



南开大学
Nankai University

南 开 大 学

计 算 机 学 院

系统综合课程设计实验报告

PA3 实验报告

周辰霏 1712991

年级：2017 级

专业：计算机科学与技术

2020 年 5 月 22 日

目录

一、 概述	1
(一) 实验目的	1
(二) 实验内容	1
二、 阶段一	1
(一) 了解 Nanos-lite	1
(二) 中断触发	1
(三) 保存上下文	3
(四) 事件打包分发	3
(五) 恢复上下文	4
三、 阶段二	5
(一) 加载用户程序	5
(二) YIELD 和 EXIT 系统调用	5
(三) 标准输出和堆区管理	6
四、 阶段三	8
(一) 文件操作	8
(二) 万物皆文件	11
(三) 运行仙剑	13
(四) DiffTest	14
(五) 展示批处理系统	15
五、 遇到的 bug 及解决	16
六、 思考题 && 必答题	17
(一) 思考题	17
(二) 必答题	18
七、 总结	19

一、 概述

(一) 实验目的

- 梳理操作系统概念
- 学习系统调用, 并实现中断机制
- 了解文件系统的基本内容, 实现简易文件系统
- 实现支持文件操作的操作系统, 要求能成功运行仙剑奇侠传

PA3 完成的内容和标题一样——批处理系统, 基本上实现了除了内存管理和进程管理 (PA4) 以外的其他基本功能, 尤以中断和系统调用为最, 贯穿整个 PA3 始终。整体而言完成了中断异常处理, 系统调用相关之后底层基础已经打好了, 再利用这些实现输出和文件系统 (极简版) 后就可以加载 pal 开始剧情的享受了。最后还有一部分基础设施的完善内容——DiffTest 的自由调试开关控制。

(二) 实验内容

PA3 实验涉及现代指令系统的实现、抽象机器 AM 的原理与应用、输入输出设备三部分

1. 熟悉操作系统的基本概念、系统调用, 实现子线操作 `_yield()`, 这其中包括异常的发现和处理以及 CTE 上下文的保存和恢复
2. 实现用户程序的加载和系统调用, 支撑 TRM 程序的运行
3. 运行仙剑奇侠传并展示批处理系统, 包括文件系统的实现
4. 对 DiffTest 实现自由开关以自行选择 DiffTest 进行调试的代码部分

二、 阶段一

(一) 了解 Nanos-lite

可以把 Nanos-lite 理解成一个运行在 AM 上的特殊的应用程序, 它协助我们管理和运行其他应用程序。阅读 main.c 了解当前 Nanos-lite 都做了些什么, 输出编译信息、初始化磁盘 (一段内存)、设备初始化 (IOE)、初始化文件以及进程 (PA4 的内容)、调用 panic 结束 Nanos-lite。

而当我们在 common.h 里面取消对 HAS_CTE 的注释以后, Nanos-lite 就会在 panic 之前调用 `_yield()` 以触发自陷操作, 在这之前还会先对 CTE 上下文进行初始化: 初始化 IDT 并对其内容进行设置使其有意义, 之后设置一个处理回调函数, 我们可以用这个来处理事件。

(二) 中断触发

中断实际上由触发中断、保存上下文、切换内核模式、根据中断号对其进行处理、恢复上下文组成。实现自陷操作先从 `_yield()` 看起, 指导书提到了 IDT 的设置中需要使用 `lidt` 指令, 而在 cte.c 中定义的 `yield` 实际上是借助 x86 中的 `int` 指令引发一个中断来让中断例程对中断进行处理, 那么显然我们需要实现 `int` 指令。两个命令都在 system.c 中, 还是和 PA2 一样的实现即可。由于加入了新的内容, 所以需要在 reg.h 中对 IDTR 进行声明, 同时为了 DiffTest 的对照还要加入一个 CS 寄存器。为了配合 DiffTest 的运行, eflags 和 CS 都需要在 restart 函数中进行相应初始化。int 指令是对触发 int 之前的 eip 进行处理, 而非当前 eip; 而 `lidt` 指令是对 `id_dest` 的地址进行操作。

int-lidt 实现

```

1 //nemu/src/isa/x86/exec/exec.c
2 /* 0x0f 0x01*/
3 make_group(gp7,
4     EMPTY, EMPTY, EMPTY, EX(lidt),
5     EMPTY, EMPTY, EMPTY, EMPTY)
6 /* 0xcc */ EMPTY, IDEXW(1, int, 1), EMPTY, EX(iret),
7 //nemu/src/isa/x86/include/isa/reg.h
8 struct
9 {
10     uint16_t limit;
11     uint32_t base;
12 }IDTR;
13 rtlreg_t CS;
14 //nemu/src/isa/x86/exec/system.c
15 make_EHelper(int) {
16     raise_intr(id_dest->val, decoding.seq_eip); // 之前的 eip
17     print_asm("int %s", id_dest->str);
18
19     #ifdef DIFF_TEST
20         diff_test_skip_nemu();
21     #endif
22 }
23 make_EHelper(lidt) {
24     rtl_li(&t0, id_dest->addr); // 不是 id_dest->val !!!
25     rtl_li(&cpu.IDTR.limit, vaddr_read(t0, 2));
26     rtl_li(&cpu.IDTR.base, vaddr_read(t0+2, 4));
27     print_asm_template1(lidt);
28 }
29 //nemu/src/isa/x86/init.c
30 static void restart() {
31     /* Set the initial program counter. */
32     cpu.pc = PC_START;
33     cpu.CS=8;
34     cpu.eflags.flags=0x2;
35 }

```

前面准备工作都做好之后, 就可以实现 raise_intr() 函数了, raise_intr() 函数实际上是根据中断号对中断进行标记追踪处理, 在跳转到新的地址的同时保存当前 cpu 的状态。

raise_intr()

```

1 //nemu/src/isa/x86/init/intr.c
2 void raise_intr(uint8_t NO, vaddr_t ret_addr) {
3     /* TODO: Trigger an interrupt/exception with NO''.
4      * That is, use NO'' to index the IDT.
5      */
6     // 当前状态压栈保存
7     rtl_push(&cpu.eflags.flags);

```

```

8   cpu.eflags.IF = 0;
9   rtl_push(&cpu.cs);
10  rtl_push(&ret_addr);
11  // 读出IDTR首地址,找到们描述符
12  rtl_li(&t0, vaddr_read(cpu.idtr.i_base+8*NO,4));
13  rtl_li(&t1, vaddr_read(cpu.idtr.i_base+8*NO+4,4));
14  // 合成目标地址并跳转
15  rtl_j(t0 | t1);
16  }

```

(三) 保存上下文

之后根据 `cte.c`, 异常入口被设定为 `vectrap(trap.S 实现)`, 它调用了 `irq_handle()` 对事件进行处理。当然暂时我们并不需要关心这个, 因为此时 `run` 以后我们发现未实现命令的出现, 这个命令实际上是 `pusha`, 而它是在保存上下文中起作用保存压栈寄存器的值。

pusha 指令

```

1 //nemu/src/isa/x86/exec/exec.c
2 /* 0x60 */      EX(pusha), EX(popa), EMPTY, EMPTY,
3 make_EHelper(pusha) {
4     t0 = cpu.esp;
5     rtl_push(&cpu.eax);
6     rtl_push(&cpu.ecx);
7     rtl_push(&cpu.edx);
8     rtl_push(&cpu.ebx);
9     rtl_push(&t0);
10    rtl_push(&cpu.ebp);
11    rtl_push(&cpu.esi);
12    rtl_push(&cpu.edi);
13    print_asm("pusha");
14 }

```

之后需要重构 `_Context` 中的顺序使其和前面 `trap.S` 中构造的一致, 实际上只要和 `pusha` 压栈的顺序一致, 保证压栈正确即可。上下文的全部组成就是八个通用寄存器以及 `eip`、`cs` 和 `eflags`。

Context

```

1 //nexus-am/am/include/arch/x86-nemu.h
2 struct _Context {
3     struct _AddressSpace *as;
4     uintptr_t edi, esi, ebp, esp, ebx, edx, ecx, eax;
5     int irq;
6     uintptr_t eip, cs, eflags;
7 };

```

(四) 事件打包分发

在上下文保存好之后, OS 就会进入内核态并根据中断号来进行相应的中断处理, 而这就是我们下来要做的打包事件并使用回调函数通过 OS 进行处理。在 PA 里面我们只需要在 `cte.c` 里面对

事件识别并打包, 然后在 irq.c 里面对对应编号事件进行处理即可。

irq

```

1 // nexus-am/am/src/x86/nemu/cte.c
2 _Context* __am_irq_handle(_Context *c) {
3     _Context *next = c;
4     if (user_handler) {
5         _Event ev = {0};
6         switch (c->irq) {
7             case 0x81: ev.event=_EVENT_YIELD; break; // int 81
8             default: ev.event = _EVENT_ERROR; break;
9         }
10        next = user_handler(ev, c);
11        if (next == NULL) {
12            next = c;
13        }
14    }
15    return next;
16 }
17 // nanos-lite/src/irq.c
18 static _Context* do_event(_Event e, _Context* c) {
19     switch (e.event) {
20         case _EVENT_YIELD: printf("system yeild\n"); break; // 识别 yield 事件并输出一句话即可
21         default: panic("Unhandled event ID = %d", e.event);
22     }
23     return NULL;
24 }

```

(五) 恢复上下文

根据 NEMU 运行 Log 显示可以得出这一步骤我们只需要实现两个指令即可: popa 恢复刚刚保存的上下文中的通用寄存器、iret 返回触发中断的地方。

popa、iret

```

1 make_EHelper(popa) {
2     rtl_pop(&cpu.edi);
3     rtl_pop(&cpu.esi);
4     rtl_pop(&cpu.ebp);
5     rtl_pop(&t0); // esp
6     rtl_pop(&cpu.ebx);
7     rtl_pop(&cpu.edx);
8     rtl_pop(&cpu.ecx);
9     rtl_pop(&cpu.eax);
10    print_asm("popa");
11 }
12 make_EHelper(iret) {
13     rtl_pop(&decoding.jump_pc);
14     decoding.is_jump = 1;

```

```

15 rtl_pop(&cpu.CS);
16 rtl_pop(&cpu.eflags.flags);
17 print_asm("iret");
18 }

```

之后就可以再一次运行看到 panic 了,PA3.1 部分就结束啦。虽然是 Bad Trap,但是到目前为止都是正确的,变好是 PA3.2 的内容。

```

Welcome to NEMU!
For help, type "help"
[src/main.c,15,main] 'Hello World!' from Nanos-lite
[src/main.c,16,main] Build time: 07:34:31, Apr 19 2020
[src/ramdisk.c,28,init_ramdisk] ramdisk info: start = 1052604, end = 1061892, size = 9288 bytes
[src/device.c,31,init_device] Initializing devices...
[src/irq.c,15,init_irq] Initializing interrupt/exception handler...
[src/irq.c,5,do_event] ID = 5: Recognize yield event!
[src/main.c,34,main] system panic: Should not reach here
nemu: HIT BAD TRAP at eip = 0x00100032

```

图 1: PA3.1 结果

三、 阶段二

(一) 加载用户程序

显然批处理系统并不可能完全按照磁盘顺序依次执行,所以我们借助上述自陷操作要实现加载程序的功能模块。这一阶段的 loader 不需要对文件操作,所以简单的使用 ramdisk 提供的读盘 API 加载即可。之后再使用 naive_load(NULL, NULL) 加载 makefile 中定义的 dummy 即可。

loader1.0

```

1 uintptr_t loader(_Protect *as, const char *filename) {
2     size_t size = get_ramdisk_size();
3     void * buff = NULL;
4     ramdisk_read(buff, 0, size);
5     return (uintptr_t)DEFAULT_ENTRY;
6 }

```

(二) YIELD 和 EXIT 系统调用

这里系统调用的实现实际上和前面的自陷操作流程是一致的,不过系统调用只需要 ring3 的用户级权限就可以了。这里指导书并没有像自陷操作一样非常详细,但实际上本质上是一样的,甚至还要更简单一些。过程大致为,中断发生,进入 syscall 入口程序,保护现场,根据系统调用号进行不同的处理,恢复现场,回到原来的调用处。所以实际上和 yield 是一样的。而当前步骤不需要对不同系统调用号进行不同处理。

syscall

```

1 // nexus-am/am/src/x86/nemu/cte.c
2 case 0x80: ev.event = _EVENT_SYSCALL; break;
3 // nanos-lite/src/irq.c
4 case _EVENT_SYSCALL: do_syscall(c); break;
5 // vecsys 以及 IDT 的设置都已经有了就不需要管了

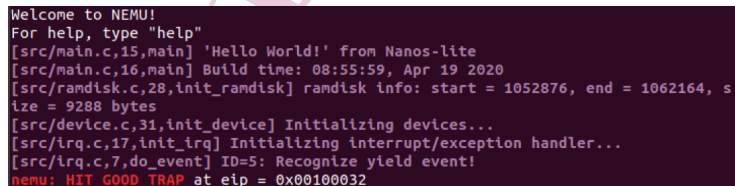
```

这个时候还是不对的, 因为我们并没能很好的使用 `do_syscall()`, 首先对 `arch.h` 里面的 `GPRx` 宏进行设定, 让它们从上下文 `c` 中获得正确的系统调用参数寄存器, 就可以用他们来实现 `syscall` 了, 其实 `GPR1-4` 对应的是 `eax,ebx,ecx,edx`, 他们也正好分别代表系统调用号和参数。之后再在 `syscall.c` 中添加 `YIELD` 系统调用并设置正确返回值即可

yield

```
1 // nexus-am/am/include/arch/x86-nemu.h
2 #define GPR1  eax
3 #define GPR2  ebx
4 #define GPR3  ecx
5 #define GPR4  edx
6 #define GPRx  eax
7 // nanos-lite/src/syscall.c
8 _Context* do_syscall(_Context *c) {
9     uintptr_t a[4];
10    a[0] = c->GPR1;
11    a[1] = c->GPR2;
12    a[2] = c->GPR3;
13    a[3] = c->GPR4;
14    switch (a[0]) {
15        case SYS_yield: _yield(); c->GPRx=0; break;
16        case SYS_exit: _halt(a[1]); break;
17    }
18    return c;
19 }
```

以上都实现完成之后就可以看到程序运行 Good Trap 了。



```
Welcome to NEMU!
For help, type "help"
[src/main.c,15,main] 'Hello World!' from Nanos-lite
[src/main.c,16,main] Build time: 08:55:59, Apr 19 2020
[src/ramdisk.c,28,init_ramdisk] ramdisk info: start = 1052876, end = 1062164, size = 9288 bytes
[src/device.c,31,init_device] Initializing devices...
[src/irq.c,17,init_irq] Initializing interrupt/exception handler...
[src/irq.c,7,do_event] ID=5: Recognize yield event!
nemu: HIT GOOD TRAP at eip = 0x00100032
```

图 2: good trap

(三) 标准输出和堆区管理

标准输出是 OS 必不可少的功能, 像 `printf` 等都是通过调用 `sys_write` 系统调用来实现标准输出的, 而为了避免一个字符一个字符输出我们需要堆的帮助, 这就需要 `sbrk` 的堆管理了。目前的 `sys_write` 不涉及到文件写, 只要根据文件描述符 `fd` 区分是否为 `stdout` 或 `stderr` 并调用 `_putc` 输出即可, 同时补全调用函数。

syswrite

```
1 // nanos-lite/src/syscall.c
2 static inline int32_t sys_write(int fd, const void *buf, size_t len)
3 { // 实现简易版 syswrite
4     if (fd == 1 || fd == 2) {
5         char *b = (char *)buf;
```



```

6     printf("len is %d\n", len);
7     for(int i=0; i<len; i++){
8         _putc(*(b++));
9     }
10    return len;
11}
12return -1;
13}
14// 系统调用号判别
15case SYS_write: c->GPRx=sys_write(a[1], (void*)a[2], a[3]); break;
16// navy-apps/libs/libos/src/nanos.c
17int _write(int fd, void *buf, size_t count) {
18    // _exit(SYS_write);
19    return _syscall_(SYS_write, fd, buf, count);
20}

```

之后就可以看到 hello 测试在进行 printf 的阶段每输出一个字符都会调用一次 sys_write, 所以为了让程序有足够的内存空间使用, 接下来实现堆的处理。

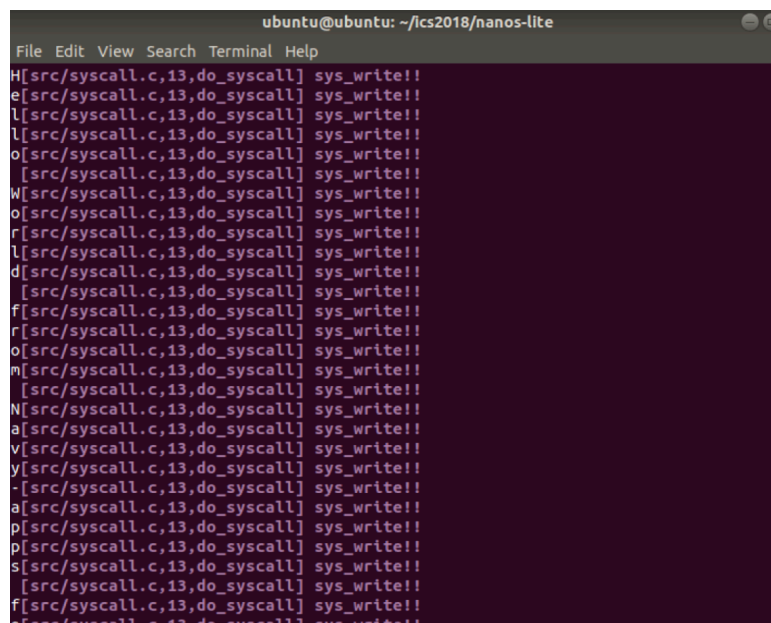


图 3: sys_write

sys_brk 是用以设置堆区大小的系统调用, 由于当前内存不受限制我们可以随意使用, 所以该系统调用只要次次返回 0, 然后根据工作方式实现 _sbrk 即可。这样 printf 就是一行一行的输出了。

sysbrk

```

1 // nanos-lite/src/syscall.c
2 case SYS_brk: c->GPRx=0; break;
3 // navy-apps/libs/libos/src/nanos.c
4 void *_sbrk(intptr_t increment) {
5     extern uint32_t _end; // pb 初始位置
6     static uint32_t program_break=&_end;
7     if(_syscall_(SYS_brk, program_break+increment, 0, 0)==0)

```

```

8      { // 计算并设置新的 pb
9          uint32_t old=program_break;
10         program_break=program_break+increment;
11         return (void*)old;
12     }
13     else { // syscall 失败
14         return (void*)-1;
15     }
16 }

```

```

[src/syscall.c,13,do_syscall] sys_write!!
Hello World from Navy-apps for the 12005th time!
[src/syscall.c,13,do_syscall] sys_write!!
Hello World from Navy-apps for the 12006th time!
[src/syscall.c,13,do_syscall] sys_write!!
Hello World from Navy-apps for the 12007th time!
[src/syscall.c,13,do_syscall] sys_write!!
Hello World from Navy-apps for the 12008th time!
[src/syscall.c,13,do_syscall] sys_write!!
Hello World from Navy-apps for the 12009th time!
[src/syscall.c,13,do_syscall] sys_write!!
Hello World from Navy-apps for the 12010th time!
[src/syscall.c,13,do_syscall] sys_write!!
Hello World from Navy-apps for the 12011th time!
[src/syscall.c,13,do_syscall] sys_write!!
Hello World from Navy-apps for the 12012th time!
[src/syscall.c,13,do_syscall] sys_write!!
Hello World from Navy-apps for the 12013th time!
[src/syscall.c,13,do_syscall] sys_write!!
Hello World from Navy-apps for the 12014th time!
[src/syscall.c,13,do_syscall] sys_write!!
Hello World from Navy-apps for the 12015th time!
[src/syscall.c,13,do_syscall] sys_write!!
Hello World from Navy-apps for the 12016th time!
[src/syscall.c,13,do_syscall] sys_write!!
Hello World from Navy-apps for the 12017th time!
[src/syscall.c,13,do_syscall] sys_write!!

```

图 4: sys_brk

四、 阶段三

(一) 文件操作

由于需要批处理系统执行的程序越来越多,单纯依靠磁盘就不够了,所以引入了文件系统,由于在 PA 中不需要多复杂,只要能支持批处理系统的基本功能就可以,且文件的大小和数量都是固定的,所以一些基本的读写操作等等就足矣。下面按照 man 给出信息一次实现文件操作函数。

文件操作

```

1 // nanos-lite/src/fs.c
2 off_t open_offset; // 目前文件操作的位置,读写指针
3 size_t fs_filesz(int fd) { // 获取文件大小
4     return file_table[fd].size;
5 }
6 int fs_open(const char *pathname, int flags, int mode)
7 { // 读写任意文件,忽视 flags 和 mode
8     Log("Pathname: %s", pathname);
9     int i;
10    for (i = 0; i < NR_FILES; i++) {
11        // printf("file name: %s\n", file_table[i].name);

```

```

12         if (strcmp(file_table[i].name, pathname) == 0) {
13             return i;
14         }
15     }
16     return -1;
17 }
18 ssize_t fs_read(int fd, void *buf, size_t len) {
19     ssize_t fs_size = fs_filesz(fd);
20     // 偏移量不可以超过文件边界 超出部分舍弃
21     if (file_table[fd].open_offset + len > fs_size)
22         len = fs_size - file_table[fd].open_offset;
23     switch(fd) {
24         case FD_STDOUT:
25         case FD_STDERR:
26         case FD_STDIN:
27             return 0;
28         default:
29             ramdisk_read(buf, file_table[fd].disk_offset +
30                 file_table[fd].open_offset, len);
31             file_table[fd].open_offset += len;
32             break;
33     }
34     return len;
35 }
36 ssize_t fs_write(int fd, const void *buf, size_t len) {
37     ssize_t fs_size = fs_filesz(fd);
38     switch(fd) {
39         case FD_STDOUT:
40         case FD_STDERR:
41             for(int i = 0; i < len; i++) {
42                 _putc(((char*)buf)[i]);
43             }
44             break;
45         default:
46             if(file_table[fd].open_offset + len > fs_size)
47                 len = fs_size - file_table[fd].open_offset;
48             // 对文件的真正读写
49             ramdisk_write(buf, file_table[fd].disk_offset +
50                 file_table[fd].open_offset, len);
51             file_table[fd].open_offset += len;
52             break;
53     }
54     return len; // 参见man 返回值
55 }
56 off_t fs_lseek(int fd, off_t offset, int whence) {
57     off_t result = -1;
58     // man 2 lseek 注意边界!
59     switch(whence) {

```

```

60         case SEEK_SET: // open 设置为 offset
61             if (offset >= 0 && offset <= file_table[fd].size) {
62                 file_table[fd].open_offset = offset;
63                 result = file_table[fd].open_offset;
64             }
65             break;
66         case SEEK_CUR: // open=current+offset
67             if ((offset + file_table[fd].open_offset >= 0)
68                 && (offset + file_table[fd].open_offset
69                     <= file_table[fd].size)) {
70                 file_table[fd].open_offset += offset;
71                 result = file_table[fd].open_offset;
72             }
73             break;
74         case SEEK_END: // open=文件大小+offset
75             file_table[fd].open_offset = file_table[fd].size + offset;
76             result = file_table[fd].open_offset;
77             break;
78     }
79     return result;
80 }
81 int fs_close(int fd) {
82     return 0;
83 }

```

除了 fs.c 中的内容, 我们还需要实现各种文件相关的系统调用才能使文件系统正常工作:

文件系统调用

```

1 // nanos-lite/src/syscall.c
2 case SYS_open: c->GPRx=fs_open((void*)a[1], a[2], a[3]); break;
3 case SYS_read: c->GPRx=fs_read(a[1], (void*)a[2], a[3]); break;
4 case SYS_write: c->GPRx=fs_write(a[1], (void*)a[2], a[3]); break;
5 case SYS_close: c->GPRx=fs_close(a[1]); break;
6 case SYS_lseek: c->GPRx=fs_lseek(a[1], a[2], a[3]); break;
7 // navy-apps/libs/libos/src/nanos.c
8 int _open(const char *path, int flags, mode_t mode) {
9     // _exit(SYS_open);
10    return _syscall_(SYS_open, path, flags, mode);
11 } int _read(int fd, void *buf, size_t count) {
12     // _exit(SYS_read);
13     // printf("%d\n", count);
14     return _syscall_(SYS_read, fd, buf, count);
15 }
16
17 int _close(int fd) {
18     // _exit(SYS_close);
19     return _syscall_(SYS_close, fd, 0, 0);
20 }
21

```

```

22 off_t _lseek(int fd, off_t offset, int whence) {
23     // _exit(SYS_lseek);
24     return _syscall_(SYS_lseek, fd, offset, whence);
25 }

```

之后再用 naive_load 加载 text 测试, 可以看到如下结果:

```

[src/monitor/monitor.c,30,welcome] Build time: 16:51:57, May 10 2020
For help, type "help"
(nemu) c
[src/mm.c,40,init_mm] free physical pages starting from 0xd91000
[src/main.c,19,main] 'Hello World!' from Nanos-lite
[src/main.c,20,main] Build time: 16:52:21, May 10 2020
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x102b40, end = 0x1d4cac1,
size = 29663105 bytes
[src/main.c,27,main] Initializing interrupt/exception handler...
[src/fs.c,39,fs_open] Pathname: /bin/text
[src/loader.c,19,loader] Load [8] /bin/text with size: 28480
[src/fs.c,39,fs_open] Pathname: /bin/videotest
[src/loader.c,19,loader] Load [12] /bin/videotest with size: 38800
[src/irq.c,7,do_event] trap event!
[src/fs.c,39,fs_open] Pathname: /share/texts/num
PASS!!!
nemu: HIT GOOD TRAP at eip = 0x00100032

```

图 5: PASS!!

(二) 万物皆文件

接下来的部分是继续完善文件系统的功能, 如前面 PA2.3 实现的一些 IO 功能, 键盘输入, VGA 等等。

串口的写入在 device.c 中由 serial_write() 提供, 实现该函数并完善 VFS 即可, 原理还是 putc。读取输入事件的实现, 步骤和串口写入一致, 完成 events_read() 函数并完善 VFS。

serial&events

```

1 // nanos-lite/src/device.c
2 size_t serial_write(const void *buf, size_t offset, size_t len)
3 { // 串口写入
4     char *b=(char*)buf;
5     for(int i=0; i<len; i++){
6         _putc(*(b++));
7     }
8     return len;
9 }
10 size_t events_read(void *buf, size_t offset, size_t len)
11 { // 输入事件
12     int key=read_key();
13     if(key & 0x8000){
14         sprintf(buf, "kd %s\n", keyname[key & ~0x8000]);
15     }
16     else if ((key & ~0x8000) != _KEY_NONE){
17         sprintf(buf, "ku %s\n", keyname[key & ~0x8000]);
18     }
19     else{
20         sprintf(buf, "t %d\n", uptime());
21     }

```

```

22     return strlen(buf);
23 }
24 // nanos-lite / src / fs . c
25 static Finfo file_table[] __attribute__((used)) = {
26     {"stdin", 0, 0, 0, invalid_read, invalid_write},
27     {"stdout", 0, 0, 0, invalid_read, serial_write},
28     {"stderr", 0, 0, 0, invalid_read, serial_write},
29 #include "files.h"
30     {"/dev/events", 0, 0, 0, events_read, invalid_write},
31     {"/dev/fbsync", 1, 0, 0, invalid_read, fbsync_write},
32     {"/dev/tty", 0, 0, 0, invalid_read, serial_write},
33     {"/proc/dispinfo", 0, 0, 0, dispinfo_read, invalid_write},
34     {"/dev/fb", 0, 0, 0, invalid_read, fb_write},
35 };

```

VGA 抽象成文件, 需要对 frame buffer 的大小进行初始化, 并实现写屏幕的相关函数, 同时完善 VFS。

serial&events

```

1 // nanos-lite / src / fs . c
2 void init_fs() {
3     // TODO: initialize the size of /dev/fb
4     file_table[FD_FB].size = _screen.height * _screen.width * 4;
5 }
6 // nanos-lite / src / device . c
7 size_t fb_write(const void *buf, size_t offset, size_t len)
8 { // 根据 offset 计算 xy 再调用 draw_rect
9     int x = (offset / 4) % screen_width();
10    int y = (offset / 4) / screen_width();
11    draw_rect((uint32_t *)buf, x, y, len / 4, 1);
12    return len;
13 }
14 size_t fbsync_write(const void *buf, size_t offset, size_t len)
15 { // 调 API
16    draw_sync();
17    return 0;
18 }
19 // 初始化 /proc/dispinfo
20 void init_device() {
21    Log("Initializing devices...");
22    _ioe_init();
23
24    // TODO: print the string to array dispinfo with the format
25    // described in the Navy-apps convention
26    sprintf(dispinfo, "WIDTH:%d\nHEIGHT:%d\n", screen_width(), screen_height());
27 }
28 size_t dispinfo_read(void *buf, size_t offset, size_t len) {
29    strncpy(buf, dispinfo + offset, len);
30    return len;

```

31 | }

以上内容都成功实现后, 对 events 和 bmp test 测试就可以看到预期的结果。

```
[src/device.c,27,events_read] Get key: 61 N up
receive event: ku N
[src/device.c,27,events_read] Get key: 55 LSHIFT down
receive event: kd LSHIFT
[src/device.c,27,events_read] Get key: 55 LSHIFT up
receive event: ku LSHIFT
[src/device.c,27,events_read] Get key: 66 RSHIFT down
receive event: kd RSHIFT
[src/device.c,27,events_read] Get key: 66 RSHIFT up
receive event: ku RSHIFT
receive event: t 27976
receive event: t 28602
receive event: t 29303
receive event: t 29938
receive event: t 30562
receive event: t 31174
receive event: t 31775
receive event: t 32397
receive event: t 33002
```

图 6: events

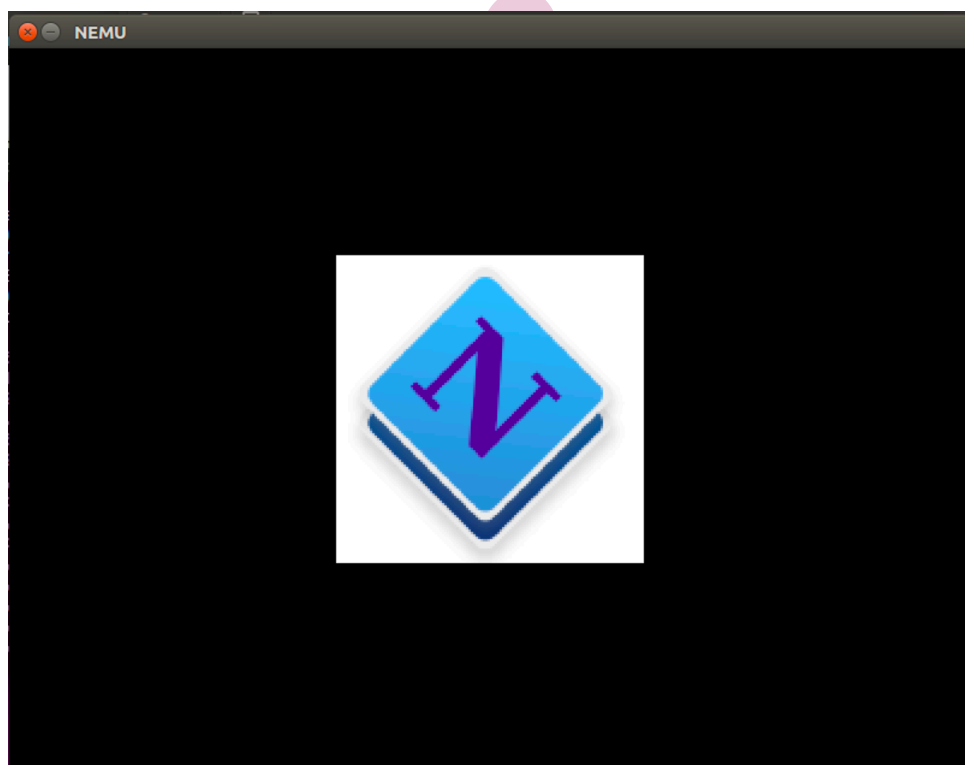


图 7: bmp test

(三) 运行仙剑

仙剑前面都实现完了之后, 仙剑就可以直接运行了。但是 NEMU 真的慢到承受不住, 假装前面都没什么问题吧就。。。由于浮点数还没有实现所以还不可以战斗, 但其实到水月宫战斗前的剧情还是挺长的, 只不过太慢了完全等不到那么久, 运行到去菜市场就假装这里已经好了 orz, 此时的存档似乎也不是很好用, 可能是 mmu 还没有实现的原因。



图 8: pal

(四) DiffTest

实际上 2019 的框架已经非常齐全了, 只需要去掉运行参数-b, 这里 2019 实验书写的路径是错误的, 应该是在 nexus-amamarchplatformnemu.mk 里面, 然后在 monitor 里面实现 detach 和 attach 两个命令, 并对 EFLAGS 和 IDTR 这两个寄存器进行处理, 以跳过这些和 QEMU 真机不太一样的地方。

difftest

```

1 //nemu/src/monitor/ui.c
2 /*cmd_table 加入描述*/
3 { "detach", "Exit DiffTest", cmd_detach }, //New
4 { "attach", "Attach DiffTest with Qemu", cmd_attach }, //New
5 /* 命令实现 */
6 static int cmd_detach(char *args){
7     difftest_detach();
8     return 0;
9 }
10 static int cmd_attach(char *args){
11     difftest_attach();
12     return 0;
13 }
14 //nemu/src/monitor/diff-test.c
15 void difftest_detach() {
16     is_detach = true;
17 }

```



```

18 void difftest_attach() {
19 #ifndef DIFF_TEST
20     return;
21 #endif
22     is_detach = false;
23     is_skip_ref = false;
24     skip_dut_nr_instr = 0;
25     isa_difftest_attach();
26 }
27 //nemu/tools/qemu-diff/src/isa/x86/init.c
28 r.eip = 0x7c00;
29 r.cs = 0x0000;
30 ok = gdb_setregs(&r);
31 assert(ok == 1);

```

(五) 展示批处理系统

实现 `execve` 系统调用使得结束当前程序运行的时候启动一个指定的程序,借助它来实现开始菜单,并修改 `exit` 系统调用使得程序运行结束的时候回到开始菜单。之后运行超级玛丽比仙剑能快那么一点点吧。。

execve

```

1 //nanos-lite/src/syscall.c
2 case SYS_exit:c->GPRx=-1;naive_upload(NULL,"/bin/init");break;//_halt(a[1]);break;
3 case SYS_execve:c->GPRx=-1;naive_upload(NULL,(char*)a[1]);break;

```



图 9: 批处理主界面

由于马里奥运行出来的结果和以前没什么两样就不放图了。

五、 遇到的 bug 及解决

PA3 重写了三次,最后在 PA4 的时候还是因为太菜回到了 2018,放弃了在 2019 的苦苦挣扎,幸好 PA4 没有再重写。

- ☒ 不知道是我理解的问题还是实验书表达的有一点点问题,3.2 的时候应该是要现在 loader 里面实现加载磁盘文件才可以继续后面系统调用等等的完善,不然一定会卡住;同样的理解问题,实验指导书说“trap.S 中实现的结构”但是 trap.S 根本就没写,所以意思应该是 trap.S 中使用的构造的上下文结构的顺序
- ☒ 2019 新增的那个 logo 从一开始的运行就开始报地址的错,然后等我把 logo.txt 的图标大小改小一点就好了?后面好像是因为文件读写长度的问题?等我文件系统改好以后这个就没有问题了
- ☒ lidt 指令里面用的是 id_dest 的地址不是 val
- ☒ text 测试样例那里应该是要先写好 serial_write 才可以
- ☒ PA3 发现了不少 PA2 留下的小坑以及没有补充完整的 opcode_table,还有 PA3 的内容我都是先行在 native 测试的,确定 nanos-lite 的实现没有问题再找前面遗留下的问题
- ☐ 其实仙剑还是有点问题的,不止是战斗无法进行,我的剧情在地图切换后就出现了问题(买鱼没买到回来)就环境变暗卡住了,但是直到 PA5 才发现是前面的指令实现有问题 orz,除了感谢我对仙剑剧情倒背如流能告诉小伙伴正确的剧情操作顺序以外,并让我一下子就知道哪里有 bug,但是这蜗牛爬的运行速度我实在不想改(实际上 PA3 测试的部分到要出客栈门就再没管,因为太慢了这就半个小时了,直觉感官上甚至比零几年的 xp 机还慢)

六、 思考题 && 必答题

(一) 思考题

1. 什么是操作系统

这个问题我只能抄网上,属于只可意会表达不出的概念。操作系统位于计算机硬件与应用软件之间,本质也是一个软件。操作系统由操作系统的内核(运行于内核态,管理硬件资源)以及系统调用(运行于用户态,为应用程序员写的应用程序提供系统调用接口)两部分组成。

操作系统的功能是:进程管理,其主要任务是对处理器的时间进行合理分配、对处理器的运行实施有效的管理;存储器管理,主要任务是对存储器进行分配、保护和扩充;设备管理,根据确定的设备分配原则对设备进行分配,使设备与主机能够并行工作,为用户提供良好的设备使用界面;文件管理,有效地管理文件的存储空间,合理地组织和管理文件系统,为文件访问和文件保护提供更有效的方法及手段;用户接口,通过用户接口,用户只需进行简单操作,就能实现复杂的应用处理。

如果放在 NEMU 上来看,操作系统就是运行在 AM 上的一个特殊的程序,他可以加载其他程序对他们进行资源分配和管理等等。

2. 这些程序状态(x86 的 eflags, cs, eip; mips32 的 epc, status, cause; riscv32 的 sepc, sstatus, scause)必须由硬件来保存吗?能否通过软件来保存?为什么?

软件崩了不就找不回来了

3. 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 calling convention 中需要调用者保存的寄存器. 而 CTE 在保存上下文的时候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.

在函数调用过程中, 寄存器被分为调用者保存寄存器和被调用者保存寄存器, 由于框架作出了相应的规定, 调用者保存寄存器依情况保存, 而在异常处理时, 处于内核态, 操作系统拥有最高的权限, 可以使用更改任何寄存器, 而标志寄存器这些与程序的运行状态息息相关, 所以要保存所以寄存器。

4. x86 的 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? Hint: 不用想太多, 其实都是你学过的知识.

call 指令压栈 esp, 查看 `irq_handle(_RegSet *tf)`, 可能是在准备入口参数 call 指令执行完后, 执行了 `esp+4`, 使 esp 指向了进入陷阱帧之前的位置, 还原了陷阱帧调用前的函数状态

5. 如果你在 GNU/Linux 下执行一个从 Windows 拷过来的可执行文件, 将会报告”格式错误”. 思考一下, GNU/Linux 是如何知道”格式错误”的?

文件头中包含的文件描述符等信息的无法识别等导致的格式错误出现

6. 使用 `readelf` 查看一个 ELF 文件的信息, 你会看到一个 segment 包含两个大小的属性, 分别是 FileSiz 和 MemSiz, 这是为什么? 再仔细观察一下, 你会发现 FileSiz 通常不会大于相应的 MemSiz, 这又是为什么?

分别表示文件大小和在内存中占据空间大小, 那必然文件大小会小

7. 在 `navy-apps/apps/pal/src/game/script.c` 中有一个 `PAL_InterpretInstruction()` 的函数, 尝试大致了解这个函数的作用和行为. 然后大胆猜测一下, 仙剑奇侠传的开发者是如何开发这款游戏的? 你对”游戏引擎”是否有新的认识?

大胆胡乱猜测如下, 该函数用以解释和执行脚本文件中的各个指令。对于不同的操作码对应不同的游戏行为, 大概就是借助这个神奇的形似 CPU exec 部分的对游戏内不同操作码进行翻译和执行。这就大概是个游戏引擎吧。。

8. 假设这些上述这些秘技并非游戏制作人员的本意, 请尝试解释这些秘技为什么能生效.

其实这三个秘技我只尝试成功过后两个, 第一个不知道什么原因玩了几次都没有成功, 再就是网上有不少修改器可以穿墙感觉是同一个原理, 可能是和程序实现中没有考虑临界值的计算以及类型转换导致的数据溢出, 导致如二号人物赵灵儿技能也处于一号人物李道遥的技能范围内了。

(二) 必答题

以下以外的必答题均为代码实现。

1. 你会在 `__am_irq_handle()` 中看到有一个上下文结构指针 `c`, `c` 指向的上下文结构究竟在哪里? 这个上下文结构又是怎么来的? 具体地, 这个上下文结构有很多成员, 每一个成员究竟在哪里赋值的? `$ISA-nemu.h`, `trap.S`, 上述讲义文字, 以及你刚刚在 NEMU 中实现的新指令, 这四部分内容又有什么联系?

指向的结构体在 esp 指向的地方, 这个结构体是前面 `pushal` 构造的, 其成员的定义在 `x86-nemu.h` 中, 包括地址空间、通用寄存器、`irq`、`eip`、`cs` 以及 `eflags`, `irq` 的赋值在 `trap.S` 中对应不同中断事件 `pushal` 对寄存器操作, 然后是 `eip`、`eflags`。 `x86-nemu.h` 定义了 `context`, `trap.S` 是中断发生时的处理过程, 实现的 `popa` 指令用来恢复 `trapframe`

2. 从 Nanos-lite 调用 `_yield()` 开始, 到从 `_yield()` 返回的期间, 这一趟旅程具体经历了什么? 软 (AM, Nanos-lite) 硬 (NEMU) 件是如何相互协助来完成这趟旅程的? 你需要解释这一过程中的每一处细节, 包括涉及的每一行汇编代码/C 代码的行为, 尤其是一些比较关键的指令/变量. 事实上, 上文的必答题”理解上下文结构体的前世今生”已经涵盖了这趟旅程中的一部分, 你可以把它的回答包含进来. 别被”每一行代码”吓到了, 这个过程也就大约 50 行代码, 要完全理解透彻并不是不可能的. 我们之所以设置这道必答题, 是为了强迫你理解清楚这个过程中的每一处细节. 这一理解是如此重要, 以至于如果你缺少它, 接下来你面对 bug 几乎是束手无策.

首先 `_yield` 在 `cte.c` 中实现是通过 `int 81` 实现的, `int` 指令借助 `raise_intr` 实现对中断号 81 的处理, 根据 `trap.S` 将上下文设置好之后调用 `__am_irq_handle`, 按照前面对事件的标号进行处理 (输出一句话) 然后返回并恢复上下文, 再返回到调用 `_yield` 的地方。

3. 我们知道 `navy-apps/tests/hello/hello.c` 只是一个 C 源文件, 它会被编译链接成一个 ELF 文件. 那么, `hello` 程序一开始在哪里? 它是怎么出现内存中的? 为什么会出现在目前的内存位置? 它的第一条指令在哪里? 究竟是怎么执行到它的第一条指令的? `hello` 程序在不断地打印字符串, 每一个字符又是经历了什么才会最终出现在终端上?

程序编译出之后在 `hello` 文件夹下的 `build` 文件夹里, 出现的具体步骤见 `Makefile.app` 是链接所有库文件之后编译生成的 ELF 文件, 因为 `Makefile` 里设定了它要放在这里, ELF 的 `header` 之后就是, 根据加载的 ELF 文件, 其头部中指示了代码段的位置即可执行, 经历了不停的调用 `write` 和 `brk`

七、 总结

PA3 代码感觉不多但是却处处致命, 虽然挺早就写完了 PA3 但是直到 PA5 我还发现了 PA2 和 PA3 的一些小问题, 费时在自己给自己挖坑上。

虽然整体时间不是很长, 但是如果算上后面部分的调试时间还是有些多, 只能怪罪于自己没能完全理解实验报告, 或者基础理论知识还不够透彻上吧。DiffTest 虽然提前写好了但其实没怎么用, 还是靠着报错提示去找, 因为 `diffTest` 的实现在 PA3 阶段还有点小问题 (和 QEMU 的协同上)。

整体上又跟随 PA 成长了许多, 但是拿仙剑做例子总是让人忍不住去玩, 幸好 NEMU 极慢极慢的运行打消了我这个念头, 然后 PA5 的优化目前并没有什么头绪, 希望在重览实验指导后能摸清一些门路。