The Art and Science of (small) Memory Allocation

Don Porter

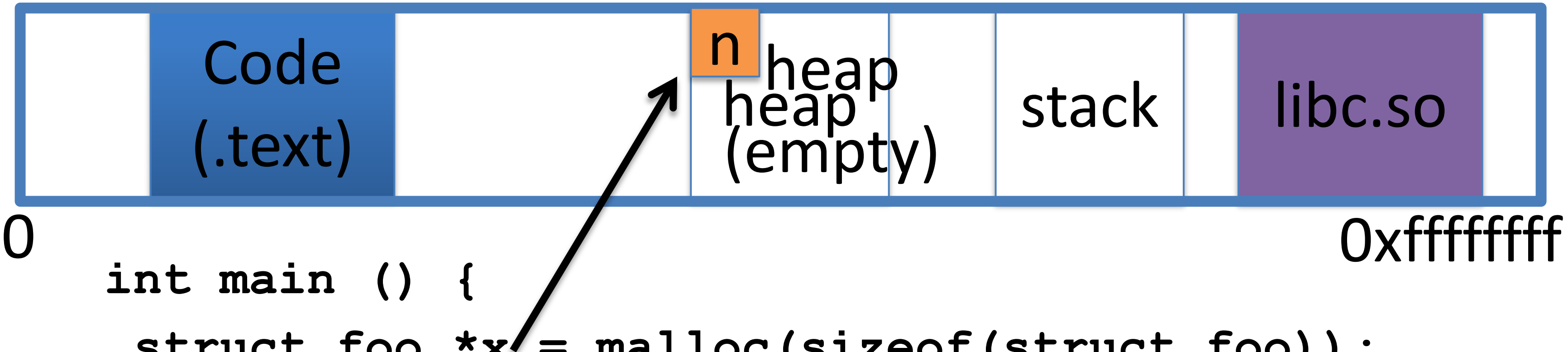
Lecture goal

- This lecture is about allocating small objects
  - Less than one page in size (<4KB)
  - Past lectures have focused on allocating physical pages or segments

- Understand how memory allocators work
- Understand trade-offs and current best practices

Big Picture

Virtual Address Space



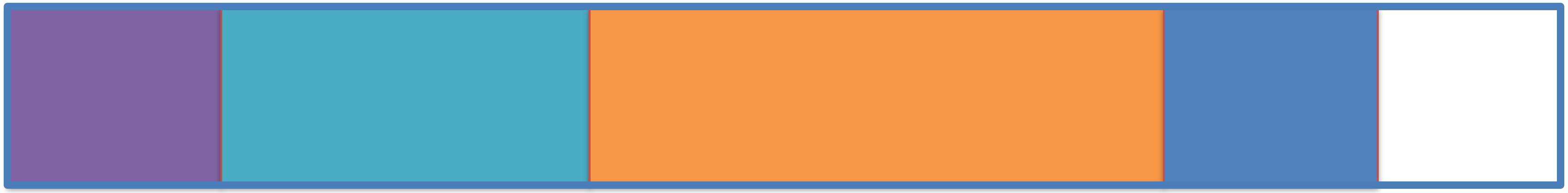
```
int main () {
 struct foo *x = malloc(sizeof(struct foo));
   ...

void * malloc (ssize_t n) {
  if (heap empty)
    mmap(); // add pages to heap
  find a free block of size n;
}
```

Key idea: Sub-divide a page for each malloc() call

Today’s Lecture

- How to implement **`malloc`**() or **`new`**
  - Note that **`new`** is essentially malloc + constructor
  - **`malloc`**() is part of libc, and executes in the application
- **`malloc()`** gets pages of memory from the OS via **`mmap()`** and then sub-divides them for the application
- A brief history of Linux-internal kmalloc implementations

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

Assume memory is limited

- Hoard: best-of-breed concurrent allocator
  - User applications
  - Seminal paper
- Your lab 2 is a simplified version of Hoard
  - No concurrency, no large (>2K) objects, no realloc etc.
- There are other good designs out there
  - jemalloc
  - supermalloc

Overarching issues

- Fragmentation
- Allocation and free latency
- Implementation complexity

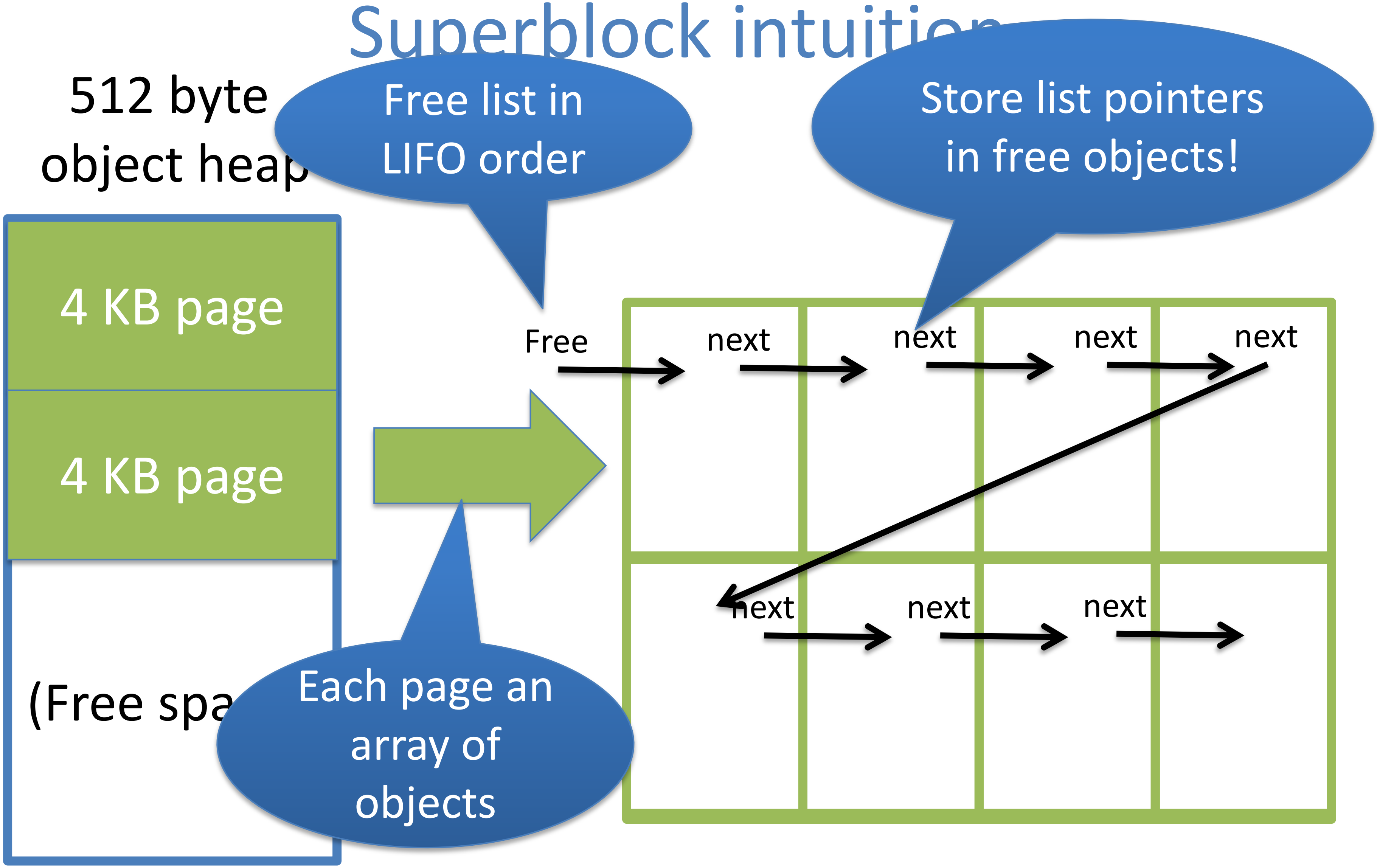
Fragmentation

- 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



512 byte object heap

Free list in LIFO order

Store list pointers in free objects!

Each page an array of objects

Superblock Intuition

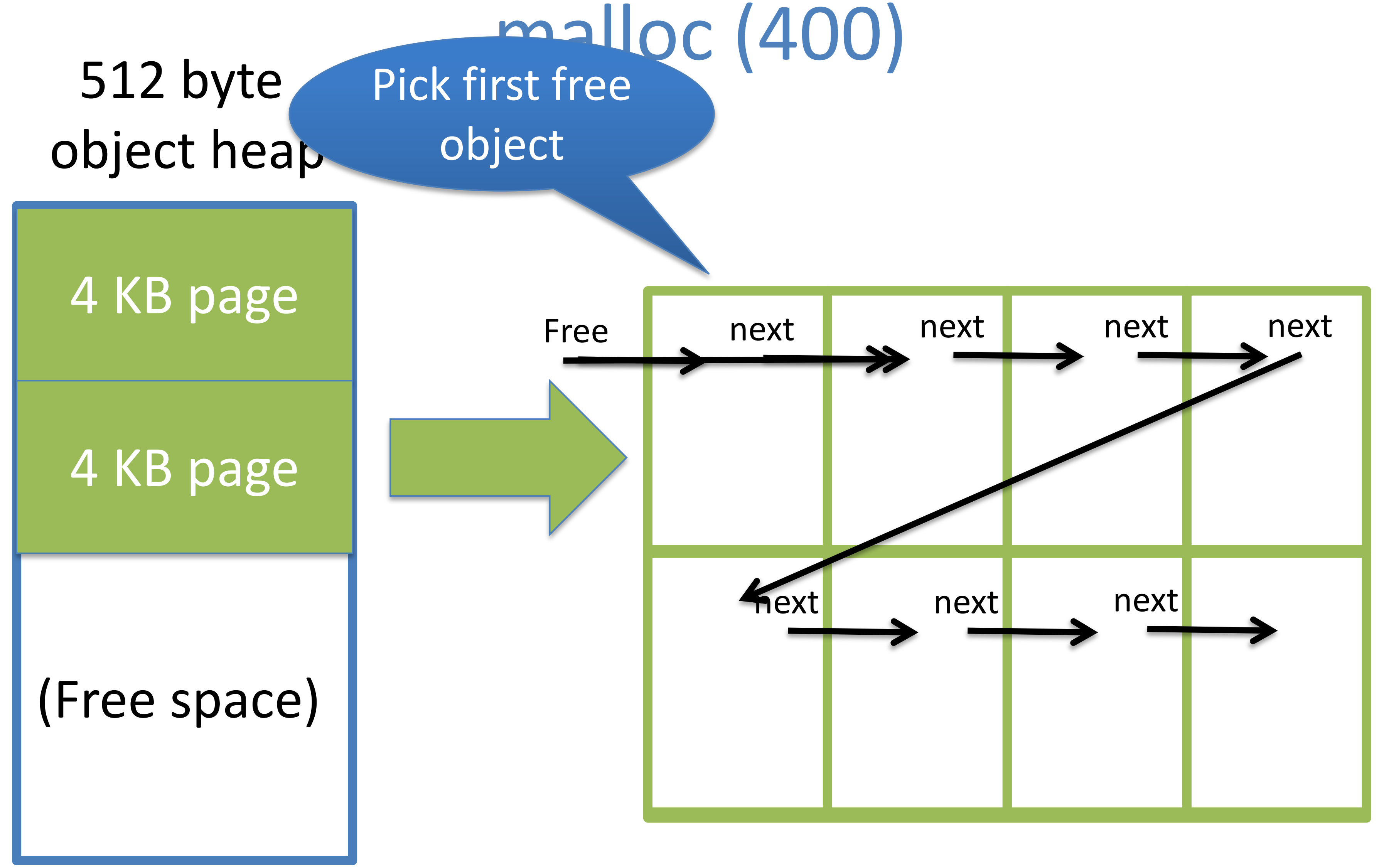
```
malloc (8);
```

1) Find the nearest power of 2 heap (8)

2) Find free object in superblock

3) Add a superblock if needed. Goto 2.

malloc (400)


512 byte object heap
Pick first free object
4 KB page
4 KB page
(Free space)
Free
next
next
next
next
next
next
next

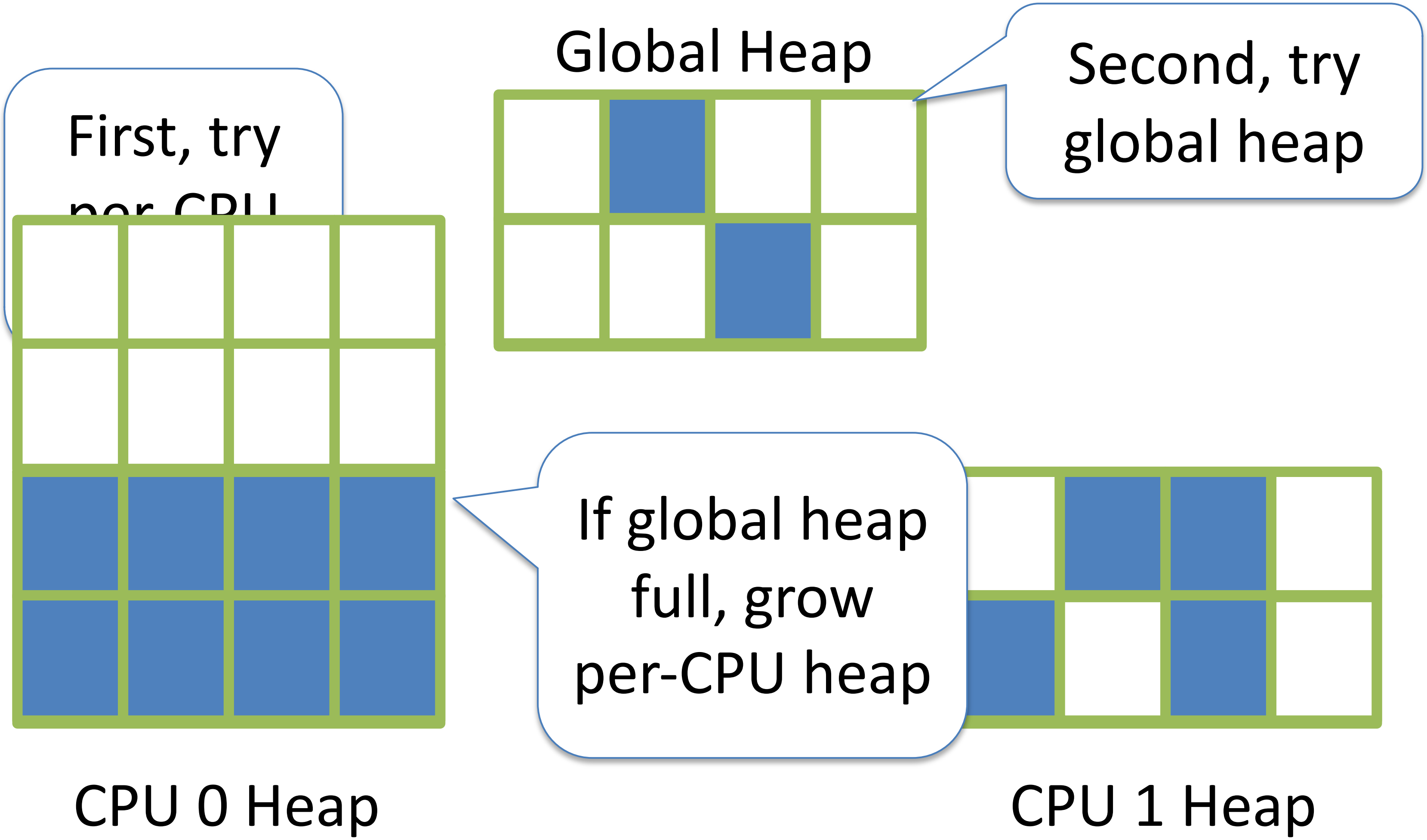
Superblock example

- Suppose my program allocates objects of sizes:
  - 14, 15, 17, 34, and 40 bytes.
- How many superblocks do I need (if b ==2)?
  - 3 – (16, 32, and 64 byte chunks)
- If I allocate a 15 byte object from an 16 byte superblock, doesn’t that yield internal fragmentation?
  - Yes, but it is **bounded to < 50%**
  - Give up some space to bound worst case and complexity

High-level strategy

- Allocate a heap for each processor, and one shared heap
  - Note: not threads, but CPUs
  - Can only use as many heaps as CPUs at once
  - Requires some way to figure out current processor
- Try per-CPU heap first
- If no free blocks of right size, then try global heap
  - Why try this first?
- If that fails, get another superblock for per-CPU heap

Example: malloc() on CPU 0



First, try per CPU

Second, try global heap

If global heap full, grow per-CPU heap

Big objects

- 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: More trouble than it is worth
    - Extra bookkeeping, potential contention, and potential bad cache behavior

Memory free

- Simply put back on free list within its 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

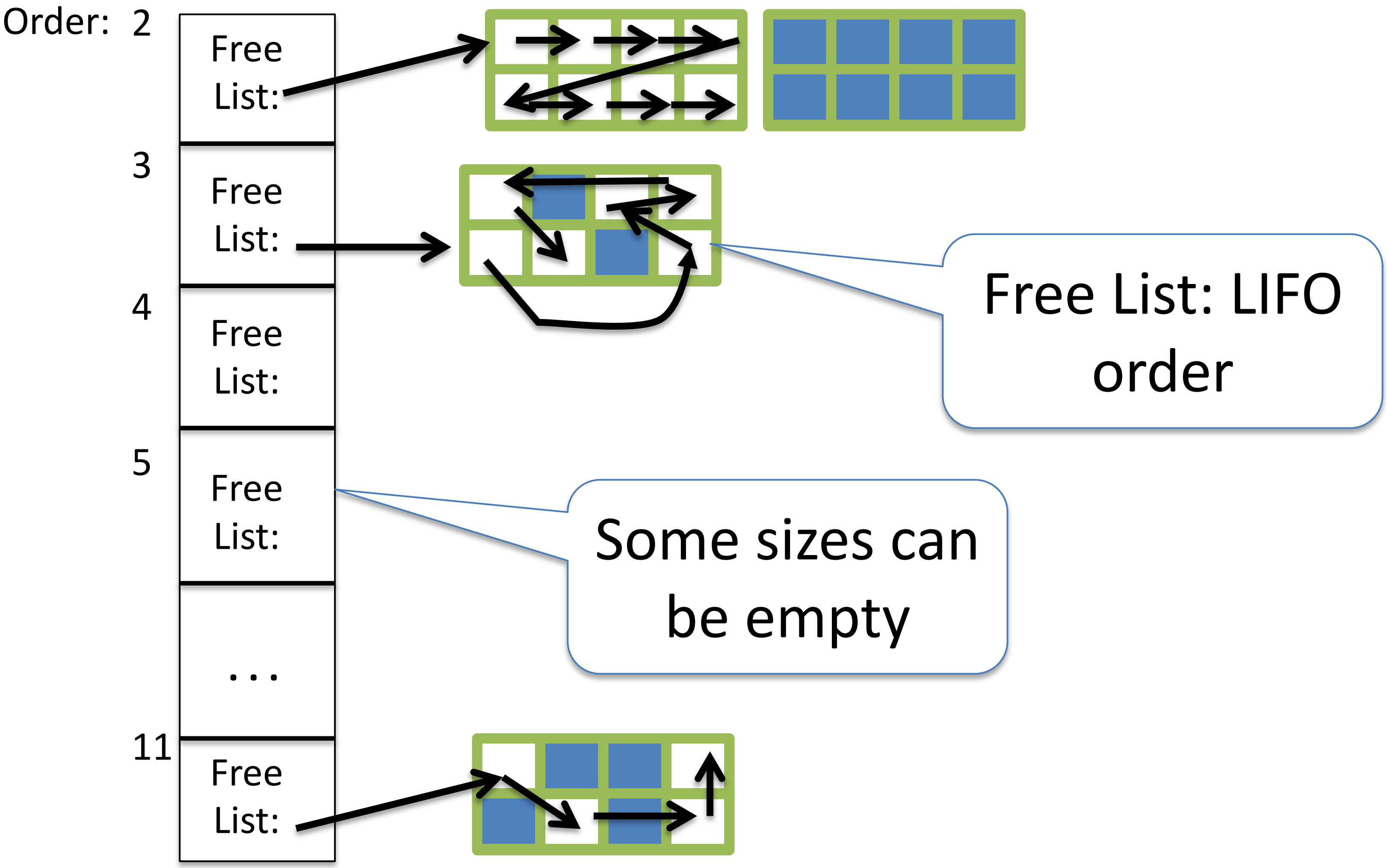
- Why are objects re-allocated most-recently used first?
  - Aren’t all good OS heuristics FIFO?
  - More likely to be already in cache (hot)
  - Recall from undergrad architecture that it takes quite a few cycles to load data into cache from memory
  - If it is all the same, let’s try to recycle the object already in our cache

Hoard Simplicity

- The bookkeeping for alloc and free is straightforward
  - Many allocators are quite complex (looking at you, slab)

- Overall: (# CPUs + 1) heaps
  - Per heap: 1 list of superblocks per object size (2^2—2^{11})
  - Per superblock:
    - Need to know which/how many objects are free
      - LIFO list of free blocks

CPU 0 Heap, Illustrated



Order: 2 Free List:

3 Free List:

4 Free List:

5 Free List:

. . .

11 Free List:

Free List: LIFO order

Some sizes can be empty

One of these per CPU (and one shared)

Hoard summary

- Really nice piece of work
- Establishes nice balance among concerns
- Good performance results
  - It is ok if you don’t understand synchronization and alignment issues

Part 2: Linux kernel allocators

- malloc() and friends, but in the kernel
- Focus today on dynamic allocation of small objects
  - Later class on management of physical pages
  - And allocation of page ranges to allocators

kmem_caches

- Linux has a kmalloc and kfree, but caches preferred for common object types
- Like Hoard, a given cache allocates a specific type of object
  - Ex: a cache for file descriptors, a cache for inodes, etc.
- Unlike Hoard, objects of the same size not mixed
  - Allocator can do initialization automatically
  - May also need to constrain where memory comes from

Caches (2)

- Caches can also keep a certain “reserve” capacity
  - No guarantees, but allows performance tuning
  - Example: I know I’ll have ~100 list nodes frequently allocated and freed; target the cache capacity at 120 elements to avoid expensive page allocation
  - Often called a **memory pool**
- Universal interface: can change allocator underneath
- Kernel has kmalloc and kfree too
  - Implemented on caches of various powers of 2 (familiar?)

Superblocks to slabs

- The default cache allocator (at least as of early 2.6) 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

Complexity backlash

- I'll spare you the details, but slab bookkeeping is complicated
- 2 groups upset: (guesses who?)
  - Users of very small systems
  - Users of large multi-processor systems

Small systems

- Think 4MB of RAM on a small device (thermostat)
- As system memory gets tiny, the bookkeeping overheads become a large percent of total system memory
- How bad is fragmentation really going to be?
  - Note: not sure this has been carefully studied; may just be intuition

SLOB allocator

- Simple List Of Blocks
- Just keep a free list of each available chunk and its size
- Grab the first one big enough to work
  - Split block if leftover bytes
- No internal fragmentation, obviously
- External fragmentation? Yes. Traded for low overheads

Large systems

- For very large (thousands of CPU) systems, complex allocator bookkeeping gets out of hand
- Example: slabs try to migrate objects from one CPU to another to avoid synchronization
  - Per-CPU * Per-CPU bookkeeping

SLUB Allocator

- The Unqueued Slab Allocator
- A much more Hoard-like design
  - All objects of same size from same slab
  - Simple free list per slab
  - No cross-CPU nonsense
- Now the default Linux cache allocator

Conclusion

- Different allocation strategies have different trade-offs
  - No one, perfect solution
- Allocators try to optimize for multiple variables:
  - Fragmentation, speed, simplicity, etc.
- Understand tradeoffs: Hoard vs Slab vs. SLOB
- [Personal note]

Misc notes

- When is a superblock considered free and eligible to be move to the global bucket?
  - See figure 2, free(), line 9
  - Essentially a configurable “empty fraction”
- Is a "used block" count stored somewhere?
  - Not clear, but probably