ECE 454
Computer Systems Programming

Modern Allocators, Garbage Collectors

Jon Eyolfson
Courtesy: Ashvin Goel
ECE Dept, University of Toronto

Contents

- Introduction to dynamic memory management
  - Alignment
  - Memory management API
  - Constraints, goals
  - Fragmentation
- Basic dynamic memory allocation
  - Implicit free list
  - Explicit free list
  - Segregated free lists / Buddy allocation
- Modern allocators
- Garbage collectors
- Application-Aware allocators

Case Study 1: phkmalloc

- Poul-Henning Kamp, The FreeBSD project, "Malloc(3) revisited", Usenix ATC 1998
  - Please read for details
- Argues that key performance goal of `malloc` should be to minimize the number of pages accessed
  - Helps improve locality, specially when the working set of programs is large, close to available physical memory

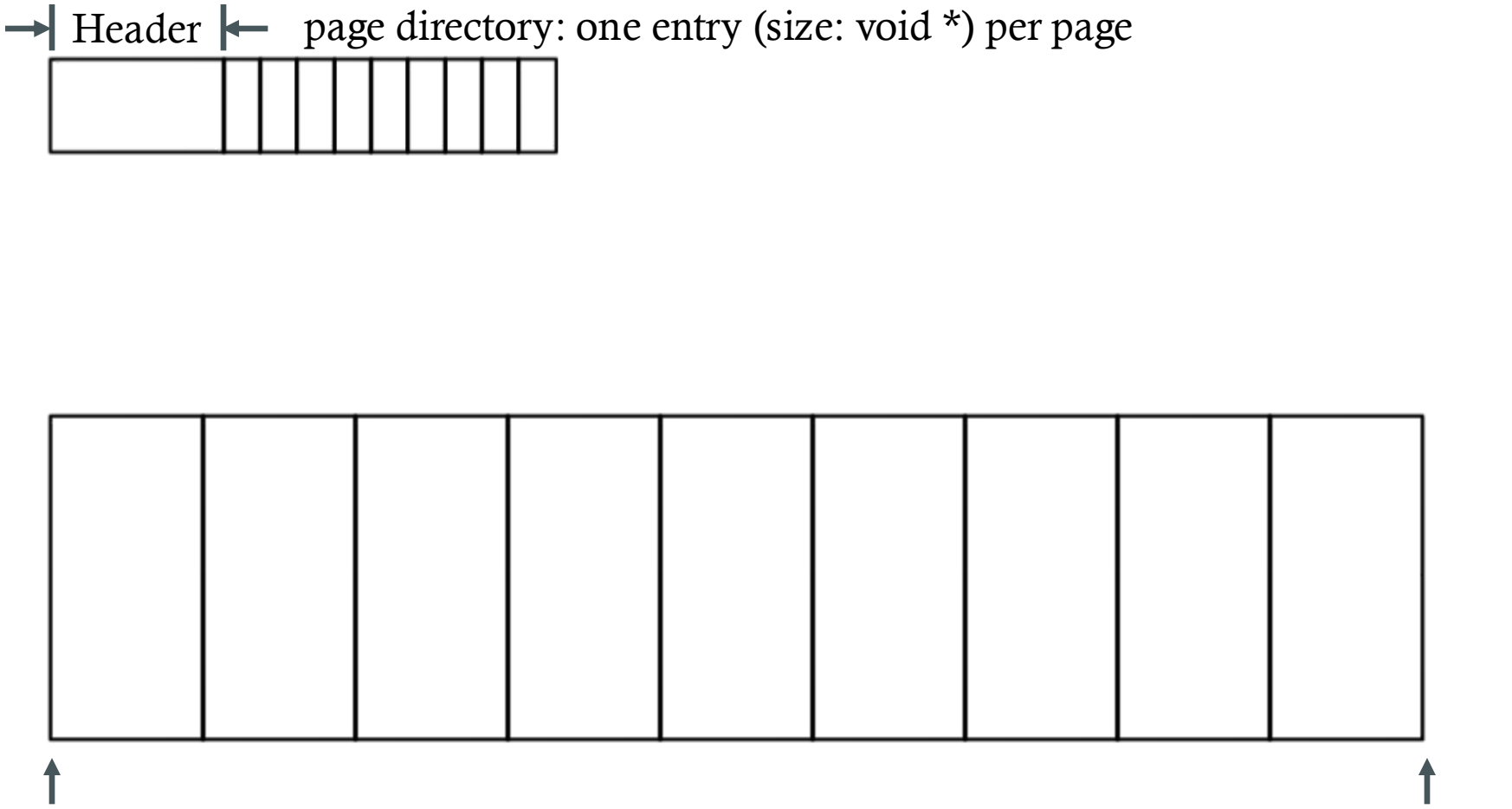
Problem with Existing Malloc

- Information about free chunks is kept in free chunks
  - Even if application doesn't need this memory, malloc needs it
  - Even if OS pages out the free chunk pages, malloc will page them in when it traverses the free list
- Malloc allocations do not work in pages
  - Allocations may span pages even when object size < page size
  - Such objects may require paging in two pages and two TLB entries

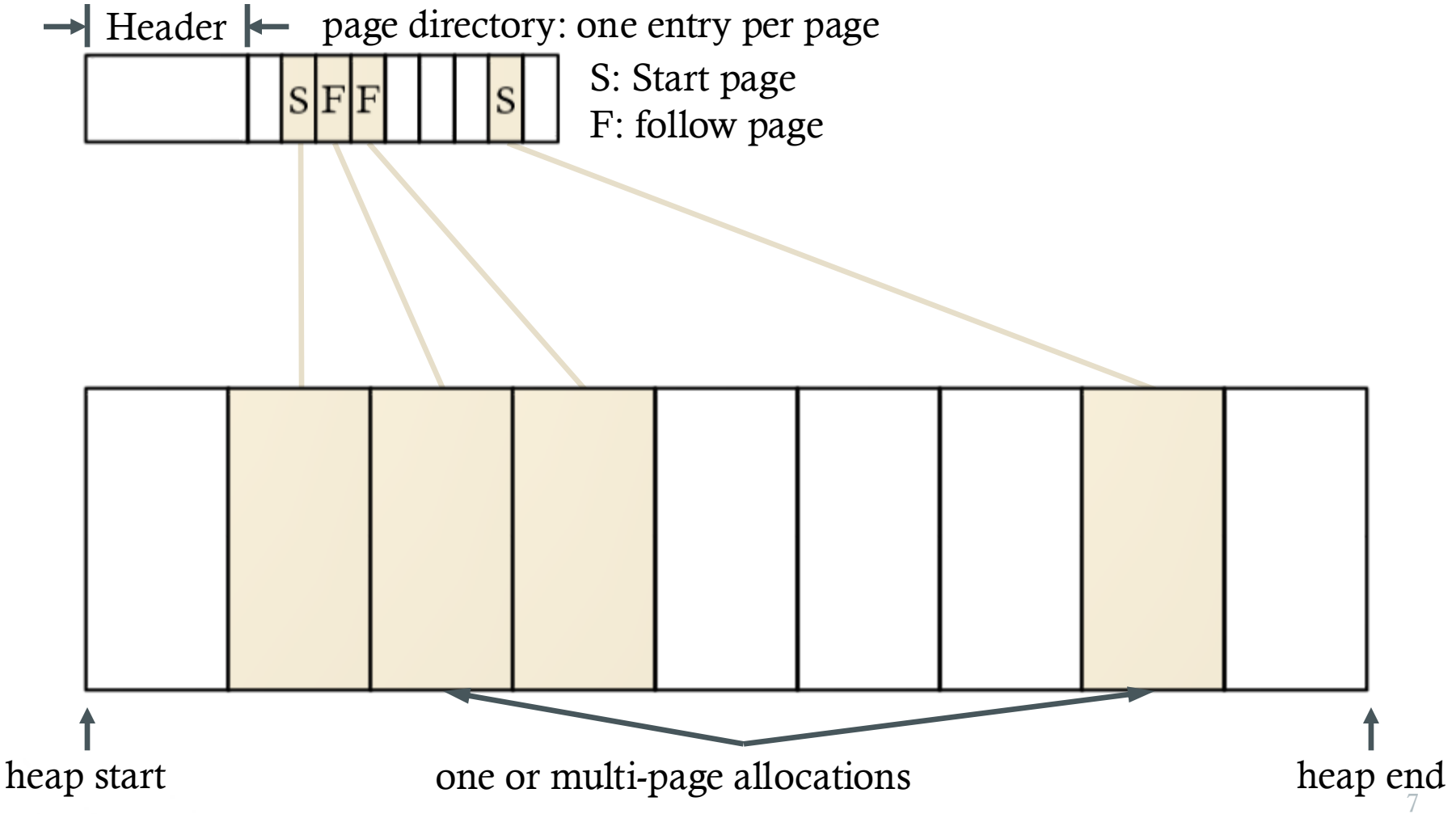
Phkmalloc Design

- phkmalloc ensures that malloc metadata is kept separately
  - Use separate mmap area for contiguous malloc metadata
  - Use malloc itself for dynamically allocated malloc metadata!
- Uses a two-level allocator
  - Large allocations performed at page granularity
    - For allocations greater than half page size
  - Small allocations performed at sub-page granularity within page
    - For allocations smaller than half page size
- Similar to simple segregated storage
  - All allocation requests are rounded to next power of 2
  - Keeps several lists of different fixed sizes

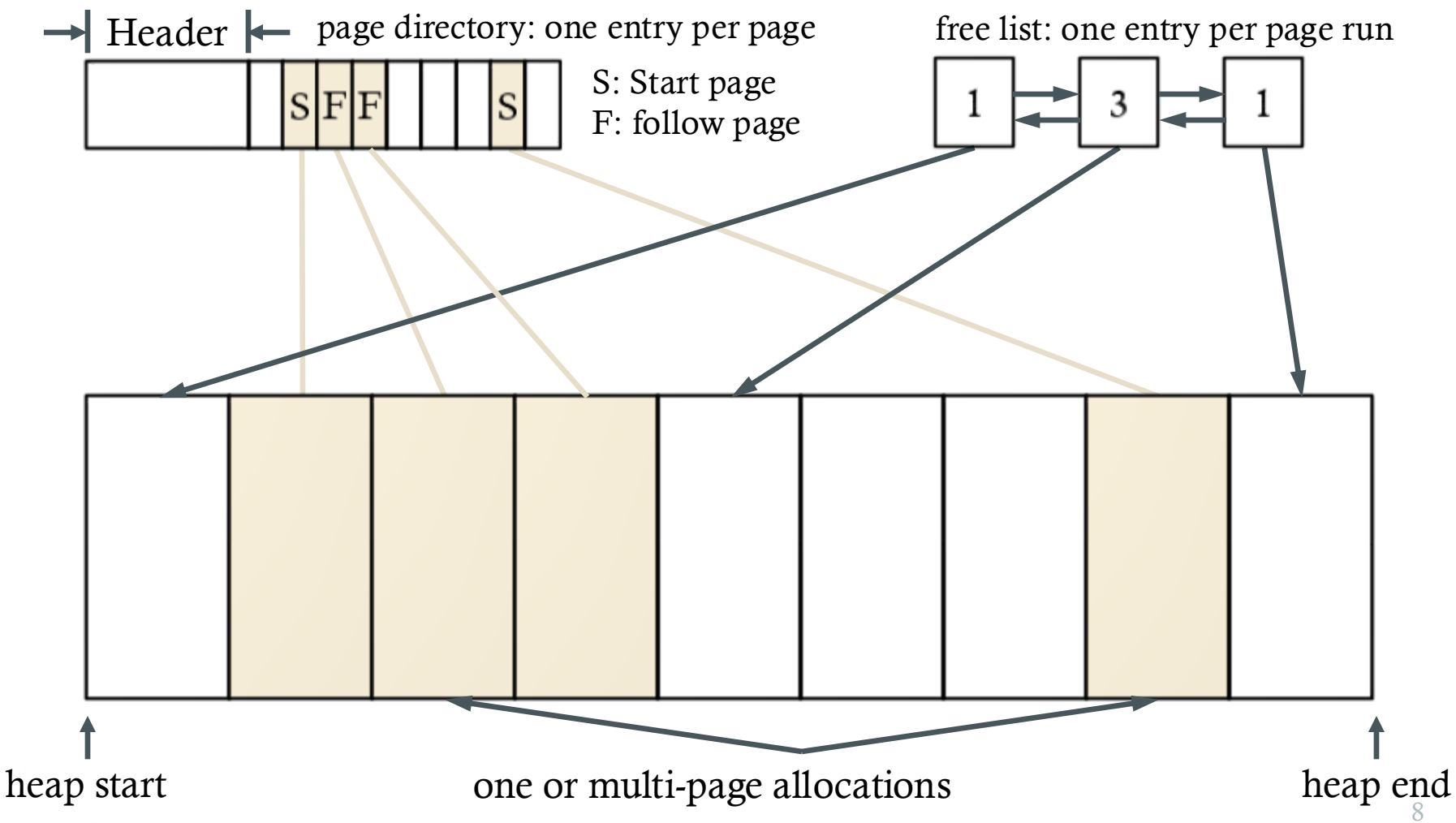
Phkmalloc Page Allocator



Phkmalloc Page Allocator



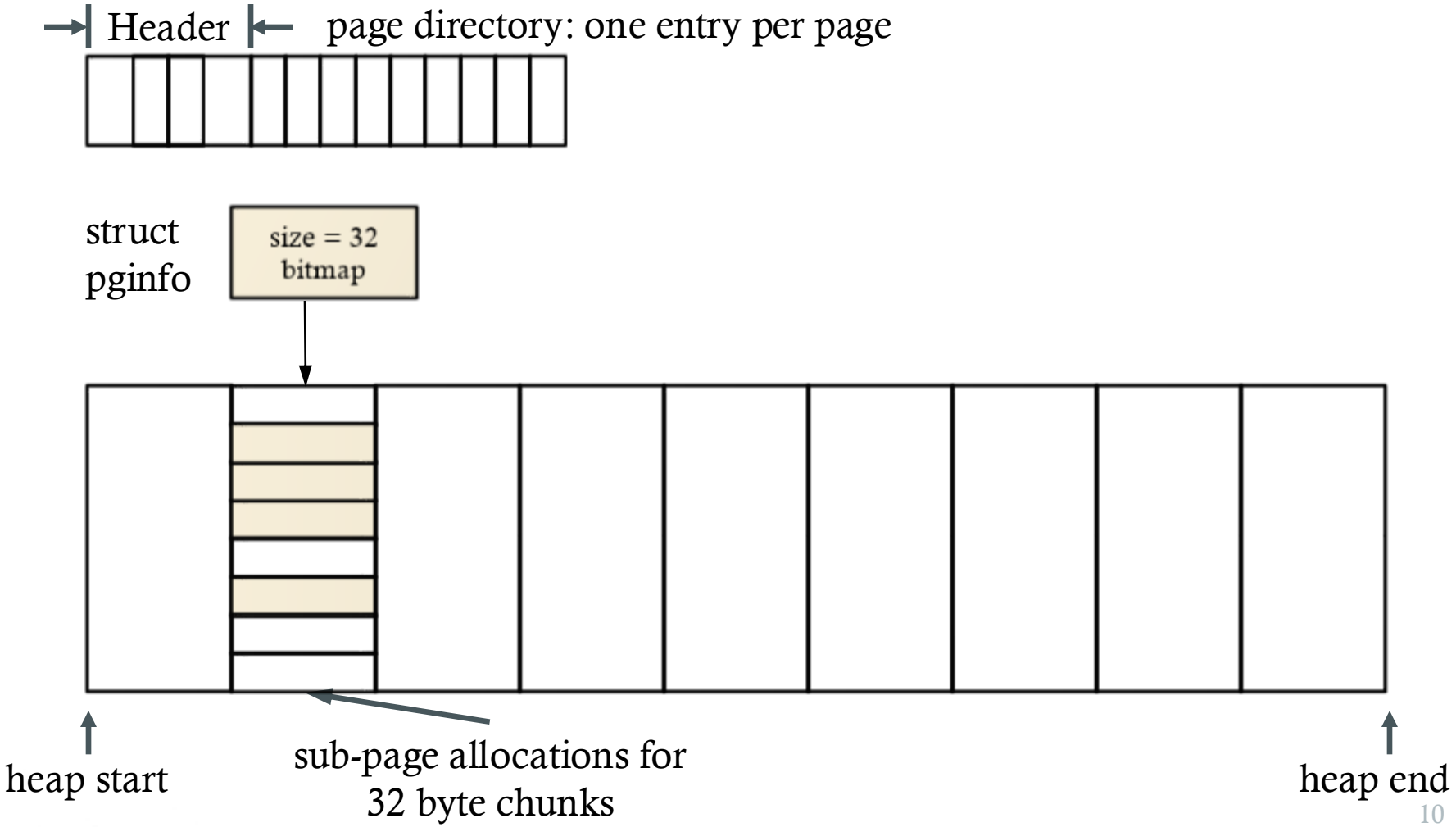
Phkmalloc Page Allocator



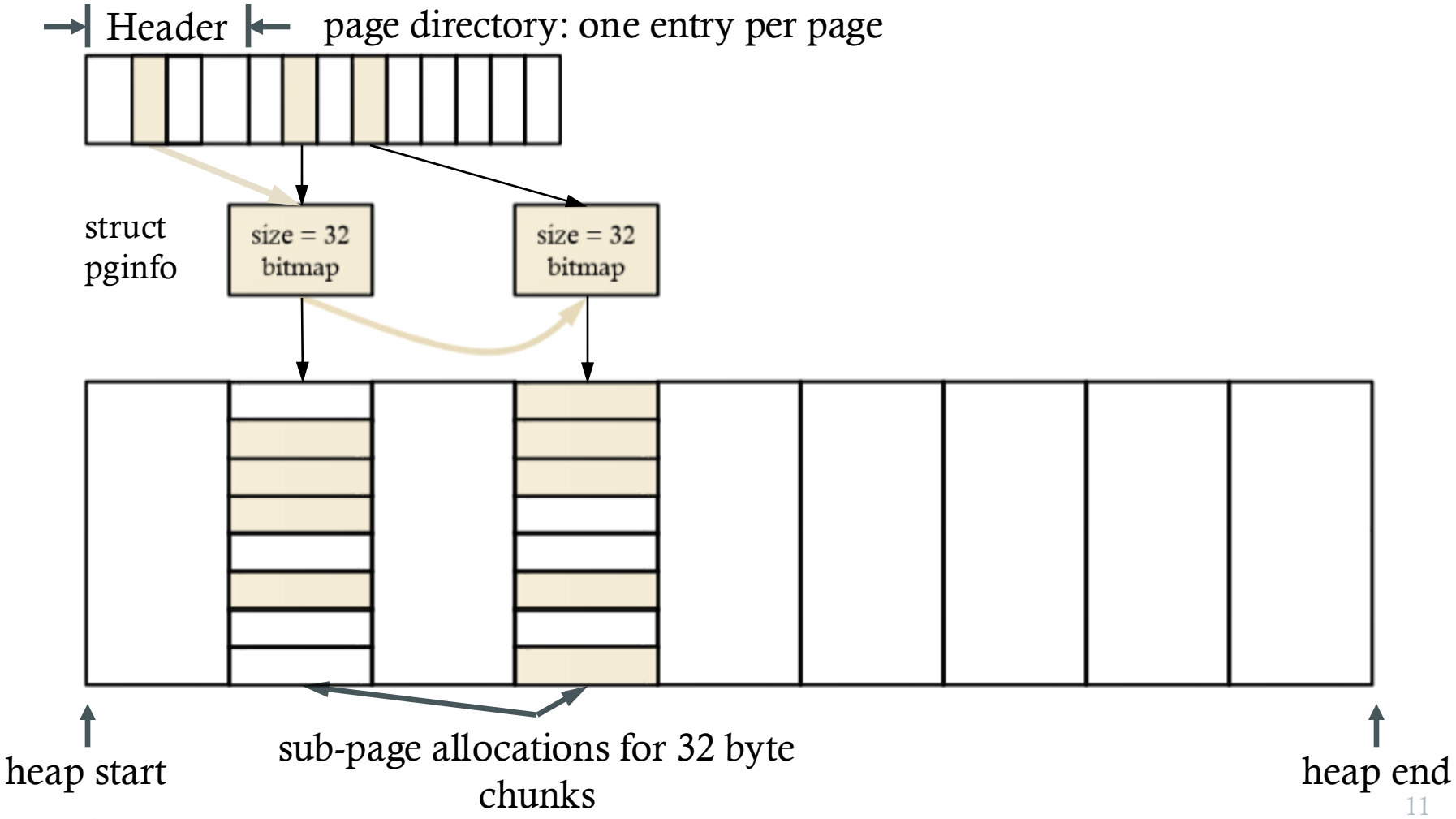
Page Allocator Design

- p = allocate(n pages)
  - Look up free list and find first run of free pages >= n pages
  - Update free list, page directory with S, F flags
  - If n contiguous pages not available
    - Grow heap
    - Reallocate contiguous page directory using mmap
- free(p)
  - Look up page in page directory, in constant time
  - Use S, F flags to determine size of page run, update page directory
  - Add page run to free list in address order, with coalescing
- Why use two structures, page directory and free list?

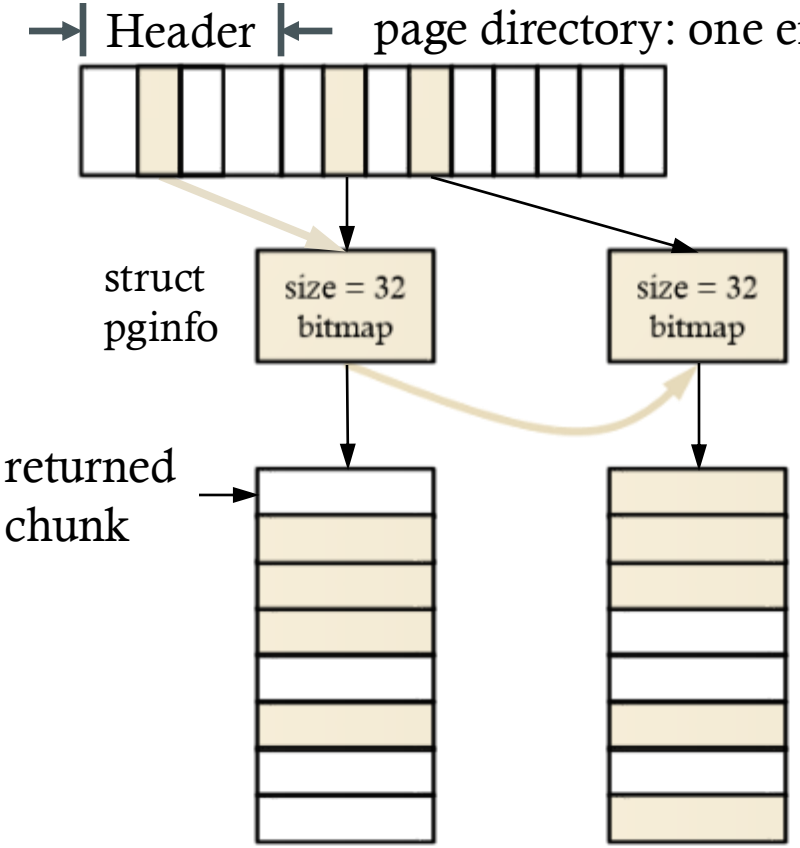
Phkmalloc Sub-Page Allocator



Phkmalloc Sub-Page Allocator



Phkmalloc Sub-Page Allocator



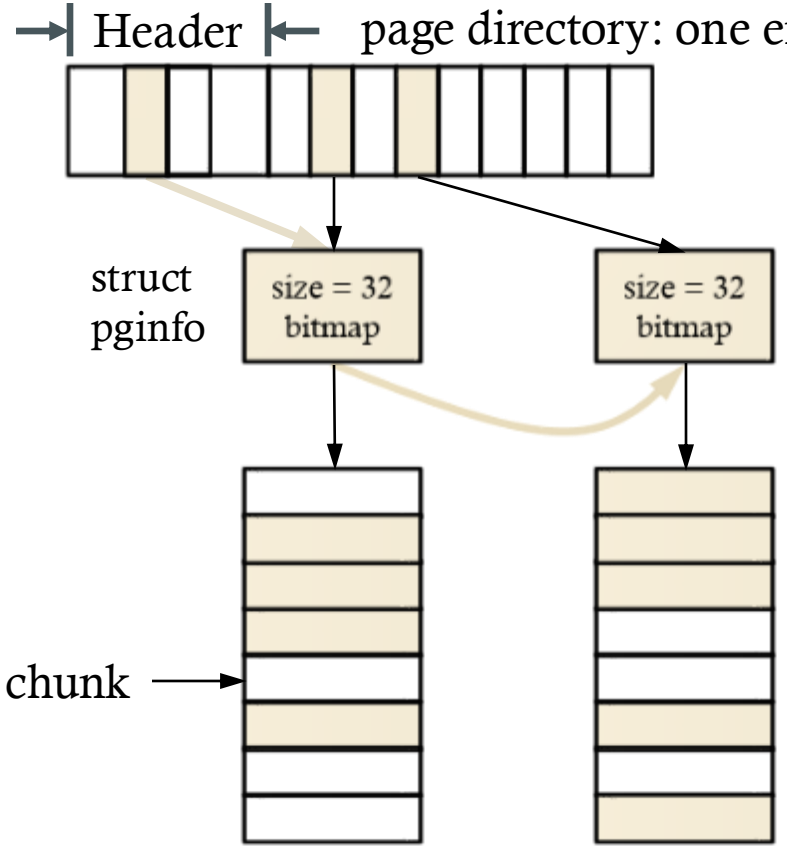
allocate(32 bytes):

1. **If linked list is empty, allocate new page and pginfo**
2. **Use bitmap to find free chunk in first pginfo in linked list, update bitmap**
3. **If all chunks allocated, remove pginfo from linked list**

How to Allocate from a Bit Vector?

```
>man ffs

ffs(3)                          Library Functions Manual                          ffs(3)

NAME
       ffs, ffsl, ffsll - find first bit set in a word

LIBRARY
       Standard C library (libc, -lc)

SYNOPSIS
       #include <strings.h>

       int ffs(int i);

       #include <string.h>
.....
ESCRIPTION
       The  ffs()  function returns the position of the first (least significant) bit set
       in the word i.  The least significant bit is position 1 and the  most  significant
       position  is, for example, 32 or 64.  The functions ffsll() and ffsl() do the same
       but take arguments of possibly different size.

RETURN VALUE
       These functions return the position of the first bit set, or 0 if no bits are  set
       in i.
```

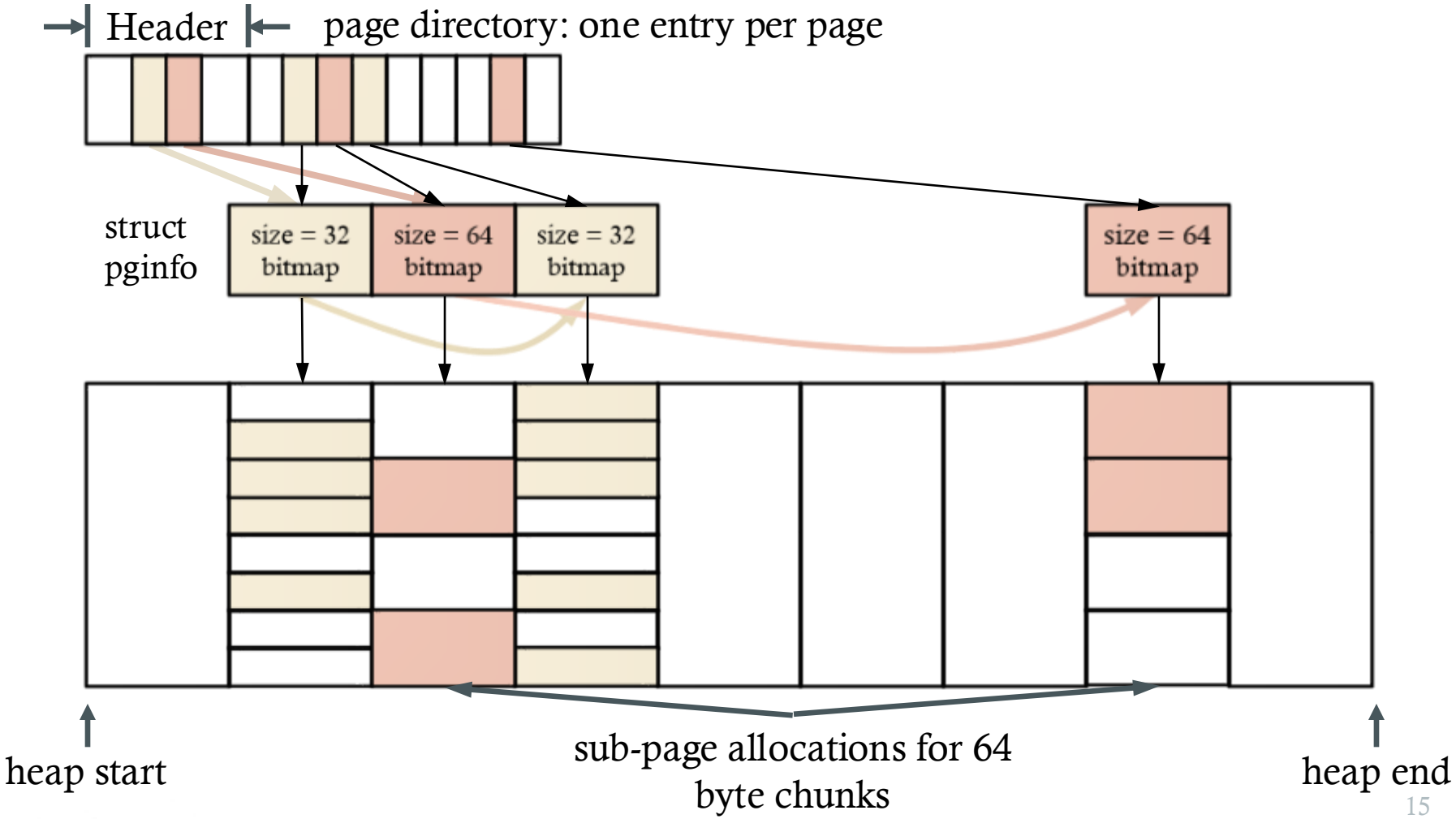
Phkmalloc Sub-Page Allocator



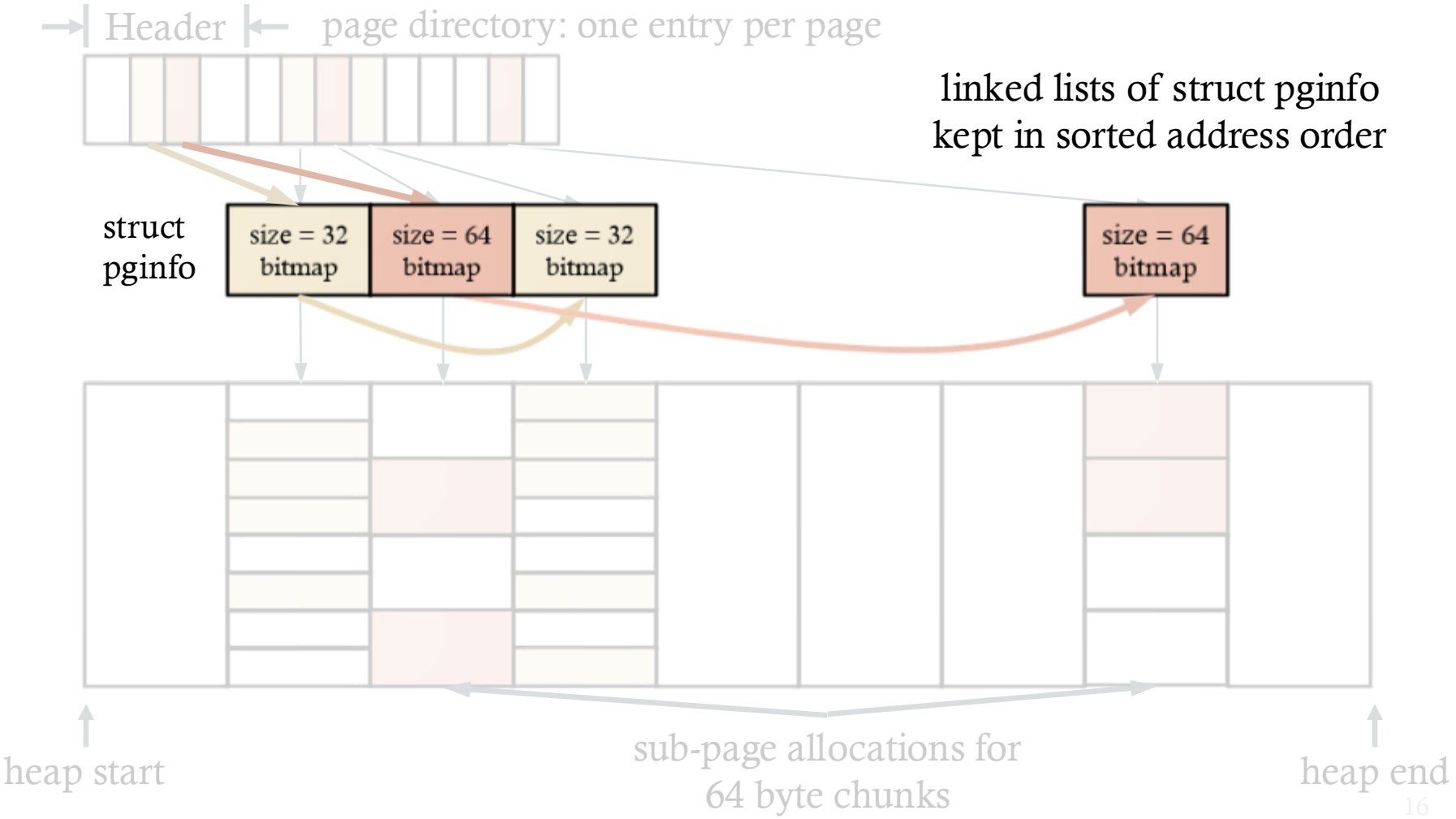
free(chunk):

1. **Find pginfo associated with chunk, update bitmap**
2. **If this is first free chunk in page, add pginfo into linked list**
3. **If all chunks free, remove pginfo from linked list, free it, free page**

Phkmalloc Sub-Page Allocator



Phkmalloc Sub-Page Allocator



Phkmalloc Summary

- Two level design for page-level and sub-page level allocations
  - Linked list for multi-page allocation
    - Simple and works well since large allocations are less frequent
  - Power of 2, segregated storage for sub-page level allocation
    - Almost constant time allocation, free
    - Low space overhead (for page header)
    - No splitting, coalescing overhead
- Comparison with simple segregated storage
  - Internal fragmentation is similar, due to rounding to power of 2
  - Phkmalloc reduces external fragmentation by 1) returning empty chunk pages, 2) preferring low addresses for allocation

Implicit Memory Management: Garbage Collection

Drawback of Malloc and Free

- Malloc and Free require explicit memory management
  - Programmers need to keep track of allocated memory and explicitly deallocate blocks that are no longer needed
- Disadvantage
  - Programming burden
    - Walking the entire data structure graph
  - Highly error-prone
    - Especially when considering quality of most programmers!
- Problems
  - Dangling pointer bugs
  - Double free bugs
  - Memory leaks

Garbage Collection

- Garbage collection: automatic reclamation of heap-allocated storage – application does not have to explicitly free memory

```
void foo() {
   int *p = malloc(128);
   return; /* p block is now garbage */
}
```

- Common in functional languages, scripting languages, and modern object-oriented languages:
  - Lisp, ML, Java, Perl, Python, Matlab, etc.
- Variants (conservative garbage collectors) exist for C and C++
  - Cannot collect all garbage

Garbage Collection

- How does the memory manager know when memory can be freed?
  - Unless you know the future
  - But we can tell memory blocks impossible to access: garbage = blocks no pointer points to them (in transitive sense)
- Need to make certain assumptions about pointers
  - Memory manager can distinguish pointers from non-pointers
  - All pointers point to the start of a block
  - Size of allocated blocks can be determined
  - Cannot hide pointers (e.g., by coercing them to an integer, and then back again)
  - These can be trivially imposed by programming languages

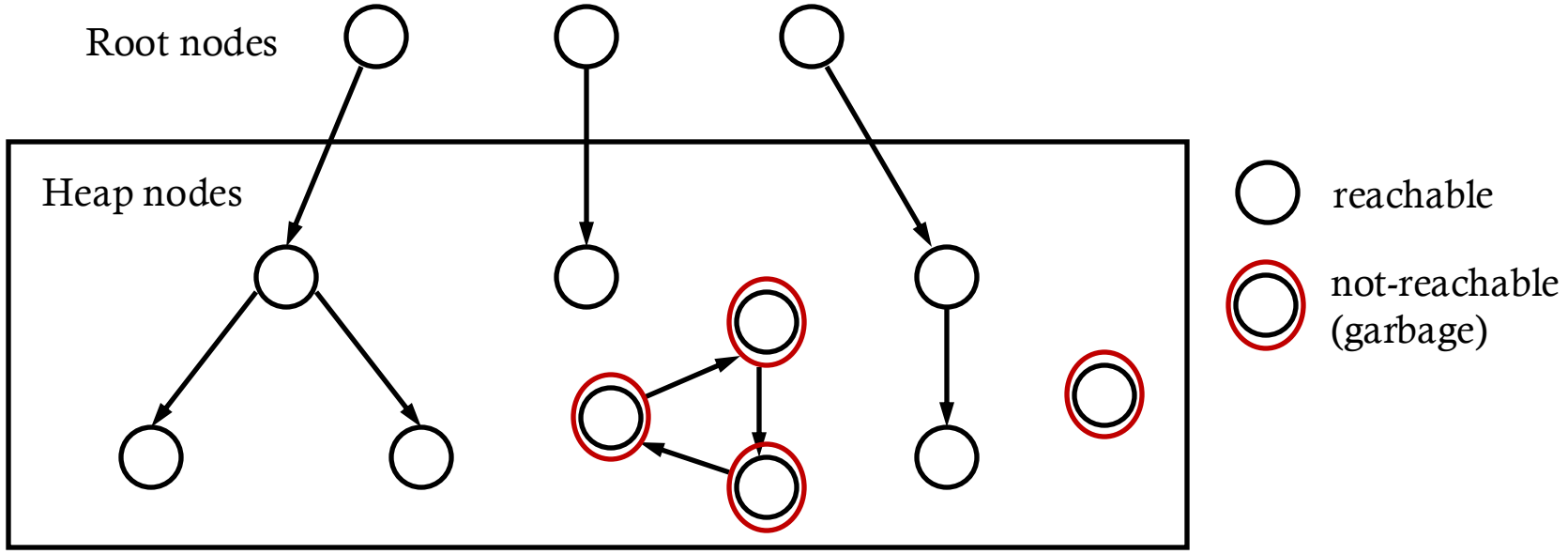
Classic GC algorithms

- Mark and sweep collection (McCarthy, 1960)
  - Does not move blocks
- Reference counting (Collins, 1960)
  - Does not move blocks
- Mark and copy collection (Minsky, 1963)
  - Moves and compacts blocks (not discussed)
- For more information, see Jones and Lin, "Garbage Collection: Algorithms for Automatic Dynamic Memory", John Wiley & Sons, 1996.

Memory as a Graph

- We view memory as a directed graph
  - Each block is a node in the graph
  - Each pointer is an edge in the graph
  - Locations not in the heap that contain pointers into the heap are called root nodes (e.g. registers, local variables = locations on the stack, global variables)

Memory as a Graph

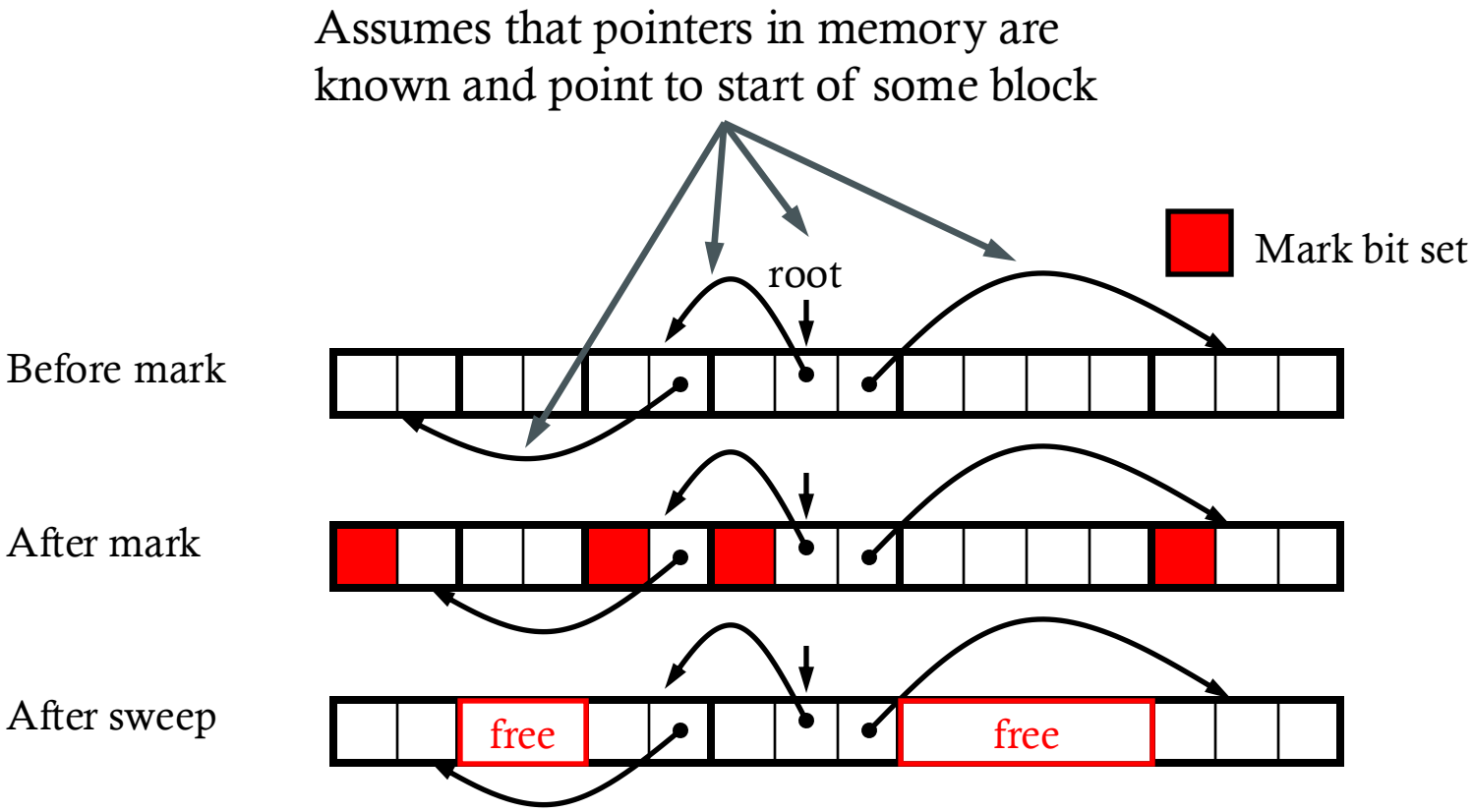


- A node (block) is reachable if there is a path from any root to that node
- Non-reachable nodes are garbage since they cannot be accessed by the application

Mark and Sweep Collection

- Can build on top of malloc/free package
  - Allocate using malloc until you "run out of space"
- When out of space:
  - Keep extra mark bit in the head of each block
  - Mark: Start at roots and set mark bit on all reachable memory
  - Sweep: Scan all blocks; free the blocks that are not marked

Mark and Sweep Example



Mark and Sweep: Some Details

- How to mark:
  - Depth first
  - Breadth first
- Need to deal with cycles
  - During search, return immediately when visiting a block that is already marked

Baker's Algorithm

- Problem: The basic mark and sweep algorithm takes time proportional to the heap size since sweep must visit all blocks
  - Sweep must visit all blocks to determine if they are marked

- Baker's algorithm keeps a list of all allocated chucks of memory, in addition to the free list
  - During sweep, look at the allocated chunks only
  - Free the allocated blocks that are not marked
  - Mark and sweep times are both proportional to size of allocated memory

Mark and Sweep in C

- Strategy:
  - Check every word to see if it points to a block in the heap, and if it does, mark that block
- Problems
  - May get some false positives
    - I.e., a word that isn't a pointer is treated as one, causing garbage to be treated as reachable memory, leads to external fragmentation
  - C pointers can point to middle of block
    - Solution: Need to keep track of start and end of each block; use a binary search tree, keyed by start address of allocated blocks
- For more details see: "A garbage collector for C and C++", https://hboehm.info/gc/index.html

Mark and Sweep Issues

- Have to identify all references & pointers
- Requires jumping all over memory
  - Terrible for performance
    - Cache hit rate, paging
- Search time proportional to non-garbage
  - May require lots of work for little reward (i.e., not much garbage collected)
- Must stop program execution while performing GC
  - Today, garbage collection is performed partially (not all garbage is collected at once) and incrementally (in parallel with program execution)

Reference Counting

- Basic idea:
  - In header, maintain count of # of references to block
  - When new reference is made, increment counter
  - When reference is removed, decrement counter
  - When counter goes to 0, free block
- Requires compiler support
  - red: code inserted by compiler
- Or use "smart pointers"
  - C++ tricks to ref count automatically

```
ptr p = new obj
p.cnt++
ptr q = new obj
q.cnt++

p.cnt–
if (p.cnt==0) free p
p = q
q.cnt++
```

Reference Counting Problems

- malloc may return null
  - Need to check, adds cost
- Garbage-collected blocks may include pointers
  - Need to recursively follow and decrement count of pointed-to blocks
- Cycles: reference counting fails conservatively
  - It may not free all garbage (but it will never free non-garbage)
- Counter increments and decrements need to be atomic
  - Adds overhead to each increment and decrement

Application-Aware Allocators

- Allocation patterns
  - As application programmer, you know your pattern
  - Leverage it!
- Allocation data structures
  - Lists
  - Other structures

Allocation Patterns

- Block lifetimes are not random
  - Ramp – allocations throughout program lifetime without releases
  - Plateau – allocations, then lengthy usage, then releases
  - Peaks – bursty behavior and short object lifetimes
  - Phases - allocations associated with lifetime of tasks, which runs in different phases
- Block sizes are not random
  - Zorn and Grunwald, 1992 study, six allocation-heavy C programs
  - Found that 53-93% of requests were for top two sizes

Strategies for Application-Aware Memory Management

- Strategy 1: Dominant pattern
  - There are likely only a few dominant data structures that impact the performance of your application, focus only on them
  - Leave the rest to general purpose allocators, which are already very good
- Strategy 2: Arena pattern
  - Allocate blocks with similar die time continuously
- Strategy 3: Slab pattern
  - Allocate blocks with similar sizes contiguously

Arena Allocator

- Data Structures
  - A stack of pages
  - A watermark pointer
- Allocate
  - return current watermark
  - increment watermark, if exceed page boundary, add new page
  - take care of alignment
- Free:
  - only at arena level when all pages are freed.

- Complexity:
  - Allocate: O(1)
  - Free: O(1)
  - Programmer burden: like garbage collector
- Usage in Compilers
  - Arenas reserved for different compiler phases
  - No need to walk data structure graph to tear it down: they die together anyway

Slab Allocator

- Data Structures
  - A stack of pages
  - A list of free chunks
- Allocate
  - Grab a free chunk if available
  - Add new page of not
- Free:
  - add to front of free list

- Complexity:
  - Allocate: O(1)
  - Free: O(1)
- Usage in Kernel
  - Network packet
  - Almost everywhere
- Usage in memcached
  - In-Memory application cache

Meta-Data Considerations: Where

- We have seen linked list(s) of variable sized free blocks
  - Implicit – link allocated and free blocks
    - Not used due to linear time allocation
  - Explicit – link free blocks, use one or more lists
    - More commonly used
- Where is the list stored?
  - Integrated: use space within the free blocks to hold the links
    - Benefit: no need to separately manage space for links
    - Problem: poor locality when traversing the list (discussed later)
  - External: use space separate from allocated or free blocks
    - Benefit: better locality when traversing the list
    - Problem: need to manage this space, how is it grown (discussed later)

Meta Data Considerations: Order

- What should be the order of free blocks in the list?
  - LIFO
    - Add freed block to beginning of list
    - Provides locality
  - FIFO
    - Add freed block to end of list
    - Benefits?
  - Sorted by block size
    - Limits traversal for smaller allocations
  - Sorted by address
    - Reduces heap fragmentation (we will see this later)

Meta-Data Data Structures

- Bitmap for fixed-size contiguous blocks
  - Can be used for segregated storage
    - Each list maintains blocks of the same size
    - Blocks of the same size must be allocated contiguously
    - leading zero / leading one detection instruction for help
- Trees
  - Heap requires searching for a free block of a given size
  - Use ordered trees to reduce search times compared to linked list
    - E.g., use red-black tree to perform best fit in `log(n)` time, where `n` is number of free blocks

Other Considerations

- Cache-Oblivious Data Structures
  - Allocator assign addresses
    - Effectively affecting memory layout
  - If somehow we can making visit order ~= address order
  - Effectively making memory accesses a sequential scan
  - We know memory hierarchy likes that!
    - No tuning: it does not matter much what cache/VM parameter values are

Case Study 2: jemalloc

- Designed to scale on multiprocessors and to minimize fragmentation
- Original design: Jason Evans, "A Scalable Concurrent malloc(3) Implementation for FreeBSD", BSDCan 2006
  - Please read for details, although the description is not great ☹
- Various versions used in several BSD releases, in Firefox
- Facebook has made several optimizations ☺
  - Please read details about the design philosophy at: "Scalable memory allocation using jemalloc", https://engineering.fb.com/2011/01/03/core-data/scalable-memory-allocation-using-jemalloc/

Motivation

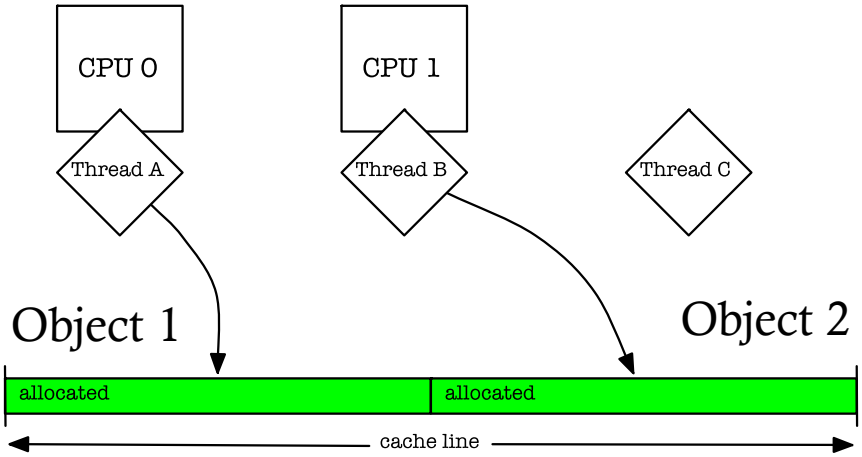
- Traditional allocators focus on optimizing for
  - Space utilization
  - Performance (on single core)
- However, these allocators have two issues that lead to poor multi-core scaling
  - Data packing
  - Locking scheme

Data Packing

- Traditional malloc packs data as efficiently as possible
  - Improves caching, which is critical for program performance
  - Helps with growing disparity between CPU and DRAM speeds
    - memory-wall
- However, packing data structures can lead to poor multi-threaded performance

Packing Data: False Cache-Line Sharing

- Heap objects 1 and 2 accessed by Thread A and Thread B
  - The two objects are not shared by the threads
- Say, malloc allocates objects within same cache line



- Reads and writes on different cores for unrelated objects cause cache line invalidation, reduce performance dramatically

Avoiding Cache-Line Sharing

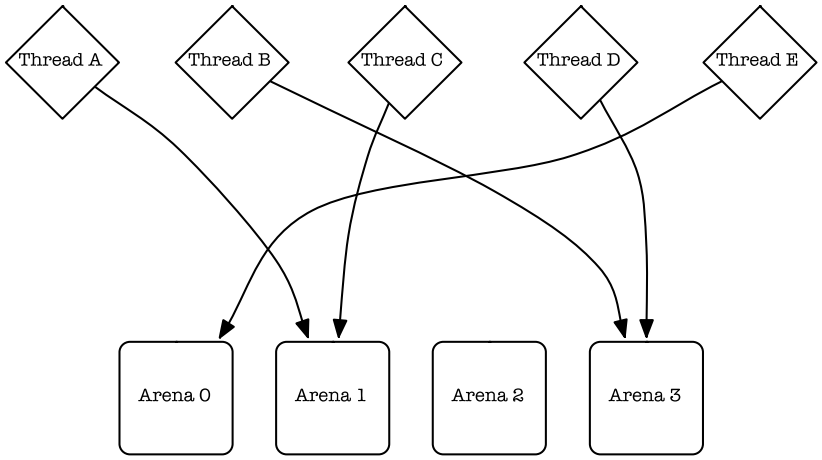
- malloc could add padding to the objects
  - However, padding causes internal fragmentation, making it cache-unfriendly
- Require user to pad objects that cause multi-core scaling bottleneck
  - Pros: Causes internal fragmentation for some objects only
  - Cons: Requires programmers to carefully align data structures that cause contention (i.e., limit scaling), requires detailed profiling to determine contention
- Neither scheme is attractive

Locking Scheme

- Traditional malloc used simple locking scheme
  - E.g., lock all data structures, serialize malloc and free calls
  - Sufficient with limited # of cores
  - Doesn't scale well today since allocator locks become a performance bottleneck

Avoiding Allocator Bottlenecks

- Use memory arenas
  - Split heap memory into arenas, or continuous blocks of memory
  - Each thread allocates memory from a single arena
  - Memory allocated from an arena is freed into same arena
- Memory arenas reduce cache line sharing and lock contention



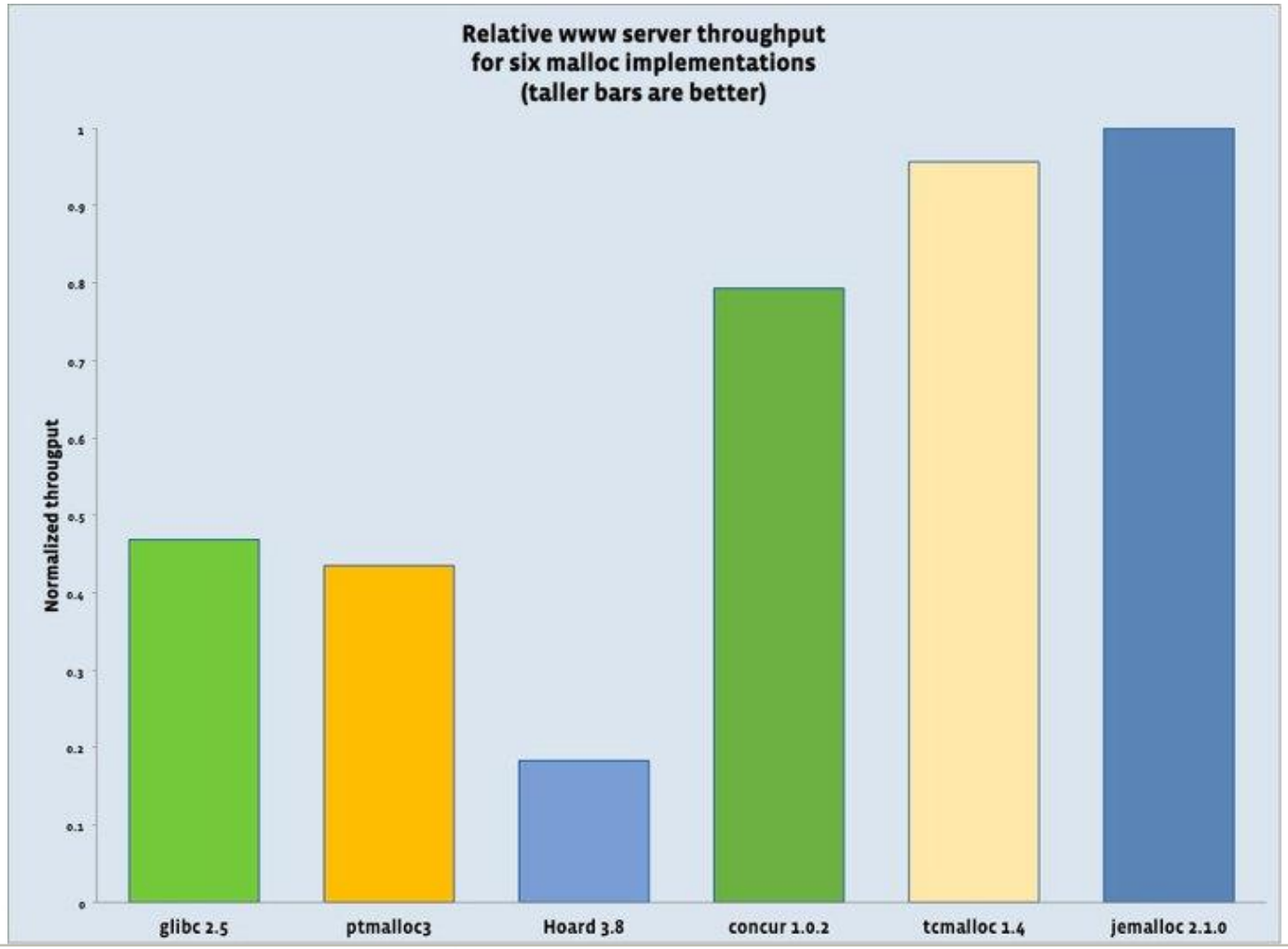
Memory Arenas for Threaded Applications

- Total arenas typically limited to 4-8 times the number of cores
- Options for mapping threads to arenas
  - Use simple round-robin
    - Need to account for thread exit, thread locking behavior, etc.
  - Use a hashing mechanism
    - Can lead to load imbalance due to poor hashing
  - Another optimization: During mapping, switch to another arena if `malloc` encounters an arena that is locked by another thread
    - Using `pthread_mutex_trylock()`)

Jemalloc: Scaling for Multi-Cores

- Each thread allocates from an arena
  - Reduces locking overheads, since few threads access each arena
  - However, arenas still require synchronization
    - Allocation requires locking a bin in the arena and/or the whole arena
- Add a per-thread allocator cache
  - Each thread maintains a cache of small objects up to 32 KiB
  - Most allocation requests hit the cache and require no locking
  - # of cached objects per size class is capped so that synchronization is reduced by ~10-100X
    - Higher caching causes increased fragmentation
    - Reduce fragmentation to improve data locality further by periodically returning unused cache objects to underlying arena

Jemalloc Performance



Jemalloc Summary

- Designed to scale on multiprocessors and to minimize fragmentation
- Multi-processor scaling
  - Minimize cache line sharing by using arenas
  - Use round-robin assignment of threads to arenas to reduce skew
  - Use per-thread allocator cache
- Minimize fragmentation
  - Carefully choose size classes (what are the tradeoffs?)
  - Prefer low addresses for allocation, similar to phkmalloc
  - Tight limits on metadata overhead
    - < 2%, ignoring fragmentation