Lab5：文件系统

Lab 5

- 磁盘文件系统结构
- 文件系统
- Spawning Processes
- Shell控制台

硬盘文件系统结构

- JOS所有文件的元数据存储在哪?
  - 没有inode结构，直接存储在唯一的directory entry中
- 最大支持多少磁盘容量?
  - 在JOS的实现中是3G
- JOS中Sectors的大小是多少?
- JOS文件系统以什么为单位分配和使用磁盘? 大小是多少?
- JOS一个磁盘有多少扇区?

block_size:4096字节
sector：512字节

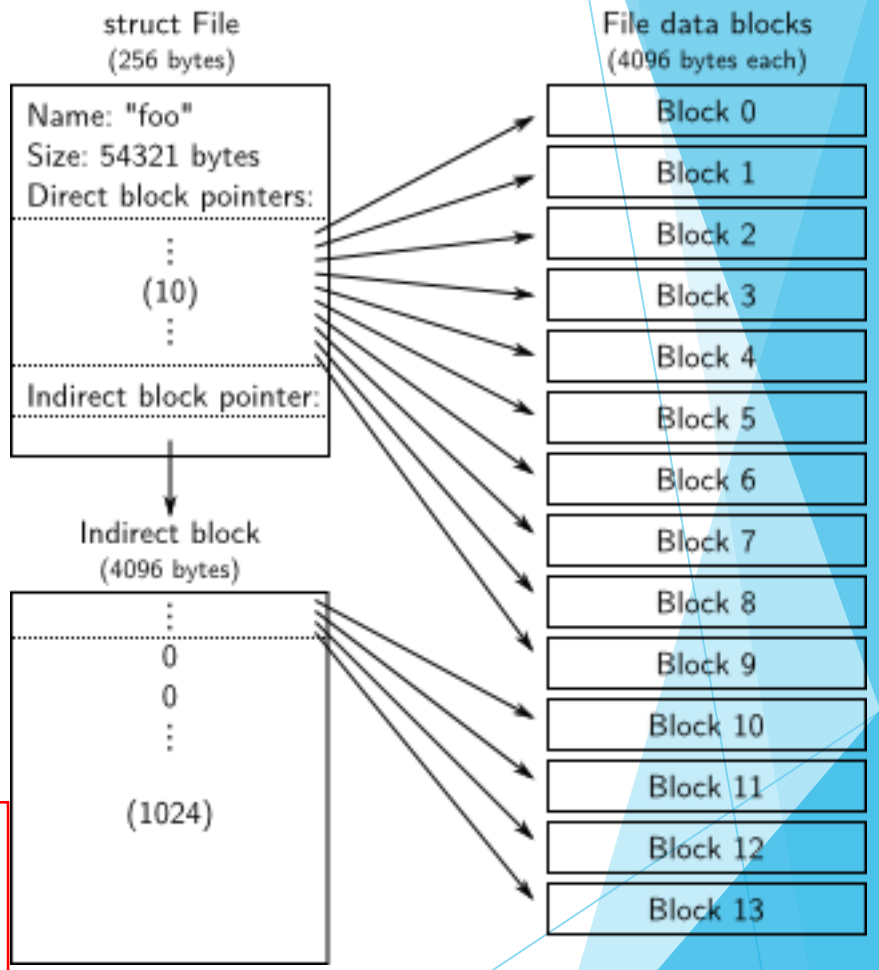
以block为单位分配

3G/512

硬盘文件系统结构

- JOS的block 0中保存了什么内容？block 1呢？
- JOS的超级块上记录了哪些内容？
- JOS文件的meta-data上记录了哪些内容？
- JOS文件系统中支持哪些文件类型？
  - 实现链接文件的关键是什么？
- JOS支持的最大文件大小是多少？
- JOS如何实现虚拟块号到物理块号的映射？

1：block0实际上都是0，如果使用那就是放bootloader
block1 superblock.
2:可用磁盘大小，根目录的数据，整个磁盘的元数据
3：文件大小 名字 block
4：目录文件和普通文件,添加文件类型就是加一个标记
5：(1024+10)*4k=4M+40k
6：<10直接映射，>10去算偏移然后取地址



FS进程

1：提供一个特殊权限标记（初始化的时候）
2：同1
3：不需要，自动保存

4.pagefault，映射到内存，写内存，flush会写回磁盘

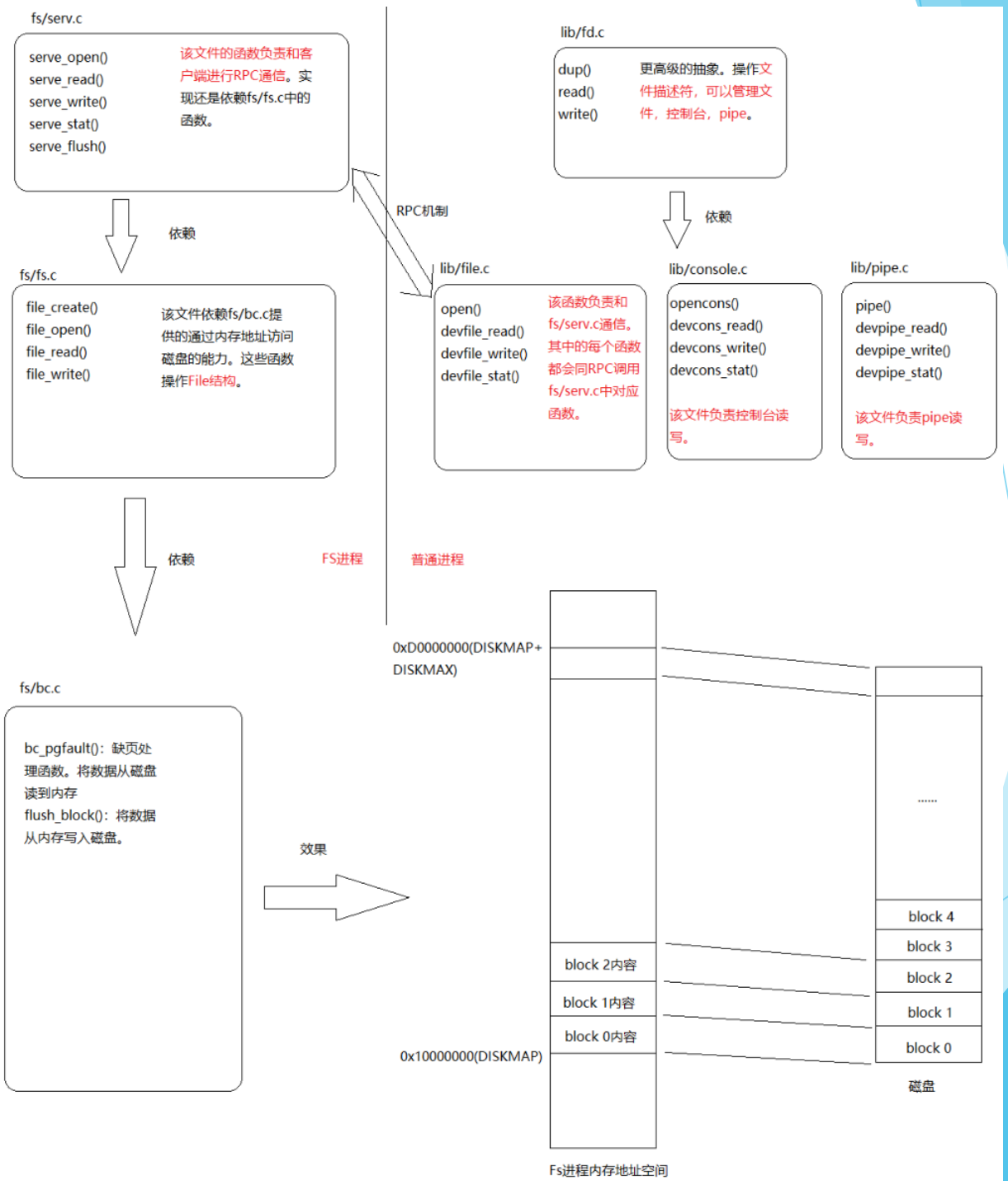
- JOS怎样让fs进程具有读写硬盘权限?
  - 怎么实现权限控制?
- Fs进程的初始化流程和普通进程有什么不同?
- 当从Fs进程切换到另一个进程时，还需要做什么来确保正确地保存和恢复这个I/O特权设置吗?为什么?
- Fs进程如何管理磁盘?
  - Read?
  - Write?
  - Flush?

FS进程——亿些细节

1:1024,由打开表数组大小确定
2：？
3：遍历数组
4：看被映射的次数
5：一整个页面？

- JOS最多允许打开多少个文件？是由什么因素决定的？
- FS文件描述符对应的Fd结构起始地址是多少？
- FS如何找到空闲的OpenFile？
- 普通进程如何找到空闲的Fd？
- 每个Fd结构占多大空间？为什么这样设计

FS IPC过程


fs/serv.c
serve_open()
serve_read()
serve_write()
serve_stat()
serve_flush()
该文件的函数负责和客户端进行RPC通信。实现还是依赖fs/fs.c中的函数。
依赖
fs/fs.c
file_create()
file_open()
file_read()
file_write()
该文件依赖fs/bc.c提供的通过内存地址访问磁盘的能力。这些函数操作File结构。
依赖
fs/bc.c
bc_pgfault()：缺页处理函数。将数据从磁盘读到内存
flush_block()：将数据从内存写入磁盘。
效果
RPC机制
FS进程
普通进程
lib/fd.c
dup()
read()
write()
更高级的抽象。操作文件描述符，可以管理文件，控制台，pipe。
依赖
lib/file.c
open()
devfile_read()
devfile_write()
devfile_stat()
该函数负责和fs/serv.c通信。其中的每个函数都会同RPC调用fs/serv.c中对应函数。
lib/console.c
opencons()
devcons_read()
devcons_write()
devcons_stat()
该文件负责控制台读写。
lib/pipe.c
pipe()
devpipe_read()
devpipe_write()
devpipe_stat()
该文件负责pipe读写。
0xD0000000(DISKMAP+DISKMAX)
block 2内容
block 1内容
block 0内容
0x10000000(DISKMAP)
Fs进程内存地址空间
.....
block 4
block 3
block 2
block 1
block 0
磁盘

如果有多个进程想修改一个磁盘File，怎么保证其内容的一致性？

- 普通进程想修改File，需要通过IPC机制借助文件系统进程实现
- 而IPC机制让文件系统进程一次只能接受一个send，所以很好的避免了同时修改同一文件的情况

JOS对不同类型的设备做了统一抽象，访问文件时如何确定设备的类型？

```
// Per-device-class file descriptor operations
struct Dev {
    int dev_id;
    const char *dev_name;
    ssize_t (*dev_read)(struct Fd *fd, void *buf, size_t len);
    ssize_t (*dev_write)(struct Fd *fd, const void *buf, size_t len);
    int (*dev_close)(struct Fd *fd);
    int (*dev_stat)(struct Fd *fd, struct Stat *stat);
    int (*dev_trunc)(struct Fd *fd, off_t length);
};
```

- 实现了哪些设备类型？

```
static struct Dev *devtab[] =
{
    &devfile,
    &devpipe,
    &devcons,
    0
};
```

Devcons是如何实现的?

- Devcons负责从命令行进行输入和输出
- 输入：devcons_read
  - sys_cgetc()
- 输出：devcons_write
  - sys_cputs

Devpipe是如何实现的?

- 创建
  - 分配两个Fd，Fd[0]只读，Fd[1]只写
  - 每个Fd都有自己的私有数据区，通过fd2data(Fd)获取
  - 私有数据区里有一个buffer，两个Fd共享同一个私有数据区
- 读：devpipe_read
  - 通过Fd[0]读共享私有数据区中的buffer
  - 如果buffer中没有足够的数据?
- 写：devpipe_write
  - 通过Fd[1]向共享能有数据区中的buffer写数据
  - 如果buffer中没有空闲区域?

Spawn的流程是怎样的?

- 从文件系统打开prog参数对应的程序文件
- 调用系统调用sys_exofork()创建一个新的Env结构
- 调用系统调用sys_env_set_trapframe()，设置新的Env结构的Trapframe字段（该字段包含寄存器信息）
- 根据ELF文件中program herder，将用户程序以Segment读入内存，并映射到指定的线性地址处
- 调用系统调用sys_env_set_status()设置新的Env结构状态为ENV_RUNNABLE

JOS如何实现父子进程共享文件描述符?

- 每个描述符独占一个页面，共享描述符 = 共享页面
- JOS中定义PTE新的标志位PTE_SHARE
- 如果有个页表条目的PTE_SHARE标志位为1，那么这个PTE在fork()和spawn()中将被直接拷贝到子进程页表
- 这样父进程和子进程共享相同的页映射关系，从而达到父子进程共享文件描述符的目的

Thanks