

# 南京航空航天大学《计算机组成原理II课程设计》报告

---

- 姓名：傅锦龙
- 班级：1621301
- 学号：162130117
- 报告阶段：PA3.1
- 完成日期：2023.6.18
- 本次实验，我完成了所有内容。

## 目录

---

### 南京航空航天大学《计算机组成原理II课程设计》报告

#### 目录

#### 思考题

- 什么是操作系统？（5）
- 我们不一样吗？（5）
- 操作系统的实质（5）
- 程序真的结束了吗？（10）
- 触发系统调用（10）
- 有什么不同？（5）
- 段错误（10）
- 对比异常与函数调用（5）
- 诡异的代码（5）
- 注意区分事件号和系统调用号（5）
- 打印不出来？（5）
- 理解文件管理函数（15）
- 不再神秘的秘技（5）
- 必答题（5）
- git log和git branch截图（5）

#### 实验内容

- 实现 loader
- 添加寄存器和 LIDT 指令
- 实现 INT 指令
- 实现其他相关指令和结构体
- 完善事件分发和 do\_syscall
- 实现堆区管理
- 实现系统调用
- 成功运行各测试用例

#### 遇到的问题及解决办法

#### 实验心得

#### 其他备注

## 思考题

---

### 什么是操作系统？（5）

操作系统是计算机系统中最基本的软件之一，它为上层应用程序提供了一个抽象的计算机模型，隐藏了底层硬件的复杂性，使得应用程序能够更加方便、高效地使用计算机资源。

## 我们不一样吗？（5）

在PA2中，我们将cputest和部分设备测试的程序直接运行在AM之上，而在Nanos-lite中，我们将操作系统运行在AM之上。因此，Nanos-lite相当于是一个完整的操作系统，它提供了更多的功能和服务，例如进程管理、内存管理、文件系统管理、设备管理等。而在PA2中，我们只是简单地运行了一些测试程序，没有提供这些功能和服务。

我们可以将Nanos-lite看作是一个和PA2中这些测试用例同地位的一个AM程序。

## 操作系统的实质（5）

操作系统就是一个较为大型的软件，它和直接运行在硬件上的程序无实质差别。

## 程序真的结束了吗？（10）

在Nanos-lite中，程序的入口是\_start()函数，它会执行一些初始化工作，例如初始化内存管理、文件系统等。然后，它会调用main()函数，等待main()函数执行结束。在main()函数执行结束后，程序会执行一些清理工作，例如释放内存、关闭文件等。最后，程序会调用exit()函数，将控制权返回给操作系统。因此，当程序运行到main()函数的return 0;语句时，main()函数执行结束，但程序并不会立即结束。程序还需要执行一些清理工作，并将控制权返回给操作系统。

## 触发系统调用（10）

代码如下

```
const char str[] = "Hello world!\n";

int main()
{
    asm volatile("movl $4, %eax;"      // system call ID, 4 = SYS_write
                  "movl $1, %ebx;"     // file descriptor, 1 = stdout
                  "movl $str, %ecx;"   // buffer address
                  "movl $13, %edx;"    // length
                  "int $0x80");

    return 0;
}
```

运行结果

```
fujinlong@ubuntu:/mnt/hgfs/course/ics2023$ gcc hello.c -o hello
fujinlong@ubuntu:/mnt/hgfs/course/ics2023$ ./hello
Hello world!
fujinlong@ubuntu:/mnt/hgfs/course/ics2023$ █
```

## 有什么不同？（5）

系统调用的过程和函数调用的过程非常相似。在函数调用时，也会将一些寄存器的值保存到栈中，以便在函数返回时能够恢复这些寄存器的值。类似地，在系统调用时，也会将一些寄存器的值保存到栈中，以便在系统调用返回时能够恢复这些寄存器的值。

我们可以将系统调用的服务程序理解为一个比较特殊的“函数”，因为它们也接受参数、执行一些操作，并返回结果。但是，系统调用的服务程序 and 用户编写的函数有一些不同之处。具体来说，系统调用的服务程序是由操作系统内核提供的，它们运行在内核态，可以访问系统资源，例如硬件设备、内存等。而用户编写的函数运行在用户态，只能访问用户空间的资源，不能直接访问系统资源。

此外，系统调用的服务程序通常比用户编写的函数更加复杂和底层，因为它们需要处理系统资源的管理

和调度等问题。系统调用的服务程序还需要进行一些安全检查，以确保用户程序不能直接访问内核态的资源，从而保证系统的安全性和稳定性。

## 段错误 (10)

在编译阶段，编译器只能检查语法错误和一些静态错误，例如类型不匹配、未声明的变量等。而对于一些动态错误，例如数组越界、空指针引用等，编译器是无法检查出来的。这些错误只有在程序运行时才会暴露出来。

“Segmentation Fault”（段错误）通常是由一些非法的内存访问行为引起的。例如，当程序试图访问一个未分配的内存区域、访问已经释放的内存区域、访问数组越界、使用空指针等，就会导致段错误的发生。这些行为都会导致程序访问了不属于它的内存区域，从而触发了操作系统的内存保护机制，导致程序崩溃。

## 对比异常与函数调用 (5)

异常保存寄存器、错误码#irq、EFLAGS、CS、EIP和各种通用寄存器

函数调用保存返回地址和调用约定中需要调用者保存的寄存器

异常处理是目的是为了处理一些不可预见的异常情况，为了能够正确地处理异常，需要保存更多的状态信息，以便在异常处理程序执行完毕后能够恢复现场，而函数调用是程序正常的执行过程，所以不需要保存太多

## 诡异的代码 (5)

在 trap.S 中，pushl %esp 的作用是将当前的栈指针压入栈中。这是为了在异常处理程序中能够访问到当前的栈指针，以便在恢复现场时能够正确地恢复栈的状态。

## 注意区分事件号和系统调用号 (5)

事件号通常用于中断处理程序中，用于标识当前发生的中断类型。例如，在处理键盘中断时，可以使用事件号来区分不同的按键事件。事件号通常是由硬件设备产生的，用于通知 CPU 发生了什么事情。而系统调用号则是用于用户程序和操作系统内核之间进行通信的。用户程序可以通过系统调用来请求操作系统内核提供一些服务，例如打开文件、读写文件等。系统调用号用于标识用户程序请求的服务类型，操作系统内核根据系统调用号来确定需要执行哪个系统调用服务程序。

## 打印不出来? (5)

解决方案:

```
#include <stdio.h>
int main(){
    int *p = NULL;
    printf("I am here!\n");//加上换行
    *p = 10;
    return 0;
}
```

printf打印是行缓冲，直到一行结束或者整个程序结束，才输出到屏幕，因为我们打印的字符串一行没有结束，所以就先执行后面的\*p=10报错了。因此，我们要让他输出字符串内容，只需要在字符串后面加上\n就表明一行结束，可以输出了。

## 理解文件管理函数 (15)

- fs\_open()用文件名参数到文件描述符表中匹配并返回下标,没找到则报错
- fs\_read()通过fd参数获取文件偏移和长度,再从ramdisk或dispinfo中读取数据到buf中

- fs\_write()通过fd选择写方式.若是文件写,则计算偏移和读取长度进行文件读写
- fs\_lseek()通过whence选择移动文件偏移的方式,然后将新的偏移赋给对应文件描述符
- fs\_close()直接返回0,因为不需要close

## 不再神秘的秘技 (5)

游戏bug, 开发时没有注意变量类型等问题, 造成在某些特定情况下, 会出现溢出现象。

## 必答题 (5)

在仙剑奇侠传中, 文件读写和屏幕更新是通过库函数、libos、Nanos-lite、AM、NEMU 相互协作来完成的。具体来说:

1. 游戏存档的读取过程:

- 在 PAL\_LoadGame() 函数中, 通过调用 fread() 函数从文件中读取游戏存档数据。
- 在 Nanos-lite 操作系统中, 文件读取接口是通过系统调用来实现的。当 PAL\_LoadGame() 函数调用 fread() 函数时, libos 会将该系统调用转换为一个中断, 并将中断传递给 Nanos-lite 操作系统, 中断处理程序会根据系统调用的参数来调用文件系统模块中的相应函数, 实现文件读取操作。
- 文件系统模块中的函数会将读取到的数据从磁盘中读取到内存中, 并将数据返回给中断处理程序, 中断处理程序将读取到的数据返回给 libos, libos 再将数据返回给 PAL\_LoadGame() 函数, 完成文件读取操作。

2. 屏幕更新的过程:

- 在 redraw() 函数中, 通过调用 NDL\_DrawRect() 函数来更新屏幕。
- 在 Nanos-lite 操作系统中, 屏幕更新接口是通过系统调用来实现的。当 NDL\_DrawRect() 函数调用屏幕更新接口时, libos 会将该系统调用转换为一个中断, 并将中断传递给 Nanos-lite 操作系统, 中断处理程序会根据系统调用的参数来调用 AM 库中的相应函数, 实现屏幕更新操作。
- AM 库中的函数会将更新后的屏幕数据传递给 Nanos-lite 操作系统, 然后由操作系统将数据传递给 NEMU 模拟器, NEMU 模拟器会将屏幕数据显示在屏幕上, 完成屏幕更新操作。

## git log和git branch截图 (5)

```
fujinlong@ubuntu:/mnt/hgfs/course/ics2023$ git log
commit 0ce3a697e4521f6b926fae2e6c3c288a019bee63
Author: tracer-ics2017 <tracer@njuics.org>
Date:   Fri Jun 16 22:49:32 2023 +0800

    > run
    162130117
    fujinlong
    Linux ubuntu 4.15.0-142-generic #146~16.04.1-Ubuntu SMP Tue Apr 13 09:26:57
    UTC 2021 i686 athlon i686 GNU/Linux
    22:49:30 up 13:29, 1 user, load average: 0.50, 0.16, 0.05
    c0ea9fcf99e1555d73bafa431c91ad32e3f60a4

fujinlong@ubuntu:/mnt/hgfs/course/ics2023$ git branch
master
pa0
pa2
* pa3
```

## 实验内容

### 实现 loader

根据讲义ramdisk\_read第一个参数为DEFAULT\_ENTRY, 第二个参数偏移量为 0, 第三个参数是ramdisk的大小, 可以用get\_ramdisk\_size函数获取, 完成loader函数



```
static inline void restart()
{
    /* Set the initial instruction pointer. */
    cpu.eflags.val = 0x2;
    cpu.max_instr_cnt = 1000000;
    cpu.eip = ENTRY_START;
    cpu.cs = 0x8;
#ifdef DIFF_TEST
    init_qemu_reg();
#endif
}
```

- LIDT 指令细节可在 i386 手册中找到

在第七个指令组找到LIDT指令，补全操作码表

```
/* 0x0f 0x01*/
make_group(gp7,
    EMPTY, EMPTY, EMPTY, EX(lidt),
    EMPTY, EMPTY, EMPTY, EMPTY)
```

若OperandSize是16，则limit读取16位，base读取24位

若OperandSize是32，则limit读取16位，base读取32位

执行函数实现如下

```
make_EHelper(lidt)
{
    cpu.idtr.limit = vaddr_read(id_dest->addr, 2);
    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);

    print_asm_template1(lidt);
}
```

## 实现 INT 指令

- 实现写在 raise\_intr() 函数中

触发异常后硬件的处理如下：

1. 依次将EFLAGS, CS(代码段寄存器),EIP寄存器的值压栈
2. 从IDTR中读出IDT的首地址
3. 根据异常号在IDT中进行索引, 找到一个门描述符
4. 将门描述符中的offset域组合成目标地址

在nemu/src/cpu/intr.c里实现raise\_intr函数如下

```
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.
     */

    vaddr_t gate_addr = cpu.idtr.base + 8 * NO;
```

```

    if (cpu.idtr.limit < 0) {
        assert(0);
    }

    rtl_push(&cpu.eflags.val);
    rtl_push(&cpu.cs);
    rtl_push(&ret_addr);

    uint32_t high, low;
    low = vaddr_read(gate_addr, 4) & 0xffff;
    high = vaddr_read(gate_addr + 4, 4) & 0xffff0000;

    decoding.jump_eip = high | low;
    decoding.is_jump = true;
}

```

- 使用 INT 的 helper 函数调用 raise\_intr()

```

make_EHelper(int)
{
    raise_intr(id_dest->val, decoding.seq_eip);
    print_asm("int %s", id_dest->str);

#ifdef DIFF_TEST
    diff_test_skip_nemu();
#endif
}

```

- 指令细节可在 i386 手册中找到

补全操作码表

```

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


```

## 运行结果

```
[src/monitor/monitor.c,33,welcome] Build time: 11:24:43, Jun 16 2023
For help, type "help"
(nemu) c
[0] [src/main.c,19,main] 'Hello World!' from Nanos-lite
[src/main.c,20,main] Build time: 11:24:31, Jun 16 2023
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x101de0, end = 0x1063bc,
[src/main.c,27,main] Initializing interrupt/exception handler...
invalid opcode(eip = 0x001012a5): 60 54 e8 0f fd ff ff 83 ...

There are two cases which will trigger this unexpected exception:
1. The instruction at eip = 0x001012a5 is not implemented.
2. Something is implemented incorrectly.
Find this eip(0x001012a5) in the disassembling result to distinguish which case

If it is the first case, see
```



```
Warning!
```

```
for more details.

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

## 实现其他相关指令和结构体

- 组织\_RegSet 结构体, 需要说明理由

现场保存的顺序为：①硬件保存 EFLAGS, CS, EIP ②vecsys() 会压入错误码和异常号 #irq ③ asm\_trap() 会把用户进程的通用寄存器保存到堆栈上，则恢复的时候倒序恢复，所以 \_RegSet 的组织方式如下

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

- pusha, popa, iret

### 在操作码表补全指令

```
/* 0x60 */ EX(pusha), EX(popa), EMPTY, EMPTY,
...
/* 0xcc */ EMPTY, IDEXW(I,int,1), EMPTY, EX(iret),
```

pusha, popa指令实现函数如下

```
make_EHHelper(pusha)
{
    rtlreg_t tmp, resp;
    if (decoding.is_operand_size_16) {
        rtl_lr_w(&resp, R_SP);
        rtl_lr_w(&tmp, R_AX);
        rtl_push(&tmp);
        rtl_lr_w(&tmp, R_CX);
        rtl_push(&tmp);
        rtl_lr_w(&tmp, R_DX);
    }
}
```



```

        rtl_push(&tmp);
        rtl_lr_w(&tmp, R_BX);
        rtl_push(&tmp);
        rtl_push(&resp);
        rtl_lr_w(&tmp, R_BP);
        rtl_push(&tmp);
        rtl_lr_w(&tmp, R_SI);
        rtl_push(&tmp);
        rtl_lr_w(&tmp, R_DI);
        rtl_push(&tmp);

    } else {
        rtl_lr_l(&resp, R_ESP);
        rtl_lr_l(&tmp, R_EAX);
        rtl_push(&tmp);
        rtl_lr_l(&tmp, R_ECX);
        rtl_push(&tmp);
        rtl_lr_l(&tmp, R_EDX);
        rtl_push(&tmp);
        rtl_lr_l(&tmp, R_EBX);
        rtl_push(&tmp);
        rtl_push(&resp);
        rtl_lr_l(&tmp, R_EBP);
        rtl_push(&tmp);
        rtl_lr_l(&tmp, R_ESI);
        rtl_push(&tmp);
        rtl_lr_l(&tmp, R_EDI);
        rtl_push(&tmp);
    }
    print_asm("pusha");
}

make_EHelper(popa)
{
    rtlreg_t tmp;
    if (decoding.is_operand_size_16) {
        rtl_pop(&tmp);
        rtl_sr_w(R_DI, &tmp);
        rtl_pop(&tmp);
        rtl_sr_w(R_SI, &tmp);
        rtl_pop(&tmp);
        rtl_sr_w(R_BP, &tmp);
        rtl_pop(&tmp);
        rtl_pop(&tmp);
        rtl_sr_w(R_BX, &tmp);
        rtl_pop(&tmp);
        rtl_sr_w(R_DX, &tmp);
        rtl_pop(&tmp);
        rtl_sr_w(R_CX, &tmp);
        rtl_pop(&tmp);
        rtl_sr_w(R_AX, &tmp);
    } else {
        rtl_pop(&tmp);
        rtl_sr_l(R_EDI, &tmp);
        rtl_pop(&tmp);
        rtl_sr_l(R_ESI, &tmp);
        rtl_pop(&tmp);
        rtl_sr_l(R_EBP, &tmp);
    }
}

```

```

        rtl_pop(&tmp);
        rtl_pop(&tmp);
        rtl_sr_l(R_EBX, &tmp);
        rtl_pop(&tmp);
        rtl_sr_l(R_EDX, &tmp);
        rtl_pop(&tmp);
        rtl_sr_l(R_ECX, &tmp);
        rtl_pop(&tmp);
        rtl_sr_l(R_EAX, &tmp);
    }

    print_asm("popa");
}

```

iret 指令将栈顶的三个元素依次解释成 EIP, CS, EFLAGS, 并恢复它们, 补全iret函数如下

```

make_EHelper(iret)
{
    rtl_pop(&decoding.jump_eip);
    decoding.is_jump = 1;
    rtl_pop(&cpu.cs);
    rtl_pop(&cpu.eflags.val);
    print_asm("iret");
}

```

运行结果

```

[src/monitor/monitor.c,68,load_img] The image is /mnt/hgfs/course/ics20
Welcome to NEMU!
[src/monitor/monitor.c,33,welcome] Build time: 11:24:43, Jun 16 2023
For help, type "help"
(nemu) c
[src/main.c,19,main] 'Hello World!' from Nanos-lite
[src/main.c,20,main] Build time: 14:14:25, Jun 16 2023
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x101de0, end = 0
[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 = 0x001000f1

```

## 完善事件分发和 do\_syscall

- 完善 do\_event, 目前阶段仅需要识别出系统调用事件即可  
识别系统调用事件 \_EVENT\_SYSCALL, 然后调用 do\_syscall()

```

_RegSet *do_syscall(_RegSet *r);
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;
}

```

- 添加整个阶段中的所有系统调用

实现SYSCALL\_ARGx(r)宏

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

添加 SYS\_none 系统调用和SYS\_exit系统调用

```
_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);
    switch (a[0]) {
        case SYS_none:
            r->eax = 1;
            break;
        case SYS_exit:
            _halt(a[1]);
            break;
        default:
            panic("Unhandled syscall ID = %d", a[0]);
    }
    return NULL;
}
```

dummy 运行结果

```
[src/monitor/monitor.c,68,load_img] The image is /mnt/hgfs/course/ics.
lite/build/nanos-lite-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,33,welcome] Build time: 11:24:43, Jun 16 2023
For help, type "help"
(nemu) c
[src/main.c,19,main] 'Hello World!' from Nanos-lite
[src/main.c,20,main] Build time: 15:01:43, Jun 16 2023
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x101ee0, end =
size = 17884 bytes
[src/main.c,27,main] Initializing interrupt/exception handler...
nemu: HIT GOOD TRAP at eip = 0x001000f1
```

- 实现 write() 系统调用

由讲义可知，在 do\_syscall() 中识别出系统调用号是 SYS\_write 之后，检查 fd 的值，如果 fd 是 1 或 2（分别代表 stdout 和 stderr），则将 buf 为首地址的 len 字节输出到串口（使用 \_putc() 即可；通过 man 2 write 可知，要返回写的字符的 bytes）。由以上两点可知，fs\_write() 符合上述要求，补全 sys\_write() 如下

```
static inline uintptr_t sys_write(uintptr_t fd, uintptr_t buf, uintptr_t len)
{
    return fs_write(fd, (void *)buf, len);
}
```

do\_syscall函数添加如下内容

```
case SYS_write:
    r->eax = sys_write(a[1], a[2], a[3]);
    break;
```

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);
}
```

运行后缺失指令f6 46test,填表如下

```
/* 0xf6, 0xf7 */
make_group(gp3,
    IDEX(test_I, test), EMPTY, EX(not), EX(neg),
    EX(mul), EX(imul), EX(div), EX(idiv))
```

缺失指令a8 test,填表如下

```
/* 0xa8 */ IDEXW(I2a,test,1), IDEX(I2a,test), EMPTY, EMPTY,
```

hello运行结果

```
[src/monitor/monitor.c,33,welcome] Build time: 11:24:43, Jun 16 2023
For help, type "help"
(nemu) c
[0] [src/main.c,19,main] 'Hello World!' from Nanos-lite
[0] [src/main.c,20,main] Build time: 15:01:43, Jun 16 2023
[0] [src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x101f20, end = 0x101f20, size = 18140 bytes
[0] [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
Hello World for the 11th time
Hello World for the 12th time
```

## 实现堆区管理

- 堆区管理相关系统调用和封装函数，如 sys\_brk, mm\_brk, \_sbrk(), brk(), sbrk()

由于目前 Nanos-lite 还是一个单任务操作系统，空闲的内存都可以让用户程序自由使用，因此我们只需要让 SYS\_brk 系统调用总是返回 0 即可，表示堆区大小的调整总是成功。

```
case SYS_brk:
    r->eax = 0;
    break;
```

\_sbrk() 通过记录的方式来对用户程序的 program break 位置进行管理，其工作方式如下：

1. program break 一开始的位置位于 `_end`
2. 被调用时, 根据记录的 program break 位置和参数 increment, 计算出新 program break
3. 通过 `SYS_brk` 系统调用来让操作系统设置新 program break
4. 若 `SYS_brk` 系统调用成功, 该系统调用会返回 0, 此时更新之前记录的 program break 的位置, 并将旧 program break 的位置作为 `_sbrk()` 的返回值返回
5. 若该系统调用失败, `_sbrk()` 会返回 -1

```
extern char _end;
static uintptr_t program_break = (uintptr_t)&_end;
void *_sbrk(intptr_t increment){
    uintptr_t old = program_break;
    if (_syscall_(SYS_brk, old + increment, 0, 0) == 0) {
        program_break = old + increment;
        return (void *)old;
    }
    return (void *)-1;
}
```

- 让 loader 使用文件

使用文件读写函数打开要让 loader 读取的文件, 并读取相应大小的内容到指定内存地址上, 读取完毕后关闭文件, 修改 `nanos-lite/src/loader.c` 实现 loader

```
uintptr_t loader(_Protect *as, const char *filename)
{
    int fd = fs_open(filename, 0, 0);
    int size = fs_filesz(fd);
    fs_read(fd, DEFAULT_ENTRY, size);
    fs_close(fd);
    return (uintptr_t)DEFAULT_ENTRY;
}
```

以后更换用户程序只需要修改传入 `loader()` 函数的文件名即可, 无需更新 ramdisk 的内容, 修改 `nanos-lite/src/main.c` 中 `loader` 函数的参数:

```
uint32_t entry = loader(NULL, "/bin/text");
```

## 实现系统调用

- 实现 `sys_[open|write|read|lseek|close|brk]` 函数

```
static inline uintptr_t sys_open(uintptr_t pathname, uintptr_t flags, uintptr_t mode)
{
    return fs_open((char *)pathname, flags, mode);
}

static inline uintptr_t sys_write(uintptr_t fd, uintptr_t buf, uintptr_t len)
{
    return fs_write(fd, (void *)buf, len);
}

static inline uintptr_t sys_read(uintptr_t fd, uintptr_t buf, uintptr_t len)
{
    return fs_read(fd, buf, len);
}
```

```

        return fs_read(fd, (void *)buf, len);
    }

    static inline uintptr_t sys_lseek(uintptr_t fd, uintptr_t offset, uintptr_t
whence)
    {
        return fs_lseek(fd, offset, whence);
    }

    static inline uintptr_t sys_close(uintptr_t fd)
    {
        return fs_close(fd);
    }

    static inline uintptr_t sys_brk(uintptr_t new_brk)
    {
        return mm_brk(new_brk);
    }

    _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);
        switch (a[0]) {
            case SYS_none:
                r->eax = 1;
                break;
            case SYS_exit:
                _halt(a[1]);
                break;
            case SYS_write:
                r->eax = sys_write(a[1], a[2], a[3]);
                break;
            case SYS_brk:
                r->eax = 0;
                break;
            case SYS_open:
                r->eax = sys_open(a[1], a[2], a[3]);
                break;
            case SYS_read:
                r->eax = sys_read(a[1], a[2], a[3]);
                break;
            case SYS_lseek:
                r->eax = sys_lseek(a[1], a[2], a[3]);
                break;
            case SYS_close:
                r->eax = sys_close(a[1]);
                break;
            default:
                panic("Unhandled syscall ID = %d", a[0]);
        }

        return NULL;
    }
}

```

修改navy-apps/libs/libos/src/nanos.c中的相应接口函数

```
int _open(const char *path, int flags, mode_t mode) {
    return _syscall_(SYS_open, (uintptr_t)path, flags, mode);
}

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

int _read(int fd, void *buf, size_t count) {
    return _syscall_(SYS_read, fd, (uintptr_t)buf, count);
}

int _close(int fd) {
    return _syscall_(SYS_close, fd, 0, 0);
}

off_t _lseek(int fd, off_t offset, int whence) {
    return _syscall_(SYS_lseek, fd, offset, whence);
}
```

/bin/text运行结果

```
[src/monitor/monitor.c,68,load_img] The image is /mnt/hgfs/course/ics2023/
lite/build/nanos-lite-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,33,welcome] Build time: 11:24:43, Jun 16 2023
For help, type "help"
(nemu) c
00000000 [src/main.c,19,main] 'Hello World!' from Nanos-lite
00000000 [src/main.c,20,main] Build time: 20:38:15, Jun 16 2023
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x102460, end =
size = 2550226 bytes
[src/main.c,27,main] Initializing interrupt/exception handler...
PASS!!!
nemu: HIT GOOD TRAP at eip = 0x001000f1
```

## 成功运行各测试用例

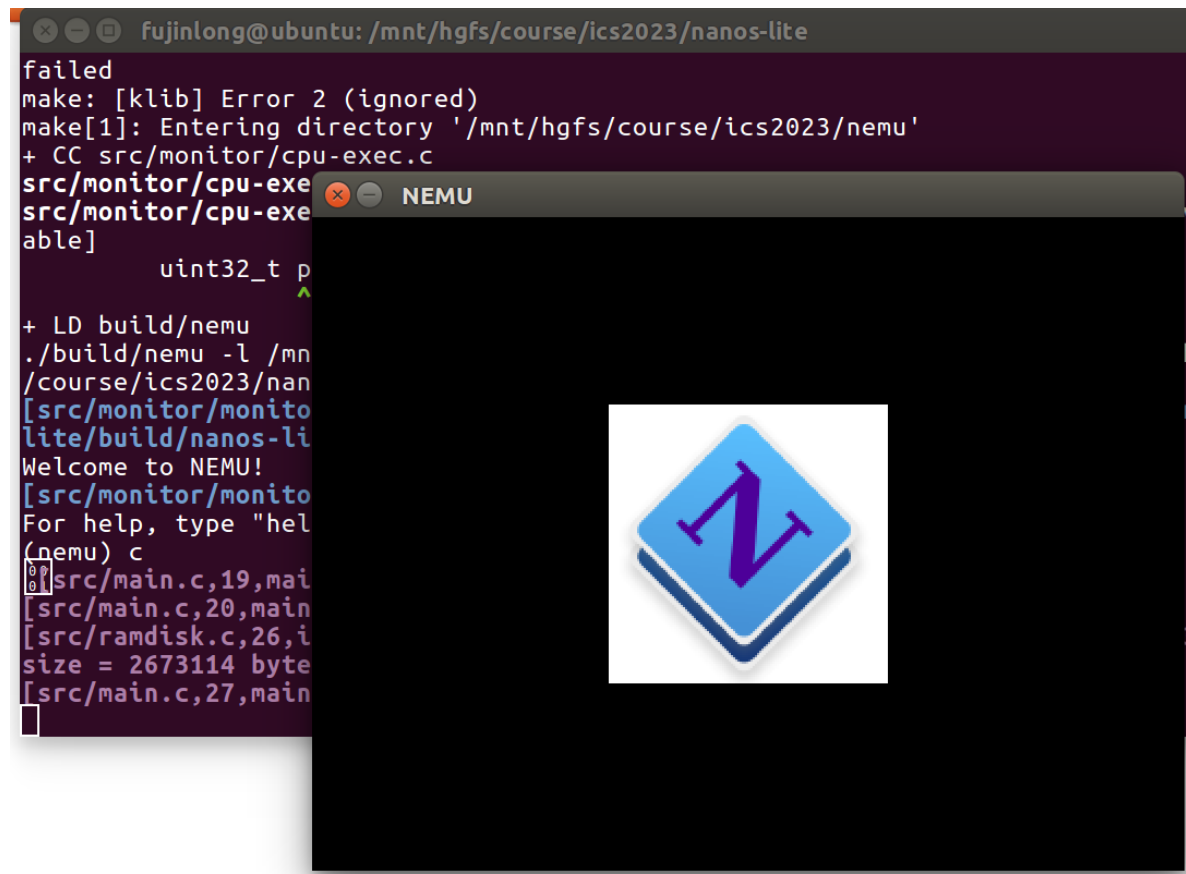
- Hello world

```
[src/monitor/monitor.c,33,welcome] Build time: 11:24:43, Jun 16 2023
For help, type "help"
(nemu) c
00000000 [src/main.c,19,main] 'Hello World!' from Nanos-lite
00000000 [src/main.c,20,main] Build time: 15:01:43, Jun 16 2023
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x101f20, end =
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
Hello World for the 11th time
Hello World for the 12th time
```

- /bin/text

```
[src/monitor/monitor.c,68,load_img] The image is /mnt/hgfs/course/ics2023/nanos-lite/build/nanos-lite-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,33,welcome] Build time: 11:24:43, Jun 16 2023
For help, type "help"
(nemu) c
00[src/main.c,19,main] 'Hello World!' from Nanos-lite
[src/main.c,20,main] Build time: 20:38:15, Jun 16 2023
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x102460, end =
size = 2550226 bytes
[src/main.c,27,main] Initializing interrupt/exception handler...
PASS!!!
nemu: HIT GOOD TRAP at eip = 0x001000f1
```

- /bin/bmptest



- /bin/events

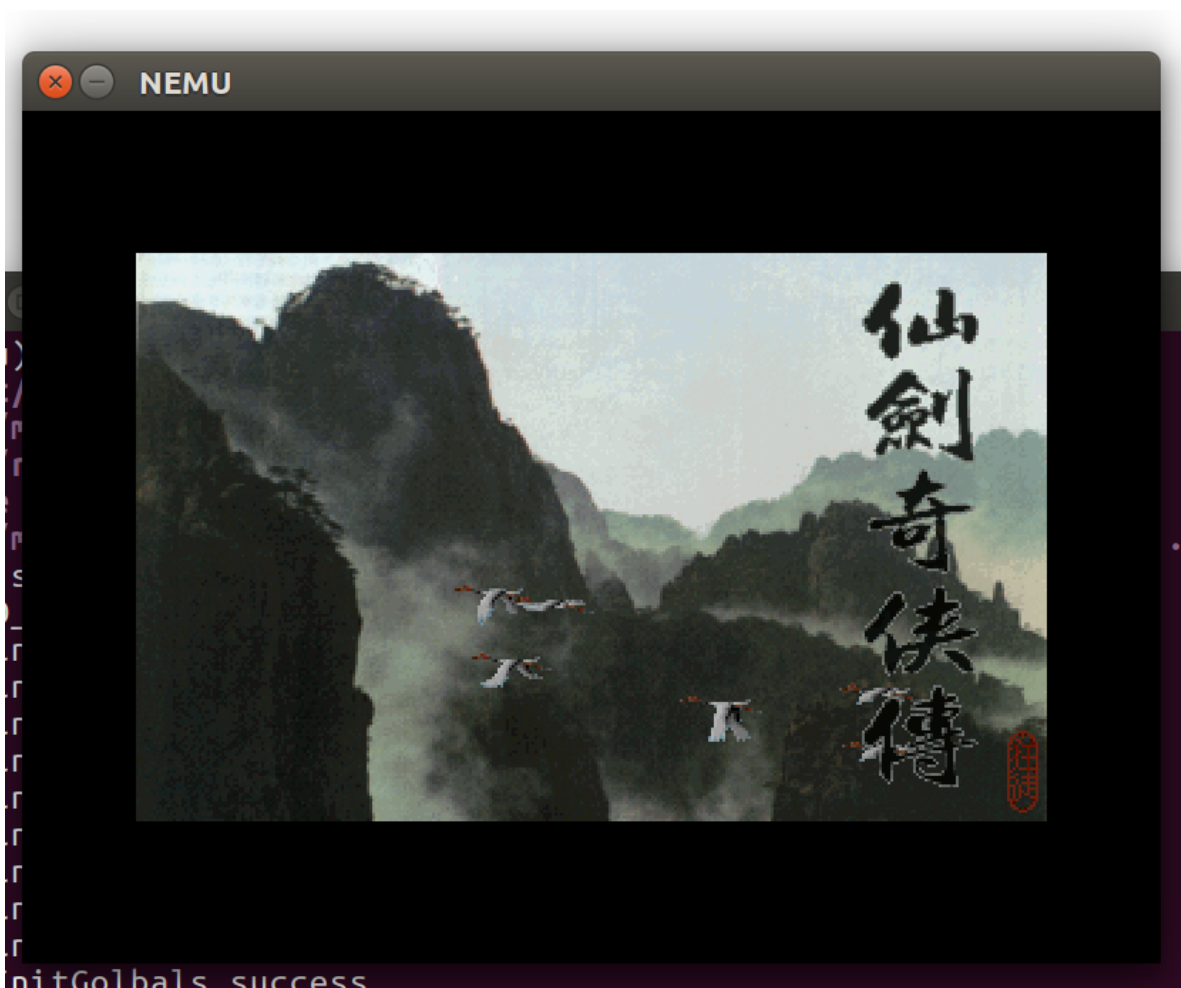


```

[src/monitor/monitor.c,68,load_img] The image is /mnt/hgfs/course/ics2
lite/build/nanos-lite-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,33,welcome] Build time: 21:19:32, Jun 16 2023
For help, type "help"
(nemu) c
[0001]src/main.c,19,main] 'Hello World!' from Nanos-lite
[src/main.c,20,main] Build time: 21:30:07, Jun 16 2023
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x102460, end =
size = 2673114 bytes
[src/main.c,27,main] Initializing interrupt/exception handler...
receive event: t 270
receive event: t 508
receive event: t 782
receive event: t 1010
receive event: kd D
receive event: kd F
receive event: ku D
receive event: kd D
receive event: ku F
receive event: kd F
receive event: ku D
receive event: ku F
receive event: kd E

```

- 仙剑奇侠传



## 遇到的问题及解决办法

1. 遇到问题：实现堆区管理时，printf()一直是按字符输出  
解决方案：全局make clean，然后重新编译

2. 遇到问题：保存现场时通用寄存器保存出现错误,超出物理地址  
解决方案：pusha和rtl\_push都使用了t0作为临时保存变量，导致入栈错误

## 实验心得

---

经过本次的实验，我学会了编写底层的逻辑代码，理清了计算机的异常和系统调用，对于堆区管理和文件系统有了一定的了解，这对于我以后学习操作系统等计算机课程有着一定的帮助。

## 其他备注

---

助教真帅