



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：李颖

学号：2110939

专业：计算机科学与技术

2024 年 5 月 14 日

目录

1 实验介绍	2
1.1 实验目的	2
1.2 实验内容	2
2 阶段一	2
2.0.1 实现 loader	2
2.1 实现中断机制	3
2.1.1 准备 IDTR	3
2.1.2 实现 lidt 指令	3
2.1.3 实现 int 指令	4
2.1.4 Q: 对比异常与函数调用	5
2.1.5 重新组织 trapframe 结构体	6
2.2 系统调用处理	7
2.2.1 识别系统调用事件	7
2.2.2 实现 SYS_none	8
2.2.3 实现 iret 指令	9
2.3 PA2 中遗漏的指令	10
2.3.1 endbr 指令	10
2.3.2 2-byte	10
3 阶段二	11
3.1 实现标准输出	11
3.2 实现堆区管理	12
3.3 实现简易文件系统	13
3.3.1 完善文件记录表相关操作	13
3.3.2 让 loader 使用文件	15
3.3.3 实现完整的文件系统	15
4 阶段三	18
4.1 把 VGA 显存抽象成文件	18
4.2 把输入设备抽象成文件	20
4.3 运行仙剑奇侠传	21
5 必答题	22
6 BUG 汇总	23
6.1 搭建的环境无法改成 x86	23
6.2 BAD TRAP 时没有输出 panic 信息	24

1 实验介绍

1.1 实验目的

PA2 中已实现在 AM 上运行小游戏，但由于缺少文件系统以及异常控制流，还无法运行较为复杂的庞大应用。因此，在 PA3 中，需要对这方面进行完善，使《仙剑奇侠传》游戏能正常运行。

1.2 实验内容

补充 Nanos-lite 操作系统的相关代码，实现在 NEMU 上运行该操作系统。实现加载器、中断机制以及系统调用，实现文件系统。

2 阶段一

加载操作系统的第一个用户程序，需要实现 loader。Navy-apps 子项目是专门用于编译操作系统上的用户程序的，这样可以解决用户程序与 AM 所提供运行时环境不匹配的问题。

首先修改 makefile 文件，让 Navy-apps 项目程序编译到 x86 上。执行 make 等一系列操作后，生成 ramdisk 镜像文件，该文件存放 loader 需要加载的内容，因此，目前的 loader 需要将 ramdisk 中从 0 开始的所有内容放置在 0x40000000，并将该地址作为程序的入口地址返回。

2.0.1 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。

在 loader.c 中实现 loader() 函数，利用 ramdisk_read() 函数将用户程序读入特定位置中。其中，需要用到的函数和宏定义需要使用 extern 进行声明。

```

1 extern uint8_t ramdisk_start;
2 extern uint8_t ramdisk_end;
3 #define RAMDISK_SIZE ((amp;ramdisk_end) - (amp;ramdisk_start))
4 extern void ramdisk_read(void *buf, off_t offset, size_t len);
5
6 uintptr_t loader(_Protect *as, const char *filename) {
7     ramdisk_read(DEFAULT_ENTRY, 0, RAMDISK_SIZE);
8     return (uintptr_t)DEFAULT_ENTRY;
9 }

```

运行后显示出如下界面，发现是有指令未实现。查看手册后，发现 cd 对应的指令为 int 中断指令。

```

[src/monitor/monitor.c,30,welcome] Build time: 01:04:59, May 8 2024
For help, type "help"
(nemu) c
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
OVM Metanul

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

```

2.1 实现中断机制

2.1.1 准备 IDTR

为了实现 int 指令，需要准备好 IDT，这样触发异常时才能跳转到准备好的地址。

首先，需要在 NEMU 中添加 IDTR 寄存器，如下所示。这样，在 main.c 中定义宏 HAS_ASYE 后，nanos-lite 就会间接调用 __asye_init() 函数对 IDT 进行初始化。

```
1 struct IDTR{
2     uint32_t base;
3     uint16_t limit;
4 } idtr;
```

2.1.2 实现 lidt 指令

之后，要实现 lidt 指令。该指令会将 idtr 的首地址和长度写入寄存器，调用 __asm_lidt() 函数设置 IDT。

具体流程同 PA2，涉及的代码如下所示。其中，exec.c 的内容以及 decode.h 中的注册已在 PA2 中实现。

```
1 /* 0x0f 0x01*/
2 make_group(gp7,
3     EMPTY, EMPTY, EMPTY, IDEX(lidt_a, lidt),
4     EMPTY, EMPTY, EMPTY, EMPTY)
```

具体的执行函数在本次实验中完善，即把操作数信息从 eax 寄存器中读取出来。

```
1 make_EHelper(lidt) {
2     // TODO();
3     t1 = id_dest -> val;
4     rtl_lm(&t0, &t1, 2);
5     cpu.idtr.limit = t0;
6
7     t1 = id_dest -> val + 2;
8     rtl_lm(&t0, &t1, 4);
9     cpu.idtr.base = t0;
10
11 #ifdef DEBUG
12     Log("idtr.limit=0x%x", cpu.idtr.limit);
13     Log("idtr.base=0x%x", cpu.idtr.base);
14 #endif
15     print_asm_template1(lidt);
16 }
```

此外，还需要在 reg.h 中设定 cs 寄存器，用于存储当前代码段的起始地址。

```
1 rtlreg_t cs;
```

PA 中不实现分段机制，没有 CS 寄存器的概念，但为了在 QEMU 中顺利进行 differential testing，

还是需要在 cpu 结构体中添加一个 CS 寄存器, 并在 restart() 函数中将其初始化为 8。

由于中断机制需要对 EFLAGS 进行压栈, 为了配合 differential testing, 我们还需要在 restart() 函数中将 EFLAGS 初始化为 0x2。

```

1 static inline void restart() {
2     /* Set the initial instruction pointer. */
3     cpu.eip = ENTRY_START;
4     cpu.cs = 8;
5     unsigned int origin = 2;
6     memcpy(&cpu.eflags, &origin, sizeof(cpu.eflags));
7
8 #ifdef DIFF_TEST
9     init_qemu_reg();
10 #endif
11 }

```

为了验证是否已经成功准备 IDT, 需要真正触发一次异常, 看是否能正确跳转到目标地址, 因此, 需要在 intr.c 中实现 raise_intr() 函数。

```

1 void raise_intr(uint8_t NO, vaddr_t ret_addr) {
2     memcpy(&t1, &cpu.eflags, sizeof(cpu.eflags));
3     rtl_li(&t0, t1);
4     rtl_push(&t0);
5     rtl_push(&cpu.cs);
6     rtl_li(&t0, ret_addr);
7     rtl_push(&t0);
8     vaddr_t gate_addr = cpu.idtr.base + NO * sizeof(GateDesc);
9     // Log("%d %d %d\n", gate_addr, cpu.idtr.base, cpu.idtr.limit);
10    assert(gate_addr <= cpu.idtr.base + cpu.idtr.limit);
11
12    uint32_t off_15_0 = vaddr_read(gate_addr, 2);
13    uint32_t off_32_16 = vaddr_read(gate_addr + sizeof(GateDesc) - 2, 2);
14    uint32_t target_addr = (off_32_16 << 16) + off_15_0;
15 #ifdef DEBUG
16    Log("target_addr=0x%x", target_addr);
17 #endif
18    decoding.is_jump = 1;
19    decoding.jump_eip = target_addr;
20 }

```

2.1.3 实现 int 指令

之后就可以进行 int 指令的实现。

完善 optable 并对其注册, 如下所示。

```

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

```

```

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

```

```

2 make_EHelper(int) {
3     // TODO();
4
5     uint8_t NO = id_dest -> val & 0xff;
6     raise_intr(NO, decoding.seq_eip);
7     print_asm("int %s", id_dest->str);
8
9 #ifdef DIFF_TEST
10     diff_test_skip_nemu();
11 #endif
12 }

```

运行后发现 fa 处指令未实现，经查发现是 cti 指令，应该是在 PA2 中遗漏的指令。

```

[src/cpu/intr.c,24,raise_intr] target_addr=0x0
invalid opcode(eip = 0x000078f4): fa ff ff 48 79 00 00 ...

There are two cases which will trigger this unexpected exception:
1. The instruction at eip = 0x000078f4 is not implemented.
2. Something is implemented incorrectly.
Find this eip(0x000078f4) in the disassembling result to distinguish which case it is.

If it is the first case, see

```

2.1.4 Q: 对比异常与函数调用

在函数调用和异常处理中，都需要保存一些状态信息以便在处理完之后能够正确地返回到原来的执行流程。然而，异常处理需要保存的信息通常比函数调用更多，这是由于异常处理的特性决定的。

通过对比，可得到以下异同点：

- 函数调用保存信息
 - 返回地址：函数调用时需要保存返回地址，以便在函数执行完毕后返回到调用点继续执行。
 - 调用约定中需要保存的寄存器：根据调用约定，调用者和被调用者之间需要协商哪些寄存器的值需要保存和恢复。
- 异常处理保存信息
 - 异常类型和原因：需要保存异常的类型和原因，以便后续的异常处理程序能够正确地处理异常。
 - 中断向量号：用于确定触发异常的具体原因。
 - 程序状态：保存当前程序的状态，包括寄存器值、程序计数器等，以便在异常处理完毕后能够恢复到异常发生前的状态。
 - 错误码：有些异常会伴随错误码，需要保存错误码以便分析异常原因。
 - 中断帧：保存当前中断的上下文信息，包括寄存器值、栈指针等。

函数调用和异常处理保存信息不同的主要原因在于它们的目的和处理方式不同：

函数调用的目的是在函数执行完毕后能够返回到调用点继续执行，因此只需要保存返回地址和调用约定中需要保存的寄存器。函数调用是程序正常执行的一部分，通常不会涉及到错误或异常情况。

异常处理的目的是在程序出现异常情况时能够正确地处理异常并恢复到正常执行流程，因此需要保存更多的信息来处理异常情况。异常处理涉及到错误、中断、异常等不正常情况，需要保存异常类型、错误码、中断帧等额外信息来确保能够正确地处理异常并恢复程序状态。

2.1.5 重新组织 trapframe 结构体

运行后发现有无实现的指令 60，查询后可知是 pusha 指令，从而可以推断 popa 指令也未实现。

```
(nemu) c
[src/cpu/exec/system.c,18,exec_ldt] idtr.limit=0x7fff
[src/cpu/exec/system.c,19,exec_ldt] idtr.base=0x1054a0
[src/cpu/intr.c,29,raise_intr] target_addr=0x100ac7
invalid opcode(eip = 0x00100ad6): 60 54 e8 03 ff ff ff 83 ...
```

首先，需要实现 pusha 指令，并注意压栈的顺序。该指令作用为将所有的寄存器压栈。popa 同理，在此一并解决。

补充 optable，并在 all-instr.h 中注册。

```
1 /* 0x60 */ EX(pusha), EX(popa), EMPTY, EMPTY,
```

补充执行函数。

```
1 make_EHelper(pusha) {
2     t0 = cpu.esp;
3     rtl_push(&cpu.eax);
4     rtl_push(&cpu.ecx);
5     rtl_push(&cpu.edx);
6     rtl_push(&cpu.ebx);
7     rtl_push(&t0);
8     rtl_push(&cpu.ebp);
9     rtl_push(&cpu.esi);
10    rtl_push(&cpu.edi);
11
12    print_asm("pusha");
13 }
14
15 make_EHelper(popa) {
16     rtl_pop(&cpu.edi);
17     rtl_pop(&cpu.esi);
18     rtl_pop(&cpu.ebp);
19     rtl_pop(&t0);
20     rtl_pop(&cpu.ebx);
21     rtl_pop(&cpu.edx);
22     rtl_pop(&cpu.ecx);
23     rtl_pop(&cpu.eax);
24     print_asm("popa");
25 }
```

查看 trap.S 的代码，可以得到以下信息：

- pushl \$0：将一个值为 0 的双字（4 字节）压入栈中，可能表示中断发生时的错误码。
- pushl \$0x80 或 pushl \$-1：将一个特定值（0x80 或 -1）的双字压入栈中，可能表示中断号或系统调用号。
- pushal：将所有通用寄存器的值压入栈中，保存了中断发生时的程序状态。

- `pushl %esp`: 将栈指针`%esp` 的值压入栈中, 可能用于保存当前栈指针的值。
- `call irq_handle`: 调用 `irq_handle` 函数进行中断处理, 可能会修改一些寄存器的值。
- `addl $4, %esp`: 调整栈指针, 可能用于清除之前压栈的错误码和中断号。
- `popal`: 弹出之前保存的所有通用寄存器的值, 恢复中断处理前的程序状态。
- `addl $8, %esp`: 调整栈指针, 清除之前压栈的内容。

实现指令以后, 重新重新组织 `arch.h` 中定义的 `_RegSet` 结构体的成员, 使得这些成员声明的顺序和 `trap.S` 中构造的 `trap frame` 保持一致, 修改后代码如下所示。

```
1 struct _RegSet {
2     // uintptr_t esi, ebx, eax, eip, edx, error_code, eflags, ecx, cs, esp, edi, ebp;
3     uintptr_t edi, esi, ebp, esp, ebx, edx, ecx, eax;
4     int      irq;
5     uintptr_t error_code;
6     uintptr_t eip;
7     uintptr_t cs;
8     uintptr_t eflags;
9 };
```

运行后显示触发 `BAD TRAP: ID=8`。这是一个未处理的 8 号事件, 其实是一个系统调用事件 `_EVENT_SYSCALL`。

```
[src/cpu/exec/system.c,19,exec_ldt] idtr.limit=0x7fff
[src/cpu/exec/system.c,20,exec_ldt] idtr.base=0x105500
[src/cpu/intr.c,17,raise_intr] 1071360 1070336 2047

[src/cpu/intr.c,24,raise_intr] target_addr=0x100b18
[src/irq.c,5,do_event] system panic: Unhandled event ID = 8
nemu: HIT BAD TRAP at eip = 0x00100032
(nemu) █
```

2.2 系统调用处理

但目前 `Nanos-lite` 没有实现任何系统调用, 因此触发了 `panic`。因此, 接下来需要添加系统调用。

2.2.1 识别系统调用事件

首先需要在 `do_event()` 中识别出系统调用事件 `_EVENT_SYSCALL`, 然后调用 `do_syscall()`。

```
1 extern _RegSet* do_syscall(_RegSet *r);
2 static _RegSet* do_event(_Event e, _RegSet* r) {
3     switch (e.event) {
4         case _EVENT_SYSCALL:
5             return do_syscall(r);
6         default: panic("Unhandled event ID = %d", e.event);
7     }
8
9     return NULL;
10 }
```

运行后可以发现, `syscall ID=0`, 即 `dummy` 触发了号码为 0 的 `SYS_none` 系统调用。


```

Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 19:14:04, May 10 2024
For help, type "help"
(nemu) [src/main.c,19,main] 'Hello World!' from Nanos-lite
[src/main.c,20,main] Build time: 20:39:55, May 10 2024
[src/randisk.c,26,init_randisk] randisk info: start = 0x100fa0, end = 0x10557c,
size = 17884 bytes
[src/main.c,27,main] Initializing interrupt/exception handler...
[src/cpu/exec/system.c,19,exec_ldtr] ldtr.limit=0x7ff
[src/cpu/exec/system.c,20,exec_ldtr] ldtr.base=0x1055a0
[src/cpu/intr.c,17,raise_intr] 1071520 1070496 2047

[src/cpu/intr.c,24,raise_intr] target_addr=0x100b60
[src/syscall.c,10,do_syscall] system panic: Unhandled syscall ID = 0
nemu: HIT BAD TRAP at eip = 0x00100032

```

2.2.2 实现 SYS_none

之后，需要在 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。

```

1 #define SYSCALL_ARG1(r) (r->eax)
2 #define SYSCALL_ARG2(r) (r->ebx)
3 #define SYSCALL_ARG3(r) (r->ecx)
4 #define SYSCALL_ARG4(r) (r->edx)

```

在 syscall.c 的 SYS_none 函数中添加系统调用，并设置系统调用的返回值。

```

1 int sys_none() {
2     return 1;
3 }
4
5 __RegSet* do_syscall(__RegSet *r) {
6     uintptr_t a[4];
7     a[0] = SYSCALL_ARG1(r);
8     a[1] = SYSCALL_ARG2(r);
9     a[2] = SYSCALL_ARG3(r);
10    a[3] = SYSCALL_ARG4(r);
11
12    switch (a[0]) {
13        case SYS_none:
14            SYSCALL_ARG1(r) = sys_none();
15            break;
16        default: panic("Unhandled syscall ID = %d", a[0]);
17    }
18
19    return NULL;
20 }

```

再次运行后，出现了未实现的指令 cf，查询后可知是 iret 指令。

```

Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 19:14:04, May 10 2024
For help, type "help"
(nemu) [src/main.c,19,main] 'Hello World!' from Nanos-lite
[src/main.c,20,main] Build time: 20:39:55, May 10 2024
[src/randisk.c,26,init_randisk] randisk info: start = 0x100fc4, end = 0x1055a0,
size = 17884 bytes
[src/main.c,27,main] Initializing interrupt/exception handler...
[src/cpu/exec/system.c,19,exec_ldtr] ldtr.limit=0x7ff
[src/cpu/exec/system.c,20,exec_ldtr] ldtr.base=0x1055c0
[src/cpu/intr.c,17,raise_intr] 1071552 1070528 2047

[src/cpu/intr.c,24,raise_intr] target_addr=0x100b84
invalid opcode(eip = 0x00100ba1): cf 00 00 6f 66 66 73 65 ...

```

2.2.3 实现 iret 指令

对于 iret 指令，实现流程如下所示。

补充 opcode_table，并注册。

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

完善执行函数。

```
1 make_EHelper(iret) {
2     rtl_pop(&t0);
3     decoding.jump_eip = t0;
4     decoding.is_jump = 1;
5     rtl_pop(&t0);
6     cpu.cs = t0;
7     rtl_pop(&t0);
8     cpu.eflags = t0;
9     print_asm("iret");
10 }
```

运行结果如下所示，触发 4 号系统调用。

```
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 19:14:04, May 10 2024
For help, type 'help'
(nemu) [src/main.c,19,main] 'Hello World!' from Nanos-lite
[src/main.c,20,main] Build time: 20:39:55, May 10 2024
[src/randisk.c,26,init_randisk] randisk info: start = 0x100fc4, end = 0x1055a0,
size = 17884 bytes
[src/main.c,27,main] Initializing interrupt/exception handler...
[src/cpu/exec/system.c,19,exec_ldtr] ldtr.limit=0x7fff
[src/cpu/exec/system.c,20,exec_ldtr] ldtr.base=0x1055c0
[src/cpu/intr.c,17,raise_intr] 1071552 1070528 2047

[src/cpu/intr.c,24,raise_intr] target_addr=0x100b84
[src/cpu/intr.c,17,raise_intr] 1071552 1070528 2047

[src/cpu/intr.c,24,raise_intr] target_addr=0x100b84
[src/syscall.c,20,do_syscall] system panic: Unhandled syscall ID = 4
nemu: HIT BAD TRAP at eip = 0x00100032
```

此时 dummy 主函数已成功运行，需要调用 exit 返回，需要触发 SYS_exit，即接收一个退出状态的参数，用该参数调用 __halt()。

```
1 void sys_exit(int a){
2     __halt(a);
3 }
4
5 __RegSet* do_syscall(__RegSet *r) {
6     // 无关代码已省略
7     switch (a[0]) {
8         // 无关代码已省略
9         case SYS_exit:
10             sys_exit(a[1]);
11             break;
12     }
```

运行后可发现到达 GOOD TRAP，说明系统调用已成功实现。此外，打开 DIFF_TEST 注释后，发现也能正常运行至 GOOD TRAP 处。

```

Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 19:14:04, May 10 2024
For help, type "help"
(nemu) [src/main.c,19,main] 'Hello World!' from Nanos-lite
[src/main.c,20,main] Build time: 20:39:55, May 10 2024
[src/randisk.c,26,init_randisk] randisk info: start = 0x100fe8, end = 0x1055c4,
size = 17884 bytes
[src/main.c,27,main] Initializing interrupt/exception handler...
[src/cpu/exec/system.c,19,exec_ldt] idtr.limit=0x7fff
[src/cpu/exec/system.c,20,exec_ldt] idtr.base=0x105600
[src/cpu/intr.c,17,raise_intr] 1071616 1070592 2047

[src/cpu/intr.c,24,raise_intr] target_addr=0x100ba8
[src/cpu/intr.c,17,raise_intr] 1071616 1070592 2047

[src/cpu/intr.c,24,raise_intr] target_addr=0x100ba8
nemu: HIT GOOD TRAP at eip = 0x00100032

```

阶段一到此结束。

2.3 PA2 中遗漏的指令

2.3.1 endbr 指令

补充 opcode_table, 并注册。

```
1 /* 0xf0 */ EMPTY, EMPTY, EMPTY, EXW(endbr, 3),
```

完善执行函数。

```
1 make_EHelper(endbr) {
2     instr_fetch(eip, id_src -> width);
3     print_asm("endbr32");
4 }
```

2.3.2 2-byte

补充 opcode_table 并注册。

```
1 /* 0x20 */ IDEX(mov_load_cr, mov), EMPTY, IDEX(mov_store_cr, mov_store_cr), EMPTY,
```

实现译码函数：

```
1 make_DHelper(mov_load_cr) {
2     decode_op_rm(eip, id_dest, false, id_src, false);
3     rtl_load_cr(&id_src -> val, id_src -> reg);
4 #ifdef DEBUG
5     snprintf(id_src -> str, 5, "%cr%d", id_dest -> reg);
6 #endif
7 }
8
9 make_DHelper(mov_store_cr) {
10    decode_op_rm(eip, id_src, true, id_dest, false);
11 #ifdef DEBUG
12    snprintf(id_src -> str, 5, "%cr%d", id_dest -> reg);
13 #endif
14 }
```

补充 rtl:

```

1 static inline rtl_load_cr(rtlreg_t* dest, int r) {
2     switch (r)
3     {
4     case 0:
5         *dest = cpu.CR0;
6         return;
7         break;
8     case 3:
9         *dest = cpu.CR3;
10        return;
11    default:
12        assert(0);
13    }
14    return;
15 }

```

3 阶段二

在这一阶段，我们需要再操作系统上运行能够循环打印输出的 Hello world 程序。

3.1 实现标准输出

Navy-apps 中提供了一个 hello 测试程序 (navy-apps/tests/hello), 它首先通过 write() 来输出一句话, 然后通过 printf() 来不断输出. 为了运行它, 我们只需要再实现 SYS_write 系统调用即可。

辅助函数和对应的系统调用如下所示。

```

1 int sys_write(int fd, void *buf, size_t len) {
2     if(fd == 1 || fd == 2){
3         char c;
4         for(int i = 0; i < len; i++) {
5             memcpy(&c, buf + i, 1);
6             _putc(c);
7         }
8         return len;
9     }
10    else{
11        panic("Unhandled fd=%d in sys_write()", fd);
12    }
13    return -1;
14 }
15
16 __RegSet* do_syscall(__RegSet *r) {
17 //无关代码已省略
18 switch (a[0]) {
19     //无关代码已省略
20     case SYS_write:
21         SYSCALL_ARG1(r) = sys_write(a[1], (void*)a[2], a[3]);

```

```

22     break;
23 }

```

同时，需要在 nanos.c 中修改辅助函数 __write()。

```

1 int __write(int fd, void *buf, size_t count){
2 return __syscall__(SYS_write, fd, (uintptr_t)buf, count);
3 }

```

修改 ramdisk 的生成规则后，重新运行，得到以下结果。可以发现系统能够持续输出”Hello world!”，说明能成功实现标准输出。

```

[src/cpu/exec/system.c,20,exec_ldt] ldtr.base=0x105840
[src/syscall.c,16,sys_write] buffer:Hello World!

Hello World!
[src/syscall.c,16,sys_write] buffer:H
H[src/syscall.c,16,sys_write] buffer:e
e[src/syscall.c,16,sys_write] buffer:l
l[src/syscall.c,16,sys_write] buffer:l
l[src/syscall.c,16,sys_write] buffer:o
o[src/syscall.c,16,sys_write] buffer:

```

3.2 实现堆区管理

malloc()/free() 库函数的作用是在用户程序的堆区中申请/释放一块内存区域。调整堆区大小是通过 sbrk() 库函数来实现的，它的原型是 void* sbrk((intptr_t increment)，用于将用户程序的 program break 增长 increment 字节，其中 increment 可为负数。

首先需要在 Nanos-lite 中实现 SYS_brk 系统调用，然后在用户层实现 __sbrk()。由于目前 Nanos-lite 为单任务操作系统，因此用户程序可随意使用内存，堆区大小调整总是成功，因此 SYS_brk 总返回 0。

```

1 int sys_brk(int addr) {
2     return 0;
3 }
4
5 _RegSet* do_syscall(_RegSet *r) {
6     //无关代码已省略
7     switch (a[0]) {
8         //无关代码已省略
9         case SYS_brk:
10             SYSCALL_ARG1(r) = sys_brk(a[1]);
11             break;
12     }

```

之后，需要在用户层实现 __sbrk()。probreak 最初位于 __end 标志处，需要根据记录的 probreak 位置和参数 increment 计算出新的 probreak，并通过 SYS_brk 系统调用更新，若成功则返回 0，更新 probreak 的位置，并将旧值返回；失败则返回-1。

```

1 void *__sbrk(intptr_t increment){
2     extern int end;
3     static uintptr_t probreak = (uintptr_t)&end;
4     uintptr_t probreak_new = probreak + increment;

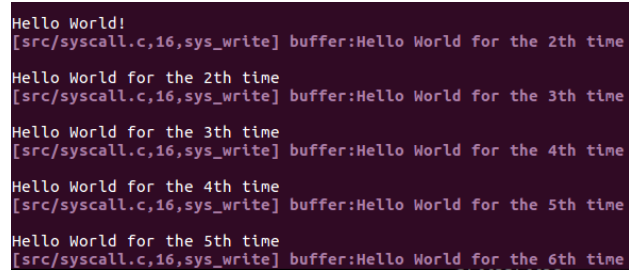
```

```

5  int r = __syscall__(SYS_brk, probreak_new, 0, 0);
6  if(r == 0) {
7      uintptr_t temp = probreak;
8      probreak = probreak_new;
9      return (void*)temp;
10 }
11 return (void *)-1;
12 }

```

注释掉所有 log 后重新运行，发现字是一起打印出来的，可与未实现堆区时逐字打印形成对比。



```

Hello World!
[src/syscall.c,16,sys_write] buffer:Hello World for the 2th time
Hello World for the 2th time
[src/syscall.c,16,sys_write] buffer:Hello World for the 3th time
Hello World for the 3th time
[src/syscall.c,16,sys_write] buffer:Hello World for the 4th time
Hello World for the 4th time
[src/syscall.c,16,sys_write] buffer:Hello World for the 5th time
Hello World for the 5th time
[src/syscall.c,16,sys_write] buffer:Hello World for the 6th time

```

3.3 实现简易文件系统

目前 ramdisk 中只有一个文件，因此使用并不繁琐。但当文件数量变多时，就需要知道指定文件在 ramdisk 中的位置，因此，需要在 Nanos-lite 中实现一个文件系统。

3.3.1 完善文件记录表相关操作

需要一张文件记录表（数据结构定义为 Finfo）来记录 ramdisk 中每个文件的名字和大小。首先修改 makefile，然后在 nanos-lite 中进行相应的更新操作。

代码中定义了 Finfo 数组 file_table，用于维护 ramdisk 中所有的文件信息。因此，需要设置一些辅助函数来对 Finfo 结构中的变量进行相应的操作，如下所示。

在 fs.h 中注册所需函数。

```

1  size_t fs_filesiz(int fd);
2  int fs_open(const char* filename, int flags, int mode);
3  ssize_t fs_read(int fd, void *buf, size_t len);
4  ssize_t fs_write(int fd, void *buf, size_t len);
5  int fs_close(int fd);
6  off_t fs_lseek(int fd, off_t offset, int whence);

```

在 fs.c 中补充 Finfo 对应的辅助函数，同时用 extern

```

1  extern void ramdisk_read(void *buf, off_t offset, size_t len);
2  extern void ramdisk_write(const void *buf, off_t offset, size_t len);
3
4  size_t fs_filesiz(int fd) {
5      assert(fd >= 0 && fd < NR_FILES);
6      return file_table[fd].size;
7  }
8

```

```

9 off_t disk_offset(int fd){
10     assert(fd >= 0 && fd < NR_FILES);
11     return file_table[fd].disk_offset;
12 }
13
14 off_t get_open_offset(int fd){
15     assert(fd >= 0 && fd < NR_FILES);
16     return file_table[fd].open_offset;
17 }
18
19 void set_open_offset(int fd, off_t n){
20     assert(fd >= 0 && fd < NR_FILES);
21     assert(n >= 0);
22     if(n > file_table[fd].size) {
23         n = file_table[fd].size;
24     }
25     file_table[fd].open_offset = n;
26 }

```

此外，还需要补充 fs_open() 函数，逻辑为根据文件名查找文件描述符 fd。

```

1 int fs_open(const char*filename, int flags, int mode) {
2     for(int i = 0; i < NR_FILES; i++){
3         if(strcmp(filename, file_table[i].name) == 0) {
4             Log("success open:%d:%s", i, filename);
5             return i;
6         }
7     }
8     panic("this file not exist");
9     return -1;
10 }

```

补充 fs_read() 函数，用于读取 fd 对应的文件。

```

1 ssize_t fs_read(int fd, void *buf, size_t len){
2     assert(fd >= 0 && fd < NR_FILES);
3     if(fd < 3 || fd == FD_FB) {
4         Log("arg invalid:fd<3");
5         return 0;
6     }
7     int n = fs_fliesz(fd) - get_open_offset(fd);
8     if(n > len) {
9         n = len;
10    }
11    ramdisk_read(buf, disk_offset(fd) + get_open_offset(fd), n);
12    set_open_offset(fd, get_open_offset(fd) + n);
13    return n;
14 }

```

补充 `fs_close()` 函数，用于关闭文件。

```
1 int fs_close(int fd) {
2     assert(fd >= 0 && fd < NR_FILES);
3     return 0;
4 }
```

3.3.2 让 loader 使用文件

做完上述处理后，需要在 `loader.c` 中添加头文件 `fs.h`，并修改 `loader` 函数，使其能加载 `text` 文件。

```
1 uintptr_t loader(_Protect *as, const char *filename) {
2     int fd = fs_open(filename, 0, 0);
3     Log("filename=%s,fd=%d", filename, fd);
4     fs_read(fd, DEFAULT_ENTRY, fs_filesz(fd));
5     fs_close(fd);
6     return (uintptr_t)DEFAULT_ENTRY;
7 }
```

最后，在 `main.c` 中加载用户程序和函数返回地址。

```
1 uint32_t entry = loader(NULL, "/bin/text");
2 ((void (*)(void))entry)();
```

3.3.3 实现完整的文件系统

实现 `fs_write()` 和 `fs_lseek()`，用于对 `fd` 对应的文件进行写操作，以及实现修改 `fd` 对应文件的 `open_offset` 功能。

```
1 ssize_t fs_write(int fd, void *buf, size_t len){
2     assert(fd >= 0 && fd < NR_FILES);
3     if(fd < 3 || fd == FD_DISPINFO) {
4         Log("arg invalid:fd<3");
5         return 0;
6     }
7     int n = fs_filesz(fd) - get_open_offset(fd);
8     if(n > len) {
9         n = len;
10    }
11    if(fd == FD_FB){
12        fb_write(buf, get_open_offset(fd), n);
13    }
14    else {
15        ramdisk_write(buf, disk_offset(fd) + get_open_offset(fd), n);
16    }
17    set_open_offset(fd, get_open_offset(fd) + n);
18    return n;
19 }
20
```



```

21 off_t fs_lseek(int fd, off_t offset, int whence) {
22     switch(whence) {
23         case SEEK_SET:
24             set_open_offset(fd, offset);
25             return get_open_offset(fd);
26         case SEEK_CUR:
27             set_open_offset(fd, get_open_offset(fd) + offset);
28             return get_open_offset(fd);
29         case SEEK_END:
30             set_open_offset(fd, fs_filesz(fd) + offset);
31             return get_open_offset(fd);
32         default:
33             panic("Unhandled whence ID = %d", whence);
34             return -1;
35     }
36 }

```

之后，需要在 syscall.c 中实现对应的系统调用，如下所示。

```

1  int sys_open(const char *pathname){
2      return fs_open(pathname, 0, 0);
3  }
4
5  int sys_read(int fd, void *buf, size_t len){
6      return fs_read(fd, buf, len);
7  }
8
9  int sys_lseek(int fd, off_t offset, int whence) {
10     return fs_lseek(fd, offset, whence);
11 }
12
13 int sys_close(int fd){
14     return fs_close(fd);
15 }
16
17 _RegSet* do_syscall(_RegSet *r) {
18     //无关代码已省略
19     switch (a[0]) {
20         //无关代码已省略
21         case SYS_read:
22             SYSCALL_ARG1(r) = sys_read(a[1], (void*)a[2], a[3]);
23             break;
24         case SYS_open:
25             SYSCALL_ARG1(r) = sys_open((char*) a[1]);
26             break;
27         case SYS_close:
28             SYSCALL_ARG1(r) = sys_close(a[1]);
29             break;
30         case SYS_lseek:

```

```

31     SYSCALL_ARG1(r)=sys_lseek(a[1],a[2],a[3]);
32     break;
33 }

```

此外，还需要到 nanos.c 中修改对应的辅助函数。

```

1  int _open(const char *path, int flags, mode_t mode) {
2      // _exit(SYS_open);
3      return _syscall_(SYS_open, (uintptr_t)path, flags, mode);
4  }
5
6  int _read(int fd, void *buf, size_t count) {
7      // _exit(SYS_read);
8      return _syscall_(SYS_read, fd, (uintptr_t)buf, count);
9  }
10
11 int _close(int fd) {
12     // _exit(SYS_close);
13     return _syscall_(SYS_close, fd, 0, 0);
14 }
15
16
17 off_t _lseek(int fd, off_t offset, int whence) {
18     // _exit(SYS_lseek);
19     return _syscall_(SYS_lseek, fd, offset, whence);
20 }

```

重新 update 后运行，得到如下结果，可以发现 fd=6 的情况未解决。

```

[src/fs.c,68,fs_open] success open:31:/bin/text
[src/loader.c,17,loader] filenames:/bin/text,fd=31
[src/fs.c,68,fs_open] success open:6:/share/texts/num
[src/syscall.c,25,sys_write] system panic: Unhandled fd=6 in sys_write()
menu: HIT BAD TRAP at eip = 0x00100032

```

将 sys_write() 代码修改后，如下所示。

```

1  int sys_write(int fd, void *buf, size_t len) {
2      if(fd == 1 || fd == 2){
3          char c;
4          Log("buffer:%s", (char*)buf);
5          for(int i = 0; i < len; i++) {
6              memcpy(&c, buf + i, 1);
7              _putc(c);
8          }
9          return len;
10     }
11     // else
12     // panic("Unhandled fd=%d in sys_write()",fd);
13     if(fd >= 3) {
14         return fs_write(fd, buf, len);
15     }

```

```

16 Log("fd <= 0");
17 return -1;
18 }

```

重新运行，得到正确结果 Hit Good Trap。

```

[src/monitor/monitor.c,30,welcome] Build time: 23:59:47, May 11 2024
For help, type "help"
(nemu) c
[src/main.c,19,main] 'Hello World!' from Nanos-lite
[src/main.c,20,main] Build time: 04:57:29, May 12 2024
[src/randisk.c,26,init_randisk] randisk info: start = 0x101c00, end = 0x37066a,
size = 2550378 bytes
[src/main.c,27,main] Initializing interrupt/exception handler...
[src/cpu/exec/system.c,19,exec_ldtr] ldtr.limit=0x7fff
[src/cpu/exec/system.c,20,exec_ldtr] ldtr.base=0x3708c0
[src/fs.c,68,fs_open] success open:31:/bin/text
[src/loader.c,17,loader] filename=/bin/text,fd=31
[src/fs.c,68,fs_open] success open:6:/share/texts/num
PASS!!!
nemu: HIT GOOD TRAP at eip = 0x00100032

```

4 阶段三

完成了阶段二后，就已经成功实现了文件系统，用户程序已经可以读写普通文件了，但仍需要对设备进行管理。而设备本质上也是字节序列，因此一切设备都可以当成文件看待。可以使用文件的接口来操作计算机上的一切，而不必对它们进行详细的区分。

Nanos-lite 也尝试将 IOE 抽象成文件。首先当然是来看输出设备。串口已经被抽象成 stdout 和 stderr 了。至于 VGA，程序为了更新屏幕，只需要将像素信息写入 VGA 的显存即可。于是，Nanos-lite 需要做的，便是把显存抽象成文件。

4.1 把 VGA 显存抽象成文件

文件/dev/fb 是显存的文件抽象，支持写操作和 lseek，以使用户把像素更新到屏幕的指定位置；文件/proc/dispinfo 存储屏幕大小信息，需要支持读操作。

第一步需要在 init_fs() 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。

因此，首先需要在 ioe.c 中添加返回屏幕大小信息的接口 getScreen()。

```

1 void getScreen(int *width, int *height) {
2     *width = __screen.width;
3     *height = __screen.height;
4 }

```

定义好函数后，就可以在 init_fs() 中使用该函数。VGA 一个像素占 32bit，因此 VGA 的每次操作均以 unit32_t 作为基本单位。

```

1 extern void getScreen(int *p_width, int *p_height);
2 int width = 0;
3 int height = 0;
4 getScreen(&width, &height);
5 file_table[FD_FB].size = width * height * sizeof(u_int32_t);
6 Log("set FD_FB size = %d", file_table[FD_FB].size);

```

VGA 需要在屏幕上绘制像素点，因此需要在 nanos-lite/src/device.c 中实现 fb_write() 函数，用于把 buf 中的 len 字节写到屏幕上 offset 处。

```

1 void fb_write(const void *buf, off_t offset, size_t len) {
2     assert(offset % 4 == 0 && len % 4 == 0);
3     int index, screen_x1, screen_y1, screen_y2;
4     int width=0,height=0;
5     getScreen(&width, &height);
6     index=offset/4;
7     screen_y1=index/width;
8     screen_x1=index%width;
9
10    index=(offset+len)/4;
11    screen_y2=index/width;
12    assert(screen_y2>=screen_y1);
13    if(screen_y2==screen_y1)
14    {
15        _draw_rect(buf, screen_x1, screen_y1, len/4, 1);
16        return ;
17    }
18    int tempw=width-screen_x1;
19    if(screen_y2-screen_y1==1)
20    {
21        _draw_rect(buf, screen_x1, screen_y1, tempw, 1);
22        _draw_rect(buf+tempw*4, 0, screen_y2, len/4-tempw, 1);
23        return ;
24    }
25    _draw_rect(buf, screen_x1, screen_y1, tempw, 1);
26    int tempy = screen_y2 - screen_y1 - 1;
27    _draw_rect(buf + tempw * 4, 0, screen_y1 + 1, width, tempy);
28    _draw_rect(buf+tempw*4+tempy*width*4, 0, screen_y2, len / 4 - tempw - tempy * width,
        1);
29 }

```

在 nanos-lite/src/device.c 中将长宽的信息写入 dispinfo, 为以后读取该格式做准备。

```

1 void init_device() {
2     _ioe_init();
3     int width = 0, height = 0;
4     getScreen(&width, &height);
5     sprintf(dispinfo, "WIDTH:%d\nHEIGHT:%d\n", width, height);
6 }

```

之后, 遍可以对 dispinfo 进行读取, 需要完善 dispinfo_read() 函数。

```

1 void dispinfo_read(void *buf, off_t offset, size_t len) {
2     strncpy(buf, dispinfo + offset, len);
3 }

```

在文件系统中添加对/dev/fb 和/proc/dispinfo 这两个特殊文件的支持, 即修改 fs_read() 函数。

```

1 if(fd == FD_DISPINFO){
2     dispinfo_read(buf, get_open_offset(fd), n);

```

```

3 }
4 else {
5     ramdisk_read(buf, disk_offset(fd) + get_open_offset(fd), n);
6 }

```

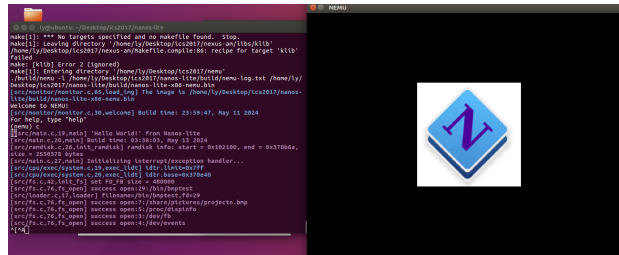
修改 main.c 中的 loader() 函数，使其加载 bmp test。

```

1 uint32_t entry = loader(NULL, "/bin/bmptest");
2 ((void (*)(void))entry)();

```

运行 bmptest，得到以下结果：可以发现 logo 的图片已输出到窗口中。



4.2 把输入设备抽象成文件

为运行一个大型游戏，除了需要 VGA 将画面信息显示在屏幕上，还需要用键盘控制人物移动、用时钟表示时间等。因此，我们需要将键盘和时钟两类输入设备封装成事件，即抽象成文件，使其以文本形式表现出来。

首先，需要实现的是 events_read() 函数，使用 read_key() 与 __uptime() 读取键盘事件和时钟事件，优先读取键盘事件，设定每次只读取一个事件。相关代码如下所示。

```

1 size_t events_read(void *buf, size_t len) {
2     char buffer[40];
3     int key = __read_key();
4     int down = 0;
5     if(key & 0x8000) {
6         key ^= 0x8000;
7         down = 1;
8     }
9     if(key != _KEY_NONE) {
10        sprintf(buffer, "%s %s\n", down ? "kd": "ku", keyname[key]);
11    }
12    else {
13        sprintf(buffer, "t %d\n", __uptime());
14    }
15    if(strlen(buffer) <= len) {
16        strncpy((char*)buf, buffer, strlen(buffer));
17        return strlen(buffer);
18    }
19    Log("strlen(event)>len, return 0");
20    return 0;
21 }

```

然后，需要在系统中添加对/dev/events 的支持。具体操作即为修改 fs_read() 函数，使其识别出 fd 为 FD_EVENTS 时，调用 events_read() 函数。

```
1  if(fd == FD_EVENTS) {
2      return events_read(buf, len);
3  }
```

最后，修改 main.c 中的 loader() 函数，使其加载/bin/events 文件。

```
1  uint32_t entry = loader(NULL, "/bin/events");
2  ((void (*)(void))entry)();
```

运行后得到如下结果。

```
[src/fs.c,42,init_fs] set FD_FB size = 480000
[src/fs.c,76,fs_open] success open:21:/bin/events
[src/loader.c,17,loader] filename=/bin/events,fd=21
[src/fs.c,76,fs_open] success open:4:/dev/events
h
xzvdsr
receive event: t 28513
435
ds12
receive event: t 56782
receive event: t 84469
receive event: t 112340
receive event: t 141723
receive event: t 157867
receive event: t 171559
receive event: t 185274
receive event: t 198815
```

4.3 运行仙剑奇侠传

首先，需要在 data-mov.c 中补充符号扩展指令 cwtl。

```
1  make_EHelper(cwtl) {
2      if (decoding.is_operand_size_16) {
3          rtl_lr_b(&t0, R_AX);
4          rtl_sext(&t0, &t0, 1);
5          rtl_sr_w(R_AX, &t0);
6      }
7      else {
8          rtl_lr_w(&t0, R_AX);
9          rtl_sext(&t0, &t0, 2);
10         rtl_sr_l(R_EAX, &t0);
11     }
12     print_asm(decoding.is_operand_size_16 ? "cbtw" : "cwtl");
13 }
```

打开 main.c，修改 loader() 函数，使其加载仙剑奇侠传所在的 pal 文件夹。

```
1  uint32_t entry = loader(NULL, "/bin/pal");
2  ((void (*)(void))entry)();
```

运行仙剑奇侠传。



5 必答题

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

【问题】: 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。

【回答】: 在《仙剑奇侠传》中, 游戏存档的读取和屏幕的更新是通过多个模块相互协助完成的。首先, 在 PAL_LoadGame() 函数中, 通过 fread() 函数从文件中读取游戏存档数据。

```

1  static INT
2  PAL_LoadGame(
3      LPCSTR          szFileName
4  )
5
6      fp = fopen(szFileName, "rb");
7      if (fp == NULL)
8      {
9          return -1;
10     }
11
12     //
13     // Read all data from the file and close.
14     //
15     fread(&s, sizeof(SAVEDGAME), 1, fp);
16     fclose(fp);

```

其中, fread() 函数实际上调用了 libc 库中的 read 函数, 最终由 libos 中的 __syscall 函数通过系统调用来实现文件读取操作。读取到的游戏存档数据被加载到 static saved 类型的变量 s 中, 准备用于游戏的恢复和继续。在调用 fwrite() 时, 先把数据放缓冲区, 等缓冲区满足条件后, 再一次性调用系统调用, 把数据拷贝到内核空间, 减少了用户态和内核态的切换次数。

```

1 static void redraw() {
2     for (int i = 0; i < W; i++)
3         for (int j = 0; j < H; j++)
4             fb[i + j * W] = palette[vmem[i + j * W]];
5
6     NDL_DrawRect(fb, 0, 0, W, H);
7     NDL_Render();
8 }

```

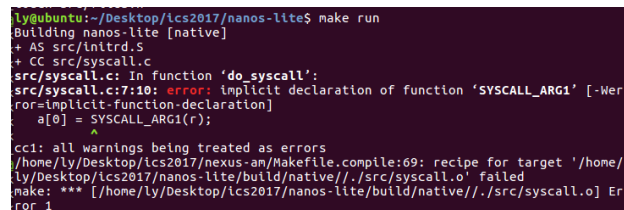
运行仙剑奇侠传时,最顶层的 Navy-app 在屏幕上进行绘制并更新屏幕,会在 hal.c 中调用 NDL_DrawRect 函数设置屏幕所需像素,并调用 NDL_Render 绘制,而 NDL_Render 会执行 lseek、fwrite、fflush 等 C 语言库函数实现屏幕更新。在这个过程中, Nanos-lite 负责管理整个操作系统的运行, AM 负责处理异常事件的传递和处理, NEMU 则负责模拟硬件的指令执行。

之后, lseek、fwrite、fflush 等库函数又会调用 libc 中的对应函数,最终调用 libos 中的 __syscall 函数进行系统调用。

6 BUG 汇总

6.1 搭建的环境无法改成 x86

直接 make run 后,会显示如下的错误,在 src/syscall.c 文件中使用了一个未声明的函数 SYSCALL_ARG1。



```

lygubuntu:~/Desktop/ics2017/nanos-lite$ make run
Building nanos-lite [native]
+ AS src/initrd.S
+ CC src/syscall.c
src/syscall.c: In function 'do_syscall':
src/syscall.c:7:10: error: implicit declaration of function 'SYSCALL_ARG1' [-Werror=implicit-function-declaration]
   a[0] = SYSCALL_ARG1(r);
          ^
cc1: all warnings being treated as errors
/home/ly/Desktop/ics2017/nexus-am/Makefile.compile:69: recipe for target '/home/ly/Desktop/ics2017/nanos-lite/build/native/./src/syscall.o' failed
make: *** [/home/ly/Desktop/ics2017/nanos-lite/build/native/./src/syscall.o] Error 1

```

可以发现,当前的 ARCH 环境为 native,并没有像预想中的那样改为 x86,可能是 makefile 出现了问题。因此,可以将指令修改为“make ARCH=x86-nemu update”, make run 同理。但很快发现, make 时链接出了问题,如下所示。


```

ly@ubuntu:~/Desktop/ics2017/nanos-lite$ make ARCH=x86-nemu run
Building nanos-lite [x86-nemu]
+ CC src/randisk.c
+ CC src/mm.c
+ CC src/main.c
+ CC src/loader.c
+ CC src/proc.c
+ AS src/initrd.S
+ CC src/syscall.c
+ CC src/irq.c
+ CC src/device.c
+ CC src/fs.c
make[1]: Entering directory '/home/ly/Desktop/ics2017/nexus-am'
make[2]: Entering directory '/home/ly/Desktop/ics2017/nexus-am/am'
Building am [x86-nemu]
make[2]: Nothing to be done for 'archive'.
make[2]: Leaving directory '/home/ly/Desktop/ics2017/nexus-am/am'
make[1]: Leaving directory '/home/ly/Desktop/ics2017/nexus-am'
make[1]: Entering directory '/home/ly/Desktop/ics2017/nexus-am/libs/klib'
make[1]: *** No targets specified and no makefile found. Stop.
make[1]: Leaving directory '/home/ly/Desktop/ics2017/nexus-am/libs/klib'
/home/ly/Desktop/ics2017/nexus-am/Makefile.compile:86: recipe for target 'klib' failed
make: [klib] Error 2 (ignored)
/home/ly/Desktop/ics2017/nanos-lite/build/x86-nemu/./src/loader.o: In function 'loader':
/home/ly/Desktop/ics2017/nanos-lite/src/loader.c:21: undefined reference to 'fs_open'
/home/ly/Desktop/ics2017/nanos-lite/src/loader.c:22: undefined reference to 'fs_flush'
/home/ly/Desktop/ics2017/nanos-lite/src/loader.c:22: undefined reference to 'fs_read'
/home/ly/Desktop/ics2017/nanos-lite/src/loader.c:23: undefined reference to 'fs_close'
objdump: '/home/ly/Desktop/ics2017/nanos-lite/build/nanos-lite-x86-nemu': No such file
objcopy: '/home/ly/Desktop/ics2017/nanos-lite/build/nanos-lite-x86-nemu': No such file
/home/ly/Desktop/ics2017/nexus-am/Makefile.app:33: recipe for target 'app' failed
make: *** [app] Error 1
ly@ubuntu:~/Desktop/ics2017/nanos-lite$

```

可能是由于 loader.c 中引入了 fsopen 这些函数的头，将他们删掉后，可正常进入 nemu。

```

./build/nemu -l /home/ly/Desktop/ics2017/nanos-lite/build/nemu-log.txt /home/ly/Desktop/ics2017/nanos-lite/build/nanos-lite-x86-nemu.bin
[src/monitor/monitor.c,65,load_img] The image is /home/ly/Desktop/ics2017/nanos-lite/build/nanos-lite-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 01:04:59, May 8 2024
For help, type "help"
(nemu)

```

需要修改的 make 命令汇总：

- navy-apps: make ISA = x86 install
- nanos-lite: make ARCH = x86-nemu update (/run)

还可以在 navy-apps 的终端中输入以下信息，可以避免在每次 make 时强制加上 ISA：

```

ly@ubuntu:~/Desktop/ics2017/navy-apps$ cat Makefile.check
ISA ?= x86
ifeq ($(NAVY_HOME), )
$(error Must set NAVY_HOME environment variable)
endif

$(shell mkdir -p $(NAVY_HOME)/fsimg/bin/ $(NAVY_HOME)/fsimg/dev/)

```

6.2 BAD TRAP 时没有输出 panic 信息

做到阶段一倒数第二步时，发现触发了如下 BAD TRAP，但并没有输出 panic 信息。

```

[src/monitor/monitor.c,30,welcome] Build time: 00:15:30, May 9 2024
For help, type "help"
(nemu) c
[src/cpu/exec/system.c,19,exec_ldtr] ldtr.limit=0x7fff
[src/cpu/exec/system.c,20,exec_ldtr] ldtr.base=0x1054c0
[src/cpu/intr.c,17,raise_intr] 1071296 1070272 2047

[src/cpu/intr.c,24,raise_intr] target_addr=0x100acc
nemu: HIT BAD TRAP at eip = 0x0010001e

```

这是因为 Nanos-lite 中的 Log 宏并非 NEMU 中的 Log 宏，因此，不能注释 common.h 中的 HAS_IOE 宏和 trm.c 中的 HAS_SERIAL 宏，否则会导致 Nanos-lite 输出失败。去掉注释后，可成功输出 panic 信息并进行下一步。