Lab5 设计文档

516030910293 姚子航

一、The File System

传统的如 linux 操作系统都是 monolithic 的架构，但是 Lab 中所涉及到的 JOS 是一个 exokernel 的架构，在 JOS 中，文件系统是单独由一个特殊的环境 ENV_FS 所管理，而 JOS 所使用的硬盘驱动 ide_driver 也并没有在内核中实现，而是在这个特殊的文件系统环境中实现。该部分主要实现了该环境对磁盘的访问，在 JOS 中使用 PIO 与硬件进行通信，而要想能够使用 PIO 需要我们在 EFLAG 中设置相应的 IO 权限，exercise1 就是通过调整内核的代码使得内核可以判断出这个特殊的文件系统环境从而设置好特殊的权限，代码实现很简单就是在 env_alloc 中加一层判断：

```
if(type == ENV_TYPE_FS){
    new_env->env_tf.tf_eflags |= FL_IOPL_3;
}else{
    new_env->env_tf.tf_eflags |= FL_IOPL_0;
}
```

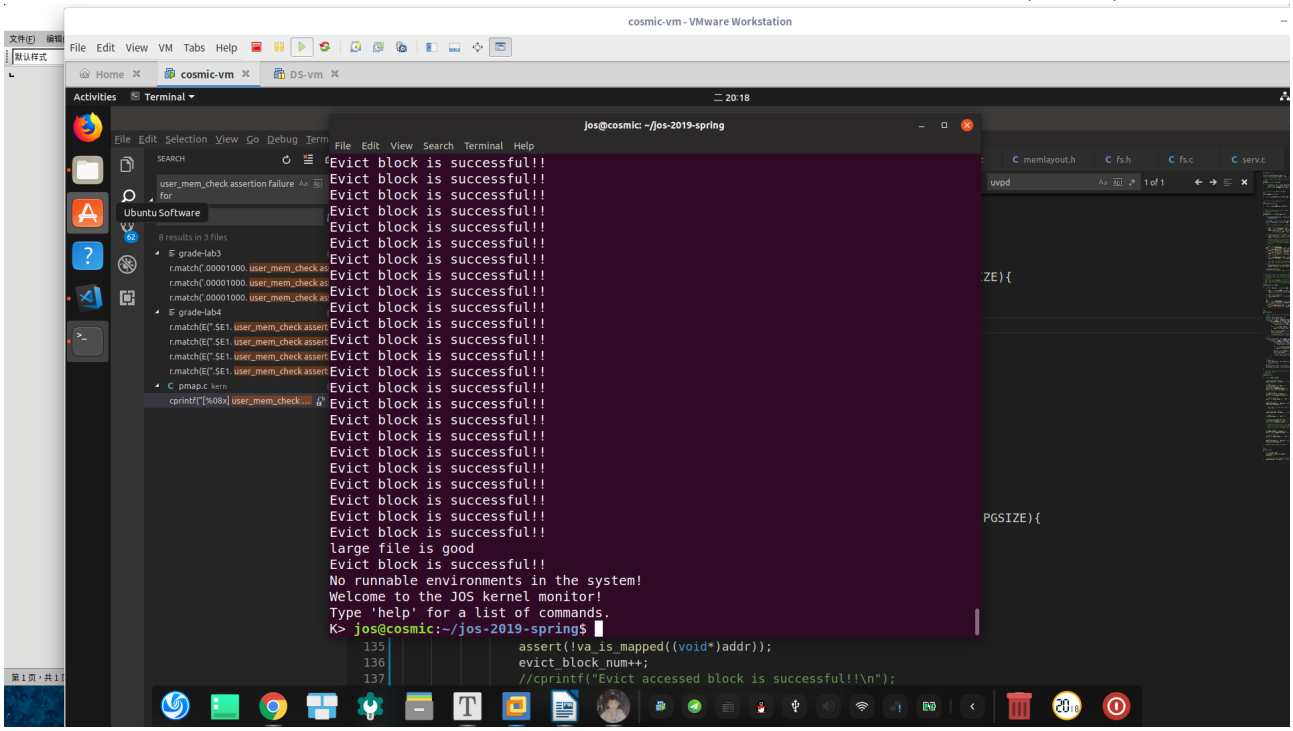
Exercise2 是要实现一层 Block cache，由于 JOS 使用一个特殊环境来维护文件系统，我们就可以充分利用每个环境都有 4GB 虚拟地址空间这一特性，简化我们的磁盘访问。在文件系统环境的内存空间中，我们将 3GB 的磁盘映射到 DISKMAP — (DISKMAP+DISKMAX)这一区域内，然后通过 on-demand allocation 的方式，当我们需要读取这个文件的时候，通过触发内存读写的 page fault，文件系统环境判断出要读取的块，并调用 ide 驱动将该块从磁盘中读取到内存中，在我们需要对文件系统进行同步的时候，就调用一个 flush 函数将该块 flush 到磁盘中。Exercise2 要实现的两个函数就是补全 pagefault handler 以及 flush_block 两个函数，page fault handler 需要补全的部分是在磁盘区域分配一个页，并将磁盘内容读取到该内存页上。只需要调用两个 syscall：

```
addr = (void*)ROUNDDOWN((uintptr_t)addr, PGSIZE);
if((r = sys_page_alloc(0, addr, PTE_P|PTE_U|PTE_W)) < 0)
    panic("in bc_pgfault, sys_page_alloc: %e", r);
if((r = ide_read(blockno * BLKSECTS, addr, BLKSECTS)) < 0)
    panic("in bc_pgfault, ide_read: %e", r);
```

flush_block 函数类似 page fault 函数，调用 ide_write 写回磁盘和 sys_page_map 系统调用清除 dirty 位就可以完成。

现有的设计有一个致命的缺陷，就是我们没有驱逐 block 的机制，这样的话随着系统的运行，我们会不断地将块从磁盘加载到内存中，内存空间会不断缩小直到 0，所以我们需要实现一个合理的驱逐块函数。这也是我完成的 challenge 内容，我的驱逐逻辑也不是很复杂：当分配一个新块时判断是否达到了 evict 阈值需要调用 evict 函数，在 evict 函数中首先遍历所有除 Block0，Block1 之外的没有被访问过的内存页，不断的 evict 这些页直到达到每次 evict 的数目，如果驱逐了所有未被访问的页还没有达到，就需要降级驱逐那些被访问过的页。由于这次的测试用例覆盖的块比较少，所以我暂时将 evict 阈值设置为 30，每次 evict 2 个 block，这样在调用 testfile 用例时就可以触发到 evict 函数。下面的截图就是调用 make run-

testfile 时的截图，可以看到 evict 被成功调用并且用例通过运行，证明实现是正确的，为了之后的运行性能，我暂时将阈值和驱逐数调整为更加合理的值：16*1024 (64MB)和 64。



evict 函数如下：

```
void
evict_block()
{
    //Every time we evict evict_batch_size blocks
    int r;
    uint32_t evict_block_num = 0;
    uint32_t evict_batch_size = 2;
    for(uintptr_t addr = DISKMAP+2*PGSIZE; addr < DISKSIZE+DISKMAP; addr += PGSIZE)
{
        if(va_is_mapped((void*)addr) && (uvpt[PGNUM(addr)] & PTE_A)==0){
            if(va_is_dirty((void*)addr)){
                flush_block((void*)addr);
            }
            if((r = sys_page_unmap(0, (void*)addr)) < 0){
                panic("sys_page_unmap: %e", r);
            }
            assert(!va_is_mapped((void*)addr));
            evict_block_num++;
            //cprintf("Evict unaccessed block is successful!!\n");
            if(evict_block_num >= evict_batch_size) break;
        }
    }
    if(evict_block_num < evict_batch_size){
        for(uintptr_t addr = DISKMAP+2*PGSIZE; addr < DISKSIZE+DISKMAP; addr +=
PGSIZE){
            if(va_is_mapped((void*)addr)){
                if(va_is_dirty((void*)addr)){
                    flush_block((void*)addr);
                }
```

```
                    if((r = sys_page_unmap(0, (void*)addr)) < 0){
                        panic("sys_page_unmap: %e", r);
                    }
                    assert(!va_is_mapped((void*)addr));
                    evict_block_num++;
                    //cprintf("Evict accessed block is successful!!\n");
                    if(evict_block_num >= evict_batch_size) break;
                }
            }
        }
        //impossible
        if(evict_block_num < evict_batch_size){
            assert(false);
        }
        cprintf("Evict block is successful!!\n");
        allocated_block_num -= evict_block_num;
}
```

Exercise3 是有关于文件系统中的 bitmap，为了维护文件系统的块分配机制，JOS 的文件系统有一个 bitmap 负责存储这个信息，这个练习我们只需要实现 alloc_block 函数，实现也很简单，在这里就不粘贴了，就是遍历 bitmap 中的每个 Bit，直到找到为 0 的 bit 就将其置为 1 并返回，这里要注意的是为了一致性我们每次修改完 bitmap 要将其 flush 会磁盘。

Exercise4 是有关文件系统中的一系列文件操作，在这里我们需要实现两个 low-level 的操作：file_block_walk 和 file_get_block，file_block_walk 类似 pg_walk，就是通过 block 索引找到文件中的一个 block，并在需要时进行分配。这里根据函数的注释提示写就可以了。

到了目前我们已经将文件系统所需的函数功能实现完全，我们需要为其编写一个对外调用的接口，由于 JOS 是 exokernel 的架构，文件系统独立运行在一个进程中，所以我们使用 IPC 进行通信，传递每个进程对于文件系统的调用，在这里我们使用了一个更高级的抽象：RPC，它建立在 IPC 的基础上，这使得文件系统与访问文件系统的进程形成了一个小型的单机内的 CS 架构，非常符合 CSE 课程中的强制模块化思想。在 Exercise5 和 Exercise6 中我们主要实现的是 filesystem server 中的 `serve_read` 和 `server_write` 接口。这两个函数的实现主要是调用一下已经写好的文件系统读写函数。

二、Spawning Processes

在这个部分我们需要实现 spawn 原语使得可以创建一个新的进程并执行所需要的程序，这个类似于 linux 中的 exec 函数，JOS 中已经写好了 spawn 函数的实现，我们只需要补全一个新添加的 syscall：`sys_env_set_trapframe`，这个函数比较简单，他的目的就是确认我们新执行的环境运行在用户态权限，开启中断，并且拥有的 `IO` 权限为 `0`（不能使用 `PIO`）.这里 `JOS` 没有实现 `unix` 风格的 `exec` 调用的原因是因为 `spawn` 函数将加载进程与被加载进程分离，使得父进程可以不借助内核的帮助完成加载过程，如果使用 `exec` 风格的函数就会因为加载过程中自身内存空间的不一致导致出错。

`Exercise8` 是要实现在 `fork` 或者 `spawn` 的过程中共享文件描述符这些状态，因为之后要实现 `pipe` 这种操作，所以我们在 `fork` 的时候应该设置为共享内存页而不是 `COW`，我们需要在两个地方做修改，一个是 `fork` 时用到的 `duppage` 函数，一个是 `spawn` 用到的 `copy_shared_page` 函数。

Exercise9 是要补全为了实现 shell 所需的代码，由于之前 kernel monitor 是关闭了中断，通过 poll 的方式来不断的获取字符，而之后我们实现的 shell 需要打开中断，所以我们需要在 trap.c 中设置好键盘中断的处理，实现很简单，只需要添加两行已经写好的函数调用即可。

Exercise10 是要实现 shell 调用所需要的输入输出重定向，我们需要做的也很简单，就是将获取到的文件名打开，并将文件描述符复制到到标准输入流就可以通过测试。代码由于很简单在这里也就不粘贴了。这样就完成了 Lab5 的所有内容。