The Art and Science of Memory Allocation

Don Porter
Today's Lecture
Lecture goal

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

• The next lecture will talk about how the kernel manages physical pages
  – For internal use, or to allocate to applications
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 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, $b = 2$
Free list in LIFO order

Store list pointers in free objects!

Each page an array of objects

Superblock intuition:

256 byte object heap

4 KB page

4 KB page

(Free space)
Superblock Intuition

\texttt{malloc (8);} \\

1) Find the nearest power of 2 heap (8) \\
2) Find free object in superbloc \\
3) Add a superbloc if needed. Goto 2.
malloc (200)

256 byte object heap

Pick first free object

4 KB page

4 KB page

(Free space)
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?
  – 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!
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 superblob 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, superblock and per-CPU heap are 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
How to find the locks?

• Again, page alignment can identify the start of a superblock

• And each superblock keeps a small amount of metadata, including the heap it belongs to
  – Per-CPU or shared Heap
  – And heap includes a lock
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
New topic: alignment

- Word
- Cacheline
struct foo {
    char x;
    int32_t y;
};

• Naïve layout: 1 byte for x, followed by 4 bytes for y
• ISA tools for loading from memory:
  – Load byte
  – Load 4 bytes (starting at address divisible by 4)
  – Load 8 bytes (starting at address divisible by 8)
  – And so on

How to load foo.y in assembly?
How to load foo.y in assembly?

• I’d like to do something like this:
  – `movw %eax, (&foo.y)`

• Problems?
  – Word-aligned mov expects foo.y to be at a word boundary

• I can solve this:
  for i in (0..4)
    Load byte foo.y[i] into eax
    Shift eax left 8 bits

Caveat: Most ISAs (e.g., ARM) only do word-aligned loads.

  x86 implements (slower) unaligned loads in hardware with a similar loop

My solution is obviously undesirable
Word Alignment

```c
struct foo {
    byte x;
    int32_t y;
};
```

- Compiler generally pads this out
  - Waste 24 bits after x
  - Save a ton of code reinventing simple arithmetic
    - Code takes space in memory too!

- Code will still break if foo isn’t aligned to a word boundary!
Memory allocator + alignment

• Compiler and allocator have a contract that `malloc()` and friends will return addresses that are word 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
CPU 0

ldw 0x1008

Cache Miss

Memory Bus

0x1000

RAM

CPU loads one word (4 bytes)

Cache operates at line granularity (64 bytes)
Cache Coherence (1)

CPU 0

Cache

Memory Bus

0x1000

RAM

CPU 1

Cache

ldw 0x1010

Lines shared for reading have a shared lock
Cache Coherence (2)

Lines to be written have an exclusive lock
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

- 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 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
• 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, low false conflicts, speed, multi-processor scalability, etc.

• Understand tradeoffs: Hoard vs Slab vs. SLOB
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