# OSLAB_Report

*181220076 周韧哲*

## L1_Report

- 数据结构：
  - 我将整个_heap按8KiB大小分为了数个页面，在结构体page_t中定义了page所属的cpu、已分配对象数和总对象数、第一个对象的地址、bitmap等：

```
typedef union page {
  struct {
    spinlock_t lock;
    int cpu;
    int slab_size;    //如果是0，则表示它不在缓存而在_heap中
    int obj_cnt;     // 页面中已分配的对象数，减少到 0 时回收页面
    int obj_num;     //总对象数
    void *addr;      //首地址
    void *s_mem;     //slab第一个对象的地址
    list_head list;
    unsigned int bitmap[31];  //往后obj_num个bit都属于bitmap;
  }; // 匿名结构体,header大小固定为256
  uint8_t data[PAGE_SIZE];
} __attribute__((packed)) page_t;
```

    这样，每次在page内申请内存时只需访问bitmap，返回s_mem+偏移量即可。当page对象都分配完后，判断一下cpu中完全闲置的page数量是否超过某个阈值从而决定是否将其返回到_heap中。
  - 每个cpu都有自己的缓存，初始化大小为n个page，且设置了10个种类的slab大小：{ 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096 }，用链表连接属于每一类slab的page。

```
typedef struct kmem_cache{
  int cpu;
  spinlock_t lock;
  int free_num[SLAB_TYPE_NUM];    //完全闲置的page数量
  list_head slab_list[SLAB_TYPE_NUM];
  list_head *freepage[SLAB_TYPE_NUM];
}kmem_cache;
```

    其中 freepage 指向 slab_list 中有空闲对象的 page，每当page满了后就遍历链表对freepage重新赋值，这样在均摊意义下就提高了slab分配的速度。在我自己的多核单alloc的测试下比不加freepage的版本速度提升了3倍左右。
- kalloc&kfree：
  - kalloc：

        - 先在cpu自己的缓存中找到对应size的slab_list，然后在空闲页面中通过bitmap访问空闲单元并返回。
        - 当缓存中page都分配完了时，加一把全局的大锁，从_heap中找未被某个cpu占用的page，初始化后返回对应地址即可。
        - 由于cpu缓存中完全闲置的page数目有一个阈值，超过阈值则会被归还到_heap中，所以在绝大多数情况下cpu在_heap中的alloc都是可以做到的，否则的话，就返回NULL。
    - kfree：
        - 通过地址找到对应page的首地址，然后给page上锁，清空对应内存。若此时page成了完全闲置的page，则它所属cpu的free_num加1，若超过阈值则归还到_heap中。
        - 因为我最终版本没有对_heap进行数据结构的加工，所以归还页面时仅需对页面上锁，清空其中的一些内容就行了，**不需要lock全局大锁**，所以我的kfree速度是很快的。
- 一些尝试：
    - 尝试 1 ：
        - 在get_free_obj函数中，对bitmap的访问查找使用了__builtin_ffs函数以提高速度，但是提交给oj总是easy test都过不了，而本地测试没有测出什么bug，不得以放弃。
    - 尝试 2 ：
        - 对_heap进行包装，初始化每个page并将所有未分配的page连接成链表，这样每次向堆区申请内存时就从链表头部划出几个page出来，每次归还页面时就从链表头部插入。
        - 理论上这样的分配速度是比之前_heap.start开始遍历要高的，但是由于代码逻辑更复杂，只能过easy test，bug到现在还没调好，不得以放弃。

# L2_Report

- 数据结构
    - 我将task的状态分为：`SLEEP`，`RUN`，`WAIT`，分别表示不在运行、正在运行、被阻塞等待。任意时刻仅有标记为`RUN`的`task`在`cpu`上运行。`task_t`的结构为：

```
typedef struct task{
  union{
    struct{
      int pid,cpu,status;
      const char *name;
      void (*entry)(void*);
      void *arg;
      _Context *context;
      list_head list;    //内核中所有task所属的list
      list_head sem_list;  //属于同一个信号量的task list
      uint32_t canary;  //金丝雀
    };
    uint8_t data[TASK_SIZE];
  };
}
```

    - 信号量`sem_t`的结构为：

```
typedef struct semaphore{
  const char *name;
  int count;
  spinlock_t lock;
  list_head blocked_task;  //被阻塞的task
}sem_t;
```

在`sem_wait`时，如果当前`count==0`，则将当前的`task`加入到`blocked_task`中，标记为`WAIT`，然后`yeild`进入中断处理。在`sem_signal`中，如果发现`blocked_task`中元素不为空，则释放一个`task`，将其状态标记为`SLEEP`。

- 调度实现
  - 我第一次写的时候是在全局`task_list`中遍历寻找可用的`task`，这样就会使得经常性地跨处理器调度，造成虚拟机卡死和重启。后来我就将每个`task`绑定到对应`cpu`上，调度时，首先从`current`结点之后的`task`中寻找状态为`SLEEP`的`task`，若不成功，则从该`cpu`的`task list`头部开始寻找。与`os_trap`和`kmt`有关的函数都用一把大锁锁住，保证了原子性。这样可以过3个test。看了一些RCU的算法描述，没能想到怎么在这里实现，可能要大改数据结构，于是放弃。