Memory Allocation

Nima Honarmand

(Based on slides by Don Porter and Mike Ferdman)

Lecture goal

- Understand how 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 does kernel manage physical memory?

Simple algorithm: bump allocator

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

Bump allocator

- Simply “bumps” up the free pointer
- How does free() work? It doesn’t
  - Well, you could try to recycle cells if you wanted, but complicated bookkeeping
- 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
- Allocation and free latency
- Implementation complexity
- Cache behavior
  - False sharing

Fragmentation

- Undergrad review: What is it? Why does it happen?
- 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?

Hoard: Superblocks

- At a high level, allocator operates on superblocks
  - Chunk of (virtually) contiguous pages
  - All objects in a superblock are the same size
- A given superblock is treated as an array of same-sized objects
  - They generalize to "powers of b > 1";
  - In usual practice, b == 2

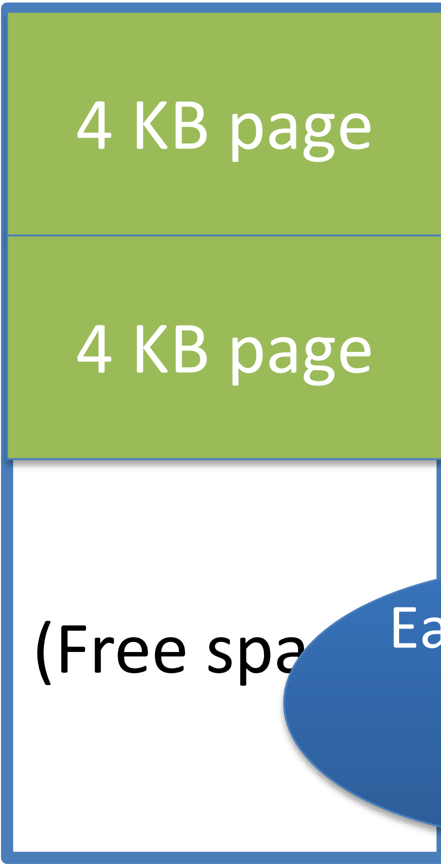
Superblock intuition

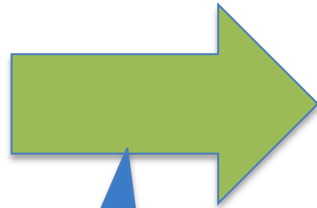
```
malloc (8);
```

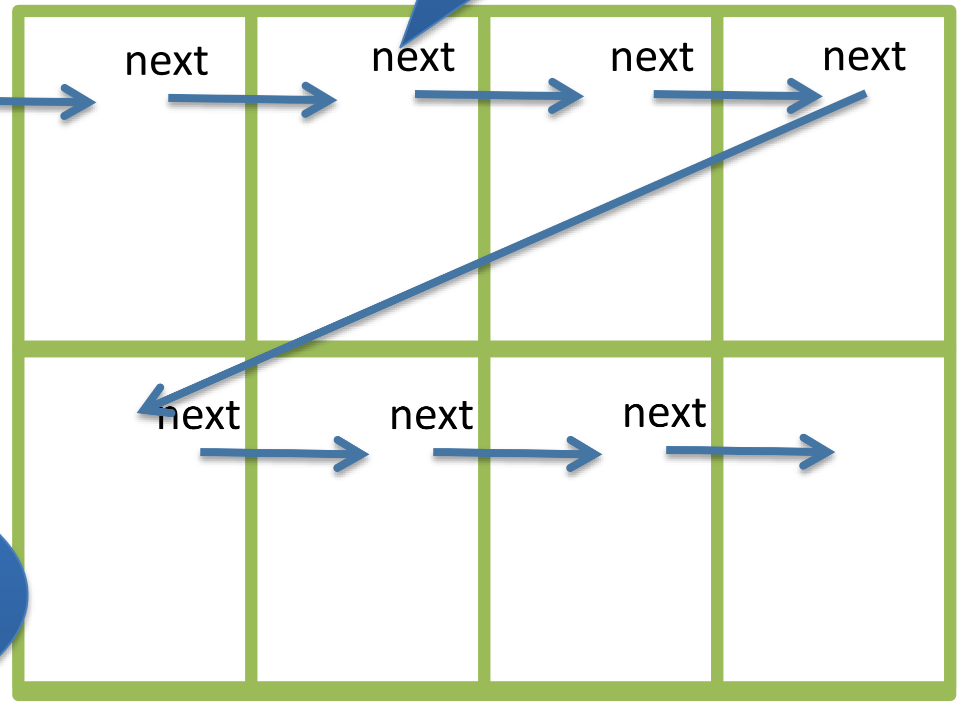
1) Find the nearest power of 2 heap (8)
2) Find free object in superblock
3) Add a superblock if needed. Goto 2.

Superblock intuition

256 byte object heap







Free list in LIFO order

Store list pointers in free objects!

Each page an array of objects

Memory free

- Simple most-recently-used list for a superblock
- How do you tell which superblock an object is from?
  - Suppose superblock is 8k (2pages)
    - 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!

LIFO

- Why are objects re-allocated most-recently used first?
  - Aren't all good OS heuristics FIFO?
  - 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

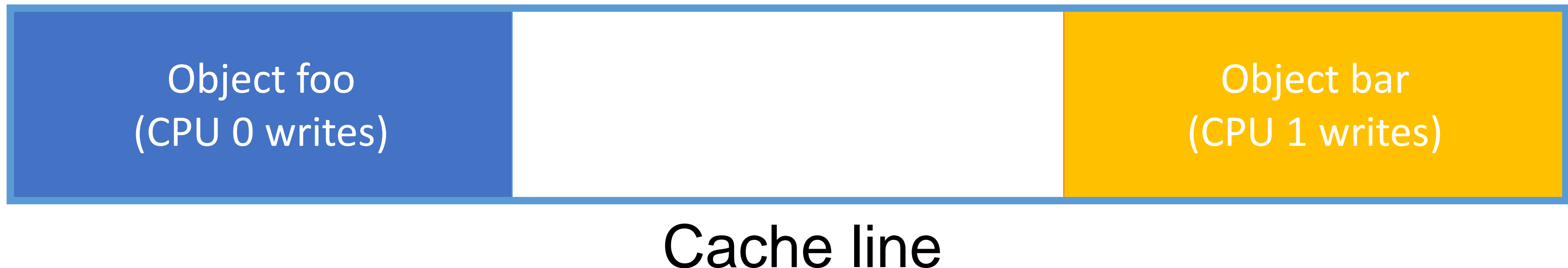
Simplicity

- The bookkeeping for alloc and free is pretty straightforward; many allocators are quite complex (slab)

- Per heap: 1 list of superblocks per object size
- Per superblock:
  - Need to know which/how many objects are free
    - LIFO list of free blocks

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

Other Hoard strategies

- Big object (> page size) allocation?
- Memory allocation for multi-processors?
  - Synchronization issues
- Dealing with false sharing?
- When to return memory to the OS?

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 kernel objects (e.g., VMAs)
  - 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