组件化OS--aceros的改进：内存分配算法

致理-信计01 张艺缤 zhangyb20@mails.tsinghua.edu.cn

功能实现

- `basic allocator`：路径为 `crates/basic_allocator`
  - 支持 `first fit, best fit, worst fit` 三种分配策略
- `TLSF`：路径为 `crates/TLSF_allocator`
  - 实现了Rust语言和C语言版（通过ffi接入）
- `mimalloc`：路径为 `crates/mimalloc_allocator`
  - Rust语言，目前为单线程版本
- `crates/allocator` 中分别对应实现了上述内存分配器的接入模块
- `modules/axalloc` 中支持通过 `features` 选择不同的内存分配器，默认为 `mimalloc`
  - 可以通过修改 `modules/axalloc/Cargo.toml` 来切换
- 用户态测试：测试用例集成在 `crates/allocator_test` 中，测试框架位于 `crates/allocator/tests`
  - Rust语言测例：`basic, align_test, multi_thread_test`
  - C语言测例：`mitest, glibc_bench, malloc_large`

内存分配概述

内存分配器需要实现的功能：

- init (addr, size)：初始化内存分配器，向内存分配器提供一段[addr,addr+size)的内存空间；
- add_memory (addr, size)：向内存分配器额外增加一段[addr,addr+size)的内存空间；
- alloc (size, align)：申请一段空间，大小为size，地址对齐要求为align（为2的幂，一般为8字节），返回分配空间的起始地址addr；
- dealloc (addr, size, align)：释放一段先前申请的空间，起始地址为addr，大小为size，地址对齐要求为align。

评价内存分配算法的指标：

- 性能：通常要求内存分配器单次O(1)；不仅取决于性能分配器的效率本身，还有分配内存的连续性等各种因素；
- 空间利用率：尽可能高，内部碎块与外部碎块尽可能少；
- 线程安全等。

算法介绍

（1）basic allocator

将全局所有的空闲块用一个链表来维护

- Alloc时，查找一个合适的空闲块，切割为合适大小后分配
- Free时，将其与前后的空闲块合并后插入回链表中

- 根据查找空闲块的策略不同，分为first fit、best fit、worst fit三种
    - first fit：选取第一个大小足够的内存块
    - best fit：选取大小足够的内存块中最小的
    - worst fit：选取大小足够的内存块中最大的

优点：实现相对容易；消除了内碎块

缺点：可能存在外碎块；查找空闲块时最坏需要遍历整个链表，效率低

上述代码约600行，于第8周完成。

（2）TLSF

用两级链表来维护大小在一定范围内的内存块

O(1)的malloc与dealloc；

每次分配的额外空间开销仅为8字节；

内存碎片较少；

支持动态添加和删除内存池；

详细介绍见附录1。

优点：单次操作复杂度严格O(1)；内碎块相对Buddy和slab更少

- 取决于二级链表的大小，如取5位则内碎块不超过1/64

缺点：

- 每次操作时，拆分和合并内存块的开销相对较大；
- 多次申请空间很可能不连续；
- 最小分配单位为16字节，分配小内存块时冗余较大

分别使用Rust语言实现和接入C语言的既有实现，Rust代码量约1000行，于第11周完成。

（3）mimalloc

原始算法是保证线程安全的：即可以支持多线程同时申请/释放，无需上锁（Mutex），仅需原子操作（Atomic）

但算法相对较复杂（原版C代码~3500行），目前实现的Rust版本为单线程的简化版

在mimalloc内存分配器中，内存维护单元分为：堆（Heap）、段（Segment）、页（Page）、块（Block）

每个Page中的块大小都是相同的

Segment以4MB对齐，承载各种Page

每个线程用一个Heap作为mimalloc的控制结构，核心为维护不同block大小的Page链表

详细介绍见附录2。

在已有的Rust实现中，简化了部分原本用于维护多线程的数据结构，如Page维护的块链表仅保留了free_list。

优点：单次操作O(1)；连续分配时内存地址大致连续；速度较现在通用的内存分配器快约7%~14%；内碎块相对较小；

缺点：

- 以段（4MB）和页（64KB）为单位维护内存，在分配请求不规则时可能有较大冗余；
- 算法较为复杂。

当前的Rust语言单线程版本，代码量约1000行，于第15周完成。

APP测试

主要用于测试内存分配的两个app为 `apps/memtest` 和 `apps/c/memtest`

用户态测试

在 `crates/allocator/tests` 中搭建了专用于用户态测试的global_allocator和测试框架

- global_allocator接入了上述各种内存分配器，以及rust自带的System分配器（linux中，使用的是__libc_malloc）
- 通过init_heap函数接入一个全局static的大数组（512MB）用于内存分配
- 通过各种分配器对应的init函数来动态切换使用的分配器
- 实现了各种测试用例的接口以及各种分配器的测试入口，集成为一个test
- 多线程支持：在每个内存分配器外面套一层mutex

新建了`crates/allocator_test`，内部集成了各种测试用例，可以在运行用户态测试时调用

各个测试用例的介绍：

(1) Basic

- Rust语言
- 与apps/memtest类似
- 主要为大量Vec和btreemap测试

(2) Align_test

- Rust语言，自行编写
- 测试align为非8字节的情况
- 仅有System、TLSF(C和Rust)、mimalloc支持该功能

(3) Multi_thread_test

- Rust语言，自行编写
- 测试多线程情况下的内存分配效率

(4) Mitest

- C语言
- Mimalloc仓库中自带的测试用例
- 仓库地址：https://github.com/microsoft/mimalloc
- 主要测试算法的正确性

(5) Glibc_bench

- C语言
- 仓库地址：https://github.com/daanx/mimalloc-bench
- 位于`bench/glibc-bench`
- 一系列比较复杂的内存操作，测试各种算法的性能

(6) Malloc_large

- C语言
- 仓库地址：https://github.com/daanx/mimalloc-bench
- 位于`bench/malloc-large`
- 用于测试对大内存（2~15MB）的申请与释放

(7) Multi_thread_c_test

- C语言，自行编写
- 功能大致类似于multi_thread_test
- 测试C语言下的多线程内存分配

上述测试用例和测试框架的构建完成于第9~15周。

运行用户态测试

```
cargo test -p allocator --release -- --nocapture
```

运行示例app

```
make A=apps/memtest ARCH=riscv64 LOG=info run
make A=apps/c/memtest ARCH=riscv64 LOG=info run
```

用户态测试结果

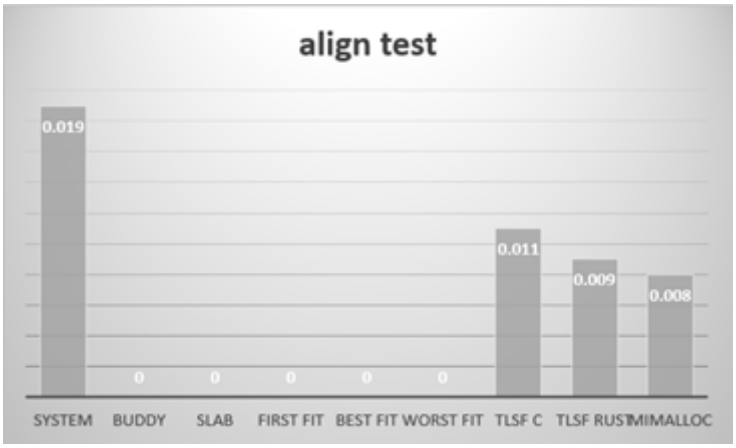
注：单位均为秒（s）；“——”表示未实现相关功能

算法	system	buddy	slab	first fit	best fit	worst fit	TLSF C	TLSF Rust	mimalloc
basic	0.361	8.251	0.584	0.346	1.79	14.557	0.346	0.35	0.336
align test	0.019	——	——	——	——	——	0.011	0.009	0.008
multi thread	3.223	6.11	3.275	3.25	3.524	8.965	3.208	3.206	3.225
mitest	0.054	0.051	0.052	0.072	0.069	0.069	0.054	0.069	0.052
glibc bench	1.539	1.727	1.39	2.086	2.37	2.382	2.374	3.028	2.198
malloc large	0.088	0.007	0.006	0.03	0.024	0.201	0.021	0.022	0.01
multi thread C	0.444	5.429	0.53	0.621	1.374	8.174	0.651	0.778	0.521

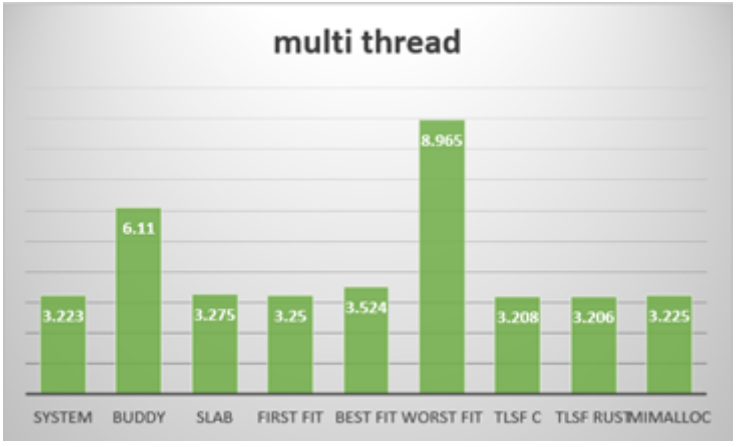
测试结果图表

basic:

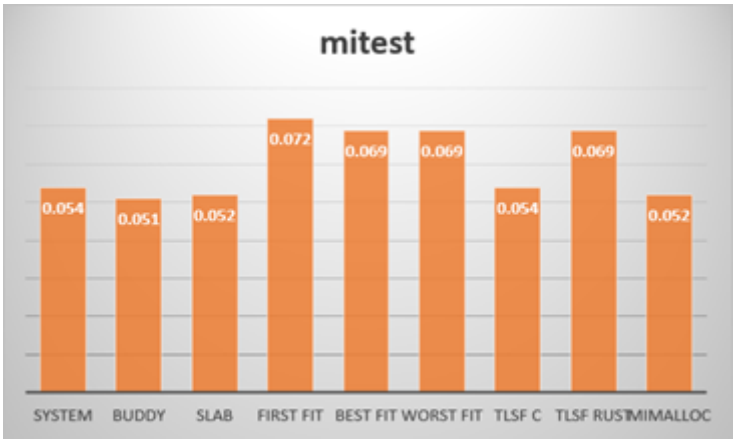
align test:



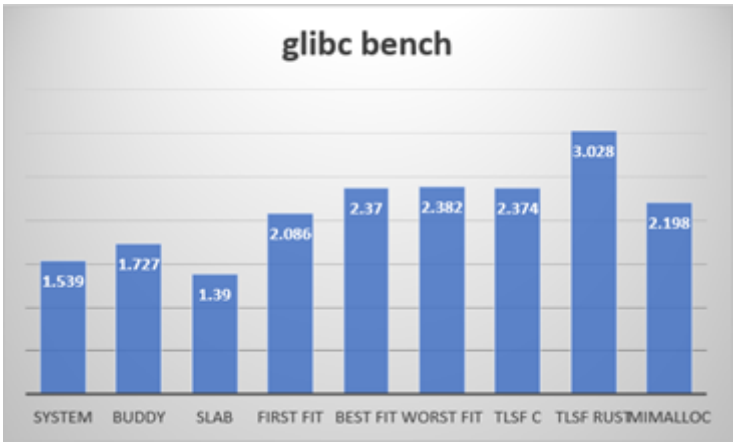
multi thread:



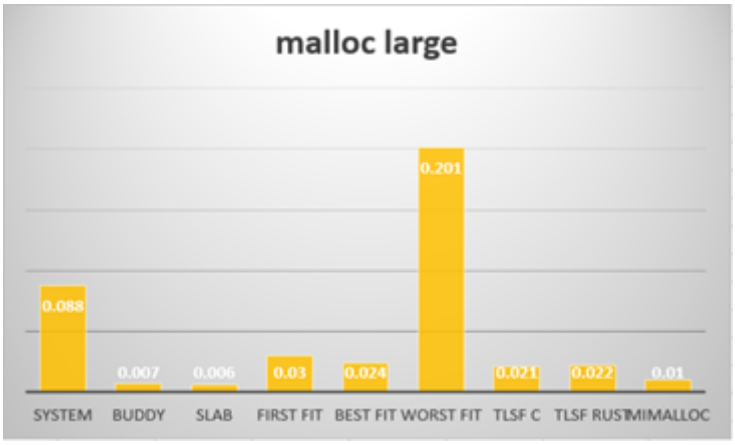
mitest:



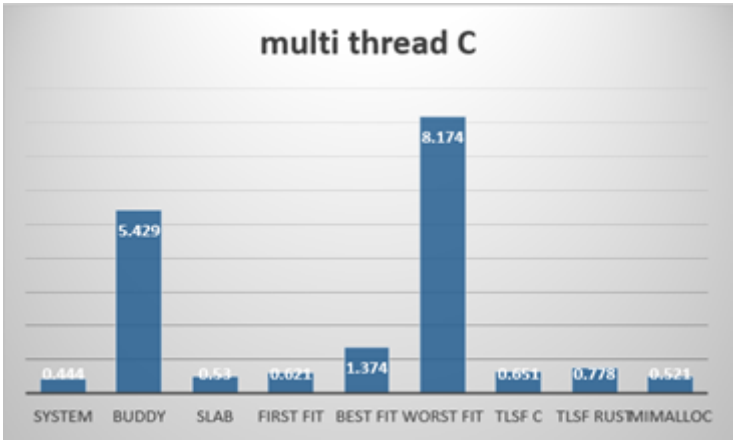
glibc bench:



malloc large:



multi thread C:



结果总结

不同的算法在各个测试点上性能各有高低，没有占据绝对优势的算法

就综合各方面表现而言，mimalloc的性能相对更优，且没有“短板”

TODO

（1）完成Rust版本mimalloc的多线程支持；

（2）将C语言的mimalloc实现接入arceos　仓库链接：https://github.com/microsoft/mimalloc

（3）增加更多多线程测例，以测试mimalloc对多线程的支持　仓库链接：https://github.com/daanx/mimalloc-bench

Reference

https://github.com/microsoft/mimalloc

https://github.com/mattconte/tlsf

https://github.com/daanx/mimalloc-bench

http://www.gii.upv.es/tlsf/files/ecrts04_tlsf.pdf

https://zhuanlan.zhihu.com/p/370239503

https://www.microsoft.com/en-us/research/uploads/prod/2019/06/mimalloc-tr-v1.pdf

https://doc.rust-lang.org/nomicon/ffi.html

https://web.mit.edu/rust-lang_v1.25/arch/amd64_ubuntu1404/share/doc/rust/html/unstable-book/language-features/global-allocator.html

附录1：TLSF内存分配算法

支持：O(1)的malloc与dealloc；每次分配的额外空间开销仅为4字节（默认4字节对齐，64位机器则改为8字节）；内存碎片较少；支持动态添加和删除内存池；不保证线程安全，需要调用者保证

Segregated Fit算法：基于一组链表，每个链表包含特定大小范围的空闲块

Two Level：采用两级链表机制

第1层：将块按照2的幂进行分类：如 $[2^4, 2^5), [2^5, 2^6)$ .....

第1层链表的指针指向一组第2层链表的表头，分别表示将这个区间进一步细分

如 $[2^6, 2^7)$ 可进一步细分为 $[2^6, 2^6 + 2^4), [2^6 + 2^4, 2^6 + 2^5), [2^6 + 2^5, 2^6 + 3 * 2^4), [2^6 + 3 * 2^4, 2^7)$

由一个大小定位到内存块，可以通过先取log2获得第1级位置，再取次高的若干位二进制（上述例子为2位）来实现

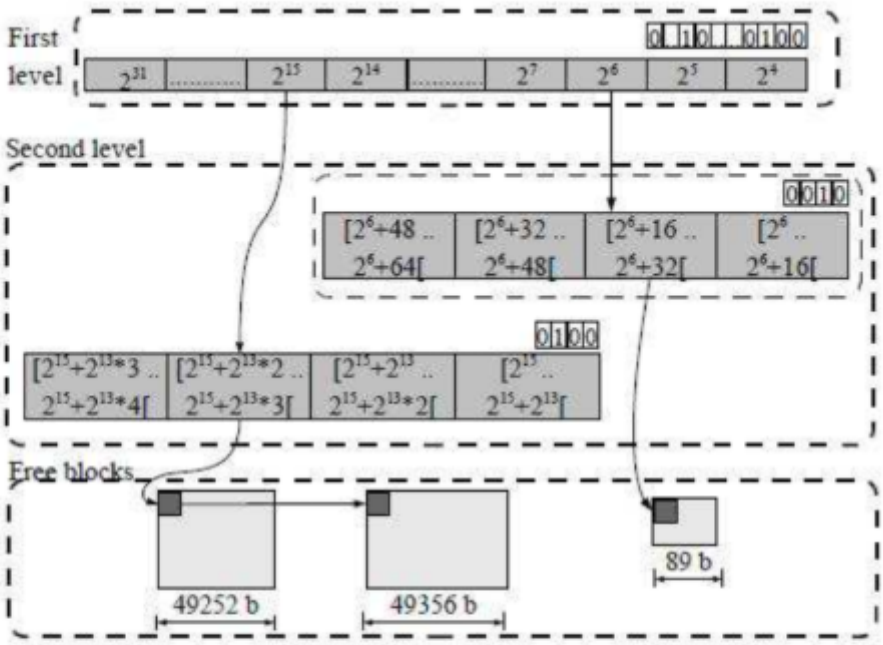


Figure 1. TLSF free data structure overview.

$$\texttt{mapping}\,(size) \rightarrow (f, s)$$

$$\texttt{mapping}\,(size) = \begin{cases} f := \lfloor log_2\,(size) \rfloor \\ s := (size - 2^f)\,\frac{2^{SLI}}{2^f} \end{cases}$$

tlsf内存块头结构：

```
typedef struct block_header_t//tlsf内存块头结构
{
    /* Points to the previous physical block. */
    struct block_header_t* prev_phys_block;//内存地址上上一个块的位置指针
    //只存储上一个块的位置，是因为下一个块可以根据这个块的大小算出来
```

```
    /* The size of this block, excluding the block header. */
    size_t size;//这个块的大小，注意是不包括分配时要带的8字节块头大小的
    //因为块大小是4对齐的，所以用低2位分别表示这个块和上一个块是否是free的

    /* Next and previous free blocks. */
    struct block_header_t* next_free;//free链表中的下一个块
    struct block_header_t* prev_free;//free链表中的上一个块
    //free链表只对free状态的块使用
} block_header_t;
```

整个tlsf的控制头结构：

```
typedef struct control_t//整个tlsf的控制结构
{
    /* Empty lists point at this block to indicate they are free. */
    block_header_t block_null;//空块

    /* Bitmaps for free lists. */
    unsigned int fl_bitmap;//一级链表的bitmap，标记每个一级链表是否非空
    unsigned int sl_bitmap[FL_INDEX_COUNT];//二级链表的bitmap，标记每个二级链表是否非空

    /* Head of free lists. */
    block_header_t* blocks[FL_INDEX_COUNT][SL_INDEX_COUNT];//二级链表结构
    //SL_INDEX_COUNT=32表示二级链表将一级链表的一个区间拆分成了32段，也就是要根据最高位后的5个二进制位来判断
} control_t;
```

将大小小于256的块单独维护（位于一级链表的0下标处，也要按照第二级链表维护），其余块按照两级链表结构维护

malloc的策略：

1、先将所需size上取整到下一个二级块大小的下界，并找到相应的一、二级链表；

2、查询是否存在一个符合要求的块：先在相应的一级链表中找：从给定的二级链表开始第一个非空的二级链表，如果存在就从中取一个块；如果不存在，就向上找第一个非空的一级链表，再找其中第一个非空的二级链表；

3、如果分配出去块比所需内存大很多，可以split，要求至少大一个block_header_t的大小（32字节）；

4、如果malloc时额外要求地址对齐（大于8字节）：则要找到一个足够大的块，使得在块上找到对齐的地址后，前面留下的部分拆成一个更小的块（至少32字节，如果不足则将分配出去的地址继续向后移动），之后仍然有足够的空间。

free的策略：物理上连续的空闲内存块要前后合并，然后插入回相应的二级链表中

realloc的策略：先看当前块和物理上的下一块（如果空闲的话）加起来够不够用

如果够用，就直接与下一块合并（如果空闲的话）再在原位重新分配

否则，直接重新alloc然后free掉原来的

附录2：mimalloc内存分配算法

保证线程安全，但无需使用锁，只需原子操作；

用free-list的分页思想，提高内存的连续性；

使用分离适配的思想，每个页维护相同大小的内存块；

O(1)的单次操作，83%以上的内存利用率，在benchmark上相较于tcmalloc和jemalloc快7%~14%

每个页维护3个链表：空闲块链表free、本线程回收链表local_free，其他线程回收链表thread_free

维护local_free而不是直接回收到free里应该是基于效率的考虑（并非每次free都要收集，而是定期通过调用page_collect来一次性收集）；thread_free的设立是为了线程安全

需要分配时，先从free中取，若free为空，则再调用page_collect：先去尝试收集thread_free，其次再收集local_free。这一机制可以确保page_collect会被定期调用到，可以同时保证效率和不产生浪费的内存块

```
void* _mi_page_malloc(mi_heap_t* heap, mi_page_t* page, size_t size) {
    //从一页中分配内存块的大致逻辑
  mi_block_t* const block = page->free;
  if (block == NULL) {//当free链表为空时，进入generic
    return _mi_malloc_generic(heap, size);      // slow path
  }
  // pop from the free list
  //fast path，直接取链表头即可
  page->used++;
  page->free = block->next;
  return block;
}

void* mi_malloc_generic(mi_heap_t* heap, size_t size) {
    size_t idx = size_class(size);      // 计算得到 size class
    mi_page_queue_t* queue = heap->pages[idx];  // 取相应的page队列
    foreach(page in queue) { // 注：这里还需要限制枚举页的数量，以确保单次调用的时间开销有上限
        page_collect(page);     // 收集页里的可用内存
        if (page->used == 0) {  // 整个页都空闲，回收掉
            page_free(page);
        } else if (page->free != NULL) {    // 收集完如果有可用内存，则重回分配入口
            return mi_heap_malloc(heap, size);
        }
    }
    // 到这儿表明找不到可用的page，从segment分配一个新鲜的page
    ...
}

void page_collect(mi_page_t* page) { // 收集页里的可用内存
```

```
  // collect the thread free list
  if (mi_page_thread_free(page) != NULL) {  // quick test to avoid an atomic operation
    _mi_page_thread_free_collect(page);//先收集thread_free
  }
  // and the local free list
  if (page->local_free != NULL) {//如果上一步没收集到东西，再收集local_free
    if (page->free == NULL) {
      // usual case
      page->free = page->local_free;
      page->local_free = NULL;
    }
  }
}
```

每个线程维护线程本地数据结构（段、堆、页），alloc时每个线程只能从自己的堆中分配空间，但是free时任何线程都可以free

因此这里需要保证线程安全，为了避免加锁，采用thread_free链表：其他线程释放本线程的块时，通过原子操作插入到这个块的thread_free里：

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

atomic_push( &page->thread_free, p );
```

整个数据结构的组织架构如下：

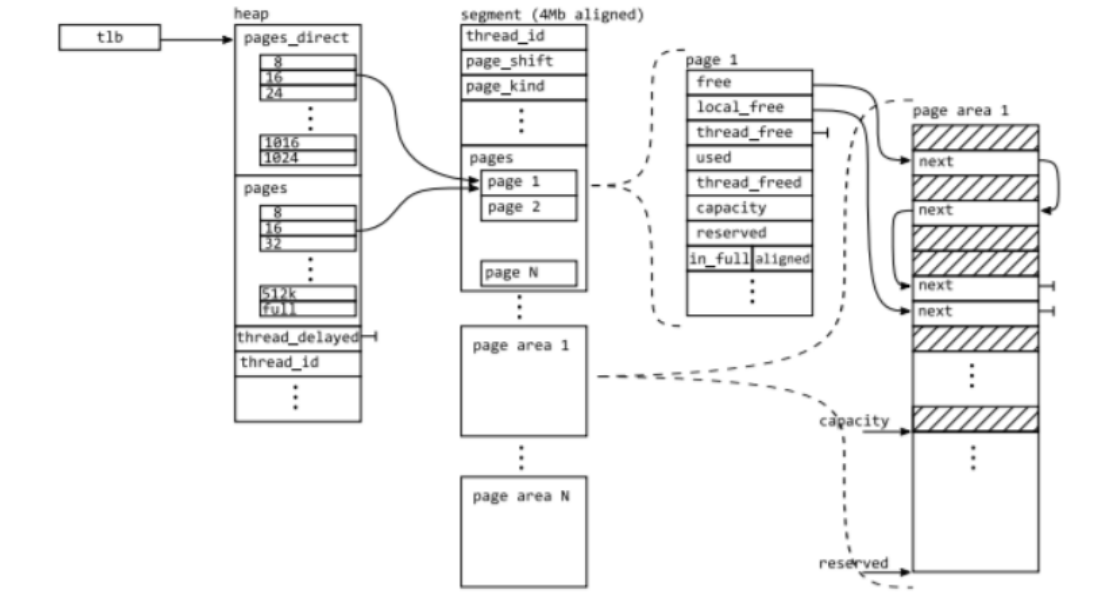
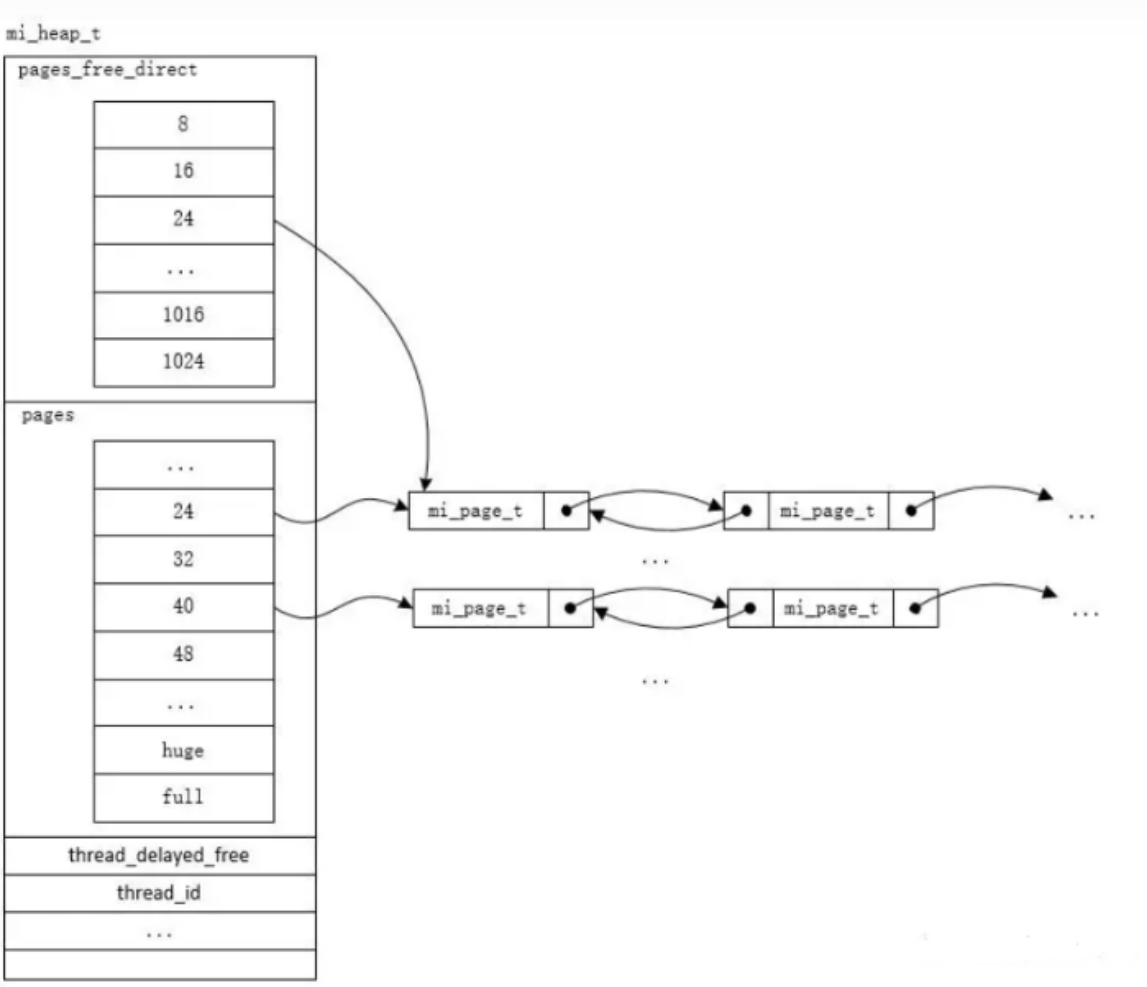
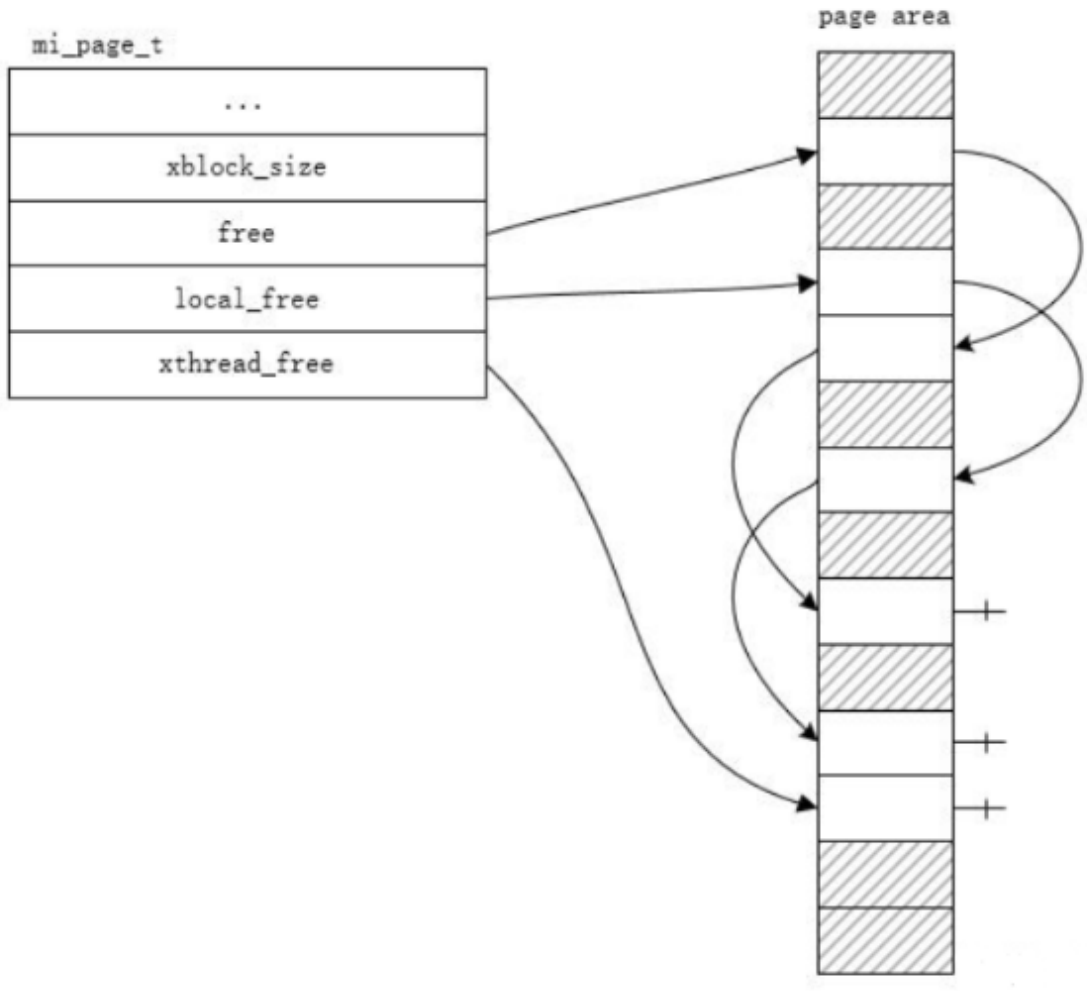


Fig. 1. Heap layout

每个segment以及pages链表的结构如下：



每个page的结构如下：



每个线程维护一个heap：其中的pages字段是一个数组，是根据内存块size大小维护的若干个page的队列（链表），不超过1024的内存块又被pages_direct指针指向，以快速获取（找一个page时，查pages_direct表相比查pages表更快）。

每个segment的开头是一段元数据，pages字段存储的是其中的page的元数据，包括线程号、块大小、free链表、local链表、thread链表等信息；segment的剩余部分就是每个page的地址空间，对于空闲的块，用next指针指向它在链表中的下一个块。

alloc的逻辑如下：

```
void* mi_malloc( size_t n ) {
    heap_t* heap = mi_get_default_heap();   // 取线程相关的堆
    return mi_heap_malloc(heap, size)
}

void* mi_heap_malloc(mi_heap_t* heap, size_t size) {
    if (size <= MI_SMALL_SIZE_MAX) {    // 如果<=1024，进入小对象分配
        return mi_heap_malloc_small(heap, size);
    } else {    // 否则进行通用分配
        return mi_malloc_generic(heap, size)
    }
}

void* mi_heap_malloc_small(mi_heap_t* heap, size_t size) {
    page_t* page = heap->pages_direct[(size+7)>>3]; // 从pages_direct快速得到可分配的页
```

```
    block_t* block = page->free;
    if (block != NULL) {                // fast path
        page->free = block->next;
        page->used++;
        return block;
    } else {
        return mi_malloc_generic(heap, size); // slow path
    }
}
```

调用mi_malloc_generic是一个比较慢的过程，程序会在这一阶段进行一些回收工作。程序机制确保它会被每间隔一段时间调用，使得操作的复杂度可以被均摊。

free的逻辑如下，注意如何通过地址找到内存块所在的段和页，以及对于本线程和其他线程的不同处理：

```
void mi_free(void* p) {
    segment_t* segment = (segment_t*)((uintptr_t)p & ~(4*MB));  // 找到对应的segment
    if (segment==NULL) return;
    // 找到对应的page，这是简化过的，第1个page要特殊处理。
    // 因为segment等分成N个page，这里只需要取相对地址，然后除去page的大小，即得到page的索引。
    page_t* page = &segment->pages[(p - segment) >> segment->page_shift];
    block_t* block = (block_t*)p;

    if (thread_id() == segment->thread_id) { // 相同线程，释放到local_free
        block->next = page->local_free;
        page->local_free = block;
        page->used--;
        if (page->used == 0) page_free(page);
    }
    else { // 不同线程，释放到 thread_free
        atomic_push(&page->thread_free, block);
    }
}
```

heap的full存储的是当前已满的页，这是为了保证分配时的效率，避免每次分配时都要遍历所有已满的页面（否则会带来最大30%的效率损失）。

当一个也从不满变成满时，将其从其所在的pages队列中取出，放进full队列

当一个已满的页被重新释放时，要将其改回不满的状态，但由于释放可能是其他线程引起的，要考虑线程安全问题（要在线程安全的前提下通知原线程这个块已经不再满了），用thread_delayed机制来解决。

一个页用2位二进制来表示状态：当一个页不满时，状态为normal，其他线程的free就将内存块正常插入到这个页的thread_free链表里；但如果已满，状态为delayed，其他线程的free就要插入到原线程的heap的thread_delayed_free链表里，并将状态改为delaying；delaying状态的page在其他线程的free时仍然插入到这个页的thread_free链表里，以保证thread_delayed_free链表里不存在来自相同page的内存块（这同样是为了确保效率，否则会带来最大30%的效率损失）。

等原线程的mi_malloc_generic时，就会遍历这个thread_delayed_free链表，把其中的内存块所在的page重新设为normal状态，并将其从full队列移回到原来的pages队列。

更多的安全措施：如初始建立free链表时以随机顺序连接等，经测试这仅会使程序的性能相较无安全措施的版本慢3%左右。