Five100Miles

窝素一株喵，从不猫猫猫~

昵称： Five100Miles
园龄： 6年7个月
粉丝： 73
关注： 0
+加关注

< 2024年9月 >

日	一	二	三	四	五	六
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	1	2	3	4	5
6	7	8	9	10	11	12

常用链接

我的随笔
我的评论
我的参与
最新评论
我的标签

随笔分类

LLVM(20)
读书心得(3)
日常点滴(22)
随笔(1)
源码笔记(10)

随笔档案

2021年6月(1)
2021年1月(2)
2020年12月(3)
2020年11月(1)

mimalloc源码笔记

mimalloc是微软去年6月开源的(竟然拖了半年才开始写笔记)新的内存分配器, 其最早由Daan Leijen开发, 为Koka与Lean等语言的runtime system提供内存管理.
根据官方文档描述, mimalloc在各类benchmark上性能均优于其它主流allocator(分别超过tcmalloc与jemalloc 7%与14%)且消耗更少内存.
mimalloc当前版本为1.2.2, 在这里可以获取源码与文档.

1. 编译与使用

mimalloc使用cmake作为构建工具, 执行以下命令即可编译.
成功编译后在构建目录下会生成所需(libmimalloc)的动态库/静态库以及测试工具.

```
mkdir -p [dir to build]
cd [dir to build]
cmake [source path]
make
```

可以通过在源文件中包含mimalloc.h并增加-lmimalloc选项来使用mimalloc.
mimalloc可以与其它allocator共存, 对于cmake构建可以使用以下命令设置:
target_link_libraries([program] PUBLIC mimalloc)
如果不希望重新编译也可以通过设置环境变量来替换:
env LD_PRELOAD=[path/to/libmimalloc.so] [program]

如果需要打印libmalloc的调试信息, 可以设置环境变量(注意后者需要debug版本):
env MIMALLOC_VERBOSE=1 [program]
env MIMALLOC_SHOW_STATS=1 [program]

其它选项:
打印错误/告警信息: MIMALLOC_SHOW_ERRORS=1
使用huge page特性: MIMALLOC_LARGE_OS_PAGES=1

尝试编译一个测试用例:

```
[23:18:20] hansy@hansy:~$ cat 1.c
#include <sys/time.h>
int main() {
  struct timeval last;
  struct timeval next;
  gettimeofday(&last, 0);
  for (int i = 0; i < 10000000; ++i) {
    int *p = malloc(i * sizeof(int));
    free(p);
  }
  gettimeofday(&next, 0);
  printf("%llu.%06llu\n",
    (ne                                    tv_sec - last.tv_sec : next.tv_sec - 1 - last
    (n                                     v_usec - last.tv_usec : 1000000 + next.tv_us
  retu
}
[23:18:25] hansy@hansy:~$ gcc 1.c -w && ./a.out
```

2020年10月(3)
2020年6月(3)
2020年5月(6)
2020年4月(1)
2020年3月(1)
2020年1月(1)
2019年12月(2)
2019年7月(1)
2019年6月(2)
2019年4月(2)
2018年10月(2)
2018年6月(2)
2018年5月(2)
2018年4月(4)
2018年3月(3)
2018年2月(14)

评论排行榜
1. LLVM笔记(3) - PASS(4)
2. LLVM笔记(16) - IR基础详解(一) underlying class(3)
3. LLVM笔记(10) - 指令选择(二) lowering (3)
4. 写在2020的尾巴(2)
5. LLVM笔记(20) - AntiDepBreaker(1)

推荐排行榜
1. LLVM笔记(13) - 指令选择(五) select(6)
2. LLVM笔记(10) - 指令选择(二) lowering (5)
3. LLVM笔记(9) - 指令选择(一) 概述(4)
4. LLVM笔记(5) - SMS(4)
5. 写在2020的尾巴(3)

```
8.606776
[23:18:42] hansy@hansy:~$ env LD_PRELOAD=./mimalloc/build_release/libmimalloc.so ./a.
1.314598
[23:18:57] hansy@hansy:~$ env MIMALLOC_VERBOSE=1 LD_PRELOAD=./mimalloc/build_release/
mimalloc: process init: 0x7f4b0d558740
mimalloc: option 'large_os_pages': 0
mimalloc: option 'secure': 0
mimalloc: option 'page_reset': 0
mimalloc: option 'cache_reset': 0
1.323777
mimalloc: option 'show_stats': 0
heap stats:     peak      total      freed       unit      count
   elapsed:     1.324 s
   process: user: 1.299 s, system: 0.012 s, faults: 0, reclaims: 479, rss: 2.6 mb
mimalloc: process done: 0x7f4b0d558740
```

2. 设计思想

建议首先阅读technical report, 配合代码可以有更好的理解. 这里是原文链接.
以下是论文摘要:
mimalloc的设计背景
现代allocator需要平衡各类需求, 包括性能, 安全, 并行化以及其它应用程序需求的特性. 在开发一个用于Koka与Lean的runtime system时微软遇到两类情况:
一是存在许多short lived的小对象分配, 一个定制的allocator表现能优于jemalloc等主流通用allocator.
二是这些runtime system都使用RC机制来管理内存对象. 在回收较大的数据结构时为减少pause需要推迟减少引用计数. 为获取最佳性能我们需要allocator的协助: 在面临内存压力时回复之前推迟的decrement.

为解决以上问题, 微软引出了free list sharding(分片空闲链表)设计.
传统allocator往往根据内存块大小分类(size-class)管理free list, 一类大小对应一个链表. 其好处是分配给定大小内存可以达到O(1), 缺点是程序局部性较差, 同一数据结构的内存块散布在整个堆上.
为改善局部性mimalloc修改了链表设计, 先将堆被分为一系列(用于分配不同大小内存块)的页(mimalloc page, 当前设定64K), 每页通过一个free list管理. 那么每类大小的内存分配接口可以实现如下:

```
struct block_t { struct block_t *next; }
void *malloc_by_size(page_t *page, int size) {
  if (block_t *block = page->free) {
    page->free = block->next;
    page->used++;
    return block;
  }
  return malloc_generic(size); // slow path
}
```

对于swift与python等使用RC机制的语言释放较大的数据结构会导致递归调用free, 影响程序性能. 对此通常可以限制free调用次数并将剩余指针加入deferred decrement list. 问题在于何时重新开始释放? 就像kernel一样, 只有面临内存压力时再释放是最优解, 因此这需要allocator配合. 在malloc_generic(slow path)中mimalloc会调用用户定义的deferred_free回调.
然而假如用户在一页内来回分配释放内存导致slow path永远不被调用怎么办? 因此mimalloc再次为free list分片: 每页增加一个local free list. 当释放对象时将其放入local free list, 保证free list最终会变空. 在分配接口中再将local free list赋值给free list回收使用.
在mimalloc中page归属于线程堆, 线程仅从本地堆中分配内存, 但其它线程同样可以释放本线程的堆. 为避免引入锁, mimalloc再为每页增加一个thread free list用于记录由其它线程释放(本线程申请)的内存, 当非z ... e->thread_free, p)将其放入thread free list中.

```
void atomic_push(block_t **list, block_t *block) {
```

```
  do {
    block->next = *list;
  } while (!atomic_compare_and_swap(list, block, block->next));
}
```

thread free list的设计不光减少了线程对本页的竞争, 同样减少了线程对不同页的竞争. 类似于local free list, thread free list的回收也在分配接口中.

这样设计的优点:

分期释放大块数据结构, 减少停顿

维持确定的心跳, 保证RC

提升thread free list的一致性

lock free

传统allocator聚焦于减少内存使用, 加速分配/释放或多线程扩展, mimalloc证明了提升程序内存局部性同样可以提升allocator性能.

3. 源码分析

mimalloc源码非常小(仅3.5k loc), 建议阅读源码配合本文.

先来看下内存组织结构, 类似于ptmalloc中malloc_chunk -> bin -> malloc_state, mimalloc也是三级架构mi_block_t -> mi_page_t -> mi_heap_s.

```
typedef uintptr_t mi_encoded_t;
typedef struct mi_block_s {
  mi_encoded_t next;
} mi_block_t;
```

mimalloc中内存块的最小单位是mi_block_t, 区别于ptmalloc中malloc_chunk复杂的结构, mi_block_t只有一个指向(同样大小的)下一空闲内存块的指针.

这是因为在mimalloc中所有内存块都是size classed page中分配的, 不需要对空闲内存块做migrate, 因此不用保存本块大小, (物理连续的块的)状态及大小等信息.

```
typedef union mi_page_flags_u {
  uint16_t value;
  struct {
    bool has_aligned;
    bool in_full;
  };
} mi_page_flags_t;
typedef struct mi_page_s {
  // 该页在段(segment)中的索引, page = &segment->pages[page->segment_idx]
  // 在段分配时初始化(mi_segment_alloc), 用于计算该页指向的实际内存地址(segment + idx * mi_se
  uint8_t segment_idx;
  // 标记该页是否已使用(已分配内存)
  bool segment_in_use:1;
  // 复位标记(什么时候复位?)
  bool is_reset:1;

  // 页属性标记
  mi_page_flags_t flags;
  // 该页当前内存块的容量
  // 这里有几个概念:
  // 页的大小 - 根据页的类型(SMALL / MEDIUM / LARGE)决定
  // 页的最大分配内存块数量 - 即reserved, 根据(页大小 / 该页管理的块大小)决定
  // 页的当前管理内存块数量 - 即capacity, 由于管理所有块需要赋值mi_block_t导致RSS膨胀, 因此空闲
  // 页的当前分配内存块数量 - 即used
  // 页的实际活动内存块数量 - used - thread_free - local_free
  uint16_t capacity;
  // 保留未申请的内存所能分配的内存块的个数, 在mi_page_init中初始化, 为page_size / mi_page_t->
  uint16 t reserved;

  // f                              n        从该链表获取空闲内存块
  mi_b
  // 随机cookie, 用于安全特性
```

```
  uintptr_t cookie;
  // 引用计数, 注意该计数同样将locl_free与thread_free包含在内
  size_t used;
  // 申请内存的线程释放内存时挂入该链表, 只有在free耗尽时才将其赋值给free
  mi_block_t *local_free;
  volatile uintptr_t thread_freed;
  // 类似于local_free, 但是只有其它线程释放内存时挂入该链表, 使用cas解决竞争
  volatile mi_thread_free_t thread_free;

  // 该页管理的内存块的大小, 一个页只管理一种大小的内存块
  size_t block_size;
  // 页所属哪个堆
  mi_heap_t *heap;
  // 双向链表记录(归属于本堆的)管理同一大小内存块的所有页
  struct mi_page_s *next;
  struct mi_page_s *prev;
} mi_page_t;
```

mimalloc的二级结构mi_page_t前文已经描述很多了, 其它的成员见注释.

```
typedef struct mi_tld_s mi_tld_t;
typedef struct mi_page_queue_s {
  mi_page_t *first;
  mi_page_t *last;
  size_t block_size;
} mi_page_queue_t;
struct mi_heap_s {
  // thread local data
  mi_tld_t *tld;
  // LRU链表数组, 每个元素存放分配8-1024字节(MI_SMALL_WSIZE_MAX为128)大小内存块的页用于加速分配
  // 若元素为空说明分配对应大小内存块的空闲页不存在, 需要先回收内存或分配新页
  // _mi_heap_get_free_small_page确保返回值不会大于MI_SMALL_WSIZE_MAX*sizeof(void*),为什么
  mi_page_t *pages_free_direct[MI_SMALL_WSIZE_MAX + 2];
  // 页队列数组, 保存所有用于分配小于512K内存块的页, 最后一个元素保存已满的页
  // 数组中每个元素与保存的页大小是非线性对应的:
  // 8-64字节每个大小各占一个元素(一共8个)
  // 之后每4个元素管理的大小区间翻倍(512K / 8 = 2**16, 16 << 2 = 64个元素)
  // 连续4个元素均分该大小区间
  mi_page_queue_t pages[MI_BIN_FULL + 1];
  volatile mi_block_t *thread_delayed_free;
  uintptr_t thread_id;
  uintptr_t cookie;
  uintptr_t random;
  size_t page_count;
  bool no_reclaim;
};
```

三级结构mi_heap_s用于描述一个堆. 每个堆对应一个线程, 但线程可能不止一个堆.

堆结构的初始化: 分为主堆与线程堆. 为防止在某些平台上访问thread local data会分配内存导致死循环, 主堆是静态分配的数据结构(尽管仍然会调用mi_process_init/mi_process_done做初始化/去初始化), 而线程堆通过mi_thread_init(被mi_malloc_generic调用)初始化.

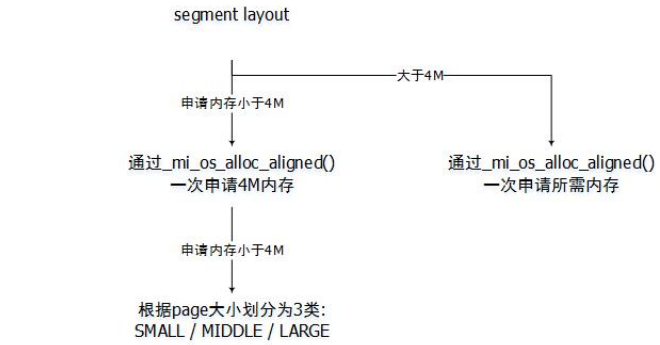
```
typedef struct mi_segment_s {
  // 双向链表用来管理mi_segment_t
  struct mi_segment_s *next;
  struct mi_segment_s *prev;
  struct mi_segment_s *abandoned_next;
  size_t abandoned;
  // 该段内已使用的页计数
  size_t used;
  // 该段容纳页的最大数量, mi_segment_alloc中初始化, 对于HUGE页固定为1, 其它为段大小(MI_SEGMEN
  size_t capacity;
  // 段大小, 对于HUGE页大小为实际申请长度加管理字节及对齐, 对于其它页为4M
  // 可以通过mi_segment_size获取segment_size与segment_info_size
  size
  // 管[illegible]构[illegible]ity * mi_page_t结构)
  // 对[illegible]的[illegible]第一个页空间共用空间, 因此第一个页实际可用地址需要额外
  // 如开启安全选项, 管理头会额外占用一个os页
```

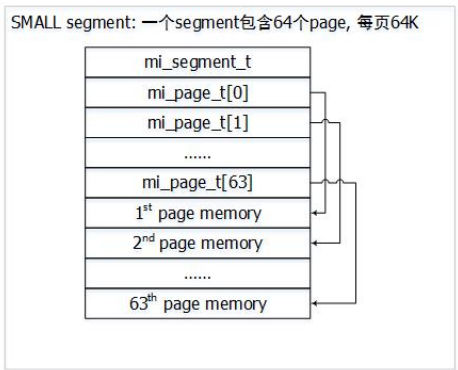
```
  size_t segment_info_size;
  uintptr_t cookie;

  // 记录页大小对应的偏移(MI_LARGE_PAGE_SHIFT / MI_MEDIUM_PAGE_SHIFT / MI_SMALL_PAGE_SHI
  size_t page_shift;
  uintptr_t thread_id;
  // 页类型, 与大小相关
  mi_page_kind_t page_kind;
  // 变长数组, 实际是capacity个元素的mi_page_t结构, 与mi_segment_t结构一起占据第一个页
  mi_page_t pages[1];
} mi_segment_t;
```

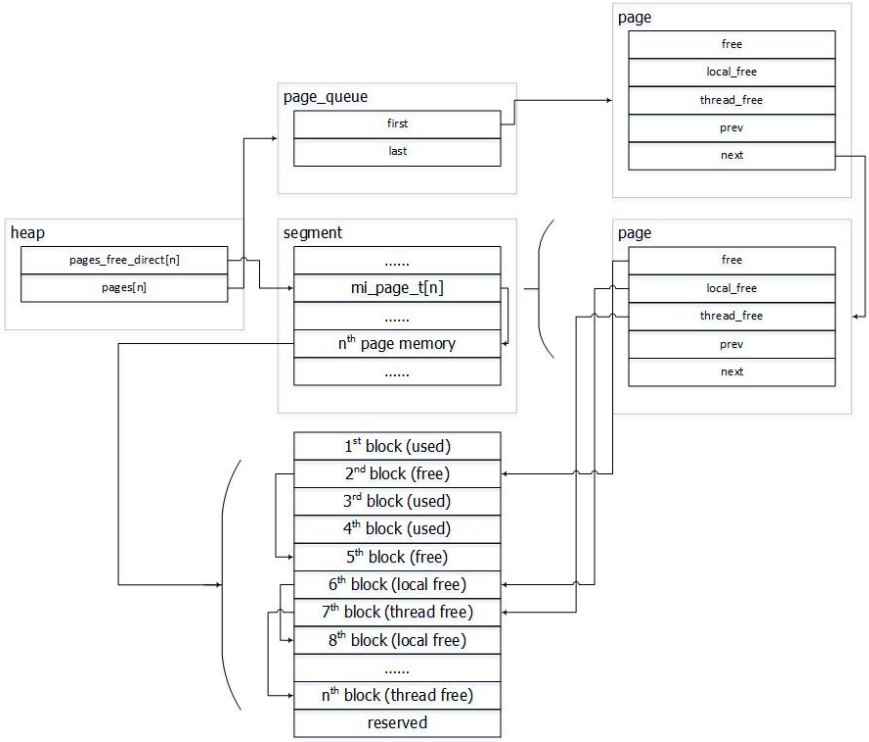
最后来看一个特殊的数据结构mi_segment_t. 为防止频繁调用mmap导致系统vma碎片化, mimalloc采用批量申请内存的方式向系统申请内存. 申请的内存使用mi_segment_t结构来管理, mi_page_t通过mi_segment_t来分配. 一个段的大小是4M(on 64bit platform), 可以被分为64 * 64K(小页), 8 * 512K(中页)或1 * 4M(大页)(再大的内存采用直接申请的方式), 相关大小的设置可以看MI_SMALL_PAGE_SIZE / MI_MEDIUM_PAGE_SIZE / MI_LARGE_PAGE_SIZE几个宏.

20.01.18 补两张图





1. 一个segment被线性划分为若干个page, 每个page大小相同(注意相同大小的page可以用于分配不同大小的block).
2. 第一个page包含了这个segment的管理前缀(即mi_segment_t与若干个mi_page_t), 因此可用的实际空间小于其它page
3. 已知内存地址索引段地址的方式是按4M对齐, 索引页结构地址的方式是减去段地址后偏移段大小获得页序号
4. MIDDLE / LARGE segment划分: 8*512K / 1*4M



1. mi_heap_t中保存两个关于页信息的成员, pages_free_direct[]指向包含空闲小内存的各类页, pages[]指向所有类型内存的页的队列
2. page中包含三个链表, free, local_free, thread_free, 分别指向本页内的空闲内存, 本线程释放的内存, 其它线程释放的内存
3. 页内的内存块每次申请时按系统页(4K)大小初始化为空闲内存块, 其余内存为reserved状态, 原因是减少RSS大小

4. 接口实现

alloc-override*.c中定义如何将标准malloc/free及c++的new/delete操作符alias为mi_*接口, 比如malloc被alias为mi_malloc:

```
#define MI_FORWARD(fun) __attribute__((alias(#fun), used, visibility("default"), copy
void *malloc(size_t size) mi_attr_noexcept MI_FORWARD1(mi_malloc, size);
```

先来看下mi_malloc的实现, 根据申请内存大小选择fast path或slow path.

```
void *mi_heap_malloc(mi_heap_t *heap, size_t size) {
  // MI_SMALL_SIZE_MAX = 128 * sizeof(void *)
  if (size <= MI_SMALL_SIZE_MAX) {
    return mi_heap_malloc_small(heap, size);
  }
  return _mi_malloc_generic(heap, size);
}
void *mi_malloc(size_t size) {
  return mi_heap_malloc(mi_get_default_heap(), size);
}
```

对于小于1M的内存分配会选择fast path. mi_heap_malloc_small会尝试从当前堆的free链表中获取空闲块, 若不存在则回退slow path. _mi_wsize_from_size会将传入的大小对齐到sizeof(void *)(machine word size)边界后取index, pages_free_direct数组存放了包含对应大小的空闲块的page.

```
void *mi_heap_malloc_small(mi_heap_t *heap, size_t size) {
  mi_page_t *page = _mi_heap_get_free_small_page(heap, size);
  return _mi_page_malloc(heap, page, size);
}
size_t _mi_wsize_from_size(size_t size) {
  retu                          t       / sizeof(uintptr_t);
}
mi_pag                          l       mi_heap_t *heap, size_t size) {
  return heap->pages_free_direct[_mi_wsize_from_size(size)];
```

```
}
void *_mi_page_malloc(mi_heap_t *heap, mi_page_t *page, size_t size) {
  mi_block_t *block = page->free;
  if (block == NULL) {
    return _mi_malloc_generic(heap, size); // slow path
  }
  page->free = mi_block_next(page, block);
  page->used++;
  block->next = 0;
  return block;
}
```

如果申请的内存大于1M或指定大小的空闲块不存在则调用_mi_malloc_generic. 如前文所述, slow path首先会尝试释放之前未释放的内存.
用户可以注册deferred_free用于释放之前未释放的对象, 该接口会在_mi_deferred_free中被调用. 另外此时也会释放其它线程延迟释放的内存块(需要该堆设置MI_USE_DELAYED_FREE标记).
在释放内存后再根据申请大小选择从候补队列或申请大页来分配内存, 找到候补页后调用fast path分配空闲块(有可能再次分配失败重新进入slow path吗?).

```
void *_mi_malloc_generic(mi_heap_t *heap, size_t size) {
  if (!mi_heap_is_initialized(heap)) {
    mi_thread_init();
    heap = mi_get_default_heap();
  }

  _mi_deferred_free(heap, false);

  _mi_heap_delayed_free(heap);

  mi_page_t *page;
  // MI_LARGE_SIZE_MAX = 512K, 32bit platform减半
  if (size > MI_LARGE_SIZE_MAX) {
    if (size >= (SIZE_MAX - MI_MAX_ALIGN_SIZE)) {
      page = NULL;
    }
    else {
      page = mi_huge_page_alloc(heap, size);
    }
  }
  else {
    page = mi_find_free_page(heap, size);
  }
  if (page == NULL) return NULL;

  return _mi_page_malloc(heap, page, size);
}
static mi_deferred_free_fun *deferred_free = NULL;
void _mi_deferred_free(mi_heap_t *heap, bool force) {
  heap->tld->heartbeat++;
  if (deferred_free != NULL) {
    deferred_free(force, heap->tld->heartbeat);
  }
}
void _mi_heap_delayed_free(mi_heap_t *heap) {
  mi_block_t *block;
  do {
    block = (mi_block_t*)heap->thread_delayed_free;
  } while (block != NULL && !mi_atomic_compare_exchange_ptr((volatile void**)&heap->t

  while(block != NULL) {
    mi_block_t *next = mi_block_nextx(heap->cookie, block);
    if (!_mi_free_delayed_block(block)) {
      mi_block_t *dfree;
      do {
                              ea        ad_delayed_free;
                              ->         block, dfree);
        ,                     re_       ge_ptr((volatile void**)&heap->thread_delayed
    }
```

```
    block = next;
  }
}
```

来看下如何选择一个合适的页来申请内存. 对于分配小块内存的页由mi_heap_t->pages[]管理, 根据申请大小计算index获取mi_page_queue_t, 其链表头指向的页是最近分配该大小(LRU)的页. 该页可能已满, 所以先要调用_mi_page_free_collect对该页做free操作.

如果释放操作后仍然没有空闲块表明LRU页不存在指定大小的空闲块, 需要调用mi_page_queue_find_free_ex遍历所有页来回收内存. 该接口会遍历队列中的每个页并尝试释放内存, 此处有三种情况:

一个页经过回收后存在空闲块则选择该页. 需要注意的是如果发现该页完全被回收则会尝试释放整个页(称作retire), 第一个空闲页又会保存到循环外retire防止之后的页都非空导致无内存分配.

如果发现一个页虽然没有空闲块但是capacity小于reserved说明该页还能扩展空闲块.

mi_page_extend_free负责扩展空闲块, 查看该接口可以发现空闲块的扩展并不是无限制的, 而是每次不超过一个物理页(OS page), 这里分次extend的原因应该是防止RSS段无意义的增长(初始化free链表时会写地址导致物理内存的分配).

如果发现一个页已满则将其挂入full链表(mi_page_to_full), 防止多次遍历降低效率.

mi_heap_t->pages[]的最后一个index用来存放已满的页(如果该页空闲以后, 什么时候将其取回到正常链表?).

在遍历所有页后仍然未能获取合适的页那么调用mi_page_fresh获取一个全新页.

```
mi_page_t *mi_find_free_page(mi_heap_t *heap, size_t size) {
  mi_page_queue_t *pq = mi_page_queue(heap, size);
  mi_page_t *page = pq->first;
  if (page != NULL) {
    if (mi_option_get(mi_option_secure) >= 3 && page->capacity < page->reserved && ((
      mi_page_extend_free(heap, page, &heap->tld->stats);
    }
    else {
      _mi_page_free_collect(page);
    }
    if (mi_page_immediate_available(page)) {
      return page;
    }
  }
  return mi_page_queue_find_free_ex(heap, pq);
}
mi_page_queue_t *mi_page_queue(const mi_heap_t *heap, size_t size) {
  return &((mi_heap_t*)heap)->pages[_mi_bin(size)];
}
bool mi_page_immediate_available(const mi_page_t *page) {
  return (page->free != NULL);
}
void _mi_page_free_collect(mi_page_t *page) {
  if (page->local_free != NULL) {
    if (page->free == NULL) {
      page->free = page->local_free;
    }
    else {
      mi_block_t *tail = page->free;
      mi_block_t *next;
      while ((next = mi_block_next(page, tail)) != NULL) {
        tail = next;
      }
      mi_block_set_next(page, tail, page->local_free);
    }
    page->local_free = NULL;
  }
  if (mi_tf_block(page->thread_free) != NULL) {
    mi_page thread free collect(page);
  }
}
mi_pag                          re      _heap_t *heap, mi_page_queue_t *pq) {
  mi_page_t *rpage = NULL;
```

```
  size_t count = 0;
  size_t page_free_count = 0;
  mi_page_t *page = pq->first;
  while( page != NULL)
  {
    mi_page_t *next = page->next;
    count++;

    _mi_page_free_collect(page);

    if (mi_page_immediate_available(page)) {
      if (page_free_count < 8 && mi_page_all_free(page)) {
        page_free_count++;
        if (rpage != NULL) _mi_page_free(rpage, pq, false);
        rpage = page;
        page = next;
        continue;
      }
      break;
    }

    if (page->capacity < page->reserved) {
      mi_page_extend_free(heap, page, &heap->tld->stats);
      break;
    }

    mi_page_to_full(page, pq);
    page = next;
  }

  mi_stat_counter_increase(heap->tld->stats.searches, count);

  if (page == NULL) {
    page = rpage;
    rpage = NULL;
  }
  if (rpage != NULL) {
    _mi_page_free(rpage, pq, false);
  }

  if (page == NULL) {
    page = mi_page_fresh(heap, pq);
  }
  return page;
}
mi_page_extend_free(mi_heap_t *heap, mi_page_t *page, mi_stats_t *stats) {
  if (page->free != NULL) return;
  if (page->capacity >= page->reserved) return;

  size_t page_size;
  _mi_page_start(_mi_page_segment(page), page, &page_size);
  if (page->is_reset) {
    page->is_reset = false;
    mi_stat_decrease(stats->reset, page_size);
  }

  mi_stat_increase( stats->pages_extended, 1);

  size_t extend = page->reserved - page->capacity;
  size_t max_extend = MI_MAX_EXTEND_SIZE / page->block_size;
  if (max_extend < MI_MIN_EXTEND) max_extend = MI_MIN_EXTEND;

  if (extend > max_extend) {
    extend = (max_extend==0 ? 1 : max_extend);
  }

  mi_page_free_list_extend(heap, page, extend, stats);
}
void m[illegible]_page_queue_t *pq) {
  _mi_[illegible]_DELAYED_FREE);
  if (page->flags.in_full) return;
```

```
  mi_page_queue_enqueue_from(&page->heap->pages[MI_BIN_FULL], pq, page);
  mi_page_thread_free_collect(page);
}
mi_page_t *mi_page_fresh(mi_heap_t *heap, mi_page_queue_t *pq) {
  mi_page_t *page = pq->first;
  if (!heap->no_reclaim &&
      _mi_segment_try_reclaim_abandoned(heap, false, &heap->tld->segments) &&
      page != pq->first)
  {
    page = pq->first;
    if (page->free != NULL) return page;
  }
  page = mi_page_fresh_alloc(heap, pq, pq->block_size);
  if (page==NULL) return NULL;
  return page;
}
```

再来看下大页的获取, mi_huge_page_alloc也是调用mi_page_fresh_alloc获取新页, 后者根据传入的大小决定申请何种页, 最终都是调用mi_segment_page_alloc.

```
mi_page_t *mi_huge_page_alloc(mi_heap_t *heap, size_t size) {
  size_t block_size = _mi_wsize_from_size(size)  *sizeof(uintptr_t);
  mi_page_queue_t *pq = mi_page_queue(heap,block_size);
  mi_page_t *page = mi_page_fresh_alloc(heap,pq,block_size);
  if (page != NULL) {
    mi_heap_stat_increase( heap, huge, block_size);
  }
  return page;
}
mi_page_t *mi_page_fresh_alloc(mi_heap_t *heap, mi_page_queue_t *pq, size_t block_siz
  mi_page_t *page = _mi_segment_page_alloc(block_size, &heap->tld->segments, &heap->t
  if (page == NULL) return NULL;
  mi_page_init(heap, page, block_size, &heap->tld->stats);
  mi_heap_stat_increase( heap, pages, 1);
  mi_page_queue_push(heap, pq, page);
  return page;
}
mi_page_t *_mi_segment_page_alloc(size_t block_size, mi_segments_tld_t *tld, mi_os_tl
  mi_page_t *page;
  if (block_size <= (MI_SMALL_PAGE_SIZE / 16) * 3)
    page = mi_segment_small_page_alloc(tld,os_tld);
  else if (block_size <= (MI_MEDIUM_PAGE_SIZE / 16) * 3)
    page = mi_segment_medium_page_alloc(tld, os_tld);
  else if (block_size < (MI_LARGE_SIZE_MAX - sizeof(mi_segment_t)))
    page = mi_segment_large_page_alloc(tld, os_tld);
  else
    page = mi_segment_huge_page_alloc(block_size,tld,os_tld);
  return page;
}
mi_page_t *mi_segment_page_alloc(mi_page_kind_t kind, size_t page_shift, mi_segments_
  mi_segment_queue_t *free_queue = mi_segment_free_queue_of_kind(kind,tld);
  if (mi_segment_queue_is_empty(free_queue)) {
    mi_segment_t *segment = mi_segment_alloc(0,kind,page_shift,tld,os_tld);
    if (segment == NULL) return NULL;
    mi_segment_enqueue(free_queue, segment);
  }
  return mi_segment_page_alloc_in(free_queue->first,tld);
}
mi_segment_t *mi_segment_alloc(size_t required, mi_page_kind_t page_kind, size_t page
  size_t capacity;
  if (page_kind == MI_PAGE_HUGE) {
    capacity = 1;
  }
  else {
    siz[illegible]1 <<[illegible]e_shift;
    ca[illegible]ze;
  }
  size_t info_size;
  size_t pre_size;
```

```
  size_t segment_size = mi_segment_size(capacity, required, &pre_size, &info_size);
  size_t page_size = (page_kind == MI_PAGE_HUGE ? segment_size : (size_t)1 << page_sh

  mi_segment_t *segment = NULL;
  segment = mi_segment_cache_find(tld,segment_size);
  if (segment != NULL && mi_option_is_enabled(mi_option_secure) && (segment->page_kin
    _mi_os_unprotect(segment,segment->segment_size);
  }

  if (segment == NULL) {
    segment = (mi_segment_t*)_mi_os_alloc_aligned(segment_size, MI_SEGMENT_SIZE, true
    if (segment == NULL) return NULL;
    mi_segments_track_size((long)segment_size,tld);
  }

  memset(segment, 0, info_size);
  if (mi_option_is_enabled(mi_option_secure)) {
    _mi_os_protect((uint8_t*)segment + info_size, (pre_size - info_size) );
    size_t os_page_size = _mi_os_page_size();
    if (mi_option_get(mi_option_secure) <= 1) {
      _mi_os_protect((uint8_t*)segment + segment_size - os_page_size, os_page_size );
    }
    else {
      for (size_t i = 0; i < capacity; i++) {
        _mi_os_protect((uint8_t*)segment + (i+1)*page_size - os_page_size, os_page_si
      }
    }
  }

  segment->page_kind  = page_kind;
  segment->capacity   = capacity;
  segment->page_shift = page_shift;
  segment->segment_size = segment_size;
  segment->segment_info_size = pre_size;
  segment->thread_id  = _mi_thread_id();
  segment->cookie = _mi_ptr_cookie(segment);
  for (uint8_t i = 0; i < segment->capacity; i++) {
    segment->pages[i].segment_idx = i;
  }
  mi_stat_increase(tld->stats->page_committed, segment->segment_info_size);
  return segment;
}
```

再来看下释放接口mi_free, 首先根据地址找到所在段, 再根据段找到页. 如果释放的内存是本线程分配的则挂入local free链表(如果整页都空闲则还会尝试释放页), 否则挂入thread free链表.

```
void mi_free(void *p) {
  const mi_segment_t *const segment = _mi_ptr_segment(p);
  if (segment == NULL) return;
  bool local = (_mi_thread_id() == segment->thread_id);
  mi_page_t *page = _mi_segment_page_of(segment, p);
  if (page->flags.value==0) {
    mi_block_t *block = (mi_block_t*)p;
    if (local) {
      mi_block_set_next(page, block, page->local_free);
      page->local_free = block;
      page->used--;
      if (mi_page_all_free(page)) { _mi_page_retire(page); }
    }
    else {
      _mi_free_block_mt(page, block);
    }
  }
  else {
    mi_free_generic(segment, page, local, p);
  }
}
mi_seg                          r         *p) {
  return (mi_segment_t *)((uintptr_t)p & ~MI_SEGMENT_MASK);
}
```

```
mi_page_t *_mi_segment_page_of(const mi_segment_t *segment, const void *p) {
  ptrdiff_t diff = (uint8_t*)p - (uint8_t*)segment;
  uintptr_t idx = (uintptr_t)diff >> segment->page_shift;
  return &((mi_segment_t*)segment)->pages[idx];
}

bool mi_page_all_free(const mi_page_t *page) {
  return (page->used - page->thread_freed == 0);
}
```

5. 遗留问题与思考

通篇阅读下来, 感觉mimalloc也没有用到特别新颖的技术. 本质还是slab分配器, 那为什么性能能提升10%呢?
直觉的感受是,

1. slab的切片机制无需migrate内存, 减少碎片化, 同时降低了整体的管理开销.
2. 将生产消费队列区分开来, local线程代码lock free, 提升并发性能.
3. 延迟释放的机制, 类似于RCU的同步, 进一步降低资源竞争的损耗.
4. 至于论文里本身提到的局部性问题, 我很好奇怎么得出的结论, 是否有测试数据.

问题:

1. bin与page对应关系?
2. OS接口还没写.
3. 有空补张图吧, 感觉内存布局这块写的太乱了.

分类: 源码笔记

Five100Miles
粉丝 - 73 关注 - 0
+加关注

« 上一篇： LLVM笔记(7) - 指令的side effect
» 下一篇： LLVM笔记(8) - tablegen介绍

posted @ 2020-01-09 02:23 Five100Miles 阅读(5441) 评论(0) 编辑 收藏 举报

编辑推荐：

· redisson 内存泄漏问题排查
· 使用.NET并行任务库(TPL)与并行Linq(PLINQ)充分利用多核性能
· Redis 内存突增时，如何定量分析其内存使用情况
· 记一次 RabbitMQ 消费者莫名消失问题的排查
· C#|.net core 基础 - 深拷贝的五大类N种实现方式

阅读排行：

· 作为博主和曾经员工，谈谈近期的园子
· 博客园终身会员小福利，送华为云服务器
· 救园倒计时：救园最后4天
· 《HelloGitHub》第 102 期
· .NET 工具库高效生成 PDF 文档