OSLAB实验报告

周晓 191240080
匡亚明学院

L1实验报告

数据结构

增加的文件为

- debug.h：debug宏
- spinlock.h/spinlock.c：根据xv6实现的自旋锁
- mm.h/mm.c：内存分配所需的数据结构以及相应操作

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

L2实验报告

数据结构

增加的文件为

- os.h/os.c：定义了os_irq，中断请求数据结构，实现了中断处理程序os_trap和注册回调函数os_on_irq

- kmt.h/kmt.c：多线程调度数据结构定义和函数实现
- sem.h/sem.c：信号量数据结构定义和函数实现

OS模块

os_irq的实现为

```
struct os_irq {
    int seq, event;
    handler_t handler;
    struct os_irq *next;
};
```

使用链表存储在中断时调用的callback，其中os_trap基本仿照讲义上的实现，os_on_irq即根据seq在链表对应位置插入一个新的结点。

KMT模块

task的实现为

```
struct task {
    struct {
      const char *name; //名字
      int state; // task的状态
      int id; // task的id
      int time; // 用于简单判断优先级
      int cpu; // 下一个应该被调度到的cpu，-1时可以被任何任何cpu调度
      struct task* next; //下一个task
      Context *context;  //上下文
    };
    uint8_t* stack;  //内核线程栈
};
```

其中task的状态设置为

- BLOCKED：线程被阻塞，当信号量睡眠时设置其为该状态
- RUNNABLE：可以被调度
- RUNNING：正在运行
- HEAD：链表头设置的状态，无实际含义。有且仅有链表头一直为该状态
- DEADED： 被teardown的task，再也不会被cpu调度

调度策略基于Round-Robin调度。使用一个链表来存储所有task，链表头为task_head，其状态为HEAD，不存储任何信息。并对每个cpu设置一个idle_task，当没有task需要cpu时便调度idle_task，其entry是

```

```

```
void ientry() {
    iset(true);
    while(1) yield();
}
```

- kmt_create时将新建的task加入链表尾部
- kmt_teardown时将task的状态设置为DEADED，并使用pmm->free释放分配的内核线程栈
- kmt_schedule时，如果task的状态为DEADED，则将其从链表中删除。选择下一个调度的task时，使用Round-Robin调度，从该task的下一个开始找状态RUNNABLE的task。为了保证特殊的正确性（即在线程数不过多的前提下，要求每个可运行的线程，给定足够长 (例如数秒) 的时间，能够被调度到每个处理器上执行），我在task结构体设置了一个cpu，表示下一个应该被调度到的cpu，-1时可以被任何任何cpu调度，task每次被调度时将cpu设置为(cpu_current() + 1) % cpu_count()。为了防止线程饿死的情况出现，在task结构体中设置了一个time属性，每次被调度时time++，time低的task优先度更高

SEM模块

semaphore的实现为

```
struct semaphore {
  struct spinlock lock;
  const char *name;
  int value;
  int head_idx, tail_idx;
  task_t* queue[QLEN];
};
```

使用一个队列来存储该信号量控制的相关线程，在sem_wait时如果value < 0，那么在队列尾部插入，并yield转换到另一个线程。sem_signal时唤醒头部的线程

遇到的bug

在分配内核线程栈时发现怎么都获取不了锁，最后发现是实现spinlock时就有的bug，原先的数据结构为

```
struct spinlock {
    bool locked;
    const char *name;
    int cpu;
};
```

在获取锁时的操作是

```
while(atomic_xchg((int *)&lk->locked, 1) != 0) ;
```

最后把locked该成int类型即可