OS Lab实验报告（至L3）

姓名：张浩南
学号：181860134

L1

架构设计

我的代码设计参照课件上提到的bitmap表，每次分配一个8KiB的page，其中7KiB为预备分配的存储空间，剩余的空间用于存储cpu编号，bitmap数组和分配空间的大小

```
struct page_info{
  uint8_t page[page_size];//4+2+1=7KiB的page，最多存的块数少于4096
  int p_cpu,type;//归属的cpu和分配大小\2 4 ... 4096
  uint64_t bitmap[64];//64*8=512B=4096 bit
  struct page_info* next;
};
```

其中锁的实现是普通的自旋锁，其中一个用于slow path，另一个用于fast path

每次fast alloc先看当前cpu以及对应的分配大小类型的页是否有空闲，若无则slow alloc申请一个page

free并不会回收页表到堆区中，而是继续留在当前归属的cpu和分配大小的属性。

fash path

用一种slab表，即将小内存分配分类（8B,16B,32B），然后对每一类分配页面

slow path

考虑自旋锁+记录目前可分配堆区的末尾分配管理

遇到的bug

一开始自旋锁的unlock实现没用 `xchg` ，导致出现了RE等错误
后来在alloc和free加上大锁后出现了“低于1Mop/s”的错误，但去掉大锁后又出现RE等错误导致连Easy test都过不了。后来发现在修改bitmap这样的全局变量时可能会导致数据冲突从而出现double allocation，所以在这些语句上加入lock之后，就可以过了

L2

锁、信号量与线程实现

自旋锁实现时借鉴了L2讲义中介绍的pmm模块中中断相关代码，即 `lock()` 时先在结构体中存储当前中断状态再关中断+上锁， `unlock()` 中先解锁，再根据之前存储的状态来判断是否要开中断

信号量用自旋锁+队列管理task的方式实现

线程task的栈和name都用 `pmm->alloc/free` 进行动态分配，防止过多task导致空间溢出

多CPU调度问题

在多CPU调度问题上，我与姚嘉和同学进行交流并各提出一种方案。我的方法是每个task对每个cpu都开一个栈空间，然后在 `kmt_schedule` 时将任务“当前cpu对应栈空间中的内容”复制到“下一个cpu对应栈空间”中，通过各自独立的栈空间来避免stack race情况；而姚嘉和同学的想法则是对“在 `kmt_schedule` 中某一cpu被换下的task”打标记，使得其他cpu不能在有标记的时候调度该任务，直到该cpu再次触发 `os_trap` 进入中断时，才把该任务的标记去掉。这种类似“延时处理”的方法同样避免了stack race

而在仔细思考后，我发现开多个栈空间的做法无论从时间复杂度（每次schedule都要将栈内容赋值）、空间复杂度（ `ntask*ncpu` 个栈）还是实现复杂度（更改context中的栈区间与 `esp,ebp` 等寄存器内容）上都显得较为困难和繁琐，而姚嘉和的做法相比之下则更为优秀

关于thread stave

在实现多CPU调度并在本地完成对括号序列、putc测试和一些官方设备测试后，我的提交却总会出现thread starvation的问题，检查和思考后我认为可能是需要让锁对需求的各进程进行公平资源分配，而询问jyy后得知并不需要这么做。

后来我觉得可能是采用课件代码

```
current->cpu = (current->cpu + 1) % ncpu;
```

从而导致current在下一cpu较忙碌时（如处于锁等关中断区域内）长时间内无法被调用，出现进程饥饿的问题

解决方法是删去cpu belong，使在顺序查找的情况下任意cpu都可以调度空闲状态下的进程

L3

文件系统结构

整个磁盘中的文件系统分为三块：inode区(32K)，跳转表(256K)和数据区
inode区的每个项保存对应文件的相关信息（大小，权限，inode编号等）；
数据区中每4KiB分成一个block，跳转表由若干个32位整数组成，用来存储各个block的链表结构。
如一个文件内容存储在3个block上，编号分别为0,2,3，则跳转表的0号位置指向2，2号位置指向3

这里我将`/proc,/dev`作为`ufs`系统的一部分进行存储，只是将其权限定为只读，并对`zero,null,random`等设备做了特殊处理

一些代码实现

代码中对路径解析`find_path`(给定一个字符串，返回最终指向文件/目录的inode)和查询文件名`dir_to_inode`（给定一个文件名字符串和一个目录，返回该目录中对应文件名的inode）等进行了功能封装

inode结构中存储的links包含执行`link`API被链接的次数和当前持有文件描述符的个数，只有当被链接次数和文件描述符个数都为0时，该文件才会被删除，这样防止了open之后unlink，导致文件描述符无法访问对应文件的问题

inode区采用缓存+磁盘的两重存储，发生修改时同时在pmm内存和磁盘中进行修改，且内存中的inode结构采用链表进行存储

```
struct list_inode{
    inode_t x;
    struct list_inode* next;
};
```

写入磁盘不必写入next指针，且在磁盘中用0xff填满最后一个inode指向的下一部分区域，从而在`vfs->init()`读取磁盘时方便判断是否读完所有inode

遇到的bug

一开始未关注到讲义中关于”自然的实现“，后来在福利test中发现有`chdir("..")`这样的”符合Linux语法“的路径，因此在目录中添加这一ufs_dirent(name=”..”,inode=父目录的inode)
在福利4.4发布后在本地进行`workload.inc`的测试，修改`chdir("..")`的问题后可以正常创建目录/文件并遍历文件系统（正确执行workload和traverse），但无论是否加信号量，OJ上单线程单cpu的hard test 1均会`missing file 51/51`，后来jyy说我的程序运行很慢，我才发现是因为我每次将inode结构写回磁盘时都会将所有文件的inode写回，导致速度极慢，然后我又花了两个小时把整个inode结构更改，过了第一个hardtest