ptmalloc、tcmalloc与jemalloc对比分析

背景介绍

在开发微信看一看期间，为了进行耗时优化，基础库这层按照惯例使用tcmalloc替代glibc标配的ptmalloc做优化，CPU消耗和耗时确实有所降低。但在晚上高峰时期，在CPU刚刚超过50%之后却出现了指数上升，服务在几分钟之内不可用。最终定位到是tcmalloc在内存分配的时候使用自旋锁，在锁冲突严重的时候导致CPU飙升。为了弄清楚tcmalloc到底做了什么，仔细了解各种内存管理库迫在眉睫。

内存管理不外乎三个层面，用户程序层，C运行时库层，内核层。allocator 正是值C运行时库的内存管理模块, 它响应用户的分配请求, 向内核申请内存, 然后将其返回给用户程序。为了保持高效的分配, allocator 一般都会预先分配一块大于用户请求的内存, 并通过某种算法管理这块内存. 来满足用户的内存分配要求, 用户 free 掉的内存也并不是立即就返回给操作系统, 相反, allocator 会管理这些被 free 掉的空闲空间, 以应对用户以后的内存分配要求. 也就是说, allocator 不但要管理已分配的内存块, 还需要管理空闲的内存块, 当响应用户分配要求时, allocator 会首先在空闲空间中寻找一块合适的内存给用户, 在空闲空间中找不到的情况下才分配一块新的内存。业界常见的库包括：ptmalloc(glibc标配)、tcmalloc(google)、jemalloc(facebook)

接下来我们将从两个角度对这些库进行分析：

1. 系统向：看内存管理库是如何管理空闲内存的
2. 用户向：看用户程序如何向内存管理库申请内存(释放大致相似，可以参考申请)

ptmalloc

GNU Libc 的内存分配器(allocator)—ptmalloc，起源于Doug Lea的malloc。由Wolfram Gloger改进得到可以支持多线程。

在Doug Lea实现的内存分配器中只有一个主分配区（main arena），每次分配内存都必须对主分配区加锁，分配完成后释放锁，在SMP多线程环境下，对主分配区的锁的争用很激烈，严重影响了malloc的分配效率。ptmalloc增加了动态分配区（dynamic arena），主分配区与动态分配区用环形链表进行管理。每一个分配区利用互斥锁（mutex）使线程对于该分配区的访问互斥。每个进程只有一个主分配区，但可能存在多个动态分配区，ptmalloc根据系统对分配区的争用情况动态增加动态分配区的数量，分配区的数量一旦增加，就不会再减少了。主分配区在二进制启动时调用sbrk从heap区域分配内存，Heap是由用户内存块组成的连续的内存域。而动态分配区每次使用mmap()向操作系统"批发"HEAP_MAX_SIZE大小的虚拟内存，如果内存耗尽，则会申请新的内存链到动态分配区heap data的"strcut malloc_state"。如果用户请求的大小超过HEAP_MAX_SIZE，动态分配区则会直接调用mmap()分配内存，并且当free的时候调用munmap()，该类型的内存块不会链接到任何heap data。用户向请求分配内存时，内存分配器将缓存的内存切割成小块"零售"出去。从用户空间分配内存，减少系统调用，是提高内存分配速度的好方法，毕竟前者要高效的多。

系统向看ptmalloc内存管理

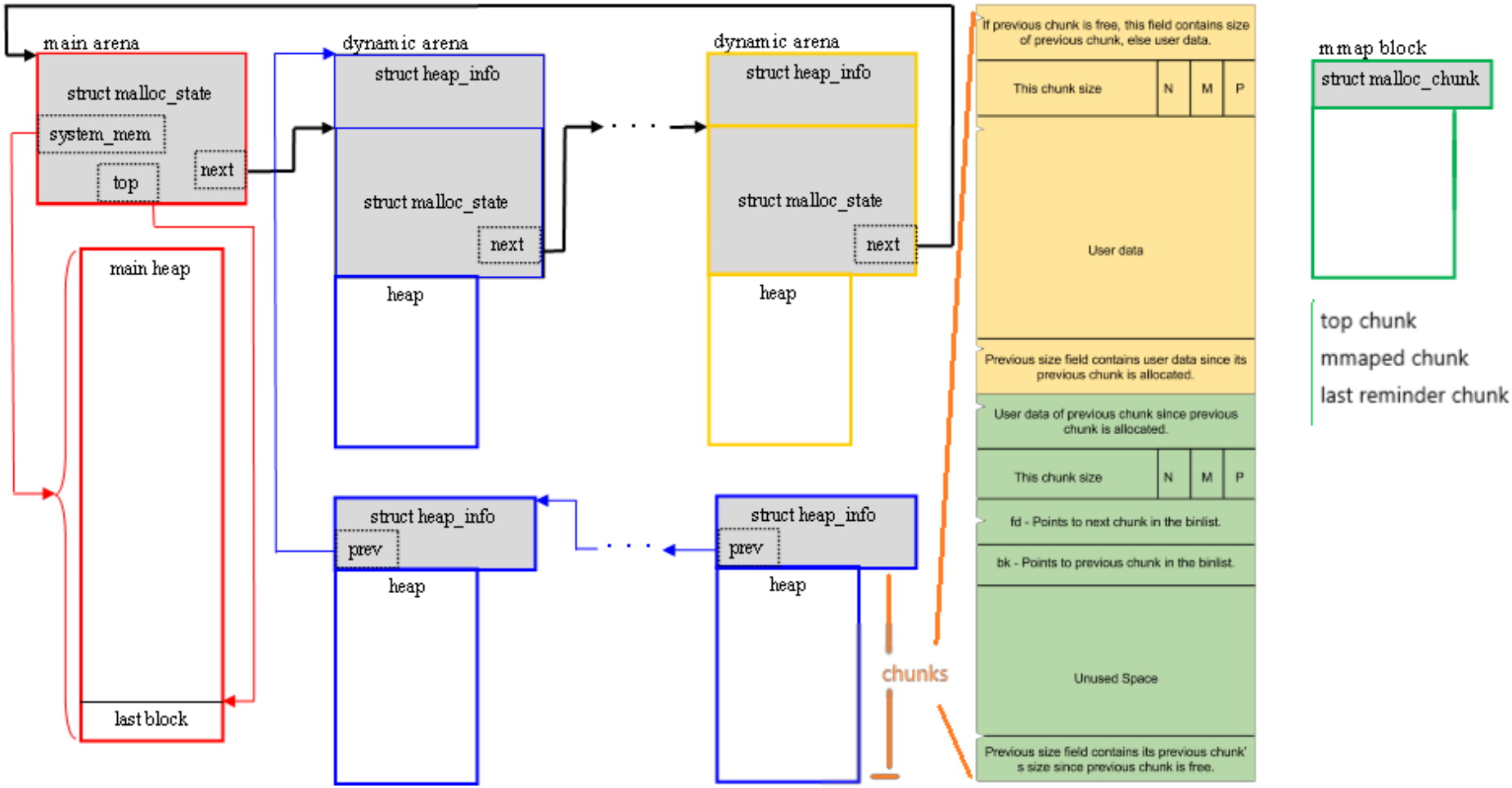
在「glibc malloc」中主要有 3 种数据结构：

- malloc_state(Arena header)：一个 thread arena 可以维护多个堆，这些堆共享同一个arena header。Arena header 描述的信息包括：bins、top chunk、last remainder chunk 等；
- heap_info(Heap Header)：每个堆都有自己的堆 Header（注：也即头部元数据）。当这个堆的空间耗尽时，新的堆（而非连续内存区域）就会被 mmap 当前堆的 aerna 里；
- malloc_chunk(Chunk header)：根据用户请求，每个堆被分为若干 chunk。每个 chunk 都有自己的 chunk header。内存管理使用malloc_chunk，把heap当作link list从一个内存块游走到下一个块。

```
struct malloc_state {
        mutex_t mutex;
        int flags;
        mfastbinptr fastbinsY[NFASTBINS];
        /* Base of the topmost chunk -- not otherwise kept in a bin */
        mchunkptr top;
        /* The remainder from the most recent split of a small request */
        mchunkptr last_remainder;
        /* Normal bins packed as described above */
        mchunkptr bins[NBINS * 2 - 2];
        unsigned int binmap[BINMAPSIZE];
        struct malloc_state *next;
        /* Memory allocated from the system in this arena. */
        INTERNAL_SIZE_T system_mem;
        INTERNAL_SIZE_T max_system_mem;
};

typedef struct _heap_info {
        mstate ar_ptr; /* Arena for this heap. */
        struct _heap_info *prev; /* Previous heap. */
        size_t size; /* Current size in bytes. */
        size_t mprotect_size; /* Size in bytes that has been mprotected
        PROT_READ|PROT_WRITE. */
        /* Make sure the following data is properly aligned, particularly
        that sizeof (heap_info) + 2 * SIZE_SZ is a multiple of
        MALLOC_ALIGNMENT. */
        char pad[-6 * SIZE_SZ & MALLOC_ALIGN_MASK];
} heap_info;
```

```
struct malloc_chunk {
        INTERNAL_SIZE_T prev_size; /* Size of previous chunk (if free). */
        INTERNAL_SIZE_T size; /* Size in bytes, including overhead. */
        struct malloc_chunk* fd; /* double links -- used only if free. */
        struct malloc_chunk* bk;
        /* Only used for large blocks: pointer to next larger size. */
        struct malloc_chunk* fd_nextsize; /* double links -- used only if free. */
        struct malloc_chunk* bk_nextsize;
};
```



注意：Main arena 无需维护多个堆，因此也无需 heap_info。当空间耗尽时，与 thread arena 不同，main arena 可以通过 sbrk 拓展堆段，直至堆段「碰」到内存映射段；

用户向看ptmalloc内存管理

当某一线程需要调用malloc()分配内存空间时，该线程先查看线程私有变量中是否已经存在一个分配区，如果存在，尝试对该分配区加锁，如果加锁成功，使用该分配区分配内存，如果失败，该线程搜索循环链表试图获得一个没有加锁的分配区。如果所有的分配区都已经加锁，那么malloc()会开辟一个新的分配区，把该分配区加入到全局分配区循环链表并加锁，然后使用该分配区进行分配内存操作。在释放操作中，线程同样试图获得待释放内存块所在分配区的锁，如果该分配区正在被别的线程使用，则需要等待直到其他线程释放该分配区的互斥锁之后才可以进行释放操作。

For 32 bit systems:
Number of arena = 2 * number of cores + 1.
For 64 bit systems:
Number of arena = 8 * number of cores + 1.

线程中内存管理

对于空闲的chunk，ptmalloc采用分箱式内存管理方式，每一个内存分配区中维护着[bins]的列表数据结构，用于保存free chunks。根据空闲chunk的大小和处于的状态将其放在四个不同的bin中，这四个空闲chunk的容器包括fast bins，unsorted bin， small bins和large bins。

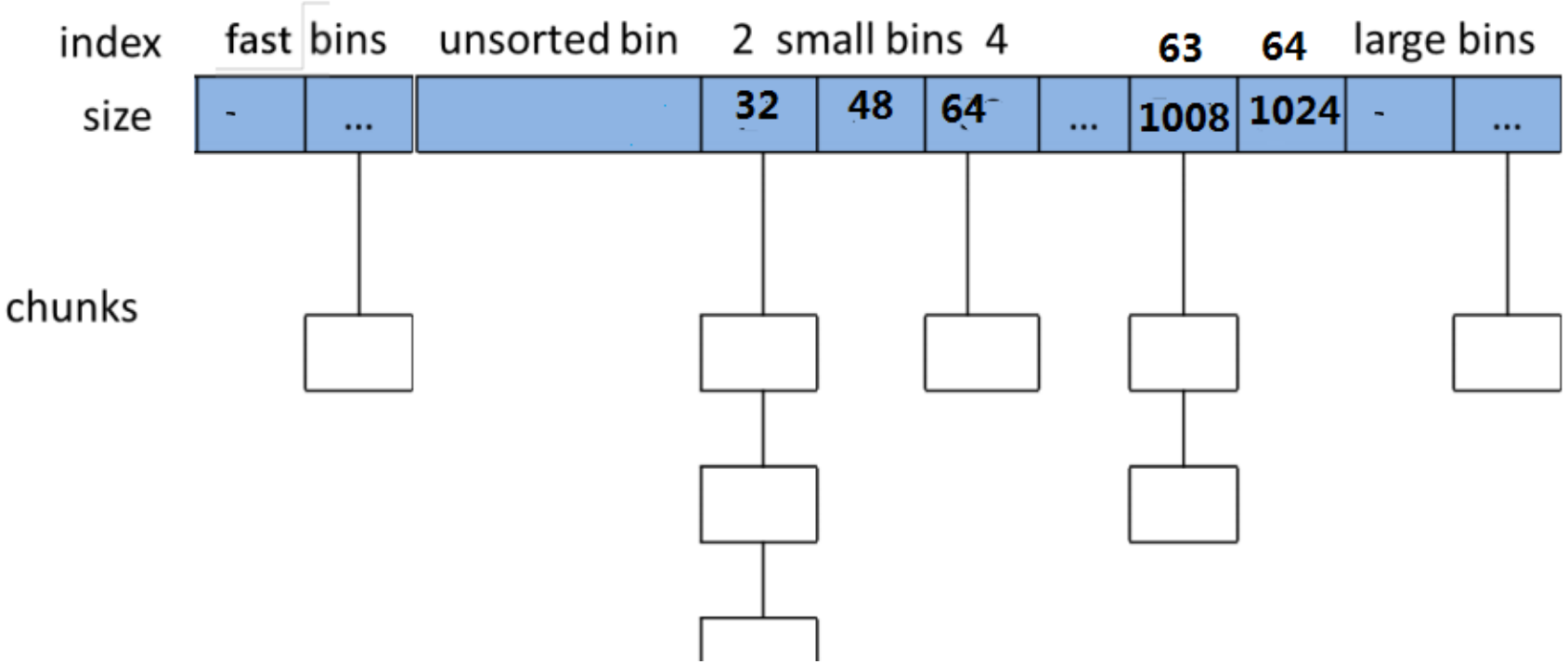
从工作原理来看：

- Fast bins是小内存块的高速缓存，当一些大小小于64字节的chunk被回收时，首先会放入fast bins中，在分配小内存时，首先会查看fast bins中是否有合适的内存块，如果存在，则直接返回fast bins中的内存块，以加快分配速度。
- Usorted bin只有一个，回收的chunk块必须先放到unsorted bin中，分配内存时会查看unsorted bin中是否有合适的chunk，如果找到满足条件的chunk，则直接返回给用户，否则将unsorted bin的所有chunk放入small bins或是large bins中。
- Small bins用于存放固定大小的chunk，共64个bin，最小的chunk大小为16字节或32字节，每个bin的大小相差8字节或是16字节，当分配小内存块时，采用精确匹配的方式从small bins中查找合适的chunk。
- Large bins用于存储大于等于512B或1024B的空闲chunk，这些chunk使用双向链表的形式按大小顺序排序，分配内存时按最近匹配方式从large bins中分配chunk。

从作用来看：

- Fast bins 可以看着是small bins的一小部分cache，主要是用于提高小内存的分配效率，虽然这可能会加剧内存碎片化，但也大大加速了内存释放的速度!
- Unsorted bin 可以重新使用最近 free 掉的 chunk，从而消除了寻找合适 bin 的时间开销，进而加速了内存分配及释放的效率。
- Small bins 相邻的 free chunk 将被合并，这减缓了内存碎片化，但是减慢了 free 的速度；
- Large bin 中所有 chunk 大小不一定相同，各 chunk 大小递减保存。最大的 chunk 保存顶端，而最小的 chunk 保存在尾端；查找较慢，且释放时两个相邻的空闲 chunk 会被合并。

其中fastbins保存在malloc_state结构的fastbinsY变量中，其他三者保存在malloc_state结构的bins变量中。



Chunk说明

一个 arena 中最顶部的 chunk 被称为「top chunk」。它不属于任何 bin 。当所有 bin 中都没有合适空闲内存时，就会使用 top chunk 来响应用户请求。当 top chunk 的大小比用户请求的大小小的时候，top chunk 就通过 sbrk（main arena）或 mmap（thread arena）系统调用扩容。

「last remainder chunk」即最后一次 small request 中因分割而得到的剩余部分，它有利于改进引用局部性，也即后续对 small chunk 的 malloc 请求可能最终被分配得彼此靠近。当用户请求 small chunk 而无法从 small bin 和 unsorted bin 得到服务时，分配

器就会通过扫描 binmaps 找到最小非空 bin。正如前文所提及的，如果这样的 bin 找到了，其中最合适的 chunk 就会分割为两部分：返回给用户的 User chunk 、添加到 unsorted bin 中的 Remainder chunk。这一 Remainder chunk 就将成为 last remainder chunk。当用户的后续请求 small chunk，并且 last remainder chunk 是 unsorted bin 中唯一的 chunk，该 last remainder chunk 就将分割成两部分：返回给用户的 User chunk、添加到 unsorted bin 中的 Remainder chunk（也是 last remainder chunk）。因此后续的请求的 chunk 最终将被分配得彼此靠近。

问题

- 如果后分配的内存先释放，无法及时归还系统。因为 ptmalloc 收缩内存是从 top chunk 开始,如果与 top chunk 相邻的 chunk 不能释放, top chunk 以下的 chunk 都无法释放。
- 内存不能在线程间移动，多线程使用内存不均衡将导致内存浪费
- 每个chunk至少8字节的开销很大
- 不定期分配长生命周期的内存容易造成内存碎片，不利于回收。
- 加锁耗时，无论当前分区有无耗时，在内存分配和释放时，会首先加锁。

从上述来看ptmalloc的主要问题其实是内存浪费、内存碎片、以及加锁导致的性能问题。

> 备注：glibc 2.26(2017-08-02)中已经添加了tcache(thread local cache)优化malloc速度

tcmalloc

tcmalloc是Google开发的内存分配器，在Golang、Chrome中都有使用该分配器进行内存分配。有效的优化了ptmalloc中存在的问题。当然为此也付出了一些代价，按下不表，先看tcmalloc的具体实现。

系统向看tcmalloc内存管理

tcmalloc把8kb的连续内存称为一个页(Page)，可以用下面两个常量来描述：
const size_t kPageShift = 13;
const size_t kPageSize = 1 << kPageShift;
对于一个指针p，p>>kPageShift即是p的页地址。同样的对于一个页地址x，管理的实际内存区间是[x <<kPageShift, (x+1) <<kPageShift)。一个或多个连续的页组成一个Span.对于一个Span，管理的实际内存区间是[start<<kPageShift, (start+length) <<kPageShift)。tcmalloc中所有页级别的操作，都是对Span的操作。PageHeap是一个全局的用来管理Span的类。PageHeap把小于的空闲Span保存在双向循环链表上，而大的span则保存在SET中。保证了所有的内存的申请速度，减少了内存查找。

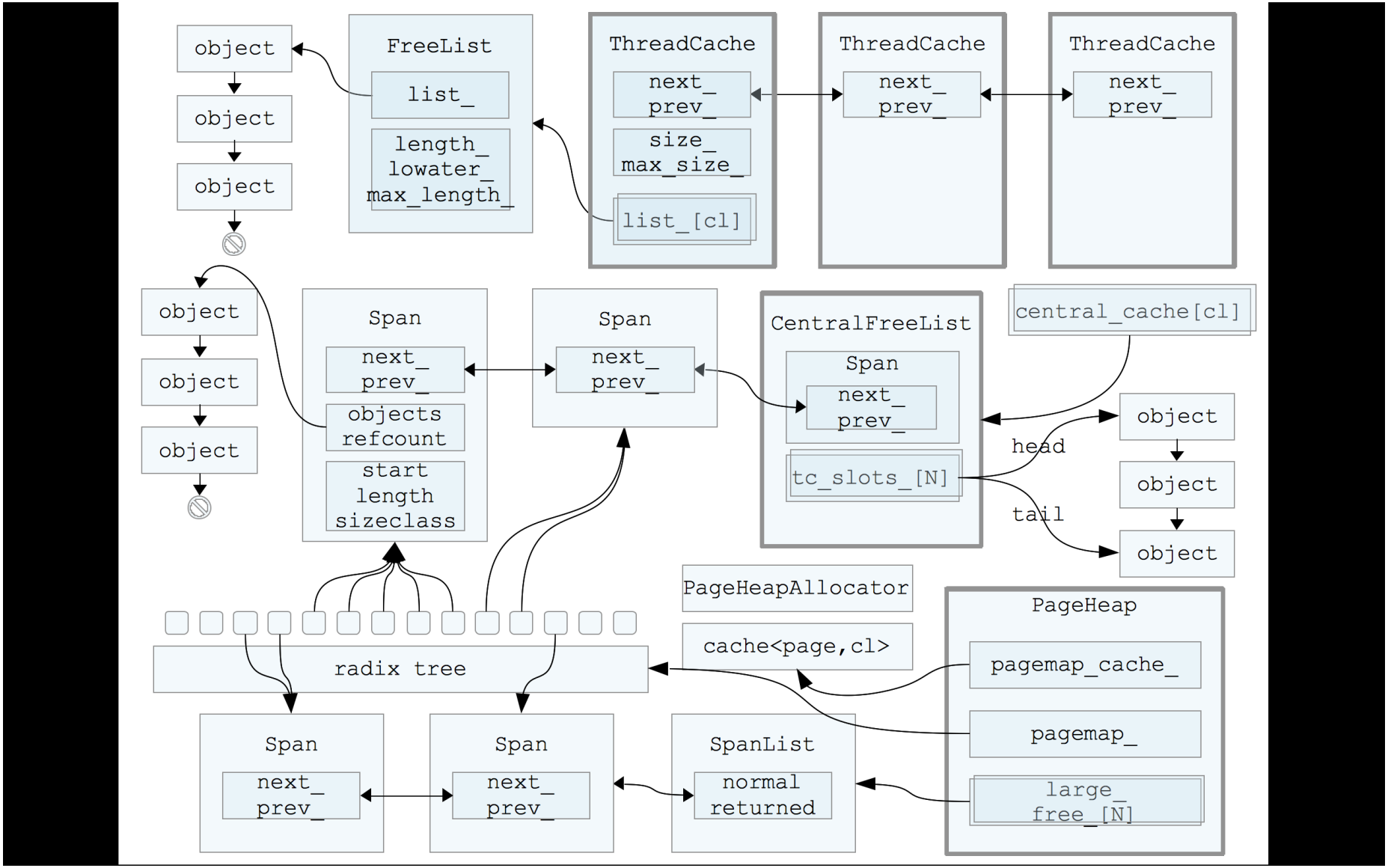
```
// Information kept for a span (a contiguous run of pages).
struct Span {
  PageID        start;          // Starting page number
  Length        length;         // Number of pages in span
  Span*         next;           // Used when in link list
  Span*         prev;           // Used when in link list
  union {
    void* objects;              // Linked list of free objects

    // Span may contain iterator pointing back at SpanSet entry of
    // this span into set of large spans. It is used to quickly delete
    // spans from those sets. span_iter_space is space for such
    // iterator which lifetime is controlled explicitly.
    char span_iter_space[sizeof(SpanSet::iterator)];
  };
  unsigned int  refcount : 16;  // Number of non-free objects
  unsigned int  sizeclass : 8;  // Size-class for small objects (or 0)
  unsigned int  location : 2;   // Is the span on a freelist, and if so, which?
  unsigned int  sample : 1;     // Sampled object?
  bool          has_span_iter : 1; // If span_iter_space has valid
                                   // iterator. Only for debug builds.
  // What freelist the span is on: IN_USE if on none, or normal or returned
  enum { IN_USE, ON_NORMAL_FREELIST, ON_RETURNED_FREELIST };
};

// We segregate spans of a given size into two circular linked
// lists: one for normal spans, and one for spans whose memory
// has been returned to the system.
struct SpanList {
Span        normal;
Span        returned;
};

// Array mapping from span length to a doubly linked list of free spans
//
// NOTE: index 'i' stores spans of length 'i + 1'.
SpanList free_[kMaxPages];
```

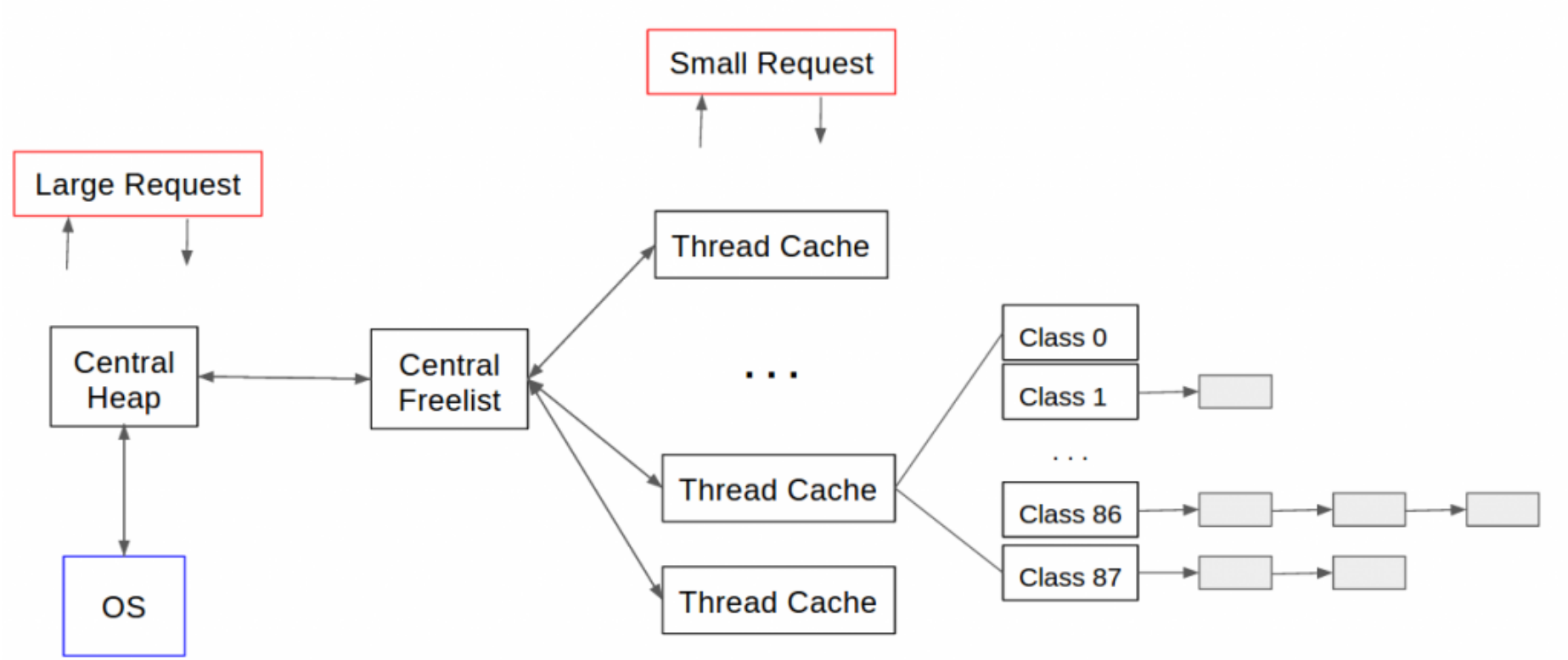
```
  // Sets of spans with length > kMaxPages.
  //
  // Rather than using a linked list, we use sets here for efficient
  // best-fit search.
  SpanSet large_normal_;
  SpanSet large_returned_;
```



用户向看tcmalloc内存管理

TCMalloc是专门对多线并发的内存管理而设计的，TCMalloc主要是在线程级实现了缓存，使得用户在申请内存时大多情况下是无锁内存分配。整个 TCMalloc 实现了三级缓存，分别是ThreadCache(线程级缓存)，Central Cache(中央缓存：CentralFreeeList)，

PageHeap(页缓存)，最后两级需要加锁访问。如图为内存分配



每个线程都一个线程局部的 ThreadCache，ThreadCache中包含一个链表数组FreeList list_[kNumClasses]，维护了不同规格的空闲内存的链表；当申请内存的时候可以直接根据大小寻找恰当的规则的内存。如果ThreadCache的对象不够了，就从 CentralCache 进行批量分配；如果 CentralCache 依然没有，就从PageHeap申请Span；PageHeap首先在free[n,128]中查找、然后到large set中查找，目标就是找到一个最小的满足要求的空闲Span，优先使用normal类链表中的Span。如果找到了一个Span，则尝试分裂(Carve)这个Span并分配出去；如果所有的链表中都没找到length>=n的Span，则只能从操作系统申请了。Tcmalloc一次最少向系统申请1MB的内存，默认情况下，使用sbrk申请，在sbrk失败的时候，使用mmap申请。

当我们申请的内存大于kMaxSize(256k)的时候，内存大小超过了ThreadCache和CenterCache的最大规格，所以会直接从全局的PageHeap中申请最小的Span分配出去(return span->start << kPageShift))；

tcmalloc的优势

- 小内存可以在ThreadCache中不加锁分配(加锁的代价大约100ns)
- 大内存可以直接按照大小分配不需要再像ptmalloc一样进行查找
- 大内存加锁使用更高效的自旋锁
- 减少了内存碎片

然而，tcmalloc也带来了一些问题，使用自旋锁虽然减少了加锁效率，但是如果使用大内存较多的情况下，内存在Central Cache或者Page Heap加锁分配。而tcmalloc对大小内存的分配过于保守，在一些内存需求较大的服务（如推荐系统），小内存上限过低，当请求量上来，锁冲突严重，CPU使用率将指数暴增。

jemalloc

jemalloc是facebook推出的，目前在firefox、facebook服务器、android 5.0 等服务中大量使用。 jemalloc最大的优势还是其强大的多核/多线程分配能力. 以现代计算机硬件架构来说, 最大的瓶颈已经不再是内存容量或cpu速度, 而是多核/多线程下的lock contention(锁竞争). 因为无论CPU核心数量如何多, 通常情况下内存只有一份. 可以说, 如果内存足够大, CPU的核心数量越多, 程序线程数越多, jemalloc的分配速度越快。

系统向看jemalloc内存管理

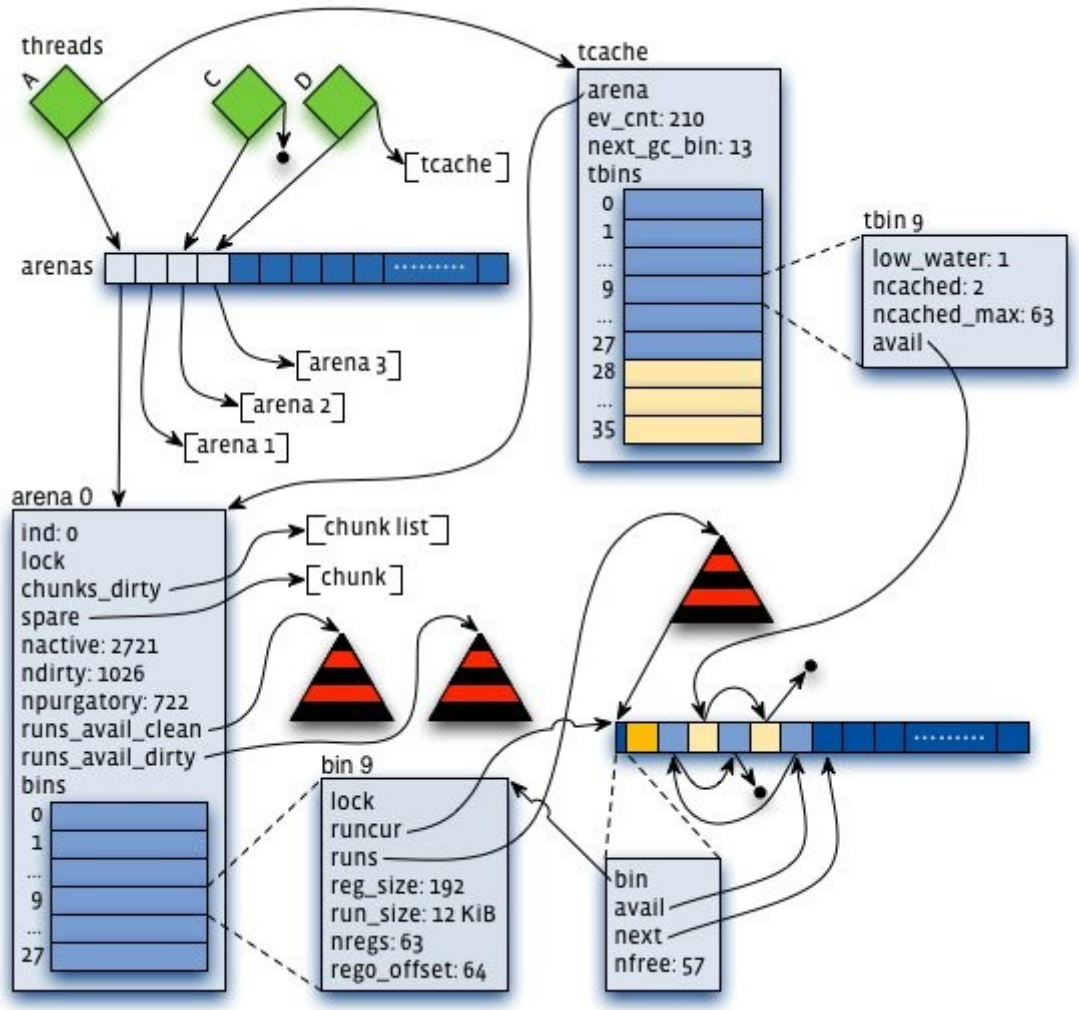
对于一个多线程+多CPU核心的运行环境, 传统分配器中大量开销被浪费在lock contention和false sharing上, 随着线程数量和核心数量增多, 这种分配压力将越来越大.针对多线程, 一种解决方法是将一把global lock分散成很多与线程相关的lock. 而针对多核心, 则要尽量把不同线程下分配的内存隔离开, 避免不同线程使用同一个cache-line的情况.按照上面的思路, 一个较好的实现方式就是引入arena.将内存划分成若干数量的arenas, 线程最终会与某一个arena绑定.由于两个arena在地址空间上几乎不存在任何联系, 就可以在无锁的状态下完成分配. 同样由于空间不连续, 落到同一个cache-line中的几率也很小, 保证了各自独立。由于arena的数量有限, 因此不能保证所有线程都能独占arena, 分享同一个arena的所有线程, 由该arena内部的lock保持同步.

chunk是仅次于arena的次级内存结构，arena都有专属的chunks, 每个chunk的头部都记录了chunk的分配信息。chunk是具体进行内存分配的区域，目前的默认大小是4M。chunk以page（默认为4K)为单位进行管理，每个chunk的前几个page（默认是6个）用于存储chunk的元数据，后面跟着一个或多个page的runs。后面的runs可以是未分配区域， 多个小对象组合在一起组成run, 其元数据放在run的头部。 大对象构成的run, 其元数据放在chunk的头部。在使用某一个chunk的时候，会把它分割成很多个run，并记录到bin中。不同size的class对应着不同的bin，在bin里，都会有一个红黑树来维护空闲的run，并且在run里，使用了bitmap来记录了分配状态。此外，每个arena里面维护一组按地址排列的可获得的run的红黑树。

```
struct arena_s {
    ...
    /* 当前arena管理的dirty chunks */
    arena_chunk_tree_t  chunks_dirty;
    /* arena缓存的最近释放的chunk, 每个arena一个spare chunk */
    arena_chunk_t       *spare;
    /* 当前arena中正在使用的page数. */
    size_t          nactive;
    /*当前arana中未使用的dirty page数*/
    size_t          ndirty;
    /* 需要清理的page的大概数目 */
    size_t          npurgatory;


    /* 当前arena可获得的runs构成的红黑树， */
    /* 红黑树按大小/地址顺序进行排列。 分配run时采用first-best-fit策略*/
    arena_avail_tree_t  runs_avail;
    /* bins储存不同大小size的内存区域 */
    arena_bin_t     bins[NBINS];
};
/* Arena chunk header. */
struct arena_chunk_s {
    /* 管理当前chunk的Arena */
    arena_t         *arena;
    /* 链接到所属arena的dirty chunks树的节点*/
    rb_node(arena_chunk_t)  dirty_link;
    /* 脏页数 */
    size_t          ndirty;
    /* 空闲run数 Number of available runs. */
    size_t          nruns_avail;
    /* 相邻的run数，清理的时候可以合并的run */
    size_t          nruns_adjac;
    /* 用来跟踪chunk使用状况的关于page的map, 它的下标对应于run在chunk中的位置，通过加map_bias不跟踪
chunk 头部的信息
     * 通过加map_bias不跟踪chunk 头部的信息
     */
```

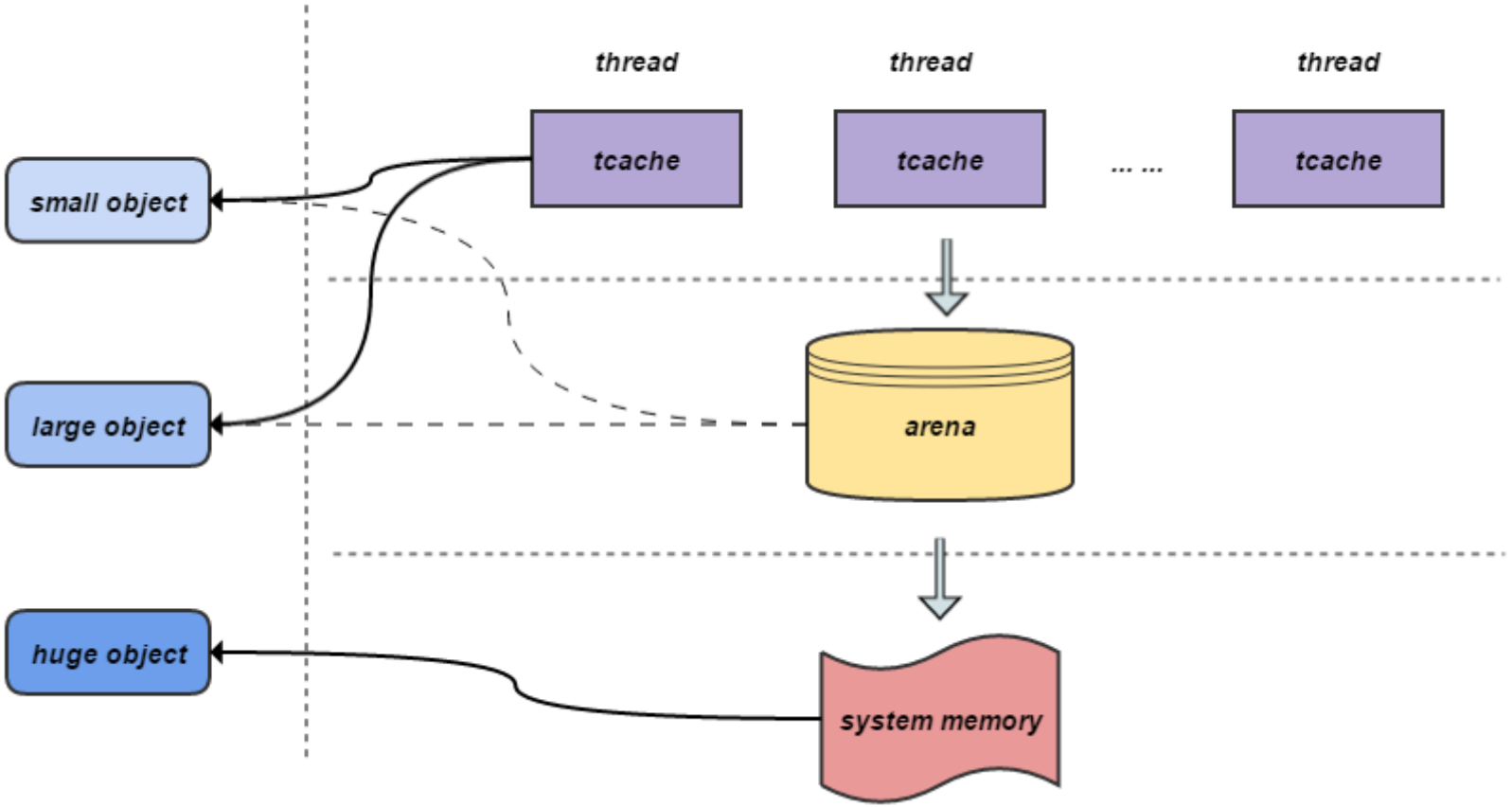
```
    arena_chunk_map_t   map[1]; /* Dynamically sized. */
};
struct arena_run_s {
    /* 所属的bin */
    arena_bin_t *bin;
    /*下一块可分配区域的索引 */
    uint32_t    nextind;
    /* 当前run中空闲块数目. */
    unsigned    nfree;
};
```



用户向看jemalloc内存管理

jemalloc 按照内存分配请求的尺寸，分了 small object (例如 1 – 57344B)、 large object (例如 57345 – 4MB)、 huge object (例如4MB以上)。jemalloc同样有一层线程缓存的内存名字叫tcache，当分配的内存大小小于tcache_maxclass时，jemalloc会首先在tcache的small object以及large object中查找分配，tcache不中则从arena中申请run，并将剩余的区域缓存到tcache。若arena找不到合适大小的内存块， 则向系统申请内存。当申请大小大于tcache_maxclass且大小小于huge大小的内存块时，则直接从arena开始分配。而huge object的内存不归arena管理， 直接采用mmap从system memory中申请，并由一棵与arena独立的红黑树进行管

理。



jemalloc的优势

- 多线程下加锁大大减少

总结

总的来看，作为基础库的ptmalloc是最为稳定的内存管理器，无论在什么环境下都能适应，但是分配效率相对较低。而tcmalloc针对多核情况有所优化，性能有所提高，但是内存占用稍高，大内存分配容易出现CPU飙升。jemalloc的内存占用更高，但是在多核

多线程下的表现也最为优异。

看一看后台系统遇到的问题最终通过链接jemalloc得到了解决，内存管理库的短板和优势其实也给我们带来了一些思考点，在什么情况下我们应该考虑好内存分配如何管理：

- 多核多线程的情况下，内存管理需要考虑内存分配加锁、异步内存释放、多线程之间的内存共享、线程的生命周期
- 内存当作磁盘使用的情况下，需要考虑内存分配和释放的效率，是使用内存管理库还是应该自己进行大对象大内存的管理。（在搜索以及推荐系统中尤为突出）

参考链接：

[0]:http://mqzhuang.iteye.com/blog/1005909

[1]:https://paper.seebug.org/papers/Archive/refs/heap/glibc%E5%86%85%E5%AD%98%E7%AE%A1%E7%90%86ptmalloc%E6%BA%90%E4%BB%A3%E7%A0%81%E5%88%86%E6%9E%90.pdf

[2]:http://core-analyzer.sourceforge.net/index_files/Page335.html

[3]:https://blog.csdn.net/maokelong95/article/details/51989081

[4]:https://zhuanlan.zhihu.com/p/29216091

[5]:https://zhuanlan.zhihu.com/p/29415507

[6]:https://blog.csdn.net/zwleagle/article/details/45113303

[7]:http://game.academy.163.com/library/2015/2/10/17713_497699.html

[8]:https://www.cnblogs.com/taoxinrui/p/6492733.html

[9]:https://blog.csdn.net/vector03/article/details/50634802

[10]:http://brionas.github.io/2015/01/31/jemalloc%E6%BA%90%E7%A0%81%E8%A7%A3%E6%9E%90-%E5%86%85%E5%AD%98%E7%AE%A1%E7%90%86/

本文作者：cyningsun
本文地址：https://www.cyningsun.com/07-07-2018/memory-allocator-contrasts.html

← Older Newer →