



南开大学
Nankai University

南 开 大 学

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

蒋薇

年级：2021 级

专业：计算机科学与技术

2024 年 5 月 14 日

目录

一、 概述	1
(一) 实验目的	1
(二) 实验内容	1
二、 Stage1 加载操作系统的第一个用户程序	1
(一) 实现 loader	1
(二) 准备 IDT	2
(三) 触发异常	4
(四) 保存现场	5
(五) 重新组织 TrapFrame 结构体	6
(六) 实现系统调用	7
(七) 恢复现场	8
三、 Stage2 在操作系统上运行 Hello World	8
(一) 在 Nanos-lite 上 Hello world	8
(二) 堆区管理	9
(三) 简易文件系统	10
四、 Stage3 一切皆文件	16
(一) 把 VGA 显存抽象成文件	16
(二) 把设备输入抽象成文件	19
(三) 在 NEMU 中运行仙剑奇侠传	20
五、 必答题	21

一、 概述

(一) 实验目的

我们需要运行仙剑奇侠传, 仙剑奇侠传需要使用文件来管理游戏相关的数据, 然文件的概念并不属于 AM, 因此, 为了运行规模更大的程序, 我们需要一个支持文件操作的操作系统, 以支撑仙剑奇侠传的运行。

(二) 实验内容

先实现一个足够简单的操作系统, 来支撑 dummy 程序的运行。
成功运行 dummy 程序后, 在操作系统上运行 Hello World。

二、 Stage1 加载操作系统的第一个用户程序

Navy-apps, 专门用于编译出操作系统的用户程序。

(一) 实现 loader

加载的过程就是把可执行文件中的代码和数据放置在正确的内存位置, 然后跳转到程序入口, 程序就开始执行了。

可执行文件位于 ramdisk 偏移为 0 处, 访问它就可以得到用户程序的第一个字节。

”正确的内存位置”, 也就是我们上文提到的约定好的 0x4000000。

loader 只需要做一件事情: 将 ramdisk 中从 0 开始的所有内容放置在 0x4000000, 并把这个地址作为程序的入口返回。

加载程序其实就是把比特串放置在正确的位置, 蕴含着”存储程序”的划时代思想。

要在 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。

ramdisk_read 和获取 ramdisk 长度的 get_ramdisk_size 函数都在 nanos-lite/src/ramdisk.c 定义。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。

实现 loader

```
1  uintptr_t loader(_Protect *as, const char *filename) {  
2  size_t len = get_ramdisk_size();  
3  ramdisk_read(DEFAULT_ENTRY, 0, len);  
4  return (uintptr_t)DEFAULT_ENTRY;  
5  }
```

声明外部函数 void ramdisk_read(void *, uint32_t, uint32_t);

size_t get_ramdisk_size();

void ramdisk_read(void *buf, off_t offset, size_t len) : 从 ramdisk 总的 offset 偏移出的 len 字节读入到 buf 中。

size_t get_ramdisk_size() : 返回 ramdisk 的大小, 单位为字节

dummy 程序执行了一条未实现的 int 指令。

等级森严的制度

这些保护相关的概念和检查过程都是通过硬件实现的, 只要软件运行在硬件上面, 都无法逃出这一天网。

在硬件中加入一些与特权级检查相关的门电路, 如果发现了非法操作, 就会抛出一个异常, 让 CPU 跳转到一个固定的地方, 并进行后续处理。

操作系统的义务管理系统中的所有资源, 为用户进程提供相应的服务。

用户进程想输出一句话, 也要经过一定的合法手续向操作系统进行申请, 这一合法手续就是系统调用。

为了让操作系统注意到用户进程提交的申请, 系统调用通常都会触发一个异常, 然后陷入操作系统。在 GNU/Linux 中, 系统调用产生的异常通过 `int$0x80` 指令触发。

穿越时空的旅程

异常是指 CPU 在执行过程中检测到的不正常事件, 例如除数为零, 无效指令, 权限不足等。i386 还向软件提供 `int` 指令, 让软件可以手动产生异常, 因此前面提到的系统调用也算是一种特殊的异常。

CPU 检测到异常之后, 述跳转的目标通过门描述符 (Gate Descriptor) 来指示。

为了方便管理门描述符, i386 把内存中的某一段数据专门解释成一个数组 IDT (Interrupt Descriptor Table, 中断描述符表), 数组的一个元素就是一个门描述符。

为了从数组中找到一个门描述符, 我们还需要一个索引。对于 CPU 异常来说, 这个索引由 CPU 内部产生 (例如除零异常为 0 号异常), 或者由 `int` 指令给出 (例如 `int $0x80`)

为了在内存中找到 IDT, i386 使用 IDTR 寄存器来存放 IDT 的首地址和长度。我们需要通过软件代码事先把 IDT 准备好, 然后通过一条特殊的指令 `lidt` 在 IDTR 中设置好 IDT 的首地址和长度, 这一中断处理机制就可以正常工作了。

异常处理结束之后, 返回异常之前的状态。在开始真正的处理异常之前应该先把触发异常之前状态为 S 保存起来, EIP 了, 它指示了 S 正在执行的指令 (或者下一条指令); 然后就是 EFLAGS (各种标志位) 和 CS (代码段寄存器, 里面包含 CPL 的信息)。

在 AM 的模型中, 异常处理的能力被划分到 ASYE 模块中。我们分别从 NEMU 和 AM 两个角度来体会硬件和软件如何相互协助来支持 ASYE 的功能。

(二) 准备 IDT

为保证触发异常时跳转到正确的目标地址, 在 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, 第二件事是注册一个事件处理函数。

IDTR 寄存器的格式可以参考 i386 手册中的第 156 页, 其中 LIMIT 16 位, BASE 32 位。i386 手册第 159 页 Figure 9-5, 提示 CS 寄存器是 16 位, 所以在 `nemu/include/cpu/reg.h` 中增加以下行, 用于存放 IDT 的首地址和长度。

添加 IDTR 寄存器

```
1 struct IDTR
2 {
3     /* data */
4     uint32_t base;
5     uint16_t limit;
6 } idtr;
```

该寄存器会在 `_asye_init()` 函数中初始化设置 `idt` 的首地址和长度, 传入对应的数据。

要实现 `lidt` 指令, 该指令会将操作数信息从 `eax` 寄存器中读出, 将 `idtr` 的首地址和长度写入寄存器中, 然后调用 `_asm_lidt()` 函数来实现 IDT 的设置。

我们要实现 `lidt` 指令, 该指令会将操作数信息从 `eax` 寄存器中读出, 将 `idtr` 的首地址和长度写入寄存器中, 然后调用 `_asm_lidt()` 函数来实现 IDT 的设置。实现该指令之前需实现译码函数 `lidt_a`, 该译码函数在 `nemu/include/cpu/decode.h` 中进行注册, 然后在 `nemu/src/decode/decode.c` 中进行实现

```
make_DHelper(lidt_a) decode_op_a(eip, id_dest, true);
```

若 `OperandSize` 是 16, 则 `limit` 读取 16 位, 表示 IDT 数组长度, `base` 读取 24 位, 表示 IDT 数组的起始地址, 若 `OperandSize` 是 32, 则 `limit` 读取 16 位, `base` 读取 32 位。通过 IDTR 中的地址对 IDT 进行索引的时候, 需要使用 `vaddr_read()`, 在 `nemu/src/cpu/exec/system.c` 填写 `lidt` 函数:

填写 `lidt` 函数

```
1  make_EHelper(lidt) {
2  cpu.idtr.limit = vaddr_read(id_dest->addr, 2); //limit 16位
3  if (decoding.is_operand_size_16)
4  cpu.idtr.base = vaddr_read(id_dest->addr + 2, 3); //base 24位
5  else
6  cpu.idtr.base = vaddr_read(id_dest->addr + 2, 4); //base 32位
7  print_asm_template1(lidt);
8  }
```

修改 `exec.c` 中的 `op_table`

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

了在 `all-instr.h` 中加上声明

cs 寄存器初始化

填写 `lidt` 函数

```
1  //修改 nemu/include/cpu/reg.h :
2  rtlreg_t cs;
3  //对cs进行初始化, 放入 cpu_state 中。同时需要在 nemu/src/monitor/monitor.c
   中进行初始化:
4  static inline void restart() {
5  cpu.eip = ENTRY_START;
6  cpu.cs = 8;
7  unsigned int origin = 2;
8  memcpy(&cpu.eflags, &origin, sizeof(cpu.eflags));
9  #ifdef DIFF_TEST
10 init_qemu_reg();
11 #endif
12 }
```

define HAS_ASYE

定义后, 程序加载的入口函数 `main()` 中能运行 `init_irq()` 函数, 该函数会调用 `_asye_init()` 函数, 主要功能在于初始化 IDT 和注册一个事件处理函数。

实现中断机制

实现 `lidt` 指令和 `int` 指令, 并实现 `raise_intr()` 函数. 实现正确后, 重新在 Nanos-lite 上运行 dummy 程序, 看到在 `vecsys()` (在 `nexum-am/am/arch/x86-nemu/src/trap.S` 中定义) 附近触发了未实现指令.

(三) 触发异常

为了测试是否已经成功准备 IDT, 我们还需要真正触发一次异常, 看是否正确地跳转到目标地址. 具体的, 你需要在 NEMU 中实现 `raise_intr()` 函数 (在 `nemu/src/cpu/intr.c` 中定义) 来模拟上文提到的 i386 中断机制的处理过程.

触发异常后硬件的处理如下:

依次将 `EFLAGS`, `CS`, `EIP` 寄存器的值压入堆栈

从 `IDTR` 中读出 IDT 的首地址

根据异常 (中断) 号在 IDT 中进行索引, 找到一个门描述符

将门描述符中的 `offset` 域组合成目标地址

跳转到目标地址

int 指令

```
1  /* 0xcc */ EMPTY, IDEXW(I, int, 1), EMPTY, EX(iret)
2  make_EHelper(int) {
3      uint8_t NO = id_dest -> val & 0xff;
4      raise_intr(NO, decoding.seq_eip);
5      print_asm("int_%s", id_dest->str);
6      #ifdef DIFF_TEST
7      diff_test_skip_nemu();
8      #endif
9  }
```

raise_intr()

```
1  void raise_intr(uint8_t NO, vaddr_t ret_addr) {
2      //1. 当前状态压栈
3      memcpy(&t1, &cpu.eflags, sizeof(cpu.eflags));
4      rtl_li(&t0, t1);
5      rtl_push(&t0);
6      rtl_push(&cpu.cs);
7      rtl_li(&t0, ret_addr);
8      rtl_push(&t0);
9      //2. 从intr中读取地址
10     vaddr_t gate_addr = cpu.idtr.base + NO * sizeof(GateDesc);
11     assert(gate_addr <= cpu.idtr.base + cpu.idtr.limit);
12     //3. 根据索引读取门描述符
13     uint32_t off_15_0 = vaddr_read(gate_addr, 2);
14     uint32_t off_32_16 = vaddr_read(gate_addr + sizeof(GateDesc) - 2, 2);
15     //4. 计算目标地址
16     uint32_t target_addr = (off_32_16 << 16) + off_15_0;
17     #ifdef DEBUG
18     Log("target_addr=0x%x", target_addr);
19     #endif
```

```

20 //5. 跳转到目标地址
21 decoding.is_jump = 1;
22 decoding.jump_eip = target_addr;
23 }

```

1. 依次将 EFLAGS, CS(代码段寄存器), EIP 寄存器的值压栈 2. 从 intr 中读取 idt 的首地址 3. 根据索引取出 IDT 数组信息 4. 计算, 门描述符中的 offset 域组合成目标地址 5. 跳转到目标地址

(四) 保存现场

成功跳转到入口函数 vecsys() 之后, 我们就要在软件上开始真正的异常处理过程了.

保存通用寄存器内容, i386 提供了 pusha 指令, 用于把通用寄存器的值压入堆栈。

pusha 和 popa

pusha 和 popa

```

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    print_asm("pusha");
12 }
13 make_EHelper(popa) {
14    rtl_pop(&cpu.edi);
15    rtl_pop(&cpu.esi);
16    rtl_pop(&cpu.ebp);
17    rtl_pop(&t0);
18    rtl_pop(&cpu.ebx);
19    rtl_pop(&cpu.edx);
20    rtl_pop(&cpu.ecx);
21    rtl_pop(&cpu.eax);
22    print_asm("popa");
23 }
24 //在 exec.c 内添加声明
25 /* 0x60 */ EX(pusha), EX(popa), EMPTY, EMPTY,

```

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?

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?

(五) 重新组织 TrapFrame 结构体

vecsyzs() 会压入错误码和异常号 `irq`, 然后跳转到 `asm_trap()`。在 `asm_trap()` 中, 代码将会把用户进程的通用寄存器保存到堆栈上。这些寄存器的内容连同之前保存的错误码, `#irq`, 以及硬件保存的 `EFLAGS, CS, EIP`, 形成了 `trap frame`(陷阱帧) 的数据结构。

理解 `trap frame` 形成的过程, `nemu` 中栈从高地址向低地址, `struct_RegSet` 结构中的内容从低地址向高地址延伸, 所以先入栈的后声明, 后入栈的先声明。我们需要重新组织定义在 `nexus-am/am/arch/x86-nemu/include/arch.h` 的 `_ResSet` 结构体, 使成员声明的顺序和 `trap.S` 中的 `trap frame` 一致。

ResSet

```

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 };

```

重新在 `Nanos-lite` 上运行 `dummy` 程序, 你会看到在 `nanos-lite/src/irq.c` 中的 `do_event()` 函数中触发了 `BAD TRAP`: `[src/irq.c,5,do_event]kernel:systempanic: Unhandled event ID = 8`

事件分发

`irq_handle()` 的代码会把异常封装成事件, 然后调用在 `_asye_init()` 中注册的事件处理函数, 将事件交给它来处理。在 `Nanos-lite` 中, 这一事件处理函数是 `nanos-lite/src/irq.c` 中的 `do_event()` 函数。`do_event()` 函数会根据事件类型再次进行分发。我们刚才触发了一个未处理的 8 号事件, 这其实是一个系统调用事件 `_EVENT_SYSCALL`(在 `nexus-am/am/am.h` 中定义)。在识别出系统调用事件后, 需要调用 `do_syscall()`(在 `nanos-lite/src/syscall.c` 中定义) 进行处理。

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

事件分发

```

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     return NULL;
9 }

```

系统调用处理

`do_syscall()` 首先通过宏 `SYSCALL_ARG1()` 从现场 `r` 中获取用户进程之前设置好的系统调用参数, 通过第一个参数 - 系统调用号 - 进行分发。但目前 `Nanos-lite` 没有实现任何系统调用, 因此触发了 `panic`。

我们需要在分发的过程中添加相应的系统调用号, 并编写相应的系统调用处理函数 `sys_xxx()`, 然后调用它。

处理系统调用的最后一件事就是设置系统调用的返回值. 我们约定系统调用的返回值存放在系统调用号所在的寄存器中, 所以我们需要通过 `SYSCALL_ARG1()` 来进行设置

(六) 实现系统调用

系统调用有两层处理, 需要先通过 `switch-case` 识别出这是系统调用事件, 再调用 `do_syscall` 函数识别出是哪个系统调用. `x86_32` 系统调用通过中断 `int 0x80` 来实现, 寄存器 `eax` 中存放系统调用号, 同时系统调用返回值也存放在 `eax` 中。

1. 当系统调用参数小于等于 6 个时, 参数则必须按顺序放到寄存器 `ebx`, `ecx`, `edx`, `esi`, `edi`, `ebp` 中
2. 当系统调用参数大于 6 个时, 全部参数应该依次放在一块连续的内存区域里, 同时在寄存器 `ebx` 中保存指向该内存区域的指针

在 `nexus-am/am/arch/x86-nemu/include/arch.h` 中实现正确的 `SYSCALL_ARGx()` 宏,, 将系统调用的参数依次放入 `%eax`, `%ebx`, `%ecx`, `%edx` 四个寄存器中。

SYSCALL_ARGx()

```

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
5 // 要添加 SYS_none 系统调用
6 _RegSet* do_syscall(_RegSet *r) {
7     uintptr_t a[4];
8     a[0] = SYSCALL_ARG1(r);
9     a[1] = SYSCALL_ARG2(r);
10    a[2] = SYSCALL_ARG3(r);
11    a[3] = SYSCALL_ARG4(r);
12    switch (a[0]) {
13    case SYS_none:
14        SYSCALL_ARG1(r) = sys_none();
15        break;
16    case SYS_exit:
17        sys_exit(a[1]);
18        break;
19    default: panic("Unhandled syscall ID=%d", a[0]);
20    }
21    return NULL;
22 }

```

添加 `SYS_none` 系统调用. 设置系统调用的返回值.

实现 `popa` 和 `iret` 指令

重新运行 `dummy` 程序, 号码为 4 的系统调用, 它是一个 `SYS_exit` 系统调用.

实现 `SYS_exit` 系统调用, 它会接收一个退出状态的参数, 用这个参数调用 `__halt()`

重新运行 `dummy` 程序, GOOD TRAP.

(七) 恢复现场

asm_trap() 将根据之前保存的 trap frame 中的内容, 恢复用户进程的通用寄存器 (注意 trap frame 中的%eax 已经被设置成系统调用的返回值了), 并直接弹出一些不再需要的信息, 最后执行 iret 指令. iret 指令用于从异常处理代码中返回, 它将栈顶的三个元素来依次解释成 EIP, CS, EFLAGS, 并恢复它们. 用户进程可以通过%eax 寄存器获得系统调用的返回值, 进而得知系统调用执行的结果.

恢复现场

```

1 make_EHelper(iret) {
2   rtl_pop(&cpu.eip);
3   rtl_pop(&cpu.cs);
4   rtl_pop(&t0);
5   memcpy(&cpu.eflags, &t0, sizeof(cpu.eflags));
6   decoding.jump_eip = 1;
7   decoding.seq_eip = cpu.eip;
8   print_asm("iret");
9 }

```

在 nanos-lite 目录下 make update 后 make run , 显示 HIT GOOD TRAP.

三、 Stage2 在操作系统上运行 Hello World

(一) 在 Nanos-lite 上 Hello world

实现 SYS_write 系统调用, 在 do_syscall 中识别出系统调用号是 SYS_write

do_syscall

```

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

```

在 navy-apps/libs/libos/src/nanos.c 中编写一个辅助函数 `_write()`，功能为：传入相应参数，通过调用 `sys_calll`，返回 `eax` 寄存器的值。

_write

```
1 int _write(int fd, void *buf, size_t count){
2 return _syscall_(SYS_write, fd, (uintptr_t)buf, count);
3 }
```

在 nanos-lite/src/syscall.c 中实现 `sys_write()`。我们需要检查 `fd` 的值，若为 1 或 2，则使用 `_putc()` 将 `buf` 首地址的 `len` 字节输出到串口，并返回 `len`，`fd` 值不能为 `<=0`。

sys_write

```
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 Log("fd<=0");
11 return -1;
12 }
```

即修改 nanos-lite/Makefile 中 `ramdisk` 的生成规则，把 `ramdisk` 中的唯一的文件换成 `hello` 程序：`OBJCOPY_FILE = $(NAVY_HOME)/test/hello/build/hello-x86`

后发现 `hello world` 是一个一个字符进行输出的，即每输出一个字符进行一次系统调用。这是因为用户程序在第一次调用 `printf()` 的时候会尝试通过 `malloc()` 申请一片缓冲区，来存放格式化的内容。若申请失败，就会逐个字符进行输出。

(二) 堆区管理

要实现 `_sbrk()`，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。所谓 `program break`，就是用户程序的数据段 (`data segment`) 结束的位置。用户程序开始运行的时候，`program break` 会位于 `_end` 所指示的位置，意味着此时堆区的大小为 0。`malloc()` 被第一次调用的时候，会通过 `sbrk(0)` 来查询用户程序当前 `program break` 的位置，之后就可以通过后续的 `sbrk()` 调用来动态调整用户程序 `program break` 的位置了。当前 `program break` 和和其初始值之间的区间就可以作为用户程序的堆区，由 `malloc()/free()` 进行管理。

实现堆区管理

在 Navy-apps 的 Newlib 中，`sbrk()` 最终会调用 `_sbrk()`，它在 navy-apps/libs/libos/src/nanos.c 中定义。为了实现 `_sbrk()`，还需要提供一个系统调用 `SYS_brk`，它接收一个参数 `addr`，用于指示新的 `program break` 的位置。这个系统调用在 `do_syscall()` 中，因为系统允许用户自由使用空闲的内存，因此 `SYS_brk` 总返回 0，代表堆区大小调整成功。

在 `do_syscall` 中添加 `caseSYS_brk : SYSCALL_ARG1(r) = sys_brk(a[1]); break;`
`_sbrk()` 通过记录的方式来对用户程序的 `program break` 位置进行管理。

program break 一开始的位置位于 `_end`
 被调用时, 根据记录的 program break 位置和参数 increment, 计算出新 program break
 通过 `SYS_brk` 系统调用来让操作系统设置新 program break
 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
 若该系统调用失败, `_sbrk()` 会返回-1

_sbrk()

```

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 }

```

后可以一句一句打印 Hello World.

(三) 简易文件系统

Nanos 假定文件数量和大小都是固定的, 然可以把每一个文件分别固定在 ramdisk 中的某一个位置.

约定文件紧挨存放, 为了记录 ramdisk 中各个文件的名字和大小, 我们还需要一张”文件记录表”。

修改 Nanos-lite 的 Makefile:

update : update - ramdisk - fsimsrc/syscall.h@touchsrc/initrd.S

运行 make update 会自动编译 Navy-apps 里面的所有程序, 并把 navy-apps/fsimg/ 目录下的所有内容整合成 ramdisk 镜像, 同时生成这个 ramdisk 镜像的文件记录表 nanos-lite/src/files.h.

文件记录表

文件记录表

```

1 "文件记录表"其实是一个数组, 数组的每个元素都是一个结构体。
2 name : 文件名
3 size : 文件大小
4 disk_offset : 文件在ramdisk中的偏移
5 open_offset : 记录目前文件操作的位置。对文件操作了多少个字节, 偏移量就会前进
   相应的字节数
6 typedef struct {
7     char *name;
8     size_t size;
9     off_t disk_offset;
10    off_t open_offset;
11 } Finfo;

```

文件读写操作

实现 `fs_open()`, `fs_read()` 和 `fs_close()`

`fs_open()` : 打开文件。第一个参数 `filename` 为需要打开的文件名, 此时会遍历查找 `file_table`, 如果找到了则返回文件描述符, 没有找到则 `panic`, 并返回 -1

`fs_read()` : 读取文件, 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。注意偏移量不能越过文件边界。

`fs_close()` : 直接返回 0, 因为不需要 `close`

`fs_fsize()` : `read` 和 `write` 操作的辅助函数, 用于返回文件描述符 `fd` 所描述的文件大小。

`lseek()` : 调整偏移量

文件读写等操作

```

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 }
11
12 ssize_t fs_read(int fd, void *buf, size_t len){
13 assert(fd >= 0 && fd < NR_FILES);
14 if(fd < 3 || fd == FD_FB) {
15 Log("arg_invalid:fd<3");
16 return 0;
17 }
18 if(fd == FD_EVENTS) {
19 return events_read(buf, len);
20 }
21 int n = fs_fsize(fd) - get_open_offset(fd);
22 if(n > len) {
23 n = len;
24 }
25 if(fd == FD_DISPINFO){
26 dispinfo_read(buf, get_open_offset(fd), n);
27 }
28 else {
29 ramdisk_read(buf, disk_offset(fd) + get_open_offset(fd), n);
30 }
31 set_open_offset(fd, get_open_offset(fd) + n);
32 return n;
33 }
34
35 int fs_close(int fd) {
36 assert(fd >= 0 && fd < NR_FILES);
37 return 0;

```

```

38 }
39
40 size_t fs_filesz(int fd) {
41     assert(fd >= 0 && fd < NR_FILES);
42     return file_table[fd].size;
43 }

```

在 nanos-lite/include/fs.h 中注册这些函数, 且需要在 nanos-lite/src/loader.c 中加入 fs.h 头文件

```

                                init_fs()
1 void init_fs() {
2     // TODO: initialize the size of /dev/fb
3     extern void getScreen(int *p_width, int *p_height);
4     int width = 0;
5     int height = 0;
6     getScreen(&width, &height);
7     file_table[FD_FB].size = width * height * sizeof(u_int32_t);
8     Log("set FD_FB size=%d", file_table[FD_FB].size);
9 }
10 off_t disk_offset(int fd){
11     assert(fd >= 0 && fd < NR_FILES);
12     return file_table[fd].disk_offset;
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 void set_open_offset(int fd, off_t n){
19     assert(fd >= 0 && fd < NR_FILES);
20     assert(n >= 0);
21     if(n > file_table[fd].size) {
22         n = file_table[fd].size;
23     }
24     file_table[fd].open_offset = n;
25 }
26 extern void ramdisk_read(void *buf, off_t offset, size_t len);
27 extern void ramdisk_write(const void *buf, off_t offset, size_t len);

```

让 loader 使用文件

修改 nanos-lite/src/loader.c 中的 loader 函数

```

                                loader()
1 #include "fs.h"
2 uintptr_t loader(__Protect *as, const char *filename) {
3     // TODO();
4     int fd = fs_open(filename, 0, 0);
5     Log("filename=%s, fd=%d", filename, fd);
6     fs_read(fd, DEFAULT_ENTRY, fs_filesz(fd));

```

```

7 fs_close(fd);
8 return (uintptr_t)DEFAULT_ENTRY;
9 }

```

完善 nanos.c 的各个函数和系统调用

系统调用

```

1 void _exit(int status) {
2     __syscall__(SYS_exit, status, 0, 0);
3 }
4 int _open(const char *path, int flags, mode_t mode) {
5     // _exit(SYS_open);
6     return __syscall__(SYS_open, (uintptr_t)path, flags, mode);
7 }
8 int _write(int fd, void *buf, size_t count){
9     return __syscall__(SYS_write, fd, (uintptr_t)buf, count);
10 }
11 void *_sbrk(intptr_t increment){
12     extern int end;
13     static uintptr_t probreak = (uintptr_t)&end;
14     uintptr_t probreak_new = probreak + increment;
15     int r = __syscall__(SYS_brk, probreak_new, 0, 0);
16     if(r == 0) {
17         uintptr_t temp = probreak;
18         probreak = probreak_new;
19         return (void*)temp;
20     }
21     return (void *)-1;
22 }
23 int _read(int fd, void *buf, size_t count) {
24     // _exit(SYS_read);
25     return __syscall__(SYS_read, fd, (uintptr_t)buf, count);
26 }
27 int _close(int fd) {
28     // _exit(SYS_close);
29     return __syscall__(SYS_close, fd, 0, 0);
30 }
31 off_t _lseek(int fd, off_t offset, int whence) {
32     // _exit(SYS_lseek);
33     return __syscall__(SYS_lseek, fd, offset, whence);
34 }

```

实现完整的文件系统

实现 fs_write() 和 fs_lseek()

fs_write() 是对代码进行写操作

fs_lseek() 为修改 fd 对应文件的 open_offset

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
以使用 `fs_write` 函数检查 `fd` 的值, 将给定缓冲区的指定长度个字节写入指定文件号的文件中

`fs_write()fs_lseek()`

```

1 extern void fb_write(const void *buf, off_t offset, size_t len);
2 ssize_t fs_write(int fd, void *buf, size_t len){
3     assert(fd >= 0 && fd < NR_FILES);
4     if(fd < 3 || fd == FD_DISPINFO) {
5         Log("arg_invalid:fd<3");
6         return 0;
7     }
8     int n = fs_fliesz(fd) - get_open_offset(fd);
9     if(n > len) {
10        n = len;
11    }
12    ramdisk_write(buf, disk_offset(fd) + get_open_offset(fd), n);
13    set_open_offset(fd, get_open_offset(fd) + n);
14    return n;
15 }
16
17 off_t fs_lseek(int fd, off_t offset, int whence) {
18     switch(whence) {
19     case SEEK_SET:
20         set_open_offset(fd, offset);
21         return get_open_offset(fd);
22     case SEEK_CUR:
23         set_open_offset(fd, get_open_offset(fd) + offset);
24         return get_open_offset(fd);
25     case SEEK_END:
26         set_open_offset(fd, fs_filesz(fd) + offset);
27         return get_open_offset(fd);
28     default:
29         panic("Unhandled whence ID=%d", whence);
30     }
31     return -1;
32 }

```

修改 nanos-lite/src/syscall.c, 实现系统调用

`do_syscall()`

```

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);

```



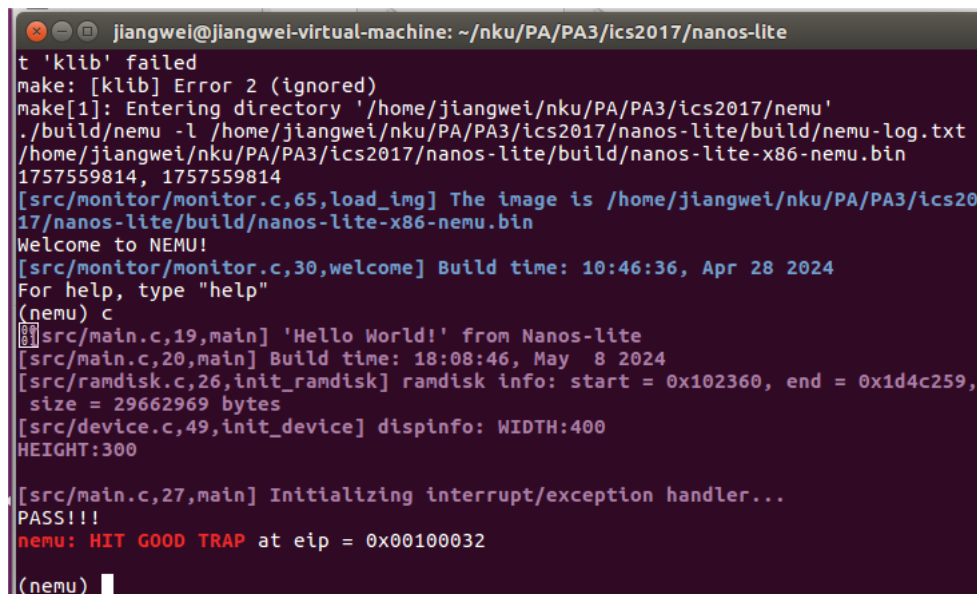
```
7 }
8 return len;
9 }
10 if(fd >= 3) { // 如果fd大于3, 则调用fs_write
11 return fs_write(fd, buf, len);
12 }
13 Log("fd<=0");
14 return -1;
15 }
16 int sys_open(const char *pathname){
17 return fs_open(pathname, 0, 0);
18 }
19 int sys_read(int fd, void *buf, size_t len){
20 return fs_read(fd, buf, len);
21 }
22 int sys_lseek(int fd, off_t offset, int whence) {
23 return fs_lseek(fd, offset, whence);
24 }
25 int sys_brk(int addr) {
26 return 0;
27 }
28 int sys_close(int fd){
29 return fs_close(fd);
30 }
31 __RegSet* do_syscall(__RegSet *r) {
32 uintptr_t a[4];
33 a[0] = SYSCALL_ARG1(r);
34 a[1] = SYSCALL_ARG2(r);
35 a[2] = SYSCALL_ARG3(r);
36 a[3] = SYSCALL_ARG4(r);
37 switch (a[0]) {
38 case SYS_none:
39 SYSCALL_ARG1(r) = sys_none();
40 break;
41 case SYS_exit:
42 sys_exit(a[1]);
43 break;
44 case SYS_write:
45 SYSCALL_ARG1(r) = sys_write(a[1], (void*)a[2], a[3]);
46 break;
47 case SYS_brk:
48 SYSCALL_ARG1(r) = sys_brk(a[1]);
49 break;
50 case SYS_read:
51 SYSCALL_ARG1(r) = sys_read(a[1], (void*)a[2], a[3]);
52 break;
53 case SYS_open:
54 SYSCALL_ARG1(r) = sys_open((char*) a[1]);
```

```

55 break;
56 case SYS_close:
57 SYSCALL_ARG1(r) = sys_close(a[1]);
58 break;
59 case SYS_lseek:
60 SYSCALL_ARG1(r)=sys_lseek(a[1],a[2],a[3]);
61 break;
62 default: panic("Unhandled syscall ID=%d", a[0]);
63 }
64 return NULL;
65 }

```

修改 loader,运行 bin/text,显示 HIT GOOD TRAP:`uint32_t entry = loader(NULL, "/bin/text");`



```

jiangwei@jiangwei-virtual-machine: ~/nku/PA/PA3/ics2017/nanos-lite
t 'klib' failed
make: [klib] Error 2 (ignored)
make[1]: Entering directory '/home/jiangwei/nku/PA/PA3/ics2017/nemu'
./build/nemu -l /home/jiangwei/nku/PA/PA3/ics2017/nanos-lite/build/nemu-log.txt
/home/jiangwei/nku/PA/PA3/ics2017/nanos-lite/build/nanos-lite-x86-nemu.bin
1757559814, 1757559814
[src/monitor/monitor.c,65,load_img] The image is /home/jiangwei/nku/PA/PA3/ics20
17/nanos-lite/build/nanos-lite-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 10:46:36, Apr 28 2024
For help, type "help"
(nemu) c
[src/main.c,19,main] 'Hello World!' from Nanos-lite
[src/main.c,20,main] Build time: 18:08:46, May 8 2024
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x102360, end = 0x1d4c259,
size = 29662969 bytes
[src/device.c,49,init_device] dispinfo: WIDTH:400
HEIGHT:300

[src/main.c,27,main] Initializing interrupt/exception handler...
PASS!!!
nemu: HIT GOOD TRAP at eip = 0x00100032
(nemu)

```

图 1: pass Hello World

四、 Stage3 一切皆文件

(一) 把 VGA 显存抽象成文件

`/dev/fb` 和 `/proc/dispinfo` 都是特殊的文件, 文件记录表中有它们的文件名, 但它们的实体并不在 ramdisk 中。因此, 我们需要在 `fs_read()` 和 `fs_write()` 的实现中对它们进行“重定向”。

Nanos-lite 中在 `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。

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。需先从 `offset` 计算出屏幕上的坐标, 然后调用 IOE 的 `_draw_rect()` 接口。

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。

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。

把 VGA 显存抽象成文件

```

1  void init_fs() {
2  extern void getScreen(int *p_width, int *p_height);
3  int width = 0;
4  int height = 0;
5  getScreen(&width, &height);
6  file_table[FD_FB].size = width * height * sizeof(u_int32_t);
7  Log("set_FD_FB_size=%d", file_table[FD_FB].size);
8  }
9
10 void fb_write(const void *buf, off_t offset, size_t len) {
11 assert(offset % 4 == 0 && len % 4 == 0);
12 int index, screen_x1, screen_y1, screen_y2; int width=0,height=0;
13 getScreen(&width, &height);
14 index=offset/4;
15 screen_y1=index/width;
16 screen_x1=index%width;
17 index=(offset+len)/4;
18 screen_y2=index/width;
19 assert(screen_y2>=screen_y1);
20 if(screen_y2==screen_y1)
21 {
22 _draw_rect(buf, screen_x1, screen_y1, len/4, 1);
23 return ;
24 }
25 int tempw=width-screen_x1;
26 if(screen_y2-screen_y1==1)
27 {
28 _draw_rect(buf, screen_x1, screen_y1, tempw, 1);
29 _draw_rect(buf+tempw * 4, 0, screen_y2, len/4-tempw, 1);
30 return ;
31 }
32 _draw_rect(buf, screen_x1, screen_y1, tempw, 1);
33 int tempy = screen_y2 - screen_y1 - 1;
34 _draw_rect(buf + tempw * 4, 0, screen_y1 + 1, width, tempy);
35 _draw_rect(buf+tempw*4+tempy*width*4, 0, screen_y2, len / 4 - tempw - tempy *
    width,
36 1);
37 }
38 int width=0,height=0;
39 getScreen(&width, &height);
40 index=offset/4;
41 screen_y1=index/width;
42 screen_x1=index%width;
43 index=(offset+len)/4;
44 screen_y2=index/width;
45 assert(screen_y2>=screen_y1);
46 if(screen_y2==screen_y1)

```

```

47 {
48 _draw_rect(buf, screen_x1, screen_y1, len/4, 1);
49 return ;
50 }
51 int tempw=width-screen_x1;
52 if(screen_y2-screen_y1==1)
53 {
54 _draw_rect(buf, screen_x1, screen_y1, tempw, 1);
55 _draw_rect(buf+tempw * 4, 0, screen_y2, len/4-tempw, 1);
56 return ;
57 }
58 _draw_rect(buf, screen_x1, screen_y1, tempw, 1);
59 int tempy = screen_y2 - screen_y1 - 1;
60 _draw_rect(buf + tempw * 4, 0, screen_y1 + 1, width, tempy);
61 _draw_rect(buf+tempw*4+tempy*width*4, 0, screen_y2, len / 4 - tempw - tempy *
    width,
62 1);
63 }
64
65 void init_device() {
66 _ioe_init();
67 int width = 0, height = 0;
68 getScreen(&width, &height);
69 sprintf(dispinfo, "WIDTH:%d\nHEIGHT:%d\n", width, height);
70 }
71
72 void dispinfo_read(void *buf, off_t offset, size_t len) {
73 strncpy(buf, dispinfo + offset, len);
74 }
75
76 ssize_t fs_read(int fd, void *buf, size_t len){
77 assert(fd >= 0 && fd < NR_FILES);
78 if(fd < 3 || fd == FD_FB) {
79 Log("arg_invalid:fd<3");
80 return 0;
81 }
82 if(fd == FD_EVENTS) {
83 return events_read(buf, len);
84 }
85 int n = fs_fliesz(fd) - get_open_offset(fd);
86 if(n > len) {
87 n = len;
88 }
89 if(fd == FD_DISPINFO){
90 dispinfo_read(buf, get_open_offset(fd), n);
91 }
92 else {
93 ramdisk_read(buf, disk_offset(fd) + get_open_offset(fd), n);

```

```

94 }
95 set_open_offset(fd, get_open_offset(fd) + n);
96 return n;
97 }

```

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, 实现正确, 屏幕上显示 ProjectN 的 Logo.

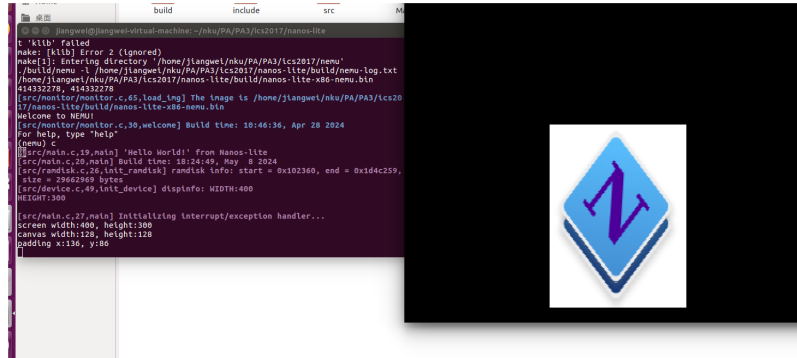


图 2: /bin/bmptest

(二) 把设备输入抽象成文件

要优先处理按键事件, 当不存在按键事件的时候, 才返回时钟事件

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. 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.

设备输入抽象成文件

```

1  size_t events_read(void *buf, size_t len) {
2  char buffer[40];
3  int key = _read_key();
4  int down = false;
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");

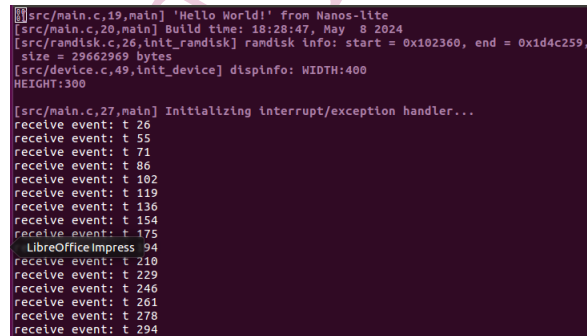
```

```

20 return strlen(buf);
21 }
22 ssize_t fs_read(int fd, void *buf, size_t len){
23     assert(fd >= 0 && fd < NR_FILES);
24     if(fd < 3 || fd == FD_FB) {
25         Log("arg_invalid:fd<3");
26         return 0;
27     }
28     if(fd == FD_EVENTS) {
29         return events_read(buf, len);
30     }
31     int n = fs_fliesz(fd) - get_open_offset(fd);
32     if(n > len) {
33         n = len;
34     }
35     if(fd == FD_DISPINFO){
36         dispinfo_read(buf, get_open_offset(fd), n);
37     }
38     else {
39         ramdisk_read(buf, disk_offset(fd) + get_open_offset(fd), n);
40     }
41     set_open_offset(fd, get_open_offset(fd) + n);
42     return n;
43 }

```

加载/bin/events, 输出时间事件的信息, 敲击按键时会输出按键事件的信息.



```

[src/main.c,19,main] 'Hello World!' from Nanos-lite
[src/main.c,20,main] Build time: 18:28:47, May  8 2024
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x102360, end = 0x1d4c259,
size = 20662969 bytes
[src/device.c,49,init_device] dispinfo: WIDTH:400
HEIGHT:300

[src/main.c,27,main] Initializing interrupt/exception handler...
receive event: t 26
receive event: t 55
receive event: t 71
receive event: t 86
receive event: t 102
receive event: t 119
receive event: t 136
receive event: t 154
receive event: t 175
LibreOfficeImpress 94
receive event: t 210
receive event: t 229
receive event: t 246
receive event: t 261
receive event: t 278
receive event: t 294

```

图 3: /bin/events

(三) 在 NEMU 中运行仙剑奇侠传

充指令 cwtl,cwtl 是一个 ATA 格式的符号扩展指令。

cwtl 指令

```

1     make_EHelper(cwtl) {
2     if (decoding.is_operand_size_16) {
3         rtl_lr_b(&t0, R_AX);
4         rtl_sxt(&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 }

```

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`

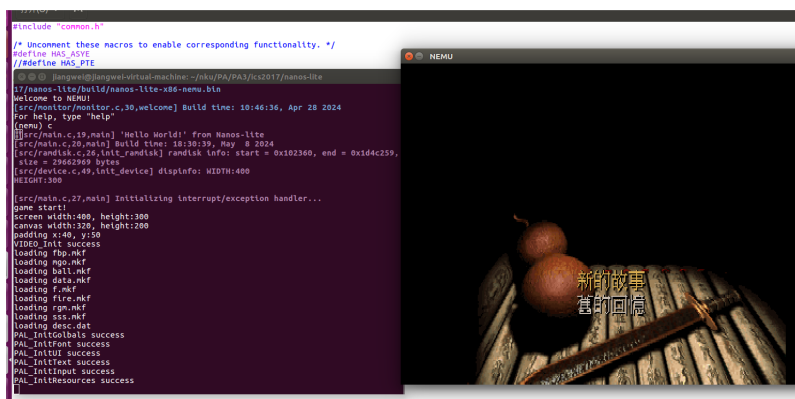


图 4: `/bin/pal`

相关文件打开成功。

五、 必答题

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。

1. 库函数: 一组预先编写好的可重用代码块, 用于执行特定的功能或任务。它们被组织在库文件中, 可以在程序开发过程中被引用和调用。

2. libos: 定义在 `navy-apps` 中, `navy-apps` 用于编译出操作系统的用户程序, 其中 `libos` 是系统调用的用户层封装。它提供了一个抽象层, 允许应用程序以类似于操作系统的方式进行操作和管理资源。

3. Nanos-lite: 它是操作系统 Nanos 的简化版, 运行在 AM 之上, 是 NEMU 的客户端

4. AM: 是计算机的抽象模型, 用于描述计算机系统的基本组成部分和操作方式, 而不考虑具体的硬件实现细节。AM 包括了若干个组件, 如存储器、处理器、IO 等, 还定义了一套基本的指令集和操作方式, 这些指令用于描述计算机的操作, 例如算术运算、逻辑运算、内存访问等。这些指令可以进行组合和序列化, 以实现更复杂的计算任务。

5. NEMU: 它是经过简化的 x86 系统模拟器。它通过程序而非电路来实现对抽象计算机的具体实现。

协作关系为:

Nanos-lite 是 NEMU 的客户端, 运行在 AM 之上; 仙剑奇侠传是 Nanos-lite 的客户端, 运行在 Nanos-lite 之上。

编译后的程序被保存在 ramdisk 文件中

make run 先运行 nemu , 然后在 nemu 上运行 Nanos-lite

Nanos-lite 的 main 函数中使用 loader 加载位于 ramdisk 存储区 (实际存在与内存中) 的 /bin/pal 程序

loader 函数从 ramdisk 文件 (磁盘) 中读取程序到内存区, 进行一些初始化操作后, 便将控制转到仙剑的 main 入口函数

仙剑程序调用库函数和 Nanos-lite 中自定义的库函数完成程序的运行, 包括文件的读写和 UI 的显示等.

仙剑奇侠传的运行离不开库函数, 它需要调用一些库函数的操作, 而库函数也会进行系统调用, 此时支持到支持仙剑的操作系统即 Nanos-lite 提供的 API, Nanos-lite 提供简易运行环境, 而其本身也运行在 AM 之上, 使用 NEMU 模拟出来的 x86 系统。

在 navy-apps/apps/pal/src/global/global.c 的 PAL_LoadGame() 中通过 fread() 读取游戏存档

读取游戏存档

```
1  static INT
2  PAL_LoadGame(
3  LPCSTR szFileName
4  )
5  /*++
6  Purpose:
7  Load a saved game.
8  Parameters:
9  [IN] szFileName - file name of saved game.
10 Return value:
11 0 if success, -1 if failed.
12 ---*/
13 {
14 FILE *fp;
15 PAL_LARGE SAVEDGAME s;
16 UINT32 i;
17 //
18 // Try to open the specified file
19 //
20 fp = fopen(szFileName, "rb");
21 if (fp == NULL)
22 {
23 return -1;
24 }
25 //
26 // Read all data from the file and close.
27 //
28 fread(&s, sizeof(SAVEDGAME), 1, fp);
29 fclose(fp);
30 ...
31 }
```


该函数主要用于打开一个已存档的游戏，传入的参数为需要读取的存档名，然后调用 `fopen` 来打开存档文件。`fread(s, sizeof(SAVEDGAME), 1, fp);` 该语句意味着，将存档中的信息读取到 `static saved` 类型的变量 `s` 中，而在其调用 `fwrite` 的时候也是一样的，先把数据放在放在缓冲区，等到缓冲区满足条件时，一次性调用系统调用，切换到内核态，把数据拷贝到内核空间，这样就能减少 `read` 和 `write` 调用的次数，减少系统开销

在 `navy-apps/apps/pal/src/hal/hal.c` 的 `redraw()` 中通过 `NDL_DrawRect()` 更新屏幕

g 更新屏幕

```

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  NDL_DrawRect(fb, 0, 0, W, H);
6  NDL_Render();
7  }
8  int NDL_DrawRect(uint32_t *pixels, int x, int y, int w, int h) {
9  if (has_nwm) {
10 for (int i = 0; i < h; i++) {
11 printf("\033[X%d;Y%d", x, y + i);
12 for (int j = 0; j < w; j++) {
13 putchar(' ');
14 fwrite(&pixels[i * w + j], 1, 4, stdout);
15 }
16 printf("\n");
17 }
18 } else {
19 for (int i = 0; i < h; i++) {
20 for (int j = 0; j < w; j++) {
21 canvas[(i + y) * canvas_w + (j + x)] = pixels[i * w + j];
22 }
23 }
24 }
25 }

```

在 `redraw` 函数中，`Palette` 是 256 色调色板，`fb` 和 `vmem` 都是 `size` 为 `W*H` 的数组。用 `palette` 给 `fb` 对应的元素赋值，然后将 `fb` 作为第一个参数传入 `NDL_DrawRect` 中。在 `ndl.c` 中可以找到这个函数的具体定义

`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` 接口显示到屏幕上面。

在 `ndl.c` 中，包含了 `stdio.h` 头文件，因此我们可以推断函数的 `printf`，`putchar` 和 `fwrite` 等读写操作都是直接调用 `stdio`，这种调用会像上文所说的 `fread` 一样陷入内核态，然后进行一系列的相关操作。在调用 `fread`，`fwrite` 等函数时标准 `IO` 会陷入 `OS` 内核进行操作，即在 `Nanos-lite` 中进行文件读写操作。而 `libos` 中的 `Nanos.c` 中提供了系统调用接口，即 `sys_call` 函数，根据不同的系统调用号进行不同的中断操作。在这个过程中，内核操作中会调用 `AM` 的 `IOE` 接口进行屏幕信息更新，`NEMU` 则提供硬件支持。