

PA3 实验报告

- 姓名：陈睿颖
- 学号：2013544
- 专业：计算机科学与技术

目录

1. 实验内容	2
2. 阶段一	2
2.1 加载操作系统的第一个用户程序	2
2.2 实现 loader	2
2.3 实现中断机制	3
2.4 重新组织 TrapFrame 结构体	6
3. 阶段二	7
3.1 实现系统调用	7
3.2 在 Nanos-lite 上运行 Hello world	9
3.3 实现堆区管理	10
3.4 实现简易文件系统	12
4. 阶段三	16
4.1 把 VGA 显存抽象成文件	16
4.2 把设备输入抽象成文件	18
4.3 在 NEMU 中运行仙剑奇侠传	20
5. 思考题	20
对比异常与函数调用	20
诡异的代码	21
6. Bug 记录及解决办法	22
6.1 在 dummy 目录下直接执行命令报错	23
6.2 运行 loader 报错物理地址越界	23
6.3 error: implicit declaration of function 'raise_intr' [-Werror=implicit-function-declaration]	23
6.4 实现_sbrk()后仍然是逐个字符地进行输出	24
6.5 实现完文件系统后显示 DEFAULT ENTRY 处指令未实现	24

1. 实验内容

1. 第一阶段，熟悉操作系统的基本概念、系统调用，实现中断机制。
2. 第二阶段，进一步完善系统调用，实现简易文件系统。
3. 第三阶段，将输入输出抽象成文件，并运行仙剑奇侠传。

2. 阶段一

2.1 加载操作系统的第一个用户程序

在 Nanos-lite 上运行的第一个用户程序是 `navy-apps/tests/dummy/dummy.c`。首先我们让 Navy-apps 项目上的程序默认编译到 x86 中：

```
PowerShell
NAME = dummy
SRCS = dummy.c

include $(NAVY_HOME)/Makefile.app
```

在 `navy-apps/` 目录下执行：

```
PowerShell
make test=dummy
```

在 `nanos-lite` 下执行：

```
PowerShell
make update
```

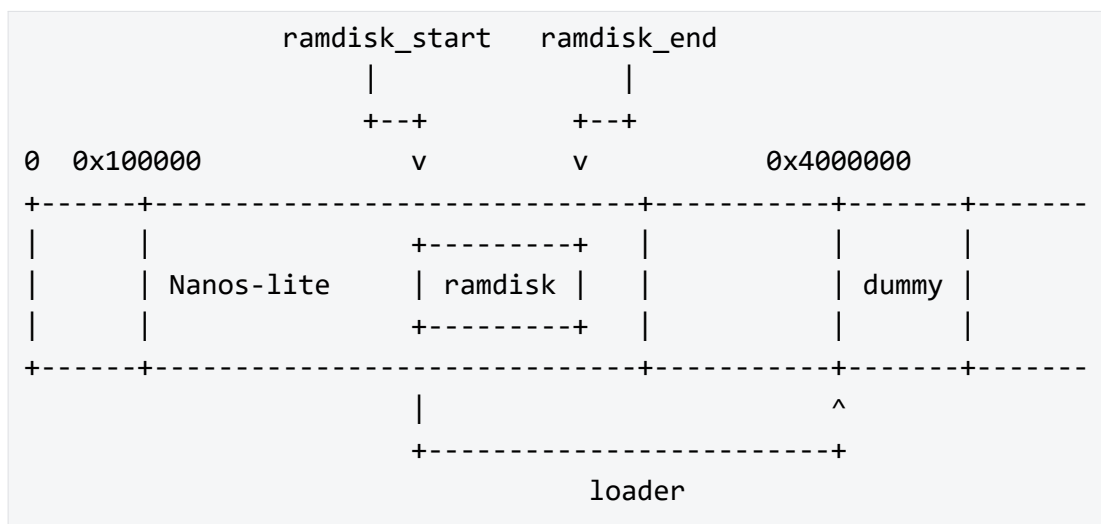
```
user@user-VirtualBox:~/ics2017/ics2017/nanos-lite$ make clean;make update
Building nanos-lite []
rm -rf /home/user/ics2017/ics2017/nanos-lite/build/
Building nanos-lite [native]
objcopy -S --set-section-flags .bss=alloc,contents -O binary /home/user/ics2017/ics2017/navy-apps/tests/dummy/build/dummy-x86 build/ramdisk.img
touch src/files.h
user@user-VirtualBox:~/ics2017/ics2017/nanos-lite$
```

2.2 实现 loader

在 Nanos-lite 中实现 loader 的功能，来把用户程序加载到正确的内存位置，然后执行用户程序。

示意图如下：

```
PowerShell
```



在 `loader.c` `loader` 函数中补全：

PowerShell

```
ramdisk_read(DEFAULT_ENTRY,0,get_ramdisk_size());
```

运行结果如下：

```
[src/monitor/monitor.c,65,load_img] The image is /home/user/ics2017/ics2017/nanos-lite/build/nanos-lite-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 18:03:54, May 8 2023
For help, type "help"
(nemu) C
Unknown command 'C'
(nemu) c
[src/main.c,19,main] 'Hello World!' from Nanos-lite
[src/main.c,20,main] Build time: 17:59:11, May 8 2023
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x100d38, end = 0x105434, size = 18172 bytes
invalid opcode(eip = 0x0400200c): cd 80 5b 5d c3 66 90 90 ...

There are two cases which will trigger this unexpected exception:
1. The instruction at eip = 0x0400200c is not implemented.
2. Something is implemented incorrectly.
Find this eip(0x0400200c) in the disassembling result to distinguish which case it is.

If it is the first case, see
0306 Manual
for more details.
```

0xcd 是 `int 1b` 指令，这说明 `loader` 已经成功加载 `dummy`，并且成功地跳转到 `dummy` 中执行了。

2.3 实现中断机制

NEMU 中添加 IDTR 寄存器和 `lidt` 指令。然后在 `nanos-lite/src/main.c` 中定义宏 `HAS_ASYE`，这样以后，Nanos-lite 会多进行一项初始化工作：调用 `init_irq()` 函数，这最终会调用位于 `nexus-am/am/arch/x86-nemu/src/asye.c` 中的 `_asye_init()` 函数。 `_asye_init()` 函数会做两件事情，第一件就是初始化 IDT：

- a. 代码定义了一个结构体数组 `idt`, 它的每一项是一个门描述符结构体
- b. 在相应的数组元素中填写有意义的门描述符, 例如编号为 `0x80` 的门描述符就是将来系统调用的入口地址. 需要注意的是, 框架代码中还是填写了完整的门描述符(包括上文中提到的 `don't care` 的域), 这主要是为了在 QEMU 中进行 `differential testing` 时也能跳转到正确的入口地址. QEMU 实现了完整的中断机制, 如果只填写简化版的门描述符, 就无法在 QEMU 中正确运行. 但我们无需了解其中的细节, 只需要知道代码已经填写了正确的门描述符即可.
- c. 在 IDTR 中设置 `idt` 的首地址和长度

- 实现 `lidt` 指令

查询 i386 手册:

Operation

```
IF instruction = LIDT
THEN
  IF OperandSize = 16
  THEN IDTR.Limit:Base ← m16:24 (* 24 bits of base loaded *)
  ELSE IDTR.Limit:Base ← m16:32
  FI;
ELSE (* instruction = LGDT *)
  IF OperandSize = 16
  THEN GDTR.Limit:Base ← m16:24 (* 24 bits of base loaded *)
  ELSE GDTR.Limit:Base ← m16:32;
  FI;
FI;
```

在 `restart` 函数中添加:

PowerShell

```
cpu.eflags = 0x2;
cpu.cs = 0x8;
```

在 `opcodetable` 中添加:

C++

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

make_EHelper(lidt) {
  rtl_li(&t0, id_dest->addr);
  rtl_li(&cpu.idtr.limit, vaddr_read(t0, 2));
  rtl_li(&cpu.idtr.base, vaddr_read(t0+2, 4));
  if(decoding.is_operand_size_16)
```

```

    cpu.idtr.base &= 0x00ffffff;
    print_asm_template1(lidt);

}

```

- 实现 int 指令

查询 i386 手册：

INT/INTO — Call to Interrupt Procedure

Opcode	Instruction	Clocks	Description
CC	INT 3	33	Interrupt 3--trap to debugger
CC	INT 3	pm=59	Interrupt 3--Protected Mode, same privilege
CC	INT 3	pm=99	Interrupt 3--Protected Mode, more privilege
CC	INT 3	pm=119	Interrupt 3--from V86 mode to PL 0
CC	INT 3	ts	Interrupt 3--Protected Mode, via task gate
CD ib	INT imm8	37	Interrupt numbered by byte
CD ib	INT imm8	pm=59	Interrupt--Protected Mode, same privilege
CD ib	INT imm8	pm=99	Interrupt--Protected Mode, more privilege
CD ib	INT imm8	pm=119	Interrupt--from V86 mode to PL 0
CD ib	INT imm8	ts	Interrupt--Protected Mode, via task gate
CE	INTO	Fail:3,pm=3; Pass:35	Interrupt 4--if overflow flag is 1
CE	INTO	pm=59	Interrupt 4--Protected Mode, privilege
CE	INTO	pm=99	Interrupt 4--Protected Mode, more privilege
CE	INTO	pm=119	Interrupt 4--from V86 mode to PL 0
CE	INTO	ts	Interrupt 4--Protected Mode, via task gate

NOTE:

Approximate values of ts are given by the following table:

Old Task	New Task		
	386 TSS VM = 0	386 TSS VM = 1	286 TSS
386 TSS VM=0	309	226	282
386 TSS VM=1	314	231	287
286 TSS	307	224	280

补全 opcode_table 及 raise_intr:

```

C++
//exec.c
/* 0xcc */          EX(int3), IDEXW(I,int,1), EMPTY, EX(iret),

//system.c
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
}

//intr.c
void raise_intr(uint8_t NO, vaddr_t ret_addr) {
    // 获取门描述符
    vaddr_t gate_addr = cpu.idtr.base + 8 * NO;

    // P 位校验
    if (cpu.idtr.limit < 0) {
        assert(0);
    }

    // 将 EFLAGS、CS、返回地址压栈
    uint32_t t0 = cpu.cs; // cpu.cs 只有 16 位，需要转换成 32 位
    rtl_push(&cpu.eflags);
    rtl_push(&t0);
    rtl_push(&ret_addr);

    // 组合中断处理程序入口点
    uint32_t high, low;
    low = vaddr_read(gate_addr, 4) & 0xffff;
    high = vaddr_read(gate_addr + 4, 4) & 0xffff0000;

    // 设置 eip 跳转
    decoding.jmp_eip = high | low;
    decoding.is_jmp = true;
    // 注意：这里直接跳转到 eip，需要在调用 raise_intr 函数之后再执行
    decode 和 execute
}

```

2.4 重新组织 TrapFrame 结构体

重新组织 `nexus-am/am/arch/x86-nemu/include/arch.h` 中定义的 `_RegSet` 结构体的成员：

```

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

```

```
};
```

使得这些成员声明的顺序和 `nexus-am/am/arch/x86-nemu/src/trap.S` 中构造的 trap frame 保持一致:

```
Assembly language
//trap.S

#----|-----entry-----|-----errorcode---|---irq id---|---handler---|
.globl vecsys;    vecsys:  pushl $0;  pushl $0x80; jmp asm_trap
.globl vecnull;   vecnull: pushl $0;  pushl $-1; jmp asm_trap

asm_trap:
    pushal

    pushl %esp
    call irq_handle

    addl $4, %esp

    popal
    addl $8, %esp

    iret
```

运行结果如下:

```
[src/monitor/monitor.c,65,load_img] The image is /home/user/ics2017/ics2017/nanos-lite/build/nanos-lite-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 08:12:22, May  9 2023
For help, type "help"
(nemu) c
[src/main.c,19,main] 'Hello World!' from Nanos-lite
[src/main.c,20,main] Build time: 09:04:27, May  9 2023
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x100f5c, end = 0x105658, size = 18172 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
```

在 `nanos-lite/src/irq.c` 中的 `do_event()` 函数中触发了 BAD TRAP。

3. 阶段二

3.1 实现系统调用

1. 在 `do_event()` 中识别出系统调用事件 `_EVENT_SYSCALL`, 然后调用 `do_syscall()`.

```
C++
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;
}
```

2. 在 `nexus-am/am/arch/x86-nemu/include/arch.h` 中实现正确的 `SYSCALL_ARGx()` 宏, 让它们从作为参数的现场 `reg` 中获得正确的系统调用参数寄存器:

系统调用分为两层处理。首先, 需要使用 `switch-case` 语句识别出这是一个系统调用事件, 然后调用 `do_syscall` 函数来确定具体是哪个系统调用。在 `x86_32` 系统中, 系统调用使用中断 `int 0x80` 来实现, 系统调用号存放在 `eax` 寄存器中。同时, 系统调用返回值也存放在 `eax` 寄存器中。

当系统调用的参数小于或等于 6 个时, 参数必须按顺序存储在 `ebx`、`ecx`、`edx`、`esi`、`edi`、`ebp` 这些寄存器中。

当系统调用的参数大于 6 个时, 所有参数应该依次存放在一个连续的内存区域中, 并在 `ebx` 寄存器中保存指向该内存区域的指针。

```
C++
#define SYSCALL_ARG1(r) 0
#define SYSCALL_ARG2(r) 0
#define SYSCALL_ARG3(r) 0
#define SYSCALL_ARG4(r) 0

//=====>更改为

#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 `syscall.c` 中加入:

C++

```
a[1] = SYSCALL_ARG2(r);
a[2] = SYSCALL_ARG3(r);
a[3] = SYSCALL_ARG4(r);
```

3. 添加 `SYS_none` 系统调用、`SYS_exit` 系统调用：

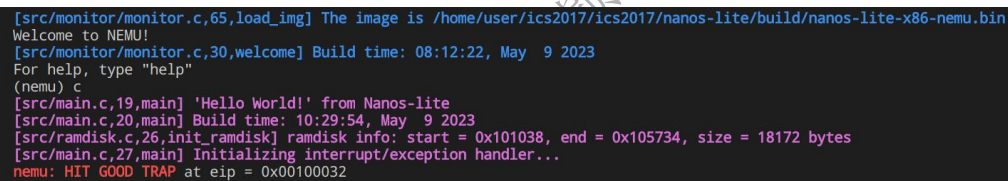
在 `nanos-lite/src/syscall.c` 补全 `do_syscall` 函数：

C++

```
case SYS_none:
    r->eax = 1;
    break;
case SYS_exit:
    _halt(a[1]);
    break;
```

4. 设置系统调用的返回值.

完成以上后运行如下：



```
[src/monitor/monitor.c,65,load_img] The image is /home/user/ics2017/ics2017/nanos-lite/build/nanos-lite-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 08:12:22, May 9 2023
For help, type "help"
(nemu) c
[src/main.c,19,main] 'Hello World!' from Nanos-lite
[src/main.c,20,main] Build time: 10:29:54, May 9 2023
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x101038, end = 0x105734, size = 18172 bytes
[src/main.c,27,main] Initializing interrupt/exception handler...
nemu: HIT GOOD TRAP at eip = 0x00100032
```

出现 GOOD TRAP 信息，说明实现成功。

3.2 在 Nanos-lite 上运行 Hello world

首先要实现 `write()` 系统调用，步骤如下：

1. 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）
2. 设置正确的返回值，否则系统调用的调用者会认为 `write` 没有成功执行，从而进行重试
3. 在 `navy-apps/libs/libos/src/nanos.c` 的 `_write()` 中调用系统调用接口函数

完成 `sys_write()` 函数：

C++

```
static inline _RegSet* sys_write(_RegSet *r) {
    //获取三个参数: fd buf len
    int fd = (int)SYSCALL_ARG2(r);
    char *buf = (char *)SYSCALL_ARG3(r);
    int len = (int)SYSCALL_ARG4(r);

    if(fd == 1 || fd == 2){
        for(int i = 0; i < len; i++) {
            _putc(buf[i]);
        }
        //根据 man 返回 len
        SYSCALL_ARG1(r) = len;
    }

    return NULL;
}
```

在 `do_syscall()` 函数中补全:

```
C++
    case SYS_write:
        sys_write(r);
        break;
```

修改 MakeFile:

```
Bash
OBJCOPY_FILE = $(NAVY_HOME)/tests/hello/build/hello-x86
```

运行结果如下:

```
[src/monitor/monitor.c,65,load_img] The image is /home/user/ics2017/ics2017/nanos-lite/build/nanos-lite-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 08:12:22, May 9 2023
For help, type "help"
(nemu) c
[src/main.c,19,main] 'Hello World!' from Nanos-lite
[src/main.c,20,main] Build time: 10:29:54, May 9 2023
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x10108c, end = 0x1058a8, size = 18460 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
```

3.3 实现堆区管理

`_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

上述代码是在用户层的库函数中实现的, 我们还需要在 `Nanos-lite` 中实现 `SYS_brk` 的功能. 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的。

在 `nanos-lite/src/syscall.c` 修改 `do_syscall` 函数:

```
C++
    case SYS_brk:
        sys_brk(r);
        break;
```

实现 `sys_brk` 函数:

```
C++
static inline _RegSet* sys_brk(_RegSet *r){
    SYSCALL_ARG1(r) = 0; //总是返回 0
    return NULL;
}
```

在 `nanos.c` 中实现 `_sbrk()`:

```
C++

extern char _end; //为了记录旧的 brk, 需要这个
static intptr_t brk = (intptr_t)&_end;
void *_sbrk(intptr_t increment){
    intptr_t old_brk = brk;
    intptr_t new_brk = old_brk + increment;
    if(_syscall_(SYS_brk, new_brk, 0, 0) == 0){
```

```

    brk = new_brk;
    return (void*)old_brk;
}
return (void *)-1;
}

```

运行如下:

```

[src/monitor/monitor.c,65,load_img] The image is /home/user/ics2017/ics2017/nanos-lite/build/nanos-lite-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 08:12:22, May  9 2023
For help, type "help"
(nemu) c
[src/main.c,19,main] 'Hello World!' from Nanos-lite
[src/main.c,20,main] Build time: 18:51:17, May 10 2023
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x101108, end = 0x105968, size = 18528 bytes
[src/main.c,27,main] Initializing interrupt/exception handler...
[src/syscall.c,9,sys_write] used sys_write!
Hello World!
[src/syscall.c,9,sys_write] used sys_write!
Hello World for the 2th time
[src/syscall.c,9,sys_write] used sys_write!
Hello World for the 3th time
[src/syscall.c,9,sys_write] used sys_write!
Hello World for the 4th time
[src/syscall.c,9,sys_write] used sys_write!
Hello World for the 5th time

```

3.4 实现简易文件系统

对 nanos-lite/Makefile 作如下修改:

```

Makefile
-update: update-ramdisk-objcopy src/syscall.h
+update: update-ramdisk-fsimg src/syscall.h

```

为每一个已经打开的文件引入偏移量属性 `open_offset`, 来记录目前文件操作的位置. 每次对文件读写了多少个字节, 偏移量就前进多少.

```

C++
typedef struct {
    char *name;           // 文件名
    size_t size;          // 文件大小
    off_t disk_offset;    // 文件在 ramdisk 中的偏移
    off_t open_offset;    // 文件被打开之后的读写指针
} Finfo;

```

接下来需要实现以下函数:

C++

```
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, const void *buf, size_t len);
off_t fs_lseek(int fd, off_t offset, int whence);
int fs_close(int fd);
```

- 由于简易文件系统中每一个文件都是固定的, 不会产生新文件, 因此"fs_open()没有找到 pathname 所指示的文件"属于异常情况, 需要使用 assertion 终止程序运行.
- 为了简化实现, 我们允许所有用户程序都可以对所有已存在的文件进行读写, 这样以后, 我们在实现 fs_open()的时候就可以忽略 flags 和 mode 了.
- 使用 ramdisk_read()和 ramdisk_write()来进行文件的真正读写.
- 由于文件的大小是固定的, 在实现 fs_read(), fs_write()和 fs_lseek()的时候, 注意偏移量不要越过文件的边界.
- 除了写入 stdout 和 stderr 之外(用 _putc()输出到串口), 其余对于 stdin, stdout 和 stderr 这三个特殊文件的操作可以直接忽略.
- 由于我们的简易文件系统没有维护文件打开的状态, fs_close()可以直接返回 0, 表示总是关闭成功.

具体实现如下:

```
C++
int fs_open(const char *pathname, int flags, int mode) {
    //可读写所有文件, 故忽略 flags mode
    Log("Pathname: %s", pathname);
    int i;

    for (i = 0; i < NR_FILES; i++) {
        //printf("file name: %s\n", file_table[i].name);
        if (strcmp(file_table[i].name, pathname) == 0) {
            Log("file opened");
            return i;
        }
    }
    assert(0);
    Log("read over");

    return -1;
}

ssize_t fs_read(int fd, void *buf, size_t len) {
    ssize_t fs_size = fs_filesz(fd);
```

```

        //if(file_table[fd].open_offset >= fs_size) //实际上不会出现这情况

        //return 0;
        if (file_table[fd].open_offset + len > fs_size) //偏移量不可以超过文件边界 超出部分舍弃
            len = fs_size - file_table[fd].open_offset;
        switch(fd) {
            case FD_STDOUT:
            case FD_STDERR:
            case FD_STDIN:
                return 0;
            case FD_EVENTS:
                len = events_read((void *)buf, len);
                break;
            case FD_DISPINFO:
                dispinfo_read(buf,
file_table[fd].open_offset, len);
                file_table[fd].open_offset += len;
                break;
            default:
                ramdisk_read(buf,
file_table[fd].disk_offset + file_table[fd].open_offset, len);
                file_table[fd].open_offset += len;
                break;
        }
        Log("file read over");
        return len;
    }

    ssize_t fs_write(int fd, const void *buf, size_t len) {
        ssize_t fs_size = fs_filesz(fd);
        switch(fd) {
            case FD_STDOUT:
            case FD_STDERR:
                // call _putc()
                // 串口已被抽象成 stdout stderr
                for(int i = 0; i < len; i++) {
                    _putc(((char*)buf)[i]);
                }
                break;
            case FD_FB:
                // write to frame buffer 显存
                // device.c:fb_write buff 中 len 字节输出到屏

```

幕上 offest 处

```

        fb_write(buf, file_table[fd].open_offset,
len);

        file_table[fd].open_offset += len;
        break;
    default:
        // write to ramdisk
        //if(file_table[fd].open_offset >=
fs_size)

            //return 0;
            if(file_table[fd].open_offset + len >
fs_size)

                len = fs_size -
file_table[fd].open_offset;
                // 对文件的真正读写
                ramdisk_write(buf,
file_table[fd].disk_offset + file_table[fd].open_offset, len);
                file_table[fd].open_offset += len;
                //Log("offset = %d",
file_table[fd].open_offset);
                break;
    }
    Log("file write over");

    return len;// 参见 man 返回值
}

off_t fs_lseek(int fd, off_t offset, int whence) {
    off_t result = -1;
    // fs.h
    // man 2 lseek 同时注意边界问题
    switch(whence) {
        case SEEK_SET:
            if (offset >= 0 && offset <=
file_table[fd].size) {
                file_table[fd].open_offset =
offset;
                result =
file_table[fd].open_offset;
            }
            break;
        case SEEK_CUR:
            if ((offset +
file_table[fd].open_offset >= 0) && (offset +
```

```

file_table[fd].open_offset <= file_table[fd].size)) {
    file_table[fd].open_offset +=
offset;

    result =
file_table[fd].open_offset;
}
break;
case SEEK_END:
    file_table[fd].open_offset =
file_table[fd].size + offset;
    result = file_table[fd].open_offset;
    break;
}
Log("file seek over");

return result;
}

int fs_close(int fd) {
    //fs_lseek(fd,0,SEEK_SET);
    Log("file closed");
    return 0;
}

```

在 Nanos-lite 和 Navy-apps 的 libos 中添加相应的系统调用, 来调用相应的文件操作。

```

[src/fs.c,82,fs_read] file read over
[src/fs.c,142,fs_lseek] file seek over
[src/fs.c,82,fs_read] file read over
[src/fs.c,82,fs_read] file read over
[src/fs.c,82,fs_read] file read over
[src/fs.c,150,fs_close] file closed
PASS!!!
[src/fs.c,115,fs_write] file write over
[src/syscall.c,34,sys_write] used sys_write!
nemu: HIT GOOD TRAP at eip = 0x00100032
(nemu)

```

4. 阶段三

4.1 把 VGA 显存抽象成文件

该部分需要完成的工作如下:

- 在 `init_fs()`(在 `nanos-lite/src/fs.c` 中定义)中对文件记录表中 `/dev/fb` 的大小进行初始化, 需要使用 IOE 定义的 API 来获取屏幕的大小.


```
C++
void init_fs() {
    // TODO: initialize the size of /dev/fb
    file_table[FD_FB].size = _screen.height * _screen.width * 4;
}
```

- 实现 `fb_write()` (在 `nanos-lite/src/device.c` 中定义), 用于把 `buf` 中的 `len` 字节写到屏幕上 `offset` 处. 需要先从 `offset` 计算出屏幕上的坐标, 然后调用 IOE 的 `_draw_rect()` 接口.

```
C++
void fb_write(const void *buf, off_t offset, size_t len) {
    int row = (offset/4)/_screen.width;
    int col = (offset/4)%_screen.width;
    _draw_rect(buf,col,row,len/4,1);
}
```

- 在 `init_device()` (在 `nanos-lite/src/device.c` 中定义) 中将 `/proc/dispinfo` 的内容提前写入到字符串 `dispinfo` 中. 实际的屏幕大小信息已经记录在 AM 的 IOE 接口中, 你需要在 `Nanos-lite` 中获取它们.

```
C++
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\n", _screen.width, _screen.height);
}
```

- 实现 `dispinfo_read()` (在 `nanos-lite/src/device.c` 中定义), 用于把字符串 `dispinfo` 中 `offset` 开始的 `len` 字节写到 `buf` 中.

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

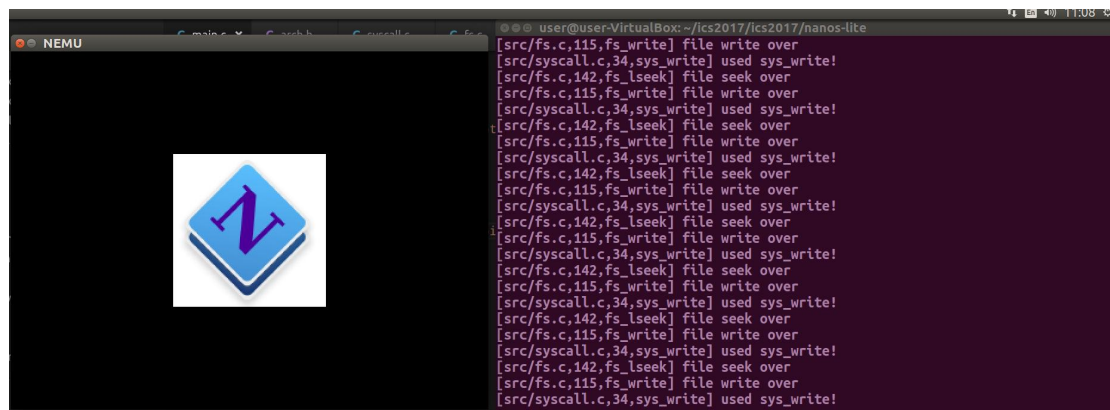
- 在文件系统中添加对 `/dev/fb` 和 `/proc/dispinfo` 这两个特殊文件的支持

让 Nanos-lite 加载 `/bin/bmptest`,

```
C++
//main.c

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

运行结果如下:



4.2 把设备输入抽象成文件

下面在 Nanos-lite 中

- 实现 `events_read()` (在 `nanos-lite/src/device.c` 中定义), 把事件写入到 `buf` 中, 最长写入 `len` 字节, 然后返回写入的实际长度. 其中按键名已经在字符串数组 `names` 中定义好了. 你需要借助 IOE 的 API 来获得设备的输入.
- 在文件系统中添加对 `/dev/events` 的支持.

```
C++
size_t events_read(void *buf, size_t len) {
    //return 0;
    int key = _read_key();
    bool down = false;
    if (key & 0x8000) {
        key ^= 0x8000;
        down = true;
    }
    if (key == _KEY_NONE) {
        unsigned long t = _uptime();
        sprintf(buf, "t %d\n", t);
    }
}
```

```

        else {
            sprintf(buf, "%s %s\n", down ? "kd" : "ku",
keyname[key]);
            // if(key == 13 && down) { //F12 DOWN
            //     current_game = (current_game == 0 ? 1 :
0);
            // }
            Log("Get key: %d %s %s\n", key, keyname[key],
down ? "down" : "up");
        }
        return strlen(buf);//xxx strlen(buf)-1
    }
}

```

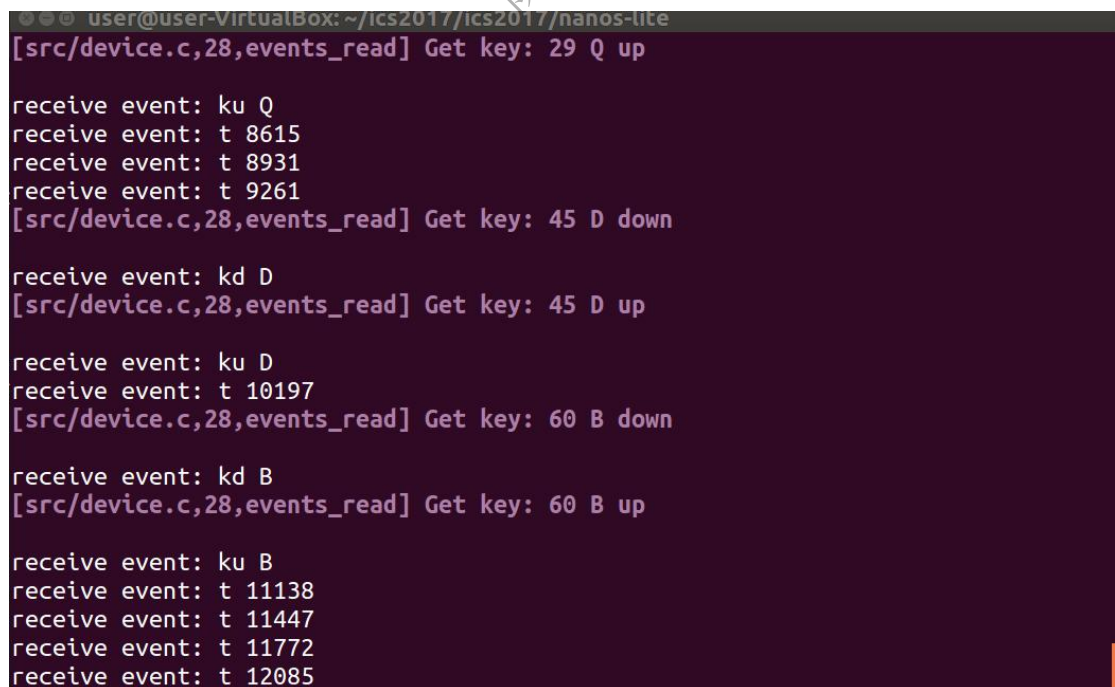
让 Nanos-lite 加载/bin/events,

```

C++
//main.c

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

```



A terminal window showing the output of a program running in Nanos-lite. The prompt is 'user@user-VirtualBox: ~/ics2017/ics2017/nanos-lite'. The output shows a sequence of key events: 'Q' (key 29), 'D' (key 45), and 'B' (key 60). For each key, there are two events: 'down' and 'up'. The output is as follows:

```

[src/device.c,28,events_read] Get key: 29 Q up
receive event: ku Q
receive event: t 8615
receive event: t 8931
receive event: t 9261
[src/device.c,28,events_read] Get key: 45 D down
receive event: kd D
[src/device.c,28,events_read] Get key: 45 D up
receive event: ku D
receive event: t 10197
[src/device.c,28,events_read] Get key: 60 B down
receive event: kd B
[src/device.c,28,events_read] Get key: 60 B up
receive event: ku B
receive event: t 11138
receive event: t 11447
receive event: t 11772
receive event: t 12085

```

如图所示，程序输出时间事件的信息，敲击按键时会输出按键事件的信息，实现正确！

4.3 在 NEMU 中运行仙剑奇侠传

下载仙剑奇侠传的数据文件, 并放到 `navy-apps/fsimg/share/games/pal/` 目录下:

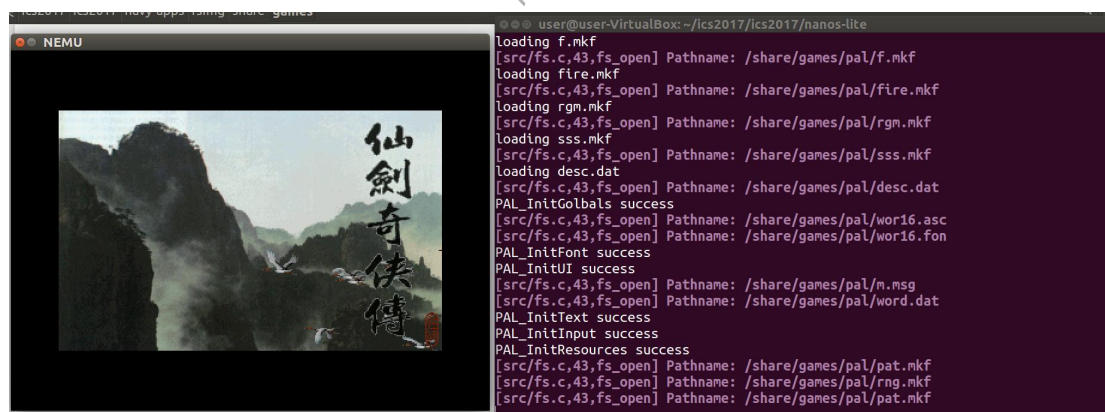


更新 ramdisk 之后, 在 Nanos-lite 中加载并运行/bin/pal.

```
C++
//main.c

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

成功运行:



5. 思考题

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.

函数调用和异常处理的确都需要保存调用者的状态, 但是异常处理需要保存的信息更多. 主要的原因是异常处理涉及到处理器状态的保存和恢复, 而这些状态包括处理器的寄存器和特权级等状态。

具体来说, 异常处理需要保存和恢复的信息通常包括:

- EFLAGS 寄存器, 保存处理器当前的标志位状态。
- CS 和 EIP 寄存器, 保存当前正在执行的代码段和指令指针。
- SS 和 ESP 寄存器, 保存当前栈的状态。
- 处理器特权级状态, 保存当前处理器的特权级状态。

而函数调用只需要保存调用者保存的寄存器以及返回地址, 一般不涉及处理器特权级等状态的变化。

因此, 异常处理需要保存更多的信息, 而函数调用只需要保存有限的状态信息。

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?

Assembly language

```
#----|-----entry-----|errorcode|---irq id---|---handler---|
.globl vecsys;    vecsys:  pushl $0;  pushl $0x80; jmp asm_trap
.globl vecnull;   vecnull: pushl $0;  pushl  $-1; jmp asm_trap

asm_trap:
    pushal

    pushl %esp
    call irq_handle

    addl $4, %esp

    popal
    addl $8, %esp
```

iret

这段代码实现了一个汇编的中断处理函数 `asm_trap`，它的主要作用是将所有的寄存器保存在栈上，然后调用 `irq_handle` 处理中断，最后将栈恢复到原来的状态并返回 `iret`。

在保存寄存器之前，中断处理函数使用 `pushl %esp` 指令将当前栈顶地址压入栈中，随后调用 `irq_handle` 处理中断，中断处理完成后，`addl $4, %esp` 指令将栈顶地址加上 4，将刚刚保存的栈顶地址弹出栈，栈指针重新指向原来的位置。

使用 `pushl %esp` 指令：因为在中断处理函数中调用其他函数时，栈指针可能会发生变化，这样就会导致返回时栈指针的值与进入时不一致，进而导致程序崩溃。因此，在调用其他函数之前，中断处理函数需要先将当前的栈指针保存下来，等到处理完成后再恢复。由于 `pushl` 指令会将栈顶地址减去 4，所以在恢复栈指针时需要使用 `addl $4, %esp` 指令将其加上 4。

文件读写的具体过程 仙剑奇侠传中有以下行为：

- 在 `navy-apps/apps/pal/src/global/global.c` 的 `PAL_LoadGame()` 中通过 `fread()` 读取游戏存档
- 在 `navy-apps/apps/pal/src/hal/hal.c` 的 `redraw()` 中通过 `NDL_DrawRect()` 更新屏幕

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。

- 存档读取

`PAL_LoadGame` 先打开指定文件然后调用 `fread` 从文件里读取存档 相关信息（其中包括调用 `nanos.c` 里的 `_read` 以及 `syscall.c` 中的 `sys_read`），随后关闭文件并把读取到的信息赋值（用 `fs_write` 修改），接着使用 `AM` 提供的 `memcpy` 拷贝数据，最后使用 `nemu` 的内存映射 I/O 修改内存。

- 更新屏幕 `redraw`

调用 `ndl.c` 里面的 `NDL_DrawRect` 来绘制矩形，`NDL_Render` 把 VGA 显存抽象成文件，它们都调用了 `nanos-lite` 中的接口，最后 `nemu` 把文件通过 I/O 接口显示到屏幕上。

6. Bug 记录及解决办法

6.1 在 dummy 目录下直接执行命令报错

make 有报错:

```
user@user-VirtualBox:~/ics2017/ics2017/navy-apps/tests/dummy$ make
make: *** 没有规则可以创建目标“$(APP)”。 停止。
```

make run 也会报错:

```
user@user-VirtualBox:~/ics2017/ics2017/navy-apps/tests/dummy$ make run
make -C /home/user/ics2017/ics2017/navy-apps/libs/libc
make[1]: 正在进入目录 `/home/user/ics2017/ics2017/navy-apps/libs/libc'
make[1]: 没有什么可以做的为 `archive'。
make[1]:正在离开目录 `/home/user/ics2017/ics2017/navy-apps/libs/libc'
make -C /home/user/ics2017/ics2017/navy-apps/libs/libos
make[1]: 正在进入目录 `/home/user/ics2017/ics2017/navy-apps/libs/libos'
make[1]: 没有什么可以做的为 `archive'。
make[1]:正在离开目录 `/home/user/ics2017/ics2017/navy-apps/libs/libos'
+ LD /home/user/ics2017/ics2017/navy-apps/tests/dummy/build/dummy-x86
/home/user/ics2017/ics2017/navy-apps/Makefile.app:53: *** Cannot run: should be
loaded by an OS。 停止。
user@user-VirtualBox:~/ics2017/ics2017/navy-apps/tests/dummy$
```

解决办法:

在 navy-apps 目录下指定 ALL=dummy 即可解决。

6.2 运行 loader 报错物理地址越界

```
31  uintptr_t loader(_Protect *as, const char *filename) {
32      // TODO();
33
34      ramdisk_read(DEFAULT_ENTRY,0,get_ramdisk_size());
35
36      return (uintptr_t)DEFAULT_ENTRY;
37  }
38
```

问题 8 输出 调试控制台 终端

```
[src/monitor/monitor.c,30,welcome] Build time: 15:28:40, May 8 2023
For help, type "help"
(nemu) c
[src/main.c,19,main] 'Hello World!' from Nanos-lite
[src/main.c,20,main] Build time: 15:32:22, May 8 2023
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x100d38, end = 0x105434, size = 18172 bytes
physical address(0xffffffff80) is out of bound
nemu: src/memory/memory.c:18: paddr_read: Assertion `addr < (128 * 1024 * 1024)' failed.
make[1]: *** [run] 已放弃 (core dumped)
make[1]:正在离开目录 `/home/user/ics2017/ics2017/nemu'
make: *** [run] 错误 2
user@user-VirtualBox:~/ics2017/ics2017/nanos-lite$
```

如图所示, 报错物理地址越界。

错误原因: PA2 中将 int 指令填了但是只实现了一部分, 导致程序半对不对。

解决办法: 先把已实现的 int 部分删去, 后续实验中完成完善。

6.3 error: implicit declaration of function 'raise_intr' [-

Werror=implicit-function-declaration]

```
+ CC src/cpu/exec/system.c
src/cpu/exec/system.c: In function 'exec_int':
src/cpu/exec/system.c:53:3: error: implicit declaration of function 'raise_intr' [-Werror=implicit-function-declaration]
  raise_intr(id_dest->val, decoding.seq_eip);
  ^
cc1: all warnings being treated as errors
make[1]: *** [build/obj/cpu/exec/system.o] 错误 1
make[1]:正在离开目录 '/home/user/ics2017/ics2017/nemu'
make: *** [run] 错误 2
```

报错如图所示，需要在使用该函数的代码文件开头进行函数的声明：

C++

```
extern void raise_intr(uint8_t NO, vaddr_t ret_addr) ;
```

加上后不再报错。

6.4 实现_sbrk()后仍然是逐个字符地进行输出

如图所示，带上其他日志进行输出时，发现仍然是逐个字符地进行输出：

```
r[src/syscall.c,9,sys_write] used sys_write!
l[src/syscall.c,9,sys_write] used sys_write!
d[src/syscall.c,9,sys_write] used sys_write!
[src/syscall.c,9,sys_write] used sys_write!
f[src/syscall.c,9,sys_write] used sys_write!
o[src/syscall.c,9,sys_write] used sys_write!
r[src/syscall.c,9,sys_write] used sys_write!
[src/syscall.c,9,sys_write] used sys_write!
t[src/syscall.c,9,sys_write] used sys_write!
h[src/syscall.c,9,sys_write] used sys_write!
e[src/syscall.c,9,sys_write] used sys_write!
[src/syscall.c,9,sys_write] used sys_write!
1[src/syscall.c,9,sys_write] used sys_write!
8[src/syscall.c,9,sys_write] used sys_write!
2[src/syscall.c,9,sys_write] used sys_write!
t[src/syscall.c,9,sys_write] used sys_write!
h[src/syscall.c,9,sys_write] used sys_write!
[src/syscall.c,9,sys_write] used sys_write!
t[src/syscall.c,9,sys_write] used sys_write!
i[src/syscall.c,9,sys_write] used sys_write!
m[src/syscall.c,9,sys_write] used sys_write!
```

解决办法：在 `navy-apps/tests/hello` 下重新编译，再运行即可。因为 `nanos.c` 文件改变，所以需要重新编译。

6.5 实现完文件系统后显示 DEFAULT ENTRY 处指令未实现

如图所示

```
[src/fs.c,43,fs_open] Pathname: /bin/text
[src/fs.c,49,fs_open] file opened
[src/loader.c,20,loader] fd=24

[src/fs.c,82,fs_read] file read over
[src/fs.c,150,fs_close] file closed
invalid opcode(eip = 0x04000000): 2f 6c 69 62 2f 6c 64 2d ...
```

There are two cases which will trigger this unexpected exception:
1. The instruction at eip = 0x04000000 is not implemented.
2. Something is implemented incorrectly.
Find this eip(0x04000000) in the disassembling result to distinguish which case it is.

解决办法：

在 `nexus-am/Makefile.check` 中修改：

```
C++  
ARCH ?= x86-nemu
```

即可解决。

陈睿颖