

武汉大学国家网络安全学院

操作系统设计与实践



专业名称：信息安全

实验内容：操作系统

指导教师：严飞教授

学生学号：2022302181148

学生姓名：黄东威

Contents

1 实验目的及实验内容	3
1.1 实验目的	3
1.2 实验内容	3
1.2.1 集成 OS	3
1.2.2 扩展 shell 要求	3
1.2.3 扩展系统日志能力	3
1.2.4 自我 OS 安全分析	4
1.2.5 自我 OS 防护	4
2 实验环境配置及工具简介	4
2.1 实验环境需求	4
2.2 工具简介	4
2.2.1 WSL2	5
2.2.2 Bochs	5
2.2.3 gdb	5
2.3 64 位系统编译 32 位代码	5
2.4 栈保护器	5
2.5 变量多重定义	6
2.6 位置无关重定位	8
2.7 对消息通信进行原子性保护	9
2.8 配置 gdb 进行调试	13
3 实验内容	17
3.1 集成 OS	17
3.1.1 硬盘启动 OS	17
3.1.2 内存分配与释放	19
3.1.3 多级反馈队列调度算法	19
3.1.4 目录树结构管理	20
3.2 扩展 shell 要求	21
3.2.1 问题介绍	21
3.2.2 进程管理	21
3.2.3 文件管理	27
3.2.4 并发运行任务	43
3.3 扩展系统日志能力	47
3.3.1 参数开关	47
3.3.2 进程运行	49
3.3.3 文件访问	53
3.3.4 系统调用	57
3.3.5 设备访问	60
3.4 自我 OS 安全分析	64
3.4.1 安全分析	64
3.4.2 自我 OS 防护之抗攻击	66
3.4.3 自我 OS 防护之抗泄露	69
4 实验总结与致谢	70

1 实验目的及实验内容

1.1 实验目的

1. 参照第 9——11 章内容，理清相关代码结构，以及 OrangeS 所支持的功能，开展实验任务
2. 结合所学的软件安全知识以及 OS 知识，分析掌握 OS 设计中潜在的安全问题
3. 学习与理解 OS 安全的基本思想与基本实现手段

1.2 实验内容

1.2.1 集成 OS

- 在已有实验代码基础上，将 1-8 章节（可以包含第 9 章）进行功能综合，形成自己的一个简易 OS
 - 可以考虑使用软盘或者硬盘，启动该 mini-OS
 - 能够实现你在前面章节所实现的，内存分配与释放
 - 能够实现你在前面章节所实现的多进程管理与调度
 - 所有代码需用目录树结构管理，并添加完整的 makefile 编译，以及文档
 - 鼓励支持较为复杂的进程调度算法、虚拟内存管理等

1.2.2 扩展 shell 要求

- 利用当前 OrangeS 所提供的系统调用和 API，扩展 Shell 命令
- 支持进程管理，包括：
 - 列出当前运行的进程
 - 终止指定的进程
- 支持文件管理，包括
 - 列出当前目录的文件，以及文件的相关属性信息
 - 创建新文件
 - 打开或编辑指定文件（如果是可执行文件，则运行，如果是文本文件，则打开可编辑）
 - 删除指定文件
- 支持在同一个 TTY 上，可并发运行多个 shell 任务

1.2.3 扩展系统日志能力

- 增加系统日志的框架，支持：
 - 使用参数开关相应的系统日志能力
 - 对进程的运行过程，可以进行日志记录

- 对文件的访问过程，可以进行日志记录
- 对系统调用过程，可以进行日志记录
- 对设备访问过程，可以进行日志记录

1.2.4 自我 OS 安全分析

- 分析提示：可执行文件的篡改、内核关键数据破坏、内存破坏漏洞、权限绕过等
- POC 实现：
 - 编写若干发现的漏洞 Demo 程序，对该 OS 实施攻击测试

1.2.5 自我 OS 防护

- 针对发现的攻击，进行防护
 - 可执行文件篡改，可采用静态装入度量
 - 运行时关键内核数据破坏，可采用运行时关键模块检查
 - 缓冲区溢出，可采用添加类似堆栈保护等方式
- 扩展 OS 的数据防泄露出能力
 - 技术路线一：对磁盘文件进行加密保护，要求文件打开时，读入内存的为明文，如果进行编辑，写回磁盘的为密文。密码算法、密钥管理可自行设计
 - 技术路线二：白名单进程允许访问指定文件资源，非白名单进程不允许访问受保护文件资源

2 实验环境配置及工具简介

本次期末大作业基于 oranges 操作系统的 chapter10 和 chapter11 为代码架构进行编码，以下将讲述如何进行环境搭建。

2.1 实验环境需求

- wsl2 ubuntu20.04
- gcc 11.4.0
- bochs 2.7
- NASM 2.15.05
- objdump 2.38
- gdb 12.1

2.2 工具简介

这里挑选比较有特征的实验工具进行介绍：

2.2.1 WSL2

WSL2 是微软提供的 Windows 子系统，允许用户在 Windows 10 和 Windows 11 操作系统上直接运行 Linux 环境。WSL2 引入了真正的 Linux 内核，提高了文件系统性能并增强了系统调用兼容性。这使得开发者可以在 Windows 系统上无缝运行 Linux 应用程序，支持软件开发、测试和运行 Linux 专有软件等场景。WSL2 支持完整的 Linux 命令行工具集，以及其他常用 Linux 应用程序和服务。

就我个人而言，WSL2 的开发方式和 vscode 结合得很好，只需要在 vscode 中安装 Remote-SSH 插件即可在 vscode 中连接 WSL2。而关于 WSL2 发行版，只需要在 Microsoft store 中安装对应的发行版即可。我在使用 WSL2 时，因为 WSL2 本身并没有自带的图形化界面，为了提升终端体验，我安装了 fish，有的用户也推荐使用 zsh，总之更换 shell 能有效提升效率，强烈建议。

2.2.2 Bochs

Bochs 是一个高度可配置的开源 IA-32 (x86) PC 模拟器，它模拟了 Intel x86 CPU、常见的 I/O 设备以及定制的 BIOS。Bochs 能够运行多种操作系统，如 Windows、Linux 或 BSD。由于 Bochs 完全在用户空间中模拟所有硬件，因此提供了极高的可移植性和调试环境的隔离性，但这也导致了相对较低的执行速度。

我在使用 bochs 时对原有的 bochs 进行了卸载使用源码重装的方式，主要的原因在于我想要配置 gdb 调试，下面将会有详细介绍。bochs 使用需要搭配 bochsrc，后文中将有多处修改 bochsrc 的地方，比如调整 RTC 频率的操作。

2.2.3 gdb

gdb 是一种功能强大的跨平台调试工具，用于调试多种编程语言编写的程序，包括 C、C++、Java、Fortran 和其他语言。GDB 允许看到程序执行过程中发生的事情。可以实时地启动程序，设定断点，改变程序内部的变量等。GDB 特别适合用于大型程序的调试，它支持从简单的单线程应用到复杂的多线程应用程序，甚至可以远程调试正在另一台计算机上运行的程序。

因为 gdb 的强大，我使用了 gdb 调试代替了原来的 bochs 内置调试器。gdb 的指令众多，作用丰富，在此不再赘述。如果想要提升 gdb 使用体验，可以安装 gdb 插件，比如 pwndbg，可以提供更丰富的指令集，高亮，调试功能等。

2.3 64 位系统编译 32 位代码

因为我们的系统是 64 位的，因此直接使用原书作者给的 Makefile 是不可以的。我们需要将编译的参数修改，使得编译时直接编译为 32 位的。

```
1 ASMFLAGS = -I include/ -I include/sys/ -f elf32
2 CFLAGS = -I include/ -I include/sys/ -c -fno-builtin -Wall -m32
3 LDFLAGS = -m elf_i386 -Ttext $(ENTRYPOINT) -Map krnl.map
```

2.4 栈保护器

编译时会遇到像下图一样的问题：

```

ld: kernel/main.o: in function `get_ticks':
main.c:(.text+0x3f8): undefined reference to `__stack_chk_fail'
ld: kernel/main.o: in function `untar':
main.c:(.text+0x7fa): undefined reference to `__stack_chk_fail'
ld: kernel/main.o: in function `panic':
main.c:(.text+0xda8): undefined reference to `__stack_chk_fail'
ld: kernel/keyboard.o: in function `keyboard_read':
keyboard.c:(.text+0x6dd): undefined reference to `__stack_chk_fail'
ld: kernel/tty.o: in function `tty_dev_write':
tty.c:(.text+0x643): undefined reference to `__stack_chk_fail'
ld: kernel/tty.o:tty.c:(.text+0x801): more undefined references to `__stack_chk_fail' follow
make: *** [Makefile:95: kernel.bin] Error 1

```

`_stack_chk_fail` 是 GCC 和其他编译器用于实现栈保护器的一个函数，链接器报错是因为编译环境启用了栈保护器，但链接时未找到 `_stack_chk_fail` 的实现，即：

- 链接时缺少标准库（如 libc），而 `_stack_chk_fail` 通常由标准库（如 glibc）提供
- 栈保护器未禁用，导致编译器插入了对 `_stack_chk_fail` 的调用

因此需要在 Makefile 中的 gcc 编译选项中加入 `-fno-stack-protector` 编译选项：

```

1 CC      = gcc
2 CFLAGS    = -I include/ -I include/sys/ -c -fno-builtin -Wall -m32 -fno-stack-
   protector

```

2.5 变量多重定义

有时在一起机器上会出现如下图所示的编译问题：

```

gcc -I include/ -I include/sys/ -c -fno-builtin -Wall -m32 -fno-stack-protector -o lib/exec.o lib/exec.c
ar rcs lib/orangescrt.a lib/syscall.o lib/printf.o lib/vsprintf.o lib/string.o lib/misc.o lib/open.o lib/read.o
ld -m elf_i386 -Ttext 0x1000 -Map krl.map -o kernel.bin kernel/kernel.o kernel/start.o kernel/main.o kernel/clo
task.o kernel/hd.o kernel/kliba.o kernel/klib.o lib/syslog.o mm/main.o mm/forkexit.o mm/exec.o fs/main.o fs/open
ld: fs/main.o:(.bss+0x0): multiple definition of `PARTITION_ENTRY'; kernel/hd.o:(.bss+0x0): first defined here
ld: fs/misc.o:(.bss+0x0): multiple definition of `PARTITION_ENTRY'; kernel/hd.o:(.bss+0x0): first defined here
ld: fs/disklog.o:(.bss+0x0): multiple definition of `PARTITION_ENTRY'; kernel/hd.o:(.bss+0x0): first defined here
make: *** [Makefile:95: kernel.bin] Error 1

```

这里提示 `multiple definition of 'PARTITION_ENTRY'`; `kernel/hd.o:(.bss+0x0): first defined here`，我们找到定义了 `PARTITION_ENTRY`，即 `include/sys/hd.h` 中，如下所示：

```

1 struct part_ent {
2     u8 boot_ind;          /**
3      * boot indicator
4      * Bit 7 is the active partition flag,
5      * bits 6-0 are zero (when not zero this
6      * byte is also the drive number of the
7      * drive to boot so the active partition
8      * is always found on drive 80H, the first
9      * hard disk).
10 */

```

```

11
12     u8 start_head;      /**
13      * Starting Head
14      */
15
16     u8 start_sector;    /**
17      * Starting Sector.
18      * Only bits 0-5 are used. Bits 6-7 are
19      * the upper two bits for the Starting
20      * Cylinder field.
21      */
22
23     u8 start_cyl;       /**
24      * Starting Cylinder.
25      * This field contains the lower 8 bits
26      * of the cylinder value. Starting cylinder
27      * is thus a 10-bit number, with a maximum
28      * value of 1023.
29      */
30
31     u8 sys_id;          /**
32      * System ID
33      * e.g.
34      * 01: FAT12
35      * 81: MINIX
36      * 83: Linux
37      */
38
39     u8 end_head;         /**
40      * Ending Head
41      */
42
43     u8 end_sector;      /**
44      * Ending Sector.
45      * Only bits 0-5 are used. Bits 6-7 are
46      * the upper two bits for the Ending
47      * Cylinder field.
48      */
49
50     u8 end_cyl;          /**
51      * Ending Cylinder.
52      * This field contains the lower 8 bits
53      * of the cylinder value. Ending cylinder
54      * is thus a 10-bit number, with a maximum
55      * value of 1023.
56      */
57
58     u32 start_sect; /**
59      * starting sector counting from
60      * 0 / Relative Sector. / start in LBA
61      */

```

```

62     u32 nr_sects;      /**
63      * nr of sectors in partition
64      */
65
66
67 } PARTITION_ENTRY;

```

有的情况下，给这个结构体加入 `extern` 关键字可以解决问题，然而对于 chapter11 中的代码，即使加入了 `extern` 仍然会有相同的问题。经过代码的审查，这里的”`PARTITION_ENTRY`“别名，并没有在别的地方使用过，因此直接删除这个别名即可解决。

2.6 位置无关重定位

在 64 位机器中，`kliba.asm` 中的 `disp_str` 函数存在问题，具体表现为，有可能启动 bochs 程序会直接停止，或者调用 `disp_str` 时打印的字符可能为乱码。该函数如下所示：

```

1 disp_str:
2     push    ebp
3     mov     ebp, esp
4
5     mov     esi, [ebp + 8] ; pszInfo
6     mov     edi, [disp_pos]
7     mov     ah, 0Fh
8 .1:
9     lodsb
10    test    al, al
11    jz     .2
12    cmp     al, 0Ah ; 是回车吗?
13    jnz    .3
14    push    eax
15    mov     eax, edi
16    mov     bl, 160
17    div     bl
18    and    eax, 0FFh
19    inc     eax
20    mov     bl, 160
21    mul     bl
22    mov     edi, eax
23    pop     eax
24    jmp    .1
25 .3:
26    mov     [gs:edi], ax
27    add     edi, 2
28    jmp    .1
29
30 .2:
31    mov     [disp_pos], edi
32
33    pop     ebp
34    ret

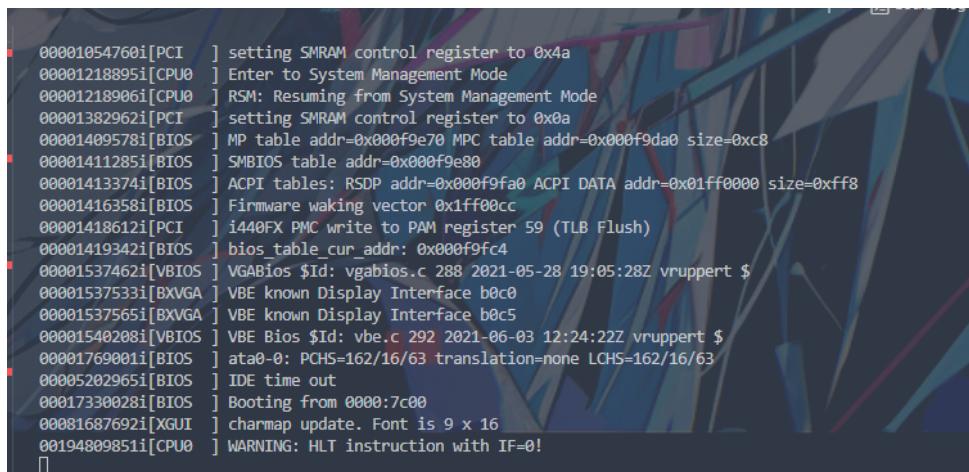
```

具体问题为新版本的编译器，在支持位置无关重定位时候，使用了 bx 寄存器，而该函数在这里没有对 bx 寄存器进行保护，因此可以通过 push 和 pop 的方式将 bx 保护起来。然而，这个解决方法，只是暂时的，对于早期代码较少时，只有 disp_str 函数出现问题时确实可以解决，但对于后期的大项目结构，难以定位问题，因此最好的解决办法为，告诉编译器启用位置无关重定位。具体为加入-fno-pie 参数即可

```
1 CFLAGS      = -I include/ -c -fno-builtin -m32 -fno-stack-protector -fno-pie
```

2.7 对消息通信进行原子性保护

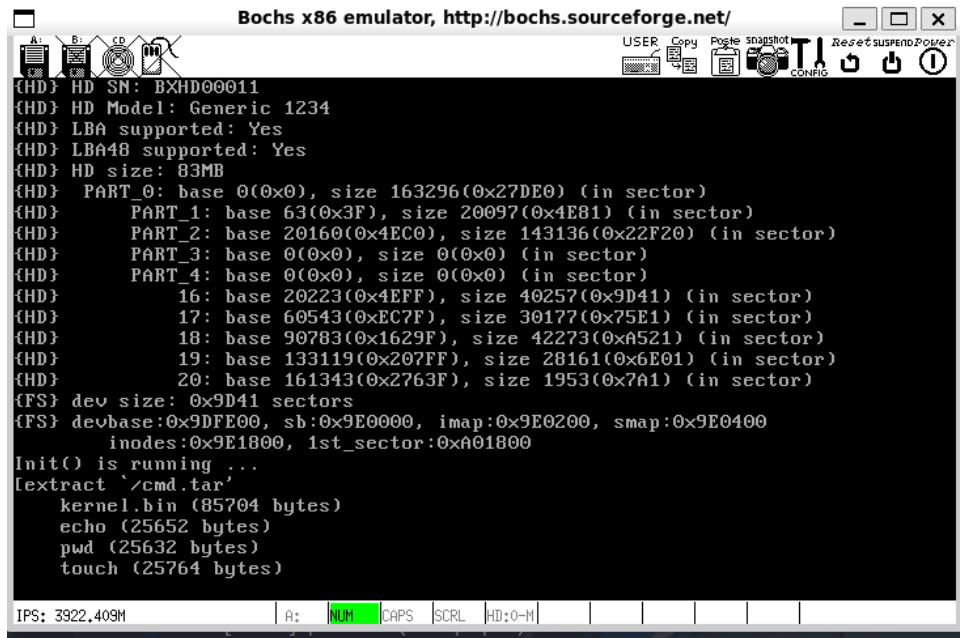
针对这部分内容主要是有的时候可能会遇见代码成功编译，然后运行 bochs，结果 bochs 直接报了 HLT 错误，HLT 在汇编语言中是处理器“暂停”指令，使程序停止运行，处理器进入暂停状态，不执行任何操作，不影响标志。报错类似于如下：



```
00001054760i[PCI] setting SMRAM control register to 0x4a
00001218895i[CPU0] Enter to System Management Mode
00001218906i[CPU0] RSM: Resuming from System Management Mode
00001382962i[PCI] setting SMRAM control register to 0x0a
00001409578i[BIOS] MP table addr=0x000f9e70 MPC table addr=0x000f9da0 size=0xc8
00001411285i[BIOS] SMBIOS table addr=0x000f9e80
00001413374i[BIOS] ACPI tables: RSDP addr=0x000f9fa0 ACPI DATA addr=0x01ff0000 size=0xff8
00001416358i[BIOS] Firmware waking vector 0x1ff00cc
00001418612i[PCI] i440FX PMC write to PAM register 59 (TLB Flush)
00001419342i[BIOS] bios_table_cur_addr: 0x000f9fc4
00001537462i[VBIOS] VGABios $Id: vgabios.c 288 2021-05-28 19:05:28Z vruppert $
00001537533i[BXVGA] VBE known Display Interface b0c0
00001537565i[BXVGA] VBE known Display Interface b0c5
00001540208i[VBIOS] VBE Bios $Id: vbe.c 292 2021-06-03 12:24:22Z vruppert $
00001769001i[BIOS] ata0-0: PCHS=162/16/63 translation=none LCHS=162/16/63
00005202965i[BIOS] IDE time out
00017330028i[BIOS] Booting from 0000:7c00
00081687692i[XGUI] charmap update. Font is 9 x 16
00194809851i[CPU0] WARNING: HLT instruction with IF=0!
```

HLT 错误

HLT 错误导致的原因很多种，我这里主要针对环境搭建过程中出现这种问题的原因进行解析。来看看 bochs 在 hlt 错误之后，显示了什么：



确实，bochs 已经停止了运行，不再继续执行，也不响应键盘中断了。别着急，仔细读一下打印内容，可以看到，Orange'S 顺利完成了初始化，甚至已经走到了开始解压 command 文件的过程，然而，在解压过程中出现了意外。经过对于代码的阅读，猜测这种问题的出现，极有可能是因为，作者在编写代码的时候没有对通信进行抗并发保护导致的。因此我们这里尝试给消息接受加上并发保护。并发保护常见的为使用锁或者原子性语句，然而这里不能用锁，只好使用 Orange'S 提供的 disable_int 指令来关中断模拟原子性。

该部分代码被修改与于 kernel/proc.c 的 msg_receive 中：

```

1 PRIVATE int msg_receive(struct proc *current, int src, MESSAGE *m)
2 {
3     struct proc *p_who_wanna_recv = current;
4
5     struct proc *p_from = 0;
6     struct proc *prev = 0;
7     int copyok = 0;
8     disable_int();
9     assert(proc2pid(p_who_wanna_recv) != src);
10
11    if ((p_who_wanna_recv->has_int_msg) &&
12        ((src == ANY) || (src == INTERRUPT)))
13    {
14
15        MESSAGE msg;
16        reset_msg(&msg);
17        msg.source = INTERRUPT;
18        msg.type = HARD_INT;
19
20        assert(m);
21
22        phys_copy(va2la(proc2pid(p_who_wanna_recv)), m), &msg,
23                  sizeof(MESSAGE));
24

```

```

25     p_who_wanna_recv->has_int_msg = 0;
26
27     assert(p_who_wanna_recv->p_flags == 0);
28     assert(p_who_wanna_recv->p_msg == 0);
29     assert(p_who_wanna_recv->p_sendto == NO_TASK);
30     assert(p_who_wanna_recv->has_int_msg == 0);
31
32     enable_int();
33     return 0;
34 }
35
36 if (src == ANY)
37 {
38
39     if (p_who_wanna_recv->q_sending)
40     {
41         p_from = p_who_wanna_recv->q_sending;
42         copyok = 1;
43
44         assert(p_who_wanna_recv->p_flags == 0);
45         assert(p_who_wanna_recv->p_msg == 0);
46         assert(p_who_wanna_recv->p_recvfrom == NO_TASK);
47         assert(p_who_wanna_recv->p_sendto == NO_TASK);
48         assert(p_who_wanna_recv->q_sending != 0);
49         assert(p_from->p_flags == SENDING);
50         assert(p_from->p_msg != 0);
51         assert(p_from->p_recvfrom == NO_TASK);
52         assert(p_from->p_sendto == proc2pid(p_who_wanna_recv));
53     }
54 }
55 else if (src >= 0 && src < NR_TASKS + NR_PROCS)
56 {
57
58     p_from = &proc_table[src];
59
60     if ((p_from->p_flags & SENDING) &&
61           (p_from->p_sendto == proc2pid(p_who_wanna_recv)))
62     {
63
64         copyok = 1;
65
66         struct proc *p = p_who_wanna_recv->q_sending;
67
68         assert(p);
69
70
71         while (p)
72         {
73             assert(p_from->p_flags & SENDING);
74
75             if (proc2pid(p) == src) /* if p is the one */

```

```

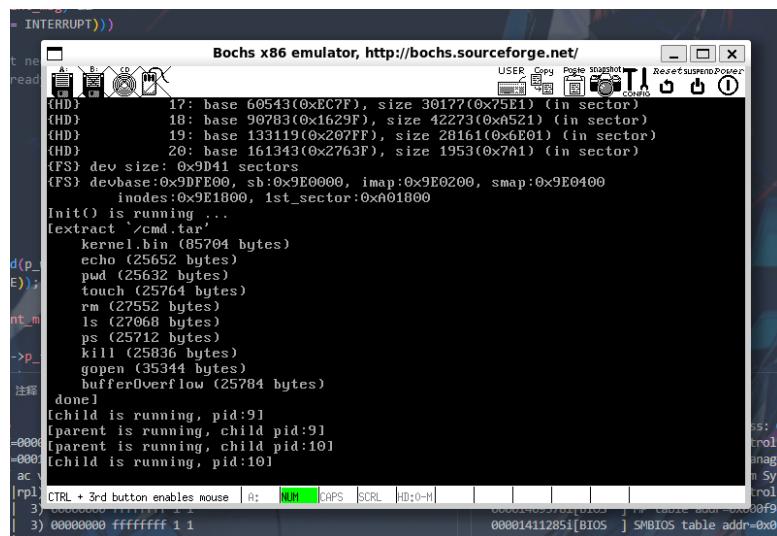
76         break;
77
78     prev = p;
79     p = p->next_sending;
80 }
81
82 assert(p_who_wanna_recv->p_flags == 0);
83 assert(p_who_wanna_recv->p_msg == 0);
84 assert(p_who_wanna_recv->p_recvfrom == NO_TASK);
85 assert(p_who_wanna_recv->p_sendto == NO_TASK);
86 assert(p_who_wanna_recv->q_sending != 0);
87 assert(p_from->p_flags == SENDING);
88 assert(p_from->p_msg != 0);
89 assert(p_from->p_recvfrom == NO_TASK);
90 assert(p_from->p_sendto == proc2pid(p_who_wanna_recv));
91 }
92 }
93
94 if (copyok)
95 {
96
97     if (p_from == p_who_wanna_recv->q_sending)
98     { /* the 1st one */
99         assert(prev == 0);
100        p_who_wanna_recv->q_sending = p_from->next_sending;
101        p_from->next_sending = 0;
102    }
103    else
104    {
105        assert(prev);
106        prev->next_sending = p_from->next_sending;
107        p_from->next_sending = 0;
108    }
109
110    assert(m);
111    assert(p_from->p_msg);
112
113    /* copy the message */
114    phys_copy(va2la(proc2pid(p_who_wanna_recv), m),
115              va2la(proc2pid(p_from), p_from->p_msg),
116              sizeof(MESSAGE));
117
118    p_from->p_msg = 0;
119    p_from->p_sendto = NO_TASK;
120    p_from->p_flags &= ~SENDING;
121    unblock(p_from);
122 }
123 else
124 { /* nobody's sending any msg */
125
126     p_who_wanna_recv->p_flags |= RECEIVING;

```

```

127
128     p_who_wanna_recv->p_msg = m;
129     p_who_wanna_recv->p_recvfrom = src;
130     block(p_who_wanna_recv);
131
132     assert(p_who_wanna_recv->p_flags == RECEIVING);
133     assert(p_who_wanna_recv->p_msg != 0);
134     assert(p_who_wanna_recv->p_recvfrom != NO_TASK);
135     assert(p_who_wanna_recv->p_sendto == NO_TASK);
136     assert(p_who_wanna_recv->has_int_msg == 0);
137 }
138 enable_int();
139
140 }
```

再次编译代码，可以看到 Orange'S 顺利运行了起来：



程序顺利运行

对于这部分，我们的解释是，因为在消息通信过程中，并没有并发保护机制，如果多个消息同时进行通信，可能造成错误的返回值、死锁等一系列问题。总之，这是应对 bochs 的“奇怪”崩溃的有用一招。

2.8 配置 gdb 进行调试

对于最终的大作业，我们如果只使用 bochs 内置的调试器（magic break）进行调试，那未免太痛苦了。这一节我将讲述如何使用 gdb 远程连接 bochs 进行代码调试。

bochs 内置的调试器和 gdb 调试是互斥的，而前面的章节我们选择使用 bochs 内置调试进行调试，所以这里要先对 bochs 的代码进行重新编译，具体编译流程中需要注意的为：

- ./configure --enable-gdb-stub 使用该选项，即可在编译时打开 gdb
- sudo make
- sudo make install

编译结束后，我们输入 bochs -help features 查看 bochs 支持的插件：

```
wisdomgo@DESKTOP-KITILU9 ~/oranges (master)> bochs --help features
=====
Bochs x86 Emulator 2.7
Built from SVN snapshot on August 1, 2021
Timestamp: Sun Aug 1 10:07:00 CEST 2021
=====
Supported features:
pci
gdbstub

00000000000i[SIM    ] quit_sim called with exit code 0
wisdomgo@DESKTOP-KITILU9 ~/oranges (master)>
```

可以看到这里已经有了 gdbstub 即为支持 gdb 调试，接下来需要修改 bochsrc，添加下面这一行内容：

```
1 gdbstub: enabled=1, port=1234
```

我们检查一下，首先 bochs -q 启动 bochs：

```
00000000000i[CPU0 ]      pge
00000000000i[CPU0 ]      pse36
00000000000i[CPU0 ]      mttr
00000000000i[CPU0 ]      pat
00000000000i[CPU0 ]      sysenter_sysexit
00000000000i[CPU0 ]      clflush
00000000000i[CPU0 ]      sse
00000000000i[CPU0 ]      sse2
00000000000i[CPU0 ]      mwait
00000000000i[CPU0 ]      xapic
00000000000i[PLUGIN] reset of 'pci' plugin device by virtual method
00000000000i[PLUGIN] reset of 'pcizisa' plugin device by virtual method
00000000000i[PLUGIN] reset of 'cmos' plugin device by virtual method
00000000000i[PLUGIN] reset of 'dma' plugin device by virtual method
00000000000i[PLUGIN] reset of 'pic' plugin device by virtual method
00000000000i[PLUGIN] reset of 'pit' plugin device by virtual method
00000000000i[PLUGIN] reset of 'vga' plugin device by virtual method
00000000000i[PLUGIN] reset of 'floppy' plugin device by virtual method
00000000000i[PLUGIN] reset of 'acpi' plugin device by virtual method
00000000000i[PLUGIN] reset of 'hpet' plugin device by virtual method
00000000000i[PLUGIN] reset of 'ioapic' plugin device by virtual method
00000000000i[PLUGIN] reset of 'keyboard' plugin device by virtual method
00000000000i[PLUGIN] reset of 'harddrv' plugin device by virtual method
00000000000i[PLUGIN] reset of 'pci_ide' plugin device by virtual method
00000000000i[PLUGIN] reset of 'unmapped' plugin device by virtual method
00000000000i[PLUGIN] reset of 'biosdev' plugin device by virtual method
00000000000i[PLUGIN] reset of 'speaker' plugin device by virtual method
00000000000i[PLUGIN] reset of 'extfpirq' plugin device by virtual method
00000000000i[PLUGIN] reset of 'parallel' plugin device by virtual method
00000000000i[PLUGIN] reset of 'serial' plugin device by virtual method
waiting for gdb connection on port 1234
```

bochs 已经启动，等待 gdb 连接，接下来，新开个 shell，进入 gdb，输入 target remote:1234 即可连接 bochs：

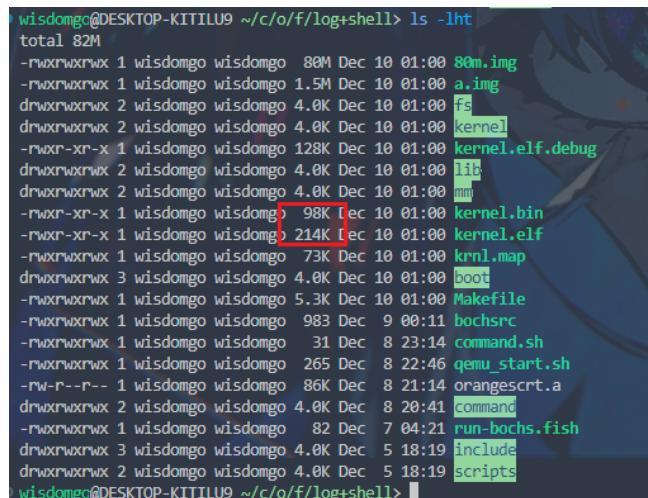
```
For help, type help .
Type "apropos word" to search for commands related to "word".
(gdb) target remote:1234
Remote debugging using :1234
warning: No executable has been specified and target does not support
determining executable automatically. Try using the "file" command.
0x0000ffff in ?? ()
(gdb)
```

可以看到能够连接成功。此时还不能够调试，因为我们缺少调试信息文件给到 gdb，我们需要生成一个带有调试信息的 elf 文件。首先想到的肯定是直接生成带有调试信息的 kernel.bin

文件，但其实，这个 kernel.bin 文件太大了，我尝试过无论是 bochs 还是 qemu 程序都会直接爆，不能够运行。本来卡在这里挺久，已经想不到好的办法解决，准备放弃的时候。想到一个好主意，我们生成一个有调试信息的 symbol 文件给到 gdb 进行调试，没有调试信息的文件给到 bochs 进行程序运行。生成调试信息，在 Makefile 中加入-g 选项，使用 objcopy 可以 elf 文件中有调试信息的部分剥离出来，具体而言，如下面一样使用 Makefile：

```
1 ASM      = nasm
2 DASM     = objdump
3 CC       = gcc
4 LD       = ld
5 ASMBFLAGS = -I boot/include/
6
7 ASMKFLAGS = -I include/ -I include/sys/ -f elf32 -g
8 CFLAGS   = -I include/ -I include/sys/ -c -fno-builtin -Wall
9          -m32 -fno-stack-protector -g
10 LDFLAGS  = -m elf_i386 -Ttext $(ENTRYPOINT) -Map krnl.map -g
11 DASMFLAGS = -D
12 ARFLAGS  = rcs
13
14 everything : $(ORANGESBOOT) kernel.elf
15
16 kernel.elf : $(OBJS) $(LIB)
17     $(LD) $(LDFLAGS) -o kernel.elf $(OBJS) $(LIB)
18     objcopy --strip-debug kernel.elf kernel.bin
19     objcopy --only-keep-debug kernel.elf kernel.elf.debug
```

编译后查看文件大小：



wisdomgo@DESKTOP-KITILU9 ~ / c/o / f / log+shell > ls -lht
total 82M
-rwxrwxrwx 1 wisdomgo wisdomgo 80M Dec 10 01:00 80m.img
-rwxrwxrwx 1 wisdomgo wisdomgo 1.5M Dec 10 01:00 a.img
drwxrwxrwx 2 wisdomgo wisdomgo 4.0K Dec 10 01:00 fs
drwxrwxrwx 2 wisdomgo wisdomgo 4.0K Dec 10 01:00 kernel
-rwxr-xr-x 1 wisdomgo wisdomgo 128K Dec 10 01:00 kernel.elf.debug
drwxrwxrwx 2 wisdomgo wisdomgo 4.0K Dec 10 01:00 lib
drwxrwxrwx 2 wisdomgo wisdomgo 4.0K Dec 10 01:00 mm
-rwxr-xr-x 1 wisdomgo wisdomgo 98K Dec 10 01:00 kernel.bin
-rwxr-xr-x 1 wisdomgo wisdomgo 214K Dec 10 01:00 kernel.elf
-rwxrwxrwx 1 wisdomgo wisdomgo 73K Dec 10 01:00 krnl.map
drwxrwxrwx 3 wisdomgo wisdomgo 4.0K Dec 10 01:00 boot
-rwxrwxrwx 1 wisdomgo wisdomgo 5.3K Dec 10 01:00 Makefile
-rwxrwxrwx 1 wisdomgo wisdomgo 983 Dec 9 00:11 bochssrc
-rwxrwxrwx 1 wisdomgo wisdomgo 31 Dec 8 23:14 command.sh
-rwxrwxrwx 1 wisdomgo wisdomgo 265 Dec 8 22:46 qemu_start.sh
-rw-r--r-- 1 wisdomgo wisdomgo 86K Dec 8 21:14 orangesrcrt.a
drwxrwxrwx 2 wisdomgo wisdomgo 4.0K Dec 8 20:41 command
-rwxrwxrwx 1 wisdomgo wisdomgo 82 Dec 7 04:21 run-bochs.fish
drwxrwxrwx 3 wisdomgo wisdomgo 4.0K Dec 5 18:19 include
drwxrwxrwx 2 wisdomgo wisdomgo 4.0K Dec 5 18:19 scripts
wisdomgo@DESKTOP-KITILU9 ~ / c/o / f / log+shell >

可以看到，在我实现结束我的大作业后的 kernel.bin 文件是 98k，kernel.elf 文件是 214k。接下来如何使用 gdb 就很简单了，在连接 bochs 时输入指令 add-symbol-file kernel.elf 即可，gdb 就有了调试信息文件，可以写如下的.gdbinit 方便使用：

```
1 # 加载符号文件
2 add-symbol-file kernel.elf
3
```

```
4 # 连接到 bochs GDB 服务器
5 target remote localhost:1234
```

```
(gdb) source .gdbinit
add symbol table from file "kernel.elf"
warning: No executable has been specified and target does not support
determining executable automatically. Try using the "file" command.
0x0000ffff in ?? ()
(gdb) l
1
2 ; ++++++-----+
3 ; ; kernel.asm
4 ; ++++++-----+
5 ;
6 ; ++++++-----+
7
8
9 %include "sconst.inc"
10
(gdb)
```

接下来就能愉快的使用 gdb 调试 bochs 了。

在此多赘述几句，在高版本的 linux 系统中，系统有可能禁止一个进程去监听另一个进程，对这里的影响为，gdb 不能够远程连接到 bochs，解决的办法有很多，比如修改系统安全等级等等。我的解决方法是，给 gdb 加入权限并且修改了配置，即使用 gdb 时直接为 sudo 启动 gdb，这样做的考量是可以使用 vscode 进行 bochs 的调试。写到这里，我将.vscode/launch.json 展示如下，该 json 文件的作用是为了使得 vscode 能够正确调用 gdb 和加载符号文件进行调试：

```
1 {
2     "version": "0.2.0",
3     "configurations": [
4         {
5             "name": "Debug Kernel with QEMU",
6             "type": "cppdbg",
7             "request": "launch",
8             "program": "${workspaceFolder}/kernel.elf",
9             "args": [],
10            "stopAtEntry": true,
11            "cwd": "${workspaceFolder}",
12            "environment": [],
13            "externalConsole": false,
14            "MIMode": "gdb",
15            "miDebuggerPath": "/usr/bin/gdb",
16            "setupCommands": [
17                {
18                    "description": "Enable pretty-printing for gdb",
19                    "text": "-enable-pretty-printing",
20                    "ignoreFailures": true
21                },
22            ],
23            // "setupCommands": [
24            //     {
```

```

25     "description": "Enable target remote debugging",
26     "text": "target remote localhost:1234",
27     "ignoreFailures": false
28   }
29 }
30 ]
31 }
32 }
```

3 实验内容

3.1 集成 OS

这一部分内容并不由我负责，我将以个人的理解简要讲讲我们组是如何集成 OS 的。

3.1.1 硬盘启动 OS

我们组使用的是基于 Orange'S 中 chapter11 的项目代码实现使用硬盘启动 Mini-OS，并且修复了其中项目代码的一些 bug。

书上 11.2 节讲述了软盘启动的过程：

- BIOS 将引导扇区读入内存 0000:7c00 处跳转到 0000:7c00 处开始执行引导代码
- 引导代码从软盘中找到 loader.bin 并将其读入内存
- 跳转到 loader.bin 开始执行
- loader.bin 从软盘中找到 kernel.bin 并将其读入内存
- 跳转到 kernel.bin 开始执行到此可认为启动过程结束

而硬盘启动和软盘启动的区别主要在于让引导扇区代码从硬盘中寻找 loader.bin 并让 loader 从硬盘中寻找 kernel.bin。只要重写 boot.asm 和 loader.asm，让它们读取硬盘即可，新文件被取名为 hdboot.asm 和 hdldr.asm

代码主体我就不再实验报告中呈现，直接查看随书代码即可。简单讲讲就是让代码先读取超级块、读取根目录，从根目录中寻找 hdldr.bin，将 hdldr.bin 读入内存并将控制权交给它。随后便只需修改 loader.asm，形成一个 hdldr.asm，这部分需要修改读取 kernel.bin 的部分，且和从 hdbboot.asm 中读取 hdldr.bin 的流程类似。

之前的实验中我们组和严飞老师进行过讨论，Orange'S 判断死锁的方式其实是不够严谨的。具体描述就是，我们能够注意到 (attention is all you need) 对于死锁而言，死锁一定有环，但是有环不一定是死锁。而于渊老师对于死锁的判断方式是检测一个环路，即 A->B->C->A，倘若出现这样的环，那么就发生了死锁。这样的想法是不够严谨的，没有考虑到如果出现 A->B->C->B 的死锁情形。因此我们组给出一个更合理的死锁检测代码：

```

1 #define HASH_TABLE_SIZE NR_PROCS+NR_TASKS
2
3 struct proc* p = proc_table + dest;
4 clear_hash_table(); // 每次检测死锁前都要清空哈希表
5
```

```

6  while (1) {
7      if (p->p_flags & SENDING) {
8          if (hash(p->p_sendto)) {
9              /* print the chain */
10             int curcle = p->p_sendto;
11             int flag = 0;
12             p = proc_table + dest;
13             printf("=_=%s", p->name);
14             do {
15                 if(p->p_sendto == curcle) {
16                     flag++;
17                 }
18                 assert(p->p_msg);
19                 p = proc_table + p->p_sendto;
20                 printf(" ->%s", p->name);
21             } while (flag != 2);
22             printf("=_=");
23             return 1; // 发生了死锁
24         }
25         p = proc_table + p->p_sendto;
26     } else {
27         break;
28     }
29 }
30 return 0;
31
32 PUBLIC int hash_table[HASH_TABLE_SIZE];
33
34 extern int hash_table[];
35
36 int hash(int src) {
37     int index = src % HASH_TABLE_SIZE; // 使用简单取模法生成哈希值
38     if (hash_table[index] == src) {
39         return 1; // 冲突, 表示可能存在死锁
40     } else if (hash_table[index] == 0) {
41         hash_table[index] = src; // 将进程ID 存入哈希表
42         return 0; // 正常插入
43     } else {
44         return 1; // 发生了冲突, 出现了死锁
45     }
46 }
47
48 void clear_hash_table() {
49     for (int i = 0; i < HASH_TABLE_SIZE; i++) {
50         hash_table[i] = 0; // 重置哈希表
51     }
52 }

```

3.1.2 内存分配与释放

好了，我们来看看内存分配与释放，这部分也在之前的实验中完成过，可以查看我们组曾经的实验报告，写的蛮详细的。这里就简单讲讲，同时因为代码不是我编写的，不厚颜无耻的将代码呈现在报告中。

我们这里使用 alloc_pages 和 free_pages 函数来进行内存的分配与释放，它们分别实现了建立地址间的映射关系和消除地址间的映射关系的功能。我们对于之前的代码进行了升级，我们将原有的 alloc_pages 修改为了 alloc_a_4k_page 函数以此进行以 4k 为粒度的页分配。alloc_a_4k_page 函数主要是从物理地址 0x00000000 开始从位图中寻找是否存在空闲页，在没超过最大位数前寻找到空闲页，随后经过左移 12 位的方式来计算物理地址，最终返回该物理地址实现分页。free_pages 将会从页目录表中获取地址，然后找到需要释放的物理地址，清除每个页表项的最后 12 位，清除位图中的相应位。

3.1.3 多级反馈队列调度算法

这里先来讲讲比较一些常见的调度算法及其特点：

1. 多级反馈队列 (MFQ)

- MFQ 允许进程在多个队列之间根据其行为和需求动态移动，优先级随着执行时间的增长而改变
- 系统有多个队列，每个队列有不同的优先级，通常优先级越高的队列时间片越短
- 长时间得不到服务的进程可以提升优先级，防止饥饿
- 算法能根据进程的行为自动调整，适合多种类型的负载

2. 保证调度

- 每个进程获得一定比例的 CPU 时间，这个比例是预先设定的
- 可以保证每个进程在特定时间内得到服务，适用于对响应时间有严格要求的环境
- 系统按照进程的重要性和需求分配处理器时间

3. 最短作业优先 (SJF)

- SJF 可以是非抢占式的，也可以是抢占式的（称为最短剩余时间优先，SRTF）
- 理论上可以提供最优的平均等待时间
- 长作业可能会不断被短作业抢占，导致饥饿

4. 公平分享调度

- 按用户或组分配 CPU 时间，确保公平
- 防止单个用户占用过多资源，适合大型多用户系统
- 可能会牺牲系统整体的吞吐量来保证公平性

结合严飞老师与我们组的交流结果和之前的多级反馈队列调度算法的实现，我们按照严飞老师的要求，对该算法进行了修改和增进，使得其更符合系统中的使用。

在这里我将大致阐述实现思路。我们将给出三个队列，用于存储不同的优先级进程，给这三个队列分配内存空间和时间片，时间片可以根据优先级不同做出一定量的区分，防止低优先级饥饿。对于每个进程，标记出其需要运行的时间、是否在队列中、在哪一个优先级队列中。接下来，对时钟中断处理程序进行修改，使其能够按照需求，不断减少进程的时间片。

进程调度过程中封装了用于寻找下一个需要运行的进程的函数 `get_next_proc`, 以及面向进程的出队入队函数。考虑了进程继续运行、进程结束、切换队列、抢占发生的四种情况进行调度算法实现。

后续在运行代码时会出现程序崩溃的情况，经过 debug, 定位到原因为 fork 时，没能给子进程初始化导致子进程没激活。修改代码后，多级反馈队列算法顺利实现，具体实现的效果可以查看我的队友的实验报告，该部分写的较为详细，我们最终呈现的效果也十分不错。

3.1.4 目录树结构管理

该部分实现较为简单，我们把整个项目作为一颗树来管理即可。其实作者给我们整理的项目结构十分的清晰，对于硬盘、库函数、内核函数、boot 阶段、系统调用等等的代码进行了统筹，管理的很好。如果想要查看我们的项目是什么结构，可以直接使用 Linux 中的 tree 指令，下图为展示结果：

```
wisdomgo@DESKTOP-KITILU9 ~/c/o/f/logs> tree
.
├── 80m.img
├── Makefile
├── a.img
├── bochssrc
└── boot
    ├── boot.asm
    ├── boot.bin
    ├── include
    │   └── fat12hdr.inc
    │       ├── load.inc
    │       └── pm.inc
    ├── loader.asm
    └── loader.bin
    └── command
        ├── Makefile
        ├── attack
        ├── attack.c
        ├── bufferOverflow
        ├── bufferOverflow.c
        ├── echo
        ├── echo.c
        ├── gopen
        ├── gopen.c
        ├── inst.tar
        ├── kernel.bin
        ├── kill
        ├── kill.c
        ├── ls
        ├── ls.c
        ├── ps
        ├── ps.c
        ├── pwd
        ├── pwd.c
        ├── rm
        ├── rm.c
        └── start.asm
```

在本次实验的过程中，我发现存在受限于 32 位虚拟机，难以安装 IDE 进行 OS 实验的情形时有发生。于我而言，我使用了 vscode 进行代码编辑，目录管理清晰，配合合适的插件能够大幅提升代码编写速度。针对不能安装 IDE 的情况，我提供以下两种解决思路：

- 使用物理机的 IDE 的远程连接功能，直接在物理机中进行代码编写
 - 可以尝试使用 nvim，配合合适的插件，能够达到 vscode 中的效果。
- 此外，因为使用 shell 极多，我强烈建议安装 fish/zsh 提升使用体验。

3.2 扩展 shell 要求

3.2.1 问题介绍

本任务是 OS 期末实验的第二个基本任务，主要目标是理清代码结构并扩展 shell，提供一些自制的指令方便后续实验。具体要求如下：

- 参照第 10 章、第 11 章内容，理清相关代码结构，以及 OrangeS 所支持的功能，扩展 shell，完成如下任务：
 - 利用当前 OrangeS 所提供的系统调用和 API，编写 2 个以上可执行程序（功能自定），并编译生成存储在文件系统中
 - 在 Shell 中调入你所编写的可执行程序，启动并执行进程
- 注意使用教材中所提供的系统调用来实现
 - 进程结束后返回 shell

这部分我们将根据 oranges 操作系统提供的 chapter10 的代码进行实验，chapter10 提供了一个简单的 shell，因此给了我们实现 shell 指令的 api。, oranges 操作系统已经具备了将 command 中的.c 文件根据 makefile 的指令进行编译、链接、并装载入操作系统中的能力。之后，在 bochs 中输入命令名称就可以调用这些指令了。

3.2.2 进程管理

列出当前运行的进程 这一部分仿照 Linux 系统的 ps 指令，具体功能为，输入 ps 后，将会列出当前系统中的所有进程，并且列出进程的 pid 和进程的状态。

具体而言，代码写在了 command/ps.c 中，内容如下：

```

1 #include "type.h"
2 #include "stdio.h"
3 #include "string.h"
4 #include "sys/const.h"
5 #include "sys/protect.h"
6 #include "sys/fs.h"
7 #include "sys/proc.h"
8 #include "sys/tty.h"
9 #include "sys/console.h"
10 #include "sys/global.h"
11 #include "sys/proto.h"

12
13 int main(int argc, char *argv[])
{
14     MESSAGE msg;
15     struct proc p;
16     printf("PID  NAME  FLAGS\n");

```

```

18     int i = 0;
19     for (i = 0; i < NR_TASKS + NR_PROCS; i++)
20     {
21         msg.PID = i;
22         msg.type = GET_PROC_INFO;
23         msg.BUF = &p;
24         send_recv(BOTH, TASK_SYS, &msg);
25         if (p.p_flags != FREE_SLOT)
26         {
27             printf("%d %s ", i, p.name);
28             if (p.p_flags == SENDING)
29             {
30                 printf("SENDING\n");
31             }
32             else if (p.p_flags == RECEIVING)
33             {
34                 printf("RECEIVING\n");
35             }
36             else if (p.p_flags == WAITING)
37             {
38                 printf("WAITING\n");
39             }
40             else if (p.p_flags == HANGING)
41             {
42                 printf("HANGING\n");
43             }
44             else
45             {
46                 printf("Unknown\n");
47             }
48         }
49     }
50 }
51 }
```

该部分使用了代码中对于进程的 flags 的设置，如下：

```

1 #define SENDING 0x02 /* set when proc trying to send */
2 #define RECEIVING 0x04 /* set when proc trying to recv */
3 #define WAITING 0x08
4 #define HANGING 0x10
5 #define FREE_SLOT 0x20 /* set when proc table entry is not used \
6
7 */
```

在 execv 运行过程中，如果解析到了收到的字符串为 “ps” 将会调用 ps.c 中的代码，这部分代码会新建一个进程，并且把该进程的 type 设置为 GET_PROC_INFO，接下来通过微内核的同步 IPC 机制调度执行该项遍历任务 GET_PROC_INFO 任务的代码在 kernel/systask.c 中，如下：

```

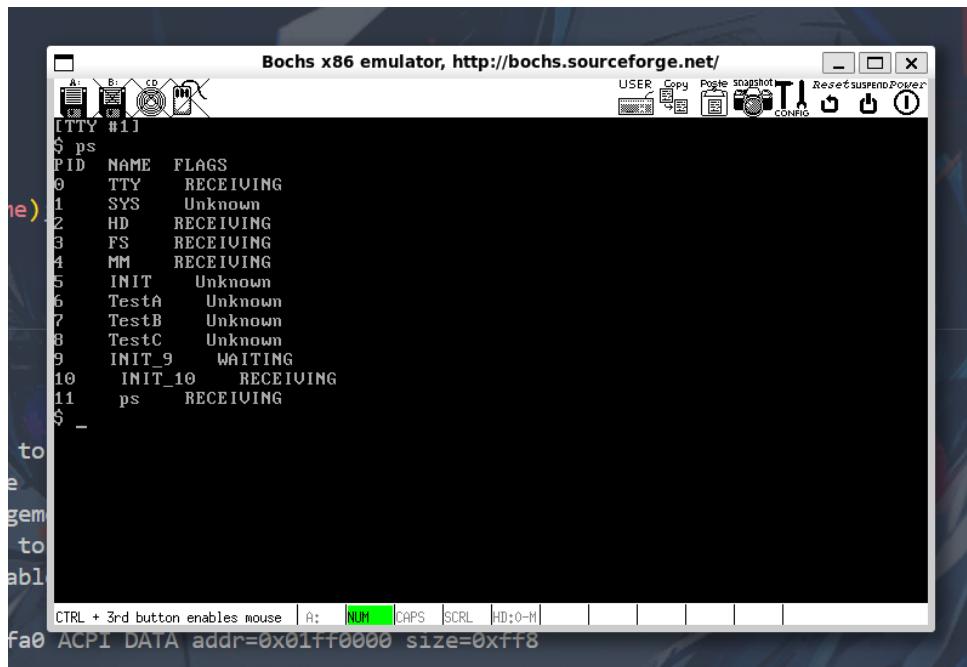
1  switch(msg.type)
2  {
3      case GET_PROC_INFO:
4          msg.type = SYSCALL_RET;
5          phys_copy(va2la(src, msg.BUF),
6                     va2la(TASK_SYS, &proc_table[msg.PID]),
7                     sizeof(struct proc));
8          send_recv(SEND, src, &msg);
9          break;
10 }

```

通过对线性地址的访问，将会遍历 proc_table 中当前存储的所有内容，proc_table 定义如下：

```
1 extern struct proc proc_table[]
```

存储了进程的具体信息，遍历之后将遍历结果返回到 ps.c 中，随后打印到 shell 中，下图是实现后的展示结果：



终止指定的进程 该部分内容主要是仿照 Linux 系统实现一个 kill 指令，kill 指令通过发送信号，将会终止进程，并且释放进程所占据的资源。接下来看看 kill 指令怎么实现。kill 会通过 execv 而传递给 command/kill.c，该部分代码编辑如下：

```

1 #include "type.h"
2 #include "stdio.h"
3 #include "string.h"
4 #include "sys/const.h"
5 #include "sys/protect.h"
6 #include "sys/fs.h"

```

```

7 #include "sys/proc.h"
8 #include "sys/tty.h"
9 #include "sys/console.h"
10 #include "sys/global.h"
11 #include "sys/proto.h"
12
13 int string_to_int(const char *str)
14 {
15     int result = 0;
16     int sign = 1; // 默认正数
17
18     // 检查是否有正负号
19     if (*str == '-')
20     {
21         sign = -1;
22         str++; // 跳过符号
23     }
24     else if (*str == '+')
25     {
26         str++; // 跳过符号
27     }
28
29     // 遍历每个字符
30     while (*str != '\0')
31     {
32         // 如果不是数字, 返回 0 或报错
33         if (*str < '0' || *str > '9')
34         {
35             printf("Error: Invalid input string.\n");
36             return 0; // 或根据需要返回错误码
37         }
38
39         // 累计计算结果
40         result = result * 10 + (*str - '0');
41         str++;
42     }
43
44     return result * sign; // 返回结果 (带符号)
45 }
46
47 int main(int argc, char *argv[])
48 {
49     if (argc != 2)
50     {
51         printf("Usage: kill -pid\n");
52         return 1;
53     }
54
55     // 检查输入格式
56     if (argv[1][0] != '-')
57     {

```

```

58     printf("Invalid format. Use: kill -pid\n");
59     return 1;
60 }
61
62 // 去掉 '--', 解析为整数
63 int pid = string_to_int(&argv[1][1]); // 跳过第一个字符 '--'
64
65 if (pid <= 0)
66 { // 检查 pid 是否有效
67     printf("Invalid PID.\n");
68     return 1;
69 }
70
71 MESSAGE msg;
72 msg.type = SHUT_PROC;
73 msg.PID = pid;
74 send_recv(BOTH, TASK_SYS, &msg); // 向内核发送消息
75
76 if (msg.RETVAL == -1)
77 {
78     printf("Error: Invalid PID.\n");
79 }
80 else
81 {
82     printf("Process %d has been killed.\n", pid);
83     // log_event_pid("Process killed", pid);
84 }
85
86 return 0;
87 }

```

代码中有较为消息的注释，简单来说，就是在 shell 中输入 kill -[pid] 的指令，代码先将 pid 从字符串解析为 int 类型数字，然后通过 IPC 传递给 TASK_SYS，由 TASK_SYS 对 pid 执行 SHUT_PROC 操作，SHUT_PROC 是新增的宏定义，定义在 include/sys/const.h 中，具体如下：

```

1 enum msgtype
2 {
3     ...
4     SHUT_PROC,
5     ...
6 }

```

通过 send_recv 将会给到 systask.c 代码，执行如下内容：

```

1 case SHUT_PROC:
2     msg.type = SYSCALL_RET;
3
4     // // 验证 PID 的合法性

```

```

5   if (msg.PID < 0 || msg.PID >= NR_PROCS + NR_TASKS
6       || proc_table[msg.PID].p_flags == FREE_SLOT)
7   {
8       msg.RETVAL = -1; // 错误码, 表示无效的进程 ID
9   }
10  else
11  {
12      struct proc *p = &proc_table[msg.PID];
13      p->p_flags = FREE_SLOT; // 标记为空闲
14      msg.RETVAL = 0;           // 成功
15  }
16  send_recv(SEND, src, &msg);
17  break;

```

在这里，将会验证 PID 的合法性，PID 要在 0 和 NR_PROCS 和 NR_TASKS 的范围内，同时能够 kill 一个进程的前提是，该进程还没有被设置为空闲。经过检查后，将会执行代码，从 proc_table 中找到对应的 PID，将该 proc 的 p_flags 标记为 FREE_SLOT，FREE_SLOT 为代码框架中提供的表示进程空闲的宏定义。经过研究发现，设置为 FREE_SLOT 后，系统会先保留这部分进程所占据的资源，但是不执行，当有新的进程执行时，会从该进程的资源这里直接开始覆盖。因此算是实现了释放资源的功能。由于 kill 指令不应该能够杀死系统中的关键进程，这很有可能导致系统的崩溃，我们演示时，主要针对可以关闭的 INIT_10 进程。以下为演示图片：

首先执行 ps 指令查看当前有哪些进程，再执行 kill 指令进行删除：

```

[TTY #1]
$ ps
PID  NAME  FLAGS
0   TTY    RECEIVING
1   SYS    Unknown
2   HD     RECEIVING
3   FS     RECEIVING
4   MM     RECEIVING
5   INIT   Unknown
6   TestA  Unknown
7   TestB  Unknown
8   TestC  Unknown
9   INIT_9  WAITING
10  INIT_10 RECEIVING
11  ps     RECEIVING
$ kill -10
Process 10 has been killed.
$ ps
PID  NAME  FLAGS
0   TTY    RECEIVING
1   SYS    Unknown
2   HD     RECEIVING
3   FS     RECEIVING
4   MM     RECEIVING
5   INIT   Unknown
IPS: 134.406M          | A:  NUM CAPS SCRL HD:0-M | | | | |

```

可以看到，系统提示我们，10 号进程已经被杀死，这个时候我们再执行 ps 指令查看 proc_table 中的 10 号进程是否被覆写：

```
1  ps      RECEIVING
$ kill -10
Process 10 has been killed.
$ ps
PID  NAME  FLAGS
0   TTY   RECEIVING
1   SYS   Unknown
2   HD    RECEIVING
3   FS    RECEIVING
4   MM    RECEIVING
5   INIT  Unknown
6   TestA Unknown
7   TestB Unknown
8   TestC Unknown
9   INIT_9 Unknown
10  ps    RECEIVING
$
```

10 号进程已经不在进程表中，同时如果运行 ps 指令，ps 指令会占据 10 号进程的资源。
kill 指令实现完毕。

3.2.3 文件管理

列出目录文件，以及文件相关属性 该部分将仿照 Linux 系统中的 ls 指令和 ls -l 指令，具体而言 ls 命令默认会列出当前目录下的文件和子目录，ls -l 列出详细的文件信息，包括文件类型、大小等。

该部分代码实现在 command/ls.c 中，如下：

```
1 #include "stdio.h"
2 #include "string.h"
3
4 int main(int argc, char *argv[])
5 {
6     char buf[1024] = {0};
7     int flag = 0;
8
9     if (argc > 1)
10    {
11        if (strcmp(argv[1], "-l") == 0 && argc == 2)
12        {
13            flag = 1;
14        }
15        else
16        {
17            printf("Usage: %s [-l]\n", argv[0]);
18            return 1;
19        }
20    }
21
22    int ret = ls("/", buf, flag);
23
24    if (!ret)
25        printf("%s\n", buf);
26    else
27        printf("Error: failed to read directory contents.\n");
28
29    return 0;
```

```
30 }
```

这部分将会分析 shell 得到的字符串是否符合输入规范，同时是否包含 “-l” 需求，如果有列出详细信息的需求，将会把 ls 的 flag 置为 1，然后进入到 ls 函数中打印文件相关信息。ls 函数实现在了 lib/ls.c 中，如下：

```
1 #include "type.h"
2 #include "stdio.h"
3 #include "const.h"
4 #include "protect.h"
5 #include "string.h"
6 #include "fs.h"
7 #include "proc.h"
8 #include "tty.h"
9 #include "console.h"
10 #include "global.h"
11 #include "keyboard.h"
12 #include "proto.h"
13 PUBLIC int ls(char *path, char *buf, int flag)
14 {
15     MESSAGE msg;
16     msg.type = LS;
17     msg.PATHNAME = (void *)path;
18     msg.BUF = (void *)buf;
19     msg.BUF_LEN = 1024;
20     msg.FLAGS = flag;
21     send_recv(BOTH, TASK_FS, &msg);
22
23     return msg.RETVAL;
24 }
```

这部分将使用一个 MESSAGE 结构体，该结构体的 type 为宏定义 LS，该宏定义加入在了 include/sys/const.h 中的 msgtype 枚举类型中。通过消息通信，TASK_FS 将会执行相关操作，让我们来看看这部分代码：

```
1     switch(msgtype)
2     {
3         case LS:
4             fs_msg.RETVAL = do_ls();
5             break;
6     }
7
8 #ifdef ENABLE_DISK_LOG
9     switch (msgtype)
10    {
11        case LS:
12    }
```

不难发现，这部分内容只是将 LS 指令解析为更底层的函数 do_ls，在这里将会实现 ls 相关操作，do_ls 函数实现在 fs/misc.c 中：

```
1 PUBLIC int do_ls()
2 {
3     // char s[1024] = {0};
4     // fs_msg.BUF = s;
5     // fs_msg.BUF_LEN = 1024;
6     char *dir = va2la(fs_msg.source, fs_msg.PATHNAME);
7     char *buf = va2la(fs_msg.source, fs_msg.BUF);
8     // char *dir = fs_msg.PATHNAME; // 输入路径
9     // char *buf = fs_msg.BUF; // 输出缓冲区
10
11    int buf_size = fs_msg.BUF_LEN; // 假设 BUF_LEN 定义了 BUF 的大小
12
13    int tt = 0;
14
15    struct inode *dir_inode = root_inode;
16
17    int dir_blk0_nr = dir_inode->i_start_sect;
18    int nr_dir_blocks = (dir_inode->i_size + SECTOR_SIZE - 1)
19    / SECTOR_SIZE;
20    int nr_dir_entries = dir_inode->i_size / DIR_ENTRY_SIZE;
21
22    int m = 0;
23    struct dir_entry *pde;
24    struct dir_entry entries[nr_dir_entries];
25    for (int i = 0; i < nr_dir_blocks; i++)
26    {
27        RD_SECT(dir_inode->i_dev, dir_blk0_nr + i);
28        pde = (struct dir_entry *)fsbuf;
29
30        for (int j = 0; j < nr_dir_entries; j++, pde++)
31        {
32            memcpy(&entries[j], pde, sizeof(struct dir_entry));
33        }
34        for (int j = 0; j < SECTOR_SIZE / DIR_ENTRY_SIZE; j++)
35        {
36
37            if (fs_msg.FLAGS)
38            {
39                struct inode *pin =
40                    get_inode(dir_inode->i_dev,
41                    entries[j].inode_nr);
42                char mode_str[11] = "-----";
43                switch (pin->i_mode & I_TYPE_MASK)
44                {
45                    case I_DIRECTORY:
46                        mode_str[0] = 'd';
47                        break; // 目录
48                    case I_REGULAR:
```

```

49         mode_str[0] = '-';
50         break; // 普通文件
51     case I_CHAR_SPECIAL:
52         mode_str[0] = 'c';
53         break; // 字符设备
54     case I_BLOCK_SPECIAL:
55         mode_str[0] = 'b';
56         break; // 块设备
57     case I_NAMED_PIPE:
58         mode_str[0] = 'p';
59         break; // 命名管道
60     default:
61         mode_str[0] = '?';
62         break; // 未知类型
63     }

64
65     // 所有者权限
66     if (pin->i_mode & 0400)
67         mode_str[1] = 'r'; // 读
68     if (pin->i_mode & 0200)
69         mode_str[2] = 'w'; // 写
70     if (pin->i_mode & 0100)
71         mode_str[3] = 'x'; // 执行

72
73     // 组权限
74     if (pin->i_mode & 0040)
75         mode_str[4] = 'r';
76     if (pin->i_mode & 0020)
77         mode_str[5] = 'w';
78     if (pin->i_mode & 0010)
79         mode_str[6] = 'x';

80
81     // 其他用户权限
82     if (pin->i_mode & 0004)
83         mode_str[7] = 'r';
84     if (pin->i_mode & 0002)
85         mode_str[8] = 'w';
86     if (pin->i_mode & 0001)
87         mode_str[9] = 'x';

88
89     // 格式化文件信息到缓冲区
90     int len = sprintf(buf + tt,
91                     "%s %4d %8d %5d %5d %4d %s\n",
92                     mode_str,           // 文件类型和权限
93                     pin->i_cnt,        // 引用计数
94                     pin->i_size,        // 文件大小
95                     pin->i_start_sect, // 起始扇区
96                     pin->i_nr_sects,   // 总扇区数
97                     pin->i_num,         // inode 编号
98                     entries[j].name    // 文件名
99     );

```

```

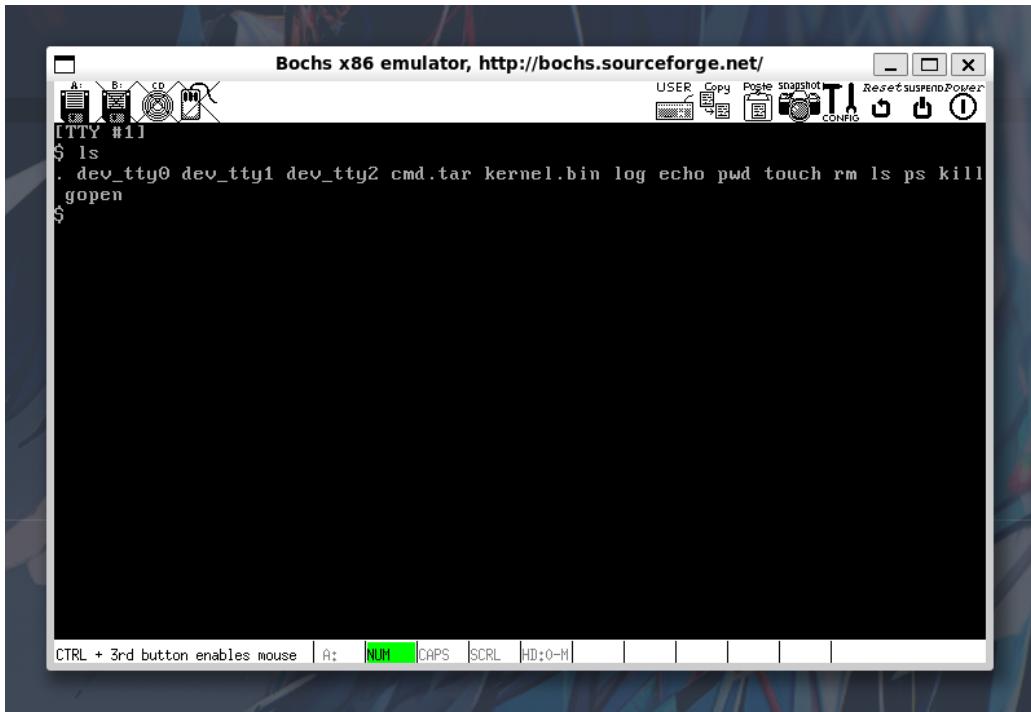
100
101     if (tt + len >= buf_size)
102     {
103
104         return -1; // 缓冲区不足, 返回错误
105     }
106
107     tt += len; // 更新缓冲区偏移
108     put_inode(pin);
109 }
110 else
111 {
112     if (tt + strlen(entries[j].name)
113         + 2 > buf_size)
114     {
115
116         return -1; // 缓冲区不足, 返回错误
117     }
118
119     memcpy(buf + tt, entries[j].name,
120             strlen(entries[j].name));
121     tt += strlen(entries[j].name);
122
123     buf[tt++] = '\n'; // 添加换行符
124 }
125 if (++m >= nr_dir_entries)
126     break;
127 }
128
129 if (m >= nr_dir_entries)
130     break;
131 }
132
133 buf[tt] = '\0'; // 确保字符串结尾
134 // printf("%s\n", buf);
135 return 0; // 成功
136 }

```

这里的主要目标是通过文件的 inode 结构体和目录项 (dir_entry) 信息，输出当前目录下的文件及其相关属性，下面展开讲讲：

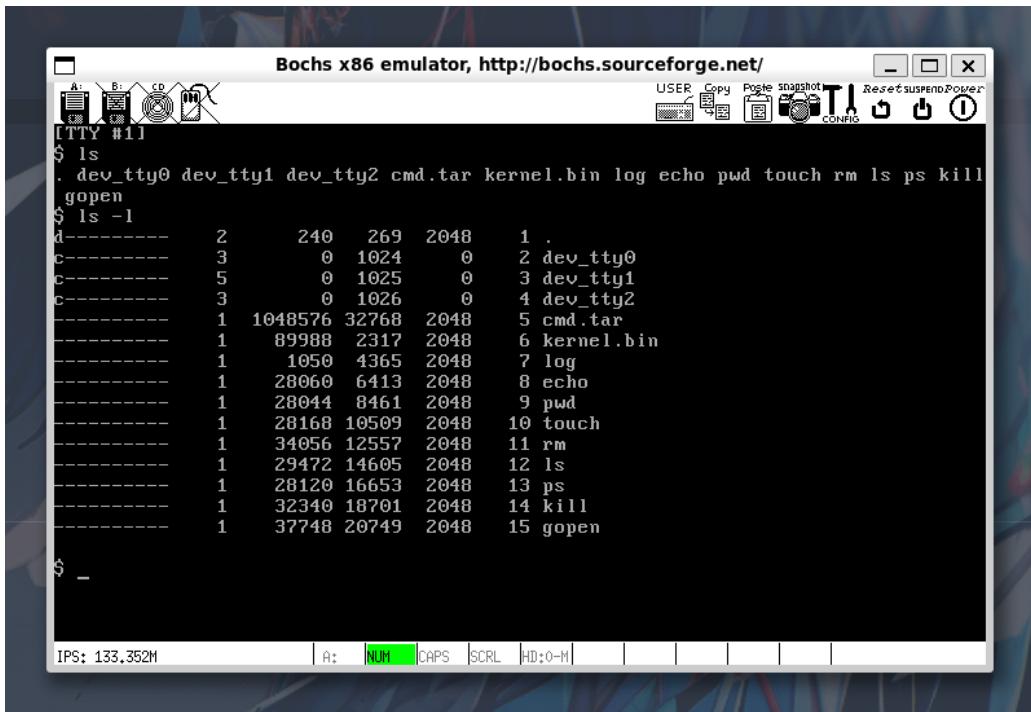
首先解析传入的目录路径 (dir)，通过 fs_msg.BUF 获取一个缓冲区，用于存储输出，buf_size 定义缓冲区大小。随后将会获得根目录的 inode 结构体，该结构体通过 root_inode 获取，接下来就是计算目录占用的块数、扇区号、包含的条目数。计算完毕将会通过遍历目录块的方式得到目录项，获取的内容保存到 fsbuf 中。

接下来为输出部分，输出内容将根据 fs_msg.FLAGS 进行区别，fs_msg.FLAGS 由 command/ls.c 中的 flag 决定，如果为真将会得到目录项的详细信息，包括文件权限、引用计数、文件大小、起始扇区、总扇区数、文件名等信息。如果为假则只得到文件名。现在将得到的信息存储到 buf 中，返回到 command/ls.c 中，ls.c 中将会把 buf 打印到 shell 中，效果展示如下：



```
Bochs x86 emulator, http://bochs.sourceforge.net/
[TTY #1]
$ ls
. dev_tty0 dev_tty1 dev_tty2 cmd.tar kernel.bin log echo pwd touch rm ls ps kill
gopen
$
```

ls 指令



```
Bochs x86 emulator, http://bochs.sourceforge.net/
[TTY #1]
$ ls
. dev_tty0 dev_tty1 dev_tty2 cmd.tar kernel.bin log echo pwd touch rm ls ps kill
gopen
$ ls -l
d----- 2 240 269 2048 1 .
c----- 3 0 1024 0 2 dev_tty0
c----- 5 0 1025 0 3 dev_tty1
c----- 3 0 1026 0 4 dev_tty2
----- 1 1048576 32768 2048 5 cmd.tar
----- 1 89988 2317 2048 6 kernel.bin
----- 1 1050 4365 2048 7 log
----- 1 28060 6413 2048 8 echo
----- 1 28044 8461 2048 9 pwd
----- 1 28168 10509 2048 10 touch
----- 1 34056 12557 2048 11 rm
----- 1 29472 14605 2048 12 ls
----- 1 28120 16653 2048 13 ps
----- 1 32340 18701 2048 14 kill
----- 1 37748 20749 2048 15 gopen
$ _
```

ls -l 指令

我们的 ls 和 ls -l 指令打印出了很多信息，甚至包括后续实现的指令文件。在这里我不想对打印的信息做过多展示，我觉得很有趣的是，虽然 orange'S 的作者于渊先生没有和 Linus 对于 Linux 系统一样使用了宏内核而是选择了微内核，但他们比较一致的思想是我曾经觉得很有趣的一句话，叫做“everything is a file”。这句话最早学到来源于 b 站偶然看到过南京大学蒋炎岩老师的操作系统课程中，他现场讲述了这句话的含义，linux 系统把所有东西都当作文件来管理，我们可以看看下面这张图，来源于我的 WSL2 ubuntu20.04 系统：

```
wisdomc@DESKTOP-KITILU9:~/c/o/f/l/log_new [1]> cd /proc
wisdomc@DESKTOP-KITILU9:/proc> ls
1 1219 15 39 48 68 757 buddyinfo config.gz devices execdomains iomen kcore kpagegroup locks modules pagetypeinfo softirqs sysvipc uptime zoneinfo
10334 127 16 44 59 69 8211 bus consoles diskstats filesystems iports key-users keys kpagecount mdstat mounts partitions stat swap version
10577 13258 170 458 60 7 9954 cgroups cpufreq dma fs interrupts kallsyms kmsg loadavg misc net schedstat self sys timer_list vmallocinfo
121 14 38 469 61 78 acpi cmdline crypto driver irq kallsyms kmsg loadavg misc net schedstat self sys timer_list vmstat
wisdomc@DESKTOP-KITILU9:/proc>
```

在/proc 文件夹中，我们惊讶的看到有许多文件，有的文件名真的很熟悉，比如 fs driver net tty 等，没错他们就是这些文件所代表的东西，还有些数字命名的目录，其实这些目录就是我们系统目前正在运行的进程。这真的很神奇，Linux 系统下的一切原来是那么的清晰，所有的东西，原来不过都是个“file”，说真的，它们真的没有多神秘，原来就在随处可见的地方。

好了，回到我们的 Orange' S，我们这里打印的信息有我们的三个 tty，有我们打包的 cmd.tar，有我们后面实现的日志文件，有内核文件，更重要的还有 shell 指令文件。原来我们自己实现的各种指令，就像 Linux 下的进程一样，当做了文件放置在目录下，everything is a file，在小小的 Orange 'S 上也得到了体现，真是让人惊喜。

创建新文件 该部分将完成一个 touch 指令，touch 指令主要能够创建一个文本类型的文件，集成在 command/touch.c 中，该部分代码如下：

```
1 #include "stdio.h"
2
3 int main(int argc, char *argv[])
4 {
5     if (argc < 2)
6     {
7         printf("Usage: %s <filename> [filename...]\\n", argv[0]);
8         return 1;
9     }
10
11    for (int i = 1; i < argc; i++)
12    {
13        int fd = open(argv[i], O_CREAT);
14        if (~fd)
15        {
16            printf("Successfully created %s.\\n", argv[i]);
17            close(fd);
18        }
19        else
20        {
21            printf("File %s already exists or
22                  cannot be created. Please check.\\n", argv[i]);
23        }
24    }
25
26    return 0;
27 }
```

除去老生常谈的检查输入是否合法以外，着重介绍当执行 touch.c 时，我们给 open 传入了什么。我们将 argv[1] 即文件的名字和 open 的操作码 O_CREAT 传递给了 open 函数。没错，O_CREAT 代表，我们这次调用 open 的意义是为了创建一个文件，显然，我们将查看

如何解析这个文件名和 O_CREAT。open 函数最终是通过 task_fs() 转到 do_open 函数中，do_open 函数中的这一部分将会执行解析创建文件的需求：

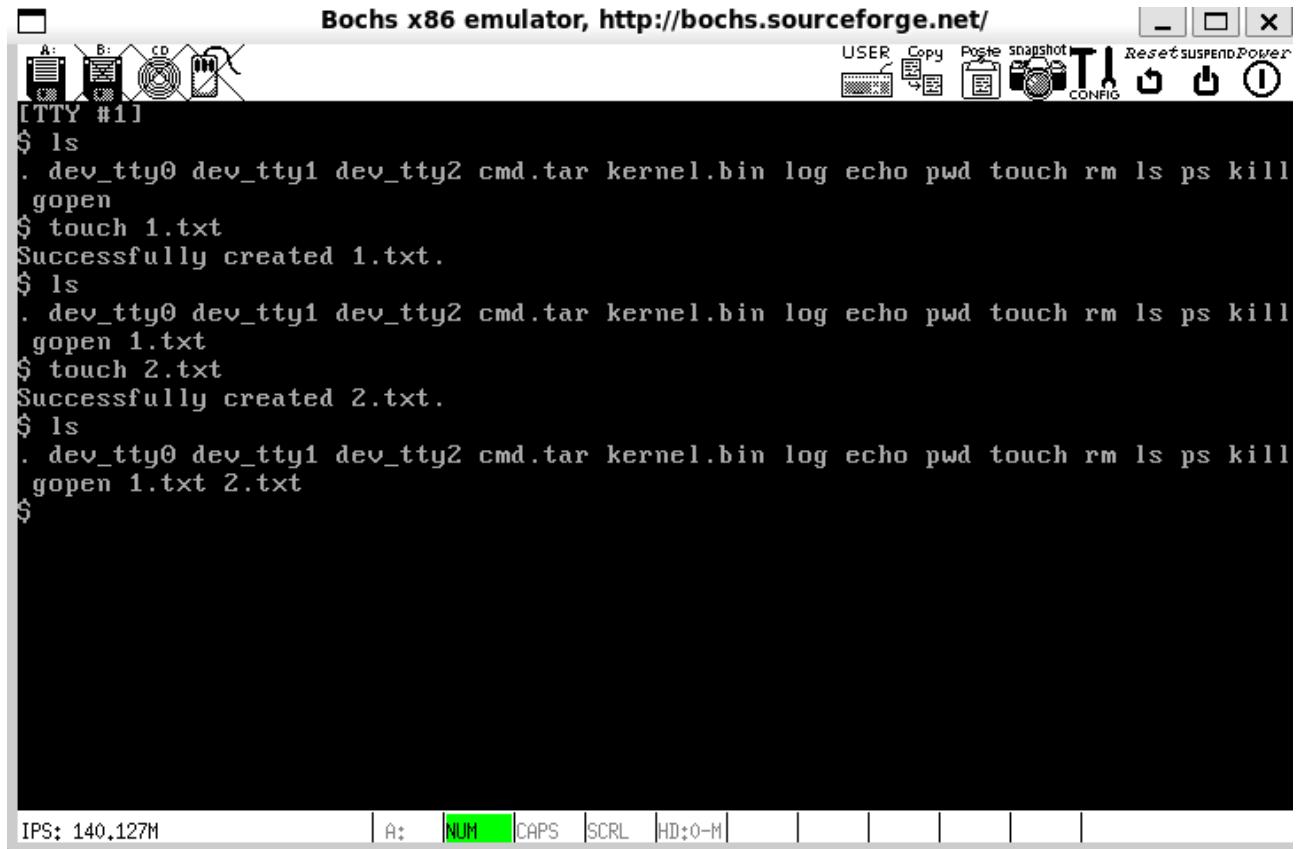
```
1 if (flags & O_CREAT)
2 {
3     if (inode_nr)
4     {
5         printf("{FS} file exists.\n");
6         return -1;
7     }
8     else
9     {
10        pin = create_file(pathname, flags);
11    }
12 }
13 else
14 {
15     assert(flags & O_RDWR);
16
17     char filename[MAX_PATH];
18     struct inode *dir_inode;
19     if (strip_path(filename, pathname, &dir_inode) != 0)
20         return -1;
21     pin = get_inode(dir_inode->i_dev, inode_nr);
22 }
```

显然，因为 O_CREAT 在 touch.c 中被传递了过来，因此为 1，接下来将执行 create_file 函数，该函数位于 fs/open.c 中，如下：

```
1 PRIVATE struct inode *create_file(char *path, int flags)
2 {
3     char filename[MAX_PATH];
4     struct inode *dir_inode;
5     if (strip_path(filename, path, &dir_inode) != 0)
6         return 0;
7
8     int inode_nr = alloc_imap_bit(dir_inode->i_dev);
9     int free_sect_nr = alloc_smap_bit(dir_inode->i_dev,
10                                         NR_DEFAULT_FILE_SECTS);
11     struct inode *newino = new_inode(dir_inode->i_dev, inode_nr,
12                                     free_sect_nr);
13
14     new_dir_entry(dir_inode, newino->i_num, filename);
15     // 清空分配的扇区内容
16     memset(fsbuff, 0, FSBUFSIZE);
17     WR_SECT(newino->i_dev, newino->i_start_sect);
18     return newino;
19 }
```

简单来说该部分将提取当前的目录路径，然后分配一个新的 inode，给其分配扇区空间，最后使用我们提供的 filename 创建新的文件项。执行完后将会清空新分配的扇区，返回创建的 inode。

接下来展示下 touch 的效果：



The screenshot shows a terminal window titled "Bochs x86 emulator, http://bochs.sourceforge.net/" running on TTY #1. The terminal displays the following session:

```
$ ls
. dev_tty0 dev_tty1 dev_tty2 cmd.tar kernel.bin log echo pwd touch rm ls ps kill
$ gopen
$ touch 1.txt
Successfully created 1.txt.
$ ls
. dev_tty0 dev_tty1 dev_tty2 cmd.tar kernel.bin log echo pwd touch rm ls ps kill
$ gopen 1.txt
$ touch 2.txt
Successfully created 2.txt.
$ ls
. dev_tty0 dev_tty1 dev_tty2 cmd.tar kernel.bin log echo pwd touch rm ls ps kill
$ gopen 1.txt 2.txt
```

At the bottom of the terminal window, there is a status bar showing "IPS: 140.127M". Below the terminal window, the keyboard state is indicated with a green "NUM" key and other keys like "A:", "CAPS", "SCRL", and "HD:0-M".

touch 指令

通过效果展示，可以看到，最开始根目录下没有用户创建的文件，用户在分别 touch 了 1.txt 和 2.txt 时使用上面实现的 ls 指令后，能够观察到对应的文件出现在了根目录下。

比较可惜的是，Orange's S 给我们提供的系统能够容纳的文件数量较少，因此只能稍微的“touch”几个文件就得草草收手，不然会导致脆弱的 os 系统的崩溃。完成 touch 指令时的兴奋感和激动感被这个小意外冲刷掉了许多。没关系，这里给出解决方案，我们可以将文件的块大小改小，作者本人给我们的文件预留的块太大了，实际上很多时候用不完那么多，我们可以略微修改块的大小，以此容纳更多文件。当然，最好的解决办法我觉得是，创建一个更大的硬盘用来装载我们的系统。

打开或编辑指定文件 很遗憾，受限于时间、技术能力、代码架构等等的限制，我们只能做个较为简易的编辑方案，无法实现像 vim、nano、emacs 那样，但我觉得还好，我们有勇于探索的心，这就够了。我们实现了 gopen 指令，该指令可以用来打开和编辑指定文件，具体表现为如果是文本文件可以编辑，可执行文件，可以运行。该部分代码集成在 command/gopen.c 中，如下：

```
1 #include "stdio.h"
2 #include "elf.h"
3 #include "string.h"
```

```

4 #define INFO_PREFIX "[INFO] "
5 #define ERROR_PREFIX "[ERROR] "
6 #define INPUT_PREFIX "[INPUT] "
7 #define CONTENT_PREFIX "[CONTENT] "
8 #define EXECUTE_PREFIX "[EXECUTE] "

9
10 int main(int argc, char *argv[])
{
11     if (argc < 2)
12     {
13         printf(ERROR_PREFIX "Usage: %s <filename>\n", argv[0]);
14         return 1;
15     }
16
17     char *filename = argv[1];
18
19     if (!check_file(filename))
20     {
21         printf(ERROR_PREFIX
22             "Failed to strip path or not a regular file: %s\n",
23             filename);
24         return 1;
25     }
26
27     int fd = open(filename, O_RDWR);
28     if (fd == -1)
29         fd = open(filename, O_CREAT | O_RDWR);
30     int n;
31     assert(fd != -1);
32
33     char rdbuf[2048];
34     int fsize = lseek(fd, 0, SEEK_END);
35     if (fsize >= sizeof(Elf32_Ehdr))
36     {
37         lseek(fd, 0, SEEK_SET);
38         Elf32_Ehdr elf_header;
39         read(fd, &elf_header, sizeof(elf_header));
40         if (elf_header.e_type ==
41             ET_EXEC ||
42             (elf_header.e_type == ET_DYN
43             && elf_header.e_entry != 0))
44         {
45             printf(INFO_PREFIX
46                 "This is an executable file. Executing...\n"
47                 EXECUTE_PREFIX "\n");
48             char *args[] = {filename};
49             execv(args[0], args);
50             close(fd);
51             return 0;
52         }
53     }
54 }
```

```

55 // 询问用户选择
56 lseek(fd, 0, SEEK_SET);
57 if (fszie)
58 {
59     n = read(fd, rdbuf, fszie < sizeof(rdbuf) ? fszie : sizeof(rdbuf));
60     if (strlen(rdbuf) < n)
61     {
62         printf(ERROR_PREFIX
63             "This is neither a text file
64             nor an executable file. \n");
65         return 1;
66     }
67     printf(INFO_PREFIX "This is a text file. Opening...\n");
68     printf(INFO_PREFIX "Current content:\n");
69     write(1, rdbuf, n);
70     write(1, "\n", 1);
71 }
72 else
73 {
74     printf(INFO_PREFIX "This is a text file. Opening...\n");
75     printf(INFO_PREFIX "File is empty.\n");
76 }
77
78 if (!strcmp(filename, "log"))
79 {
80     printf(INFO_PREFIX "You have no access to modify this file.\n");
81     return 1;
82 }
83 printf(INPUT_PREFIX
84 "Do you want to (A)ppend or (O)verwrite the file? ");
85
86 char choice = ' ';
87 int fd_stdin = 0;
88 read(fd_stdin, &choice, 1); // 从标准输入读取用户选择
89 if (choice == 'o' || choice == 'O')
90 {
91     // 覆盖文件内容
92     lseek(fd, 0, SEEK_SET); // 重置文件指针
93     printf(INFO_PREFIX "File content will be overwritten.\n");
94 }
95 else if (choice == 'a' || choice == 'A')
96 {
97     // 追加内容
98     lseek(fd, 0, SEEK_END); // 将文件指针移动到文件末尾
99     printf(INFO_PREFIX "File content will be appended.\n");
100 }
101 else
102 {
103     printf(ERROR_PREFIX "Invalid choice. Exiting edit mode.\n");
104     close(fd);

```

```

106     return 0;
107 }
108
109 // 写入新内容
110 printf(INPUT_PREFIX "Enter new
111 content to write to the file: \n");
112 char input_buffer[512] = {0};
113 n = read(fd_stdin, input_buffer, sizeof(input_buffer));
114
115 n = write(fd, input_buffer, n);
116 printf(INFO_PREFIX "%d bytes written to the file.\n", n);
117
118 // 显示更新后的内容
119 lseek(fd, 0, SEEK_SET);
120 fsize = lseek(fd, 0, SEEK_END);
121 lseek(fd, 0, SEEK_SET);
122
123 n = read(fd, rdbuf, fsize < sizeof(rdbuf) ? fsize : sizeof(rdbuf));
124 printf(INFO_PREFIX "Updated content:\n");
125 // printf(CONTENT_PREFIX "%s\n", rdbuf);
126 write(1, rdbuf, n);
127 write(1, "\n", 1);
128
129 close(fd);
130
131 return 0;
132 }
```

这部分内容很长，除去编辑时打印在终端用于提示用户的信息，主要分为以下部分：

- 验证并打开文件，如果没有提供文件路径，会打印错误信息并退出
- 检查文件类型，如果文件是可执行文件，则通过 execv 执行该文件。
- 如果文件是普通文本文件，它会读取文件内容并输出到 shell 中
- 如果是文本文件，gopen 将询问用户是选择 append（追加）还是 overwrite（覆盖）文本中的内容，以此达到可以将新内容写入文件的目的，或者修改文本的目的
- 修改完毕，gopen 将把完整的文本内容再次输出到 shell 中。

这里有一些内容不得不提：

```

1  char rdbuf[2048];
2  int fsize = lseek(fd, 0, SEEK_END);
3  if (fsize >= sizeof(Elf32_Ehdr))
4  {
5      lseek(fd, 0, SEEK_SET);
6      Elf32_Ehdr elf_header;
7      read(fd, &elf_header, sizeof(elf_header));
8      if (elf_header.e_type == ET_EXEC ||
9          (elf_header.e_type == ET_DYN && elf_header.e_entry != 0))
```

```

10     {
11         printf(INFO_PREFIX "This is an
12             executable file. Executing...\\n" EXECUTE_PREFIX "\\n");
13         char *args[] = {filename};
14         execv(args[0], args);
15         close(fd);
16         return 0;
17     }
18 }
```

该部分通过 lseek 和 read 来识别文件是否为 ELF 文件，下面的部分比如

```

1 lseek(fd, 0, SEEK_SET);
2 if (fsiz)
3 {
4     n = read(fd, rdbuf, fsiz < sizeof(rdbuf) ? fsiz : sizeof(rdbuf));
5     if (strlen(rdbuf) < n)
6     {
7         printf(ERROR_PREFIX "This is
8             neither a text file nor an executable file. \\n");
9         return 1;
10    }
11    printf(INFO_PREFIX "This is a text file. Opening...\\n");
12    printf(INFO_PREFIX "Current content:\\n");
13    write(1, rdbuf, n);
14    write(1, "\\n", 1);
15 }
16 else
17 {
18     printf(INFO_PREFIX "This is a text file. Opening...\\n");
19     printf(INFO_PREFIX "File is empty.\\n");
20 }
21
22 if (!strcmp(filename, "log"))
23 {
24     printf(INFO_PREFIX "You have no access to modify this file.\\n");
25     return 1;
26 }
27 printf(INPUT_PREFIX "Do you want to (A)ppend or (O)verwrite the file? ");
```

则是通过 write 调用，用于往文件中写入内容，当然也有对于鉴权的设定，比如系统日志(log)文件，是不应该进行修改的。

可以看下 fopen 的具体使用过程：首先我使用 touch 创建一个文件，往 fopen 中写入 hdw2022302181148：

```
[TTY #1]
$ touch hdw.txt
Successfully created hdw.txt.
$ gopen hdw.txt
[INFO] This is a text file. Opening...
[INFO] File is empty.
[INPUT] Do you want to (A)ppend or (O)verwrite the file? a
[INFO] File content will be appended.
[INPUT] Enter new content to write to the file:
hdw2022302181148
[INFO] 16 bytes written to the file.
[INFO] Updated content:
hdw2022302181148
$
```

gopen

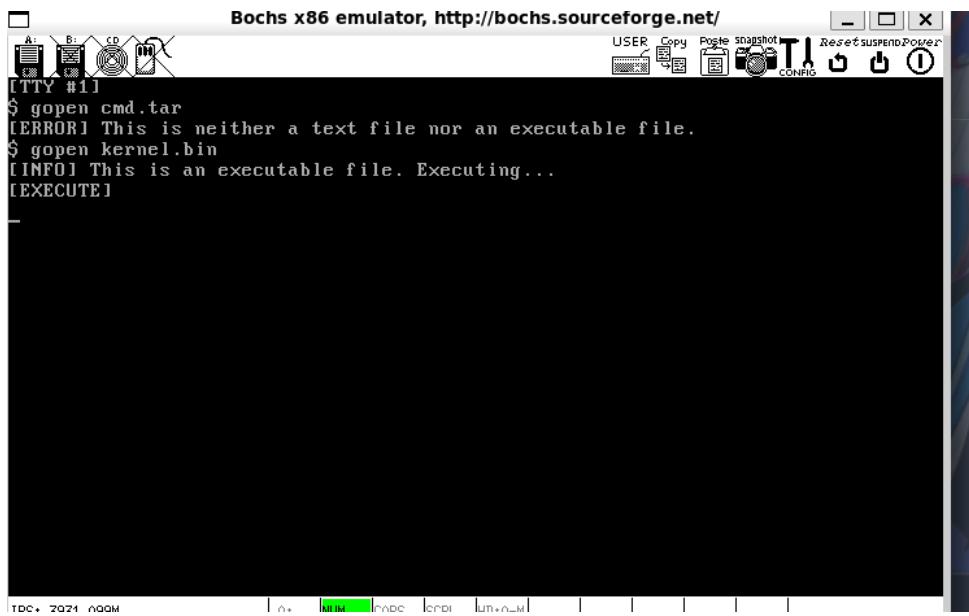
可以看到写入后，gopen 提示我们，这次写入了 16 字节，然后再次将写入内容打印出来提示我们，接下俩我们可以再次打开文件进行文件的修改：

```
[TTY #1]
$ touch hdw.txt
Successfully created hdw.txt.
$ gopen hdw.txt
[INFO] This is a text file. Opening...
[INFO] File is empty.
[INPUT] Do you want to (A)ppend or (O)verwrite the file? a
[INFO] File content will be appended.
[INPUT] Enter new content to write to the file:
hdw2022302181148
[INFO] 16 bytes written to the file.
[INFO] Updated content:
hdw2022302181148
$ gopen hdw.txt
[INFO] This is a text file. Opening...
[INFO] Current content:
hdw2022302181148
$ ddao
[INPUT] Do you want to (A)ppend or (O)verwrite the file? a
[INFO] File content will be appended.
[INPUT] Enter new content to write to the file:
just a simple try
[INFO] 17 bytes written to the file.
[INFO] Updated content:
hdw2022302181148just a simple try
$
```

gopen append

我们往文本内容中追加了“just a simple try”，gopen 再次给我们展示了文本中现在的内容，为“hdw2022302181148just a simple try”。

除了文本文件，我们看看 gopen 对于可执行文件会做什么：



gopen elf

可以看到，我们的系统中，除了文本文件、elf 文件还有个压缩文件 (cmd.tar)，gopen 提示，我们不能够打开它。当我们使用 gopen 打开可执行文件 kernel.bin 时，gopen 提示我们程序已经开始执行了，当然 shell 里没有任何回显，程序也已经卡死了，仔细想想，这也是应该的，Orange ‘S 自然没有执行 kernel.bin 文件的能力，强行让它执行，只会让它崩溃。别说 Orange’S 了，就算我们的 Linux 或者 Windows 系统，也不能够直接执行 kernel.bin 文件，因此 gopen 的反应，也只是符合预期。

删除指定文件 这部分就是做一个 rm 指令，用于删除一些文件，rm 指令实现在 command/rm.c 中，该部分如下：

```
1 #include "stdio.h"
2
3 int main(int argc, char *argv[])
4 {
5     if (argc < 2)
6     {
7         printf("Usage: %s <filename> [filename...]\\n", argv[0]);
8         return 1;
9     }
10
11     int result = 0;
12
13     for (int i = 1; i < argc; i++)
14     {
15         int fd = unlink(argv[i]);
16         if (~fd)
17         {
18             printf("File %s deleted.\\n", argv[i]);
19         }
20     }
21 }
```

```

20     else
21     {
22         printf("Failed to delete file %s.\n", argv[i]);
23         result = -1;
24     }
25 }
26
27 return result;
28 }
```

解析完 shell 提供的指令后，主要通过 unlink 函数实现删除操作，unlink 函数在 lib/unlink.c 中，代码中我只截取有关部分：

```

1 PUBLIC int unlink(const char *pathname)
2 {
3     // Check if the filename ends with ".log"
4     MESSAGE msg;
5     if (strcmp(pathname, "log") == 0)
6     {
7         msg.RETVAL = -1;
8     }
9     else
10    {
11        msg.type = UNLINK;
12        msg.PATHNAME = (void *) pathname;
13        msg.NAME_LEN = strlen(pathname);
14        send_recv(BOTH, TASK_FS, &msg);
15    }
16    return msg.RETVAL;
17 }
```

这部分除去常规的告知 log 是不可以修改的，我们用 UNLINK 宏（定义在 msgtype 类型中）来指定要完成的 TASK_FS，UNLINK 宏将会时消息通信跳转到 do_unlink 函数。do_unlink 函数很长，简单介绍就是，可以用于删除一个文件，do_unlink 会做一些检查，比如只能删除 I_REGULAR 类型的文件，该类型文件代表常规文件，除此以外，它会释放文件的资源、删除目录项、通过重置 inode 中的文件属性，代表文件已删除。

rm 指令演示如下，我将先创建 1.txt 和 2.txt，然后通过 rm 删除，最终使用 ls 查看是否删除：

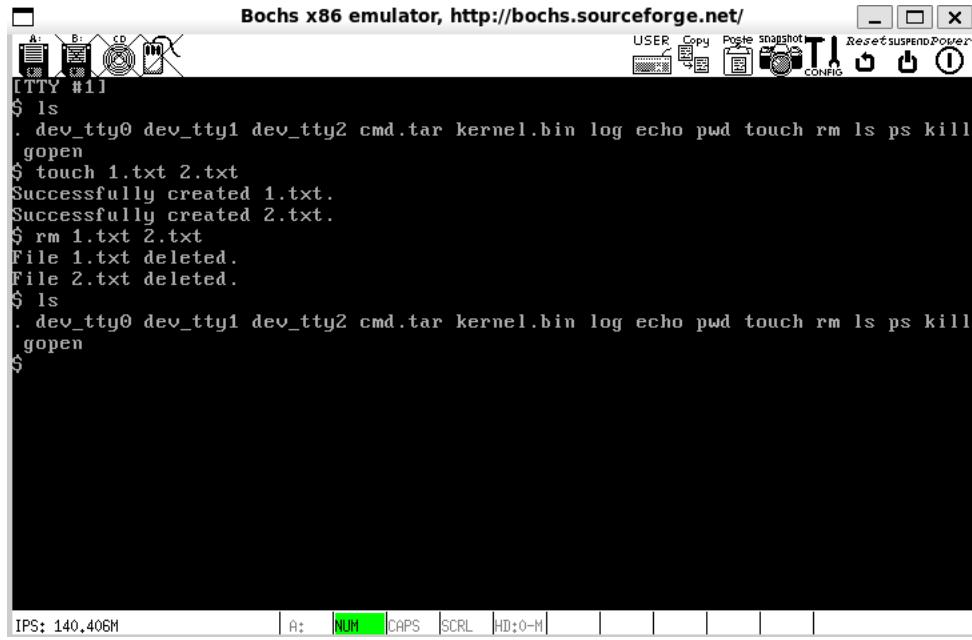


Figure 1: rm 指令

我先使用 ls 指令，此时没有新文件创建，后使用 touch 1.txt 2.txt 创建两个文件，随后使用 rm 1.txt 2.txt 提示已经完成删除，最终再次 ls，1.txt 和 2.txt 已经不存在。

3.2.4 并发运行任务

并发运行任务我们类似 Linux 系统中的 `&&` 指令，具体形式为 `command1 && command2 && command3`。Linux 这里并不是并发的，Linux 的操作逻辑为条件执行，即只有前一个命令成功执行（返回值为 0）时，才会执行后一个命令。显然这种逻辑更类似于串行的。我们这里要有所区分，任务是并发的，我们就不应该条件执行，换言之，`command2` 或者 `command3` 的操作并不会因为 `command1` 是否成功执行而受到影响。同时应该有并发的体现，即 `command1` 执行过程中，可能能够看到 `command2` 或者 `command3` 操作的回显内容，这样才能说明是并发的。

为了区分 Linux，指令我们的指令并不是 `&&` 而是 `&`，该部分代码修改于 `kernel/main.c` 中，实现了那么多 shell 指令，终于还是对 `shabby_shell` 下手了：

```

1 void shabby_shell(const char *tty_name)
2 {
3     int fd_stdin = open(tty_name, O_RDWR);
4     assert(fd_stdin == 0);
5     int fd_stdout = open(tty_name, O_RDWR);
6     assert(fd_stdout == 1);
7
8     char rdbuf[128];
9
10    while (1)
11    {
12        write(1, "$ ", 2);
13        int r = read(0, rdbuf, 70);
14        rdbuf[r] = 0;
15    }

```

```

16     int argc = 0;
17     char *argv[PROC_ORIGIN_STACK];
18     char *p = rdbuf;
19     char *s;
20     int word = 0;
21     char ch;
22     do
23     {
24         ch = *p;
25         if (*p != ' ' && *p != 0 && !word)
26         {
27             s = p;
28             word = 1;
29         }
30         if ((*p == ' ' || *p == 0) && word)
31         {
32             word = 0;
33             argv[argc++] = s;
34             *p = 0;
35         }
36         p++;
37     } while (ch);
38     argv[argc] = 0;
39
40     char *child_argv[10]; // 存储分隔后的单个命令
41     int index = 0, cnt = 0;
42
43     for (int i = 0; i < argc + 1; i++)
44     {
45         if (i == argc || strcmp(argv[i], "&") == 0)
46         {
47             child_argv[index] = 0; // 命令结束
48             if (index > 0) // 如果存在有效命令
49             {
50                 int fd = open(child_argv[0], O_RDWR);
51                 if (fd == -1)
52                 {
53                     // 命令无效，直接输出错误信息
54                     write(1, "Command
55                         not found: ", 19);
56                     // 拼接完整命令
57                     for (int j = 0; j < index; j++)
58                     {
59                         write(1, child_argv[j],
60                               strlen(child_argv[j]));
61                         write(1, " ", 1);
62                     }
63                     write(1, "\n", 1);
64                 }
65             else
66             {

```

```

67         close(fd);
68         int pid = fork();
69         // milli_delay(1000);
70         if (pid == 0)
71         {
72             // 子进程: 执行命令
73             execv(child_argv[0],
74                   child_argv);
75             panic("execv failed");
76             // 如果 execv 失败
77             exit(1);
78         }
79         else if (pid > 0)
80         {
81             // 父进程:
82             // 记录子进程 PID
83             cnt++;
84         }
85         else
86         {
87             panic("fork failed");
88         }
89     }
90     index = 0; // 重置索引, 开始解析下一个命令
91 }
92 else
93 {
94     // 收集当前命令参数
95     child_argv[index++] = argv[i];
96 }
97 for (int i = 0; i < cnt; i++)
98 {
99     int status;
100    wait(&status); // 等待子进程
101 }
102
103 }
104
105 close(1);
106 close(0);
107 }

```

代码中除去原来的内容外，额外添加了对于 & 符号的解析，如果遇到了 & 符号，在检查命令是否有效后，讲通过 fork 的方式，调用一个子进程来执行指令。也就是说对于 command1 & command2 & command3 这样的指令，我们的 shell 会先 fork 一个子进程执行 command1，无论 command1 是否已经返回，同时 fork 一个新的子进程用于执行 command2，以此类推。

好了，接下来让我们看看实现的效果吧：

```

Bochs x86 emulator, http://bochs.sourceforge.net/
[TTY #1]
$ touch 1.txt 2.txt & ps & kill -10 & rm 1.txt
Successfully created 1.txt.
PID NAME FLAGS
0 TTY Successfully created 2.txt.
RECEIVING
1 SYS Unknown
2 HD RECEIVING
3 FS RECEIVING
4 MM Unknown
5 INIT Unknown
6 TestA Process 10 has been killed.
Unknown
7 TestB Unknown
8 TestC Unknown
File 1.txt deleted.
9 INIT_9 SENDING
12 ps RECEIVING
14 rm Unknown
$ -

```

IPS: 163,459M | A: NUM CAPS SCRL HD:0-M | | | | | |

Figure 2: 并发运行任务

这真是一个非常好的例子，我们运行了指令“touch 1.txt 2.txt & ps & kill -10 & rm 1.txt”。仔细看看输出，真是惊讶，在创建完 1.txt 后，ps 指令就已经开始打印，正在 ps 开始打印时出现了创建完 2.txt 的提示，正在 ps 继续打印时，kill 突然提示 10 号进程已经被删除了，果然，再往下 ps，没有列出 10 号进程，此时的 rm 已经执行完毕将 1.txt 删除，最终 rm 以 pid 14 出现在了 ps 的打印结果中。

输出的结果真是复杂，完全可以说并行运行任务这个模块我们完成的很好，充分体现了“并发”性，就连 ps 的结果也能展示，我们的指令执行时彼此影响了对方。

当然，这里我需要做出解释，可能会提出问题，为什么 ps 只列出了 ps 和 rm 进程，但是没有列出 touch 和 kill 指令呢。这里需要对 ps 指令做出补充，Orange'S 系统的代码逻辑是，如果一个进程的 p_flags 如果比较为了 Free_SLOT，并不会直接将 proc_table 中有关它的部分，比如 name、pid 等做出修改。它做的是运行下一个进程直接的 pid 从该进程的 pid 开始编号，同时覆盖它原有的资源。如果没有进程进来，进程表的 proc_name 是还存在的。结合打印结果，当打印到 pid=8 时，touch 和 kill 都已经结束了，而此时 ps 还没有遍历到他们的位置，当 pid=12 时，ps 还没有结束，因此打印 ps 是合理的，当遍历到最后一个进程 rm 时，rm 虽然已经结束，但正如前文所述，它还是存储在 proc_table 中，自然被打印到了 shell。

好了，言归正传，我们的 shell 拥有了并行任务的能力，但我觉得这样的并行并不是件好事，《Operating Systems: Three Easy Pieces》这本书大概将 os 分成了并行、虚拟化、持久化三部分内容，其中并行是最复杂的一部分。并行会带来许多的危害，引用别的大佬的话概括来说就是，并行导致的 bug 难以定位，不好复现，并且带有几分玄学滋味。我们的 Orange'S 里对于并行的保护很少，比如常见的原子化操作、锁管理等基本没有涉及，我十分的害怕并行导致问题的出现。事实上，这种情况真的出现了，没有任何保护的系统真的不堪一击，我们来看看下面的指令：

我先通过 ps 查看到了 ps 将会在 pid 为 10 的位置执行，接下来我如果 ps & kill -10 会发生什么？按照 Linux 的操作逻辑，因为是条件执行，所以其实什么危险也不会有，kill 并不会关闭一个正在运行的进程，然而 Orange'S 里一切变了几分滋味：

```

Bochs x86 emulator, http://bochs.sourceforge.net/
PID NAME FLAGS
0 TTY RECEIVING
1 SYS Unknown
2 HD RECEIVING
3 FS RECEIVING
4 MM RECEIVING
5 INIT Unknown
6 TestA Unknown
7 TestB Unknown
8 TestC Unknown
9 INIT_9 WAITING
10 ps RECEIVING
$ ps & kill -10
PID NAME FLAGS
0 TTY RECEIVING
1 SYS Unknown
2 HD RECEIVING
3 FS RECEIVING
4 MM RECEIVING
5 INIT Unknown
6 TestA Unknown
7 TestB Unknown
8 TestC Unknown
Process 10 has been killed.

```

IPS: 132.764M A: NUM CAPS SCRL HD:0-M

并发问题

可以看到，ps 正在执行并且打印到了 pid8，但是随着 kill 成功显示 kill 了 10 号进程，ps 就再也不打印了，此时的程序已经崩溃，什么操作也不能执行，唯一的办法只有重启 bochs 了。因此虽然实现了并发能力，我不建议在系统中使用该功能，很有可能意外就会来临。

3.3 扩展系统日志能力

3.3.1 参数开关

系统日志我们设计了是否启用的功能，具体而言，将会通过一个总的开关来决定，是否记录系统日志，具体定义如下：

```
1 #define LOG_ENABLE 1
```

由于 Orange'S 作者提供的写日志 api disklog 函数在我们的 64 位机器上有很多意想不到的问题，经过大量时间的 debug，仍然不能得到很好的解决。因此我们选择了使用 write 函数封装一个写日志 api，即我们所使用的编写在 lib/write_log.c 中的 write_log 函数，如下：

```
1 PUBLIC void write_log(char *log_msg)
2 {
3     if(!LOG_ENABLE)
4     {
5         int log_fd = open("log", O_RDWR, 0);
6         if(log_fd == -1)
7         {
8             log_fd = open("log", O_CREAT | O_RDWR, 0);
9             assert(log_fd != -1);
10        }
11        lseek(log_fd, 0, SEEK_END);
12        write(log_fd, log_msg, strlen(log_msg));
13        close(log_fd);
```

```
14     }
15 }
```

通过 LOG_ENABLE 的定义，决定了是否会启用这个函数，因此也就决定了是否会写日志文件。当然，我们会在系统初始化结束之前，往系统里添加一个名为“log”的日志文件用于记录日志。在系统初始化结束之前的内容不应该记录到日志里，主要考虑到系统初始化时使用了大量的接口，如果写到 log 里，将会导致 log 文件十分的臃肿，同时受到 Orange ‘S 系统的限制，如果写太多内容到 log 中，log 本身并不能承载。另外，有个关键考量在于，当系统的初始化阶段，我们用于写 log 的许多 api，并没有被设置完全，此时如果直接调用，代码本身并不能识别到 api，还会导致程序的崩溃。因此在写日志前，我们额外添加了个变量名为 sys_init 当做写日志的锁，只有在 init 结束后才解锁，该变量定义在 lib/write.c 中：

```
1 PUBLIC int sys_init = 0;
```

声明在 include/sys/global.h 中，将会在 kernel/main.c 的 init 函数中被解锁：

```
1 void Init()
2 {
3
4     int fd_stdin = open("/dev_tty0", O_RDWR);
5     assert(fd_stdin == 0);
6     int fd_stdout = open("/dev_tty0", O_RDWR);
7     assert(fd_stdout == 1);
8
9     printf("Init() is running ...\n");
10    untar("/cmd.tar");
11
12    char *tty_list[] = {" /dev_tty1", " /dev_tty2"};
13
14    int i;
15    for (i = 0; i < sizeof(tty_list) / sizeof(tty_list[0]); i++)
16    {
17        int pid = fork();
18        if (pid != 0)
19        { /* parent process */
20            printf("[parent is running, child pid:%d]\n", pid);
21        }
22        else
23        { /* child process */
24            printf("[child is running, pid:%d]\n", getpid());
25            close(fd_stdin);
26            close(fd_stdout);
27
28            shabby_shell(tty_list[i]);
29            assert(0);
30        }
31    }
32    sys_init = 1; //这里解锁日志
33    device_log = 1;
```

```

34
35     while (1)
36     {
37         int s;
38         int child = wait(&s);
39         printf("child (%d) exited with status: %d.\n", child, s);
40     }
41
42     assert(0);
43 }
```

好了，我们已经有了个 log 文件，有了对于 log 文件的控制参数，让我们来看看系统 init 后，log 文件是否在系统中吧：



log 文件设置

有了 log 文件，接下来我们将开始写日志的征程

3.3.2 进程运行

进程主要是由 execv 来实现，下面来讲讲如何给 execv 记录日志。首先要明白，这里相当于给了我们所有的 shell 指令进行日志的记录，想来也合理，做为“管理员”一定很想知道“用户”在使用我们的 os 时使用了哪些指令操作了什么内容。

进程的状态无非就是“SENDING”、“RECEIVING”、“WAITING”、“HANGING”、“FREE_SLOT”，“UNKNOWN”，通过对比 p_flags 就可以知道进程的状态，再使用封装好的 write_log 即可将一个进程的运行过程记录到日志里，下面是 lib/write_log.c 中的内容：

```

1 PUBLIC void execv_log(const char *pathname, int src, int p_flags)
2 {
3     char log_msg[256] = {0};
```

```

4   struct time t;
5   MESSAGE t_msg;
6   t_msg.type = GET_RTC_TIME;
7   t_msg.BUF = &t;
8   char* flags;
9   send_recv(BOTH, TASK_SYS, &t_msg);
10  switch (p_flags)
11  {
12      case 2:
13          flags = "SENDING";
14          break;
15      case 4:
16          flags = "RECEIVING";
17          break;
18      case 8:
19          flags = "WAITING";
20          break;
21      case 16:
22          flags = "HANGING";
23          break;
24      case 32:
25          flags = "FREE_SLOT";
26          break;
27      default:
28          flags = "UNKNOWN";
29          break;
30  }
31  sprintf(log_msg, "<%d-%02d-%02d %02d:%02d:%02d>\n"
32          "[process] name = %s pid = %d flags = %s\n",
33          t.year, t.month, t.day, t.hour,
34          t.minute, t.second, pathname, src, flags);
35  write_log(log_msg);
36 }

```

在这里除了刚才讲述的东西，有两个 api 还是得提一下，会一直用到，之后不再赘述。关于 GET_RTC_TIME，是给我们提供好的系统调用，具体的功能就是通过和 bochs 的 RTC 交互以获取时间，我们来看看这个函数：

```

1 PRIVATE u32 get_rtc_time(struct time *t)
2 {
3
4     t->year = read_register(YEAR);
5     t->month = read_register(MONTH);
6     t->day = read_register(DAY);
7     t->hour = read_register(HOUR);
8     t->minute = read_register(MINUTE);
9     t->second = read_register(SECOND);
10
11    if ((read_register(CLK_STATUS) & 0x04) == 0)
12    {

```

```

13     /* Convert BCD to binary (default RTC mode) */
14     t->year = BCD_TO_DEC(t->year);
15     t->month = BCD_TO_DEC(t->month);
16     t->day = BCD_TO_DEC(t->day);
17     t->hour = BCD_TO_DEC(t->hour);
18     t->minute = BCD_TO_DEC(t->minute);
19     t->second = BCD_TO_DEC(t->second);
20 }
21
22 t->year += 2000;
23
24 return 0;
25 }
```

代码很简单，就是通过读寄存器获取现在的时间，精确到秒，稍微展开讲讲，就是通过读端口，将 bochs 的 rtc 时间传递进来。显然，这里涉及到一个问题，如果 bochs 的 rtc 频率很快，那么日志中获取的时间就会变得不准确，形象地说就是 bochs 内部的时间比外部时间更快。有没有解决办法呢，当然是有的，很好理解，我们 bochs 的 clk 时间是经过加速的，只要我通过 bochs 将时间调慢，即可，将下面这行加入到 bochsrc 中：

```
1 clock: sync=realtime,rtc_sync=1,time0=local
```

这是获取到的时间就和现实时间同步了，可以说是分毫不差。当然这样做其实是把 bochs 强行降速实现的，启动 bochs 后明显的能够感觉到运行的速度比以前慢了很多很多，因此建议代码的编写和调试阶段不要调整 bochs 的 rtc。

除了获取时间外，刚才的代码中还涉及到了 sprintf 这个函数，简单来说 sprintf 函数可以将多个信息格式化为一个字符串。我们的日志文件内容往往是由多项内容拼凑而成的，使用这个函数能够帮助我们统一写入规范，后续会多次使用。

讲述完如何 execv_log 函数是如何封装后，我们来看看 execv_log 使用的地方。不难理解，execv_log 一定是发生在执行了 execv 函数后，所以我只需要将该函数写入 execv 相关函数中，就起到了类似监听的效果。思前想后，最后 execv_log 被写入到了 do_exec 函数中，在这里，刚好有个 proc_table，并且提供了正在执行的进程的下标 src，可以很方便的帮助我们进行函数传参，代码如下：

```

1 PUBLIC int do_exec()
2 {
3     /* get parameters from the message */
4     int name_len = mm_msg.NAME_LEN; /* length of filename */
5     int src = mm_msg.source;        /* caller proc nr. */
6     assert(name_len < MAX_PATH);
7
8     char pathname[MAX_PATH];
9     phys_copy((void *)va2la(TASK_MM, pathname),
10            (void *)va2la(src, mm_msg.PATHNAME),
11            name_len);
12    pathname[name_len] = 0; /* terminate the string */
13
14    /* get the file size */
15    struct stat s;
```

```

16     int ret = stat(pathname, &s);
17     if (ret != 0)
18     {
19         printf("{MM}
MM::do_exec()::stat() returns error. %s", pathname);
20         return -1;
21     }
22
23
24     /* read the file */
25     int fd = open(pathname, O_RDWR);
26     if (fd == -1)
27         return -1;
28     assert(s.st_size < MMBUF_SIZE);
29     read(fd, mmbuf, s.st_size);
30     close(fd);
31
32     /* overwrite the current proc image with the new one */
33     Elf32_Ehdr *elf_hdr = (Elf32_Ehdr *) (mmbuf);
34     int i;
35     for (i = 0; i < elf_hdr->e_phnum; i++)
36     {
37         Elf32_Phdr *prog_hdr = (Elf32_Phdr *)
38             (mmbuf + elf_hdr->e_phoff +
39             (i * elf_hdr->e_phentsize));
40         if (prog_hdr->p_type == PT_LOAD)
41         {
42             assert(prog_hdr->p_vaddr + prog_hdr->p_memsz <
43                   PROC_IMAGE_SIZE_DEFAULT);
44             phys_copy((void *)va2la
45             (src, (void *)prog_hdr->p_vaddr),
46             (void *)va2la(TASK_MM,
47             mmbuf + prog_hdr->p_offset),
48             prog_hdr->p_filesz);
49         }
50     }
51
52     /* setup the arg stack */
53     int orig_stack_len = mm_msg.BUF_LEN;
54     char stackcopy[PROC_ORIGIN_STACK];
55     phys_copy((void *)va2la(TASK_MM, stackcopy),
56               (void *)va2la(src, mm_msg.BUF),
57               orig_stack_len);
58
59     u8 *orig_stack = (u8 *) (PROC_IMAGE_SIZE_DEFAULT - PROC_ORIGIN_STACK);
60
61     int delta = (int)orig_stack - (int)mm_msg.BUF;
62
63     int argc = 0;
64     if (orig_stack_len)
65     { /* has args */
66         char **q = (char **)stackcopy;

```

```

67     for ( ; *q != 0; q++, argc++)
68         *q += delta;
69     }
70
71     phys_copy((void *)va2la(src, orig_stack),
72                (void *)va2la(TASK_MM, stackcopy),
73                orig_stack_len);
74
75     proc_table[src].regs.ecx = argc; /* argc */
76     proc_table[src].regs.eax = (u32)orig_stack; /* argv */
77
78     /* setup eip & esp */
79     proc_table[src].regs.eip = elf_hdr->e_entry; /* @see _start.asm */
80     proc_table[src].regs.esp =
81     PROC_IMAGE_SIZE_DEFAULT - PROC_ORIGIN_STACK;
82
83     strcpy(proc_table[src].name, pathname);
84     execv_log(pathname, src, proc_table[src].p_flags);
85
86     return 0;
87 }
```

这里将把进程的名字，状态等信息传递到 execv_log 中，最后写入到日志中。让我们来看看效果：

```

<2024-12-11 20:30:34> File created: touch
<2024-12-11 20:30:41> File created: rm
<2024-12-11 20:30:49> File created: ls
<2024-12-11 20:30:56> File created: ps
<2024-12-11 20:31:03> File created: kill
<2024-12-11 20:31:11> File created: gopen
<2024-12-11 20:31:16> fork: parent name: INIT pid: 5 is running, child pid:9
<2024-12-11 20:31:17> fork: parent name: INIT pid: 5 is running, child pid:10
<2024-12-11 20:32:27> visited the Device: device:TTY sub_device:2
<2024-12-11 20:32:27> fork: parent name: INIT 9 pid: 9 is running, child pid:11
<2024-12-11 20:32:28> [process] name = ls pid = 11 flags = RECEIVING
<2024-12-11 20:32:28> wait: parent name: INIT_9 pid: 9 waited for child pid 11,
status: 0
<2024-12-11 20:32:56> visited the Device: device:Hard disk sub_device:32
<2024-12-11 20:32:56> fork: parent name: INIT 9 pid: 9 is running, child pid:11
<2024-12-11 20:32:56> [process] name = ps pid = 11 flags = RECEIVING
<2024-12-11 20:32:56> wait: parent name: INIT_9 pid: 9 waited for child pid 11,
status: 0

```

我在 shell 中先后执行了 ls 和 ps 指令，日志中如实的展现了这一流程，记录包括时间、进程名字、pid 号、状态这些相关信息。

3.3.3 文件访问

对于文件的操作，主要是由三个系统调用层级的 api 实现，分别为：open、unlink、write。针对这三个函数的日志实现较为复杂。可能会涉及到函数调用的死锁情形，也会涉及到消息通信之前的未知 bug。而这两个问题，也是贯穿了文件访问、系统调用、设备访问的 log 实现。这里将围绕如何避免这些问题的同时实现 log 记录。

先来看看 open 记录，对于 open，我们对原有的 open 函数进行了一定程度的修改，具体实现在 lib/open.c 中：

```

1 PUBLIC int _open(const char *pathname, int flags, int log)
2 {
3     MESSAGE msg;
4     msg.type = OPEN;
5     msg.PATHNAME = (void *)pathname;
6     msg.FLAGS = flags;
7     msg.NAME_LEN = strlen(pathname);
8     char buf[128] = {0};
9     msg.BUF = buf;
10    send_recv(BOTH, TASK_FS, &msg);
11    assert(msg.type == SYSCALL_RET);
12    if (log && flags & 1)
13    {
14        char log_msg[256] = {0};
15        struct time t;
16        MESSAGE t_msg;
17        t_msg.type = GET_RTC_TIME;
18        t_msg.BUF = &t;
19        send_recv(BOTH, TASK_SYS, &t_msg);
20
21        int logbufpos = sprintf(log_msg,
22            "<%d-%02d-%02d %02d:%02d:%02d>",
23            t.year, t.month, t.day, t.hour, t.minute, t.second);
24        if (~msg.FD)
25        {
26            logbufpos += sprintf(
27                log_msg + logbufpos, "
28                File created: %s\n", msg.PATHNAME);
29        }
30        else
31        {
32            logbufpos += sprintf(
33                log_msg + logbufpos, "
34                Failed to create file: %s\n", msg.PATHNAME);
35        }
36        write_log(log_msg);
37    }
38    return msg.FD;
39 }

```

该 `_open` 函数替换了原来的 `open` 函数，这里算是个小伎俩，使得所有使用了 `open` 接口的都会转到使用 `_open`：

```

1 PUBLIC int _open(const char *pathname, int flags, int log);
2 #define open(pathname, flags, ...) _open(pathname, flags, (0xff, ##__VA_ARGS__))

```

代码中使用了 `flag & 1` 是为了确定这次使用 `open` 接口，是否是为了创建文件，如果是为了创建文件再写入日志。考虑一种情形，当用户查看 `log` 时，使用了 `gopen` 打开 `log`，`gopen` 会使用到 `open` 函数，此时如果不进行上锁，那么就会出现，因为 `open` 了 `log`，所以要写入

log，这样做意义不大，因此直接将这种情形避免是最好的。这也是为了保护我们 log 文件那点本身不大的存储空间。

unlink 函数主要用来删除文件。unlink 写入日志的过程在 lib/unlink.c 中如下：

```
1 PUBLIC int unlink(const char *pathname)
2 {
3     // Check if the filename ends with ".log"
4     MESSAGE msg;
5     if (strcmp(pathname, "log") == 0)
6     {
7         msg.RETVAL = -1;
8     }
9     else
10    {
11         msg.type = UNLINK;
12         msg.PATHNAME = (void *) pathname;
13         msg.NAME_LEN = strlen(pathname);
14         send_recv(BOTH, TASK_FS, &msg);
15    }
16
17    struct time t;
18    MESSAGE t_msg;
19    t_msg.type = GET_RTC_TIME;
20    t_msg.BUF = &t;
21    send_recv(BOTH, TASK_SYS, &t_msg);
22    char log_msg[256] = {0};
23    int logbufpos = sprintf(log_msg,
24        "<%d-%02d-%02d %02d:%02d:%02d>",
25        t.year, t.month, t.day, t.hour,
26        t.minute, t.second);
27    if (~msg.RETVAL)
28        sprintf(log_msg + logbufpos,
29            " File deleted: %s\n", pathname);
30    else
31        sprintf(log_msg + logbufpos,
32            " Failed to delete file: %s\n", pathname);
33    write_log(log_msg); // Write to log
34
35    return msg.RETVAL;
36 }
```

这里通过判断 msg 的返回值的方式来确定文件删除操作是否成功，并且最终如实的记录到日志文件中。

关于 write，主要用于在文件中写入内容，是极为频繁的接口调用，我们在 write 接口内部做出了修改，使得所有通过 write 的请求都被记录：

```
1 PUBLIC int write(int fd, const void *buf, int count)
2 {
3     MESSAGE msg;
```

```

5     msg.type = WRITE;
6     msg.FD = fd;
7     msg.BUF = (void *)buf;
8     msg.CNT = count;
9     char filename[128];
10    msg.PATHNAME = filename;
11    read_write_flag = 1;
12    send_recv(BOTH, TASK_FS, &msg);
13
14    // 是否记录write日志
15    if (sys_init && msg.FLAGS)
16    {
17        struct time t;
18        MESSAGE t_msg;
19        t_msg.type = GET_RTC_TIME;
20        t_msg.BUF = &t;
21        send_recv(BOTH, TASK_SYS, &t_msg);
22
23        // 构造日志信息
24        char log_msg[256] = {0};
25        int logbufpos = sprintf(log_msg,
26            "<%d-%02d-%02d %02d:%02d:%02d>",
27            t.year, t.month, t.day, t.hour, t.minute, t.second);
28        sprintf(log_msg + logbufpos,
29            " Written to file: %s, Bytes: %d\n",
30            filename, count);
31        write_log(log_msg); // 调用 `write_log` 函数
32    }
33    return msg.CNT;
34 }

```

这里有个细节需要讲述，即 sys_init，正如上文所言，为了避免系统初始化时的 write 调用被记录到日志中，我们使用了这个变量用来当锁。同时为了确保这次操作是需要写入日志的，我们使用 msg.FLAGS 用于标记。比如 write 这个操作本身是需要写入 log 的，写入 log 又要使用 write 操作，如果不做任何保护的话，那会导致无尽的死锁问题，因此这里确保了 write_log 这个操作的 FLAGS 不会是正值即可避免。

好了，让我们在创建文件，修改文件，删除文件 (touch, fopen, rm) 之后打开 log 看看里面都记录了些什么：

```

<2024-12-11 21:27:49> File created: bufferOverflow
<2024-12-11 21:27:52> fork: parent name: INIT pid: 5 is running, child pid:9
<2024-12-11 21:27:53> fork: parent name: INIT pid: 5 is running, child pid:10
<2024-12-11 21:27:53> visited the Device: device:TTY sub_device:1
<2024-12-11 21:30:12> visited the Device: device:TTY sub_device:2
<2024-12-11 21:30:12> fork: parent name: INIT_9 pid: 9 is running, child pid:11
<2024-12-11 21:30:13> [process] name = touch pid = 11 flags = RECEIVING
<2024-12-11 21:30:17> File created: 1.txt
<2024-12-11 21:30:17> wait: parent name: INIT_9 pid: 9 waited for child pid 11
<2024-12-11 21:31:35> visited the Device: device:Hard disk sub_device:32
<2024-12-11 21:31:35> fork: parent name: INIT_9 pid: 9 is running, child pid:11
<2024-12-11 21:31:35> [process] name = gopen pid = 11 flags = RECEIVING
<2024-12-11 21:34:36> Written to file: 1.txt, Bytes: 13
<2024-12-11 21:34:36> wait: parent name: INIT_9 pid: 9 waited for child pid 11
<2024-12-11 21:37:14> fork: parent name: INIT_9 pid: 9 is running, child pid:11
<2024-12-11 21:37:14> visited the Device: device:Hard disk sub_device:32
<2024-12-11 21:37:15> [process] name = rm pid = 11 flags = RECEIVING
<2024-12-11 21:37:15> File deleted: 1.txt
<2024-12-11 21:37:16> wait: parent name: INIT_9 pid: 9 waited for child pid 11
<2024-12-11 21:38:44> fork: parent name: INIT_9 pid: 9 is running, child pid:11
<2024-12-11 21:38:44> visited the Device: device:Hard disk sub_device:32
<2024-12-11 21:38:45> [process] name = gopen pid = 11 flags = RECEIVING

```

文件访问日志

可以看到这里，记录了，我们创建了文件 1.txt，往其中写入了 13bytes 的内容，最后删除了该文件。

3.3.4 系统调用

系统调用部分这一节主要实现 fork 和 wait，其实有很多系统调用，但很多的日志记录在前面实现了。

fork 主要用于在系统调用进程时执行，即从父进程 fork 一个子进程，用子进程来执行任务。有关 fork 部分的代码主要在 lib/fork.c:

```

1 PUBLIC int fork()
2 {
3     MESSAGE msg;
4     msg.type = FORK;
5     char buf[128];
6     msg.BUF = buf;
7     send_recv(BOTH, TASK_MM, &msg);
8     assert(msg.type == SYSCALL_RET);
9     assert(msg.RETVAL == 0);
10
11    struct time t;
12    MESSAGE t_msg;
13    t_msg.type = GET_RTC_TIME;
14    t_msg.BUF = &t;
15    send_recv(BOTH, TASK_SYS, &t_msg);
16
17    if (msg.PID)
18    {
19        char log_msg[256] = {0};
20        int logbufpos = sprintf(log_msg,
21            "<%d-%02d-%02d %02d:%02d:%02d>",
22            t.year, t.month, t.day, t.hour, t.minute, t.second);
23        // 记录父进程的名称和子进程的名称
24        sprintf(log_msg + logbufpos,

```

```

25     " fork: parent %s is running, child pid:%d\n",
26         msg.BUF, msg.PID);
27     write_log(log_msg); // 调用 `write_log` 函数
28 }
29 return msg.PID;
30 }
```

这里会直接对通过了 fork 函数的过程进行记录，只要 msg.PID 是 > 0 (即存在) 就记录下父进程和子进程的名称，然后通过 write_log 写入到日志中。在这里我们将详细聊聊如何通过消息通信来传递参数，让我们 msg.BUF=buf 这一行。没错，先申请了一个 buf 空间，然后让 msg 结构体的 BUF 指向这片空间，这样在消息传递时就可以通过 msg 来修改 buf 中需要编写的内容。让我们来看看在 do_fork 这段代码中，是如何修改 buf 的内容的 (mm/forkexit.c):

```

1 PUBLIC int do_fork()
{
3 ...
4     char *ret = va2la(mm_msg.source, mm_msg.BUF);
5
6     sprintf(ret,
7         "name: %s pid: %d", proc_table[pid].name, pid);
8
9 }
```

稍微解释下代码，使用 ret 指针指向原来的 BUF 对应的空间，即在 fork() 中的 buf，这里需要注意，因为涉及到地址切换，需要将虚拟地址和逻辑地址之间进行转换，如果不进行转换，指针指向的地址空间将会有问题。在指向正确的空间后，使用 sprintf，将需要在这里获取的信息及时的存入到 buf 中。这样我们巧妙地运用了 IPC 机制完成了数据传递。

接下来看看 fork 实现的效果：

```

2024-12-11 22:19:04> visited the Device: device:TTY sub_device:0
2024-12-11 22:19:08> visited the Device: device:TTY sub_device:0
2024-12-11 22:19:12> File created: kernel.bin
2024-12-11 22:19:23> File created: echo
2024-12-11 22:19:29> File created: pwd
2024-12-11 22:19:36> File created: touch
2024-12-11 22:19:42> File created: rm
2024-12-11 22:19:49> File created: ls
2024-12-11 22:19:56> File created: ps
2024-12-11 22:20:03> File created: kill
2024-12-11 22:20:10> File created: fopen
2024-12-11 22:20:18> File created: bufferOverflow
2024-12-11 22:20:22> fork: parent name: INIT pid: 5 is running, child pid:9
2024-12-11 22:20:23> fork: parent name: INIT pid: 5 is running, child pid:10
2024-12-11 22:20:23> visited the Device: device:TTY sub_device:1
2024-12-11 22:22:53> visited the Device: device:TTY sub_device:2
2024-12-11 22:22:53> fork: parent name: INIT_9 pid: 9 is running, child pid:11
2024-12-11 22:22:54> [process] name = ls pid = 11 flags = RECEIVING
2024-12-11 22:22:54> wait: parent name: INIT_9 pid: 9 waited for child pid 11
2024-12-11 22:27:44> fork: parent name: INIT_9 pid: 9 is running, child pid:11
2024-12-11 22:27:45> visited the device: device:hard disk sub_device:32
2024-12-11 22:27:45> [process] name = fopen pid = 11 flags = RECEIVING
[INFO] You have no access to modify this file.
```

Figure 3: fork

日志记录了如果通过 INIT 进程创建了 INIT_9 和 INIT_10 进程，又记录了我在输入 ls 指令后，通过 INIT_9 进程 fork 了子进程 (pid 为 11) 用于执行 ls 操作。

wait 主要用于父进程等待子进程的结束时使用，相关修改在 lib/wait.c 中：

```
1 PUBLIC int wait(int *status)
2 {
3     MESSAGE msg;
4     msg.type = WAIT;
5     char buf[128];
6     msg.BUF = buf;
7     send_recv(BOTH, TASK_MM, &msg);
8
9     *status = msg.STATUS;
10
11    // 获取当前时间
12    struct time t;
13    MESSAGE t_msg;
14    t_msg.type = GET_RTC_TIME;
15    t_msg.BUF = &t;
16    send_recv(BOTH, TASK_SYS, &t_msg);
17
18    // 构造日志信息
19    char log_msg[256] = {0};
20    int logbufpos = sprintf(log_msg,
21                           "<%d-%02d-%02d %02d:%02d:%02d> ",
22                           t.year, t.month, t.day,
23                           t.hour, t.minute, t.second);
24
25    // 记录等待的子进程信息
26    if (msg.PID != NO_TASK)
27    {
28        sprintf(log_msg + logbufpos,
29                "wait: parent %s waited for child pid %d\n",
30                buf, msg.PID);
31    }
32    else
33    {
34        sprintf(log_msg + logbufpos,
35                "wait: parent %s, no child process to wait for.\n",
36                buf);
37    }
38
39    // 调用 write_log 函数，写入日志
40    write_log(log_msg); // 写入日志
41
42    return (msg.PID == NO_TASK ? -1 : msg.PID);
43 }
```

在父进程执行 wait 等待子进程结束的过程中，将被日志记录下来，至于子进程的相关信息，仍然是通过 buf 进行消息传递来获取。这个外层封装的 wait 函数是不知道子进程信息的，只能通过 do_wait 来获取。

来看看 wait 的 log 记录效果：

```

<2024-12-11 22:19:04> visited the Device: device:TTY sub_device:0
<2024-12-11 22:19:08> visited the Device: device:TTY sub_device:0
<2024-12-11 22:19:12> File created: kernel.bin
<2024-12-11 22:19:23> File created: echo
<2024-12-11 22:19:29> File created: pwd
<2024-12-11 22:19:36> File created: touch
<2024-12-11 22:19:42> File created: rm
<2024-12-11 22:19:49> File created: ls
<2024-12-11 22:19:56> File created: ps
<2024-12-11 22:20:03> File created: kill
<2024-12-11 22:20:10> File created: gopen
<2024-12-11 22:20:18> File created: bufferOverflow
<2024-12-11 22:20:22> fork: parent name: INIT pid: 5 is running, child pid:9
<2024-12-11 22:20:23> fork: parent name: INIT pid: 5 is running, child pid:10
<2024-12-11 22:20:23> visited the Device: device:TTY sub_device:1
<2024-12-11 22:22:53> visited the Device: device:TTY sub_device:2
<2024-12-11 22:22:53> fork: parent name: INIT_9 pid: 9 is running, child pid:11
<2024-12-11 22:22:54> [process] name = ls pid = 11 flags = RECEIVING
<2024-12-11 22:22:54> wait: parent name: INIT_9 pid: 9 waited for child pid 11
<2024-12-11 22:27:44> fork: parent name: INIT_9 pid: 9 is running, child pid:11
<2024-12-11 22:27:45> visited the Device: device:Hard disk sub_device:32
<2024-12-11 22:27:45> [process] name = open pid = 11 flags = RECEIVING

[INFO] You have no access to modify this file.
$_

```

wait

这里很好的体现了一个 ls 指令的一生。首先父进程 INIT_9 通过 fork 创建了 pid 号为 11 的子进程 ls，然后 ls 进入执行，ls 执行过程中，父进程 wait，最终 ls 执行完毕，父进程结束 wait。

3.3.5 设备访问

有关设备有哪些，我们需要查看相关的代码，即 dd_map 是如何定义的：

```

1 struct dev_drv_map dd_map[] = {
2     /* driver nr.      major device nr.
3      -----      -----
4      {INVALID_DRIVER}, /*< 0 : Unused */
5      {INVALID_DRIVER}, /*< 1 : Reserved for floppy driver */
6      {INVALID_DRIVER}, /*< 2 : Reserved for cdrom driver */
7      {TASK_HD},        /*< 3 : Hard disk */
8      {TASK_TTY},       /*< 4 : TTY */
9      {INVALID_DRIVER} /*< 5 : Reserved for scsi disk driver */
10 };

```

不难发现，其实我们的主设备，除了 HD 也就只有 TTY 了，别的并没有投入到使用中，因此我们就记录这两种设备的访问记录。设备访问无非是比如创建、编写文件时访问了硬盘或者 TTY，因此最好的办法是在 write 和 open 函数这里进行处理。当然，有一个地方转发了所有的设备访问过程，即 kernel/hd.c 中的：

```

1 send_recv(RECEIVE, ANY, &msg);
2
3 int src = msg.source;
4
5 switch (msg.type)
6 {
7     case DEV_OPEN:

```

```

8     hd_open(msg.DEVICE);
9     break;
10
11 case DEV_CLOSE:
12     hd_close(msg.DEVICE);
13     break;
14
15 case DEV_READ:
16 case DEV_WRITE:
17     hd_rdwt(&msg);
18     break;
19
20 case DEV_IOCTL:
21     hd_ioctl(&msg);
22     break;
23
24 default:
25     dump_msg("HD driver::unknown msg", &msg);
26     spin("FS::main_loop (invalid msg.type)");
27     break;
28 }
29
30 send_recv(SEND, src, &msg);

```

但最终我们没有选择在这里进行日志记录。其实上面的进程、文件、系统调用的日志记录也有这样的问题，为什么不在这种带有 switch case 的地方进行日志的记录，在这里直接 write_log。经过我们的研究，在这里如果编写日志，那就会涉及到同时有两个通信在这里发生，比如，将 write_log 写在 send_recv 前面会导致 write_log 这个消息通信的返回值错误的传导给了下面的通信，最终触发断言错误。总之无论如何，在这里使用两个消息通信会出现意外中的 bug，想来是有关并发的错误，因此最终我们选择在更小的子模块中分片进行 log 记录。

对于 write，即往设备中修改内容，在 lib/write.c 的 write 函数中加入了如下代码：

```

1 if(device_log && sys_init && msg.FLAGS)
2 {
3     char log_msg[256] = {0};
4     struct time t;
5     MESSAGE t_msg;
6     t_msg.type = GET_RTC_TIME;
7     t_msg.BUF = &t;
8     send_recv(BOTH, TASK_SYS, &t_msg);
9     int logbufpos = sprintf(log_msg,
10         "<%d-%02d-%02d %02d:%02d:%02d>",
11         t.year, t.month, t.day, t.hour, t.minute, t.second);
12     logbufpos += sprintf(log_msg + logbufpos,
13         "visited the Device: %s\n", device_buf);
14     device_log = 0;
15     write_log(log_msg);
16     memset(device_buf, 0, sizeof(device_buf));
17 }

```

让我们来关注下如何往全局变量 device_buf 中存入有关设备的信息的，相关代码在 fs/read_write.c 的 do_rdwt 中：

```
1     int device_number = pin->i_dev;
2     if((MAJOR(device_number) <= 5) && sys_init && read_write_flag)
3     {
4         device_log = 1;
5         read_write_flag = 0; //防止回环调用
6         char *device = device_name(MAJOR(device_number));
7         sprintf(fs_msg.BUF,
8             "%s:%s %s:%d",
9             "device",device,
10            "sub_device",MINOR(device_number));
11         sprintf(device_buf,"%s",fs_msg.BUF);
12     }
13     return bytes_rw;
```

这里，通过 i_dev 获取设备号，这个设备号不能直接使用，通过 MAJOR 操作能够获得主设备号，MINOR 操作可以获得次设备号。经过设备号是否合法的检查后，将主次设备号写入到 device_buf 中，主设备号的下标和 name 的对应通过 device_name 函数，函数在 kernel/hd.c 中如下：

```
1 PUBLIC char* device_name(int number)
2 {
3     switch(number)
4     {
5         case 0:
6             return "Unused";
7         case 1:
8             return "Reserved for floppy driver";
9         case 2:
10            return "Reserved for cdrom driver";
11        case 3:
12            return "Hard disk";
13        case 4:
14            return "TTY";
15        case 5:
16            return "Reserved for scsi disk driver";
17    }
18 }
```

函数通过 dd_map 的映射来获得设备的名称。前面有使用 device_log 变量，该变量的主要作用在于可以防止回环问题，即通过 write 写有关设备访问时的内容时不要再写因此写日志，起到一个上锁的作用。

对于 open 函数，编写了日志记录代码在 lib/open.c 的 _open 函数中：

```

1   if(log && device_log)
2   {
3       char log_msg[256] = {0};
4       struct time t;
5       MESSAGE t_msg;
6       t_msg.type = GET_RTC_TIME;
7       t_msg.BUF = &t;
8       send_recv(BOTH, TASK_SYS, &t_msg);
9       int logbufpos = sprintf(log_msg,
10      "<%d-%02d-%02d %02d:%02d:%02d>",
11      t.year, t.month, t.day,
12      t.hour, t.minute, t.second);
13      logbufpos += sprintf(log_msg + logbufpos,
14      "visited the Device: %s\n", device_buf);
15      device_log = 0;
16      write_log(log_msg);
17  }

```

大体思路与前面呈现的一致，有关 device_buf 相关信息如何获取在 fs/open.c 的 do_open 函数中有所涉及：

```

1   int device_number = pin->i_start_sect;
2   if(MAJOR(device_number) <= 5)
3   {
4       device_log = 1;
5       char *device = device_name(MAJOR(device_number));
6       sprintf(fs_msg.BUF, "%s:%s %s:%d",
7           "device",
8           device, "sub_device", MINOR(device_number));
9       sprintf(device_buf, "%s", fs_msg.BUF);
10  }

```

整体没什么区别，接下来看看实现效果：

```
<2024-12-11 22:51:17> visited the Device: device:TTY sub_device:0
<2024-12-11 22:51:21> File created: kernel.bin
<2024-12-11 22:51:32> File created: echo
<2024-12-11 22:51:38> File created: pwd
<2024-12-11 22:51:45> File created: touch
<2024-12-11 22:51:51> File created: rm
<2024-12-11 22:51:58> File created: ls
<2024-12-11 22:52:05> File created: ps
<2024-12-11 22:52:12> File created: kill
<2024-12-11 22:52:19> File created: gopen
<2024-12-11 22:52:27> File created: bufferOverflow
<2024-12-11 22:52:31> fork: parent name: INIT pid: 5 is running, child pid:9
<2024-12-11 22:52:32> fork: parent name: INIT pid: 5 is running, child pid:10
[2024-12-11 22:52:32]> visited the Device: device:TTY sub_device:1
[2024-12-11 22:55:37]> visited the Device: device:TTY sub_device:2
<2024-12-11 22:55:38> [process] name = touch pid = 11 flags = RECEIVING
pid:11
<2024-12-11 22:55:42> File created: 1
<2024-12-11 22:55:42> wait: parent name: INIT_9 pid: 9 waited for child pid 11
<2024-12-11 22:57:23> visited the Device: device:Hard disk sub_device:32
<2024-12-11 22:57:23> fork: parent name: INIT_9 pid: 9 is running, child pid:11
<2024-12-11 22:57:24> [process] name = gopen pid = 11 flags = RECEIVING

[INFO] You have no access to modify this file.
$
```

设备访问

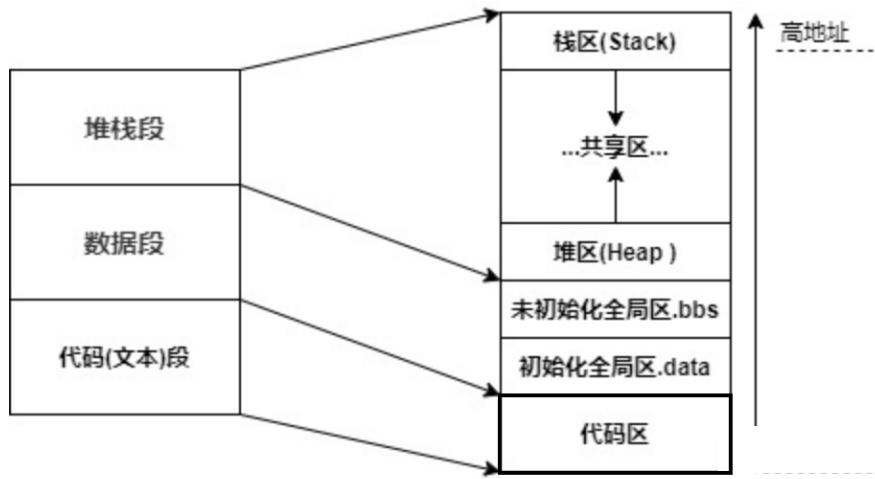
可以看到，最开始记录了访问 TTY，子设备分别为 1、2 的过程。在我使用 touch 创建一个文件后，显示了访问了 Hard disk 设备，子设备号为 32。

3.4 自我 OS 安全分析

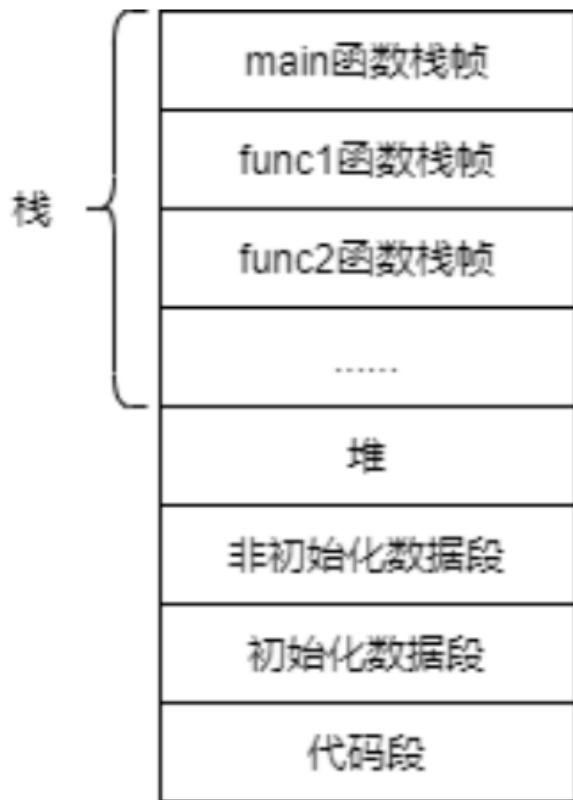
3.4.1 安全分析

本次使用的 Orange'S 代码框架，使用的语言版本低，gcc 编译器版本低，存在很多漏洞可以被利用。围绕如何具体利用漏洞我将不在本实验报告中进行深入浅出的讲述，我的组员编写了完整的攻击脚本后进行了测试，给出了攻击结果，具体可以查看组员的实验报告，我在乎只讲讲出现这些漏洞的原理。

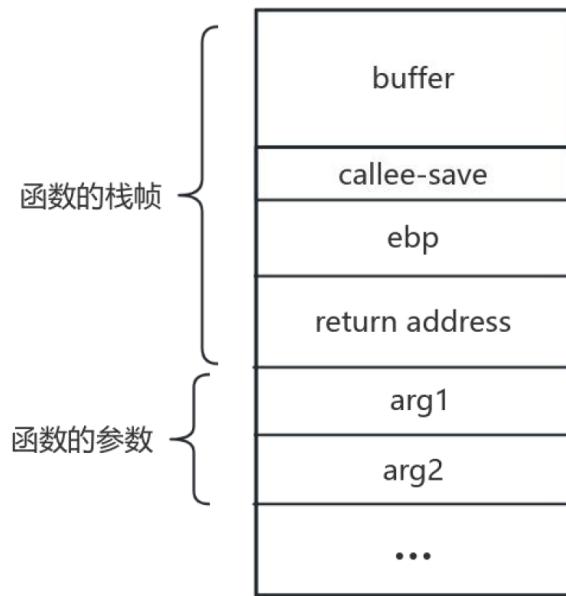
缓冲区溢出 缓冲区溢出是一个内存破坏类的漏洞，在 C 系语言中比较常见（rust 语言在安全性方面做得很好），主要还是源于 C 对缓冲区的边界检查不够“严厉”导致的。C 语言中，一个程序主要包括静态代码段和数据段，这些段被映射到操作系统创建的进程的虚拟地址空间中，以便执行程序。而仅有静态代码和数据段是不够的，进程在运行时还需要动态环境。通常，默认的动态存储环境是通过堆栈机制建立的。所有局部变量和按值传递的函数参数都会自动分配在堆栈上的内存空间，相邻的内存块分配给相同数据类型的内存区域被称为缓冲区。如下图所示：



需要注意的是，缓冲区溢出也分为栈溢出、堆溢出、BBS 溢出等类型。我们组在 OS 实验中着重分析了栈溢出，在软件安全课程中研究了堆溢出和 BBS 溢出。当程序被操作系统调入内存运行时，进程在内存的映射结果为：



进程的栈由栈帧构成，每个栈帧对应一个函数调用。函数将返回地址保存在了堆栈中，确没有对这样的重要数据进行任何保护。函数的逻辑为直接执行返回地址处的代码。倘若我构造一个足够长的 buf 放入到栈帧中，buf 超过了缓冲区的大小，一路“蔓延”到了返回地址中，那么就可以会胁迫函数执行一段未知的代码，即 shellcode。栈帧如下图所示：



栈缓冲区溢出主要有两种方案，包括：

- 溢出代码文件中函数的缓冲区，或者 C 语言内置函数或库函数的缓冲区，然后返回到恶意代码 shellcode 的位置
- 恶意代码 shellcode 的位置是栈中恶意代码地址，或者为一个恶意函数地址

这里我们主要采用了第一套方案，并且呈现了良好的实验结果。

格式化字符串漏洞 格式化字符串漏洞是内存泄漏类型的漏洞，主要分为了“任意写”和“任意读”两部分。

任意地址写主要来源于，printf 的%*n* 占位符，本用于和参数一一对应，但是 printf 并没有做参数个数和%*n* 占位符个数的限制。当参数小于占位符个数时，函数除了以此读取栈上数据满足前面的占位符需求时，在遇到参数不够时，仍会继续读取，直到满足占位符需求。因此，只要攻击者不按规则输入字符串，即可恶意利用漏洞。比如对于“ABCD %x”字符串，%*x* 会得到参数的地址，又会计算% 前的字符串长度，因此可以实现将% 前的字符串长度写入到栈帧中别的数据的位置，达到任意写的目的。

任意地址读来源于攻击者不填充占位符，函数仍然会从栈帧中不断读取数据，因此将会泄露处正常参数以外的站数据。因此恶意利用占位符可以达到读出任意地址的目的。

本组组员已经就利用格式化字符串漏洞编写了攻击脚本，在 Oragne'S 中查看到了攻击成功的效果，在此不再赘述。

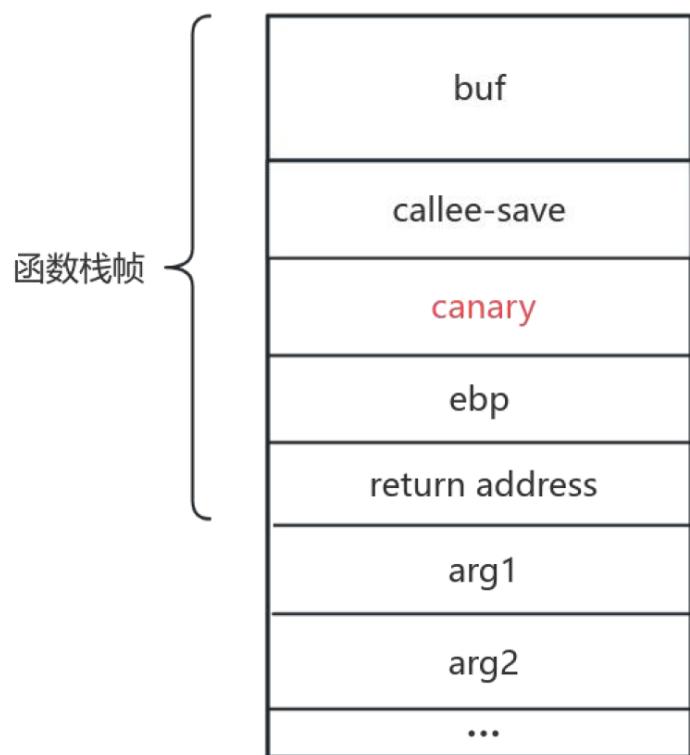
3.4.2 自我 OS 防护之抗攻击

canary 技术的应用 canary 是用于防御栈缓冲区溢出的一种实现成本低，效果好的方法，canary 意为金丝雀，来源于矿井工人为了检测是否发生瓦斯泄露，会在下矿时携带一只金丝雀，金丝雀的感知瓦斯能力远胜于人类，如果金丝雀在矿井中不再“歌唱”，那么此时可以认为发生了瓦斯泄露，下图为南京大学蒋炎岩老师的课件中的矿井工人与 canary：



矿井工人与 canary

当然，大可以放心，在我们的 canary 技术应用中，没有任何一只 canary 做出了牺牲，我们仅仅利用了一些数据。canary 将被用于探测，在每次函数调用时插入到函数的返回地址和局部变量之间，这样当发生溢出时，canary 一定会先于返回地址被覆盖。这样在进行函数返回时只要对 canary 是否不再“歌唱”进行检测即可明白是否被恶意攻击。在有 canary 时的堆栈结构如下：



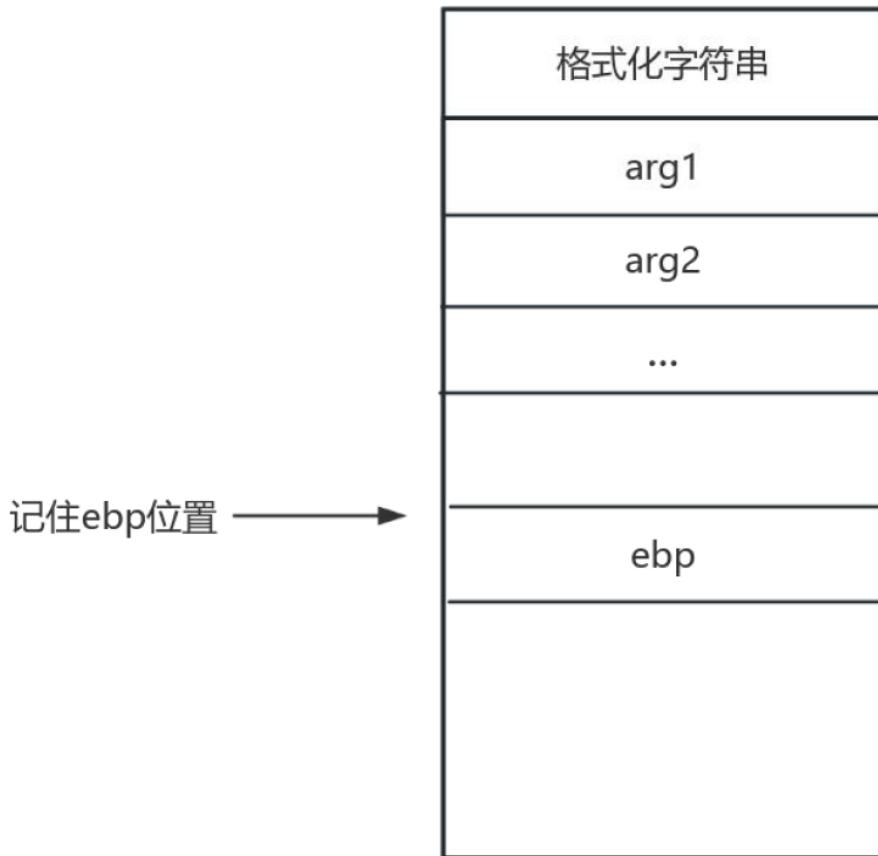
canary 有多种实现机制，比如依赖于 libc 的 arc4random 获取伪随机数的方案，或者 GS 安全编译方案。我们将使用 GS 方案实现 canary，具体流程为：

- 对于 oranges 的每个进程，我们找出它的 LDT 中的第一个 DWORD 作为 COokie 的种子
- 在进程发生函数调用时，栈帧会进行初始化，此时使用 esp 寄存器的值异或种子作为当前函数的 Cookie。这里的涉及十分精妙，当函数调用内部又存在函数调用时，将把新的 Cookie 和新的函数 esp 寄存器进行异或，以此类推。这样可以保证 Cookie 基本不同（esp 很难相同），同时在函数返回时只要再次异或就能回到上一级的 Cookie。
- 在执行完函数代码需要进行返回时，用 esp 还原 Cookie 的种子进行检查，因为 LDT 中的第一个 DWORD 基本不会修改，只要匹配不上，就大概率发生了栈溢出。

详细代码见组员实验报告。

内存边界检查和输入验证 针对格式化字符串漏洞，我们可以采用内存边界检查和增加输入验证模块的方式进行防御。

对于内存边界，在每次函数调用时，保存栈帧的基地址 ebp，printf 读取到的参数的地址不可以越过 ebp，这样就可以确保栈帧内读取的参数通常是合法的，即“法无授权不可为”。有了 ebp 的栈帧图示如下：



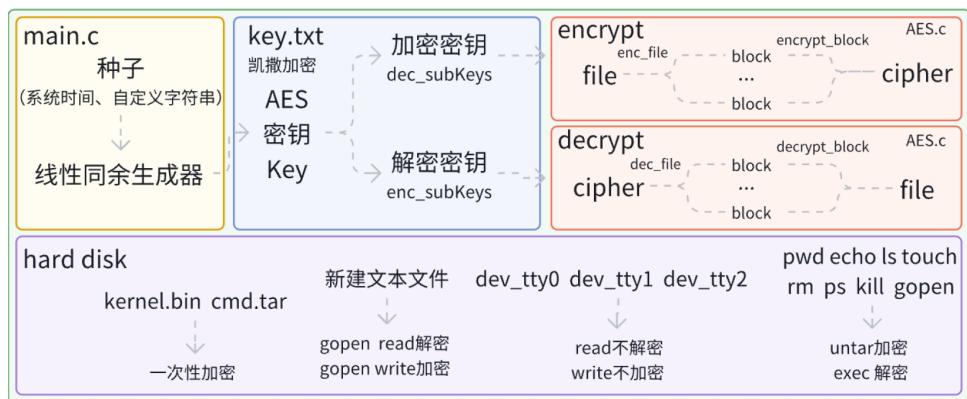
输入验证功能，即我们需要确保用户写入缓冲区的数据是合法的，比如是否构造了 "%n" 类型的疑似用于攻击的占位符。内存检查本身无法将栈帧外的内存读写全部几角，因此最好的办法是检测用户输入，当出现疑似构造的攻击字符时拒绝执行代码。我们组将内存边界检查的 printf 函数更新为了 safe_printf，格式化字符串解析函数 vsprintf 更新为 safe_vsprintf。

3.4.3 自我 OS 防护之抗泄露

这部分，我们组极具挑战精神，我们实现了两个可选的技术路线，A or B 我们当然选择 both。

磁盘文件加密 对于磁盘里的文件进行加密防护，主要目的是保护磁盘文件的安全性，防止敏感数据被未鉴权的使用者获取。具体的实现效果为实现文件读入内存时为明文，写回磁盘加密为密文。

我们采用了线性同余生成器来生成密钥，以此实现轻量级的密钥生成算法，并将密钥及其子密钥加密存储在磁盘中，设置权限保护密钥。加解密算法方面选择了 AES-128，AES 相比 DES 提高了安全性和效率，同时适合我们的 OS 系统。具体的思路可以由图片概括：



有关密钥算法如何实现与本次 os 代码联系并不直接，更多牵涉到信息安全的另一门必修课程即密码学，组员已经将完整的代码编写完毕同时呈现在了实验报告中，不再赘述。

考虑到不可能对密钥文件本身使用同样的加密算法进行加密，这样并不合理，因此我们采用了古典算法中的凯撒密码将密钥文件加密保护起来，避免密钥的泄露。

最终对于写磁盘的操作主要有三类：

- 硬盘启动之前用 dd 指令写入磁盘的 hdboot.bin 和 cmd.tar
- 初始化磁盘的时候需要将 inst.tar 压缩包解压并写入磁盘
- 调用 shell 指令对磁盘进行操作的时候需要写磁盘

对于引导过程，这部分是直接嵌入到磁盘中，因此不该对其进行加密，我们的加密主要针对压缩包和 shell 指令使用时产生的各种读写磁盘操作。

具体的磁盘加密的细节探讨、注意事项、代码实现查看组员周业营同学的实验报告。

访问控制模块 这部分主要是对于进程设置级别，用于指定进程访问特定的文件资源，非达到级别的进程不可以访问受保护文件资源。访问控制的目标是对提出的资源访问请求进行控制，进行鉴权，控制的三要素为：主体、客体、策略。本次我们采用自主访问控制机制，主题的拥有者可以灵活的设置访问权限，这样的策略比较灵活。

本实验中，我们将设置系统进程的用户为 root，用户进程的用户组为 administrator，每个进程的 group 设置为了该进程的 pid，即每个进程单独被分配给一个单独的角色。

设计的策略主要是，当进程和文件所在的用户和 group 是系统时可以进行访问，而 root 用户可以自由访问系统中的任意文件。

具体的访问控制模块的原理描述和设计实现查看组员程序同学的实验报告。

4 实验总结与致谢

本次实验我在和王浚杰同学的合作下完成了 shell 和日志的实现。这部分内容需要对整个代码框架有比较清晰的认知，开始的很长一段时间我们都把精力花在了阅读代码和课本上。对于很少接触带有那么多文件的项目的我们，开始编码的过程是极为痛苦的，NCC 的 C 栋 5 单元漫长的夜回荡着我们小组无尽的哀叹，2024 年的 12 月初我们是在熬夜写代码和调试中度过的。不得不说，Orange'S 代码有一些上古年代的宝物的感觉，有的地方不得不感慨其设计精妙，有的地方不得不抨击它的陈旧，可以说我们的苦与乐受到它的影响是极大的。

好在，我们是一个好的团队，我们一起撑了过来不是吗，当最后一个功能实现，代码跑通，测试没问题的时候，我们的心中真是相当的畅快，甚至颇有几分扬眉吐气的感觉，想到小半月来的奇妙合作写码经历真是人生的一段有趣体验。我们最终完成了所有需要完成的任务，代码不说多么的优雅，至少，它是能够运行的，它能够完成基本的工作，这足以令人欣慰。同时我们小组是相当充满探索精神的：我们努力给 Orange'S 用上更新，更现代的环境，我们使用了 wsl2 进行代码的编写，64 位的环境运行这 32 位的代码真是困难重重，有代码逻辑本身的问题，有 gcc 版本导致的问题等等，有的时候我们挖苦自己，为什么非要折磨自己呢；我们尝试过使用 qemu 运行程序，而此启发了我们如何使用 gdb 对于 bochs 运行的代码进行调试，我不敢说有多少人愿意像我们一样折腾，至少我们做了，gdb 的使用可以说是我们编写代码过程中最重要的一环。我们仔细阅读了项目代码，认真尝试使用了原本代码中提供的接口，对于能够使用的使用，能够改进的改进，不能使用的弃用重写等操作；我们针对 os 系统的安全性提出了很多见解，我们想办法验证了种种猜想；

回望本学期的 os 课程，到这里就要走向结局，整个小组一共完成了 10 次实验内容 +1 次大作业，每个人手上的 11 份实验报告浓缩了我们一学期的收获。os 实验可以说在大三这个阶段，除了让我更加深入的了解 os 运转原理外还充分提升了我的代码编写、调试能力。整个计算机世界无非就是一个大型的状态机，计算机是绝对理性的，它只会忠诚的执行一条接一条的指令。在 os 实验的过程中，这种唯物的思想勉励了我在遇到无法解决的问题时险些崩溃的心态，一步一步探索，计算机它并不可怕，我能够研究清楚其中的原理。

真心感谢整个小组的成员，整个小组不摆烂，应对问题积极，想起半夜几个人围在一台电脑面前观察各种意外的报错和打印结果的经历时，不由得感觉我的队友都给予了我强烈的正反馈。感谢我的 os 理论课授课老师杨敏老师，杨敏老师讲课给人如沐春风的感觉，是最早带领我“深入理解操作系统”的老师；感谢南京大学蒋炎岩老师和教材《Operating Systems: Three Easy Pieces》的作者，在此强烈推荐蒋炎岩老师开源在 B 站的给南大本科生的操作系统课程，是学习国外 cs 教育授课模式的先锋课程，而其教材以一种诙谐幽默的笔风行文又不乏对内容的深度挖掘令我印象深刻，书中的每一章开始都是从教授和学生的对话开始，让人很有阅读兴趣。感谢 Orange'S 的作者于渊老师，他所编写的 Orange'S 是一个很好的供新手入门的有趣玩具，他无形中手把手带领我用一个学期的时间实现了包括进程调度、shell 等内容的 os 系统。感谢严飞老师，严飞老师是我遇见过的极其负责任的一位好老师。严飞老师是我本次 os 实验的授课老师，严飞老师教学认真，并且乐意帮助学生解决问题，愿意陪同学们进行思考，愿意在自己的休息时间回答学生的各种奇怪问题（比较有趣的是严飞老师有次回复我的来自 QQ 的提问竟然是在开车时的空隙），可以说严飞老师是指导我对 Orange'S 进行深度探索的绝对核心。最后稍微感谢下自己，希望自己不停下学习的脚步。

这就是全部了，感谢你将本实验报告读到这里。

黄东威
2024 年 12 月 13 日 NCC 凌晨