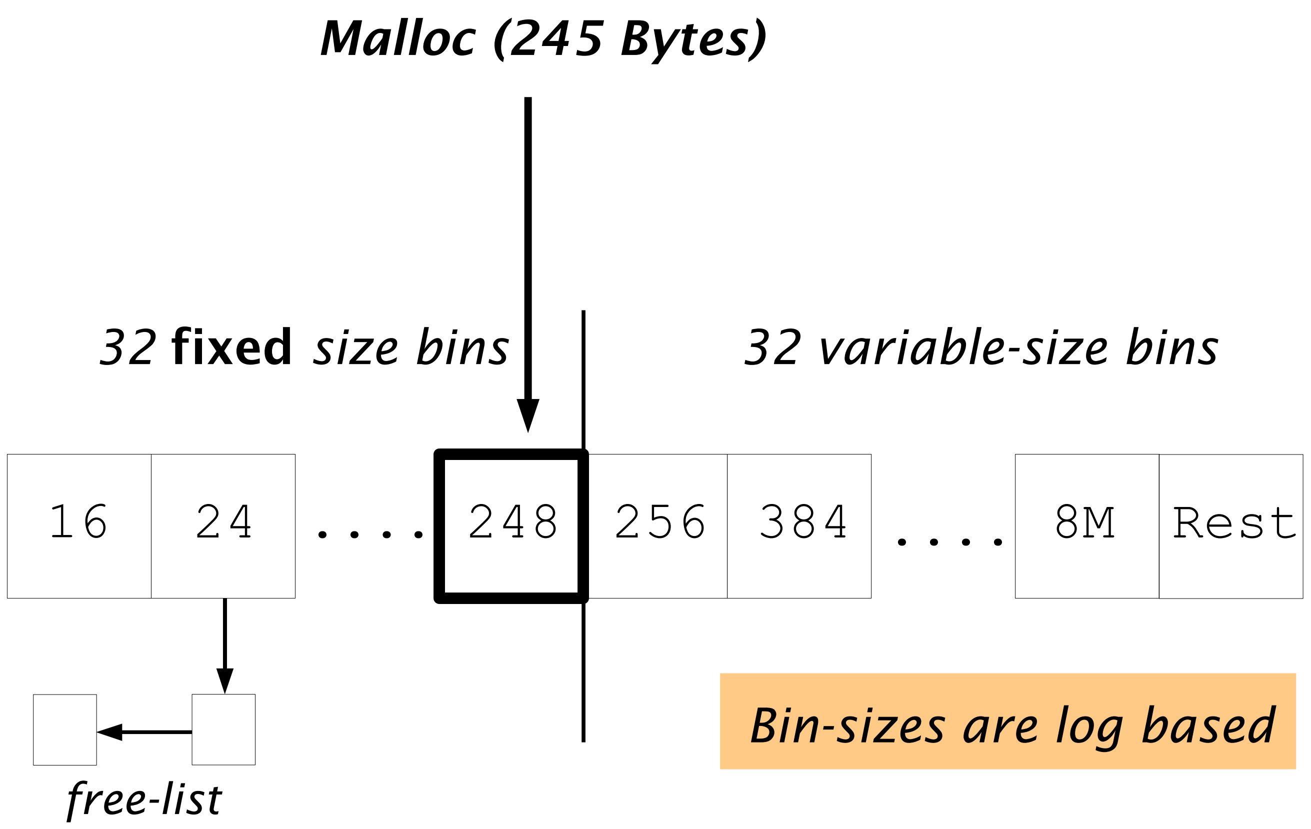
- Goal: Find best-fitting *free* chunk list
- *Observation: We receive many small (< 256 bytes) requests*



Designing *dlmalloc*

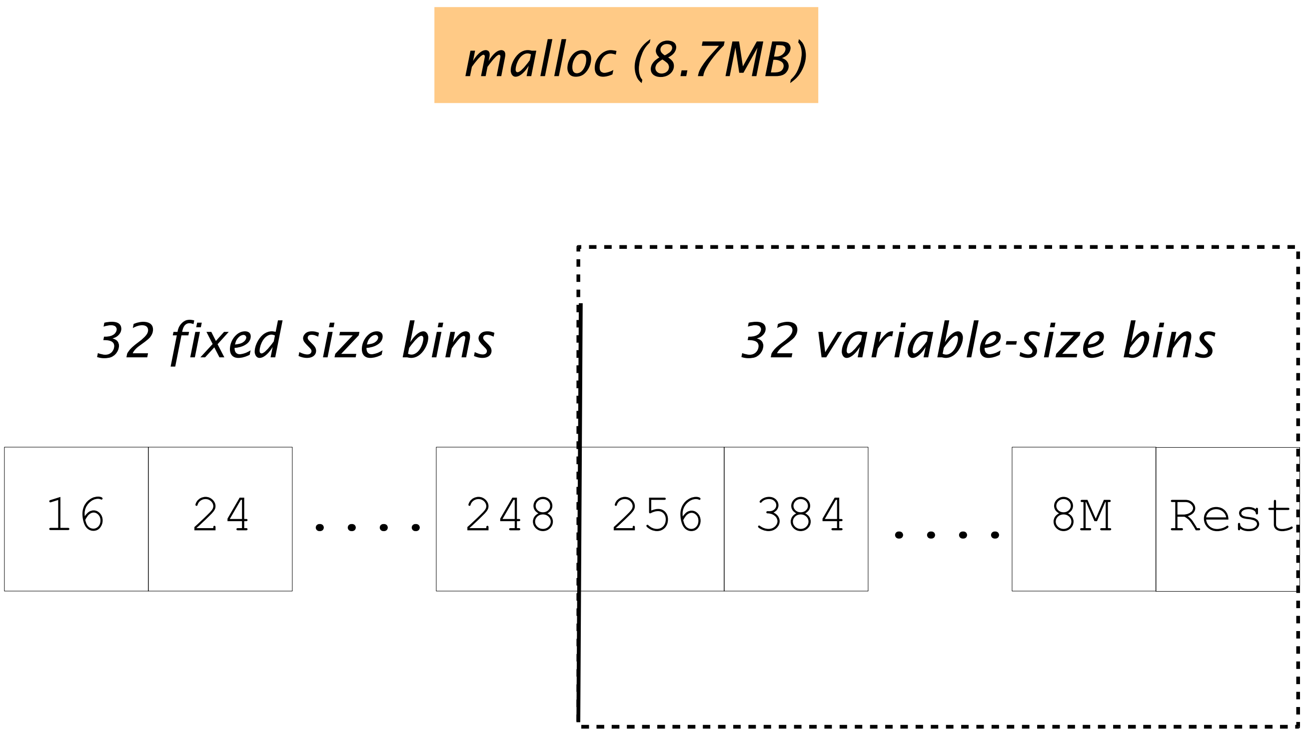


Designing *dlmalloc*



dlmalloc: Servicing Large Requests

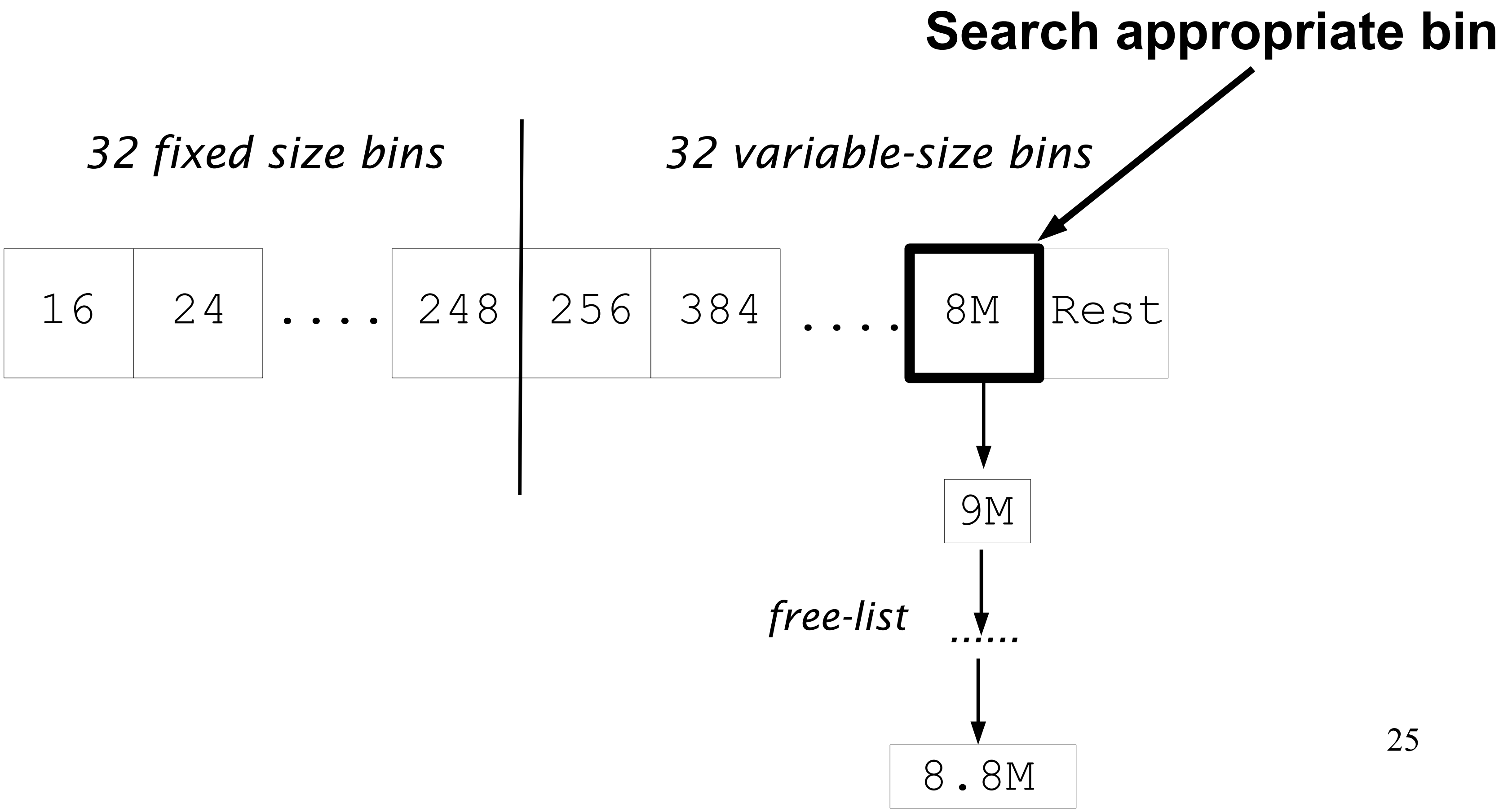
- Search variable-sized bins



dlmalloc: Servicing Large Requests

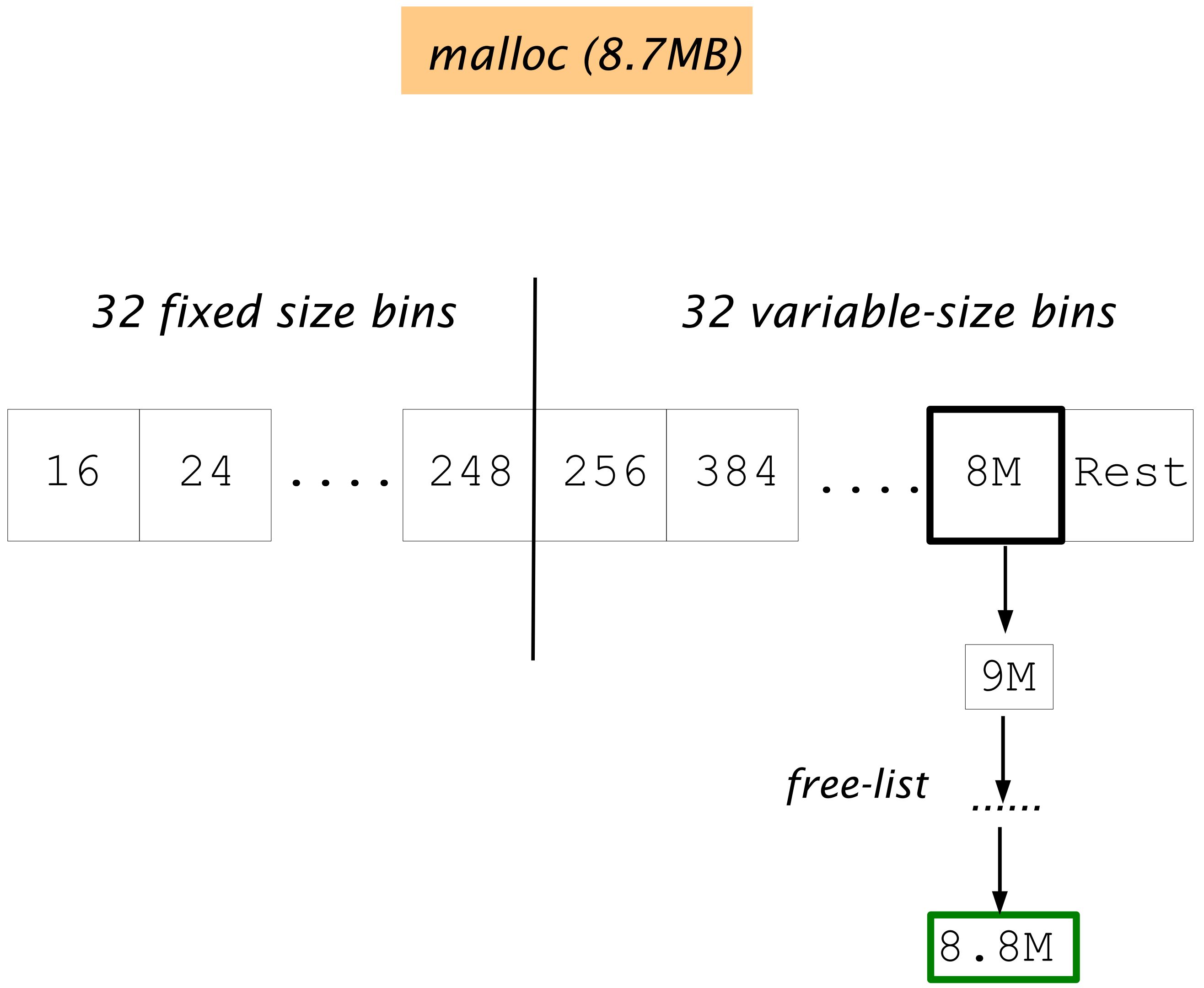
- Search variable-sized bins

malloc (8.7MB)



dlmalloc: Servicing Large Requests

- Search variable-sized bins



dlmalloc: Optimizing for Large Requests

- For a given bin,
  - Linear search to find chunk size that fits
    - Potentially expensive

We want to find best-fit chunk!

- Can we optimize the search for a chunk?
  - Sort free-list based on chunk sizes?
    - Sorting takes time
    - Impacts performance

- Can we optimize this further?

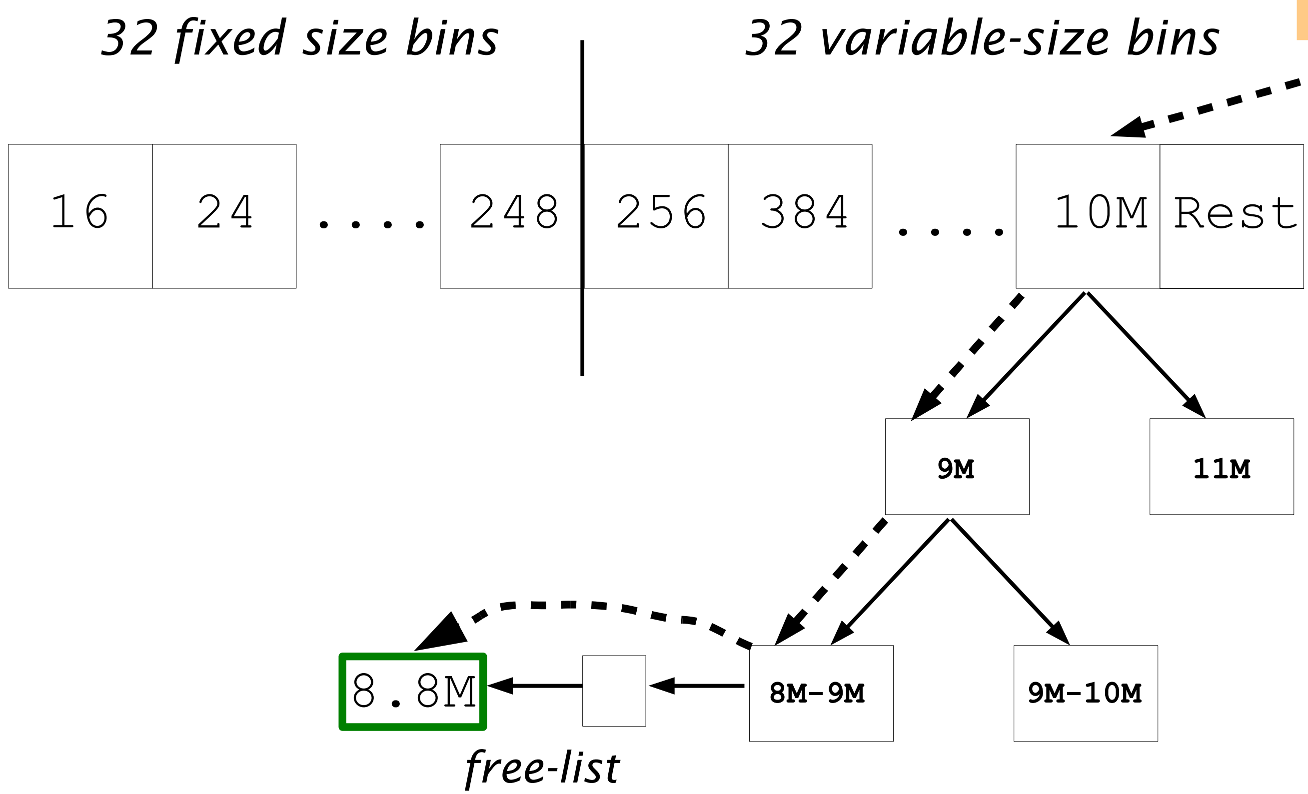
dlmalloc: Optimizing for Large Requests

- Search variable-sized bins
  - Stores chunks as binary trees

Avg. Search: O(log n)

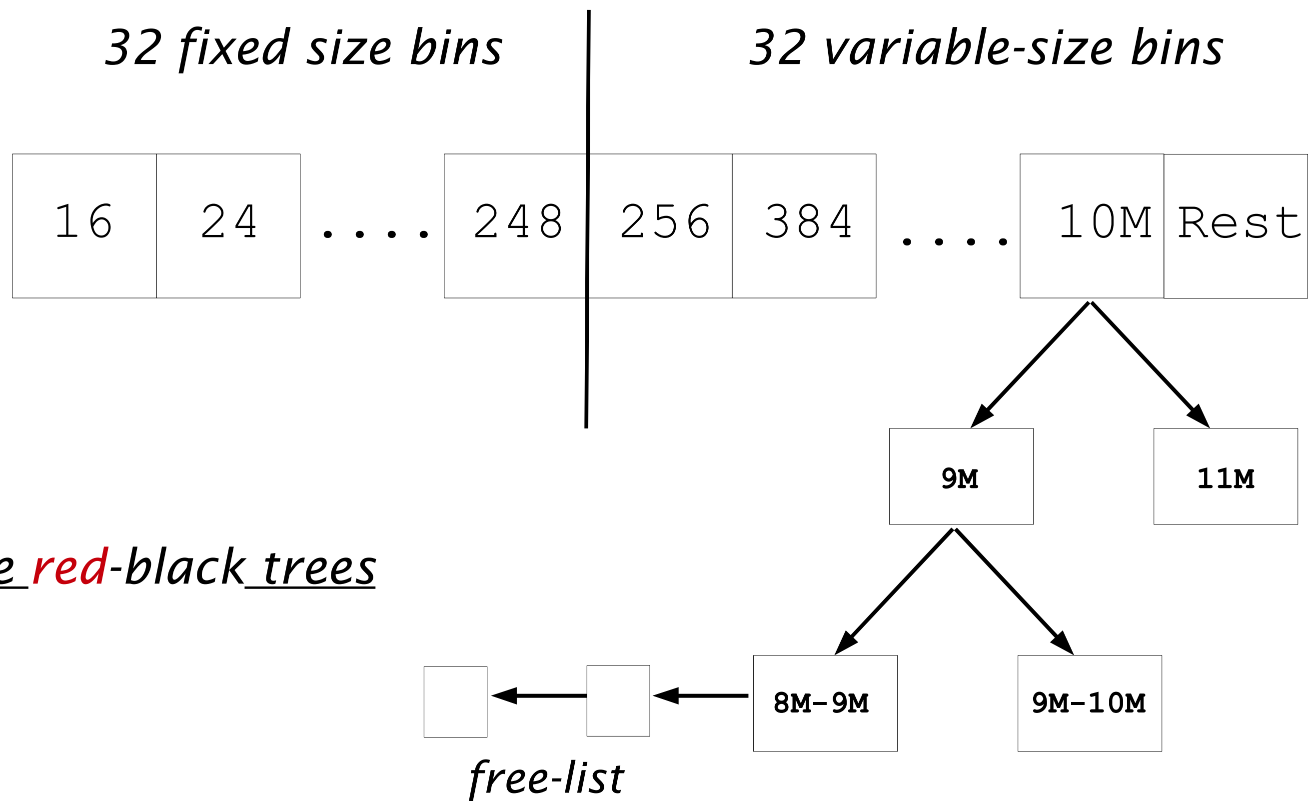
Avg. Insertion: O(log n)

malloc (8.7MB)



dlmalloc: Optimizing for Large Requests

- Search variable-sized bins
  - Stores chunks as binary trees
    - Each node stores list of chunks of *same* size



Alternative: use red-black trees

dlmalloc: Freeing Chunks

- Coalesce chunk with surrounding chunks
- Re-insert coalesced chunk into appropriate free list
- Fast coalescing because of 'boundary tags'
  - Size of chunk at the beginning
  - Size of chunk at the end

dlmalloc: Optimizing Performance

- Introducing quicklists/fastbins (< 64 bytes)
  - Fixed size bins for very small requests

- Treat different size chunks differently
  - Small chunks: do not coalesce, return to free list
  - Medium chunks: occasionally formed by coalescing objects from quicklists (optimistic strategy)
    - When free()-ed, performs immediate coalescing and occasionally splits to
      - Find best-fit
      - Allocate more space to quick lists
  - Large chunks: Coalescing is sometimes deferred

Summarizing dlmalloc

- Bins to store free lists (of chunks)
- Divide bins into two sets
  - First set, for small requests (occur often)
  - Second set, for larger requests (spaced at log. Intervals)
    - Binary tree used to quickly insert and search chunks in second set
- Chunks store boundary tags
  - Coalesced during free
- Successive refinements performed over the years
  - Foundation for various malloc implementations
    - ptmalloc2, a variant of dlmalloc used in glibc 2.4

But how is malloc implemented for the rest of the world (80%+)?

Windows XP Allocator

- Best-fit search
- 127 exact-size quicklists
  - For multiple of 8 bytes
- Objects > 1024 bytes
  - Obtained from a sorted list
  - Sacrifice speed for a good fit

Buddy (Memory) Allocation

- Described by Don Knuth
  - First proposed by economist Harry Markowitz
- Very fast and simple for 2^n size blocks
- Idea: *split* memory into *halves* until best-fit is found
  - When free-ing
    - Combine adjacent empty halves

Buddy (Memory) Allocation: Example

Lower bound for block size 5 → 2^4 = 16K (*minimum* block size)
Upper bound for block size 9 → 2^9 = 512K (**maximum** block size)

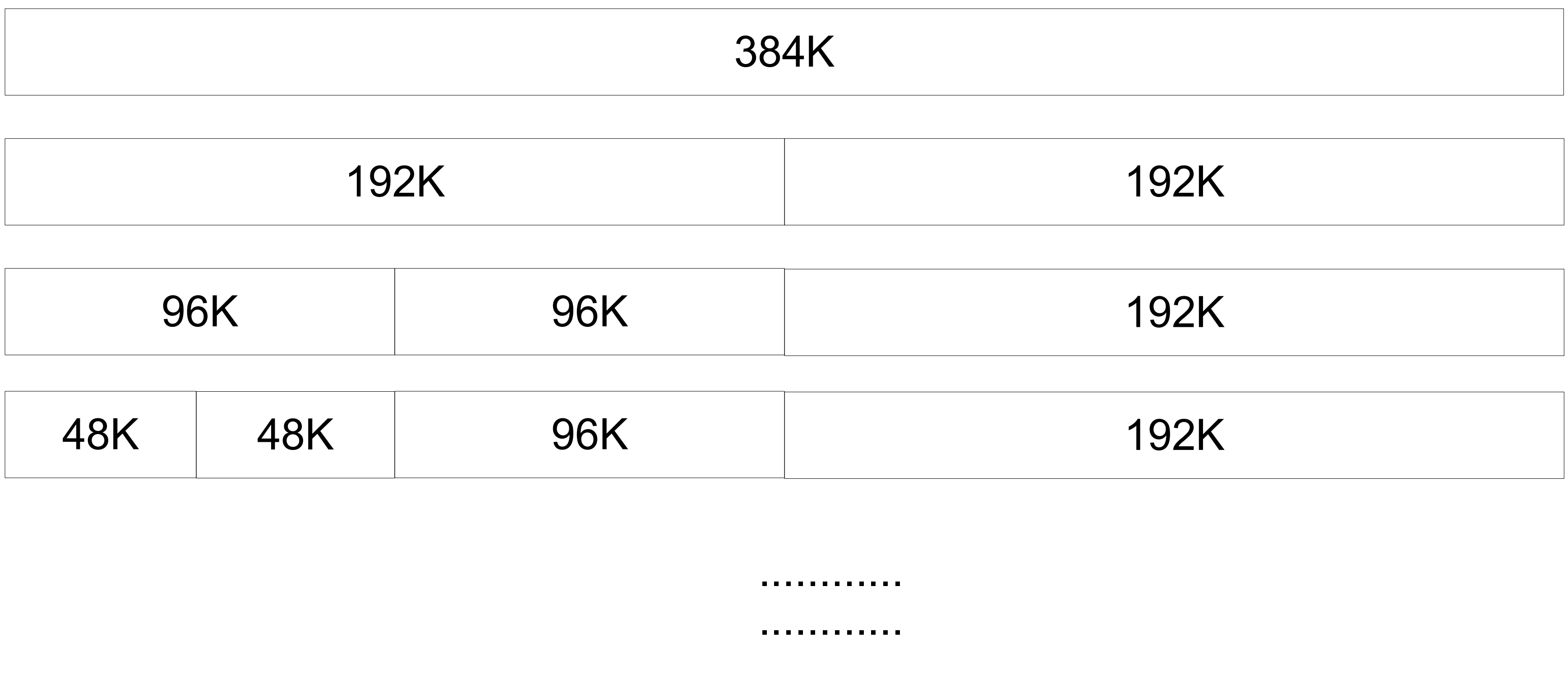
	32K	32K	32K	32K	32K	32K	32K	32K	32K	32K	32K	32K
Memory	384K											

malloc (20K)

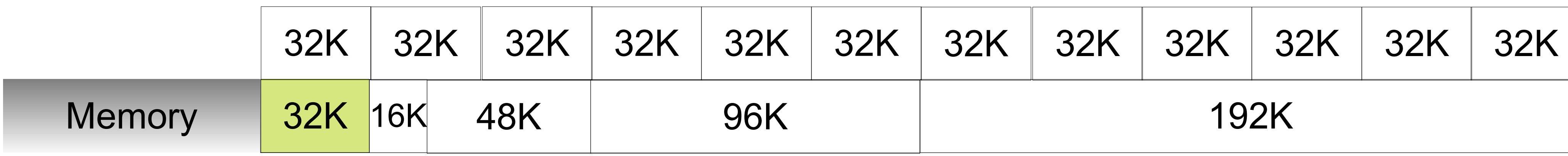
Simple Strategy: Allocate the entire 384K → results in *severe* internal fragmentation

Idea: ***Split*** memory until we get *smallest* (permissible) block that satisfies the request

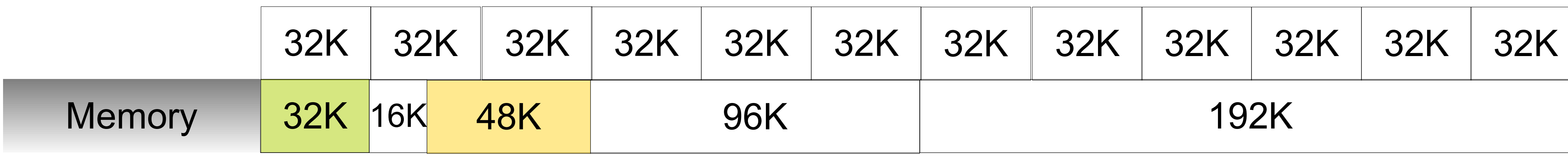
Buddy Allocation: Splitting Memory



Buddy (Memory) Allocation: Example



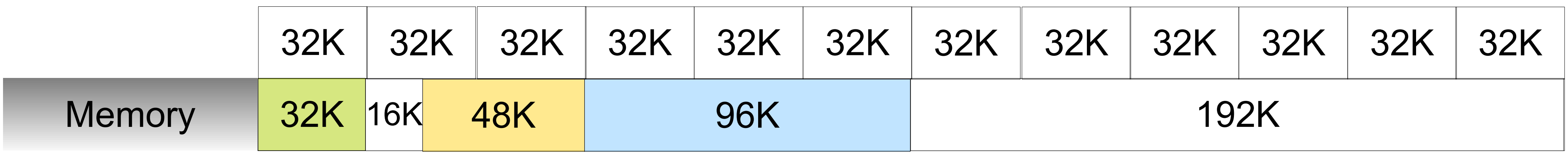
Buddy (Memory) Allocation: Example



malloc (38K)

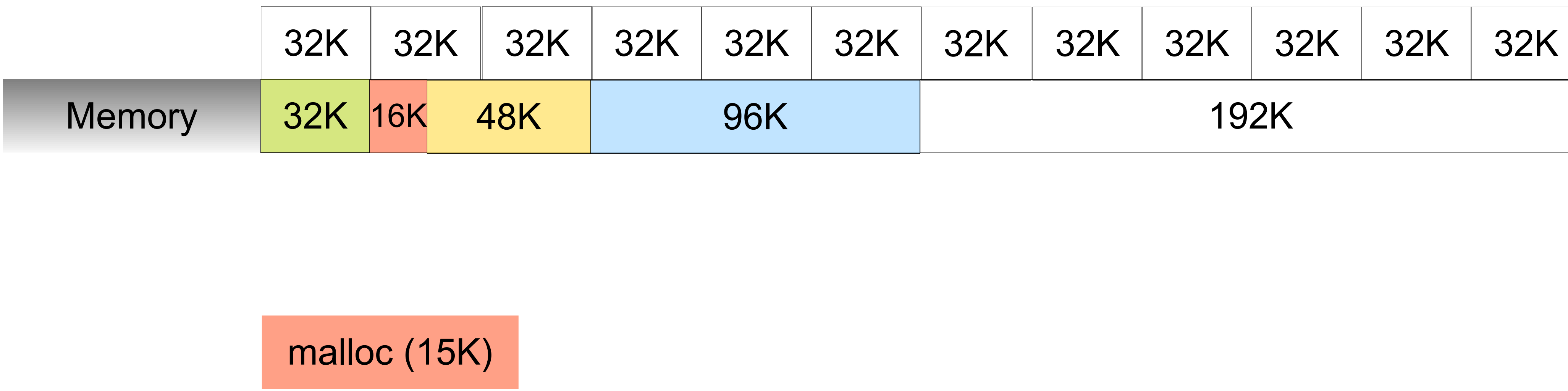
No need to split any blocks. 48K block will suffice!

Buddy (Memory) Allocation: Example

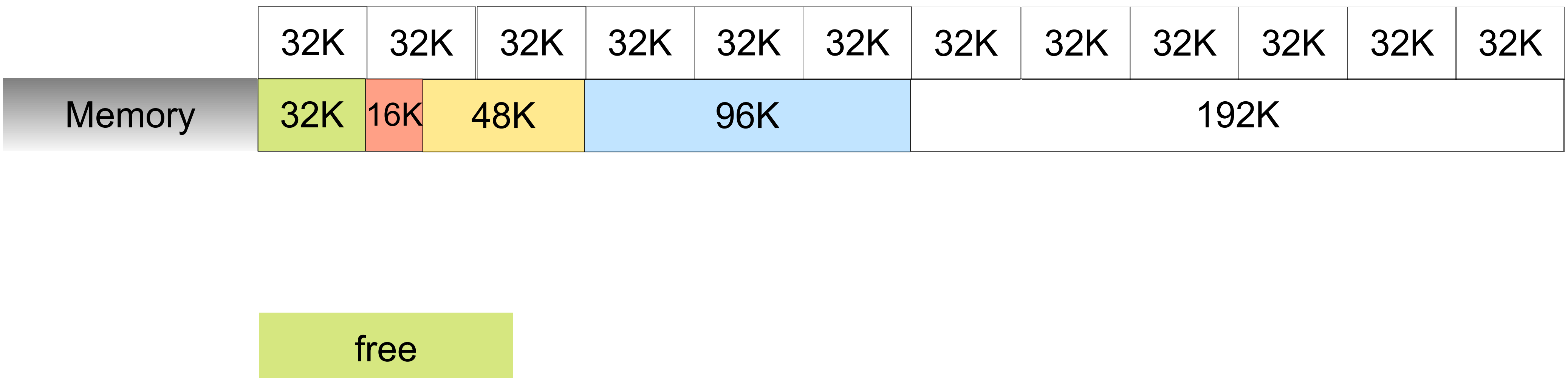


malloc (89K)

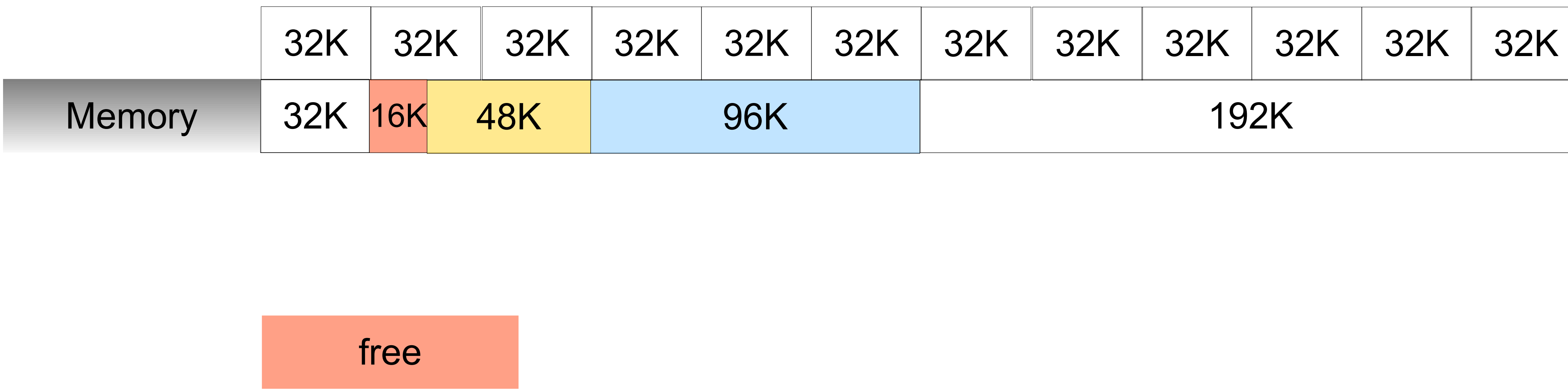
Buddy (Memory) Allocation: Example



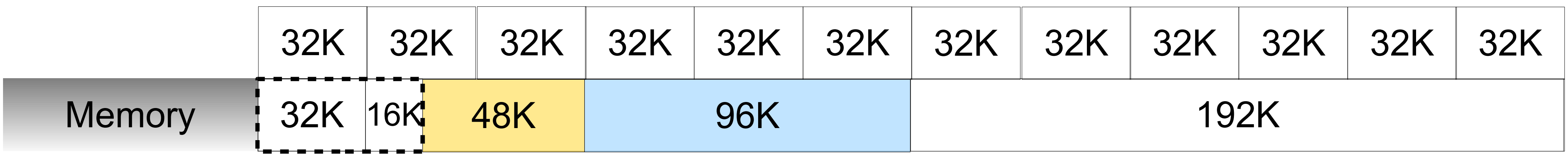
Buddy (Memory) Allocation: Example



Buddy (Memory) Allocation: Example

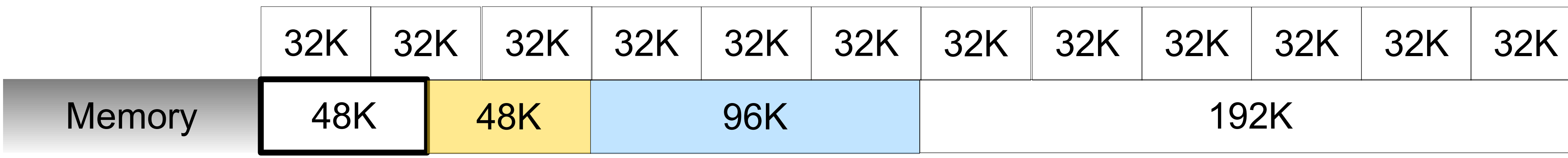


Buddy (Memory) Allocation: Example



Opportunity to coalesce to adjacent free blocks (32K and 16K)

Buddy (Memory) Allocation: Example



Buddy Memory Allocation

- Pros:
  - Easy to implement (binary tree)
  - Little external fragmentation
    - Why??
- Cons:
  - Internal fragmentation
    - Why??

Not all requests will be on the order of 2^k → More than requested memory allocated

Bounds on block sizes impact internal fragmentation

What constitutes a *good* malloc impl.?

- Time
  - Minimize time taken for allocation and free
- Space
  - Minimize fragmentation
- Locality
  - Allocations around time t are stored near each other
  - Similar sized blocks are stored near by

Performance: Throughput

- Given a sequence of requests

  malloc(...) , malloc (…), free (…), malloc(…) ....

- Measure: Throughput
  - Number of requests satisfied per unit time

- Goal: Maximize throughput

Performance: Peak Utilization

- Given a sequence of requests

  malloc(...) , malloc (…), free (…), malloc(…) ....

- Measure: Peak Utilization
  - Maximum currently allocated memory / Current Heap Size

- Goal: Maximize peak utilization

Overall Performance Goal

- Maximize throughput *and* peak utilization

- Conflict with each other. Why?
  - Throughput: Minimize time taken to allocate
  - Peak Utilization: Minimize waste of space
    - Need more time to do this efficiently
    - Conflicts with the Throughput goal

- Find sweet spot between
  - maximizing throughput and minimizing peak utilization

SPEC Benchmarks

- Popular tool used to benchmark system performance
  - Returns a single performance metric: *time*
- Suite of applications to stress test a system
  - Mesh generation, Monte carlo, circuit simulation and analysis, fluid dynamics applications, compression, neural networks, etc.
- Manipulates large and small arrays in loops
  - Sensitive to performance of memory allocation
  - Good stress test for malloc

References

- Lecture notes: *Sebastian Hack* and *Christoph Weidenbach (MPI)*
- “Understanding the heap by breaking it” - *Justin Ferguson*
- “Back to basics: Generational GC” - *Abhinaba, Microsoft Crop*
- “A Locality Improving Dynamic Memory Allocator” - *Yi Eng and Emery Berger, MSP '05*
- “Reconsidering Custom Memory Allocation” - *Berger, Zorn, and McKinley, OOPSLA 2002*
- Wikipedia