ECE 454 Computer Systems Programming

Modern Allocators, Garbage Collection

Ashvin Goel, Ding Yuan
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
- Other memory management considerations
- Modern allocators
- Garbage collection

Case Study: 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

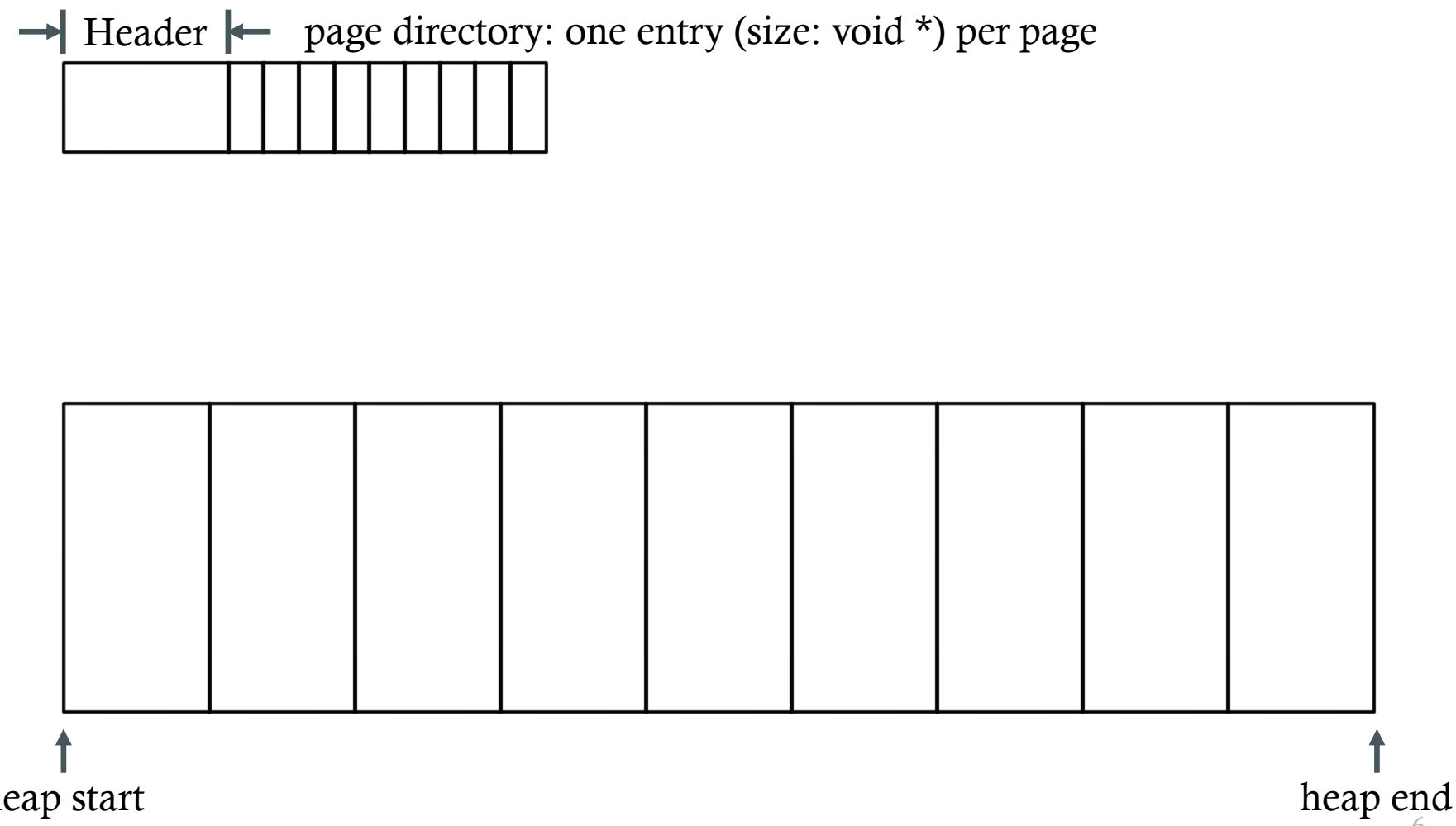
Problem with Existing Malloc

- 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

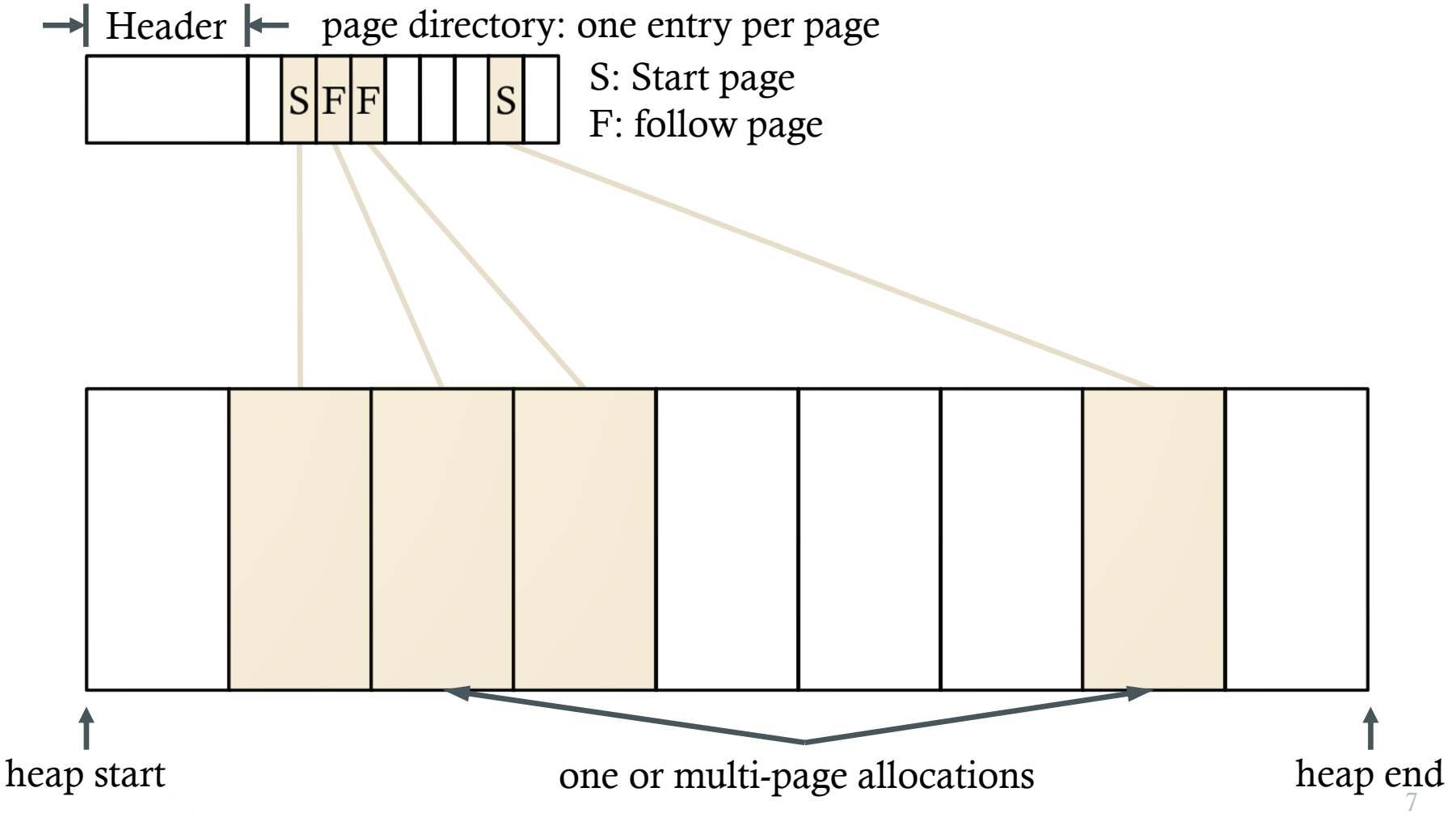
Phkmalloc Design

- 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
  - Keeps several lists of different fixed sizes
  - All allocation requests are rounded to next power of 2

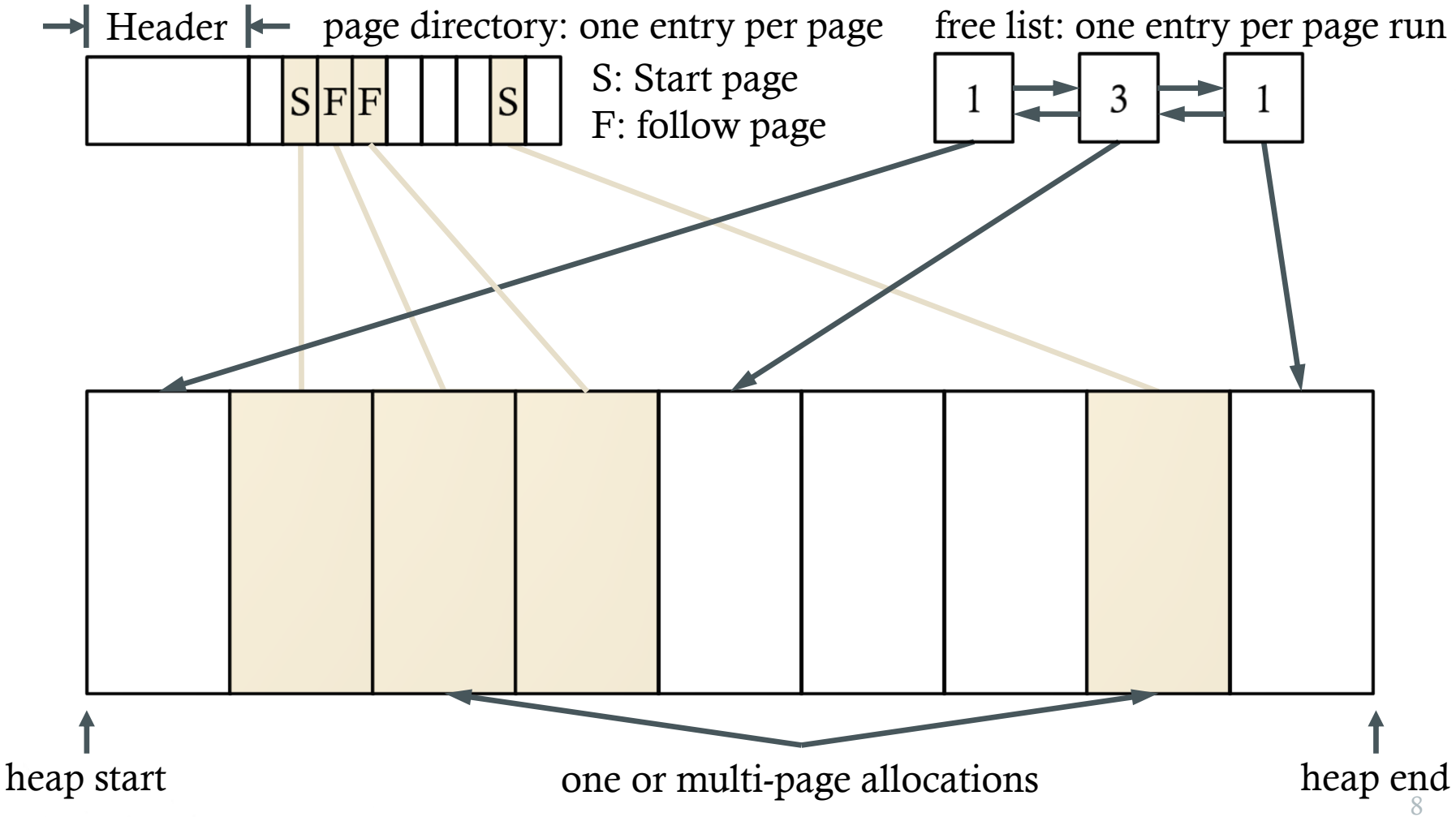
Phkmalloc Page Allocator



Phkmalloc Page Allocator



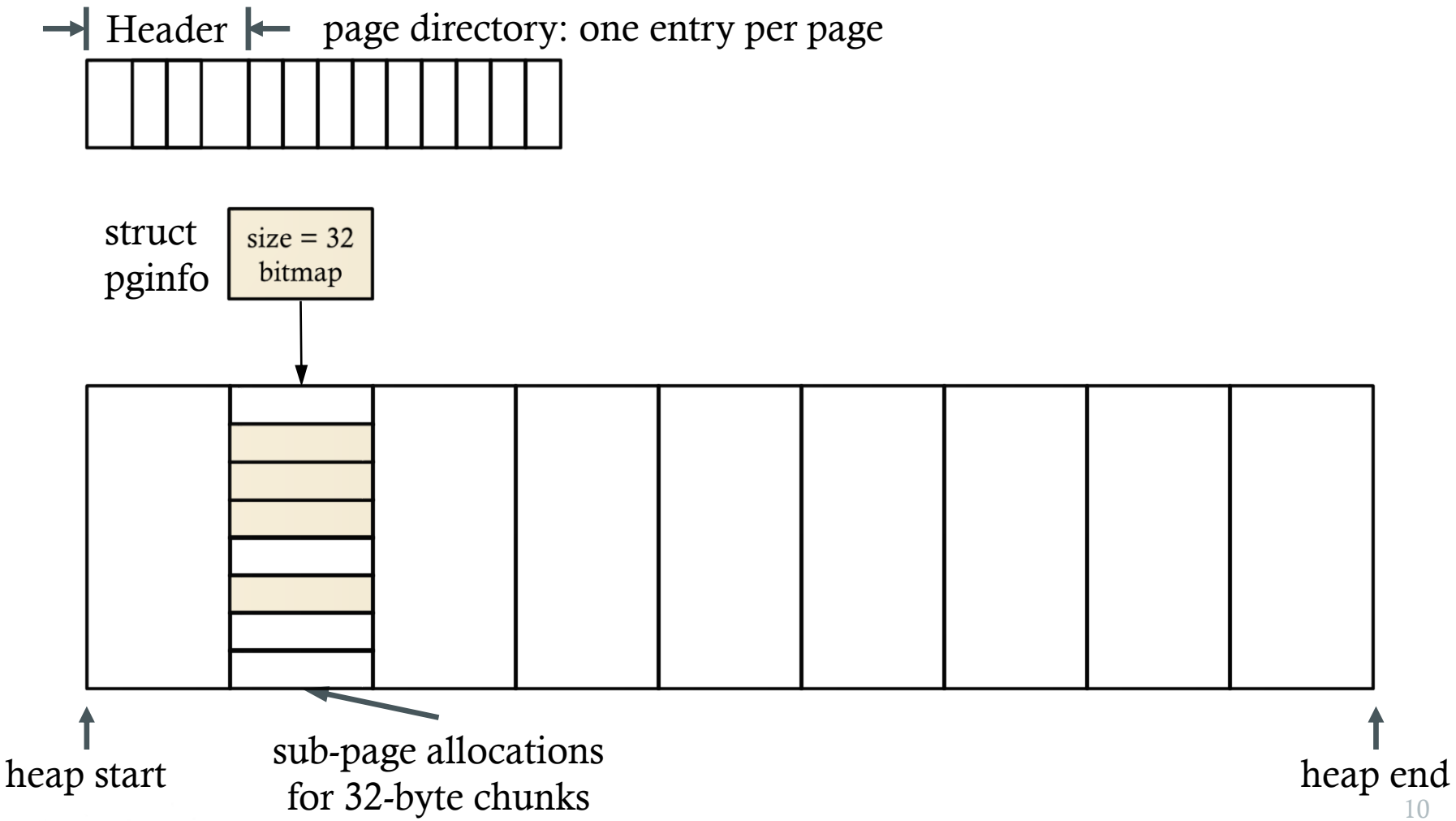
Phkmalloc Page Allocator



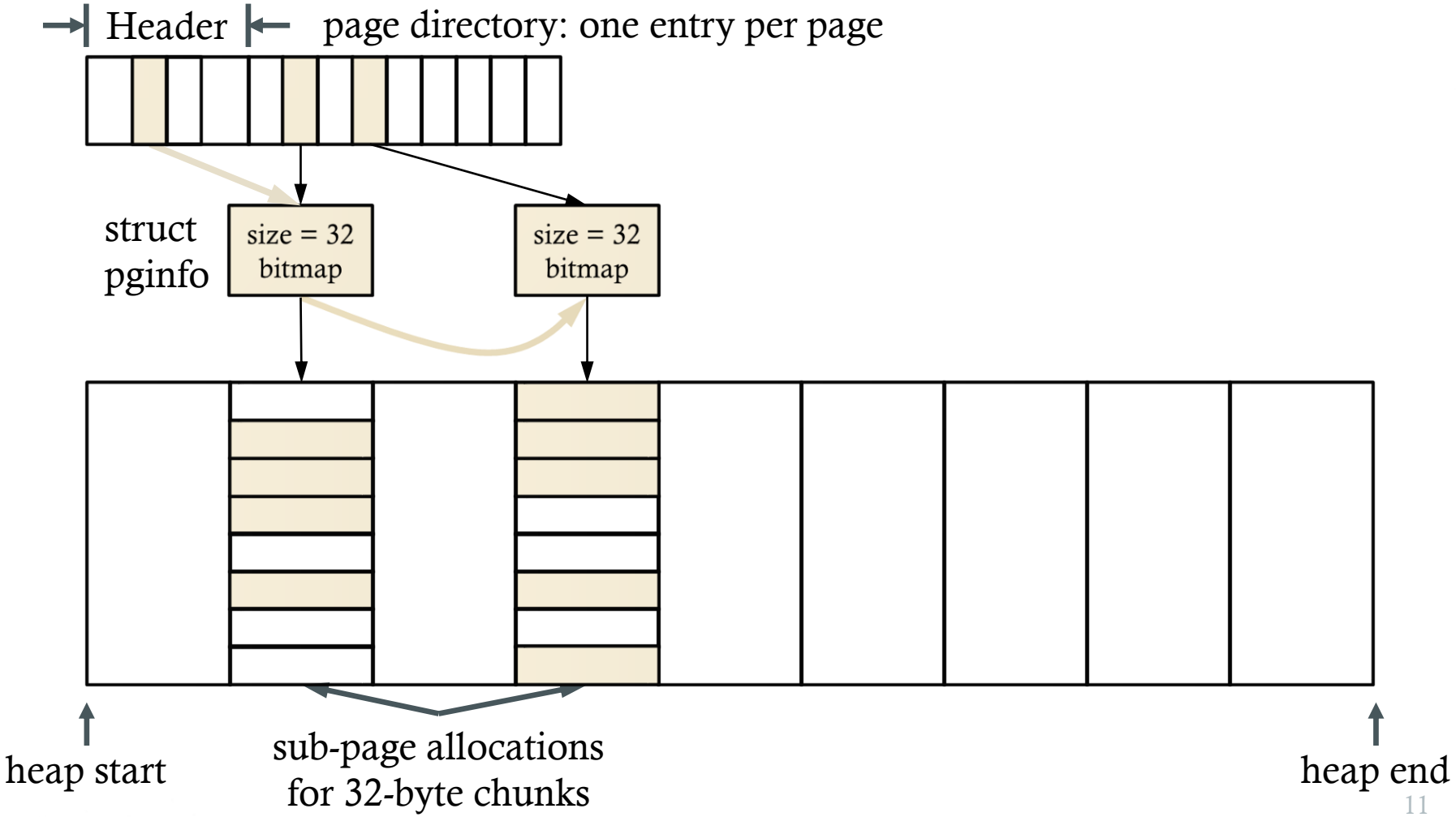
Page Allocator Design

- 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?

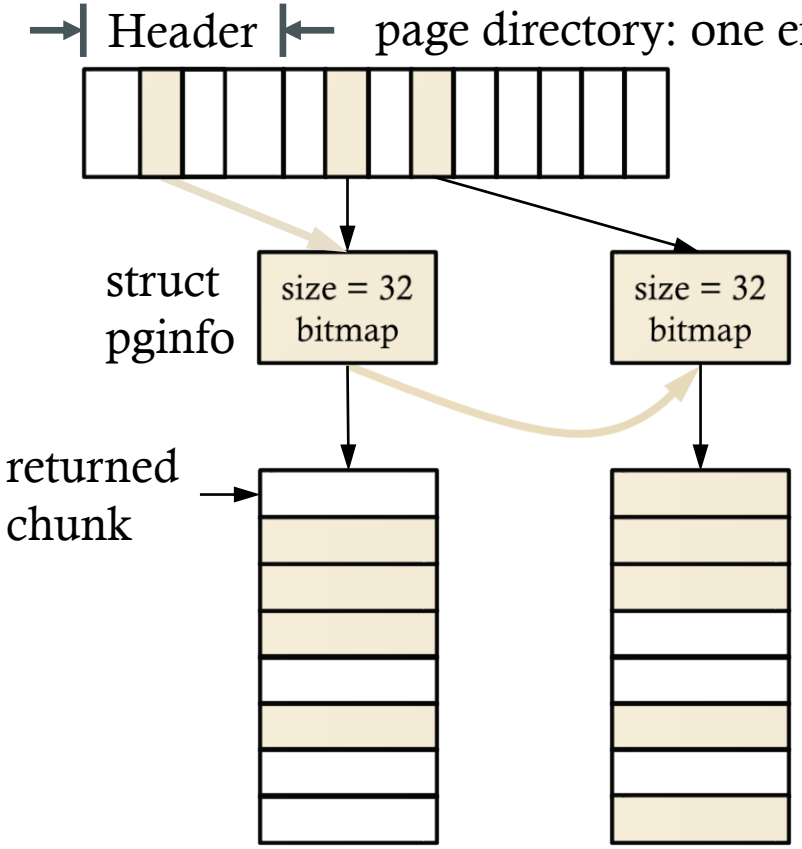
Phkmalloc Sub-Page Allocator



Phkmalloc Sub-Page Allocator



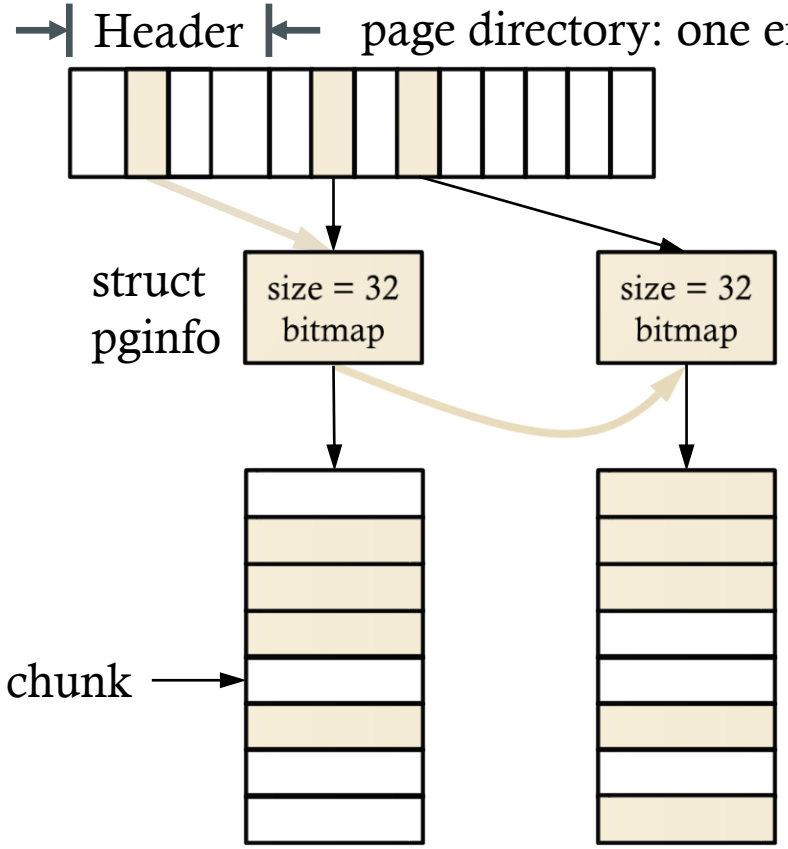
Phkmalloc Sub-Page Allocator



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**

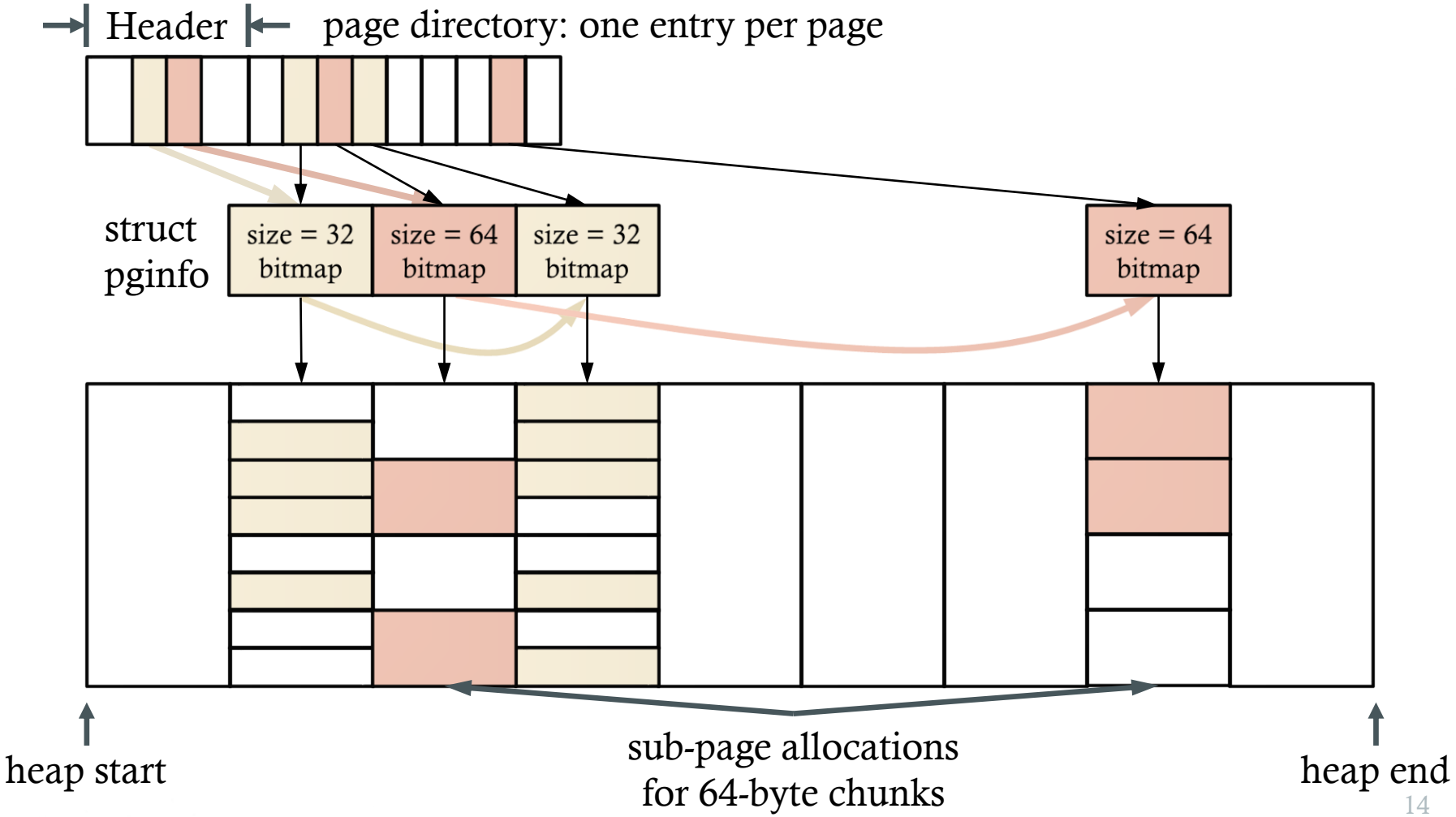
Phkmalloc Sub-Page Allocator



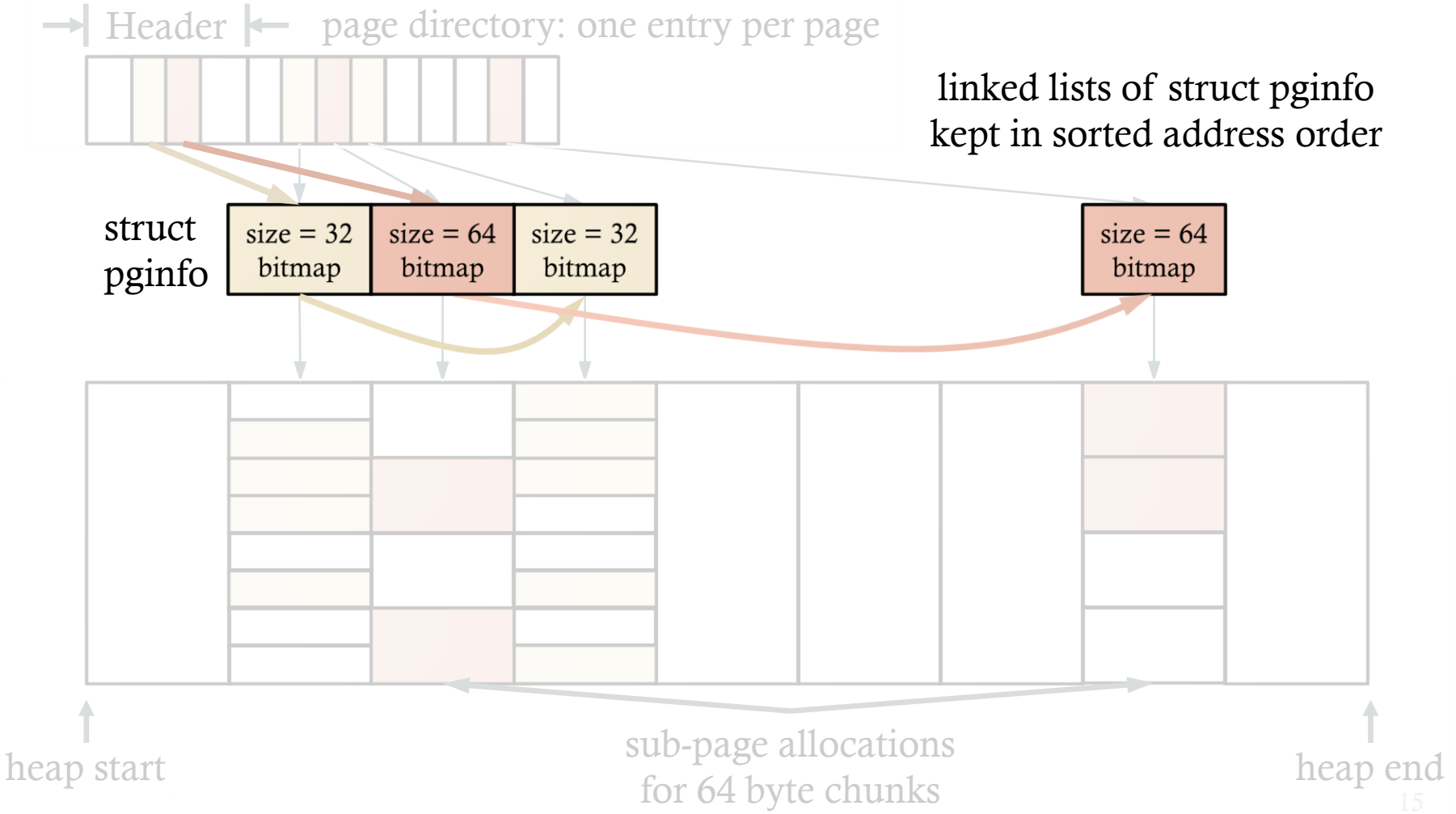
free(chunk):

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
  - Single free 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 free chunk pages to page allocator, 2) returning high-address pages to OS
  - Phkmalloc reduces paging since it doesn't touch free pages

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
  - Highly error-prone
    - Especially when considering quality of most programmers!
- Problems
  - Memory leaks
  - Dangling pointer bugs
  - Double free bugs

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, Haskell, Matlab, Java, Perl, Python, Go, 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?
  - We do not know what memory is going to keep being used in the future
  - But we can tell that certain blocks cannot be used if there are no pointers to them
- Need to make certain assumptions about pointers
  - Memory manager can distinguish pointers from non-pointers
    - Cannot hide pointers (e.g., by coercing them to an integer, and then back again)
  - All pointers point to the start of a block
  - Size of allocated blocks can be determined

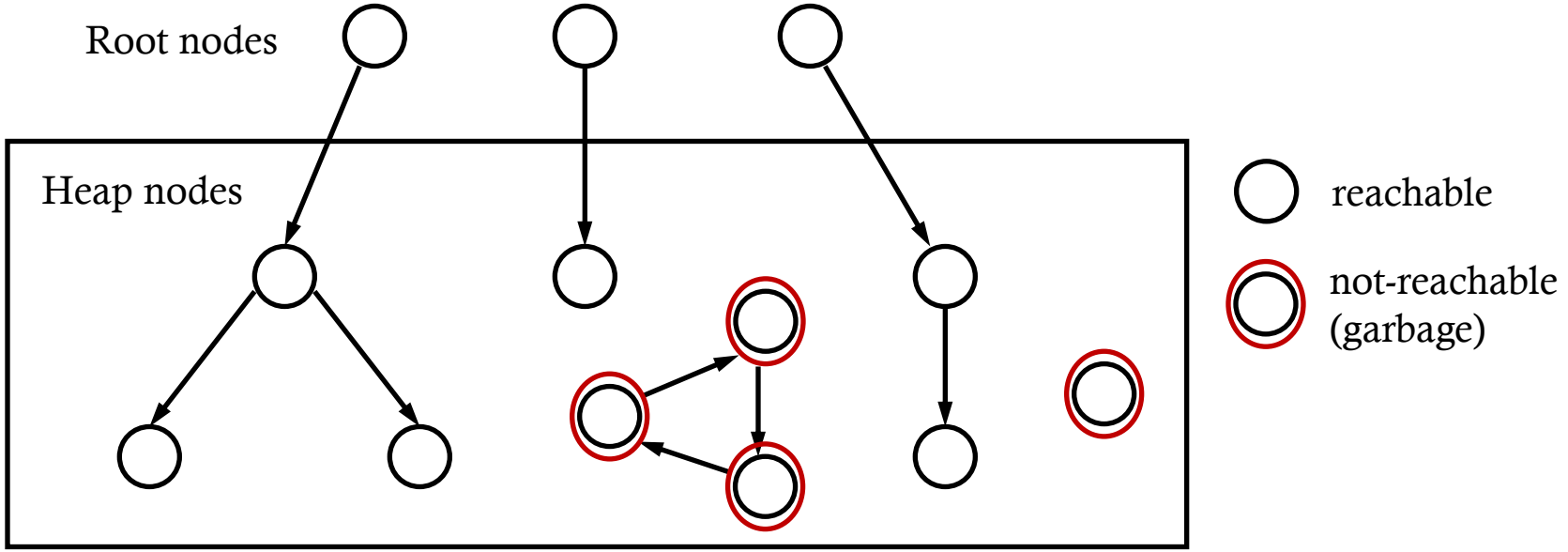
Classic GC algorithms

- 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 outside the heap (e.g. registers, stack variables, global variables) that contain pointers into heap are called root nodes

Memory as a Graph

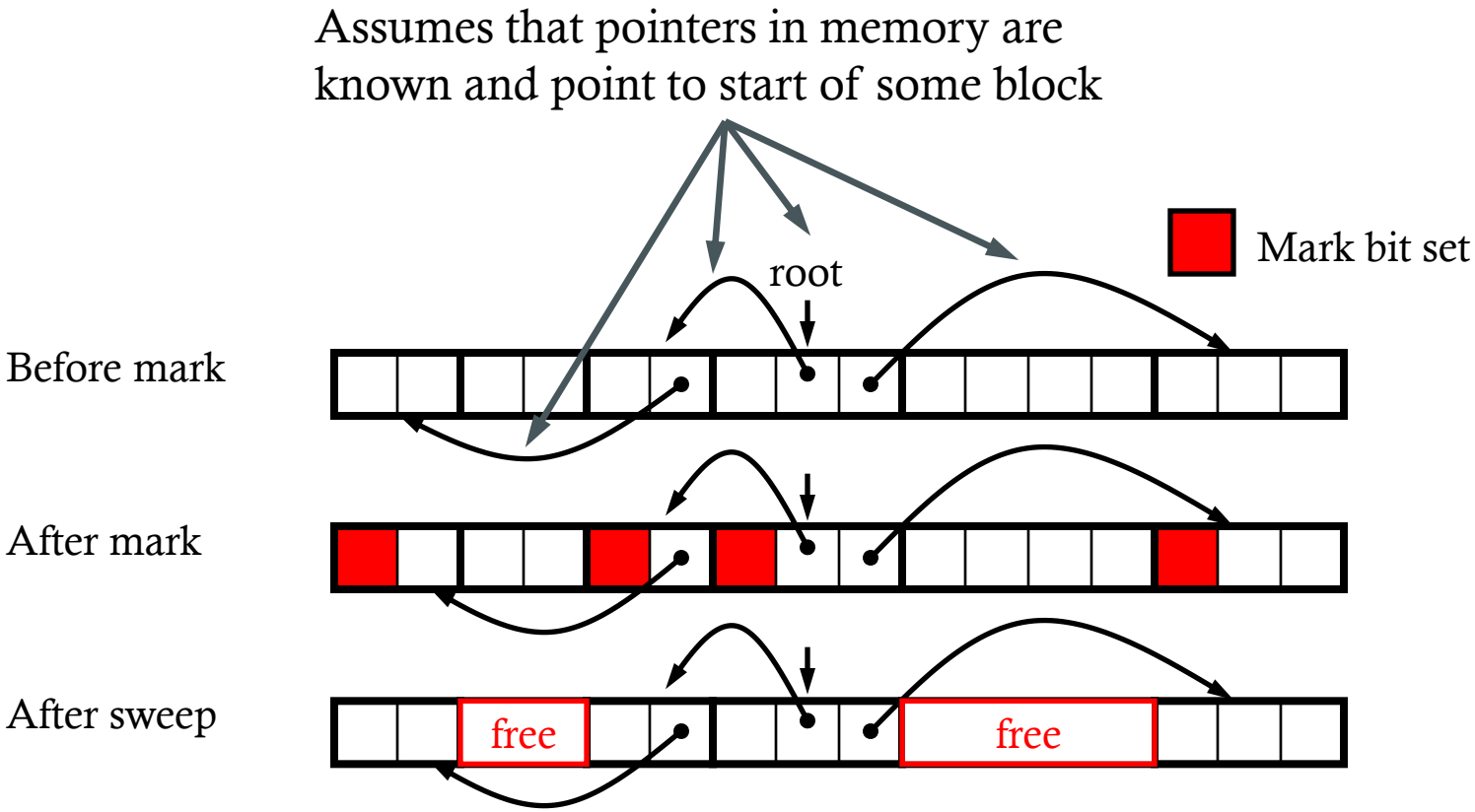


- A node (block) is reachable if there is a path from any root to that node
- Non-reachable nodes are garbage since they cannot be accessed by the application

Mark and Sweep Collection

- Can build on top of malloc/free package
  - Keep extra mark bit in the head of each block
- Allocate using malloc until you "run out of space"
- When out of space, run two phases:
  - Mark: Start at roots and set mark bit on all reachable memory
  - Sweep: Scan all blocks and 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 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
  - Poor cache locality: lowers cache hit rate, increases 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 or use of custom class objects
  - red: code inserted by compiler

```
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