OSLAB1

架构设计

我采用了 linux 中 buddy system 和 slab 的设计方案。将内存前一段作为 buddysystesm 的控制块，用哈希表以 64KB/页 为单位映射内存；用`MAX_ORDER = 9`个链表管理 64KB,128KB...16MB 的空闲空间。

```
typedef union{
    struct{
        void* free_list[MAX_ORDER];
        page_t units[8192];
        spinlock_t tree_lock;
    };
    uint8_t data[1<<18];
}tree;
```

`struct cpu_t`以链表结构维护 slab ，每个 slab 大小为 64KB，其中前 4KB 作为`header`储存该块的信息，包括魔数、链表指针、`bitmap`等。

```
struct cpu_t{
  void* link_head[MAX_SIZE];
  spinlock_t cpu_lock[MAX_SIZE];
};
```

buddy system

`buddy_alloc(size_t size)`分配时先从`size`大小的空闲链表中搜索，如果为空则递归调用`buddy_alloc(size<<1)`，找到`size<<1`的一块空闲空间，一块作为结果返回，另一块加入到`size`对应的的空闲链表中。显然，`buddy_alloc()`的复杂度是 O(1) 的。

`buddy_free(void* ptr)`在回收时，先根据传入的指针在控制块中找到对应的信息，从控制块头中找到该块的`size`和相邻块的占用情况，如果相邻块为空闲，则将其移出空闲链表，与本块合并，修改控制块中的对应信息。我采用了循环（而不是递归）的方式实现合并操作。当无法再合并时，将得到的大空闲块加入空闲链表中。`buddy_free()`的复杂度是 O(log n) 的。

slab

`slab_alloc(size_t size)`会先从本线程的 slab 链表中寻找空闲空间，当对应`size`的链表为空或全满时，调用`buddy_alloc(PAGE_SIZE)`获得新的一页并初始化，然后将其插入到链表的第一位。分配 slab 内的空间：从`slab->bitmap`中找到一块空闲空间，计算其相对`slab->data[0]`的偏移量得到地址，然后返回。在不调用`buddy_alloc()`的情况下，`slab_alloc()`的复杂度是 O(log n) 的。

`slab_free(void* ptr)`先计算出`header`的地址和`ptr`在 slab 中的位置，然后修改`slab->bitmap`和`slab->count`。`slab_free()`的复杂度是 O(1) 的。

函数封装

为了准确区分 slab 和 slow path 分配的大内存，我在每块 slab head 处添加了魔数`0x7355608`，`slab_alloc()`时进行初始化，`free()`时检查魔数并依此调用相应的 free .

并发问题

本设计可能引起数据竞争的部分有两个，`buddy_alloc()`、`buddy_free`对全局链表的操作，和从一个线程上`slab_free()`其它线程申请的内存。对于前者，只能通过一把全局大锁来维护其正确性，但由于 slab 机制，4KB 及以下的内存申请很少用到`buddy_alloc()`，其速度还是可以接受的。对于后者，在每个 slab head 处有一个页面锁，对`slab->bitmap`作出修改时始终上锁。由于这种情况很少出现，并且`slab_free()`很快，其速度也是可以接受的。

测试框架

我在本地并未进行足够强的压力测试。对于 buddy system 的正确性测试，在若干次申请和释放后，通过`void print_mem_LL()`检查是否正确释放和合并大块内存。

性能优化

考虑将维护两条 slab 链表，分别为`FREE`和`FULL`，可以有效降低`slab_alloc()`的时间复杂度。

考虑将空的 slab 返回到伙伴系统，以减缓内存的碎片化。

TODO

修改 klib 使其支持对`long int`类型的`printf`. (%p %lx %ld)