OSLAB实验报告

周晓 191240080
匡亚明学院

L1实验报告

数据结构

将堆区划分为两个区域，第一个区域使用slab分配小内存（小于4KiB），第二个区域使用freelist分配大内存。
第一个区域将内存分为一个一个的page，大小设置为32KiB，page的数据结构如下

```
//item size 设置为{2, 4, 8, 16,... ,4096} 共12项
typedef union page {
    struct {
        struct spinlock lock; //锁，用于串行化分配和并发的free
        int obj_cnt; //页面已分配的对象数，减少到0的时候回收页面
        int max_obj; //最多可分配的对象数
        uintptr_t offset; //start_ptr + offset就是目前分配到的地方，在这个位置上有一个obj_head代表下一个可供分配的位置
        union page *next; //同一个cpu下的其他页面
        union page *prev;
        int obj_order; //分配的大小为2^obj_order
        uintptr_t start_ptr; //第一个数据的起始位置
        int cpu;
    };
    struct {
        uint8_t header[HDR_SIZE];
        uint8_t data[PAGE_SIZE - HDR_SIZE];
  } __attribute__((packed));
} page_t;

struct obj_head {
    uint16_t next_offset; //下一个分配的地址离start_ptr的offset
};

page_t* cache_chain[MAX_CPU + 1][NR_ITEM_SIZE + 1];
```

每个页面中分配的内存大小是一致的，由于在分配前内存的内容是不限制的，我设计了一个obj_head数据结构用于指向当前分配位置的下一个offset， 在初始化page时也初始化其中所有的obj_head。

为了提高速度，我为每一个cpu中的每一个item_size都设计了一个page的链表，且此处的链表我设计为**循环链表**

分配与释放过程

在kalloc中，先判断申请的内存大小，从而选择freelist或者page。如果是小内存分配，那么根据cpu和相应的order(1<< order是大于申请内存的最小2次幂)找到对应的cache_chain

- 如果cache_chain指向的page仍有空闲，那么直接本地进行分配，并更新offset，如果在分配后该page已经满了，那么使用cache_chain->next来对cache_chain进行更新
- 如果cache_chain指向的page已经满了，那么说明没有空闲的page了，从全局申请一个page，并插在表头，再进行分配

在kfree中，根据指针指向的位置判断是freelist的释放还是page内内存的释放。如果是page的，使用ROUNDDOWN找到page的头

- 如果obj_cnt = 1, 说明释放完之后页面变成完全空闲的了，此时将其释放回全局，并从链表中删除
- 否则，将obj_cnt减1,并根据ptr来更新offset，如果原本的页面是满的，则将其插入到表头

遇到的bug

在某一次运行时发现怎么都上不了锁，最后发现是因为在初始化page时初始化obj_head越界修改了下一个page的锁，修改了边界判定条件后正常

发现kfree时会re，bug是free时的cpu不一定是alloc时的cpu，所以将页面插入到表头时可能会插入到另一个cpu的cache_chain中，只需要在page的结构体里加一个cpu就好