Hoard: A Scalable Memory Allocator
UT Austin and UMass Amherst; ASPLOS 2000

Errata:
- use of `f` parameter is mixed: sometimes it's the fraction of empty blocks sometimes its the fraction of in-use blocks. In the description of figure 1, they should say "f = (1 - 1/4)", not "f = 1/4".
- In 3.1: "As long as a heap is not more than f empty, and has K or fewer superblocks", and should be or.
- In figure 2, lock on global heap should be taken when modifying its stats

Destruct the title:
- Multithreaded Applications: Like what?
- Memory Allocator: What is it? How do they generally work?
- Scalable: What is 'scalability'?
- Why the name hoard?

Concepts in the intro:
- Dynamic memory allocation. Why do we need it? What about static allocation?
- What's a 'serial' memory allocator?
  A: A strange way to refer to a memory allocator intended to be run only on a single thread. This is in contrast to a 'concurrent' memory allocator.
- Why might a concurrent allocator be slower than a serial allocator?

False Sharing
- What is false sharing?
  A: (Physically) share memory without (logically) sharing data. In Hoard, it's sharing a cache-line without sharing data.
- How does this happen?
  A: Two different allocations with words in same cache line
- Why is it so bad to share?
  A. Reads/writes on different processors to the same cache line are effectively synchronized due to cache coherence.

Let's talk about MESI:
- This is a bit of architecture background.
  - When writing OS/system software, knowing your architecture is crucial!
- Modern multicore processors are SMP: Symmetric Multiprocessing
  - Several homogeneous processors sharing memory
- They are also NUMA (Non-uniform memory access)
  - Time to access memory may differ on CPU and memory location.
  - Influenced heavily by cache hierarchies in modern CPUs.
    - Access to local L1 is incredibly cheap. Access to main memory isn't.
- Here's what memory hierachies and timings look like on modern intel CPUs:
  (split cache = cache that is partitioned into two exclusive sections)
  (n-way associative: an address maps to n different slots)

```
Haswell:
* L1/L2 per core, L3 shared.
* L1 Split: Instruction/Data
  L1 Data cache = 32 KB, 64 B/line, 8-WAY.
  L1 Instruction cache = 32 KB, 64 B/line, 8-WAY.
  L1 Data Cache Latency = 4 cycles for simple access via pointer
  L1 Data Cache Latency = 5 cycles for access with complex address calculation
* L2 cache = 256 KB, 64 B/line, 8-WAY
  L2 Cache Latency = 12 cycles
* L3 cache = 8 MB, 64 B/line
  L3 Cache Latency = 36 cycles
* RAM Latency = 36 cycles + 57 ns

Skylake:
* L1/L2 per core, L3 shared.
* L1 Data cache = 32 KB, 64 B/line, 8-WAY.
  L1 Instruction cache = 32 KB, 64 B/line, 8-WAY.
  L1 Data Cache Latency = 4 cycles for simple access via pointer
```

L1 Data Cache Latency = 5 cycles for access with complex address calculation
* L2 cache = 256 KB, 64 B/line, 4-WAY
  L2 Cache Latency = 12 cycles
* L3 cache = 8 MB, 64 B/line, 16-WAY
  L3 Cache Latency = 42 cycles (core 0)
* RAM Latency = 42 cycles + 51 ns

RAM Read B/W (Parallel Random Read) = 5.9 ns / cache line = 10800 MB/s
RAM Read B/W (Read, 16-64 Bytes step) = 26000 MB/s
RAM Read B/W (Read, 32 Bytes step - pointer chasing) = 16500 MB/s
RAM Write B/W (Write, 8 Bytes step) = 17800 MB/s

- Caches are very, very helpful. So we want to use them!
  - But we want to have _shared memory_, so two processors should approximately see the same memory.
- How do we keep these caches coherent (i.e, synchronized)?
  - Via a cache coherence protocol. In this case, MESI.

- In MESI, each cache line is one of the following states:
  Modified (can read/write)
    The cache line is present only in the current cache, and is dirty.
  Exclusive (can read/write)
    The cache line is present only in the current cache, and is clean.
  Shared (read only)
    Indicates that this cache line may be stored in other caches of the machine and is clean.
  Invalid (can be reused)
    Indicates that this cache line is invalid (unused).

- How does this work?
  - Every cache line starts off in "invalid": there’s nothing in the cache.
  - Reads/write requests are issued on the bus via messages by CPUs.
    - Every CPU "snoops" on the bus.
    - Not _all_ reads/writes. For instance, if cache line is in ’E’ state, don’t need to tell anyone else about reads/writes.
  - Anyone can answer requests for read/write.
    - The memory controller, for instance, must answer when all caches are I.
  - Each cache keeps track of its own states.
    - For instance, if cache line is currently ’M’ and a ’RD’ for that cache line is posted to the bus, then that cache will likely set ’M’ to ’S’ and forward to cache line to poster.

- SUMMARY: parallel reads and writes will cause invalidation messages to be sent across the bus. This is expensive.

Why not just pad to cache-line size? How big is a cache-line anyways?
  A: Many reasons.
    1) waste of memory if many objects are less than a cache-line in size
    2) may worsen cache by reducing number of things that fit in it
    3) may worsen cache by reducing spatial locality of objects within cache
    Q: What does this mean?
    A: Well, the processor pulls in things from memory in cache-line sizes. So, if you have two objects on the same cache-line, and you use one after the other, having them next to eachother means we read from RAM once instead of twice.
  A: 64 bytes, but different procesors may need different sizes to avoid false sharing.

What is the different between actively and passively induced false sharing?
- Active:
  - happens when allocator gives allocations from same cache line to different processors
- Passive:
  - free() from one processors followed by malloc() from different processor satisfied using the just freed() memory

```
Can we fully get rid of false sharing (barring cache-line size allocs)?
    A: No.
    1) Program can still induce false sharing by sending object that shares
       cache line with other object in current thread to a different thread
    2) Scheduler might induce it, too. Can move thread sharing cache line with
       another thread on same processor to different processor

Blowup
  - Q: What is Blowup?
       A: a "special" kind of fragmentation caused by being unable to reuse
       freed memory in the future

                          max(mem. allocated from OS)
       blowup = ------------------------------------------------------
                  max(mem. allocated by ideal serial alloc. from OS)

                      (or, slightly easier to follow....)

                   max(mem. allocated from OS)
       blowup = ----------------------------------
                  max(mem. required by program)

  - Q: Why might producer/consumer cause bad blow-up?
    A:

      T1                                        T2

      while (true) {                            while (true) {
        buf = malloc(SIZE);                       buf = recv();
        write(buf, interesting());                do_something_with(buf);
        send(T2, buf);                            free(buf);
      }                                         }

    Unbounded case: free() from T2 isn't useable by T1.
    Factor of P case: per-core heap, no global (!), no returning to global.
      - Need more producers and consumers to show this.
      - alloc from private heap, return to heap allocated from.
      - There are P heaps. Each Ti allocates K blocks, T(i + 1) frees them.
      - This will lead to P * K memory allocated when only need K at a time.
        - Issue is that when K is freed, it stays allocated in per-proc heap.
        - Hoard will return them to global heap.

The Hoard Memory Allocator
  - A detailed description of Hoard
    - An allocator that avoids false sharing
    - Trades increased (but bounded) memory consumption for reduced sync. costs
  - Hoard augments per-processor heaps with a global-heap
    - each per-processor heap only access by one processor: heaps 1 through P
      - in their implementation they actually have 2P heaps so that
        1) threads on the same processor probably don't use same heap
        2) migrated threads don't use the same heap
    - global-heap can be accessed by all: heap 0
  - Have a hash function: f(thread_id) -> per-processor heap
    - multiple threads can run on one processor, so want collisions
    - threads can be reassigned to different processors
  - Hoard maintains usage statistics for each heap i
    - u_i = amount of memory in use (live) by heap i
    - a_i = amount of memory allocated for heap from OS by heap i
  - Hoard allocates memory from system in chunks called superblocks
    - Each is an array of some number of blocks
    - Keeps a free list (linked list in LIFO order) of blocks
    - All superblocks are same size (S = multiple of system page size)
  - Objects larger than half a superblock are allocated using mmap
  - All blocks in a superblock are in same size class
```

    - size classes are a power of $b$ apart ($b > 1$)
    - so could have $b$ sized blocks, $b^2$, $b^3$, etc.
  - Allocations are rounded up to nearest size class
  - Internal fragmentation:
    - How much of a superblock is being wasted
    - Bounded to a factor of $b$ since allocations are rounded up to power of $b$
      - $b^n / b^{(n-1)} == b$
  - External fragmentation:
    - How much of the heap (superblocks entirely) is being wasted
    - To reduce, completely unused superblocks are returned to global heap and can be re-used for any size class

Bounding Blowup

- Each heap is said to own a number of superblocks
- When all owned superblocks are in use, new one is obtained from global heap
  - If global heap is empty, Hoard allocates new superblock from OS
- Unused superblocks in global heap are currently not returned to OS
- Superblocks from the per-processor heap are returned to the global heap
  - ...if the per-processor heap crosses the emptiness threshold

  $(u_i < (1 - f) * a_i)$ AND $(u_i < a_i - K * S)$

  In English: the fraction of free blocks is more than $f * a_i$ and there are more than K superblocks worth of free memory on the heap. In positive terms, say $r_i$ is the number of free memory, so $r_i = (a_i - u_i)$. Then, we have (verified with math):

  $r_i > f * a_i$ AND $r_i > K * S$

  where

  $f$ = empty fraction = parameter of how many blocks should be not in use

  $F$ = parameter set to some number of superblocks that should be not in use

  $u_i$, $a_i$, $S$ = as defined before
- Hoard will not move supers from per-processor heap unless condition is met
- When the condition is met:
  - Hoard transfers a superblock that is >= f empty to global heap
  - Q: Why must this exist? Because $r_i > f * a_i$. So either all blocks are f empty, or at least one block is more than f empty while others may not be f empty.
- This policy maintains the invariant:

  $(u_i >= a_i - K * S)$ OR $(u_i >= (1 - f) * a_i)$

  - This is the logical inverse of the emptiness threshold.
  - Maintaining this invariant is what bounds blowup.
- To find f-empty superblocks in constant time:
  - superblocks divided into fullness bins
  - each bin is a double-linked list of superblocks in some fullness range
  - Hoard maintains those lists appropriately
  - Hoard uses lists to allocate from most full bins, too.
  - Superblock that was last freed from is moved/at the front

    Q: Why?

    A: Idea is to improve locality: concerned about swapping and cache

    Q: What is swapping?

    A: Page memory into/out of disk.

Avoiding False Sharing

- Different threads make allocations from different superblocks, avoiding actively induced false sharing.
- Each allocation is always returned back to the superblock it came from, avoiding passively induced false sharing.
- False sharing isn't entirely avoided
  - Hoard may move superblocks from one heap to another while there are allocations outstanding from that superblock
  - If this happens, and the allocation size is less than a cache-line size, then false sharing can occur.

- Thankfully, this doesn't happen often. Most superblocks moved are empty.

Per-processor heaps AND global heaps

- Is Hoard the first to do this?
  A: Nah. It references Vee and Hsu. They're the first to simultaneously have a global + local heap, allocate + return from/to the 'correct' per-processor heap, and have a precise strategy for when to move superblocks to/from the global heap from/to the correct per-processor heap.
  A scalable and efficient storage allocator on SMP
- What does Hoard do that's different to similar previous work?