Lab 6 实验报告

- 于凡奇 18307130182

1 队列的实现

总体上搬运了 linux 中 list.h 的写法实现了双向环形链表，个人感觉写法十分巧妙且通用。

1.1 结构体的定义

```
// In list.h
struct list_head {
    struct list_head *next, *prev;
};
```

```
// In buf.h
struct buf {
    int flags;
    uint32_t blockno;
    uint8_t data[BSIZE];

    /* TODO: Your code here. */
    struct list_head node_buf;
};
```

`list_head` 只是起到指针的作用，却也是任何类型的链表都必须有的部分，因此 `list_head` 可被用于包括 `buf` 在内的各种结构体类型中，并为它们提供了统一的接口和操作方法。

1.2 队列操作方法

这里根据需求实现了尾部追加、删除、检查是否为空、取队头等操作，其中部分用普通函数实现，部分用宏定义实现。用函数实现的操作比较常规，在此不再赘述。下面主要分析几个利用宏定义的巧妙实现。

```
/**
 * container_of - cast a member of a structure out to the containing structure
 * @ptr:        the pointer to the member.
 * @type:       the type of the container struct this is embedded in.
 * @member:     the name of the member within the struct.
 *
 */
#define container_of(ptr, type, member) ({                  \
    void *__mptr = (void *)(ptr);                           \
    ((type *)(__mptr - ((size_t)&((type *)0)->member))); })
```

`container_of` 通过一系列“骚操作”，可以通过结构体的成员找出它的母结构体。在我们的实验中，它可以通过一个 `list_head` 指针返回包含它的 `buf` 结构体。事实上其原理也并不复杂，主要是借了宏定义的方便计算出 `member` 在结构体中的相对地址，再用 `ptr` 减去相对地址得到结构体的起始地址。

```
/**
 * list_entry - get the struct for this entry
 * @ptr:    the &struct list_head pointer.
 * @type:   the type of the struct this is embedded in.
 * @member: the name of the list_head within the struct.
 */
#define list_entry(ptr, type, member) \
    container_of(ptr, type, member)
```

在 `container_of` 的基础上就很容易实现列表元素的获取了，而我们需要的“取队头”操作 `list_first_entry` 的实现也是类似的，在此不再赘述。

2 sleep & wakeup

```
void
sleep(void *chan, struct spinlock *lk)
{
    struct proc *p = thiscpu->proc;

    if (lk == NULL || !holding(lk)) {
        panic("sleep: no lock held\n");
    }
    if (lk != &ptable.lock) {
        acquire(&ptable.lock);
        release(lk);
    }
    p->chan = chan;
    p->state = SLEEPING;
    sched();

    p->chan = 0;
    if (lk != &ptable.lock) {
        release(&ptable.lock);
        acquire(lk);
    }
}
```

```
void
wakeup(void *chan)
{
    struct proc *p;

    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; ++p) {
        if (p->state == SLEEPING && p->chan == chan) {
            p->state = RUNNABLE;
        }
    }
    release(&ptable.lock);
}
```

不考虑锁的话，`sleep` 是一个简单的 设置进程状态 => 发起调度 => 恢复进程状态 的过程；`wakeup` 的任务是把相应频道上的所有睡眠进程状态改为可执行，等待下次调度时再将进程真正唤醒。加上锁之后考虑的会更复杂一些。为了防止 `wakeup` 在 `sleep` 执行过程中唤醒，在 `sleep` 中需要先拿到 `ptable.lock` 再释放 `lk` 。当然，这样同时持有两个锁在某些条件下可能引发死锁的问题。

不过经过分析，在本实验的代码逻辑中不会出现死锁。唯一可能并行执行的两个线程是分别以 `sd_intr` 和 `sd_rw` 为开始的，它们申请、释放锁的顺序如下表所示。

Function in Thread 1	Operation	Function in Thread 2	Operation
sd_intr	acquire sdlock	sd_rw	acquire sdlock
wakeup	acquire ptable.lock	sd_start (sleep)	acquire ptable.lock
	release ptable.lock		release sdlock
sd_intr	release sdlock	sched	release ptable.lock
			acquire ptable.lock
		sleep	release ptable.lock
			acquire sdlock
		sd_rw	release sdlock

仔细分析可以发现，两个进程唯一需要同时拥有两把锁的时刻是在最初的两个请求处，但此时它们申请锁的顺序是一样的，因此一定有一方能获得两把锁并顺利执行下去而不会出现“持有并等待”的情况。其余任意时刻下它们各自都只需要持有一把锁，不存在死锁的问题。

3 sd 相关函数的设计

3.1 sd 函数的实现

在 `sd_init` 中对列表（队列）和 `sdlock` 进行了简单的初始化，`sd_intr` 中完全参考了注释中的写法，在此就不贴代码了。`sdrw` 的逻辑也比较简单：当队列为空时，向队尾追加并调用 `sd_start` ；当队列非空时，只向队尾追加而无需调用 `sd_start` ，因为在当前请求处理完后 `sd_intr` 会自动为队列中下一个元素发起请求。做完这些后 `sdrw` 在对应的频道上睡眠，直到 `sd_intr` 完成请求后将它唤醒。由于涉及到共享队列 `sdque` 的操作，这里需要获取 `sdlock` 。

```
void
sdrw(struct buf *b)
{
    acquire(&sdlock);
    if (list_empty(&sdque)) {
        list_add_tail(&b->node_buf, &sdque);
        sd_start(b);
    } else {
        list_add_tail(&b->node_buf, &sdque);
    }
    sleep(b, &sdlock);
    release(&sdlock);
}
```

3.2 sd 函数的调用

根据许大爷提供的思路，我将 `sd_test` 的调用写在了 SYS_exec 的系统调用中。在系统启动时开启4个 initcode 进程，每个CPU上都会运行一个进程，且都会进入 SYS_exec。在此作一判断，只让 CPU1 上的进程执行 `sd_test` ，其余如常。另外对 `initcode.S` 作一修改，若从 sys_exit 返回则进行一个无限循环（充当运行在用户态的 idle process）。

`sd_init` 仅由 CPU0 在初始化时调用一次。

4 MBR 的解析

```
struct buf b;
memset(b.data, 0, sizeof(b.data));
b.blockno = 0;
b.flags = 0;

sd_start(&b);
sdWaitForInterrupt(INT_READ_RDY);
uint32_t* intbuf = (uint32_t*)b.data;
for (int done = 0; done < 128; )
    intbuf[done++] = *EMMC_DATA;
sdWaitForInterrupt(INT_DATA_DONE);
disb();

uint32_t partition2[4];
memcpy(partition2, &b.data[0x1CE], sizeof(partition2));
cprintf("- sd init: Partition type ID: 0x%x\n", partition2[1] & 0xff);
cprintf("- sd init: LBA of first absolute sector: 0x%x\n", partition2[2]);
cprintf("- sd init: Number of sectors in partition: 0x%x\n", partition2[3]);
```

这一部分仿照 `sd_intr` 中的写法，通过 `sd_start` 开始读请求后等待中断，随后从 `EMMC_DATA` 中逐个读出数据到 `b` 中。对 `b` 中数据的解析按照 MBR 的标准，可以读出分区类型、LBA等等。`INT_READ_RDY` 和 `INT_DATA_DONE` 会触发 CPU 中断，但由于 `sd_init` 在 CPU0 上的内核态中执行，不会引发额外的陷入。

5 性能分析与优化

下面主要从多核调度的角度进行简要分析。

当前的（最快的）实现方法是：开始时4个CPU各跑一个线程，其中 CPU1 上的线程执行 `sd_test`，其余3个线程在用户态进入无限循环。在每次 timer interrupt 时，各线程执行 `yield`，进行重新调度，也就是说 `sd_test` 在此后可能跑在任意一个 CPU 上。当 `sd_test` 线程进入睡眠时，原本执行它的 CPU 进入空闲状态。因此在中断发生，`sd_test` 线程状态被改为 runnable 后，这个空闲的 CPU 可以马上将其接过来执行，而无需等待下一次调度点。

当系统中的线程数不小于5时，会有两方面的问题使测试速度下降。

- `sd_test` 线程进入睡眠后第5个进程会马上顶上来占用其 CPU，这时即使立即 `wakeup` `sd_test` 线程也需要等待下一个调度点（时钟中断）到来才能让它真正跑起来。由于我们系统中 timer 的时间片设置的非常长，这个等待时间是巨大的。具体表现即是速度会低至 0.0 MB/s。这个问题可以通过实现跨核中断（暂未实现）来解决，即在 `wakeup` 时通知其它核立即进行重新调度。
- `sd_test` 线程未必总是能获得调度机会，若将所有进程一视同仁，则总进程数越多其被调度的机会越小。这个问题可以通过设置进程调度优先级来解决。在本实验中，我实现了一个简单的**多级反馈队列**调度器，支持2种优先级的进程调度。由于 `sd_test` 进程总是会主动睡眠，不会用完时间片，它的优先级永远不会下降，因此可以保证它相对于其它进程的优先调度。

当系统中的线程数小于4时，某一时刻 CPU0 上可能没有 idle process 在运行，故无法马上响应中断，这会带来性能的下降，对此我们可以将一个 idle process 绑定在 CPU0 上。具体实现上，为 `struct proc` 增加一个 `cpus_allowed` 域，其中第 i 位为1表示该进程可以运行在 CPUi 上，调度器调度进程前将进行检查。经过测试，这一方法对于线程数为3的情况有着巨大的提升。