L1: 物理内存管理

主要思路

对于多处理器的分配，如果使用一把大锁保护整个管理结构，将导致很长的自旋等待。受实际系统里的buddy分配器和slab分配器的启发，我把内存分配的过程分为了两个path:

- slow path

某个cpu申请**物理页大小**级别的内存。此时用一把大锁锁住buddy的管理结构，从中分配出页面。

- fast path

某个cpu申请**小于物理页大小**级别的内存。此时先查询该cpu自己的slab缓存里还有没有可用的piece，如果有就将一个piece从链表里摘下来分配出去；如果没有，再**走slow path**，从buddy分配器里获取一个物理页。将此页划分出一些空间用于储存管理结构之后，再将其切成若干个对应大小的piece，分配一个piece出去，剩余的piece留在该cpu的slab缓存链表里，留给后续使用。

归还方式也有所区分:

> 由于slab开头需要一些头信息，故slab分配器分配的内存总是不和物理页对齐，很容易区分出来内存是哪个分配器分出去的。

- buddy分配器分配出去的内存

一把大锁锁住buddy的管理结构，正常归还到buddy分配器。

- slab分配器分配出去的内存

直接把所归还的piece放回当前cpu的slab缓存链表里。由于一个cpu申请的内存可能在其他cpu上被释放，这里的归还方式有可能使得某个cpu“偷取”了另一个cpu的piece。不过这并不要紧，由于slab分配上有一些特别的地址结构设计，一个piece的管理结构和他真正管理的空间可以轻易地互相定位，而并不需要知道是哪个cpu持有着它。

如果一个slab所切成的全部piece都被归还，我选择把他继续留在cpu的slab缓存里而不是归还buddy。因为归还buddy就违背了fast path的初衷——workload可以频繁申请一个小内存再释放之，使得cpu忙碌于反复地从buddy申请/归还。

一些精巧的设计

区分buddy或slab

分配时分配器是我们定的，但归还时cpu仅仅交给我们一个地址，我们需要区别它应如何归还。

由于buddy机制按物理页为单位分配内存，分配出去的地址总是和页面对齐。而slab机制是先从buddy里取一个页，用去头部一些空间后再划分piece，其分配出去的地址总是不和页面对齐。我们可以以此来区别二者:

```
static void kfree(void *ptr) {
    if ((((uintptr_t)ptr) & (PAGE_SIZE - 1)) == 0) {
        // aligned to page, it must be allocate by buddy
```

```
            buddy_free(ptr);
        } else {
            slab_free(ptr);
        }
    }
```

充分榨干堆空间

由于堆的头部空间被拿来存管理结构，同时又要照顾到对齐的要求，堆空间的形状可能会变成下面这样:

```
| management structure | padding | available space | free space |
```

其中padding是为了对齐必须保留的，而free space还可以继续利用(由于buddy机制的需要，最开始available space是2的幂，没有往后拓展)。于是我从最大的2的幂开始不断向下探索，如果padding还够management structure吃掉，且free space还够available space吃掉，就往后拓展。新吃掉的空间放入buddy对应level链表里。

管理结构与真正空间的互定位

不管是slab还是buddy，我都会在其管理空间的头部存下一些信息，这些信息可以帮助管理结构定位到其真正管理的空间。而对于另一个方向的定位，我则通过地址结构上的设计，使一个地址能轻松找到其管理结构的头部，从而定位到其管理结构。

印象深刻的Bug

我搭建测试框架的时候写了一个C++程序checker，读入若干次kalloc、kfree的记录，判断其中是否有矛盾:

- kfree一个还没kalloc的内存区间
- kalloc的内存区间与未被kfree的另一个内存区间相交

结果我如愿得到了assertion failed。再反复排查之后我坚信自己的分配代码没有问题。最后我发现了是我输出kalloc、kfree日志的方式有问题——我的输出方式类似于:

```
void *ptr = pmm->alloc(size);
printf("kalloc %p %p\n", ptr, ptr + size);
```

由于我把上锁的过程封装到alloc里面了，所以很长时间我都没有意识到，printf并未和alloc锁在一起，这也就意味着可能申请的空间都被释放了，我还没能输出我的kalloc日志，所以就导致了checker的assertion failed。在fix了这个bug之后，我成功通过了checker测试。

这个Bug算是让我体会到了并发Bug的无处不在，以及它超强的隐蔽性——在上锁的代码被封装到函数里面的时候更是如此。