OS拾贝-6 结合代码来理解：文件系统(file system, fs)初始化

一些fs.h中的宏的解读

```
/* IDE disk number to look on for our file system */
#define DISKNO      1

#define BY2SECT     512 /* Bytes per disk sector */
// 一个磁盘块拥有的扇区数目
#define SECT2BLK    (BY2BLK/BY2SECT)    /* sectors to a block */

/* Disk block n, when in memory, is mapped into the file system
 * server's address space at DISKMAP+(n*BY2BLK). */
#define DISKMAP     0x10000000  // 磁盘映射到内存中的基地址

/* Maximum disk size we can handle (1GB) */
#define DISKMAX     0x40000000  // 磁盘的最大尺寸：1G
```

重要的全局变量

```
// 相当于超级磁盘块的指针
struct Super *super;
// 相当于位图磁盘块的指针
u_int *bitmap;      // bitmap blocks mapped in memory
```

fs_init()

从这个函数我们就可以看出，文件系统的初始化实际上是分为三步：

1. 读超级块（read super block），调用 `read_super()`
2. 检查整个磁盘是否可以工作（check if the disk can work），调用 `check_write_block()`
3. 将记录整个磁盘的磁盘块是否空闲的位图磁盘块由磁盘读入内存（read bitmap blocks from disk to memory），调用 `read_bitmap()`

```
// Overview:
//  Initialize the file system.
void fs_init(void)
{
    read_super();
    check_write_block();
    read_bitmap();
}
```

在解读这三个函数之前，我们需要先理解一些通用且重要的工具函数。

文件系统中重要的三个工具函数：diskaddr()、read_block()、write_block()

diskaddr()

fs/fs.c 中的 diskaddr 函数用来计算指定磁盘块对应的虚存地址。

根据一个块的序号bno (block number)，计算这一磁盘块对应的 512 bytes 虚存的起始地址。

核心公式：$\mathrm{DISKMAP} + \mathrm{blockno} * \mathrm{BY2BLK};$

```
// Overview:
//  Return the virtual address of this disk block. If the `blockno` is greater
//  than disk's nblocks, panic.
u_int diskaddr(u_int blockno)
{
    if (super && blockno >= super->s_nblocks) {// 这里的super->s_nblocks就是NBLOCK
= 1024
        user_panic("reading non-existent block %08x\n", blockno);
    }
    return DISKMAP + blockno * BY2BLK;
}
```

read_block()

将指定编号的磁盘块读入到内存中。

将传入的参数 blockno 代表的磁盘块读入到内存中。首先检查这块磁盘块是否已经在内存中，如果不在，先分配一页物理内存，然后调用 ide_read 函数来读取磁盘上的数据到对应的虚存地址处。

- 如果说参数blk != 0，就设置参数 *blk 为 blockno 代表的磁盘块在**虚拟内存中的地址**。
- 如果说参数isnew != 0
    - 并且 blockno 代表的磁盘块早就已经读入到了物理内存和进程虚拟空间中，就设置isnew = 0；
    - 并且 blockno 代表的磁盘块还没有读入到物理内存和进程虚拟空间中，就设置isnew = 1。然后为当前进程的进程空间下的该磁盘块对应的虚拟地址 `va = diskaddr(blockno)` 申请一页实际的物理内存用于保存磁盘块上的数据，再将该磁盘块上的数据加载到物理内存和进程虚拟空间；

根据该函数的oveview，这个函数一个重要的作用是确保某个特定的磁盘块已经被加载到了物理内存和用户进程空间。

```
// Overview:
//  Make sure a particular disk block is loaded into memory.
//
// Post-Condition:
//  Return 0 on success, or a negative error code on error.
//
//  If blk!=0, set *blk to the address of the block in memory.
//
//  If isnew!=0, set *isnew to 0 if the block was already in memory, or
//  to 1 if the block was loaded off disk to satisfy this request. (Isnew
//  lets callers like file_get_block clear any memory-only fields
//  from the disk blocks when they come in off disk.)
//
// Hint:
//  use diskaddr, block_is_mapped, syscall_mem_alloc, and ide_read.
int read_block(u_int blockno, void **blk, u_int *isnew)
{
    u_int va;
    // Step 1: validate blockno. Make file the block to read is within the disk.
    /* 检验 blockno 的有效性，如果super块指针不为空，且blockno超过了总共磁盘块数目，那么就
说明读到了不存在的磁盘块，报错 */
    if (super && blockno >= super->s_nblocks) {
        user_panic("reading non-existent block %08x\n", blockno);
    }
    // Step 2: validate this block is used, not free.
    /* 检验 blockno 代表的磁盘块是否是空闲的，如果是，则报错（因为我们要读出一个已经使用了的
磁盘块）
        ! 这里值得注意的是:
```

```
        每当我们使用 block_is_free(blockno) 函数去检查 blockno 代表的磁盘块是否是空闲磁盘块的时候，我们必须确保全局变量 bitmap!=NULL，即我们已经把 位图磁盘块 从磁盘中读入了内存。
        bitmap就是指向 位图磁盘块的指针
    */
    if (bitmap && block_is_free(blockno)) {
        user_panic("reading free block %08x\n", blockno);
    }
    // Step 3: transform block number to corresponding virtual address.
    // 根据 blockno 计算出 该磁盘块应该被加载到的 512B内存虚拟空间的 基地址
    va = diskaddr(blockno);
    // Step 4: read disk and set *isnew.
    // 如果 blockno 对应的磁盘块已经映射到了内存（即加载到了内存中）
    // 那么只在 isnew 不为0时，设置 *isnew 为0（说明不是新的磁盘块）
    // 否则就在 isnew 不为0时，设置 *isnew 为1（说明是新的磁盘块）
    // 并且为新的磁盘块申请一页物理内存（PTE_V | PTE_R），并将磁盘块读到 内存中 上一步计算出的va处
    // 我们只有一个磁盘，所以 使用 ide_read 函数时，磁盘号应该是 0
    // Hint: if this block is already mapped, just set *isnew, else alloc memory and
    // read data from IDE disk (use `syscall_mem_alloc` and `ide_read`).
    // We have only one IDE disk, so the diskno of ide_read should be 0.
    if (block_is_mapped(blockno))
    { //the block is in memory
        if (isnew)
            *isnew = 0;
    }
    else
    { //the block is not in memory
        if (isnew)
            *isnew = 1;
        syscall_mem_alloc(0, va, PTE_V | PTE_R);
        /* 为当前进程的进程空间下的该磁盘块对应的虚拟地址 va = diskaddr(blockno)
           申请一页实际的物理内存用于保存磁盘块上的数据，
           并将这页物理内存的物理页号+权限（PTE_V | PTE_R）存放在
           当前进程页目录映射的二级页表体系中对应的二级页表项中 */
        ide_read(0, blockno * SECT2BLK, (void *)va, SECT2BLK);
        /* 从0号磁盘的第 blockno * SECT2BLK 个扇区开始，将 SECT2BLK 个扇区（也就是一整个磁盘块）
           的磁盘内容读入到 虚拟内存空间 的 va 处*/
    }
    // Step 5: if blk != NULL, set `va` to *blk.
    if (blk)
        *blk = (void *)va;
    return 0;
}
```

write_block()

将序号为 blockno 的磁盘块 disk[blockno] 对应的**虚存空间内中整个磁盘块的数据** 写入到 **磁盘块** 中。

```
void write_block(u_int blockno)
{
    u_int va;
    // Step 1: detect is this block is mapped, if not, can't write it's data to disk.
    // 检查序号为 blockno 的磁盘块是否已经加载到了物理内存
    if (!block_is_mapped(blockno)) {
        user_panic("write unmapped block %08x", blockno);
    }
    // Step2: write data to IDE disk. (using ide_write, and the diskno is 0)
    va = diskaddr(blockno);
    // 选择0号磁盘，将 va 代表的地址开始 SECT2BLK 个扇区（也就是一整个磁盘块）的数据，写入到
```

```
    // 0号磁盘的 从第 blocknoo * SECT2BLK个扇区开始 的 SECT2BLK(8) 个扇区中
    ide_write(0, blockno * SECT2BLK, (void *)va, SECT2BLK);
    // 这里由于同一个进程的同一个地址，所以这里的syscall_mem_map调用的page_insert其实只是
    // 更新tlb 和 更新pp对应物理页面基地址的权限（因为在lab4里面会给父进程的页表项新增
PTE_COW用于保护，lab5里面会新增PTE_LIBRARY）
    syscall_mem_map(0, va, 0, va, (PTE_V | PTE_R | PTE_LIBRARY));
}
```

那么，下面我们来看一下文件系统初始化过程中调用的三个函数

read_super() 将磁盘中的超级块读入内存并检查

```
// Overview:
//  Read and validate the file system super-block.
//
// Post-condition:
//  If error occurred during read super block or validate failed, panic.
void read_super(void)
{
    int r;
    void *blk;
    // Step 1: read super block.
    // 读取超级磁盘块（即1号磁盘块）到内存中，并将加载到的内存地址存入blk，再存入super
    if ((r = read_block(1, &blk, 0)) < 0) {
        user_panic("cannot read superblock: %e", r);
    }
    super = blk;

    // Step 2: Check fs magic nunber.    核对 magic nunber
    if (super->s_magic != FS_MAGIC) {
        user_panic("bad file system magic number %x %x", super->s_magic,
FS_MAGIC);
    }

    // Step 3: validate disk size.
    // 检查当前文件系统占用的磁盘块数目是否超过了磁盘的最多磁盘块 1G / 4K = 1024*1024/4 =
256 K
    if (super->s_nblocks > DISKMAX / BY2BLK) {
        user_panic("file system is too large");
    }

    writef("superblock is good\n");
}
```

check_write_block() 检查write_block()函数能否正常工作，实际上读入了启动磁盘块

```
// Overview:
//  Test that write_block works, by smashing the superblock and reading it back.
void check_write_block(void)
{
    super = 0;

    // backup the super block.
    // copy the data in super block to the first block on the disk.
    read_block(0, 0, 0); // 将启动磁盘块（即0号磁盘块）读入内存
    // 用 虚拟内存中 1 号磁盘块中的内容 覆盖 刚才读入的 虚拟内存中 0 号磁盘块中的内容
    // 即用 0 号磁盘块对应的虚拟空间 去 备份一份 1 号磁盘块的数据，这里调用一次read_block是
以防
    // 0号磁盘块还没有申请对应的物理空间和虚拟空间。
    user_bcopy((char *)diskaddr(1), (char *)diskaddr(0), BY2PG);
```

```
    // smash it
    // 这里也是利用s_magic成员是struct Super结构体中的第一个成员，所以可以用基地址直接找到并覆盖
    strcpy((char *)diskaddr(1), "OOPS!\n");
    // 将内存中 1 号磁盘块 对应的虚拟空间的数据 写入磁盘的 1 号磁盘块区域
    write_block(1);
    user_assert(block_is_mapped(1));

    // clear it out
    // 解除原先建立的 1 号磁盘块的虚拟地址和物理内存的映射关系
    // 将进程页目录的二级页表体系中 va=diskaddr(1) 对应的二级页表项清零，顺便减少物理页面引用次数，看情况释放物理页面。做完这些后更新TLB。
    syscall_mem_unmap(0, diskaddr(1));
    user_assert(!block_is_mapped(1));

    // validate the data read from the disk.
    read_block(1, 0, 0); // 将超级磁盘块（即1号磁盘块）读入内存 （实际上是再次读入）
    // 比较刚才 先是在 1号磁盘对应虚拟内存 中写入的“OOPS”
    // 是否真的被写入了实际磁盘1号磁盘块区域
    user_assert(strcmp((char *)diskaddr(1), "OOPS!\n") == 0);

    // restore the super block.
    // 用 0 号磁盘块对应的虚拟空间 备份的 1 号磁盘块的数据还原磁盘 1 号磁盘块的数据
    user_bcopy((char *)diskaddr(0), (char *)diskaddr(1), BY2PG);
    write_block(1);
    // IMPORTANT 在这里，全局变量 super 正式被赋值为 虚拟内存中超级磁盘块（1号）的指针
    super = (struct Super *)diskaddr(1);
}
```

read_bitmap() 将所有位图磁盘块读入内存

```
// Overview:
//  Read and validate the file system bitmap.
//
// Hint:
//  Read all the bitmap blocks into memory.
//  Set the "bitmap" pointer to point ablocknot the beginning of the first bitmap block.
//  For each block i, user_assert(!block_is_free(i))).Check that they're all marked as inuse
void read_bitmap(void)
{
    u_int i;
    void *blk = NULL;

    // Step 1: calculate this number of bitmap blocks, and read all bitmap blocks to memory.
    // 计算出当前文件系统占用的磁盘块个数 需要多少个 位图磁盘块
    nbitmap = super->s_nblocks / BIT2BLK + 1;
    // 将这些位图磁盘块 读入内存
    for (i = 0; i < nbitmap; i++) {
        read_block(i + 2, blk, 0);
    }
    // IMPORTANT 在这里，全局变量 bitmap 正式被赋值为 虚拟内存中位图磁盘块（2号）的指针
    bitmap = (u_int *)diskaddr(2);

    // Step 2: Make sure the reserved and root blocks are marked in-use.
    // Hint: use `block_is_free`
    // 利用读入内存的 位图磁盘块 检查 0号 1号 磁盘块是否是已经使用的状态
    user_assert(!block_is_free(0));
    user_assert(!block_is_free(1));

    // Step 3: Make sure all bitmap blocks are marked in-use.
    // 利用读入内存的 位图磁盘块 检查 所有的位图磁盘块是否是已经使用的状态
```

```
    for (i = 0; i < nbitmap; i++) {
        user_assert(!block_is_free(i + 2));
    }

    writef("read_bitmap is good\n");
}
```