Dynamic Memory Allocation

Nima Honarmand

Lecture Goals

- Understand how dynamic memory allocators work
  - In both kernel and applications
- Understand trade-offs and current best practices

What is Memory Allocation?

- Dynamically allocate/deallocate memory
  - As opposed to static allocation

- Common problem in both user space and OS kernel

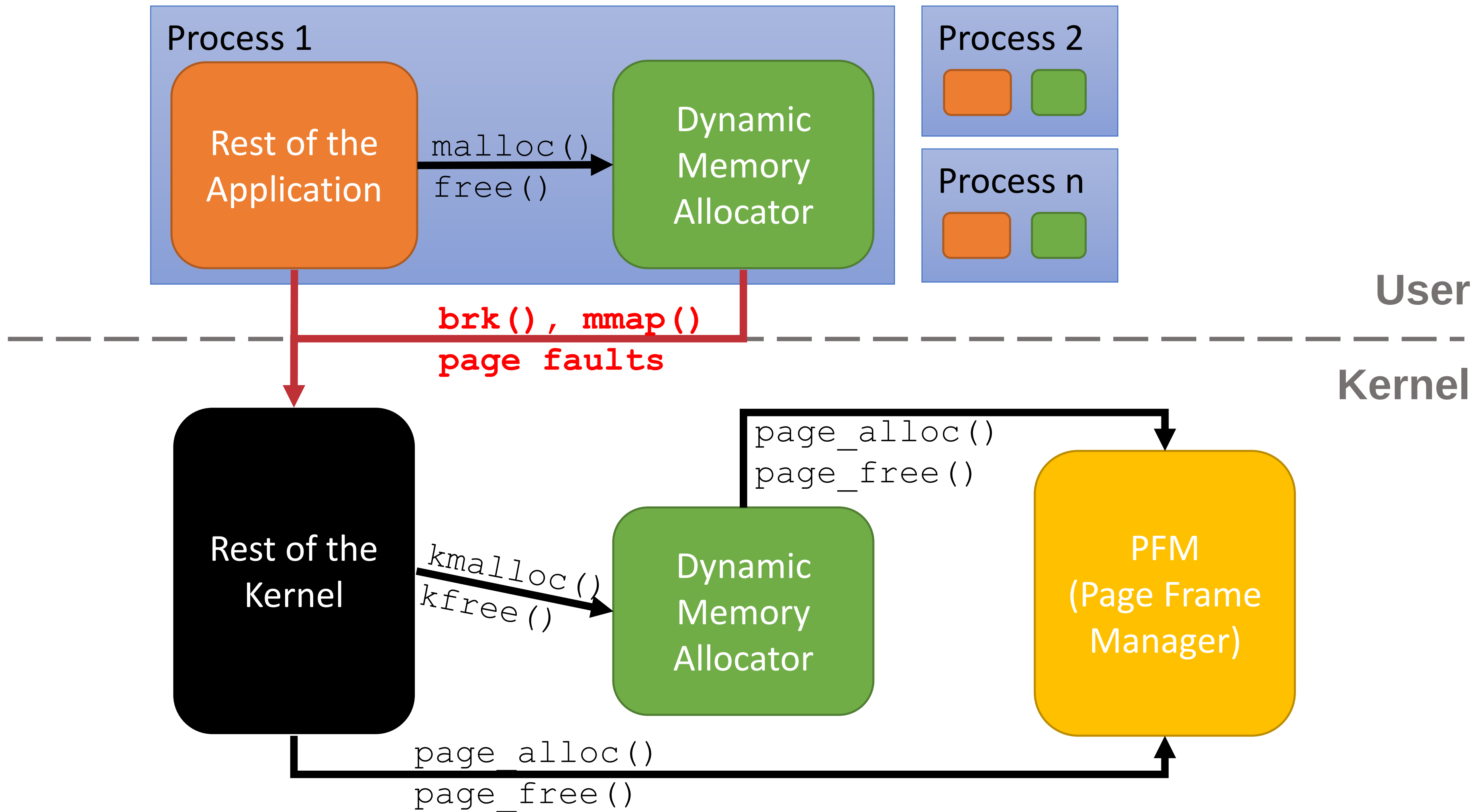
- *User space*: how to implement `malloc()/free()`?
  - **`malloc()`** gets pages of memory from the OS via **`mmap()`** and then sub-divides them for the application

- *Kernel space*: how to implement `kmalloc()/kfree()`?
  - Get pages from the physical page manager and sub-divide between memory requests in the kernel

Assumed API

- `void *malloc(int sz)`
  - Return a memory object that is at least of size `sz`
- `void free(void *ptr)`
  - Free the object pointed to by `ptr`
  - Note: no size provided
  - What if `ptr` does not point to a valid allocated object?

Overall Picture



Simple Algorithm: **Bump Allocator**

- malloc (6)
- malloc (12)
- malloc(20)
- malloc (5)

Example: Bump Allocator

- Simply “bumps” up the free pointer
- How does free() work?
  - It doesn’t; it’s a no-op
- Controversial observation: This is ideal for simple programs
  - You only care about free() if you need the memory for something else
- What if memory is limited?

  → **Need more complex allocators**

Overarching Issues

- Fragmentation
- Splitting and coalescing
- Free space tracking
- Allocation strategy
- Allocation and free latency
- Implementation complexity
- Cache behavior
  - Locality issues
  - False sharing

Fragmentation

- Undergrad review: What is it? Why does it happen?
  - Happens due to variable-sized allocations
- What is
  - *Internal* fragmentation?
    - Wasted space when you round an allocation up
  - *External* fragmentation?
    - When you end up with small chunks of free memory that are too small to be useful
- Which kind does our bump allocator have?

Splitting and Coalescing

- ***Split*** a free object into smaller ones *upon allocation*
  - Why?
  - To reduce/avoid internal fragmentation
- ***Coalesce*** a freed object with neighboring free objects upon deallocation
  - Why?
  - To reduce/avoid external fragmentation
- We need extra meta-data for these
  - We need the object size at least
  - Data/mechanisms to find the neighboring objects for coalescing

Keeping Per-region Meta-data

- Prepend the meta-data to the object (as a header)
  - On `malloc(sz)`, look for a free object of size at least `sz + sizeof(header)`

- For free objects, can keep the meta-data in the object itself

Tracking Free Regions

- Link the free objects in a linked list
  - Using the `next` field in the free object header
  - Keep in the list head in a global variable
- `malloc()` is simple using this representation
  - Traverse the free list
  - Find a big-enough object
    - Split if necessary
  - Return the pointer
- What about `free()`?
  - Easy to add the object to the free list
  - What about coalescing?
    - Not easy to do dynamically on every `free()` — Why?
    - Can periodically traverse the free list and merge neighboring free objects

Performance Issues (1)

- Allocation
  - Need to quickly find a big-enough object
  - Searching a free list can take long
  - Can use other data structures
    - All sorts of trees have been proposed
  - Or, can avoid searching altogether by having **pools of same-size objects**
- ***Segregated pools***: on `malloc(sz)`, round up `sz` to the next available object size, and allocate from the corresponding pool

Performance Issues (2)

- Deallocation
  - Returning free object to free list is easy and fast
  - Bit more overhead if using other data structures
- Coalescing
  - Not easy in any case
    - Have to find neighboring free objects
    - Book-keeping can be complex
  - Alternative: avoid coalescing by using segregated pools
    - All objects of the same size, no need to coalesce at all

Performance Issues (3)

- Concurrency issues
  - Need locking for concurrent `malloc()`s and `free()`s
    - Why? lots of shared data-structures
- Types of concurrency-related overheads
  1. Waiting for locks: contended locks cause serialized execution
     - If locks are used, only one thread can allocate/deallocate at any point of time
  2. lock/unlock is pure overhead, even when uncontended
     - Often use atomic instructions
     - Can take tens of cycles
- Alternative: avoid concurrency issues by having per-thread heaps
  - Or, at least, reduce contention by having multiple heaps and distributing the threads across them

Performance Issues (4)

- Single-processor issue:
  - Cache misses due to loss of temporal locality: too long between deallocation and reallocation
    - The memory object will be kicked out of cache
  - Solution: make the free list LIFO (i.e., last-freed first allocated)
- Why LIFO?
  - Last object more likely to be already in cache (hot)
  - Recall from undergrad architecture that it takes quite a few cycles to load data into cache from memory
  - If it is all the same, let's try to recycle the object already in our cache

Performance Issues (5)

- Multi-processor issues:
  - Cache misses due to loss of processor affinity: if deallocated on one processor and allocated on another
  - Cache misses due to false sharing: more on this later
- Solution: per-thread (multiple) heaps can mitigate the problem
  - Cannot completely solve the problem due to thread migration (moving threads between processors)

Hoard: A Scalable Memory Allocator

Let’s put these good ideas to work

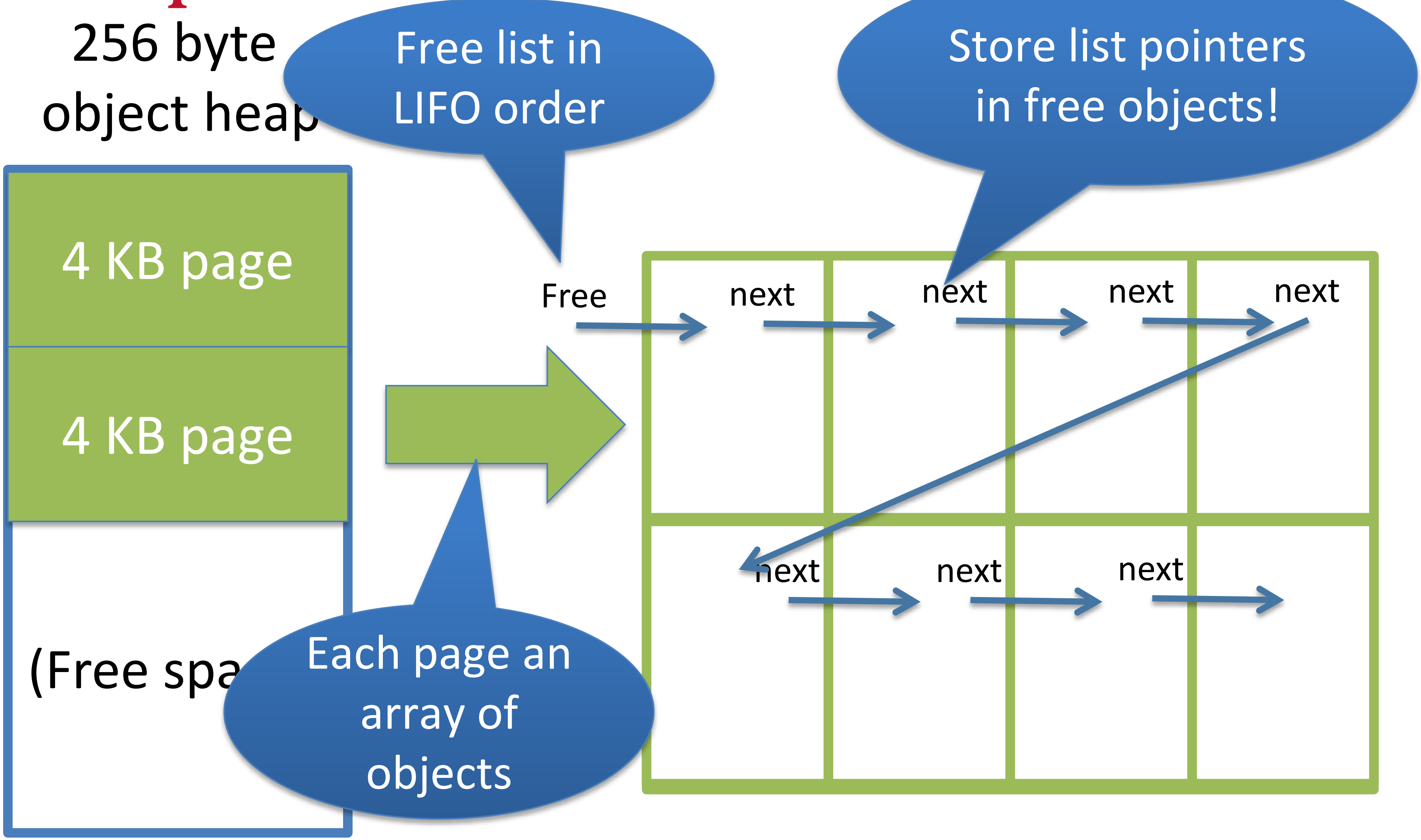
Hoard Superblocks

- Hoard uses a variation of the “segregated pools” idea
- ***Superblock***
  - Chunk of a few (virtually) contiguous pages
    - All superblocks of the same size (say 2 pages)
  - All objects in a superblock are the same size
- A given superblock is treated as an array of same-sized objects
  - Each superblock belongs to a size-class where sizes are “powers of b > 1”;
  - In usual practice, b == 2
- Each superblock has a LIFO list of its free objects

Multi-Processor Strategy

- Allocate a heap for each processor, and one global heap
  - Note: not threads, but CPUs
  - Can only use as many heaps as CPUs at once
  - Requires some way to figure out current processor
    - No such mechanism on x86
    - Read the Hoard paper to figure out how they deal with this
- On `malloc()`
  - Try per-CPU heap first
  - If no free blocks of right size, then try global heap
  - If that fails, get another superblock for per-CPU heap

Superblock intuition



Hoard `malloc(sz)` in Nutshell

- For example, `malloc(7)`
- Round up to next power of 2 (8)
- Find a size-8 superblock with a free object
  - First check the per-CPU heap
  - Then the global heap
- If no free objects, allocate another superblock for the per-CPU heap
  - Initialize by putting all of its objects on the free list
  - Then allocate the first object

Hoard `free()` in a Nutshell

- Return the object to the head of the superblock's LIFO list
- But: how do you tell which superblock an object is from?
  - Suppose superblock size is 8k (2 pages)
    - And always mapped at an address evenly divisible by 8k
  - Object at address 0x431a01c
  - Just mask out the low 13 bits!
  - Came from a superblock that starts at 0x431a000
- Simple math can tell you where an object came from!

  → Hoard doesn't need to keep per-object meta-data header

Superblock Example

- Suppose my program allocates objects of sizes:
  - 5, 8, 13, 15, 34, and 40 bytes.
- How many superblocks do I need
  - Assuming b == 2 and smallest size-class is 8
  - 3 – (8, 16, and 64 byte chunks)
- If I allocate a 5 byte object from an 8 byte superblock, doesn't that yield internal fragmentation?
  - Yes, but it is bounded to < 50% (1/b)
  - Give up some space to bound worst case and complexity

Big Objects in Hoard

- If an object size is bigger than half the size of a superblock, just `mmap()` it
  - Recall, a superblock is on the order of pages already
- What about fragmentation?
  - Example: 4097 byte object (1 page + 1 byte)
  - Argument (preview): More trouble than it is worth
  - Big allocations are much less frequent than the small ones

Simplicity

- The bookkeeping for `malloc()` and `free()` is pretty straightforward
- Per heap: 1 list of superblocks per size class
- Per superblock:
  - Meta-data: size-class, corresponding heap, num free objects, pointer to free list (LIFO), locks, etc.
- Only keep meta-data per superblock (no need for per-object meta-data)
  - On `free()`, when you find the superblock, can get the metadata from there

New Topic: False Sharing

- Cache lines are bigger than words
  - Word: 32-bits or 64-bits
  - Cache line: 64—128 bytes on most CPUs
- Lines are the basic unit at which memory is cached

Cache line

- These objects have nothing to do with each other
  - At program level, private to separate threads
- At cache level, CPUs are fighting for the line

False sharing is BAD

- Leads to pathological performance problems
  - Super-linear slowdown in some cases
- Rule of thumb: any performance trend that is more than linear in the number of CPUs is probably caused by cache behavior
- Strawman solution: round everything up to the size of a cache line
- Thoughts?
  - Wastes too much memory; a bit extreme

Strawman Solution

- Round every allocation up to the size of a cache line
- Thoughts?
  - Wastes too much memory for small objects; a bit extreme

Hoard Strategy (Pragmatic)

- Rounding up to powers of 2 helps
  - Once your objects are bigger than a cache line
- Locality observation: things tend to be used on the CPU where they were allocated
- Always return free to the original heap
  - Remember idea about extra bookkeeping to avoid synchronization: some allocators do this
    - Save locking, but introduce false sharing!
- This only helps to mitigate the problem; in general, it is not the programmer's job to avoid false sharing
  - The allocator does not know the application logic

Linux Kernel Allocators

Kernel Allocators

Three types of dynamic allocators in Linux:

- Big objects (entire pages or page ranges)
  - Just take pages off of the appropriate free list
- Pools of small common kernel objects (e.g., inodes)
  - Uses page allocator to get memory from system
  - Gives out small pieces
- Small arbitrary-size chunks of memory (`kmalloc`)
  - Looks very much like a user-space allocator
  - Uses page allocator to get memory from system

Memory Pools (kmem_cache)

- Each ***pool*** is an array of objects
  - To allocate, take element out of pool
  - Can use bitmap or list to indicate free/used
    - List is easier, but can't pre-initialize objects
- System creates pools for common objects at boot
  - If more objects are needed, have two options
    - Fail (out of resource – reconfigure kernel for more)
    - Allocate another page to expand pool

`kmalloc`: SLAB Allocator

- The default allocator (until 2.6.23) was the slab allocator
- Slab is a chunk of contiguous pages, similar to a superblock in Hoard
- Similar basic ideas, but substantially more complex bookkeeping
  - The slab allocator came first, historically
- 2 groups upset: (guesses who?)
  - Users of very small systems
  - Users of large multi-processor systems

`kmalloc`: SLOB for Small Systems

- Think 4MB of RAM on a small device/phone/etc.
  - Bookkeeping overheads a large percent of total memory
- SLOB: Simple List Of Blocks
  - Just keep a free list of each available chunk and its size
- Grab the first one that is big enough (first-fit algorithm)
  - Split block if leftover bytes
- No internal fragmentation, obviously
- External fragmentation? Yes.
  - Traded for low overheads
  - Worst-case scenario?
    - Allocate fails, phone crashes (don't use in pacemaker)

`kmalloc`: SLUB for Large Systems

- For very large systems, complex bookkeeping gets out of hand (default since 2.6.23)
- SLUB: The Unqueued Slab Allocator
- A much more Hoard-like design
  - All objects of same size from same slab
  - Simple free list per slab
  - Simple multi-processor management
- SLUB status:
  - Outperforms SLAB in many cases
  - Still has some performance pathologies
    - Not universally accepted

Memory Allocation Wrapup

- General-purpose memory allocation is tricky business
  - Different allocation strategies have different trade-offs
  - No one, perfect solution
- Allocators try to optimize for multiple variables:
  - Fragmentation, low false sharing, speed, multi-processor scalability, etc.
- Understand tradeoffs: Hoard vs. Slab vs. SLOB