



南開大學
Nankai University

计算机学院
计算机系统设计实验报告

PA3—穿越时空的旅程：异常控制流

姓名：杨馨仪

学号：2011440

专业：计算机科学与技术

2024 年 5 月 13 日

目录

1 概述	2
1.1 实验目的	2
1.2 实验内容	2
2 阶段一	2
2.1 操作系统——更方便的运行环境	2
2.1.1 代码：实现 Loader	2
2.2 穿越时空的旅程	3
2.2.1 实现中断机制	3
2.2.2 保存现场	5
2.2.3 思考题 - 对比异常与函数调用	6
2.2.4 思考题 - 诡异的代码	7
2.2.5 实现系统调用	7
3 阶段二	9
3.1 标准输出与堆区管理的实现	9
3.1.1 标准输出	9
3.1.2 堆区管理	10
3.2 简易文件系统	11
4 阶段三	14
4.1 一切皆文件	14
4.1.1 把 VGA 显存抽象成文件	14
4.1.2 把设备输入抽象成文件	16
4.2 运行仙剑奇侠传	17
4.2.1 仙剑奇侠传的运行	17
4.2.2 必答题	18
5 总结	18

1 概述

1.1 实验目的

首先，深入研究操作系统的概念，明晰其核心原理与功能；其次，探索系统调用的机制，以及中断的实现方式，为操作系统提供必要的交互和响应能力；然后，深入了解文件系统的基本构成与功能，从而实现一个简单而有效的文件系统；最终，通过整合以上所学，实验设计一个支持文件操作的操作系统，并确保其能够成功运行仙剑奇侠传，以验证系统的完整性与可用性。

1.2 实验内容

实验内容主要分为三个关键阶段：首先，着重于熟悉操作系统的基本概念、系统调用以及中断机制的实现；其次，深入完善系统调用，逐步实现简易文件系统以支持基本的文件操作功能；最后，将输入输出抽象为文件的形式，并确保实现的操作系统能够成功运行仙剑奇侠传，从而全面验证系统的可用性和稳定性。

2 阶段一

2.1 操作系统——更方便的运行环境

2.1.1 代码：实现 Loader

根据实验手册，需要在 loader.c 中补全 loader 函数，将 ramdisk 中从 0 开始的所有内容放置在 0x4000000，并把这个地址作为程序的入口。根据提示，需要借助 nanos-lite/src/ramdisk.c 中的 ramdisk_read 函数和 get_ramdisk_size 函数。

```
extern void ramdisk_read(void *buf, off_t offset, size_t len);
extern size_t get_ramdisk_size();

uintptr_t loader(_Protect *as, const char *filename) {
    // TODO();

    ramdisk_read(DEFAULT_ENTRY, 0, get_ramdisk_size());
    return (uintptr_t)DEFAULT_ENTRY;
}
```

首先修改 makefile 默认编译到 x86 环境，，在 navy-apps/tests/dummy 目录下执行 make；在 nanos-lite 目录下执行 make update，再执行 make run，在 nemu 中执行程序，执行到 int 指令由于指令未补全，程序中止。

```
(nemu) c
[src/main.c,19,main] 'Hello World!' from Nanos-lite
[src/main.c,20,main] Build time: 05:40:47, May 12 2024
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x100cec, end = 0x1052c8, size = 17884 bytes
invalid opcode(eip = 0x04001f98): cd 80 5b 5d c3 66 90 90 ...

There are two cases which will trigger this unexpected exception:
1. The instruction at eip = 0x04001f98 is not implemented.
2. Something is implemented incorrectly.
Find this eip(0x04001f98) in the disassembling result to distinguish which case it is.

If it is the first case, see
0303 Manual
for more details.

If it is the second case, remember:
* The machine is always right!
* Every line of untested code is always wrong!
```

图 2.1: loader 成功加载 dummy

由此确定 loader 已成功加载 dummy。

2.2 穿越时空的旅程

2.2.1 实现中断机制

首先需要完善 CPU 结构：

1. 添加 CS 寄存器

为了在 QEMU 中顺利进行 differential testing, 需要在 cpu 结构体中添加一个 CS 寄存器, 并初始化为 8。

2. 添加 IDTR 寄存器

根据指导手册中的描述, IDTR 寄存器中存放着 IDT 的首地址和长度。

在 nemu/include/cpu/reg.h 中的 CPU_state 补全 CPU 结构：

```
uint32_t CS;
struct{
    uint32_t Base;
    uint32_t Limit;
}IDTR;
} CPU_state;
```

在 nemu/src/monitor/monitor.c 的 restart() 函数中初始化 CS：

```
cpu.CS = 8;
```

接着, 在 i386 手册 P416 中查找到 lidt 指令的位置在 gpr7 中 011 位置。

在 exec.c 中补全从而实现 lidt 指令：

```
make_group(gp7,
    EMPTY, EMPTY, EMPTY, EX(lidt),
    EMPTY, EMPTY, EMPTY, EMPTY)
```

阅读 i386 手册 P330 可以得知, `lidt` 指令在不同的操作数大小下有不同的行为动作, 如果是 16 位操作数 `IDTR` 的 `Base` 会读取 24 位; 如果是 32 位操作数 `IDTR` 的 `Base` 会读取 32 位。在这两种情况下, `Limit` 都是读取 16 位。补全 `lidt` 的 `Helper` 函数如下, 在读取时借助 `vaddr_read()` 函数:

```
make_EHelper(lidt) {
    //TODO();
    if(decoding.is_operand_size_16) cpu.IDTR.Base = vaddr_read(id_dest->addr+2,3);
    else cpu.IDTR.Base = vaddr_read(id_dest->addr+2,4);
    cpu.IDTR.Limit = vaddr_read(id_dest->addr,2);
    print_asm_template1(lidt);
}
```

然后实现 `int` 指令, 在 i386 手册 P414 中查找到 `int` 指令的位置在 `opcode_table` 中 `0xcd` 位置。在 `exec.c` 中补全:

```
/* 0xcc */    EMPTY, IDEXW(I, int, 1), EMPTY, EMPTY,
```

根据实验指导手册, `int` 指令的执行需要借助 `raise_intr()` 函数来触发中断。触发中断的过程如下:

1. 将 `eflags`、`cs`、返回地址压栈

在压栈的过程中需要用到 `rtl_push` 函数。

2. 从 `IDTR` 中读出 `IDT` 的首地址

3. 根据中断号来获取门描述符

由于门描述符是一个 8 字节的结构体, 因此要索引到的门描述符应该是 `IDT` 的首地址 `Base` 加上 8 倍的中断号。

4. 根据门描述符获取目标地址

从门描述符的位置读取 `offset` 域, 组合得到目标地址。

5. 进行跳转

将目标地址作为跳转地址, 同时要记得设置跳转。

`raise_intr()` 函数具体实现代码如下:

```
void raise_intr(uint8_t NO, vaddr_t ret_addr) {
    /* TODO: Trigger an interrupt/exception with ``NO``.
     * That is, use ``NO`` to index the IDT.
     */
    //TODO();
    rtl_push(&cpu.EFLAGS);
    rtl_push(&cpu.CS);
    rtl_push(&ret_addr);

    vaddr_t Gate = cpu.IDTR.Base + 8 * NO;

    uint32_t low_addr = vaddr_read(Gate,4);
    uint32_t high_addr = vaddr_read(Gate+4,4);
    uint32_t target_addr = (low_addr & 0x0000ffff) | (high_addr & 0xffff0000);

    decoding.is_jump = 1;
    decoding.jump_eip = target_addr;
}
```



```

make_EHelper(pusha) {
    //TODO();
    t0 = cpu.esp;
    rtl_push(&cpu.eax);
    rtl_push(&cpu.ecx);
    rtl_push(&cpu.edx);
    rtl_push(&cpu.ebx);
    rtl_push(&t0);
    rtl_push(&cpu.ebp);
    rtl_push(&cpu.esi);
    rtl_push(&cpu.edi);

    print_asm("pusha");
}

```

实验指导手册中说明了保存现场的步骤：

1. 保存 EFLAGS, CS, EIP;
2. vecsys() 压入错误码和异常号 #irq;
3. asm_trap() 把用户进程的通用寄存器保存到堆栈上。

_RegSet 结构体中的成员声明顺序需要与 TrapFrame 一致, 因此声明顺序为: 通用寄存器、#irq、错误码、EIP、CS、EFLAGS(其中通用寄存器声明顺序应与 cpu 结构中的顺序相反)

因此将 _RegSet 结构体重新组织为:

```

struct _RegSet {
    //uintptr_t esi, ebx, eax, eip, edx, error_code, eflags, ecx, cs, esp, edi, ebp;
    //int      irq;
    uintptr_t edi, esi, ebp, esp, ebx, edx, ecx, eax;
    int      irq;
    uintptr_t error_code;
    uintptr_t eip, cs, eflags;
};

```

重新运行 dummy, 可以看到在 do_event() 函数中触发了 BAD TRAP:

```

(nemu) c
[src/main.c,19,main] 'Hello World!' from Nanos-lite
[src/main.c,20,main] Build time: 06:28:11, May 12 2024
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x100ef0, end = 0x1054cc, size = 17884 bytes
[src/main.c,27,main] Initializing interrupt/exception handler...
[src/irq.c,5,do_event] system panic: Unhandled event ID = 8
nemu: HIT BAD TRAP at eip = 0x00100032

```

图 2.2: 保存现场成功

说明保存现场实现成功。

2.2.3 思考题 - 对比异常与函数调用

Q: 对比异常与函数调用: 我们知道进行函数调用的时候也需要保存调用者的状态: 返回地址, 以及调用约定 (calling convention) 中需要调用者保存的寄存器. 而进行异常处理之前却要保存更多的信息. 尝试对比它们, 并思考两者保存信息不同是什么原因造成的。

A:

- 异常是指程序在执行过程中遇到了错误或异常情况，导致程序不能正常执行。这时程序会抛出一个异常对象，并且在程序的调用链中查找能够处理该异常的异常处理代码。如果找到了异常处理代码，程序就会跳转到该代码处执行处理逻辑；如果没有找到，程序就会终止并输出异常信息。异常处理用于处理程序中不可预测的异常情况。在处理异常之前，程序的执行状态会被保存下来，包括当前指令的地址、寄存器的值、被调用函数的参数和局部变量等信息。这些信息可以帮助异常处理程序定位并解决异常。因此，异常处理程序需要保存更多的信息。
- 函数调用是指程序执行到函数调用语句时，暂停当前函数的执行，转而去执行被调用的函数，直到被调用函数返回之后，再回到原来的函数继续执行。函数调用通常用于封装和组织代码。

函数调用时，被调用函数的参数和返回值会被保存在栈中，而函数的局部变量也会被保存在栈中。当函数执行完毕后，栈中保存的这些信息就会被清除，恢复到函数调用前的状态。因此，函数调用时需要保存的信息相对少一些。

2.2.4 思考题 - 诡异的代码

Q: trap.S 中有一行 `pushl %esp` 的代码，乍看之下其行为十分诡异。你能结合前后的代码理解它的行为吗？

A: 这行代码保存了栈顶，在完成函数调用后可以恢复栈顶。

2.2.5 实现系统调用

根据 i386 手册 P414 得知 `popa` 指令位于 `opcode_table` 中的 `0x61` 位置。在 `exec.c` 中补全：

```
/* 0x60 */ EX(pusha), EX(popa), EMPTY, EMPTY,
```

阅读 i386 手册 P364，了解到 `popa` 指令的实现需要依次 `pop edi`、`esi`、`ebp`、`esp`、`ebx`、`edx`、`ecx`、`eax`。因此 `popa` 指令的 Helper 函数实现为：

```
make_EHelper(popa) {  
    //TODO();  
    rtl_pop(&cpu.edi);  
    rtl_pop(&cpu.esi);  
    rtl_pop(&cpu.ebp);  
    rtl_pop(&t0);  
    rtl_pop(&cpu.ebx);  
    rtl_pop(&cpu.edx);  
    rtl_pop(&cpu.ecx);  
    rtl_pop(&cpu.eax);  
    print_asm("popa");  
}
```

根据 i386 手册 P414 得知 `iret` 指令位于 `opcode_table` 中的 `0xcf` 位置。在 `exec.c` 中补全：


```
/* 0xcc */ EMPTY, IDEXW(I, int, 1), EMPTY, EX(iret),
```

阅读 i386 手册 P311 了解到在当前的实地址模式下, iret 指令依次 pop EIP、CS、EFLAGS。

```
make_EHelper(iret) {
    //TODO();
    rtl_pop(&decoding.jump_eip);
    decoding.is_jump = 1;
    rtl_pop(&cpu.CS);
    rtl_pop(&cpu.EFLAGS);
    print_asm("iret");
}
```

在 nanos-lite/src/irq.c 的 do_event() 函数中识别系统调用事件 _EVENT_SYSCALL 并调用 do_syscall 函数。

```
static _RegSet* do_event(_Event e, _RegSet* r) {
    switch (e.event) {
        case _EVENT_SYSCALL: {
            do_syscall(r);
            break;
        }
        default: panic("Unhandled event ID = %d", e.event);
    }
    return NULL;
}
```

在 nexus-am/am/arch/x86-nemu/include/arch.h 中完成 SYSCALL_ARG1、SYSCALL_ARG2、SYSCALL_ARG3、SYSCALL_ARG4 的宏定义, 获得正确的系统调用参数寄存器。

```
#define SYSCALL_ARG1(r) r->eax
#define SYSCALL_ARG2(r) r->ebx
#define SYSCALL_ARG3(r) r->ecx
#define SYSCALL_ARG4(r) r->edx
```

在 do_syscall 函数中使用设置好的宏:

```
_RegSet* do_syscall(_RegSet *r) {
    uintptr_t a[4];
    a[0] = SYSCALL_ARG1(r);
    a[1] = SYSCALL_ARG2(r);
    a[2] = SYSCALL_ARG3(r);
    a[3] = SYSCALL_ARG4(r);
```

然后来增加系统调用。在 do_syscall 函数中增加 SYS_none 系统调用, 该系统调用什么都不用做, 只需要设置返回值为 1。

```
switch (a[0]) {
    case SYS_none: {
        //Log("sys_none");
        SYSCALL_ARG1(r) = 1;
        break;
    }
}
```

重新运行 dummy，触发了 ID 为 4 的系统调用：

```
(nemu) c
[src/main.c,19,main] 'Hello World!' from Nanos-lite
[src/main.c,20,main] Build time: 08:26:39, May 12 2024
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x100fbc, end = 0x105598, size = 17884 bytes
[src/main.c,27,main] Initializing interrupt/exception handler...
[src/syscall.c,17,do_syscall] system panic: Unhandled syscall ID = 4
nemu: HIT BAD TRAP at eip = 0x00100032
```

根据手册，还需要实现 SYS_exit 系统调用。SYS_exit 系统调用会接收一个退出状态的参数，用这个参数调用 __halt() 即可：

```
case SYS_exit:{
    //Log("sys_exit");
    __halt(a[1]);
    break;
}
```

再次运行 dummy，可以看到 GOOD TRAP:

```
(nemu) c
[src/main.c,19,main] 'Hello World!' from Nanos-lite
[src/main.c,20,main] Build time: 08:26:39, May 12 2024
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x100fd4, end = 0x1055b0, size = 17884 bytes
[src/main.c,27,main] Initializing interrupt/exception handler...
nemu: HIT GOOD TRAP at eip = 0x00100032
```

3 阶段二

3.1 标准输出与堆区管理的实现

3.1.1 标准输出

根据手册，需要在 do_syscall 函数中对系统调用号 SYS_write 进行识别

```
case SYS_write:{
    SYSCALL_ARG1(r) = sys_write(a[1],(void *)a[2],a[3]);
    break;
}
```

识别后检查 fd 的值：若 fd 为 1 或 2，使用 __putc 将 buf 为首地址的 len 字节输出到串口，查阅 man 2 write 得知返回值写的字符的字节数：

```
int sys_write(int fd,void* buf,size_t len) {
    if(fd==1||fd==2){
        for(int i=0;i<len;i++){
            __putc(((char*)buf)[i]);
        }
        return len;
    }
    else
        panic("Unhandled fd = %d in sys_write", fd);
    return -1;
}
```

在 navy-apps/libs/libos/src/nanos.c 的 __write() 中调用系统调用接口函数：

```
int _write(int fd, void *buf, size_t count){
    return _syscall(SYS_write, fd, (uintptr_t)buf, count);
}
```

把 Nanos-lite 上运行的用户程序切换成 hello 程序并运行：

```
(nemu) c
[src/main.c,19,main] 'Hello World!' from Nanos-lite
[src/main.c,20,main] Build time: 08:26:39, May 12 2024
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x1010c8, end = 0x1057a4, size = 18140 bytes
[src/main.c,27,main] Initializing interrupt/exception handler...
Hello World!
Hello World for the 2th time
Hello World for the 3th time
Hello World for the 4th time
Hello World for the 5th time
Hello World for the 6th time
Hello World for the 7th time
Hello World for the 8th time
Hello World for the 9th time
Hello World for the 10th time
```

可以看到持续输出了 Hello World for the ***th time, 说明实现成功。

3.1.2 堆区管理

在 sys_write 函数中加上 Log("sys_write"); 再次运行

```
(nemu) c
[src/main.c,19,main] 'Hello World!' from Nanos-lite
[src/main.c,20,main] Build time: 08:26:39, May 12 2024
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x101120, end = 0x1057fc, size = 18140 bytes
[src/main.c,27,main] Initializing interrupt/exception handler...
[src/syscall.c,36,do_syscall] sys_write
Hello World!
[src/syscall.c,36,do_syscall] sys_write
H[src/syscall.c,36,do_syscall] sys_write
e[src/syscall.c,36,do_syscall] sys_write
l[src/syscall.c,36,do_syscall] sys_write
l[src/syscall.c,36,do_syscall] sys_write
o[src/syscall.c,36,do_syscall] sys_write
[src/syscall.c,36,do_syscall] sys_write
W[src/syscall.c,36,do_syscall] sys_write
o[src/syscall.c,36,do_syscall] sys_write
r[src/syscall.c,36,do_syscall] sys_write
l[src/syscall.c,36,do_syscall] sys_write
d[src/syscall.c,36,do_syscall] sys_write
[src/syscall.c,36,do_syscall] sys_write
f[src/syscall.c,36,do_syscall] sys_write
```

可以看到字符串是逐字符输出的。要想字符串依次输出，就需要分配缓冲区来存放格式化的内容。因此需要实现堆区大小调整库函数 sbrk() 中调用的 __sbrk() 函数。

__sbrk() 函数功能的实现需要先实现设置堆区大小的系统调用 SYS_brk, SYS_brk 系统调用成功返回 0，系统调用失败返回-1。

SYS_brk 实现如下：

```
case SYS_brk:{
    SYSCALL_ARG1(r) = 0;
    break;
}
```

__sbrk() 函数位于 navy-apps/libs/libos/src/nanos.c 中,在该函数中需要进行 SYS_brk 系统调用，如果调用成功，需要更新 program break 的位置，并返回旧的 program break 值；如果调用失败返回-1。

```
extern char _end;
intptr_t pb = (intptr_t)&_end;
void *_sbrk(intptr_t increment){
    //return (void *)-1;
    intptr_t old_pb = pb;
    if(_syscall_(SYS_brk, old_pb+increment, 0, 0)==0){
        pb += increment;
        return (void *)old_pb;
    }
    return (void *)-1;
}
```

再次运行，可以看到是以字符串为单位输出的：

```
(nemu) c
[src/main.c,19,main] 'Hello World!' from Nanos-lite
[src/main.c,20,main] Build time: 08:26:39, May 12 2024
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x10113c, end = 0x10585c, size = 18208 bytes
[src/main.c,27,main] Initializing interrupt/exception handler...
[src/syscall.c,36,do_syscall] sys_write
Hello World!
[src/syscall.c,36,do_syscall] sys_write
Hello World for the 2th time
[src/syscall.c,36,do_syscall] sys_write
Hello World for the 3th time
[src/syscall.c,36,do_syscall] sys_write
Hello World for the 4th time
[src/syscall.c,36,do_syscall] sys_write
Hello World for the 5th time
[src/syscall.c,36,do_syscall] sys_write
Hello World for the 6th time
[src/syscall.c,36,do_syscall] sys_write
```

3.2 简易文件系统

按照手册指示需修改 makefile，make update 后会整合出 ramdisk 镜像及其文件记录表，文件记录表的下标被用作文件的文件描述符来标识文件。

```
typedef struct {
    char *name;
    size_t size;
    off_t disk_offset;
} Finfo;
```

涉及到文件的操作包括打开 open、关闭 close、读 read、写 write。

为避免每次读写操作都需要从头开始，引入读写偏移量记录当前文件的操作位置：

```
typedef struct {
    char *name;
    size_t size;
    off_t disk_offset;
    off_t open_offset;
} Finfo;
```

对于偏移量的调整通过 lseek 系统调用实现。

因此需要实现的文件操作包括：fs_open、fs_close、fs_read、fs_write、fs_lseek。

打开文件时需要遍历文件记录表，逐一比对遍历到的文件名（文件路径）与要打开的文件是否相同，如果相同说明找到了要打开的文件，那么就把当前文件的读写偏移量 open_offset 置为 0。如果遍历完所有文件都没有找到要打开的文件，说明出现异常，需要终止程序。

```
int fs_open(const char *pathname, int flags, int mode){
    int i;
    for(i = 0; i < NR_FILES; i++){
        if (strcmp(file_table[i].name, pathname) == 0) {
            file_table[i].open_offset = 0;
            return i;
        }
    }
    Log("pathname:%s\n",pathname);
    panic("File not exists");
    return -1;
}
```

为打开文件，需要在 do_syscall 函数中对系统调用号 SYS_open 进行识别，识别到后调用 fs_open 打开文件：

```
case SYS_open:{
    //Log("sys_open");
    SYSCALL_ARG1(r) = fs_open((char *)a[1],a[2],a[3]);
    break;
}
```

根据手册，文件关闭函数 fs_close 可以直接返回 0 表示关闭成功。

之后在 do_syscall 函数中添加对应系统调用 SYS_close，识别系统调用后调用 fs_close 函数关闭文件：

```
case SYS_close:{
    //Log("sys_close");
    SYSCALL_ARG1(r) = fs_close(a[1]);
    break;
}
```

文件读取时需要借助 ramdisk_read 函数进行真正的读写，需要的参数有读取位置距镜像起始位置的偏移量和读取长度。距镜像起始位置的偏移量就是文件磁盘偏移量加上读写偏移量，读取完毕后需要更新读写偏移量，函数返回读取长度。

```
ssize_t fs_read(int fd, void *buf, size_t len){
    if(read_len > len) read_len = len;
    ramdisk_read(buf, file_table[fd].disk_offset + file_table[fd].open_offset, read_len);
    file_table[fd].open_offset += read_len;
    return read_len;
}
```

之后在 do_syscall 函数中添加对应系统调用 SYS_read：

```
case SYS_read:{
    //Log("sys_read");
    SYSCALL_ARG1(r) = fs_read(a[1],(void *)a[2],a[3]);
    break;
}
```

根据手册，loader 在读取文件时需要获取文件大小，因此添加通过文件描述符（文件记录表下标）获取结构属性值的辅助函数 fs_filesz。

实现好文件系统的打开、关闭、读取后，就可以使用文件系统来 load 文件。在 loader 需要打开文件，读取整个文件，然后关闭文件：

```
uintptr_t loader(_Protect *as, const char *filename) {
    //v1
    // ramdisk_read(DEFAULT_ENTRY, 0, get_ramdisk_size());\

    //v2
    Log("opening file %s", filename);
    int fd = fs_open(filename, 0, 0);
    int file_size = fs_filesz(fd);
    fs_read(fd, DEFAULT_ENTRY, file_size);
    fs_close(fd);
    Log("successfully open file");

    return (uintptr_t)DEFAULT_ENTRY;
}
```

同时在 main 函数中需要把 loader 的第二个参数改为要传入的文件名。例如：`loader(NULL, "/bin`
在文件系统中，写的系统调用需要改写。基本思路和读取一致，需要在 fs.c 中编写 `fs_write` 函数。

首先判断写入类型，如果是标准输出 `stdout` 或标准错误 `stderr`，还是使用 `_putc` 来写，如果是其他情况，需要借助 `ramdisk_write` 函数来完成，需要的参数有写入位置距镜像起始位置的偏移量和写入长度。距镜像起始位置的偏移量就是文件磁盘偏移量加上读写偏移量，读取完毕后需要更新读写偏移量，函数返回读取长度。

`fs_write` 函数实现如下：

```
ssize_t fs_write(int fd, const void *buf, size_t len){
    int write_len = file_table[fd].size - file_table[fd].open_offset;
    if(write_len > len) write_len = len;
    switch(fd){
        case FD_STDOUT:
        case FD_STDERR:
            for (int i = 0; i < len; i++) _putc( ((char *)buf)[i] );
            return len;
        default:
            ramdisk_write(buf, file_table[fd].disk_offset + file_table[fd].open_offset, write_len);
            break;
    }
    file_table[fd].open_offset += write_len;
    return write_len;
}
```

文件读写偏移量对应 `SYS_lseek` 函数调用。

```
case SYS_lseek:{
    //Log("sys_lseek");
    SYSCALL_ARG1(r) = fs_lseek(a[1], a[2], a[3]);
    break;
}
```

参数 `whence` 指明了调整方式：

1. `SEEK_SET`：基准位置为文件开头，即 `offset` 表示距离文件开头的偏移量。
2. `SEEK_CUR`：基准位置为文件当前位置，即 `offset` 表示距离文件当前位置的偏移量。

3. SEEK_END: 基准位置为文件末尾, 即 offset 表示距离文件末尾的偏移量。

因此 fs_lseek 函数实现如下:

```
off_t fs_lseek(int fd, off_t offset, int whence){
    off_t src_offset, dst_offset;
    src_offset = file_table[fd].open_offset;
    dst_offset = 0;
    switch(whence){
        case SEEK_SET:
            dst_offset = offset;
            break;
        case SEEK_CUR:
            dst_offset = src_offset + offset;
            break;
        case SEEK_END:
            dst_offset = fs_filesz(fd) + offset;
            break;
        default:
            assert(0);
    }
    if(dst_offset < 0) dst_offset = 0;
    else if(dst_offset > fs_filesz(fd)) dst_offset = fs_filesz(fd);
    file_table[fd].open_offset = dst_offset;

    return dst_offset;
}
```

在 main.c 中将调用的 loader 函数的第二个参数改为"/bin/text",make run :

```
(nemu) c
[src/main.c,19,main] 'Hello World!' from Nanos-lite
[src/main.c,20,main] Build time: 02:26:16, May 13 2024
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x101a30, end = 0x37044e, size = 2550302 bytes
[src/main.c,27,main] Initializing interrupt/exception handler...
[src/main.c,38,main] loading!...
[src/loader.c,22,loader] opening file /bin/text
[src/loader.c,27,loader] successfully open file
PASS!!!
nemu: HIT GOOD TRAP at eip = 0x00100032
```

看到输出" PASS!!! ", 并 GOOD TRAP 说明实现正确。

4 阶段三

4.1 一切皆文件

4.1.1 把 VGA 显存抽象成文件

VGA 抽象成文件涉及到/dev/fb 和/proc/dispinfo 两个特殊的文件。

1. /dev/fb

- 初始化: 在 fs.c 的 init_fs 中设置/dev/fb 的大小

```
void init_fs() {
    // TODO: initialize the size of /dev/fb
    file_table[FD_FB].size = (_screen.width * _screen.height) * sizeof(uint32_t);
}
```

- 写：要将 buf 中的 len 字节写到屏幕 offset 处，需要先计算在屏幕上的坐标，像素储存是以行优先方式存储的，所以横坐标是用大小除宽度取余数，纵坐标是用大小除宽度取商。

在 fb_write 函数中实现：

```
void fb_write(const void *buf, off_t offset, size_t len) {
    off_t Size = offset/sizeof(uint32_t);
    _draw_rect(buf, Size % _screen.width, Size / _screen.width, len / sizeof(uint32_t), 1);
}
```

2. /proc/dispinfo

- 初始化：在 init_device 中将/proc/dispinfo 提前写入 dispinfo 字符串

```
void init_device() {
    _ioe_init();
    // TODO: print the string to array `dispinfo` with the format
    // described in the Navy-apps convention
    sprintf(dispinfo, "WIDTH: %d\nHEIGHT: %d", _screen.width, _screen.height);
}
```

- 读：把字符串 dispinfo 中 offset 开始的 len 字节写到 buf 中，字符串的写入借助 memcpy 库函数。

在 dispinfo_read 函数中实现：

```
void dispinfo_read(void *buf, off_t offset, size_t len) {
    memcpy(buf, dispinfo + offset, len);
}
```

3. 文件系统的支持

文件系统中涉及到/dev/fb 的是 fs_write 函数，涉及到/proc/dispinfo 的是 fs_read 函数：

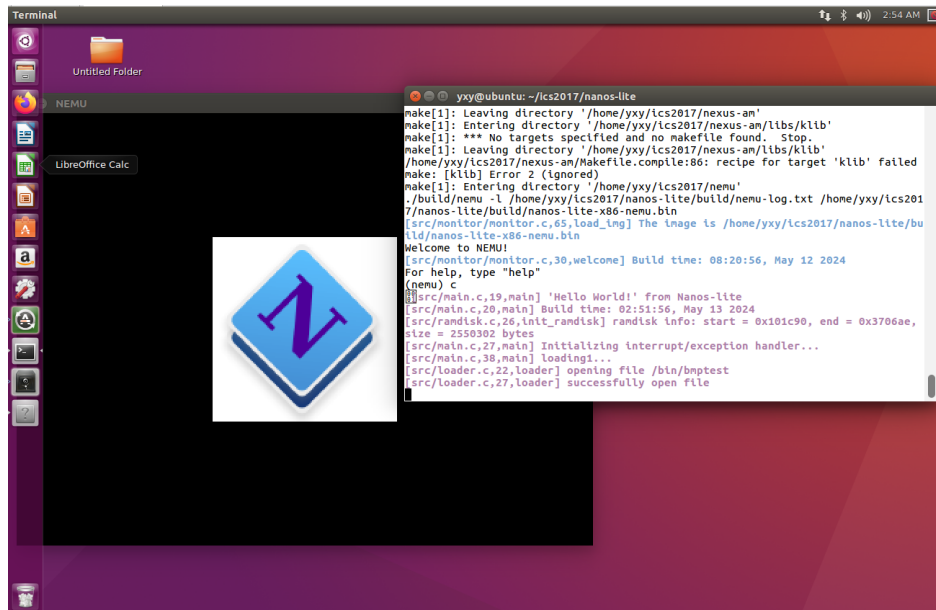
在 fs_write 函数中 fd 为 FD_FB 的情况下调用前面完善好的 fb_write 函数；

```
case FD_FB:
    fb_write(buf, file_table[fd].open_offset, write_len);
    break;
```

在 fs_read 函数中 fd 为 FD_DISPINFO 的情况下调用前面完善好的 dispinfo_read 函数。

```
if(fd == FD_DISPINFO)
    dispinfo_read(buf, file_table[fd].disk_offset + file_table[fd].open_offset, read_len);
else
    ramdisk_read(buf, file_table[fd].disk_offset + file_table[fd].open_offset, read_len);
```

最后，加载/bin/bmptest 并运行：



屏幕上正确显示了 logo，说明实现正确。

4.1.2 把设备输入抽象成文件

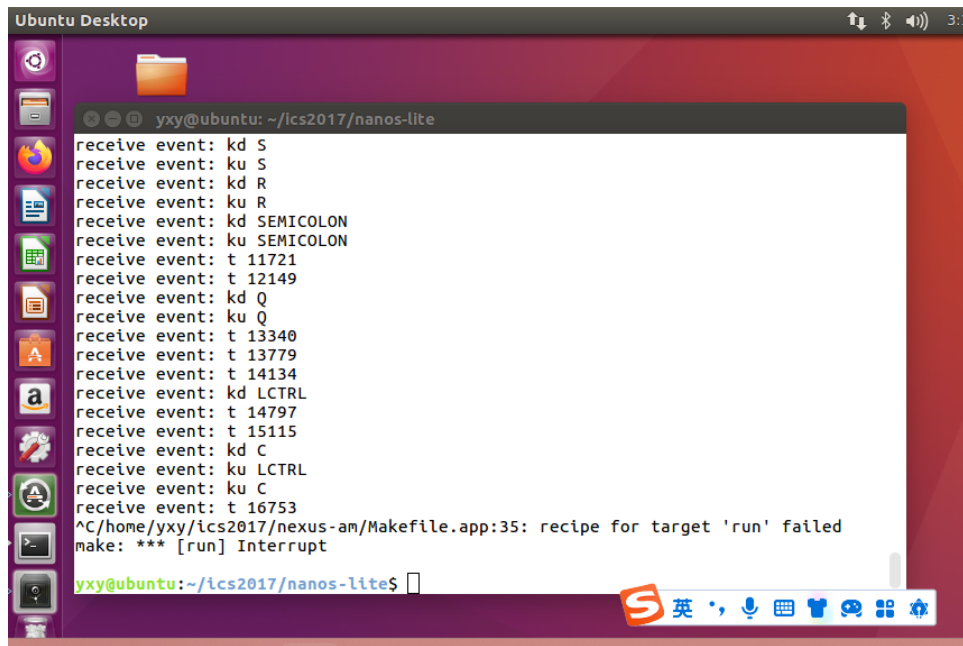
将时间事件和按键事件抽象成特殊文件 `/dev/events`。

在 `events_read` 中实现对时间的读取，将事件写入到 `buf` 中。首先需要获取按键；如果没有按键，应该输出启动后的时间；如果是按键需要获取按键内容，根据内容获取对应事件，具体区别为按键按下 `ka` 和松开 `ku`，然后输出事件信息；最后返回写入的实际长度。

```
size_t events_read(void *buf, size_t len) {
    int key = _read_key();
    if(key == _KEY_NONE){
        unsigned long event_time = _uptime();
        return snprintf(buf, len, "t %d\n", event_time) - 1;
    }
    else{
        char kd_ku;
        if(key & 0x8000) {kd_ku = 'd'; key = key ^ 0x8000;}
        else kd_ku = 'u';
        return snprintf(buf, len, "k%c %s\n", kd_ku, keyname[key]) - 1;
    }
}
```

在文件系统中读写都涉及到 `/dev/events`，因此在 `fs_read`、`fs_write` 函数函数中增加对 `fd` 为 `FD_EVENTS` 情况的处理：在 `fs_read` 中调用前面实现的 `events_read` 函数，因为 `events_read` 函数返回写入的实际长度，所以在 `fs_read` 可以直接返回 `events_read` 的调用结构；在 `fs_write` 中直接返回长度。

加载 `/bin/events` 并运行：



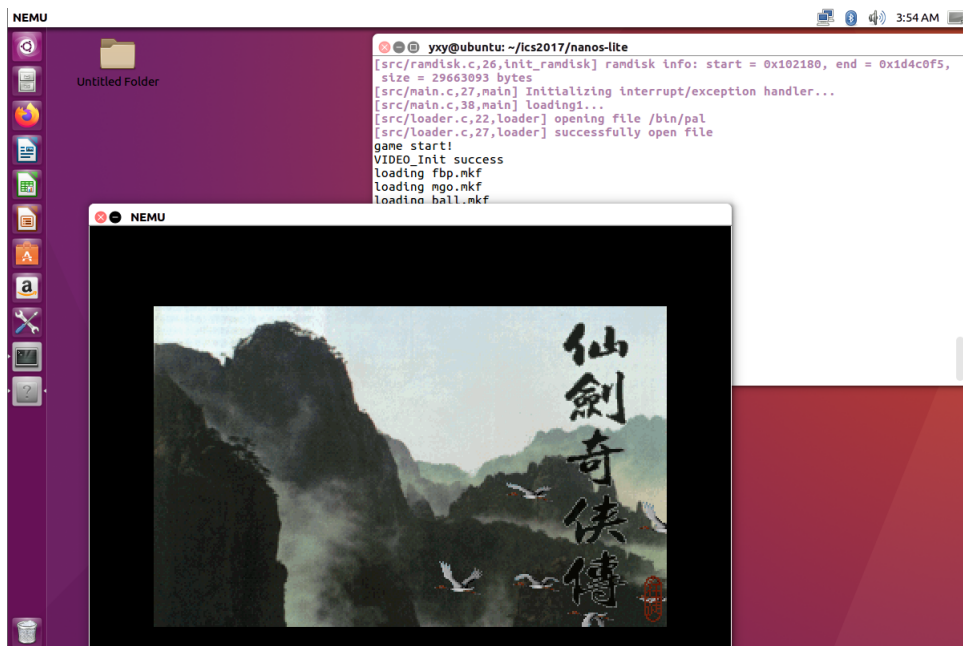
```
xyx@ubuntu: ~/ics2017/nanos-lite
receive event: kd S
receive event: ku S
receive event: kd R
receive event: ku R
receive event: kd SEMICOLON
receive event: ku SEMICOLON
receive event: t 11721
receive event: t 12149
receive event: kd Q
receive event: ku Q
receive event: t 13340
receive event: t 13779
receive event: t 14134
receive event: kd LCTRL
receive event: t 14797
receive event: t 15115
receive event: kd C
receive event: ku LCTRL
receive event: ku C
receive event: t 16753
^C/home/xyx/ics2017/nexus-am/Makefile.app:35: recipe for target 'run' failed
make: *** [run] Interrupt
xyx@ubuntu:~/ics2017/nanos-lite$
```

屏幕上输出了运行时间，按下按键也会有显示，说明实现正确。

4.2 运行仙剑奇侠传

4.2.1 仙剑奇侠传的运行

将仙剑奇侠传数据文件放到 navy-apps/fsimg/share/games/pal/目录下，make update 更新 ramdisk，加载/bin/pal 并运行：



正确运行。

4.2.2 必答题

Q: 结合代码解释仙剑奇侠传, 库函数,libos,Nanos-lite,AM,NEMU 是如何相互协助, 来分别完成游戏存档的读取和屏幕的更新。

A:

- 首先 make update 编译 pal, 编译好的文件会存在 ramdisk 镜像中。
- make run 会先运行 NEMU, 然后再在 NEMU 上运行 Nanos-lite。
- Nanos-lite 的 main 函数中 `uint32_t entry = loader(NULL, "/bin/pal");` 会从 ramdisk 中加载/bin/pal 程序, 完成初始化后会运行到仙剑奇侠传的入口。
- 仙剑奇侠传的程序执行过程中会调用库函数、进行函数调用; libos 中准备好了函数接口。
- 读取游戏存档时 `fread()` 函数发生中断, `SYS_read` 系统调用, 调用 nanos-lite 中的 `fs_read` 读取文件中的存档, 然后就可以成功读取文件内容。
- 进行读档屏幕更新时也是产生中断, 然后 nanos-lite 的 IOE 函数开始执行, 调用 am 中的 IOE 函数, 成功返回文件的更新内容, 就可以实现屏幕的更新。
- Nanos-lite 是 NEMU 的客户程序, 它运行在 AM 之上; 仙剑奇侠传是 Nanos-lite 的用户程序, 它运行在 Nanos-lite 之上。

5 总结

本次实验主要是完成了加载、中断和系统调用等, 然后又改到文件系统中给进行了实现。实验的 debug 还是比较难, 如果哪里出现了问题可能会卡住的时间比较长, 一方面要适当的去做版本回退, 因此也需要及时地做提交; 另一方面需要静下心来有耐心地去回顾流程, 找到问题所在。