

The Art and Science of Memory Allocation

Don Porter
CSE 506

Lecture goal

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

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
- ✦ We'll also talk about how Linux allocates its own memory

Overarching issues

- ✦ Fragmentation
- ✦ Allocation and free latency
 - ✦ Synchronization/Concurrency
- ✦ Implementation complexity
- ✦ Cache behavior
 - ✦ Alignment (cache and word)
 - ✦ Coloring

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 superblocks 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 example

- ✦ Suppose my program allocates objects of sizes:
 - ✦ 4, 5, 7, 34, and 40 bytes.
- ✦ How many superblocks do I need (if $b == 2$)?
 - ✦ 3 – (4, 8, 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 $< 50\%$
 - ✦ Give up some space to bound worst case and complexity

Memory free

- ✦ Simple most-recently-used list for a superblock
- ✦ How do you tell which superblock an object is from?
 - ✦ Round address down: suppose superblock is 8k (2pages)
 - ✦ Object at address 0x431a01c
 - ✦ Came from a superblock that starts at 0x431a000 or 0x4319000
 - ✦ Which one? (assume superblocks are virtually contiguous)
 - ✦ Subtract first superblock virtual address and it is the one divisible by two
- ✦ Simple math can tell you where an object came from!

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

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

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
- ✦ If that fails, get another superblock for per-CPU heap

Simplicity

- ✦ The bookkeeping for alloc and free is pretty straightforward; many allocators are quite complex (slab)
- ✦ Overall: Need a simple array of (# CPUs + 1) heaps
- ✦ 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

Locking

- ✦ On alloc and free, even per-CPU heap is locked
- ✦ Why?
 - ✦ An object can be freed from a different CPU than it was allocated on
- ✦ Alternative:
 - ✦ We could add more bookkeeping for objects to move to local superblock
 - ✦ Reintroduce fragmentation issues and lose simplicity

Locking performance

- ✦ Acquiring and releasing a lock generally requires an atomic instruction
 - ✦ Tens to a few hundred cycles vs. a few cycles
- ✦ Waiting for a lock can take thousands
 - ✦ Depends on how good the lock implementation is at managing contention (spinning)
 - ✦ Blocking locks require many hundreds of cycles to context switch

Performance argument

- ✦ Common case: allocations and frees are from per-CPU heap
- ✦ Yes, grabbing a lock adds overheads
 - ✦ But better than the fragmented or complex alternatives
 - ✦ And locking hurts scalability only under contention
- ✦ Uncommon case: all CPUs contend to access one heap
 - ✦ Had to all come from that heap (only frees cross heaps)
 - ✦ Bizarre workload, probably won't scale anyway

Alignment (words)

- ```
struct foo {
 bit x;
 int y;
};
```
- ✦ Naive layout: 1 bit for x, followed by 32 bits for y
  - ✦ CPUs only do aligned operations
    - ✦ 32-bit add expects arguments to start at addresses divisible by 32

## Word alignment, cont.

- ✦ If fields of a data type are not aligned, the compiler has to generate separate instructions for the low and high bits
  - ✦ No one wants to do this
- ✦ Compiler generally pads this out
  - ✦ Waste 31 bits after x
  - ✦ Save a ton of code reinventing simple arithmetic
    - ✦ Code takes space in memory too!

## Memory allocator + alignment

- ✦ Compiler generally expects a structure to be allocated starting on a word boundary
  - ✦ Otherwise, we have same problem as before
  - ✦ Code breaks if not aligned
- ✦ This contract often dictates a degree of fragmentation
  - ✦ See the appeal of  $2^n$  sized objects yet?

## Cacheline alignment

- ✦ Different issue, similar name
- ✦ 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

## Simple coherence model

- ✦ When a memory region is cached, CPU automatically acquires a reader-writer lock on that region
  - ✦ Multiple CPUs can share a read lock
  - ✦ Write lock is exclusive
- ✦ Programmer can't control how long these locks are held
  - ✦ Ex: a store from a register holds the write lock long enough to perform the write; held from there until the next CPU wants it

## False sharing



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 a write lock

## 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

- ✦ Round everything up to the size of a cache line
- ✦ Thoughts?
  - ✦ Wastes too much memory; 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
- ✦ For small objects, 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!

## Hoard strategy (2)

- ✦ Thread A can allocate 2 small objects from the same line
- ✦ "Hand off" 1 to another thread to use; keep using 2<sup>nd</sup>
- ✦ This will cause false sharing
- ✦ Question: is this really the allocator's job to prevent this?

## Where to draw the line?

- ✦ Encapsulation should match programmer intuitions
  - ✦ (my opinion)
- ✦ In the hand-off example:
  - ✦ Hard for allocator to fix
  - ✦ Programmer would have reasonable intuitions (after 506)
- ✦ If allocator just gives parts of same lines to different threads
  - ✦ Hard for programmer to debug performance

## Hoard summary

- ✦ Really nice piece of work
- ✦ Establishes nice balance among concerns
- ✦ Good performance results

## Linux kernel allocators

- ✦ 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/phone/etc.
- ✦ 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

## SLUB status

- ✦ Does better than SLAB in many cases
- ✦ Still has some performance pathologies
  - ✦ Not universally accepted
- ✦ General-purpose memory allocation is tricky business

## Forward pointer

- ✦ Hoard gets more Superblocks via mmap
- ✦ What is the kernel's equivalent of mmap?
  - ✦ Everything we've talked about today posits something that can give us reasonably-sized, contiguous chunks of pages

## Conclusion

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