

计算机系统设计实验

PA3 - 穿越时空的旅程: 异常控制流

安祺 1913630

南开大学计算机学院

日期: 2022 年 5 月 3 日

目录

1 概述	2	3.2 实现堆区	7
1.1 实验目的	2	3.3 简易文件系统	8
1.2 实验内容	2	3.3.1 让 loader 使用文件	8
2 阶段一	2	3.3.2 实现完整的文件系统	8
2.1 实现 loader	2	4 一切皆文件	8
2.2 实现中断机制	2	4.1 把 VGA 显存抽象成文件	8
2.2.1 lidt 指令	2	4.2 把设备输入抽象成文件	10
2.2.2 int 指令	3	4.3 在 NEMU 中运行仙剑奇侠传	11
2.2.3 iret 指令	4	5 bug 汇总	11
2.3 保存现场	4	5.1 PUSHA 和 POPA 顺序	11
2.3.1 对比异常与函数调用	4	5.2 FD_STDOUT 文件需要单独处理	11
2.3.2 诡异的代码	4	5.3 显存按照 uint32 对齐	12
2.3.3 重新组织 TrapFrame 结构体	5	6 必答题	12
2.4 实现系统调用	6	7 体会与感悟	12
3 文件系统	7		
3.1 标准输出	7		

1 概述

1.1 实验目的

1. 理解体会系统调用和文件系统的原理

1.2 实验内容

阶段一 实现第一个系统调用。

阶段二 实现简易文件系统。

阶段三 运行仙剑奇侠传。

2 阶段一

2.1 实现 loader

首先是对 loader 进行实现，这里使用 ramdisk.c 中提供的读取函数直接将用户程序加载到 0x40000000 处。实现 loader 函数后，执行make run后提示 int 指令未实现。

Listing 1: loader 函数的实现

```
uintptr_t loader(_Protect *as, const char *filename) {
    extern size_t get_ramdisk_size();
    extern size_t ramdisk_read(void*, off_t, size_t);
    size_t len=get_ramdisk_size();
    ramdisk_read(DEFAULT_ENTRY, 0, len);
    return (uintptr_t)DEFAULT_ENTRY;
}
```

2.2 实现中断机制

2.2.1 lidt 指令

需要使用 lidt 指令，设置 IDTR 为 IDT 的首地址和长度。在 NEMU 中，需要先实现 IDTR 寄存器，这里在 CPU_state 添加一个匿名结构体成员作为 idtr。

Listing 2: idtr 的实现

```
struct {
    uint32_t base;
    uint16_t limit;
} idtr;
```

然后实现 lidt 的 helper 函数，并在 gp7 中插入执行函数。这里需要注意的是操作数类型为 m16&32，16bit 的 limit 保存在低位，而 32bit 的 base 保存在高位。

Listing 3: lidt 的 helper 实现

```
make_EHelper(lidt) {
    cpu.idtr.limit = vaddr_read(id_dest->addr,2);
    int base_len=decoding.is_operand_size_16?3:4;
    cpu.idtr.base = vaddr_read(id_dest->addr+2,base_len);

    print_asm_template1(lidt);
}
```

2.2.2 int 指令

实现 int 指令，先在 opcode_table 中插入表项 IDEXW(I,int,1)，然后再 int 的 EHelper 中调用 raise_intr 函数即可。这里需要注意，保存的地址应当是下一条指令的地址。

需要实现 raise_intr 函数。首先需要在 CPU_state 中添加一个 CS 寄存器，用于压栈使用，无实际意义。而在 nemu 已经定义了门描述符的结构 (GateDesc)，这里可以直接进行使用。

Listing 4: lidt 的 helper 实现

```
void raise_intr(uint8_t NO, vaddr_t ret_addr) {
    vaddr_t desc_addr=cpu.idtr.base+8*NO;
    GateDesc desc;
    *((uint32_t*)&desc)=vaddr_read(desc_addr,4);
    *(((uint32_t*)&desc)+1)=vaddr_read(desc_addr+4,4);

    decoding.is_jump = true;
    decoding.jump_eip = ((uint32_t)(desc.offset_15_0))
        | (((uint32_t)(desc.offset_31_16))<<16);

    rtl_push(&cpu.eflags.reg);
    t2=cpu.cs;
    rtl_push(&t2);
    rtl_push(&ret_addr);
}
```

这里注意到在 restart 函数中对 cs 寄存器进行初始化，在 nanos-lite 中定义 HAS_ASYE 宏，否则会有错误。

在进行完成以上实现工作后，看到在 vecsys() 附近触发了未实现指令，说明中断机制实现正确，如图1所示。

```
4001f98:  cd 80                                int $0x80
100b14:  6a 00                                pushb $0x0
100b16:  68 80 00 00 00                      pushl $0x80
100b1b:  eb 06                                jmp 100b23
100b23:  60 60 54 e8 f2 fe ff ff 83          invalid opcode
```

图 1: 中断执行后的 NEMU 执行日志

区别	函数调用	异常
发起者	主动调用	主动发起或 CPU 产生异常
功能	执行另一段程序，无特权级转换	可以提高特权级，调用内核功能
状态保存	需要当前函数局部变量和返回地址	通用寄存器、EFLAGAS、错误码、EIP、CS

表 1: 对比函数调用和异常

2.2.3 iret 指令

iret 的实现更简单，出栈跳转即可。

2.3 保存现场

2.3.1 对比异常与函数调用

实验指导书书中提到了这样一个问题：

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。

下面对比一下异常和函数调用，见表1。两种保存信息不同是由于异常比函数转移更彻底。函数发起时，程序所执行指令地址发送转移，仍然在同一用户空间内运行，CPU 控制权仍然保留在同一用户空间。异常发起时，除了所执行指令地址发送变换，也不能保证在同一用户空间，例如发起系统调用时 CPU 控制器会转移到操作系统内核。

2.3.2 诡异的代码

`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？

“`pushl %esp`” 后面的一条指令是 “`call irq_handle`” 查看 `irq_handle` 的函数声明可以知道 `esp` 其实是函数的参数。

Listing 5: irq_handle 函数声明

```
_RegSet* irq_handle(_RegSet *tf);
```

2.3.3 重新组织 TrapFrame 结构体

为了实现保存现场的功能，需要首先实现 PUSH 指令，该指令将这些寄存器的值依次压栈：EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI。与 PUSH 对应的 POP 指令，从栈中恢复这些寄存器的值。这里需要注意保存和恢复的 ESP 的值应当是 PUSH 执行前的值。这两条指令均只有一个 OPCODE 字节，不需要额外译码。

Listing 6: PUSH 执行函数实现

```
make_EHelper(push) {
    if (decoding.is_operand_size_16) {
        TODO();
    } else {
        rtl_lr(&t1, R_ESP, 4);
        rtl_lr(&t0, R_EAX, 4); rtl_push(&t0);
        rtl_lr(&t0, R_ECX, 4); rtl_push(&t0);
        rtl_lr(&t0, R_EDX, 4); rtl_push(&t0);
        rtl_lr(&t0, R_EBX, 4); rtl_push(&t0);
        rtl_push(&t1);
        rtl_lr(&t0, R_EBP, 4); rtl_push(&t0);
        rtl_lr(&t0, R_ESI, 4); rtl_push(&t0);
        rtl_lr(&t0, R_EDI, 4); rtl_push(&t0);
    }
    print_asm("push");
}
```

Listing 7: POP 执行函数实现

```
make_EHelper(pop) {
    if (decoding.is_operand_size_16) {
        TODO();
    } else {
        rtl_pop(&t0); rtl_sr(&t1, R_EDI, 4);
        rtl_pop(&t0); rtl_sr(&t1, R_ESI, 4);
        rtl_pop(&t0); rtl_sr(&t1, R_EBP, 4);
        rtl_pop(&t0);
        rtl_pop(&t0); rtl_sr(&t1, R_EBX, 4);
        rtl_pop(&t0); rtl_sr(&t1, R_EDX, 4);
        rtl_pop(&t0); rtl_sr(&t1, R_ECX, 4);
        rtl_pop(&t0); rtl_sr(&t1, R_EAX, 4);
    }
}
```

```

}
print_asm("popa");
}

```

int 指令执行时, eflags, cs 和 eip 依次压栈, vecsys() 执行时 error_code 和 irq 依次压栈, 在 asm_trap 中所有通用寄存器依次压栈。因此, trap frame 的结构应当是这些值从低地址到高地址排列, 见代码段8。

Listing 8: POPA 执行函数实现

```

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

```

最后, 成功触发触发了 BAD TRAP, 如图4

```

[src/main.c,19,main] 'Hello World!' from Nanos-lite
[src/main.c,20,main] Build time: 16:25:49, Apr 29 2022
[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: 触发了 BAD TRAP

2.4 实现系统调用

popa 和 iret 指令前面已经实现了, 这里需要注意系统调用的参数依次放入%eax, %ebx, %ecx, %edx 四个寄存器, 返回值放入%eax, 其余照做即可。

```

[src/main.c,20,main] Build time: 17:08:07, Apr 29 2022
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x100fb0, end = 0x10558c, size = 17884 bytes
[src/main.c,27,main] Initializing interrupt/exception handler...
[src/syscall.c,14,do_syscall] system panic: Unhandled syscall ID = 4
nemu: HIT BAD TRAP at eip = 0x00100032

```

图 3: SYS_exit 未实现

```

[src/main.c,20,main] Build time: 17:22:31, Apr 29 2022
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x100fc8, end = 0x1055a4, size = 17884 bytes
[src/main.c,27,main] Initializing interrupt/exception handler...
nemu: HIT GOOD TRAP at eip = 0x00100032

```

图 4: 触发了 GOOD TRAP

3 文件系统

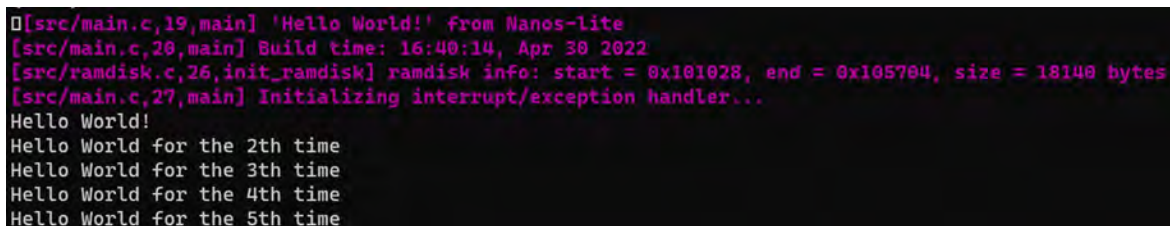
3.1 标准输出

实验过程不一一叙述了，参考书中有一点描述的不是很清楚，“实现 write() 系统调用”一开始以为是在 do_syscall 添加 SYS_write 的处理就行了，其实不然。追溯 hello.c 调用的 write 函数，发现还需要修改 nanos.c 文件中的 _write() 函数（代码9），默认是直接退出，需要修改为调用 SYS_write。最终成功运行 hello(图5)。

Listing 9: _write() 函数

```
//修改前
int _write(int fd, void *buf, size_t count){
    _exit(SYS_write);
}

//修改后
int _write(int fd, void *buf, size_t count){
    _syscall_(SYS_write, fd, (uintptr_t)buf, count);
}
```



```
0[src/main.c,19,main] 'Hello World!' from Nanos-lite
[src/main.c,20,main] Build time: 16:40:14, Apr 30 2022
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x101028, end = 0x105704, 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
```

图 5: 运行 Hello

3.2 实现堆区

这一步的难点在于如何使用 _end 符号。通过“man 3 end”得到，可以按如下方式获得 _end 地址。

Listing 10: 获得 _end 地址

```
extern char _end;
static void *program_break = (void *)&_end;
```


3.3 简易文件系统

3.3.1 让 loader 使用文件

首先需要在 fs.h 中声明如下函数，然后依次实现。

```
int fs_open(const char *pathname, int flags, int mode);
ssize_t fs_read(int fd, void *buf, size_t len);
ssize_t fs_write(int fd, uint8_t *buf, size_t len);
off_t fs_lseek(int fd, off_t offset, int whence);
int fs_close(int fd);
size_t fs_filesz(int fd);
```

实现 fs_open 函数只需要遍历 file_table 依次比较文件名即可，并断言一定能找到知道文件。其中，NR_FILES 宏表示文件数量。fs_close 不做操作，直接返回成功即可。fs_read 预留的 6 个文件先不做处理，其他文件调用 ramdisk_read 函数进行读取。最后在 loader 中换用 fs_read 实现加载即可。

3.3.2 实现完整的文件系统

fs_write 的实现类似 fs_read，但是需要使用 _putc 输出 stderr 和 stdout。在实现 fs_lseek 需要对 SEEK_SET、SEEK_CUR、SEEK_END 三种情况单独处理。

4 一切皆文件

4.1 把 VGA 显存抽象成文件

在 am.h 中声明了 _screen 变量，保存屏幕信息，可以使用该变量信息初始化/dev/fb 大小（代码段14），其中每个像素点使用一个 32 为整数表示。

Listing 11: 初始化/dev/fb 大小

```
void init_fs() {
    file_table[FD_FB].disk_offset=sizeof(uint32_t)*_screen.height*
        _screen.width;
    file_table[FD_FB].open_offset=0;
}
```

调用 _draw_rect() 即可实现 fb_write，注意坐标顺序即可；init_device 模仿 native 格式输出；dispinfo_read 检查一下内存地址越界即可。

最后在 fs_write 和 fs_read 单独处理特殊文件即可，最终成功显示图案(图7)。

实验期间遇到 bug，在 native 中实验 fprintf 对文件依次输出，而在 init_device 使用 sprintf 需要注意数组偏移，否则会覆盖之前输出（代码段12）。

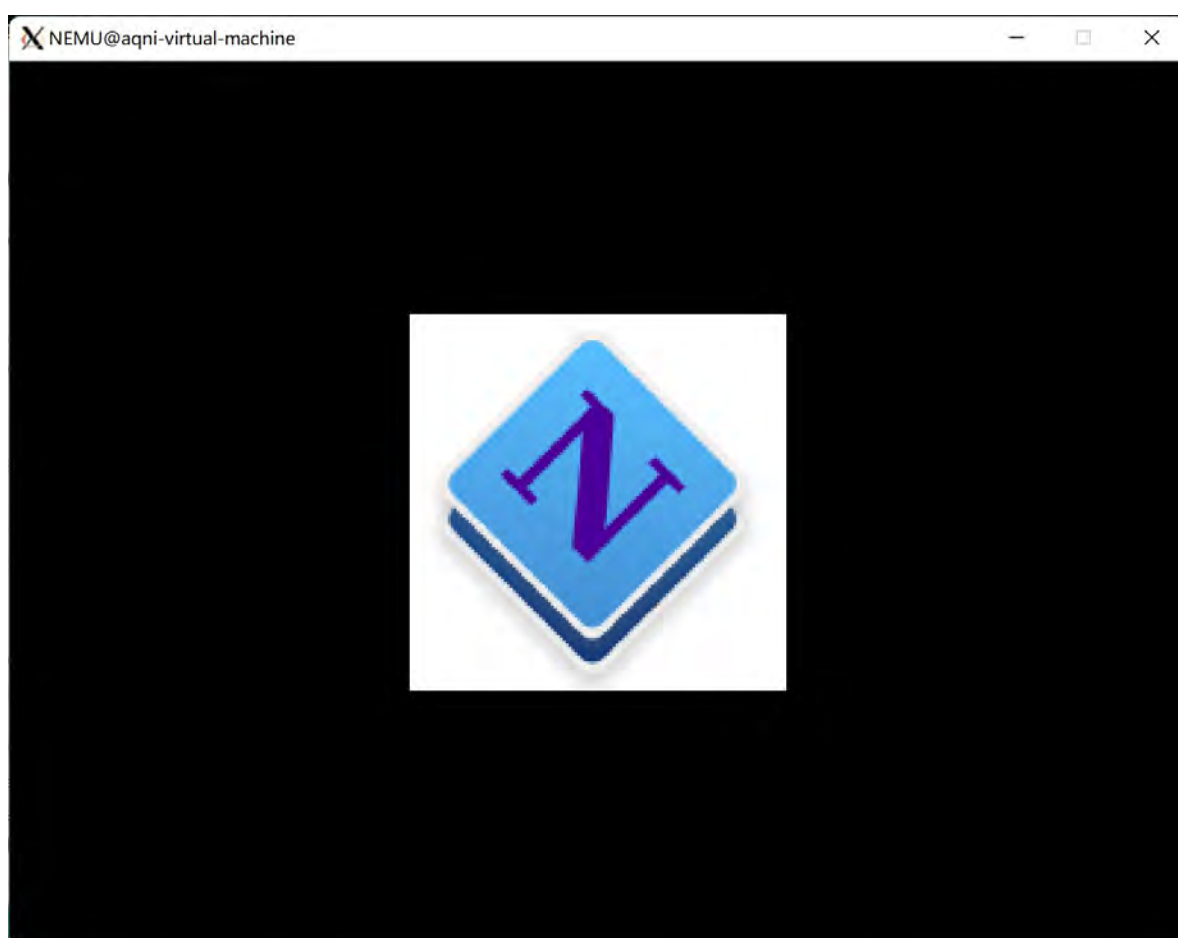


图 6: 显示 logo

Listing 12: init_device 的疏忽

```
fprintf(fp, "WIDTH: %d\n", disp_w);
fprintf(fp, "HEIGHT: %d\n", disp_h);

sprintf(dispinfo, "WIDTH: %d\n", _screen.width);
sprintf(dispinfo, "HEIGHT: %d\n", _screen.height);

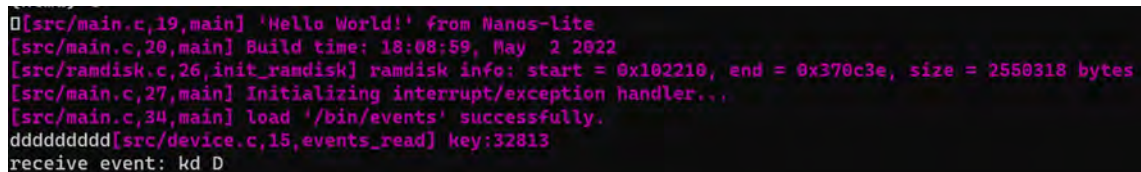
sprintf(dispinfo, "WIDTH: %d\nHEIGHT: %d\n", _screen.width, _screen.
    height);
```

4.2 把设备输入抽象成文件

分别处理按下和释放事件即可（代码段13），最终可成功运行/bin/events(图??)。

Listing 13: events_read 实现

```
extern int _read_key();
size_t events_read(void *buf, size_t len) {
    // int key_code;
    int keyinput=_read_key();
    Log("key:%d",keyinput);
    if(keyinput == _KEY_NONE){
        // time event
        snprintf(buf, len, "t %d\n", _uptime());
    }else if(keyinput & 0x8000){
        // key down event
        keyinput &= ~0x8000;
        snprintf(buf, len, "kd %s\n", keyname[keyinput]);
    }else{
        // key up event
        snprintf(buf, len, "ku %s\n", keyname[keyinput]);
    }
    return strlen(buf);
}
```



```
0[src/main.c,19,main] 'Hello World!' from Nanos-lite
[src/main.c,20,main] Build time: 18:08:59, May  2 2022
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x102210, end = 0x370c3e, size = 2550318 bytes
[src/main.c,27,main] Initializing interrupt/exception handler...
[src/main.c,34,main] load '/bin/events' successfully.
ddddddddd[src/device.c,15,events_read] key:32813
receive event: kd D
```

图 7: 成功运行 events

4.3 在 NEMU 中运行仙剑奇侠传

这里需要注意将 `diff_test` 和 `debug` 关闭，否则运行非常慢。经过测试，当前实现可以正常进入游戏(图8)

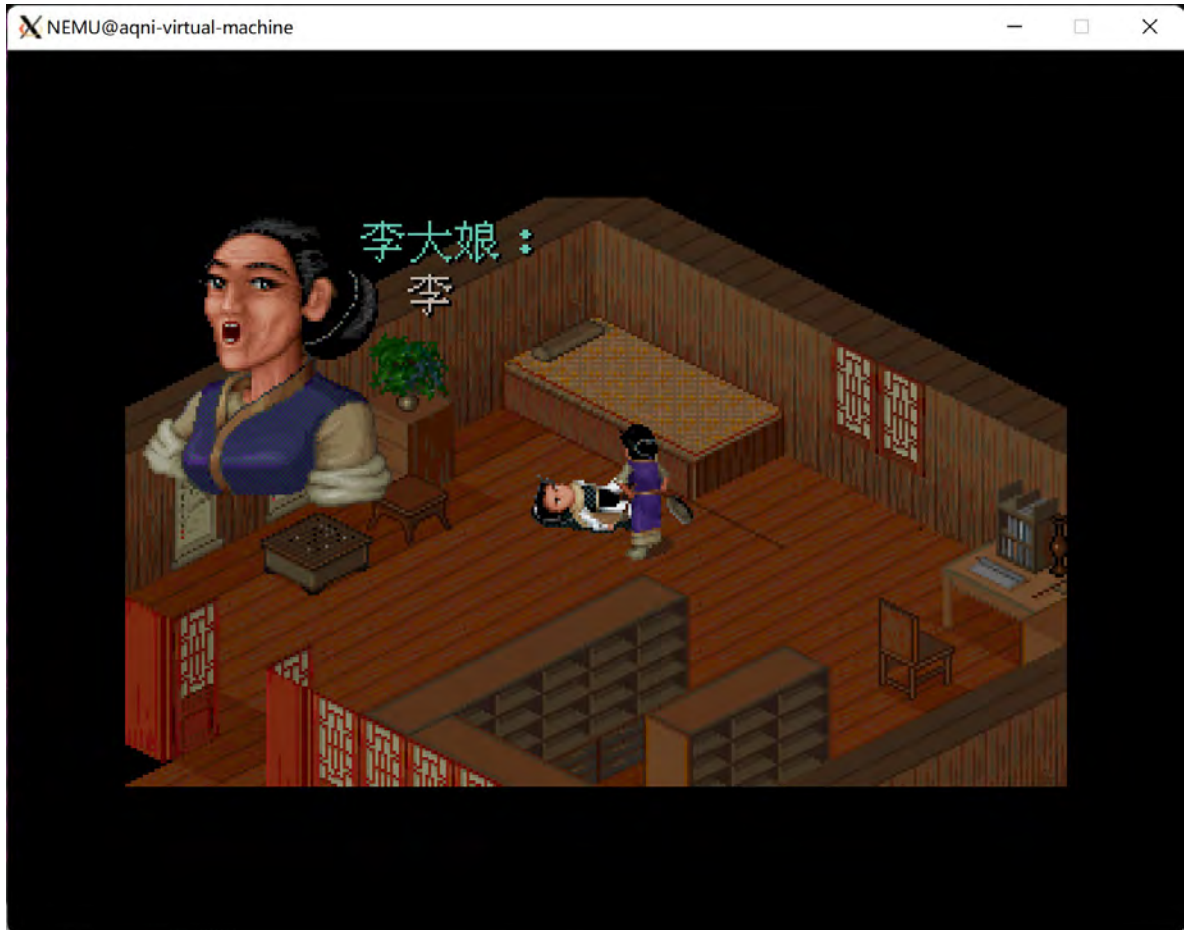


图 8: 成功运行仙剑奇侠传

5 bug 汇总

5.1 PUSHASH 和 POPA 顺序

在实现 PUSHASH 和 POPA 这个地方时，非常容易粗心导致错误，需要仔细检查。

5.2 FD_STDOUT 文件需要单独处理

在 `main.c` 中修改 `loader` 的传入参数即可切换加载的文件。此外，当时忘记在 `nanos.c` 中实现相应系统调用函数，忘记返回值，忘记 `FD_STDOUT` 的长度是 0，这些均导致程序未能通过，最终 `/bin/text` 成功运行(图9)。

```
0[src/main.c,19,main] 'Hello World!' from Nanos-lite
[src/main.c,20,main] Build time: 19:15:35, May 1 2022
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x101910, end = 0x37033e, size = 2550318 bytes
[src/main.c,27,main] Initializing interrupt/exception handler...
[src/main.c,34,main] load '/bin/text' success.
PASS!!!
nemu: HIT GOOD TRAP at eip = 0x00100032
```

图 9: /bin/text 程序通过

5.3 显存按照 uint32 对齐

实验中，实现 fb_write 函数时，忘记 offset 和 len 是按照字节对齐的，导致无法正常显示图标。

Listing 14: 初始化/dev/fb 大小

```
void fb_write(const void *buf, off_t offset, size_t len) {
    offset /= sizeof(uint32_t);
    len /= sizeof(uint32_t);
    assert(offset < _screen.width * _screen.height);
    int x=offset%_screen.width,y=offset/_screen.width;
    _draw_rect(buf,x,y,len,1);
}
```

6 必答题

请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。

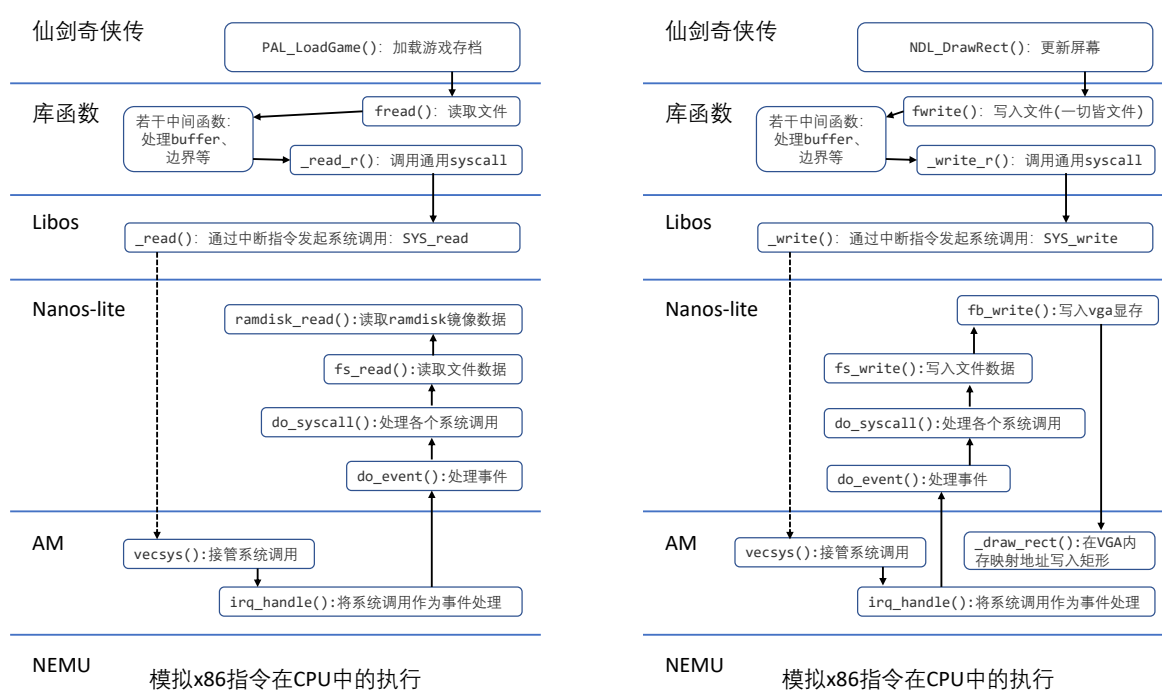
这里不使用大段文字来描述他们之间的配合，直接上图。图10梳理了读取游戏存档（图）和更新屏幕过程中仙剑奇侠传，库函数，libos, Nanos-lite, AM, NEMU 之间的相互配合。

7 体会与感悟

相较前两次实验，PA3 的难度不高，但是容易犯一些低级错误。

参考文献

- [1] YU Z. Ics2017 programming assignment[EB/OL]. [March 22, 2022]. <https://nju-ics.gitbooks.io/ics2017-programming-assignment/>.



(a) 读取游戏存档

(b) 更新屏幕

图 10: 结合代码, 梳理各组件之间的配合 (实现箭头表示函数调用, 虚线箭头表示异常处理)