TSINGHUA UNIVERSITY

Summary on *Multiprocessor Memory Management*

XINGGAO YANG

LAST EDIT:March 11, 2017

1 ABSTRACT

Several memory allocators have be presented to maintain the performance and scalability of programs on multi-processor systems. Heap is widely used to organize pages in different allocators, and different strategies are used to divide or manage heaps. Three types of allocator is recommended in *Operating System: Three Easy Piece*: Hoard, jmalloc and glibc malloc. In this report, I summarize several fundamental materials from main aspects they are concerned with, and conclude some method and throught they use.

2 PROBLEMS

As it is a multiprocess problem, the contention of memory and cache is the bottleneck of scalability. Nevertheless, the allocator should perform efficiently on single process. Thus, these problem should be solved to design an allocator with good performance[1]:

- **Speed**: Guarantee the performance of multiprocessor allocator on single allocator.
- **Contension**: Maintain the scalability to reduce the contention:
    1. **False sharing**: Several thread try to acquire access to the same cache line simutaneously.
    2. **Heap or Chunk**: A memory region is used by more than one thread. The solution is to use multiple arenas for allocation, and assign threads to arenas. However, different method is implemented to manage the arenas and the chunks inside it.
- **Fragmentation**: Internal fragmentation may lead to unnecessary space consumption. External fragmentation is the cause of low space locality. They will degrade performance severely.
- **Blowup**: It is a concept presented in Hoard only, I believe it is vital but find no other paper mentioned it. It is a behaviour that a thread cannot use the memory region that another thread just free'd, lead to a linear invalid memory consumption.

Another problem is how to show the performance of an allocator, how to test it with some benchmarks or circumstances designed to verifiy specific feature. Measurement of the time consumed by the allocator code in isolation is not sufficient, memory layout can **have significant impact on how quickly the rest of the application runs, due to the effects of CPU cache, RAM, and virtual memory paging.**[2] In other words, the factors to influence memory performance are not isolated, a real world benchmark or application is need to test the behavious of an allocator. Thus, allocator performance is measured via some **combination of application execution time and average or peak application memory usage**. However, an allocator may perform well on some applications, but poorly on specific ones. Testing with a wide variety of applications is important, and some corner cases should be considered when designing an allocator.

3 HOARD

Two kind of heap are used in Hoard: a global heap that every process can access, and per-processor heaps. The memory in a heap is organized in chunks called "superblocks":

1. **Superblock**: Each superblock is an array of objects and contains a free list of its available blocks in LIFO order. All the blocks in a superblock is of the same size. All the superblocks of the same fullness groups are in a doubly-linked list, and the lately used superblock s are in the front of each list, therefore, the locality is maintained and they are more likely to be in the cache.
2. **Algorithms**
    - **Allocation**:When a tread on processor i calls $malloc$, Hoard locks heap i and gets a block of superblock with free space. If not Hoard checks the global heap for a superblock, and transfer it to the processer's heap, or create a new superblock if the global heap is empty.

- **Deallocation**: When a block is free'd, Hoard gets the lock of the superblock the block belongs to, then returns the block and check if the emptiness is under certain threshold. If it is crossing the threshold, Hoard will transfer a superblock under the certain empty fraction from the heap to the global heap, leading to heap transfer maybe, which is quite rare.

4 JMALLOC

Each applications is configured at run-time to have a fixed number of arenas depending on the number of processors in jmalloc.

1. **Arena**: The first time when a thread allocates or deallocates memory, it is assigned to an arena. And the arena is chosen in round-robin fashion, to be guaranteed to have approximately the **same number of threads assigned to them.** It is efficent to reduce contention for a particular arena with round-robin.
2. **Chunk**: The memory is devied into multiples of the "chunk" size, and the address can be easily calculated via the base address and the chunk size. Chunks are classified into three major categories: small, large, and huge. All allocation requests are rounded up into the nearest size class.
   (a) **Huge**: Huge allocations are larger than half of a chunk. Metadata about a huge chunk is stored in a single red-black tree, since it is rare for huge allocations.
   (b) **Small and large**: Buddy algorithm is used to reduce fragmentations. Metadata is stored as a page map at the beginning of each chunk. The size is cut into smaller pieces in small chunks, because most application allocate objects with small size, internal fragmentation will be reduced if the density of sizes less than a page is larger then it is in the large size.
3. **Region bitmap**: Unlike the method of slab, small allocations has a bitmap to handle the search or modification in allocation and deallocation. It is easy to be implement and make the memory region compact which leads to better locality.

5 GLIBC MALLOC

As we can see, the introduction is more complete and more details is showed the allocator involved through 15 years. Glibc will be discussed in detail in this report as it is the most complete one in the three allocators, and widely used as it is in glibc library.
The glibc malloc is derived from ptmalloc (pthreads malloc), which is derived from dlmalloc (Doug Lea malloc)

5.1 FEATURES

- **"heap" style**: chunks of various sizes exist within a larger region of memory;
- **Chunk**: Chunk-oriented, devides a large region of memory(heap) into chunks of various sizes:
  - Each includes metadata about its size and the position of adjacent chunks;
  - A "chunk pointer" or *mchunkptr* does **not** point to the beginning of the chunk, but the last word in the previous chunk;
  - Chunks are adjacent to each other, all chunks **in the heap** can be iterated through by using the size information from the address of the first chunk in a heap;
  - In-used chunks are not tracked by the arena, free chunks are stored in **various lists based on size and history** called **"bins"**;
- **Arena**: A structure shared among one or more threads which contains references to one or more heaps, and the linked lists of chunks within those heaps which are "free":
  - Main arena" uses the application's heap, which owns a static variable in the malloc pointing to it, and other arenas use mmap'd heaps. Each of them has a **next** pointer to link additional arenas;

  - More than one region of memory are active at a time, to reduce contention;
  - Arena is created via syscall *mmap*, the number of arenas is capped at eight times the number of CPUs in the system(modified by *mallopt*). **Heavily threaded application will still see some contention, but the trade-off is that there will be less fragmentation**;
  - Mutex is used to control access to that arena. Some operations, such as **access to the fastbins** can be down with **atomic operations**.

- **Bins**:
  - **Fast**: Designed to make the operations fast. **Small chunks** are stored in **size-specific bins**. Chunks in fastbins are **not** coalesced, and stored in a **singly linked list**.
  - **Unsorted**: Chunks are initially stored in a single bin when they are just free'd. They are later sorted together.
  - **Small** & **Large**: Chunks in small-bins are of the same size, and in a range of size in large-bins. Coalesce with adjascent after chunks added in, and are doubly-linked. Pick the first chunk in small and use it, and find the "best" in large chunk.
  - The first chunk of each size contains the ptr and size information. To achieve better performance, when multiple chunk of a dozen size are present, the *second* is chosen or added, not to adjust the next-size linked list.

5.2 Algorithm

I moved the nutshell in the MallocInternals[3] directly here, because it is the core of this allocator.

- **Malloc**:
  1. If the request is large enough, mmap() is used to request memory directly from the operating system. Note that the threshold for mmap'ing is dynamic, unless overridden by M_MMAP_THRESHOLD (see mallopt() documentation), and there may be a limit to how many such mappings there can be at one time.
  2. If the appropriate fastbin has a chunk in it, use that.
  3. If the appropriate smallbin has a chunk in it, use that.
  4. If the request is "large", take a moment to take everything in the fastbins and move them to the unsorted bin, coalescing them as you go.
  5. Start taking chunks off the unsorted list, and moving them to small/large bins, coalescing as you go (note that this is the only place in the code that puts chunks into the small/large bins). If a chunk of the right size is seen, use that.
  6. If the request is "large", search the appropriate large bin, and successively larger bins, until a large-enough chunk is found.
  7. If we still have chunks in the fastbins (this may happen for "small" requests), consolidate those and repeat the previous two steps.
  8. Split off part of the "top" chunk, possibly enlarging "top" beforehand.

- **Free**
  1. If the chunk is small enough, place it in the appropriate fastbin.
  2. If the chunk was mmap'd, munmap it.
  3. See if this chunk is adjacent to another free chunk and coalesce if it is.
  4. Place the chunk in the unsorted list, unless it's now the "top" chunk.
  5. If the chunk is large enough, coalesce any fastbins and see if the top chunk is large enough to give some memory back to the system. Note that this step might be deferred, for performance reasons, and happen during a malloc or other call.

- **Realloc**
  - MMAP'd chunks:
    Allocations that are serviced via individual mmap calls (i.e. large ones) are realloc'd by mremap() if available, which may or may not result in the new memory being at a different address than the old memory, depending on what the kernel does.
    If the system does not support munmap() and the new size is smaller than the old size, nothing happens and the old address is returned, else a malloc-copy-free happens.
  - Arena chunks:
    1. If the size of the allocation is being reduced by enough to be "worth it", the chunk is split into two chunks. The first half (which has the old address) is returned, and the second half is returned to the arena as a free chunk. Slight reductions are treated as "the same size".
    2. If the allocation is growing, the next (adjacent) chunk is checked. If it is free, or the "top" block (representing the expandable part of the heap), and large enough, then that chunk and the current are merged, producing a large-enough block which can be possibly split (as above). In this case, the old pointer is returned.
    3. If the allocation is growing and there's no way to use the existing/following chunk, then a malloc-copy-free sequence is used.

6 Similarity and Problem Discussion

I need to mention here that a real world test of different allocator on multicore, and the following feature is needed in the test OS:

- Easy to implement or replant another allocator;
- Support major benchmarks and applications;

As we can see, these three allocators envolve from the former one.

- **Contention and False Sharing**: Arena and chunk are used to avoid contention. Usually, in a slab-like memory allocator, the allocator may devide a page into pieces and allocate them to different requests. This method reduces internal fragmentation of a page, but lead to severe cache line contention. By isolate different memory region to threads with arena, false sharing is no more a problem, and contention of a heap is reduced.
- **Speed**: The bottleneck of speed is the management of free space, locality and waiting caused by contention. In glibc malloc, free space list is categorized in a rather detailed way, the speed of small chunks, coalescence and keeping locality. And the locality, is still the problem that will be highlighted in future design.
- **Fragmentation**: Fragmentation is a trade-off problem with performance. Glibc sacrificed some performance to reduce fragmentation, and some really detailed and perfect methods are used.
- **Blowup**: There is no discussion about blowup in glibc, and indeed, glibc malloc has no blowup problem. The released memory region will soon be coalesced and put into the free bins.

References

[1] Berger E D, Mckinley K S, Blumofe R D, et al. Hoard: a scalable memory allocator for multithreaded applications[J]. architectural support for programming languages and operating systems, 2000, 28(5): 117-128.

[2] Evans J. A Scalable Concurrent malloc(3) Implementation for FreeBSD, 2006.

[3] MallocInternals. glibc wiki: sourceware.org/glibc/wiki/MallocInternals, 2016.

[4] Sploitfun. Understanding glibc malloc: https://sploitfun.wordpress.com/2015/02/10/understanding-glibc-malloc/, 2015.