Homework #5

周雨扬
2000013061

1 Challanges

本次作业完成了所有的代码补全任务，做了如下 challange.

- The block cache has no eviction policy. Once a block gets faulted in to it, it never gets removed and will remain in memory forevermore. Add eviction to the buffer cache. Using the `PTE_A` "accessed" bits in the page tables, which the hardware sets on any access to a page, you can track approximate usage of disk blocks without the need to modify every place in the code that accesses the disk map region. Be careful with dirty blocks.

2 Exercise 1

env_create()

根据代码中给出的描述简单模拟即可。只需要找到 IO 权限对应的位即可。
最后在 `inc/mmu.h` 中找到了用于 IO 权限的位 `FL_IOPL_MASK`。之后直接模拟即可，没有细节。

Question 1

实际上并不需要。切换的时候 `env_pop_tf` 实质上已经保存了包括 IO 权限信息在内的数据。

3 Exercise 2

bc_pgfault()

首先我们需要在内存上声明一个页，用于存储从磁盘中提取上来的数据。该页的权限需要同时设置为 `P,U,W` 使得用户有权限其上的内容。注意声明页的视乎首先需要将地址按照页取整。

之后根据提示观察 `fs/ide.c`, 发现其中有函数 `ide_read` 可以支持从磁盘上读入信息。由于我们读入的信息总是一页/一块，而磁盘管理单元是段，将其转换成段数，段编号后调用即可。

flush_block()

首先判断其是否有更新到磁盘的必要。如果当前地址所在的页没有被映射，亦或者没有被修改 (`PTE_D` 为假)，则没有必要进行修改。

之后我们需要将内存上的信息写入磁盘。类似于 `bc_pgfault()`, 我们发现 `fs/ide.c` 中有函数 `ide_write` 可以支持向磁盘上写入信息。调用参数设置也非常类似，略去。

最后我们将当前页上的 `PTE_D` 位抹除即可。将权限位和 `PTE_SYSCALL` 取交即可。

4 Exercise 3

alloc_block()

我们观察 free_block。观察发现在释放块的时候，他将 bitmap 的某一位赋值成 0。据此我们可以推断：bitmap 是一个用类似于 bitset 的结构存储块是否被使用的数组，没有被使用当且仅当这一位值为 1. 这一推论也可以从 block_is_free 证实。

据此我们可以直接枚举所有块，如果其没有被使用的将其赋值为使用。最后我们需要强制将 bitmap 更新到磁盘中，否则会引起信息不同步，重新加载时候会出错。

5 Exercise 4

file_block_walk()

观察 inc/fs.h 中的代码，发现至多有 NDIRECT 个直接访问的块，据此结合注释可以直接处理块编号小于 NDIRECT 或者块编号大于 NINDIRECT + NDIRECT 的情况。

否则我们需要先判断该文件的 indirect 部分有没有声明。如果没有声明且不允许声明则返回错误。否则我们可以通过 alloc_block 声明一个块存储 indirect 块的信息，并将其初始全部赋值为 0.

最后类似于直接块的寻址方式寻址即可。

file_get_block()

类似的，如果块编号大于 NINDIRECT + NDIRECT 的情况，可以直接返回越界。否则我们可以直接使用 file_block_walk() 获取存储页编号的位置。

如果这个位置已经被赋值了，则可以直接返回。否则我们需要给这位置赋值一个全新块的编号。赋值方式类似于 file_block_walk()，此处略去细节。

Note

如果 alloc_block() 了一个编号为 0 的块，当前的实现其会被当成没有连接块。但是实际上它的确被赋值了。观察后文发现系统全部把 0 当成了没有赋值的标记。

这可能带来磁盘泄露的问题。一个解决方案是强制分配的时候重编号是的没有编号为 0 的块，另一个解决方案是修改判定没有块编号存储的条件。两者要修改的地方都很多 QAQ。

6 Exercise 5

serve_read()

参考上面的 serve_set_size(), 我们可以用相同代码获取已经打开文件的信息。之后我们观察注释，调用 file_read 完成即可。注意操作的参数在 req 中，返回值存储在 ret 中。

最后记得维护 offset，其记录了已经读取的长度。同时记得将读入的长度限制在 BLKSIZE, 否则最后一个练习会发生问题。

7 Exercise 6

serve_write()

参考上面的 `serve_read()` 即可。传除去调用 `file_write()` 参数略有不同之外其余几乎全部相同。

devfile_write()

可以通过参考上面的 `devfile_read` 获得向磁盘写的操作细节。除去调用参数，调用顺序略有不同，其余基本一致。注意仍然需要保证每次写入的长度不超过 `BLKSIZE`.

之后我加上了一层循环，向文件中不断的写入信息直到写完亦或者是发生错误。这里我们需要维护已经写入的长度，指向没有写入信息的指针。维护细节不多此处略去。

8 Exercise 7

sys_env_set_trap_frame()

注释里的三个要求依次为：

- `cs` 最低两位设置为 3.
- `eflags` 的 `FL_IF` 设置为真.
- `eflags` 的 `FL_IOPL_MASK` 全部设置为 0.

简单模拟即可。之后记得加入 `syscall()` 中。

i386_init()

插入 `ENV_CREATE(user_spawnhello, ENV_TYPE_USER)` 用于代码测试。

9 Exercise 8

duppage()

如果当前页权限位包含 `PTE_SHARE`, 简单的将其映射到子环境中相同的地址即可。由于是共享，权限位设置为 `PTE_SYSCALL` 即可。

copy_shared_pages()

判定一个页是否需要拷贝与 `fork()` 中的规则类似，除了我们需要在页权限位上额外检查是否包含 `PTE_SHARE`。拷贝过程与 `duppage()` 类似，复制即可。

10 Exercise 9

trap_dispatch()

两个错误和 IRQ_OFFSET 的偏移量分别为 IRQ_KBD 和 IRQ_SERIAL，处理函数分别为 kbd_intr() 和 serial_intr(). 据此可以直接实现。处理完后由于仅是中断，返回即可。

11 Exercise 10

runcmd()

直接抄下面处理输出重定向的代码即可。注意打开文件时候权限仅有 O_RDONLY, 不需要创建文件。

12 Challange 2

alloc_block()

如果没有空余的块，我们需要去删除某些已经不再使用的块。仍然忽略所有的块，并且忽略掉没有 PTE_P 的块。

首先我们判断这个块是否为脏块。如果是的话需要先 flush_block() 刷新后，删除 PTE_D 位。(不能用 PTE_SYSCALL 消除，会将 PTE_A 一起删除！)

之后我们再判断是否有 PTE_A。如果有则说明这个块已经不再使用。此时则可以用 sys_page_unmap 和 free_block 来释放这个块。

如果上述过程中我们释放了至少一个块，则我们可以直接记录其中一个被释放的块的信息，将其返回；否则如果一个块都没有被释放，则直接返回错误。