L1实验报告

架构设计

我把整个堆区分为两部分：后半部分有13*8个`kmem_cache`，其中有一个链表头，13对应满足$2^n \leq 4096$的13个n，8是CPU的数目。每个处理器上每种大小的分配请求都被对应到一个`kmem_cache`中，也就是一条链表。

堆区中除`kmem_cache`以外的部分划分成8KB的页，每页由1KB的header和7KB的data部分组成。header中除必要信息外还包含一张bitmap表明该页中哪些内存单元可用。当一页用于分配1B的内存空间时，最大可以拥有$7 * 1024 = 7168$个内存单元，每个内存单元需要1位存储，因此共需$7168/8 = 896$字节作为bitmap，比起定义的header大小1024还有不少富余用于其他信息。

分配方法

接收`alloc`请求后，将`size`向上取整到最近的2的幂次，如果$kmem_cache[cpu][n]$中没有页或者所有页面已满,分配一张新页插入链表头部，并作为当前页，否则找到链表中第一张未满的页作为当前页。

fast path:不需要分配新页，只需要在当前页中找到一个空闲的内存单元。我采用了next fit的方式，也即指针从该页上一次分配的位置开始继续遍历所有位置，效果优于每次都从头开始寻找。需要对当前页上锁。

slow path:从`freePageHead`取一新页，初始化后插入链表头部，之后回到fast path。需要对`freePageHead`上锁。

释放方法

接收free请求后由地址获得所在的页，清除bitmap对应位。

实验心得

1. 我原本采用header中所有元素大小之和作为页头的大小，但是后来发现这样一来由于对齐的缘故，页头实际占用的空间比我设想的要大，从而一页的大小超出了`PageSize`，后来我通过将页头大小设置为给定的上界(1KB)解决了这一问题。
2. 页头的加入还导致了对齐问题的出现，例如对于2KB的分配请求，就不能直接把1KB处的data域起始地址分除去。我的解决方法是引入了`data_align`变量作为data域中考虑对齐后可供分配的第一个地址。
3. 通过面向OJ二分调整锁的位置可以确定到底是哪一部分引起了并发bug(误)

遇到的bug

1. 我使用一个`new_page`局部变量来标示一次`alloc`中是否要申请新页，但是没有初始化该局部变量导致非预期结果，事实上只有全局变量才会被初始化为0。
2. 在双线程运行时，尽管已经给每一页上了锁却还是出现double allocation的问题。经过调试发现，空闲页链表的起始位置`freePageHead`同样会导致数据竞争，因此需要🔒的保护。
3. 我在使用全局🔒保护整个`alloc`和`free`时发现可以通过Easy test 2, 但是后来发现这是一种错觉，我其实漏掉了一个return分支上的unlock，导致free被卡死。而恰恰Easy test 2是一个只要在指定的timeout内不出现error就能通过的样例，这些阴差阳错导致了我的误判。这个故事告诉我依赖OJ是不可取的，还是需要在本地测试，这个bug在本地是很好测出来的。

4. 另外一个问题也跟Easy test 2有关，我尝试了一种两个链表的实现，也就是对每个`kmem_cache`维护一个满页的链表，一个有空闲位置页的链表，这种方法理应效率更高。然而在我只实现`alloc`测试时，发现效率较低的那种能够通过，效率高的反而因为failure when low pressure而WA。询问蒋老师后得知效率低的因为在timeout时间内执行的分配较少，没有触发25% pressure的界限从而通过。
5. 在实现设置bitmap的`setUnit`函数时，它看起来应该长这样：

```
void setUnit(page_t* page, int num, bool b)
{
  assert(b == 0 || b == 1);
  if (b == 1)
    page->bitmap[num / 64] |= ((uint64_t)1 << (num % 64));
  else
    page->bitmap[num / 64] &= ~((uint64_t)1 << (num % 64));
}
```

   但是我在相当长的一段时间内，都把以上条件中的if和else写反了，这样一来bitmap相当于没有任何效果，在每一页上永远都是从头到尾分完所有内存单元后就再也不能再次使用。但是这样的buggy实现能够通过相当一部分测试，只是在需要free的时候无法发挥作用。
6. 还是上面这段代码，我的page->bitmap是`uint64_t`数组，但是我一上来没有把左移的那个1强制类型转换成`uint64_t`，导致它只有32位，左移次数超过32就会溢出，导致函数行为不合预期。

L2实验报告

OS模块

`os_trap()`函数采用实验讲义中的实现形式，并发性方面是用一把大锁保护整个函数体。

`on_irq()`函数使用一个数组维护各个`handler`以及他们的优先级

spinlock模块

参考了xv6中的实现，使用了封装之后的`pushcli`和`popcli`函数，它们是匹配的，也就是两次`pushcli`需要用两次`popcli`来撤销。此外还实现了`holding`函数判断一个锁是否已被持有，这是用于多次lock/unlock未被上锁的锁等错误的检测。

sem模块

使用一个数组和首尾指针模拟队列，同样使用一把大锁保护整个函数。

kmt模块

我使用数组模拟的环状链表来存储当前的task.系统中维护有上一个添加的TASK的指针，当添加一个新TASK时需要修改上一个添加的TASK来维持链表的环状结构。

实验心得

1. 这个实验要实现的部分由多个模块组成，一种很好的调试手段就是分开每个模块单独调试。比如我构造了3个测试用例，分别用来测试自旋锁（`os_run`中用if/else结构控制，在两个处理器上运行的printf函数），kmt模块（使用`create`创建的不断打印信息的线程），sem模块（生产者消费者）。

2. 当不把任务与CPU核心绑定时：之所以会产生错误是因为在`os_trap`解开锁后的return语句时发生了中断。

`os_trap()`大致如下：

```
kmt->spin_lock(&trapLock);
  _Context* next = NULL;
  for (int i = 0; i <= MAX_INTR;i++) {
    if (INTR[i].valid == 1 && (INTR[i].event == _EVENT_NULL || INTR[i].event == ev.event)) {
      _Context* r = INTR[i].handler(ev, context);
      panic_on(r && next, "returning multiple contexts");
      if (r) next = r;
    }
  }
  panic_on(!next, "returning NULL context");
  kmt->spin_unlock(&trapLock);
  return next;
```

一旦发生了这种情况：核心0获得了任务1的上下文，并运行完11行解开了自旋锁。恰恰在此时时钟中断到来了：核心0还没有来得及return。此时如果核心1恰好进入了中断处理程序，恰好获得了任务1的上下文，并执行return从核心1的上下文恢复。此时核心0从中断中恢复之后，任务1的上下文已经被核心1破坏。

为了解决这个问题，其实就是让os_trap释放锁之后仍要保持对任务栈的占有，一种办法就是在下一次的中断处理程序中再释放上一次的任务。于是我使用了sticky位来表示一个任务被“黏”在了某个cpu上，当sticky位为0的时候才能被调度到其他的cpu.

```
_Context* scheduler(_Event ev, _Context* _Context)
{
  if (cpu_local[_cpu()].sticky != NULL) {
    cpu_local[_cpu()].sticky->sticky = 0;
    cpu_local[_cpu()].sticky = NULL;
  }
  /* ..... */
  if (current) {
    current->context = _Context;
    current->sticky = 1;
    cpu_local[_cpu()].sticky = current;
  }
 /* ..... */
  return current->context;
}
```

不过这个实现尚还存在一些问题，因此我在提交中使用的是绑定cpu的scheduler.

遇到的bug

1. 在实现spin_lock时我参考了xv6的实现，不过一个问题是我仿照它把未使用的锁的CPU序号值设为0，然而xv6里这个序号值是指向cpu结构体的指针，设为0即为空指针。但我们的实验里是有序号值为0的cpu的。这就导致了一个愚蠢的bug。
2. 在使用不绑定的CPU后出现了多种bug，包括重复上锁/信号量失效/重启等等。因为这种bug只会出现在生产者/消费者问题中，而不会出现在仅使用自旋锁的打印样例中，因此我怀疑可能是信号量的实现存在问题，目前还在调试中。

L3实验报告

架构设计(ufs)

我让文件系统从硬盘中1MB的固定偏移量处开始，将整个硬盘上的文件系统分为4个部分：

超级块(superblock)	32B	文件系统metadata
inode区	10KB	每个文件的metadata
FAT区	10KB	文件分配表
data区	剩余空间	存储文件内容

data区分为等大小的块作为分配的最小单元，我使用的块大小是32B.

超级块除了存储各部分的大小外，还存储了第一个空闲的inode和第一个空闲块的编号以便分配。

inode的结构如下：

```
typedef struct _dinode{
   struct ufs_stat stat;
   uint32_t firstBlock;
   uint32_t refCnt;
} dinode_t;
```

除`ufs_stat`外，还包括引用计数和第一个数据块的编号，其余数据块编号通过FAT即可获得。

挂载时，根据硬盘中记载的信息构造出内存中的数据结构，用于目录搜索。树状结构的每个结点如下：

```
typedef struct _inode inode_t;
struct _inode {
   uint32_t dInodeNum;
   char name[28];
   uint32_t firstBlock;

   inode_t* parent;
   inode_t* firstChild;
   inode_t* nxtBrother;
};
```

每个结点存储了这个文件所对应的硬盘上的inode编号以及文件名。为了降低读硬盘的次数，还存储了文件起始块的编号。

实验心得

1. 对于链接的处理

   一开始，我在树状结构的结点中存储的其实是硬盘inode内容的一个超集，但是这样一来对链接的文件就会有两个不同的叶子与之对应，修改无法保证同步，这显然不符合hard link的要求。所以我索性只把inode信息存放在硬盘上，避免了hard link带来的同步问题。

   此外，在mkfs时也会涉及到硬链接，对于所有指向同一inode的文件，只需要存储其一份拷贝即可。所幸`struct stat`结构里已经包含了成员`st_nlink`，可以识别硬链接。对于硬链接的文件，只要记录已经处理过的inode编号，下次遇到时直接映射到同一inode写入文件系统即可。

2. 对于打开的文件，我原本对于每个进程维护一个打开文件列表。但是根据APUE所述：每个进程在进程表中有一个记录项，记录项包含一张打开文件描述符表，每个描述符占有一项，包含文件描述符标志(file descriptor flag)以及指向一个文件表项的指针。所有进程共享一张文件表，包含所有当前打开文件的信息。每一项包括文件状态标志(file status flag)，文件偏移量以及指向inode的指针。所以我把文件系统结构改成了这样。这样做的好处是让fork之后，父子进程的文件描述符能够指向同一个打开文件列表项。

3. 有一些特殊文件操作的情况需要考虑：`lseek`和`write`结合使用允许生成一个具有空洞的文件，也就是跳到当前文件大小之后的偏移量再写入。只改变offset是不会影响文件大小的，在write之后文件大小才发生变化，而且要把空洞的大小也算进去，这些实现应该参照UNIX文件系统的行为并仿照实现。类似的还有：

    1. `open`只创建路径中的最后一级文件，而不会创建不存在的目录
    2. 同一进程打开同一文件两次是合法的，而且会生成两个文件描述符，同时偏移量也不会共享。

    这些行为我原本都无法说出正确结果，只有实验之后才能对我的文件系统进行实现。

    此外，文件中的空洞并不要求在硬盘上占用存储区，不过我的实现没有这么复杂，只是简单的在空洞部分加上了空的数据块而已。

4. 有一个关于路径处理的小问题：路径中的'/'应该作为上下两级目录的分割符使用，但是根目录的名字又刚好是"/"，此外，路径的最后可以带'/'也可以不带，这就给字符串解析带来一些困难。我的解决方法是把所有路径统一为以'/'结尾的形式，并用一些特判处理根目录名"/"的情形。

遇到的bug

1. 在调试`dev`模块时，我遇到这样一个问题：如果不创建任何线程（即只有`dev->init()`创建的`dev_input_task`和`dev_tty_task`两个线程），此时这两个线程会因为`read()`函数的信号量而双双陷入等待，造成`schedule()`函数的死循环。为了解决这个问题只要加入另一个trivial的线程供调度使用即可。

    那么为什么这两个线程会被trivial的线程唤醒呢？其实并不是这个线程唤醒了它们，而是在初始化输入设备时注册了键盘中断和时钟中断的处理函数：

    ```
    os->on_irq(0, _EVENT_IRQ_IODEV, input_notify);
    os->on_irq(0, _EVENT_IRQ_TIMER, input_notify);
    static _Context *input_notify(_Event ev, _Context *context) {
    kmt->sem_signal(&sem_kbdirq);
    return NULL;
    }
    ```

2. 在实验中我还遇到了一些比较低级的问题，比如`klib`中`printf`的祖传bug，把局部变量地址存在目录树里导致悬浮指针，以及在一个没有被重置的硬盘镜像上debug等等。

结语

这次实验让我对课本上的知识有更深入的理解，比如文件系统中一些数据结构的相互关联（所有进程共享的打开文件列表，进程独占的打开文件描述符表），文件系统中的目录如何实现（其实和文件没有区别，只是存储了文件名-inode号的二元对作为dirent而已），mkfs和mount究竟做了哪些工作，硬链接和软连接的区别，inode存储了文件的metadata，等等。不过也有一些特性没有在我的实现里得到体现，比如inode使用直接寻址和一次/二次寻址快速访问小文件，同时兼容大文件，体现了文件系统中大部分为小文件这一性质。日后有机会我希望继续加以完善，加深对文件系统的认识和了解。