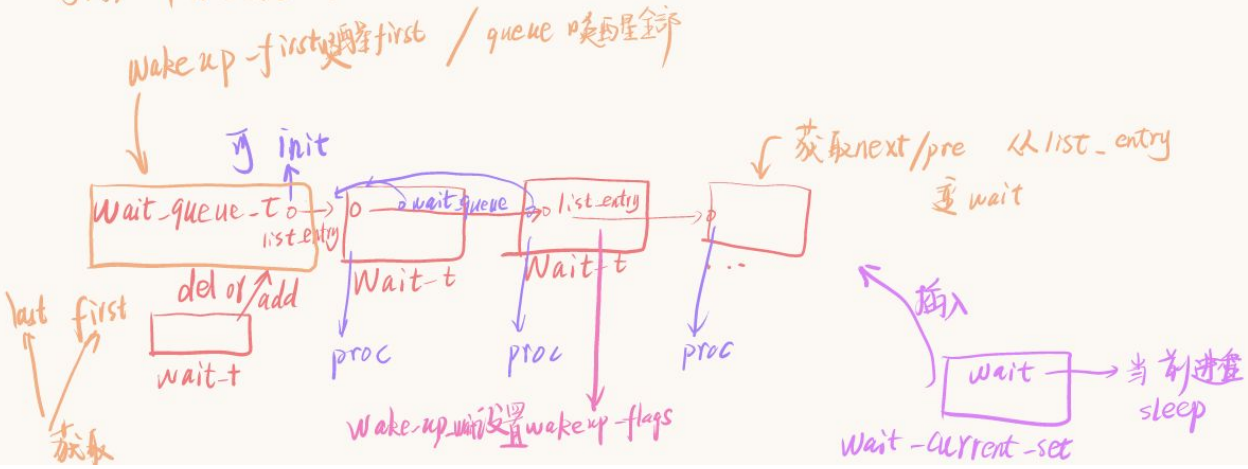
lab 8

fs : file system
sfs : simple file system
Inode : index node
VOP : virtual operation
stat : status
d_block : device_block
Iobuf : Input/output buffer
semaphore_t { wait_queue_t ; 链着 wait_t
value
引导 bootfs { 内核镜像 / 引导程序 } 内存
vfs_dev_t : 虚拟文件系统设备(如磁盘)
vdev_list : 虚拟设备(打印机…)
iob: IO buffer

数组 int 32 or 64
↑
(intptr_t) 0x7FFF1234
uintptr_t
size_t → memory
off_t → file offset
ppn_t → page number
device ✓

intr_save : 关中断
intr_store : 开中断

等待队列的数据结构:



sem.c

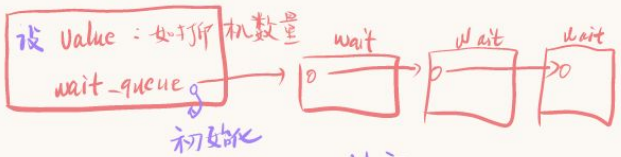
--up : V操作

--down : P操作

但不同于教材

value不可以为负

信号量 Semaphore

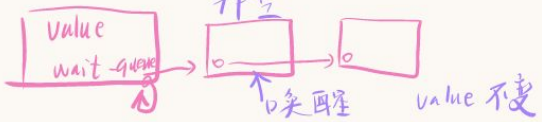


教材

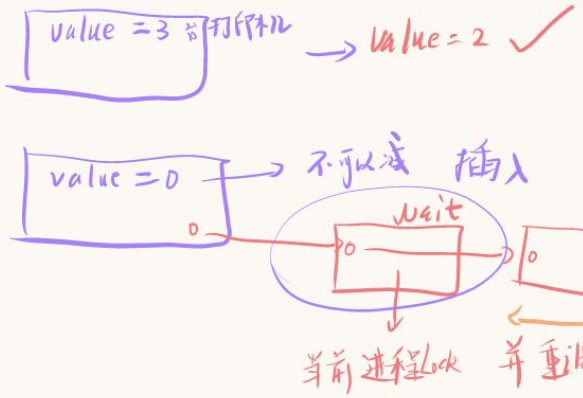
P: value--
if (value < 0)
Lock

V: value ++
if (value >= 0)
wake-up

--up



--down



举例 2号打印机

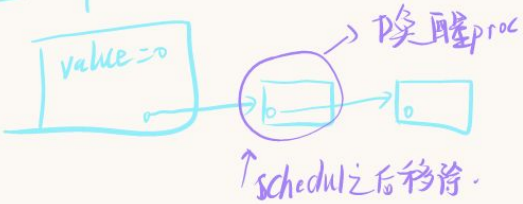
① --down

② --down

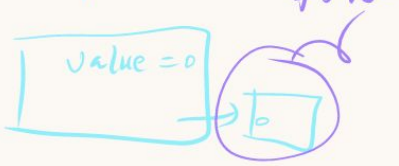
③ --down

④ --down

⑤ --up

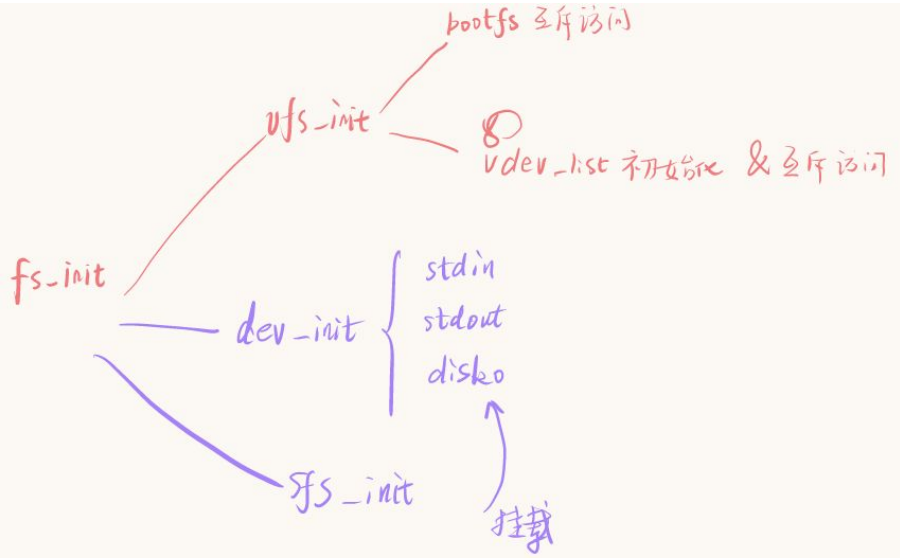


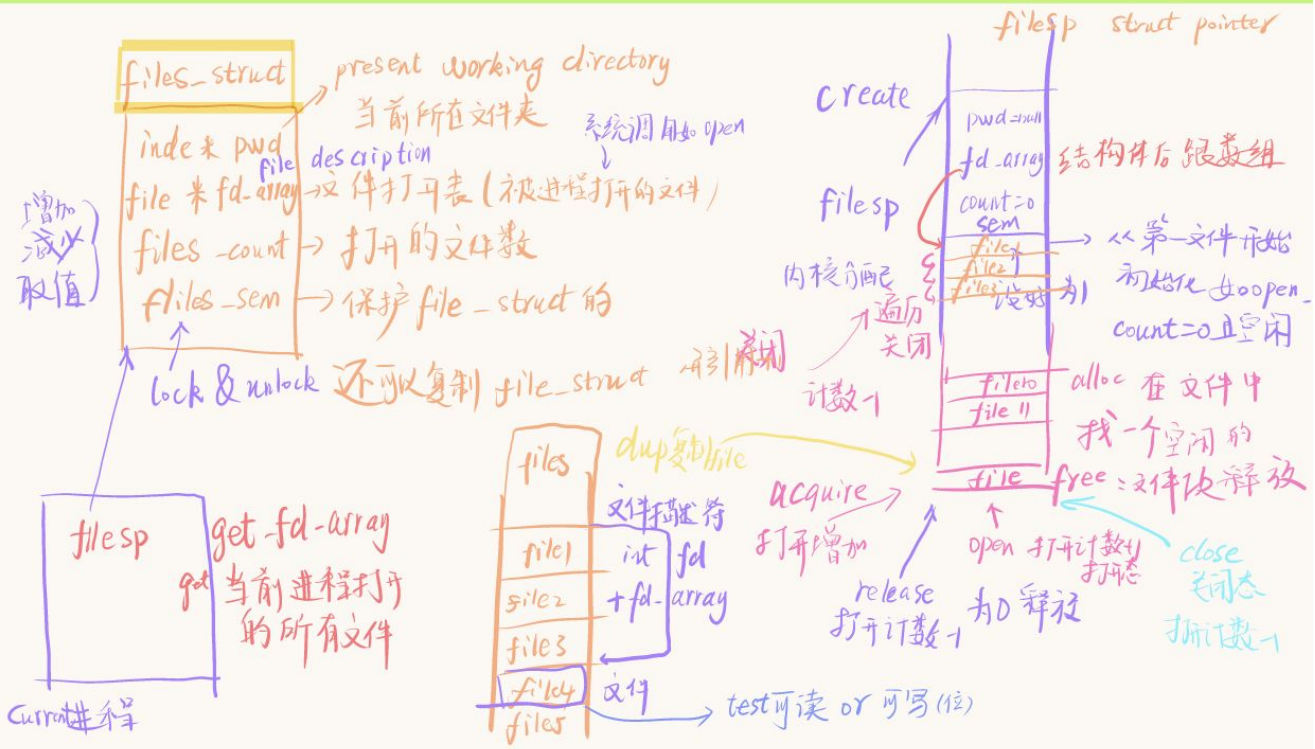
⑥ --up

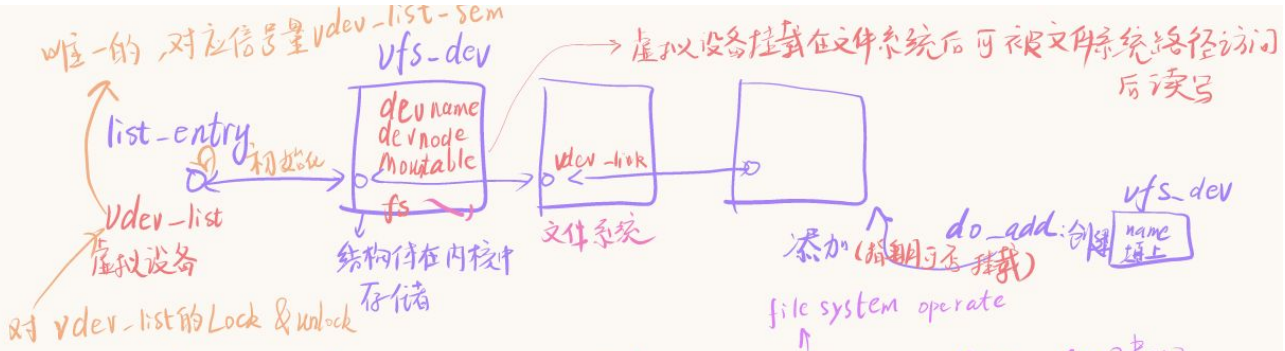


⑥ --up

① --up


bootfs 互斥访问
vfs_init
vdev_list 初始化 & 互斥访问
fs_init
dev_init
stdin
stdout
disk0
sfs_init
挂载


files_struct
present working directory
当前所在文件夹
inode 来 pwd
file 来 fd_array
description
文件打开表（被进程打开的文件）
files_count → 打开的文件数
files_sem → 保护 file_struct 的
增加
减少
取值
lock & unlock 不可以复制 file_struct
filesp
get_fd_array
get
当前进程打开的所有文件
Current进程
files
file0
file1
file2
file3
file4
dup复制file
文件描述符
int
fd
+fd_array
文件
test可读 or 可写(位)
filesp struct pointer
create
pwd=null
fd_array
结构体后跟数组
count=0
sem
从第一文件开始
filesp
内核态
遍历
关闭
为1
初始化为open_count=0且空闲
计数-1
alloc 在文件中找一个空闲的
free 文件块释放
acquire
打开增加
release
打开计数-1
为0释放
open 打开计数+1 打开态
close 关闭态 打开计数-1



cleanup：遍历 vdev_list，转为 vfs_dev，用 fsop_cleanup(vdev→fs) 清理

get_root：遍历 vdev_list 找一个设备（通过 name）返回 inode

get_devname，遍历，由 fs 找 devname　　check_devname_conflict 检查名字冲突

find_mount：遍历找可挂载设备

vfs_mount { devname / mountfunc } → 找可挂载设备 → 用 mountfunc 挂载

unmount → find 找设备

1. 写盘
2. fsop_unmount 不挂载

unmount_all：遍历设备：fsop_unmount 全部 unmount

挂载：

接一个文件系统连接到文件系统树的特定脉　使之可用

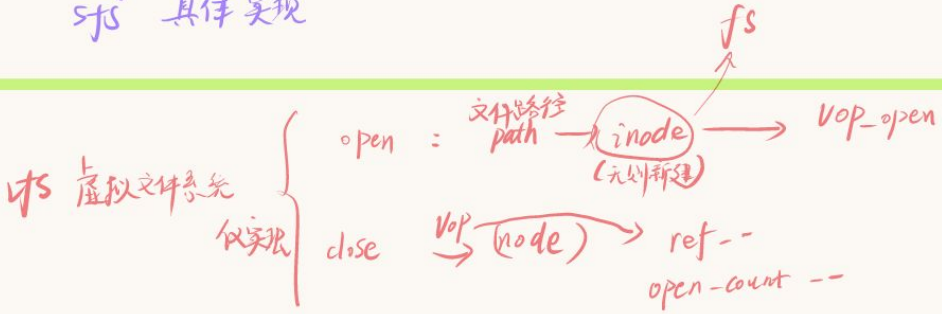
mount(dev, &(vde→fs))　记录信息（文件系统）

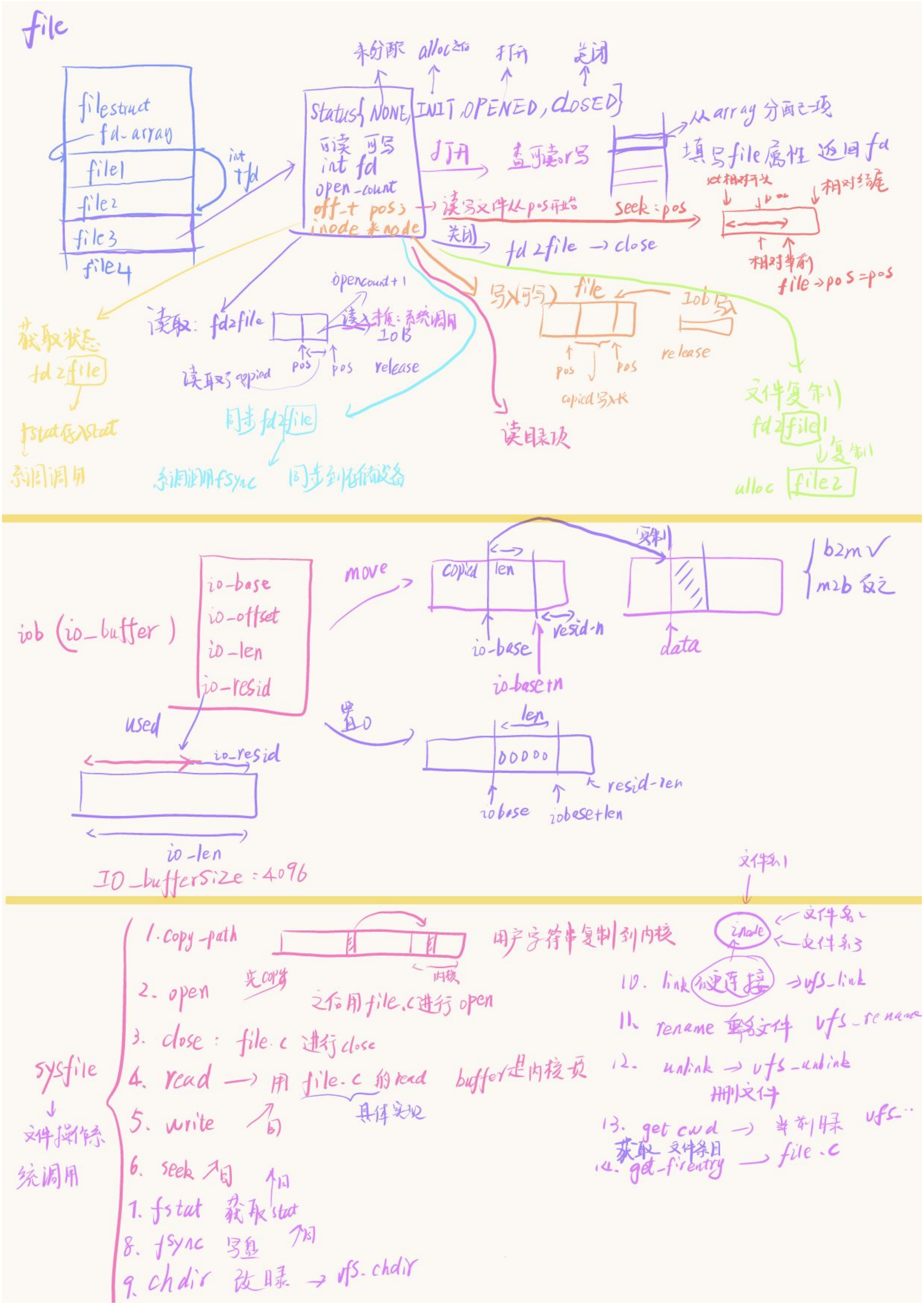
dev：文件系统挂在哪个设备

vfs 通用接口

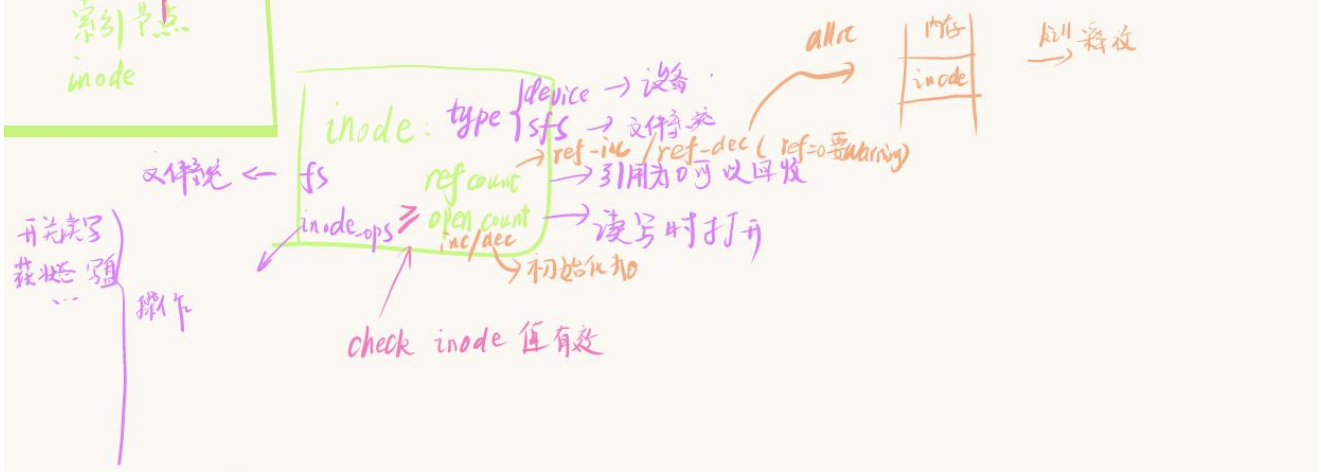
↓

sfs 具体实现

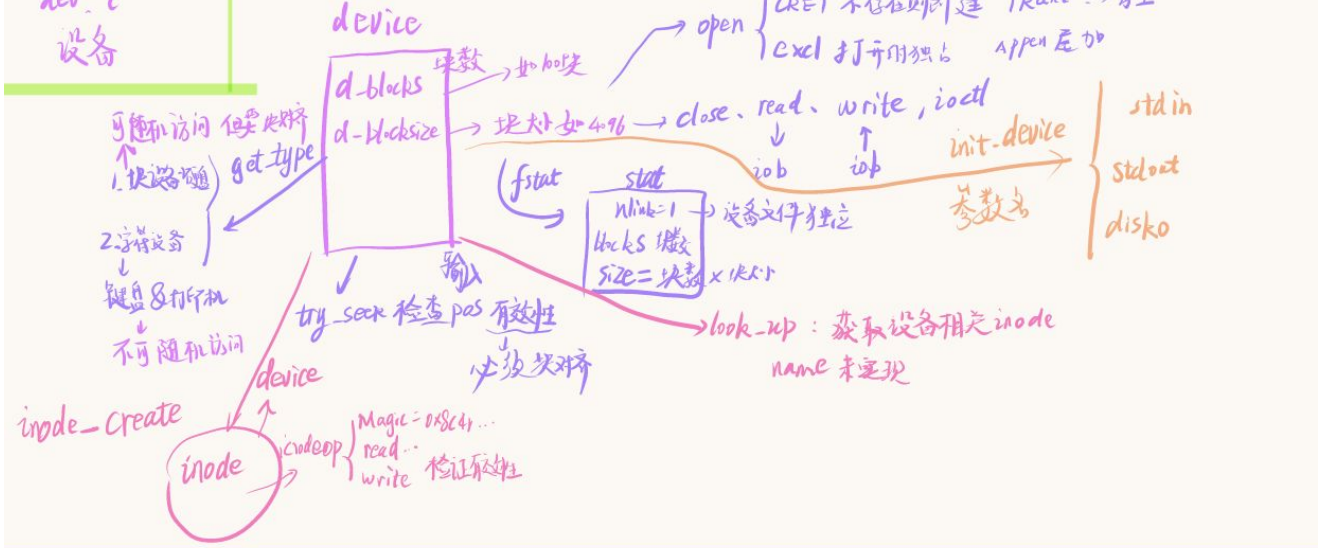



file
filestruct
fd-array
file1
file2
file3
file4
int fd
Status{NONE, INIT, OPENED, closed}
未分配 alloc之后 打开 关闭
可读 可写
int fd
open_count
off_t pos;
inode *node
打开 → 查可读or写
从array分配一项
填写file属性 返回fd
读写文件从pos开始
seek: pos
set相对开头
相对结尾
相对当前
file→pos=pos
关闭 → fd2file → close
获取状态
fd2file
fstat 存入stat
系统调用
读取: fd2file
opencount+1
读入块: 系统调用 IOB
读取了 copied
pos pos release
同步 fd2file
系统调用fsync
同步到存储设备
写入可写
file
Iob
写入
release
pos pos
copied 写入长
读目录项
文件复制
fd2file1
复制
alloc file2
iob (io_buffer)
io_base
io_offset
io_len
io_resid
move
copied len
io-base
resid-n
io-base+n
复制
data
b2m ✓
m2b 反之
置0
len
00000
resid-len
iobase
iobase+len
used
io_resid
io_len
IO_bufferSize = 4096
sysfile
文件操作系统调用
1. copy_path
用户字符串复制到内核
内核
2. open 先copy
之后用file.c进行open
3. close: file.c 进行close
4. read → 用file.c的read buffer是内核页
具体实现
5. write 同
6. seek 同
7. fstat 获取stat 同
8. fsync 写盘 同
9. chdir 改目录 → vfs.chdir
10. link 创建连接 → vfs_link
inode
文件别1
文件名2
文件名3
11. rename 重命名文件 vfs_rename
12. unlink → vfs_unlink
删除文件
13. getcwd → 当前目录 vfs...
14. get_direntry 获取文件条目 → file.c

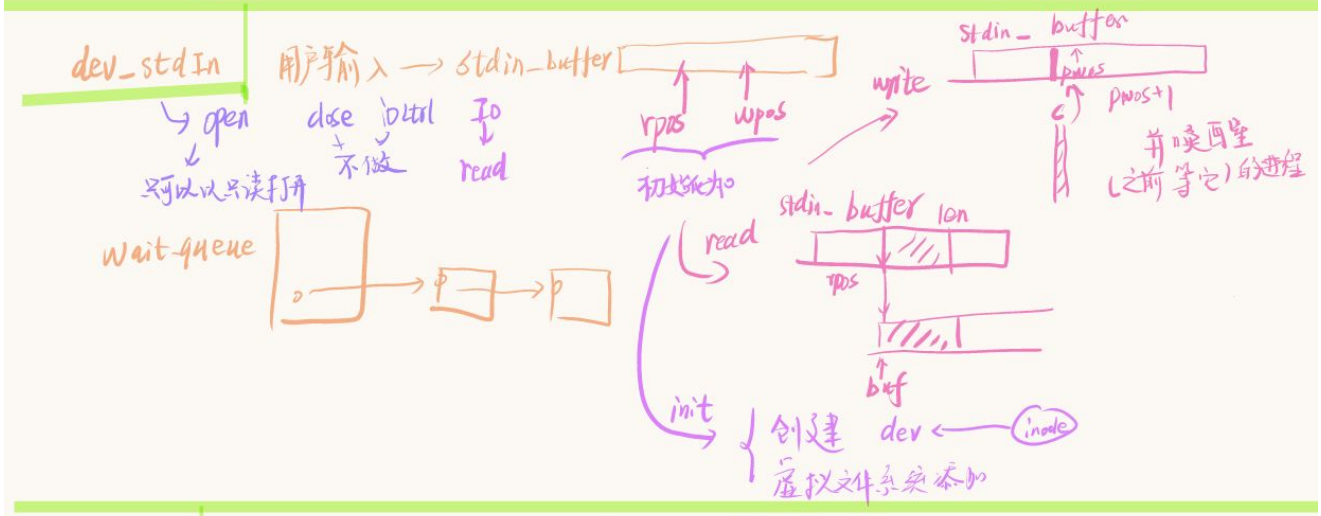
索引节点
inode



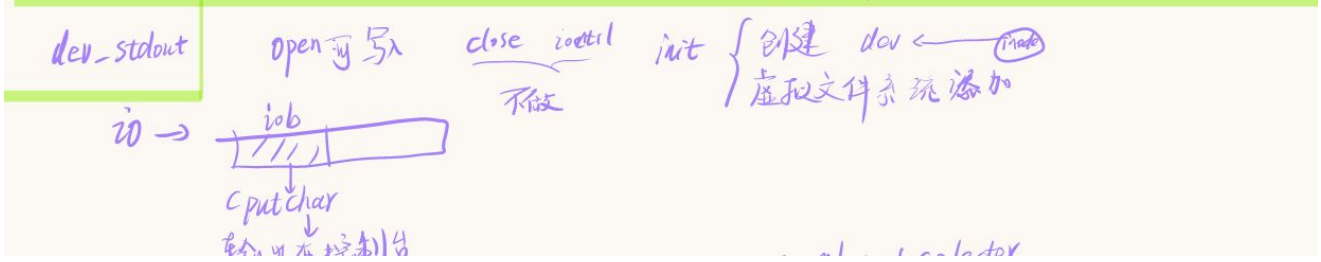
dev.c
设备

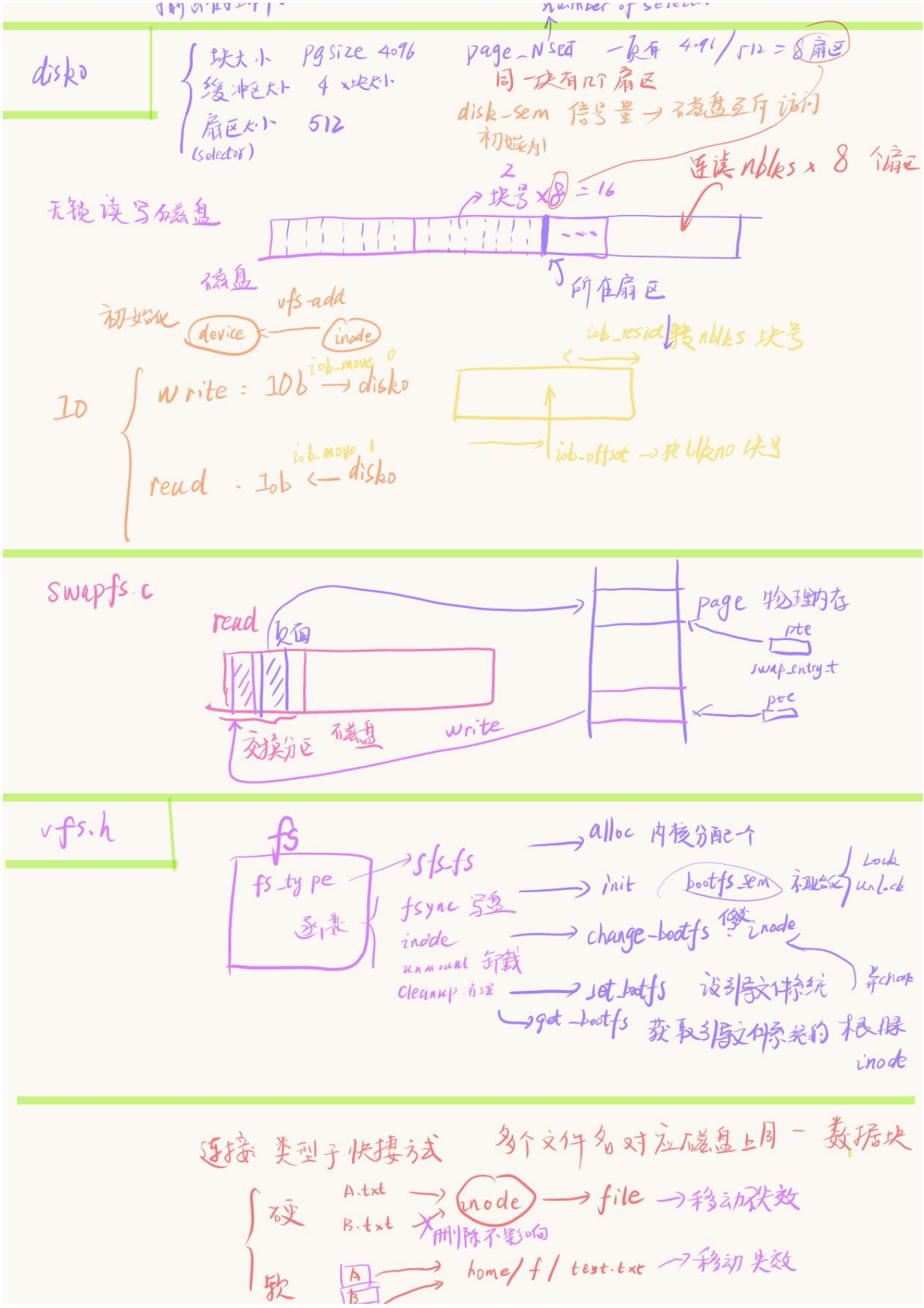


dev_stdin



dev_stdout





11.1.2.1.1.1 练习 0：填写已有实验

本实验依赖实验 1/2/3/4/5/6/7。请把你做的实验 1/2/3/4/5/6/7 的代码填入本实验中代码中有“LAB1”/“LAB2”

/“LAB3”/“LAB4”/“LAB5”/“LAB6”/“LAB7”的注释相应部分。并确保编译通过。注意：为了能够正确执行 lab8 的测试应用程序，可能需对已完成的实验 1/2/3/4/5/6/7 的代码进行进一步改进。

11.1.2.1.1.2 练习 1: 完成读文件操作的实现（需要编码）

首先了解打开文件的处理流程，然后参考本实验后续的文件读写操作的过程分析，填写在 kern/fs/sfs/sfs_inode.c 中的 sfs_io_nolock() 函数，实现读文件中数据的代码。

```
  //LAB8:EXERCISE1 YOUR CODE HINT: call sfs_bmap_load_nolock, sfs_rbuf, sfs_rblock,etc. read different kind of blocks in file
    /*
     * (1) If offset isn't aligned with the first block, Rd/Wr some content from offset to the end of the first block
     *       NOTICE: useful function: sfs_bmap_load_nolock, sfs_buf_op
     *               Rd/Wr size = (nblks != 0) ? (SFS_BLKSIZE - blkoff) : (endpos - offset)
     * (2) Rd/Wr aligned blocks
     *       NOTICE: useful function: sfs_bmap_load_nolock, sfs_block_op
     * (3) If end position isn't aligned with the last block, Rd/Wr some content from begin to the (endpos % SFS_BLKSIZE) of the last block
     *       NOTICE: useful function: sfs_bmap_load_nolock, sfs_buf_op
    */

    /*第一部分：如果偏移量与第一个块不对齐，则从偏移量到第一个块的末尾读取/写入一些内容。
在这部分中，首先检查偏移量是否与块大小（SFS_BLKSIZE）对齐。如果不对齐，那么需要执行以下操作：
使用 sfs_bmap_load_nolock 函数获取偏移量对应的块号（块索引）。
计算需要读取/写入的大小，即 (nblks != 0) ? (SFS_BLKSIZE - blkoff) : (endpos - offset)，其中 blkoff 是偏移量与块大小的余数，nblks 是剩余的块数。
使用 sfs_buf_op 函数读取/写入数据，将数据从磁盘块复制到缓冲区或从缓冲区写入到磁盘块。
更新已读取/已写入的字节数 alen。
如果没有剩余的块（nblks == 0），则跳出循环。*/
    blkoff=offset%SFS_BLKSIZE;
    if (blkoff != 0) {
        size = (nblks != 0) ? (SFS_BLKSIZE - blkoff) : (endpos - offset);
        ret = sfs_bmap_load_nolock(sfs, sin, blkno, &ino);
        if (ret != 0) {
            goto out;
        }
        ret = sfs_buf_op(sfs, buf, size, ino, blkoff);
        if (ret != 0) {
            goto out;
        }
        alen += size;
        buf += size;
        if(nblks == 0)goto out;
        blkno++;
        nblks--;
    }
/*
第二部分：读取/写入对齐的块。
在这部分中，处理对齐的块，即从块边界开始的整块数据。
使用 sfs_bmap_load_nolock 函数获取块号。
使用 sfs_block_op 函数来读取/写入整个块的数据。
更新已读取/已写入的字节数 alen，以及缓冲区指针 buf。
更新块号和剩余块数 blkno 和 nblks。
*/
```

```
    if (nblks>0) {
        ret = sfs_bmap_load_nolock(sfs, sin, blkno, &ino);
        if (ret < 0) {
            goto out;
        }
        ret = sfs_block_op(sfs, buf, blkno, nblks);
        if (ret < 0) {
            goto out;
        }
        alen += nblks * SFS_BLKSIZE;
        buf += nblks * SFS_BLKSIZE;
        blkno += nblks;
        nblks = 0;
    }
    size = endpos % SFS_BLKSIZE;

/*
第三部分：如果结束位置与最后一个块不对齐，则从开头读取/写入到最后一个块的末尾的一些内容。
这部分处理结束位置不对齐的情况。
使用 sfs_bmap_load_nolock 函数获取块号。
计算需要读取/写入的大小，即 endpos % SFS_BLKSIZE。
使用 sfs_buf_op 函数读取/写入数据，将数据从磁盘块复制到缓冲区或从缓冲区写入到磁盘块。
更新已读取/已写入的字节数 alen。

*/

    if (endpos % SFS_BLKSIZE!=0) {
        ret = sfs_bmap_load_nolock(sfs, sin, blkno, &ino);
        if (ret != 0) {
            goto out;
        }
        ret = sfs_buf_op(sfs, buf, size, ino, 0);
        if (ret != 0) {
            goto out;
        }
        alen += size;
    }
```

11.1.2.1.1.3 练习 2: 完成基于文件系统的执行程序机制的实现（需要编码）

改写 proc.c 中的 load_icode 函数和其他相关函数，实现基于文件系统的执行程序机制。执行：make qemu。如果能看看到 sh 用户程序的执行界面，则基本成功了。如果在 sh 用户界面上可以执行”ls”,”hello”等其他放置在 sfs 文件系统中的其他执行程序，则可以认为本实验基本成功。

```
    uint32_t argv_size=0, i;
    for (i = 0; i < argc; i ++) {
        argv_size += strnlen(kargv[i],EXEC_MAX_ARG_LEN + 1)+1;
    }

    uintptr_t stacktop = USTACKTOP - (argv_size/sizeof(long)+1)*sizeof(long);
    char** uargv=(char **)(stacktop  - argc * sizeof(char *));

    argv_size = 0;
```

```
    for (i = 0; i < argc; i ++) {
        uargv[i] = strcpy((char *)(stacktop + argv_size ), kargv[i]);
        argv_size +=  strnlen(kargv[i],EXEC_MAX_ARG_LEN + 1)+1;
    }
    //(7) setup trapframe for user environment
    stacktop = (uintptr_t)uargv - sizeof(int);
    *(int *)stacktop = argc;


/*首先，计算所有命令行参数（kargv）的总大小，包括参数之间的空间，存储在argv_size中。
接着，计算用户栈的顶部地址（stacktop），以便在栈上分配参数和参数指针。
然后，为用户栈上的参数指针（uargv）分配了空间，该空间可以容纳所有的参数指针，并将uargv指向该空间。
接下来，代码使用一个循环将命令行参数复制到用户栈中，并更新argv_size来跟踪已复制的字节数。
最后，代码设置用户栈的顶部地址为整数argc的地址，并将argc的值写入栈顶。这是为了在用户程序中访问参数数量。
*/
```

11.1.2.1.1.4 扩展练习 Challenge1：完成基于“UNIX 的 PIPE 机制”的设计方案

如果要在 ucore 里加入 UNIX 的管道（Pipe) 机制，至少需要定义哪些数据结构和接口？（接口给出语义即可，不必具体实现。数据结构的设计应当给出一个 (或多个）具体的 C 语言 struct 定义。在网络上查找相关的 Linux 资料和实现，请在实验报告中给出设计实现”UNIX 的 PIPE 机制“的概要设方案，你的设计应当体现出对可能出现的同步互斥问题的处理。

数据结构：

1. **管道结构**：表示管道的基本信息，包括读端和写端文件描述符、读写指针等。

```
struct pipe {
    struct file *read_end;
    struct file *write_end;
    char data[PIPE_BUF_SIZE]; // 管道数据缓冲区
    int read_ptr;
    int write_ptr;
};
```

接口：

1. **创建管道**：创建一个新的管道，返回读写端的文件描述符。

```
int pipe(int fd[2]);
```

2. **读取管道数据**：从管道中读取数据到指定缓冲区。

```
ssize_t read(int fd, void *buf, size_t count);
```

3. **写入管道数据**：将数据写入管道。

```
ssize_t write(int fd, const void *buf, size_t count);
```

4. **关闭管道端**：关闭管道的读端或写端。

```
int close(int fd);
```

5. **销毁管道**：释放管道的资源，包括文件描述符和数据缓冲区。

```
int destroy_pipe(struct pipe *p);
```

1. **互斥锁**：对于管道的数据缓冲区，可以使用互斥锁来确保同时只有一个进程或线程可以访问。在读取或写入数据之前，进程需要获取锁，操作完成后释放锁，以防止多个进程同时修改数据。
2. **条件变量**：可以使用条件变量来实现等待和通知机制。例如，在管道的读取端或写入端没有数据可用时，进程可以等待条件变量的通知，当数据可用时，其他进程可以通知等待的进程。
3. **原子操作**：某些操作可能需要原子性，例如，更新读写指针。可以使用原子操作来确保这些操作的完整性，以避免竞态条件。
4. **缓冲区管理**：管道的数据缓冲区需要进行合理的管理，包括数据的读写指针、缓冲区大小等。确保数据的读写不会越界或覆盖其他数据是很重要的。

11.1.2.1.1.5 扩展练习 Challenge2：完成基于UNIX 的软连接和硬连接机制的设计方案

如果要在 ucore 里加入 UNIX 的软连接和硬连接机制，至少需要定义哪些数据结构和接口？（接口给出语义即

可，不必具体实现。数据结构的设计应当给出一个 (或多个）具体的 C 语言 struct 定义。在网络上查找相关

的 Linux 资料和实现，请在实验报告中给出设计实现”UNIX 的软连接和硬连接机制“的概要设方案，你的

设计应当体现出对可能出现的同步互斥问题的处理。）

要实现UNIX的软连接和硬连接机制，需要定义以下数据结构和接口：

数据结构：

1. **inode结构**：表示文件或目录的元数据信息，包括文件类型、大小、创建时间等。可以根据需要扩展字段来支持连接机制。

```
struct inode {
    // 元数据信息
    uint32_t size;
    uint16_t type;
    // ...其他字段...
    // 连接信息
    uint16_t link_count; // 硬连接数
    // ...其他连接相关字段...
};
```

2. **目录项结构**：表示目录中的文件或子目录，包括文件名和对应的inode号。

```
struct dirent {
    char name[DIRSIZ];
    uint32_t ino;
};
```

3. **文件表**：记录打开的文件信息，包括文件描述符、打开模式、当前读写位置等。

```
struct file {
    int fd;
    int mode;
    off_t offset;
    struct inode *ip;
};
```

接口（函数原型，不需要实现）：

1. **创建软连接**：创建一个软连接文件，将目标文件路径保存在文件的数据部分中。

```
int symlink(const char *target, const char *linkpath);
```

2. **创建硬连接**：创建一个硬连接，即多个文件名指向同一个inode。

```
int link(const char *oldpath, const char *newpath);
```

3. **删除连接**：删除软连接或硬连接，需要根据连接类型进行不同的操作。

```
int unlink(const char *pathname);
```

4. **读取连接**：读取软连接的目标路径或硬连接所指向的inode。

```
ssize_t readlink(const char *pathname, char *buf, size_t bufsize);
```

5. **统计连接数**：获取一个inode的硬连接数。

```
int link_count(const char *pathname);
```

6. **更新连接信息**：更新连接的元数据信息，如inode中的链接计数。

```
int update_link_info(const char *pathname);
```

1. **互斥锁**：使用互斥锁来保护关键数据结构的访问，如文件表、inode表、目录项等。在多个进程或线程之间对这些数据结构的访问需要加锁，以确保同时只有一个进程能够修改它们。
2. **引用计数**：对于硬连接的处理，可以使用引用计数来跟踪每个inode的链接数。每当创建一个硬连接时，链接数增加，删除连接时链接数减少。只有当链接数为零时，才可以删除inode和相关数据块。
3. **事务操作**：对于某些操作，如创建硬连接或删除连接，可以将它们封装成事务操作。这意味着要么执行完整的操作，要么不执行，以防止部分操作导致文件系统状态不一致。
4. **原子操作**：使用原子操作来确保一些关键操作的完整性。例如，在增加或减少链接计数时，可以使用原子操作来避免竞态条件。