MIT 6.828 JOS 实验报告

计 52 于纪平 2015011265

2018 年 3 月

目录

0 总览

这个实验是我在 2017 年 2 月到 3 月完成的，采用的是 MIT 2016 秋的实验要求。我完成了全部六个实验，符合 MIT 的课程要求。

其中，前五个是比较基础的实验，从系统的启动、内核的内存管理到启动进程并管理多个进程，直到能够实现一个简单的 shell 环境。其中第 3 个与第 4 个实验内容较多，内部又细分成为了几个小部分。

最后一个实验是课程的最终项目，按 MIT 的课程要求，可以 1 至 3 人组队自由选题（推荐选题内容主要是向 JOS 加入各种新功能），也可以 1 人独立完成对网络的简单支持。这里我选择了网络的题目。

每个实验的主要内容包括：

Lab1	熟悉实验环境：Stack Backtrace
Lab2	内存管理：物理页管理、页表管理、设置地址空间
Lab3A	进程与中断：建立进程、建立页表、载入 ELF、运行进程；中断处理框架
Lab3B	中断处理与系统调用：页错误、断点、系统调用处理框架、参数合法性检查
Lab4A	多核：多核初始化、加锁、调度、用于 fork 的系统调用
Lab4B	Fork：用户态 Pagefault Handler、Copy-on-Write Fork
Lab4C	内核抢占与 IPC：时钟中断、IPC 相关系统调用
Lab5	Shell：简单文件系统、spawn（类似 fork+exec）、键盘中断、输入重定向
Lab6A	网络发送：MMIO 配置网卡，DMA 发送数据
Lab6B	网络接收：接收数据，简单 HTTP 服务器

下面依次介绍每个实验的内容。

1 Lab1 启动与熟悉环境

1.1 引导流程

这一部分，我结合代码与实际运行，观察了 JOS 在（QEMU 虚拟的）计算机上的启动流程：

- 加电后，初始化 CS:IP 为 0xf000:fff0
- 该位置是 ROM BIOS 保留区域的尾部，其中的指令是跳到 BIOS 的相关代码
- BIOS 会建立一个基本的中断表，初始化 VGA 显示等，以便输出提示信息
- BIOS 会枚举设备，并选择其中一个可启动的设备用来启动（这里是硬盘）
- BIOS 从硬盘的第一个扇区载入引导程序，将其存放在 0x7c00 地址处，并跳转到该位置，此时 JOS 的引导程序接管控制
- 引导程序会进行一些必要的设置，例如：关中断、设置数据段寄存器、开启 A20、切换到 32 位保护模式、建立引导程序的 C 栈

- 引导程序从磁盘载入操作系统映像到内存中，并跳转到入口处，引导过程完成！
- 操作系统入口会开启分页并切换到高虚拟地址、建立操作系统的 C 栈，然后跳转到操作系统初始化的 C 函数，之后绝大部分时候就可以使用 C 语言编写代码了！

1.2 熟悉内核编程

1.2.1 格式化输出

这部分的要求是在 `printf` 中加入对八进制数输出格式“`%o`”的支持。

由于它对十进制已经有支持了，直接照猫画虎就可以了……

1.2.2 打印调用栈信息

需要在内核中实现一个能够打印当前函数调用栈信息的函数 `mon_backtrace`。要求输出格式如下：

```
K> backtrace
Stack backtrace:
  ebp f010ff78  eip f01008ae  args 00000001 f010ff8c 00000000 f0110580 00000000
         kern/monitor.c:143: monitor+106
  ebp f010ffd8  eip f0100193  args 00000000 00001aac 00000660 00000000 00000000
         kern/init.c:49: i386_init+59
  ebp f010fff8  eip f010003d  args 00000000 00000000 0000ffff 10cf9a00 0000ffff
         kern/entry.S:70: <unknown>+0
K>
```

首先，对于逐层向上寻找每个栈帧，可以根据 x86 的函数调用约定根据每个栈帧的 ebp 来依次寻找。注意，最初一层的 ebp 需要用内嵌汇编的方式获取。

对于获取对应的文件名、行号、函数名等，JOS 的编译选项已经保证了映像文件中含有相关的 stabs，并已实现了对于某个 eip 寻找其文件名、行号、函数名等信息的函数，我这里的工作主要是理解他提供的相关的函数接口并调用。

1.3 总结

代码量：约 20 行。

这部分的代码量不大，主要收获是观察并了解了 x86 计算机的启动流程与体验了内核编程的方式。最大的收获可能是学到了强制内嵌汇编 `asm volatile`。这个其实在底层编程以外也可以有应用，例如对于以下性能测试程序：

```
const int n = 10000;
for(int i = 0; i < n; ++i)
    for(int j = 0; j < n; ++j)
        asm volatile("");
```

可以避免编译器将循环直接优化掉（如果不开编译优化或在其中加一句其他运算语句，性能会下降；如果不内嵌汇编，会被优化掉）。

2 Lab2 内存管理

2.1 物理页管理

实现了以下函数:

```
void *boot_alloc(uint32_t n);
//分配至少n字节的页对齐空间，仅在内存管理建立完毕之前使用
void page_init(void);
//建立初始的空闲物理页链表，清空引用计数
struct PageInfo *page_alloc(int alloc_flags);
//从空闲页链表中分配一个物理页，并在需要时清零
void page_free(struct PageInfo *pp);
//释放一个物理页，加入空闲页链表
```

2.2 页表管理

实现了以下函数（在 JOS 中，目前虚拟地址与线性地址是等价的，以下不区分):

```
//pde_t代表单个页目录项，可以用pde_t *表示一个页目录的头指针
//pte_t代表单个页表项
pte_t *pgdir_walk(pde_t *pgdir, const void *va, int create);
//在页目录pgdir中寻找虚地址va，返回对应的页表项
//如果找不到且create不为0，则分配一个物理页并增加引用计数
//这是一个辅助函数，会被下面的函数调用
void boot_map_region(pde_t *pgdir, uintptr_t va, size_t size,
   physaddr_t pa, int perm);
//将长度为size，起始虚地址va的一段映射到物理地址pa上，权限为
   perm，插入在页目录pgdir中
//此函数只用于添加内核空间的静态映射
struct PageInfo *page_lookup(pde_t *pgdir, void *va, pte_t **
   pte_store);
//在页目录pgdir中查询虚地址va，返回对应的物理页结构，并将对应
   的页表项指针存入pte_store（如果有）
void page_remove(pde_t *pgdir, void *va);
//解除页目录pgdir中虚地址va的映射，必要时释放物理页并使TLB无效
int page_insert(pde_t *pgdir, struct PageInfo *pp, void *va,
   int perm);
//在页目录pgdir中新建虚地址va到物理页pp权限为perm的映射，必要
   时解除原映射并释放物理页、无效TLB
//需要正确处理在同一个虚地址映射一个新页的情况
```

2.3 内核地址空间

实现了函数：`void mem_init(void)` 中的以下部分：

- 映射内核栈
- 额外映射一份物理页结构，使得用户程序能通过某个虚地址读取物理页信息
- 映射内核代码与之前分配过的区域

2.4 总结

代码量：约 100 行。

这部分实验的主要内容是根据 JOS 框架的要求，实现每个函数的功能。虽然都是写代码的任务，但代码量并不大，重点是需要理解 x86 的二级页表结构，并且明确每一个地址是物理地址还是虚地址。我在实验过程中有一次因为混淆了虚地址与实地址，导致出现了莫名其妙的错误，并且由于只是一处错误，并没有立即出错，导致调试起来很麻烦。

3 Lab3 用户进程

3.1 Lab3A 用户进程和异常处理

3.1.1 用户进程

首先，分配了进程结构数组 envs 来记录每个进程的信息（包括进程号、父进程号、进程类型、中断时的寄存器等），并组织成一个空闲进程链表。envs 被映射到某个虚拟地址，使得用户程序能够访问。

实现了下列函数：

```
1  void env_init(void);
2  //建立envs数组并初始化空闲进程链表
3  int env_setup_vm(struct Env *e);
4  //为一个进程建立初始页表（以内核页表为模板）
5  void region_alloc(struct Env *e, void *va, size_t len);
6  //为一个进程分配若干物理页，映射到该进程的虚地址[va, va+len)中
7  void load_icode(struct Env *e, uint8_t *binary);
8  //为一个进程载入ELF格式的映像，装入其页表，分配初始栈，设置寄
       存器
9  void env_create(uint8_t *binary, enum EnvType type);
10 //创建一个进程，载入指定的ELF映像并指定进程类型
11 void env_run(struct Env *e);
12 //启动一个进程，并转交控制权
```

3.1.2 异常处理

用汇编语言编写中断处理程序的入口，并跳转到 C 语言中断处理函数 trap_dispatch 中，在其中根据中断号转入具体每种中断的处理。这一步仅实现处理的框架，具体处理流程见后述。

最后，建立 IDT，并应用。

3.2 Lab3B 处理页错误、断点和系统调用

3.2.1 页错误

在 trap_dispatch 中，对于页错误要转到内核处理函数 page_fault_handler 中。目前的处理仅为结束出错的进程。

3.2.2 断点

在 trap_dispatch 中，对于断点要进入内核监控函数。这样在其中可以检查栈的情况，一定程度上可以用来调试（虽然没啥用……）。

3.2.3 系统调用

在 trap_dispatch 中，对于页错误要转到内核处理函数 syscall 中，其中会根据寄存器保存的系统调用号与参数，转到对应的系统调用处理函数进行处理，并将返回值保存到寄存器中。

目前仅支持输入与输出字符、获取进程号和结束进程的系统调用。

此外，在用户库主函数中通过上述系统调用来得到自己进程结构的指针，以便用户程序（通常是库）进行某些操作。

最后，实现了下列函数：

```
1 int user_mem_check(struct Env *env, const void *va, size_t len
    , int perm);
2 //查询进程env是否在其虚地址[va, va+len)有perm权限
3 void user_mem_assert(struct Env *env, const void *va, size_t
    len, int perm);
4 //确保某进程有该权限，否则杀掉该进程
```

通过这些函数，可以实现对系统调用参数的检查，如果检查不通过则可能返回调用失败或结束进程。

3.3 总结

代码量：约 150 行。

配置 IDT 的过程有大量的重复代码，这些并没有计算在这个代码量中。

在这一部分中我没有遇到太大的问题，而相对较大的问题是在配置 IDT 时可能对其中的数据段等配置有误，导致出现该中断时没有按照预期的方式进行处理。

4 Lab4 抢占式多任务

4.1 Lab4A 多处理器支持与协作式多任务

4.1.1 多处理器支持

为了使用 LAPIC，需要先配置 MMIO 的框架。首先实现了下列函数：

```
void *mmio_map_region(physaddr_t pa, size_t size);
//在MMIO的区域中添加一块足够大的映射，对应到[pa, pa+size)区域中
```

修改内存布局，使得不会在 MMIO 区域中分配物理页。

修改中断处理初始化程序，使得能处理多个 CPU 的情况，包括记录每个 CPU 的段寄存器（之前是仅有一个全局变量）。

在进入与退出内核时获取和释放内核锁，确保至多有一个处理器进入内核。

4.1.2 协作式多任务

实现一个内核函数 `sched_yield`，能在所有进程中寻找一个当前可被运行的，并运行之。具体的调度算法采用循环的方式，即从上一次运行的进程的下一个进程开始循环查找可运行的进程。

同时，添加以下系统调用：

```
//以下各系统调用，只能修改当前进程或直接孩子进程的信息
void sys_yield(void);
//让出CPU给其他进程执行
envid_t sys_exofork(void);
//创建一个空白的新进程
int sys_env_set_status(envid_t envid, int status);
//设置一个进程的状态（可运行或暂停）
int sys_page_alloc(envid_t envid, void *va, int perm);
//在一个进程中分配一个物理页并以指定权限映射到其中
int sys_page_map(envid_t srcenvid, void *srcva, envid_t
   dstenvid, void *dstva, int perm);
//将一个进程的一个映射拷贝到另一个进程中
int sys_page_unmap(envid_t envid, void *va);
//解除一个进程在一个地址的映射
```

4.2 Lab4B 写入时复制的 `fork`

4.2.1 用户态页错误处理

实现了相关的内核函数、系统调用和用户库。

内核函数：修改 page_fault_handler，使得在用户程序发生页错误，并存在页错误处理程序时，转该程序处理。

系统调用：

```
1 int sys_env_set_pgfault_upcall(envid_t envid, void *func);
2 //为一个进程设置其页错误处理程序
```

用户库函数：

```
1 void set_pgfault_handler(void (*handler)(struct UTrapframe *
    utf));
2 //设置C语言的页错误处理程序，包括调用上述系统调用，并保存该C函
    数指针
```

用户库中存在一个汇编语言完成的函数 _pgfault_upcall，用来调用上述 C 语言处理程序，然后恢复页错误出现之前的现场。

为 C 语言处理程序建立合适的栈、参数等是相对简单的问题；这里最难的一个地方是不经过内核恢复当时出现页错误的现场。

首先，恢复现场的一个通常的问题是可用的寄存器越来越少。例如：恢复某个通用寄存器后，它就不能在以后再使用了；恢复标志位后，就不能再进行任何算术运算了……并且，x86 没有提供额外的寄存器用于这种用途。

更大的问题是恢复 esp 与 eip 寄存器。由于用户态页错误处理是在一个单独的栈上进行（因为页错误可能出现嵌套的情况），在处理完毕后必须切换到出现页错误时的 esp。

我们不能直接使用 jmp 指令，因为这需要占用一个寄存器来进行；也不能直接使用 ret 指令，因为这样无法切换栈；类似 iret 的指令就更不能使用了。

实验指导中给出了一个提示：先在要切换到的栈上额外压入要切换到的 eip，然后依次恢复通用寄存器、标志位、栈指针，最后通过 ret 指令完成所有操作。我实现的具体步骤（原理）以伪代码描述如下：

1. （初始时，栈依次存储了：目的 esp、eflags、通用寄存器）
2. 改动目标栈：将目标 esp 存入 eax；eax-=4；*eax= 目标 eip
3. 调整当前栈：将改动后的 eax 存回栈中
4. 依次从栈中弹出通用寄存器（popal）、标志位、栈指针
5. 此时已经切换到目标栈上了，用 ret 回到目标 eip 即可

注意，对于页错误嵌套的情况，需要在页错误处理程序的各个栈帧之间留下一个 4 字节的空隙，以便这个压栈操作不会损坏其他数据。

4.2.2 写入时复制的 fork

实现以下用户库函数：

```
1 void pgfault(struct UTrapframe *utf);
2 //处理写入时复制的页错误，符合上一节的C语言页处理程序接口
3 int duppage(envid_t envid, unsigned pn);
4 //将当前进程的一个页映射复制到另一个进程中，如果原来具有写权
    限，则之后都标记为写入时复制的页
5 envid_t fork(void);
6 //功能类似于Linux的fork
7 //实现步骤：先安装页错误处理程序，然后创建空白新进程；父进程遍
    历自己的页表，将存在的页用duppage复制到子进程；子进程初始化
    自己的信息，并安装同样的页错误处理程序
```

至此，写入时复制的 fork 就实现完毕了。值得一提的是，这种 fork 的实现方法将大部分的逻辑都放在用户库的层次实现，仅使用系统调用提供的简单的服务。这也体现了 JOS 的微内核的思想。

4.3 Lab4C 抢占式多任务和进程间通信（IPC）

4.3.1 处理时钟中断

在之前配置与处理中断的部分中加入了对时钟中断的处理，包括在 IDT 中加入相关项目，并在内核中断处理函数 trap_dispatch 遇到时钟中断时转调度程序以便下一个进程运行。

4.3.2 进程间通信

实现了以下系统调用：

```
1 int sys_ipc_recv(void *dstva);
2 //从IPC接收一个信息，如果有页发来则将其映射到dstva上
3 int sys_ipc_try_send(envid_t envid, uint32_t value, void *
    srcva, unsigned perm);
4 //尝试发送一个IPC信息给某个进程，包括一个整数和可选的页及其权
    限
```

上述函数中用到了内核中实现的页表管理函数来辅助完成。

基于上述系统调用，实现了以下用户库函数：

```
1 int32_t ipc_recv(envid_t *from_env_store, void *pg, int *
    perm_store);
2 //从IPC接收一个信息，如果有页发来则映射到pg上，并存储发来信息
    的进程号与该页的权限
3 void ipc_send(envid_t to_env, uint32_t val, void *pg, int perm
    );
4 //发送一条IPC消息
```

至此，我们可以支持多个用户进程运行，并允许它们互相发送消息。下面是一个测试程序的例子：

```
// Ping-pong a counter between two processes.
// Only need to start one of these -- splits into two with fork.

#include <inc/lib.h>

void
umain(int argc, char **argv)
{
    envid_t who;

    if ((who = fork()) != 0) {
        // get the ball rolling
        cprintf("send␣0␣from␣%x␣to␣%x\n", sys_getenvid(), who);
        ipc_send(who, 0, 0, 0);
    }

    while (1) {
        uint32_t i = ipc_recv(&who, 0, 0);
        cprintf("%x␣got␣%d␣from␣%x\n", sys_getenvid(), i, who);
        if (i == 10)
            return;
        i++;
        ipc_send(who, i, 0, 0);
        if (i == 10)
            return;
    }

}
```

该程序可以得到这样的结果：

```
[00000000] new env 00001000
[00001000] new env 00001001
send 0 from 1000 to 1001
1001 got 0 from 1000
1000 got 1 from 1001
1001 got 2 from 1000
1000 got 3 from 1001
1001 got 4 from 1000
1000 got 5 from 1001
1001 got 6 from 1000
1000 got 7 from 1001
1001 got 8 from 1000
1000 got 9 from 1001
[00001000] exiting gracefully
[00001000] free env 00001000
1001 got 10 from 1000
[00001001] exiting gracefully
[00001001] free env 00001001
No runnable environments in the system!
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K>
```

4.4 总结

代码量：约 450 行。同上，冗余的代码并没有计算在内。

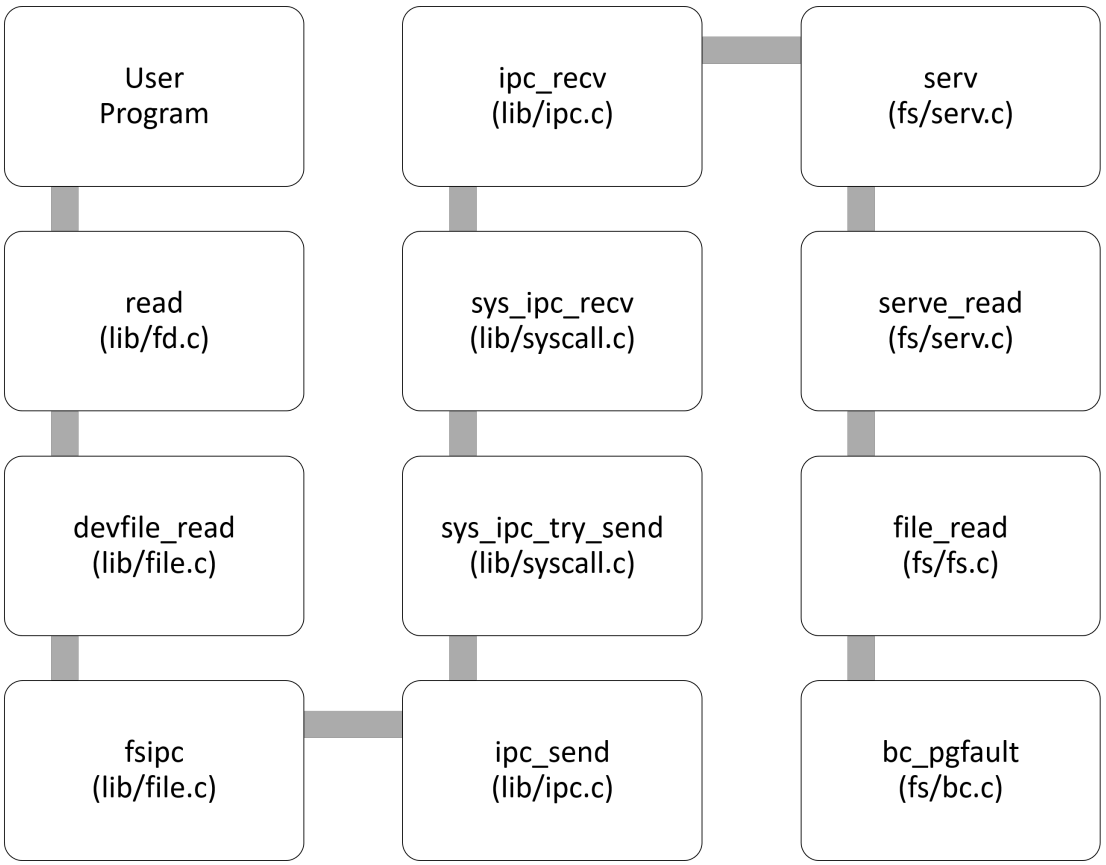
我认为这次实验是所有实验中最难的一个，不仅是前面提到的用户态页错误处理程序需要不经过内核而完成到另一个现场的切换，同时这部分实验的各个部分在代码调试方面都有很大的难度。尤其是在设置页错误处理程序时涉及到函数指针的传递，如果这里面传入了一个其他的函数，则在页出错调用时不会立即出错，而是会有一些奇怪的行为，最后出现奇怪的错误。这部分直接用 gdb 调试可能很麻烦，输出调试的结果也让我摸不着头脑。最后我选择的调试方法是：先找到最小的出问题的例子，然后使用 gdb 从引导启动一开始单步（用脚本）运行数十万步直到出现运行错误，记录每一步所有寄存器的状态，总输出大小约百兆，然后从错误处反推查找可能的异常之处，最终终于花费了两天的时间找出了这个问题！

5 Lab5 文件系统、`spawn` 和 shell

JOS 的内核实在是太小了，甚至文件系统都不归内核管。用户程序进行文件操作时，需要用 IPC 与文件服务进程交互，全部过程类似 RPC。

值得一提的是，由于采用单独的服务进程，可以将整块磁盘映射到其地址空间中，采用 PagefaultHandler 的形式仅在需要时读取信息，利用硬件提高了编码与执行效率。

下面是用户程序希望读取文件时的控制流图,我在本实验中实现的是 `devfile_read` 及其以后的部分。



5.1 文件系统

首先建立文件系统进程，分配特殊的“文件系统类型”进程权限，并加入内核初始化的末尾，使得内核初始化完毕后创建该文件系统进程。

实现块缓存，机制是将整块磁盘映射到文件系统进程的地址空间中，每次读取磁盘只需在对应地址用 memcpy 等方式复制出数据，如果数据没有加载进来的话只需通过自定义页错误处理程序的方式从磁盘读取即可。这也是允许自定义页错误处理程序的一个优美之处!!

在文件系统服务进程中实现了以下用户态函数（实验提供了封装好的通过 IDE 读写磁盘的函数，不必手动操心例如 in 和 out 指令的细节）：

```
1  void bc_pgfault(struct UTrapframe *utf);
2  //上面提到的C语言页错误处理程序，符合规定的接口
3  void flush_block(void *addr);
4  //（在必要时）将addr所在页写回磁盘
5  int alloc_block(void);
6  //分配一个空闲磁盘块
7  int file_block_walk(struct File *f, uint32_t filebno, uint32_t
       **ppdiskbno, bool alloc);
8  //寻找一个文件的某一块在磁盘上的信息（辅助函数）
9  int file_get_block(struct File *f, uint32_t filebno, char **
     blk);
10 //寻找一个文件的某一块在磁盘上的位置
11 int serve_read(envid_t envid, union Fsipc *ipc);
12 //读取文件，具体参数（偏移量、目的地址等）从ipc信息中读取
13 int serve_write(envid_t envid, struct Fsreq_write *req);
14 //写入文件，具体参数（偏移量、源地址等）从ipc信息中读取
```

实现了用户库函数：

```
1  ssize_t devfile_write(struct Fd *fd, const void *buf, size_t n
     );
2  //与Linux的write接口很相似了!
```

5.2 spawn

spawn 的功能相当于 fork 紧接着 exec。但我们这里不实现 exec，因为 spawn 更简单。

添加对“共享页”的支持，这些页面会在 fork 与 spawn 之后会被两个进程共享（而不是写入时复制）。

修改 duppage 支持共享页。实现函数 int copy_shared_pages(envid_t child)，用来在进程间复制共享页的映射。

5.3 shell

处理键盘中断，流程与处理其他中断类似，不再赘述了。

修改原型用户程序 sh.c，使其支持输入/输出重定向。实现方法为通过上述资源共享的方式建立管道，以在 `spawn` 前后获得对应到同一管道（两端）的文件描述符。

现在我们有了一个“能用”的操作系统了，可以通过 shell 进行交互，下面是一个实际的例子：

```
init: running sh
init: starting sh
$ ls -l
          37 - newmotd
          92 - motd
         447 - lorem
         132 - script
        2916 - testshell.key
         113 - testshell.sh
       26176 - init
       20148 - cat
       20004 - echo
       26176 - init
       20344 - ls
       20264 - lsfd
       20176 - num
       20244 - forktree
       20212 - primes
       20212 - primespipe
       24904 - sh
       20284 - testfdsharing
       20192 - testkbd
       20252 - testpipe
       24480 - testpteshare
       24424 - testshell
       20008 - hello
       20008 - faultio
$ cat script | num
    1 echo This is from the script.
    2 cat lorem | num | cat
    3 echo These are my file descriptors.
    4 lsfd -1
    5 echo This is the end of the script.
$ sh script
This is from the script.
    1 Lorem ipsum dolor sit amet, consectetur
    2 adipisicing elit, sed do eiusmod tempor
    3 incididunt ut labore et dolore magna
    4 aliqua. Ut enim ad minim veniam, quis
    5 nostrud exercitation ullamco laboris
    6 nisi ut aliquip ex ea commodo consequat.
    7 Duis aute irure dolor in reprehenderit
    8 in voluptate velit esse cillum dolore eu
    9 fugiat nulla pariatur. Excepteur sint
   10 occaecat cupidatat non proident, sunt in
   11 culpa qui officia deserunt mollit anim
   12 id est laborum.
These are my file descriptors.
```

```
fd 0: name script isdir 0 size 132 dev file
fd 1: name <cons> isdir 0 size 0 dev cons
This is the end of the script.
$
```

5.4 总结

代码量：约 150 行。

这部分就是轻松加愉快的节奏了，操作系统实验越做越简单，还真不是开玩笑……

6 Lab6 网络驱动

类似于文件系统，内核只提供发送和接收数据包的系统调用，其他的事情交给网络服务进程和用户库。

这里用到的“技术栈”如下图所示，我完成的内容是“收发数据包的用户库”和以下的部分。

此外，这部分实验对系统调用之类的接口完全没有规定，只要符合某个用户库函数的要求即可。这也是 6 个实验中最“自由”的一个了！

6.1 Lab6A 初始化与发送数据包

6.1.1 维护系统时间

每次时钟中断时增加当前系统时间，并增加系统调用 `sys_time_msec` 使得用户程序能获得之。

6.1.2 配置网卡

实验系统提供了 `pci_func_enable` 函数，可以通过其获取网卡的 MMIO 基地址和长度（虽然长度我们已经在用户手册知道了）。

接下来是一系列冗长乏味的配置工作，这一部分几乎就是体力活，对照 E1000 的开发手册，确定需要填进去的参数及其位置，然后用 MMIO 的方法填进去（比如配置工作模式等等）……

6.1.3 发送数据包

首先要分配一个物理页用作发送缓冲区，并通过 MMIO 方式传给网卡作为 DMA 区域。

实现了函数 `int try_transmit(physaddr_t pa, int cnt)`，用来尝试将指定的区域加入发送队列。

添加了相关的系统调用 `int sys_net_try_transmit(const char *buf, int cnt)` 和网络服务进程工作函数。

6.2 Lab6B 接收数据包与网络服务器

6.2.1 接收数据包

首先要分配一个物理页用作接收缓冲区，并通过 MMIO 方式传给网卡作为 DMA 区域。

实现了函数 `int try_receive(struct jif_pkt *jp)`，用来尝试接收一个数据包，具体参数描述在 jp 中，其中最有用的信息包括缓冲区的起始位置和长度等。

添加了相关的系统调用 `int sys_net_try_receive(struct jif_pkt *jp)` 和网络服务进程工作函数。

6.2.2 网络服务器

实验系统已经有了一个基本的 lwIP 协议栈，并在上面实现了一个基本的 HTTP 服务器框架。

我在这里实现了 `send_file` 与 `send_data` 函数，用来根据指定文件名的文件内容生成相应的 HTTP 响应，包括文件不存在等情况。

至此，我们可以在 QEMU 之外通过 HTTP 访问虚拟机中的内容了：

```
1 yjp@yjp-Virtual-Machine:~/6.828$ curl 192.168.3.115:26002/index.html
2 <html>
3 <head>
4         <title>jhttpd on JOS</title>
5 </head>
6 <body>
7         <center>
8                 <h2>This file came from JOS.</h2>
```

```
                <marquee>Cheesy web page!</marquee>
        </center>
</body>
</html>
yjp@yjp-Virtual-Machine:~/6.828$ curl 192.168.3.115:26002/lorem
Lorem ipsum dolor sit amet, consectetur
adipisicing elit, sed do eiusmod tempor
incididunt ut labore et dolore magna
aliqua. Ut enim ad minim veniam, quis
nostrud exercitation ullamco laboris
nisi ut aliquip ex ea commodo consequat.
Duis aute irure dolor in reprehenderit
in voluptate velit esse cillum dolore eu
fugiat nulla pariatur. Excepteur sint
occaecat cupidatat non proident, sunt in
culpa qui officia deserunt mollit anim
id est laborum.
yjp@yjp-Virtual-Machine:~/6.828$ curl 192.168.3.115:26002/script
echo This is from the script.
cat lorem | num | cat
echo These are my file descriptors.
lsfd -1
echo This is the end of the script.
yjp@yjp-Virtual-Machine:~/6.828$
```

6.3 总结

代码量：约 250 行。

这一部分尽管难度不大，基本属于照着 Intel 的开发手册的要求编码，但也花费了我大量时间，尤其是一开始在读开发手册上面。我对驱动程序编写有了初步的经验，而其他的关于新建系统调用这种工作已经完全是轻车熟路了！

7 总结

虽然时隔一年，但是对于当时实验的一些比较大的体会，我现在仍是历历在目，除了某些编码的细节记不太清楚了。

这个实验给我带来的最大收获是，除了内核编程和调试经验外，还有关于计算机组成、汇编语言等其他课程在讲课中没有涉及到的一些细节而实用的知识有了了解（例如在操作系统中实用的几条汇编指令等）。

总代码量约为 1100 行，不存在任何意义下的“借鉴”。