Operating System MIT 6.828 JOS Lab5 Report

Computer Science
ChenHao(1100012776)

2013 年 12 月 19 日

目录

Lab5 主要是要实现了一个简单的，只读，基于磁盘的文件系统，文件系统实际上是一个用户级的进程，其它用户进程对磁盘文件进行访问是通过和文件系统进程进行 IPC 通信来实现的。不过这个 Lab 材料太少了，看不太懂代码呀！

Unix 的文件系统将磁盘分为两个区域：inode 和 data 区域。inode 区域保存文件的状态属性，以及所指向数据块的指针，data 区域存放文件的内容和元信息。JOS 磁盘由两个基本单位：扇区和块。磁盘中一般第一块或最后一块是 Super Block，里面存放了包含文件系统的各项元数据，例如 block 的大小，磁盘大小，磁盘布局等。

1 Exercise 1

x86 处理器通过 EFLAGS 寄存器中的 IOPL 位来决定保护模式下是否可以执行特殊的 I/O 指令，因此我们需要给 file system environment I/O 权限，如果同时有多个 environment 享有 I/O 权限，则会对于中断的分配到对应的 user-mode environment 造成很大困扰。

IOPL 有 4 种特权级，其中 0 级的特权最高，3 级最低，而此处是给予用户进程权限，因此给予 FL_IOPL_3。

```
    if (type == ENV_TYPE_FS)
        e->env_tf.tf_eflags |= FL_IOPL_3;
```

2 Question 1

e->env_tf 会在产生 trap 时，由硬件以及中断处理程序进行保存，在 env_pop_tf() 中恢复。

3 Exercise 2

Block Cache 是物理磁盘上磁盘块的缓存，用来加速对磁盘的访问。JOS 在文件系统进程中使用最多 3G(DISKSIZE) 内存在保存磁盘内容，从 DISKMAP 开始的 3G 来进行内存到磁盘空间的映射。开始并不会进行映射，只有在需要访问某个磁盘块的时候才会进行 Block Cache。因此本质上这就是一个 page fault handler，不同之处在于拷贝信息，一个是从内存中，这个是从硬盘中。

在真实的文件系统中一般在会限制 the block cache 的大小，当超过大小的时候，则会将一部分磁盘块缓存写回磁盘块中。但是 JOS 的文件系统中并没有实现回收。

```
        addr = ROUNDDOWN(addr, PGSIZE);
    r = sys_page_alloc(0, addr, PTE_W | PTE_U | PTE_P);
    if (r < 0) panic("bc_pgfault sys_page_alloc error : %e\n", r);

    r = ide_read(blockno * BLKSECTS, addr, BLKSECTS);
    if (r < 0) panic("bc_pgfault ide_read error : %e\n", r);
```

4 Exercise 3

这里 JOS 实现了一个简易的类似 exec 功能的过程——spaw。spaw 大致流程：从磁盘中打开文件，读取文件，调用 sys_exofork 创建子进程，并设置子进程的 trapframe 以及初始化其栈空间，并将 elf 文件读入子进程相应内存位置，设置 eip 和 esp，并设置为 RUNNABLE 即可。

sys_env_set_trapframe 使得进程根据 Trapframe 来使某一个进程的状态变化，例如设置进程的 eip 和 esp 位置，从而实现类似 exec 的效果。

```
static int
sys_env_set_trapframe(envid_t envid, struct Trapframe *tf)
{
    // LAB 5: Your code here.
    // Remember to check whether the user has supplied us with a good
    // address!
    struct Env * env;
    int r = envid2env(envid, &env, 1);
    if (r < 0) return -E_BAD_ENV;

    user_mem_assert (env, tf, sizeof(struct Trapframe), PTE_U);

    env->env_tf = *tf;
    env->env_tf.tf_cs |= 3;
    env->env_tf.tf_eflags |= FL_IF;

    return 0;
}
```

5 Challenge 2: Unix-style exec

这里提到了 spawn 会比 Unix-style exec 要容易实现，这是为什么呢？

注意到执行 spawn 的时候，实际上当前进程 fork 了一个子进程，然后将需要执行的 elf 文件导入至子进程。而 Unix style exec 是 replaces the current process image with a new process image. 是不通过 fork 的。这就是两者的区别，Unix-style exec 是比较困难实现的，因为很可能需要执行的 elf 文件需要存放的虚拟内存位置会和当前进程的代码和数据冲突，而导致崩溃。

那么真实的操作系统是如何做到的呢？通过动态链接，这样用户库的 exec 就不会被读入的静态数据和代码而覆盖掉。

赤手空拳难以写出动态链接，那么我们是不是可以实现一个伪动态链接，lab4 中 pgfault 里面 copy-on-write 的时候通过临时页表进行保存。因此我们可以先把 elf 文件都存放在别的页面，等陷入 kernel 里面再移送到正确的位置。于是我很邪恶地将某一块地址 0x80000000 开始的地方作为临时的缓冲区域。

大致流程就是 exec 先将 elf 文件以及新的栈内容拷贝到临时内存中作为缓冲，再引发一个 system call 来实现将临时内存的内容拷贝至真是的地址上。因为在系统调用中处于内核，因此不用担心新的代码和数据将原用户代码和数据覆盖掉。

在 spawn.c 中设置 exec 和 execl 函数，execl 函数与 spawnl 几乎完全一样，exec 如下。由

于栈的内容不能马上进行覆盖，因此新的栈的内容也需要先放置在新的临时内存中。因此需要更改一下 init_stack 函数中，即设置一下映射的地址即可。

```
// exec: Since I don't know how to built dynamic linking in JOS, so I use
    virtual address that starts from
//       0x80000000(MYTEMPLATE) to be template block cache. Then sys_exec is a
    system call to complete
//       memory setting.
// Remember: When there is virtual memory in ELF linking address overlaped with
    MYTEMPLATE, exec will fail.
int
exec(const char *prog, const char **argv)
{
    unsigned char elf_buf[512];
    uintptr_t tf_esp;

    int fd, i, r;
    struct Elf *elf;
    struct Proghdr *ph;
    int perm;


    if ((r = open(prog, O_RDONLY)) < 0)
        return r;
    fd = r;

    // Read elf header
    elf = (struct Elf*) elf_buf;
    if (readn(fd, elf_buf, sizeof(elf_buf)) != sizeof(elf_buf)
        || elf->e_magic != ELF_MAGIC) {
        close(fd);
        cprintf("elf magic %08x want %08x\n", elf->e_magic, ELF_MAGIC);
        return -E_NOT_EXEC;
    }


    // Set up program segments as defined in ELF header.
    uint32_t now_addr = MYTEMPLATE;
    ph = (struct Proghdr*) (elf_buf + elf->e_phoff);
    for (i = 0; i < elf->e_phnum; i++, ph++) {
        if (ph->p_type != ELF_PROG_LOAD)
            continue;
        perm = PTE_P | PTE_U;
        if (ph->p_flags & ELF_PROG_FLAG_WRITE)
            perm |= PTE_W;
        if ((r = map_segment(0, PGOFF(ph->p_va) + now_addr, ph->p_memsz,
                    fd, ph->p_filesz, ph->p_offset, perm)) < 0)
            goto error;
        now_addr += ROUNDUP(ph->p_memsz + PGOFF(ph->p_va), PGSIZE);
    }
    close(fd);
    fd = -1;

    // Set up Stack
    if ((r = init_stack(0, argv, &tf_esp, now_addr)) < 0)
        return r;
```

```
    // Syscall
    if (sys_exec(elf->e_entry, tf_esp, (void *)(elf_buf + elf->e_phoff), elf->
        e_phnum) < 0)
        goto error;
    return 0;

error:
    sys_env_destroy(0);
    close(fd);
    return r;
}
```

设置新的 system call，sys_exec。将临时内存中的内容移动到真实的地址中，以及构建新的栈内容。代码如下：

```
static int
sys_exec(uint32_t eip, uint32_t esp, void * v_ph, uint32_t phnum)
{
    // set new eip and esp
    memset((void *)(&curenv->env_tf.tf_regs), 0, sizeof(struct PushRegs));
    curenv->env_tf.tf_eip = eip;
    curenv->env_tf.tf_esp = esp;

    int perm, i;
    uint32_t now_addr = MYTEMPLATE;
    uint32_t va, end_addr;
    struct PageInfo * pg;

    // Elf
    struct Proghdr * ph = (struct Proghdr *) v_ph;
    for (i = 0; i < phnum; i++, ph++) {
        if (ph->p_type != ELF_PROG_LOAD)
            continue;
        perm = PTE_P | PTE_U;
        if (ph->p_flags & ELF_PROG_FLAG_WRITE)
            perm |= PTE_W;

        // Move to real virtual address
        end_addr = ROUNDUP(ph->p_va + ph->p_memsz, PGSIZE);
        for (va = ROUNDDOWN(ph->p_va, PGSIZE); va != end_addr; now_addr +=
            PGSIZE, va += PGSIZE) {
            if ((pg = page_lookup(curenv->env_pgdir, (void *)now_addr, NULL)) ==
                 NULL)
                return -E_NO_MEM;       // no page
            if (page_insert(curenv->env_pgdir, pg, (void *)va, perm) < 0)
                return -E_NO_MEM;
            page_remove(curenv->env_pgdir, (void *)now_addr);
        }
    }

    // New Stack
    if ((pg = page_lookup(curenv->env_pgdir, (void *)now_addr, NULL)) == NULL)
        return -E_NO_MEM;
    if (page_insert(curenv->env_pgdir, pg, (void *)(USTACKTOP - PGSIZE), PTE_P|
        PTE_U|PTE_W) < 0)
```

```
        return -E_NO_MEM;
    page_remove(curenv->env_pgdir, (void *)now_addr);

    env_run(curenv);        // never return
    return 0;
}
```

测试：利用 spawnhello.c 来测试，对于原始的 spawnhello，在最后增加一行 I come back!。

```
#include <inc/lib.h>

void
umain(int argc, char **argv)
{
    int r;
    cprintf("i am parent environment %08x\n", thisenv->env_id);
    if ((r = spawnl("hello", "hello", 0)) < 0)
        panic("spawn(hello) failed: %e", r);
    cprintf("I come back!\n");
}
```

则结果应该是会执行"hello.c" 并且会输出 I come back!，因为 spawn 是基于 fork 的。结果符合预期

```
Physical memory: 66556K available, base = 640K, extended = 65532K
check_page_alloc() succeeded!
check_page() succeeded!
check_kern_pgdir() succeeded!
check_page_installed_pgdir() succeeded!
SMP: CPU 0 found 1 CPU(s)
enabled interrupts: 1 2 4
FS is running
FS can do I/O
Device 1 presence: 1
superblock is good
i am parent environment 00001001
I come back!
hello, world
i am environment 00001002
No runnable environments in the system!
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K> _
```

将 spawnhello 改为 exec, 即：

```
#include <inc/lib.h>

void
umain(int argc, char **argv)
{
    int r;
    cprintf("i am parent environment %08x\n", thisenv->env_id);
    if ((r = execl("hello", "hello", 0)) < 0)
        panic("spawn(hello) exec: %e", r);
    cprintf("I come back!\n");
}
```

则结果应该会执行”hello.c” 并且不会有输出 I come back，以及 exec 的进程号与原进程号一样。结果符合预期

```
qemu -hda obj/kern/kernel.img -serial mon:stdio -gdb tcp::25000 -D qemu.log -smp
 1 -hdb obj/fs/fs.img
6828 decimal is 15254 octal!
Physical memory: 66556K available, base = 640K, extended = 65532K
check_page_alloc() succeeded!
check_page() succeeded!
check_kern_pgdir() succeeded!
check_page_installed_pgdir() succeeded!
SMP: CPU 0 found 1 CPU(s)
enabled interrupts: 1 2 4
FS is running
FS can do I/O
Device 1 presence: 1
superblock is good
i am parent environment 00001001
hello, world
i am environment 00001001
No runnable environments in the system!
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K> _
```

6 Exercise 4

还记得在以前的 fork 中，采用了 Copy-on-write 的技术，因此需要在将 PTE_W 或者 PTE_COW 的页面标记为 PTE_COW。而对于文件，由于 file descriptor 是共享的，因此在 duppage 考虑一下 PTE_SHARE 即可。

```
static int
duppage(envid_t envid, unsigned pn)
{
    // do not dup exception stack
    if (pn * PGSIZE == UXSTACKTOP - PGSIZE) return 0;

    int r;
    void * addr = (void *)(pn * PGSIZE);
    if (uvpt[pn] & PTE_SHARE) {
        r = sys_page_map(0, addr, envid, addr, uvpt[pn] & PTE_SYSCALL);
        if (r < 0) panic("duppage sys_page_map error : %e\n", r);
    } else
    if ((uvpt[pn] & PTE_W) || (uvpt[pn] & PTE_COW)) {
        // cow
        r = sys_page_map(0, addr, envid, addr, PTE_COW | PTE_P | PTE_U);
        if (r < 0) panic("duppage sys_page_map error : %e\n", r);

        r = sys_page_map(0, addr, 0, addr, PTE_COW | PTE_P | PTE_U);
        if (r < 0) panic("duppage sys_page_map error : %e\n", r);
    } else {
        // read only
        r = sys_page_map(0, addr, envid, addr, PTE_P | PTE_U);
        if (r < 0) panic("duppage sys_page_map error : %e\n", r);
    }
```

```
    return 0;
}
```

类似上一部分，对于 spawn 也需要进行这部分的映射。

```
// Copy the mappings for shared pages into the child address space.
static int
copy_shared_pages(envid_t child)
{
    // LAB 5: Your code here.
    uint32_t i;
    int r;
    for (i = 0; i != UTOP; i += PGSIZE)
    if ((uvpd[PDX(i)] & PTE_P) && (uvpt[i / PGSIZE] & PTE_P) && (uvpt[i / PGSIZE
        ] & PTE_SHARE)) {
        r = sys_page_map(0, (void *)i, child, (void *)i, uvpt[i / PGSIZE] &
            PTE_SYSCALL);
        if (r < 0) return r;
    }
    return 0;
}
```

7 Exercise 5

这部分比较简单，增加 trap 处理判断即可。

8 Question 2

2. About 10 hours.

3. 这部分的 exercise 比较少，需要看的代码比较多，对于 file I/O 有了解，但是因为经过写的训练，总觉的有点陌生。

make grade 纪念，所有 lab 做完了：

```
internal FS tests: OK (1.1s)
  fs i/o: OK
  check_super: OK
spawn via spawnhello: OK (0.9s)
PTE_SHARE [testpteshare]: OK (1.1s)
PTE_SHARE [testfdsharing]: OK (2.0s)
start the shell [icode]: OK (2.0s)
testshell: OK (1.8s)
primespipe: OK (6.4s)
Score: 75/75
lcch@lcch:~/OS/jos/lab5$ _
```

9 Summary

这个 Lab 是 JOS 中的最后一个 Lab，Exercise 比较少，最后实现起来的代码也少，基本上按照注释就可以实现了。而这个 Lab 提到的 JOS 中的文件系统，因为在做这个 Lab 的时候，对

于文件系统还不是很熟悉，而且这个 Lab 并没有很多预备的材料，因此花了许多时间在查找资料上面。在大致了解了 Unix 的文件系统的原理和结构之后，又花了非常多的时间在看 JOS 实现的文件系统上面，这一部分 Lab 中并没有很多地提到，而且代码量比较大，因此对于阅读代码对我造成了一定的困难，在这个 Lab 中我花在查资料以及阅读代码的时间远远多于写 Exercise 的时间。并且到现在我对文件系统中的代码也没有完全吃透，还是有一些只能明白个大概。

在完成了 Lab5 后，基本上就结束了本学期的操作系统实习，在不断地做 Lab 的过程中，对如何实现一个操作系统有了即宏观又细节的了解。很大地锻炼了我的代码阅读能力以及代码编写和调试 bug 的能力。通过代码阅读，我感觉到很多底层代码有一定的 trick，以及一些很巧妙的实现的地方，以及对 c 语言和汇编有了更深的理解，对我以后代码能力有了很大的提高。在系统编程中，遇到 bug 非常麻烦而且非常耗时，由于系统编程所涉及到的代码量和文件量比较大，而且许多 bug 的重现率很低，因此遇到 bug 真是一个非常头疼的事情。通过 5 个 Lab 的锻炼，我总结出遇到 bug，首先要自己对这个系统要比较清楚，这样对于 bug 才可能有一定的判断，另外在调试 bug 中我大量使用输出调试，通过输出调试来看每一阶段的状态，我感觉这个调试方法在我做 Lab 中帮助非常大。

最后，终于把 Lab 都做完了，哈哈哈，非常开心。