PA3 异常控制流

- 姓名：管昀玫
- 学号：2013750
- 专业：计算机科学与技术

1 概述

1.1 实验目的

- 更深入理解操作系统概念
- 学习系统调用，并实现中断机制
- 了解文件系统的基本内容，实现简易的文件系统
- 最终实现支持文件操作的操作系统，最终成功运行仙剑奇侠传小游戏

1.2 实验内容

- 了解OS的系统调用，实现中断机制
- 进一步完善系统调用，实现简易文件系统
- 输入输出抽象文件，运行仙剑奇侠传

2 阶段一

2.1 加载操作系统的第一个用户程序

`loader` 是一个用于加载程序的模块。程序中包括代码和数据都是存储在可执行文件中，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。

`navy-apps/libs/libc` 中是一个名为 `Newlib` 的项目，它是一个专门为嵌入式系统提供的 C 库，库中的函数对运行时环境的要求极低。用户程序的入口位于 `navy-apps/libs/libc/start.c` 中的 `_start()` 函数，它会调用用户程序的 `main()` 函数，从 `main()` 函数返回后会调用 `exit()` 结束运行。我们要在 Nanos-lite 上运行的第一个用户程序是 `navy-apps/tests/dummy/dummy.c`。

首先修改 `navy-apps/Makefile.check`，让Navy-apps项目上的程序默认编译到 x86 中:

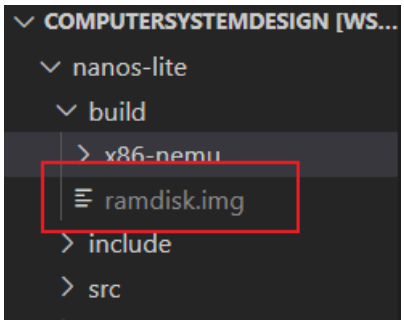
```
ISA ?= x86  //原先为ISA ?= native
ifeq ($(NAVY_HOME), )
  $(error Must set NAVY_HOME environment variable)
endif
```

然后在 `navy-apps/tests/dummy` 下执行 `make`，就可以生成 `dummy` 的可执行文件：

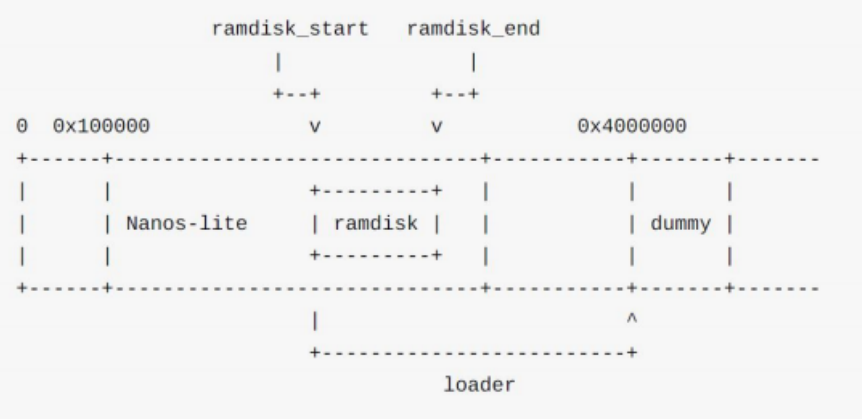
```
+ CC src/time/asctime.c
+ CC src/time/clock.c
+ CC src/time/ctime.c
+ CC src/time/difftime.c
+ CC src/time/gmtime.c
+ CC src/time/localtime.c
+ CC src/time/mktime.c
+ CC src/time/strftime.c
+ CC src/time/time.c
+ AR /mnt/d/LessonProjects/SystemDesign/ComputerSystemDesign/navy-apps/libs/libc/build/libc-x86.a
make[1]: Leaving directory '/mnt/d/LessonProjects/SystemDesign/ComputerSystemDesign/navy-apps/libs/libc'
make -C /mnt/d/LessonProjects/SystemDesign/ComputerSystemDesign/navy-apps/libs/libos
make[1]: Entering directory '/mnt/d/LessonProjects/SystemDesign/ComputerSystemDesign/navy-apps/libs/libos'
+ CC src/nanos.c
+ AR /mnt/d/LessonProjects/SystemDesign/ComputerSystemDesign/navy-apps/libs/libos/build/libos-x86.a
make[1]: Leaving directory '/mnt/d/LessonProjects/SystemDesign/ComputerSystemDesign/navy-apps/libs/libos'
+ CC dummy.c
+ LD /mnt/d/LessonProjects/SystemDesign/ComputerSystemDesign/navy-apps/tests/dummy/build/dummy-x86
myrrolinz@Myrrolinz:/mnt/d/LessonProjects/SystemDesign/ComputerSystemDesign/navy-apps/tests/dummy$
```

为了避免和 Nanos-lite 的内容产生冲突，我们约定目前用户程序需要被链接到内存位置 `0x4000000` 处，Navy-apps已经设置好了相应的选项(`navy-apps/Makefile.compile` 中的 `LDFLAGS` 变量)。然后在 `nanos-lite/` 目录下执行 `make update`，`nanos-lite/Makefile` 中会将其生成 `ramdisk` 镜像文件 `ramdisk.img`，并包含进 Nanos-lite 成为其中的一部分(在 `nanos-lite/src/initrd.S` 中实现)

```
myrrolinz@Myrrolinz:/mnt/d/LessonProjects/SystemDesign/ComputerSystemDesign/nanos-lite$ make update
Building nanos-lite [x86-nemu]
objcopy -S --set-section-flags .bss=alloc,contents -O binary /mnt/d/LessonProjects/SystemDesign/ComputerSystemDesign/navy-apps/tests/dummy/build/dummy-x86 build/ramdisk.img
touch src/files.h
ln -sf /mnt/d/LessonProjects/SystemDesign/ComputerSystemDesign/navy-apps/libs/libos/src/syscall.h src/syscall.h
myrrolinz@Myrrolinz:/mnt/d/LessonProjects/SystemDesign/ComputerSystemDesign/nanos-lite$
```



可以看到此时已生成了 `ramdisk.img`。但是现在的 `ramdisk` 十分简单，且可执行文件位于 `ramdisk` 偏移0处，我们需要做的是将 `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即可。



`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，完成loader函数：

```
uintptr_t loader(_Protect *as, const char *filename) {
  // TODO();
  size_t len = get_ramdisk_size();
  ramdisk_read(DEFAULT_ENTRY, 0, len);
  return (uintptr_t)DEFAULT_ENTRY;
}
```

注：这里调用的函数功能如下：

- `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。

注意，还需要声明外部函数：

```
void ramdisk_read(void *, uint32_t, uint32_t);
size_t get_ramdisk_size();
```

且之后更新都需要在 `nanos-lite/` 目录下手动执行 `make update` 来更新 `ramdisk` 内容，再通过 `make run` 运行最新版的 `Nanos-lite`。

2.2 准备IDT

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 DescriptorTable，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)。最后，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。一旦触发异常，CPU 就会按照设定好的 IDT 跳转到目标地址。

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

2.2.1 添加IDTR寄存器

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。

```
  struct IDTR
  {
    /* data */
    uint32_t base;
    uint16_t limit;
  } 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。

2.2.2 实现lidt指令

该指令描述如下：

```
Operation

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

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翻码函数 `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：

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

之后，需要修改`exec.c`中的op_table

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

最后不能忘了在`all-instr.h`中加上声明

2.2.3 cs寄存器初始化

修改`nemu/include/cpu/reg.h`：

```
  rtlreg_t cs;
```

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：

```
static inline void restart() {
  /* Set the initial instruction pointer. */
  cpu.eip = ENTRY_START;
  cpu.cs = 8;
  unsigned int origin = 2;
  memcpy(&cpu.eflags, &origin, sizeof(cpu.eflags));

#ifdef DIFF_TEST
  init_qemu_reg();
#endif
}
```

在`restart()`中将`cpu.cs`初始化为8。

2.2.4 宏HAS_ASYE

定义宏`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。

```
3   /* Uncomment these macros to enable corresponding functionality. */
4   #define HAS_ASYE
5   //#define HAS_PTE
6
```

2.3 触发异常

我们需要现实现int指令：

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

```
make_EHelper(int) {
  uint8_t NO = id_dest -> val & 0xff;
  raise_intr(NO, decoding.seq_eip);
  print_asm("int %s", id_dest->str);
#ifdef DIFF_TEST
  diff_test_skip_nemu();
#endif
}
```

需要在NEMU 中实现 `raise_intr()` 函数(在 `nemu/src/cpu/intr.c` 中定义)来模拟i386中断机制的处理过程。需要经过5个步骤帮助CPU从5状态从S状态跳转到新地方，并保存当前状态：

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

```
void raise_intr(uint8_t NO, vaddr_t ret_addr) {
  /* TODO: Trigger an interrupt/exception with ``NO''.
   * That is, use ``NO'' to index the IDT.
   */

  //1. 当前状态压栈
  memcpy(&t1, &cpu.eflags, sizeof(cpu.eflags));
  rtl_li(&t0, t1);
  rtl_push(&t0);
  rtl_push(&cpu.cs);
  rtl_li(&t0, ret_addr);
  rtl_push(&t0);

  //2. 从intr中读取地址
  vaddr_t gate_addr = cpu.idtr.base + NO * sizeof(GateDesc);
  assert(gate_addr <= cpu.idtr.base + cpu.idtr.limit);

  //3. 根据索引读取门描述符
  uint32_t off_15_0 = vaddr_read(gate_addr,2);
  uint32_t off_32_16 = vaddr_read(gate_addr+sizeof(GateDesc)-2,2);

  //4. 计算目标地址
  uint32_t target_addr = (off_32_16 << 16) + off_15_0;
#ifdef DEBUG
  Log("target_addr=0x%x",target_addr);
#endif
  //5. 跳转到目标地址
  decoding.is_jmp = 1;
  decoding.jmp_eip = target_addr;
}
```

2.4 保存现场

2.4.1 pusha和popa

在这里，我们需要实现 `pusha` 和 `popa`，以保存上下文。首先查阅i386手册说明，发现pusha的作用是将所有寄存器进行压栈和出栈操作，而popa的作用正好相反。

PUSHA/PUSHAD —— Push all General Registers

Opcode	Instruction	Clocks	Description
60	PUSHA	18	Push AX, CX, DX, BX, original SP, BP, SI, and DI
60	PUSHAD	18	Push EAX, ECX, EDX, EBX, original ESP, EBP, ESI, and EDI

Operation

```
IF OperandSize = 16 (* PUSHA instruction *)
THEN
   Temp ← (SP);
   Push(AX);
   Push(CX);
   Push(DX);
   Push(BX);
   Push(Temp);
   Push(BP);
   Push(SI);
   Push(DI);
ELSE (* OperandSize = 32, PUSHAD instruction *)
   Temp ← (ESP);
   Push(EAX);
   Push(ECX);
   Push(EDX);
   Push(EBX);
   Push(Temp);
   Push(EBP);
   Push(ESI);
   Push(EDI);
FI;
```

Description

PUSHA and PUSHAD save the 16-bit or 32-bit general registers, respectively, on the 80386 stack. PUSHA decrements the stack pointer (SP) by 16 to hold the eight word values. PUSHAD decrements the stack pointer (ESP) by 32 to hold the eight doubleword values. Because the registers are pushed onto the stack in the order in which they were given, they appear in the 16 or 32 new stack bytes in reverse order. The last register pushed is DI or EDI.

POPA/POPAD —— Pop all General Registers

```
Opcode    Instruction    Clocks    Description

61        POPA           24        Pop DI, SI, BP, SP, BX, DX, CX, and AX
61        POPAD          24        Pop EDI, ESI, EBP, ESP, EDX, ECX, and EAX


Operation

IF OperandSize = 16 (* instruction = POPA *)
THEN
   DI ← Pop();
   SI ← Pop();
   BP ← Pop();
   throwaway ← Pop (); (* Skip SP *)
   BX ← Pop();
   DX ← Pop();
   CX ← Pop();
   AX ← Pop();
ELSE (* OperandSize = 32, instruction = POPAD *)
   EDI ← Pop();
   ESI ← Pop();
   EBP ← Pop();
   throwaway ← Pop (); (* Skip ESP *)
   EBX ← Pop();
   EDX ← Pop();
   ECX ← Pop();
   EAX ← Pop();
FI;


Description

POPA pops the eight 16-bit general registers. However, the SP value is
discarded instead of loaded into SP. POPA reverses a previous PUSHA,
restoring the general registers to their values before PUSHA was
executed. The first register popped is DI.

POPAD pops the eight 32-bit general registers. The ESP value is
discarded instead of loaded into ESP. POPAD reverses the previous
PUSHAD, restoring the general registers to their values before PUSHAD
was executed. The first register popped is EDI.
```

在`nemu/src/cpu/exec/data-mov.c`中分别实现`pusha`和`popa`：

```
make_EHelper(pusha) {
  // TODO();
  t0 = cpu.esp;
  rtl_push(&cpu.eax);
  rtl_push(&cpu.ecx);
  rtl_push(&cpu.edx);
  rtl_push(&cpu.ebx);
  rtl_push(&t0);
  rtl_push(&cpu.ebp);
  rtl_push(&cpu.esi);
  rtl_push(&cpu.edi);

  print_asm("pusha");
}
```

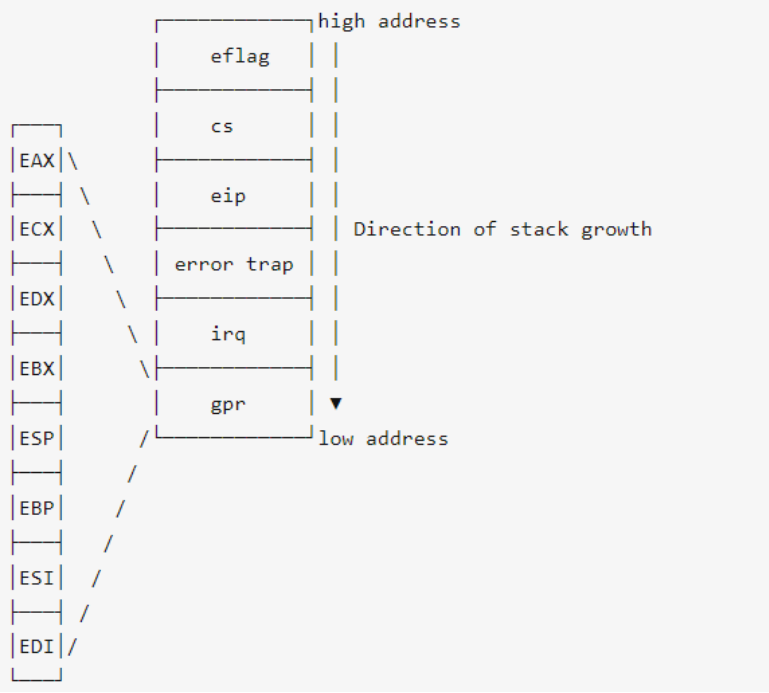
```
make_EHelper(popa) {
  // TODO();
  rtl_pop(&cpu.edi);
  rtl_pop(&cpu.esi);
  rtl_pop(&cpu.ebp);
  rtl_pop(&t0);
  rtl_pop(&cpu.ebx);
  rtl_pop(&cpu.edx);
  rtl_pop(&cpu.ecx);
  rtl_pop(&cpu.eax);
  print_asm("popa");
}
```

之后在`exec.c`内添加声明：

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

2.4.2 重新组织TrapFrame结构体

触发异常后硬件处理第一步是依次将EFLAGS，CS(代码段寄存器)，EIP寄存器的值压栈，程序运行后会触发vecsys函数，`vecsys()`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。由此形成了 trap frame（陷阱帧）的数据结构。trap frame 的结构如下所示：



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。因此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，才不至于引发问题：

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

2.5 事件分发

通过`irq_handle()`识别出异常，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。在`do_event()`中识别出系统调用事件`_EVENT_SYSCALL`，需要调用`do_syscall()`(在`nanos-lite/src/syscall.c`中定义)进行处理。

```
extern _RegSet* do_syscall(_RegSet *r);
static _RegSet* do_event(_Event e, _RegSet* r) {
  switch (e.event) {
    case _EVENT_SYSCALL:
      return do_syscall(r);
    default: panic("Unhandled event ID = %d", e.event);
  }

  return NULL;
}
```

2.6 系统调用处理

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

`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。我们只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()`，然后调用它即可。处理系统调用的最后一件事就是设置系统调用的返回值。系统调用的返回值存放在系统调用号所在的寄存器中，所以只需要通过`SYSCALL_ARG1()`来进行设置就可以。

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

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

然后需要添加 `SYS_none` 系统调用。此系统调用不做任何事，仅直接返回。

```
_RegSet* do_syscall(_RegSet *r) {
  uintptr_t a[4];
  a[0] = SYSCALL_ARG1(r);
  a[1] = SYSCALL_ARG2(r);
  a[2] = SYSCALL_ARG3(r);
  a[3] = SYSCALL_ARG4(r);

  switch (a[0]) {
    case SYS_none:
      SYSCALL_ARG1(r) = sys_none();
      break;
    case SYS_exit:
      sys_exit(a[1]);
      break;
    default: panic("Unhandled syscall ID = %d", a[0]);
  }
  return NULL;
}
```

2.7 恢复现场

系统调用处理结束后，代码将会一路返回到 `trap.S` 的 `asm_trap()` 中。`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，同时修改跳转eip信息。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。

```
make_EHelper(iret) {
  // TODO();
  rtl_pop(&cpu.eip);
  rtl_pop(&cpu.cs);
  rtl_pop(&t0);
  memcpy(&cpu.eflags, &t0, sizeof(cpu.eflags));
  decoding.jmp_eip = 1;
  decoding.seq_eip = cpu.eip;
  print_asm("iret");
}
```

2.8 实验结果

```
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x100fe4, end = 0x1055c0, size = 17884 bytes
[src/main.c,27,main] Initializing interrupt/exception handler...
nemu: HIT GOOD TRAP at eip = 0x00100032
```

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

3 阶段二

3.1 在Nanos-lite上运行Hello world

为了能让`Hello world`正常运行，我们需要先实现`SYS_write()`系统调用。首先需要在`do_syscall`中识别出系统调用号是`SYS_write`：

```
_RegSet* do_syscall(_RegSet *r) {
  uintptr_t a[4];
  a[0] = SYSCALL_ARG1(r);
  a[1] = SYSCALL_ARG2(r);
  a[2] = SYSCALL_ARG3(r);
  a[3] = SYSCALL_ARG4(r);

  switch (a[0]) {
    case SYS_none:
      SYSCALL_ARG1(r) = sys_none();
      break;
    case SYS_exit:
      sys_exit(a[1]);
      break;
    case SYS_write:
      SYSCALL_ARG1(r) = sys_write(a[1], (void*)a[2], a[3]);
      break;
    default: panic("Unhandled syscall ID = %d", a[0]);
  }

  return NULL;
}
```

之后在`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。

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

最后，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.

```
int sys_write(int fd, void *buf, size_t len) {
    if(fd == 1 || fd == 2){
        char c;
        for(int i = 0; i < len; i++) {
            memcpy(&c ,buf + i, 1);
            _putc(c);
        }
        return len;
    }
  Log("fd <= 0");
    return -1;
}
```

以上过程完成后需要切换到`navy-apps/tests/hello/`目录下执行 make 编译 hello 程序：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
```

之后在`nanos-lite`下重新执行`make update`和`make run`，得到以下结果：

```
l[src/syscall.c,5,sys_write] sys_write:fd 1  len 1
l[src/syscall.c,5,sys_write] sys_write:fd 1  len 1
o[src/syscall.c,5,sys_write] sys_write:fd 1  len 1
 [src/syscall.c,5,sys_write] sys_write:fd 1  len 1
W[src/syscall.c,5,sys_write] sys_write:fd 1  len 1
o[src/syscall.c,5,sys_write] sys_write:fd 1  len 1
r[src/syscall.c,5,sys_write] sys_write:fd 1  len 1
l[src/syscall.c,5,sys_write] sys_write:fd 1  len 1
d[src/syscall.c,5,sys_write] sys_write:fd 1  len 1
 [src/syscall.c,5,sys_write] sys_write:fd 1  len 1
f[src/syscall.c,5,sys_write] sys_write:fd 1  len 1
o[src/syscall.c,5,sys_write] sys_write:fd 1  len 1
r[src/syscall.c,5,sys_write] sys_write:fd 1  len 1
 [src/syscall.c,5,sys_write] sys_write:fd 1  len 1
t[src/syscall.c,5,sys_write] sys_write:fd 1  len 1
h[src/syscall.c,5,sys_write] sys_write:fd 1  len 1
e[src/syscall.c,5,sys_write] sys_write:fd 1  len 1
 [src/syscall.c,5,sys_write] sys_write:fd 1  len 1
3[src/syscall.c,5,sys_write] sys_write:fd 1  len 1
```

可以看到此时的`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。事实上，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。

3.2 堆区管理

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。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。

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。

```
_RegSet* do_syscall(_RegSet *r) {
  uintptr_t a[4];
  a[0] = SYSCALL_ARG1(r);
  a[1] = SYSCALL_ARG2(r);
  a[2] = SYSCALL_ARG3(r);
  a[3] = SYSCALL_ARG4(r);
  switch (a[0]) {
    case SYS_none:
      SYSCALL_ARG1(r) = sys_none();
      break;
    case SYS_exit:
      sys_exit(a[1]);
      break;
    case SYS_write:
      SYSCALL_ARG1(r) = sys_write(a[1], (void*)a[2], a[3]);
      break;
    case SYS_brk:
      SYSCALL_ARG1(r) = sys_brk(a[1]);
      break;
    default: panic("Unhandled syscall ID = %d", a[0]);
  }
  return NULL;
}
```

之后我们开始实现`_sbrk()`。`_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

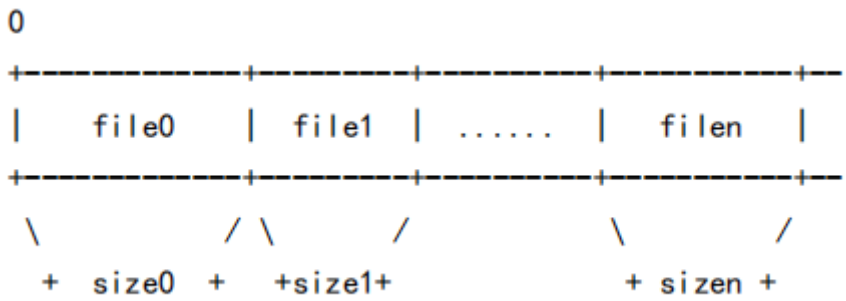
```
void *_sbrk(intptr_t increment){
  extern int end;
  static uintptr_t probreak = (uintptr_t)&end;
  uintptr_t probreak_new = probreak + increment;
  int r = _syscall_(SYS_brk, probreak_new, 0, 0);
  if(r == 0) {
    uintptr_t temp = probreak;
    probreak = probreak_new;
    return (void*)temp;
  }
  return (void *)-1;
}
```

实现之后，打印的指令就能一句一句展示了：

```
Hello World for the 120th time
[src/syscall.c,5,sys_write] sys_write:fd 1  len 31

Hello World for the 121th time
[src/syscall.c,5,sys_write] sys_write:fd 1  len 31

Hello World for the 122th time
[src/syscall.c,5,sys_write] sys_write:fd 1  len 31

Hello World for the 123th time
[src/syscall.c,5,sys_write] sys_write:fd 1  len 31
```

3.3 简易文件系统

文件的本质就是字节序列，另外还由一些额外的属性构成。由于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。这些简化的特性大大降低了文件系统的实现难度。我们约定文件紧挨存放：



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"。首先要对Nanos-lite的`Makefile`作一下修改：

```
update: update-ramdisk-fsimg src/syscall.h
    @touch src/initrd.S
```

然后运行`make update`就会自动编译 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`

3.3.1 文件记录表

"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。

- `name`：文件名
- `size`：文件大小
- `disk_offset`：文件在ramdisk中的偏移
- `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

```
typedef struct {
  char *name;
  size_t size;
  off_t disk_offset;
  off_t open_offset;
} Finfo;
```

3.3.2 文件读写操作

`nanos-lite/src/fs.c` 中定义的 file_table 会包含 `nanos-lite/src/files.h`，其中前面还有 6 个特殊的文件,前三个分别是 stdin，stdout 和 stderr 的占位表项，它们只是为了保证我们的简易文件系统和约定的标准输入输出的文件描述符保持一致。在这一部分，我们需要在文件系统中实现以下文件操作：

- `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_filesz()`：`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

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

```
ssize_t fs_read(int fd, void *buf, size_t len){
  assert(fd >= 0 && fd < NR_FILES);
  if(fd < 3 || fd == FD_FB) {
    Log("arg invalid:fd<3");
    return 0;
  }
```

```
  if(fd == FD_EVENTS) {
    return events_read(buf, len);
  }
  int n = fs_fliesz(fd) - get_open_offset(fd);
  if(n > len) {
    n = len;
  }
  if(fd == FD_DISPINFO){
    dispinfo_read(buf, get_open_offset(fd), n);
  }
  else {
    ramdisk_read(buf, disk_offset(fd) + get_open_offset(fd), n);
  }
  set_open_offset(fd, get_open_offset(fd) + n);
  return n;
}
```

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

```
size_t fs_filesz(int fd) {
  assert(fd >= 0 && fd < NR_FILES);
  return file_table[fd].size;
}
```

注意，我们需要在`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。

```
void init_fs() {
  // TODO: initialize the size of /dev/fb
  extern void getScreen(int *p_width, int *p_height);
  int width = 0;
  int height = 0;
  getScreen(&width, &height);
  file_table[FD_FB].size = width * height * sizeof(u_int32_t);
  Log("set FD_FB size = %d", file_table[FD_FB].size);
}

off_t disk_offset(int fd){
    assert(fd >= 0 && fd < NR_FILES);
    return file_table[fd].disk_offset;
}

off_t get_open_offset(int fd){
    assert(fd >= 0 && fd < NR_FILES);
    return file_table[fd].open_offset;
}
```

```
void set_open_offset(int fd,off_t n){
    assert(fd >= 0 && fd < NR_FILES);
    assert(n >= 0);
    if(n > file_table[fd].size) {
        n = file_table[fd].size;
    }
    file_table[fd].open_offset = n;
}

extern void ramdisk_read(void *buf, off_t offset, size_t len);
extern void ramdisk_write(const void *buf, off_t offset, size_t len);
```

3.3.3 让`loader`使用文件

需要修改`nanos-lite/src/loader.c`中的`loader`函数

```
#include "fs.h"
uintptr_t loader(_Protect *as, const char *filename) {
  // TODO();
  int fd = fs_open(filename, 0, 0);
  Log("filename=%s,fd=%d",filename,fd);
  fs_read(fd, DEFAULT_ENTRY, fs_filesz(fd));
  fs_close(fd);
  return (uintptr_t)DEFAULT_ENTRY;
}
```

我们还需要完善`nanos.c`的各个函数和系统调用

```
void _exit(int status) {
  _syscall_(SYS_exit, status, 0, 0);
}

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

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

void *_sbrk(intptr_t increment){
  extern int end;
  static uintptr_t probreak = (uintptr_t)&end;
  uintptr_t probreak_new = probreak + increment;
  int r = _syscall_(SYS_brk, probreak_new, 0, 0);
  if(r == 0) {
    uintptr_t temp = probreak;
    probreak = probreak_new;
    return (void*)temp;
  }
```

```
  return (void *)-1;
}

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

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

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

3.4 实现完整的文件系统

我们还需要实现`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`。

实现`sys_write`的完整步骤为：

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

可以使用`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`原型：

```
extern void fb_write(const void *buf, off_t offset, size_t len);
ssize_t fs_write(int fd, void *buf, size_t len){
  assert(fd >= 0 && fd < NR_FILES);
  if(fd < 3 || fd == FD_DISPINFO) {
    Log("arg invalid:fd<3");
    return 0;
  }
  int n = fs_fliesz(fd) - get_open_offset(fd);
  if(n > len) {
    n = len;
  }
  ramdisk_write(buf, disk_offset(fd) + get_open_offset(fd), n);
  set_open_offset(fd, get_open_offset(fd) + n);
  return n;
}
```

```
off_t fs_lseek(int fd, off_t offset, int whence) {
  switch(whence) {
    case SEEK_SET:
      set_open_offset(fd, offset);
      return get_open_offset(fd);
    case SEEK_CUR:
      set_open_offset(fd, get_open_offset(fd) + offset);
      return get_open_offset(fd);
    case SEEK_END:
      set_open_offset(fd, fs_filesz(fd) + offset);
      return get_open_offset(fd);
    default:
      panic("Unhandled whence ID = %d", whence);
      return -1;
    }
}
```

之后需要修改`nanos-lite/src/syscall.c`，实现系统调用。

```
int sys_write(int fd, void *buf, size_t len) {
    if(fd == 1 || fd == 2){
        char c;
        for(int i = 0; i < len; i++) {
            memcpy(&c ,buf + i, 1);
            _putc(c);
        }
        return len;
    }
  if(fd >= 3) { // 如果fd大于3，则调用fs_write
    return fs_write(fd, buf, len);
  }
  Log("fd <= 0");
    return -1;
}

int sys_open(const char *pathname){
    return fs_open(pathname, 0, 0);
}

int sys_read(int fd, void *buf,size_t len){
    return fs_read(fd, buf, len);
}

int sys_lseek(int fd, off_t offset, int whence) {
    return fs_lseek(fd, offset, whence);
}

int sys_brk(int addr) {
  return 0;
}
```

```
int sys_close(int fd){
    return fs_close(fd);
}

_RegSet* do_syscall(_RegSet *r) {
  uintptr_t a[4];
  a[0] = SYSCALL_ARG1(r);
  a[1] = SYSCALL_ARG2(r);
  a[2] = SYSCALL_ARG3(r);
  a[3] = SYSCALL_ARG4(r);

  switch (a[0]) {
    case SYS_none:
      SYSCALL_ARG1(r) = sys_none();
      break;
    case SYS_exit:
      sys_exit(a[1]);
      break;
    case SYS_write:
      SYSCALL_ARG1(r) = sys_write(a[1], (void*)a[2], a[3]);
      break;
    case SYS_brk:
      SYSCALL_ARG1(r) = sys_brk(a[1]);
      break;
    case SYS_read:
      SYSCALL_ARG1(r) = sys_read(a[1],(void*)a[2],a[3]);
      break;
    case SYS_open:
      SYSCALL_ARG1(r) = sys_open((char*) a[1]);
      break;
    case SYS_close:
      SYSCALL_ARG1(r) = sys_close(a[1]);
      break;
    case SYS_lseek:
      SYSCALL_ARG1(r)=sys_lseek(a[1],a[2],a[3]);
      break;
    default: panic("Unhandled syscall ID = %d", a[0]);
  }

  return NULL;
}
```

在此之后，修改`loader`，运行`bin/text`，显示HIT GOOD TRAP：

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

```
[src/monitor/monitor.c,65,load_img] The image is /mnt/d/LessonProjects/SystemDesign/ComputerSystemDesign/nanos-lite/build/nanos-lite-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 17:50:08, May 15 2023
For help, type "help"
(nemu) c
[src/main.c,19,main] 'Hello World!' from Nanos-lite
[src/main.c,20,main] Build time: 18:16:19, May 15 2023
[src/ramdisk.c,25,init_ramdisk] ramdisk info: start = 0x1030e0, end = 0x46a065f, size = 72996223 bytes
[src/main.c,27,main] Initializing interrupt/exception handler...
[src/fs.c,32,init_fs] set FD_FB size = 480000
[src/fs.c,67,fs_open] success open:16:/bin/text
[src/loader.c,17,loader] filename=/bin/text,fd=16
[src/fs.c,67,fs_open] success open:83:/share/texts/num
PASS!!!
nemu: HIT GOOD TRAP at eip = 0x0010003a

(nemu)
```

4 阶段三

4.1 把VGA显存抽象成文件

Nanos-lite 和 Navy-apps 约定，把显存抽象成文件 `/dev/fb`(fb 为 frame buffer 之意)，它需要支持写操作和 `lseek`，以便于用户程序把像素更新到屏幕的指定位置上。除此之外，用户程序还需要获得屏幕大小的信息，然后才能决定如何更好地显示像素内容。Nanos-lite 和Navy-apps 约定，屏幕大小的信息通过 `/proc/dispinfo` 文件来获得，它需要支持读操作。

需要注意的是，`/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"。

首先我们在 `init_fs()` 中对文件记录表中 `/dev/fb` 的大小进行初始化，这里我们使用一个自定义函数 `getScreen()` 来获取屏幕大小。在 `init_fs()` 中我们需要设置 `File_table[FD_FB]` 的大小，具体应为显示屏幕大小的图形所需要的字节数，即屏幕长度宽度的乘积再乘以32 位数据的大小：

```
void init_fs() {
  // TODO: initialize the size of /dev/fb
  extern void getScreen(int *p_width, int *p_height);
  int width = 0;
  int height = 0;
  getScreen(&width, &height);
  file_table[FD_FB].size = width * height * sizeof(u_int32_t);
  Log("set FD_FB size = %d", file_table[FD_FB].size);
}
```

`getScreen()` 的函数原型定义在 `nexus-am\am\arch\x86-nemu\src\ioe.c` 中：

```
void getScreen(int *width, int *height) {
  *width = _screen.width;
  *height = _screen.height;
}
```

之后我们实现 `fb_write()`，把 buf 中的 `len` 字节写到屏幕上 offset 处，绘制像素点。需要从offset中计算出屏幕上的坐标，然后调用IOE的 `_draw_rect()` 接口。

```
void fb_write(const void *buf, off_t offset, size_t len) {
  assert(offset % 4 == 0 && len % 4 == 0);
  int index, screen_x1, screen_y1, screen_y2;
```

```
  int width=0,height=0;
  getScreen(&width, &height);
  index=offset/4;
  screen_y1=index/width;
  screen_x1=index%width;
  index=(offset+len)/4;
  screen_y2=index/width;
  assert(screen_y2>=screen_y1);
  if(screen_y2==screen_y1)
  {
    _draw_rect(buf,screen_x1,screen_y1,len/4,1);
    return ;
  }
  int tempw=width-screen_x1;
  if(screen_y2-screen_y1==1)
  {
    _draw_rect(buf,screen_x1,screen_y1,tempw,1);
    _draw_rect(buf+tempw * 4 ,0,screen_y2,len/4-tempw,1);
    return ;
  }
  _draw_rect(buf, screen_x1, screen_y1, tempw, 1);
  int tempy = screen_y2 - screen_y1 - 1;
  _draw_rect(buf + tempw * 4, 0, screen_y1 + 1, width, tempy);
  _draw_rect(buf+tempw*4+tempy*width*4,0,screen_y2, len / 4 - tempw - tempy * width,
1);
}
```

之后实现 `init_device()`。在这个函数中，将 `/proc/dispinfo` 的内容提前写入到字符串 `dispinfo` 中。

实现 `dispinfo_read()`，用于把字符串 `dispinfo` 中 offset 开始的 `len` 字节写到 buf 中。

```
void init_device() {
  _ioe_init();
  int width = 0, height = 0;
  getScreen(&width, &height);
  sprintf(dispinfo, "WIDTH:%d\nHEIGHT:%d\n", width,height);
}

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

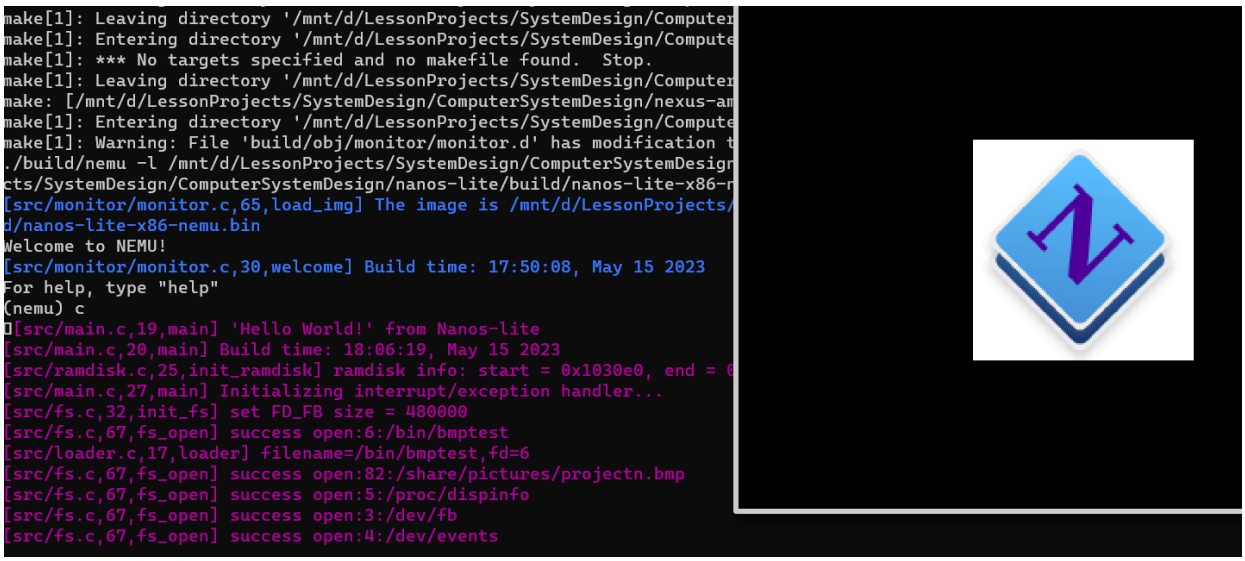
在 `fs_read()` 中对 `FD_DISPINFO` 进行重定向：

```
ssize_t fs_read(int fd, void *buf, size_t len){
  assert(fd >= 0 && fd < NR_FILES);
  if(fd < 3 || fd == FD_FB) {
    Log("arg invalid:fd<3");
    return 0;
  }
  if(fd == FD_EVENTS) {
```

```
    return events_read(buf, len);
  }
  int n = fs_fliesz(fd) - get_open_offset(fd);
  if(n > len) {
    n = len;
  }
  if(fd == FD_DISPINFO){
    dispinfo_read(buf, get_open_offset(fd), n);
  }
  else {
    ramdisk_read(buf, disk_offset(fd) + get_open_offset(fd), n);
  }
  set_open_offset(fd, get_open_offset(fd) + n);
  return n;
}
```

4.2 实验结果

让Nanos-lite加载 `/bin/bmptest`，可以看见 `ProjectN` 的Logo



4.3 把设备抽象成文件

在这部分，需要在Nanos-lite中实现 `events_read()`(在 `nanos-lite/src/device.c` 中定义)，把事件写入到 `buf` 中，最长写入 `len` 字节，然后返回写入的实际长度。其中按键名已经在字符串数组 names 中定义好了。需要借助 IOE 的 API 来获得设备的输入。

```
size_t events_read(void *buf, size_t len) {
  char buffer[40];
  int key = _read_key();
  int down = false;
  if(key & 0x8000) {
      key ^= 0x8000;
      down = 1;
  }
  if(key != _KEY_NONE) {
     sprintf(buffer, "%s %s\n", down ? "kd": "ku", keyname[key]);
```

```
  }
  else {
      sprintf(buffer,"t %d\n", _uptime());
  }
  if(strlen(buffer) <= len) {
    strncpy((char*)buf, buffer,strlen(buffer));
      return strlen(buffer);
  }
  Log("strlen(event)>len, return 0");
  return strlen(buf);
}
```

然后需要在文件系统的`fs_read()`函数中添加对`/dev/events`的支持:

```
ssize_t fs_read(int fd, void *buf, size_t len){
  assert(fd >= 0 && fd < NR_FILES);
  if(fd < 3 || fd == FD_FB) {
    Log("arg invalid:fd<3");
    return 0;
  }
  if(fd == FD_EVENTS) {
    return events_read(buf, len);
  }
  int n = fs_fliesz(fd) - get_open_offset(fd);
  if(n > len) {
    n = len;
  }
  if(fd == FD_DISPINFO){
    dispinfo_read(buf, get_open_offset(fd), n);
  }
  else {
    ramdisk_read(buf, disk_offset(fd) + get_open_offset(fd), n);
  }
  set_open_offset(fd, get_open_offset(fd) + n);
  return n;
}
```

4.4 实验结果

修改loader，加载`/bin/events`，程序输出时间事件的信息。

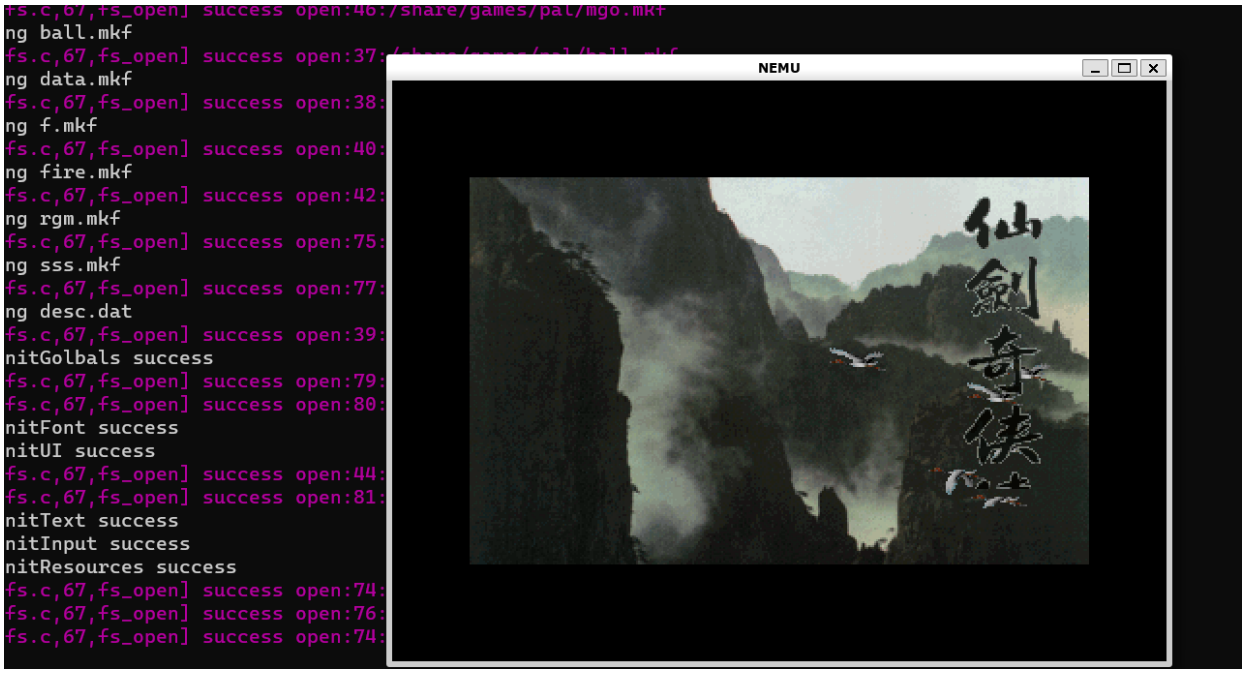
```
[src/monitor/monitor.c,30,welcome] Build time: 17:50:08, May 15 2023
For help, type "help"
(nemu) c
[src/main.c,19,main] 'Hello World!' from Nanos-lite
[src/main.c,20,main] Build time: 18:00:21, May 15 2023
[src/ramdisk.c,25,init_ramdisk] ramdisk info: start = 0x1030e0, end = 0x46a065f, size = 72996223 bytes
[src/main.c,27,main] Initializing interrupt/exception handler...
[src/fs.c,32,init_fs] set FD_FB size = 480000
[src/fs.c,67,fs_open] success open:8:/bin/events
[src/loader.c,17,loader] filename=/bin/events,fd=8
[src/fs.c,67,fs_open] success open:4:/dev/events
receive event: t 192
receive event: t 336
receive event: t 474
receive event: t 623
receive event: t 761
receive event: t 904
receive event: t 1049
receive event: t 1195
receive event: t 1339
ereceive event: t 1482
wereceive event: t 1623
receive event: t 1761
wreceive event: t 1923
dreceive event: t 2063
wreceive event: t 2205
freceive event: t 2347
sreceive event: t 2486
receive event: t 2626
areceive event: t 2764
```

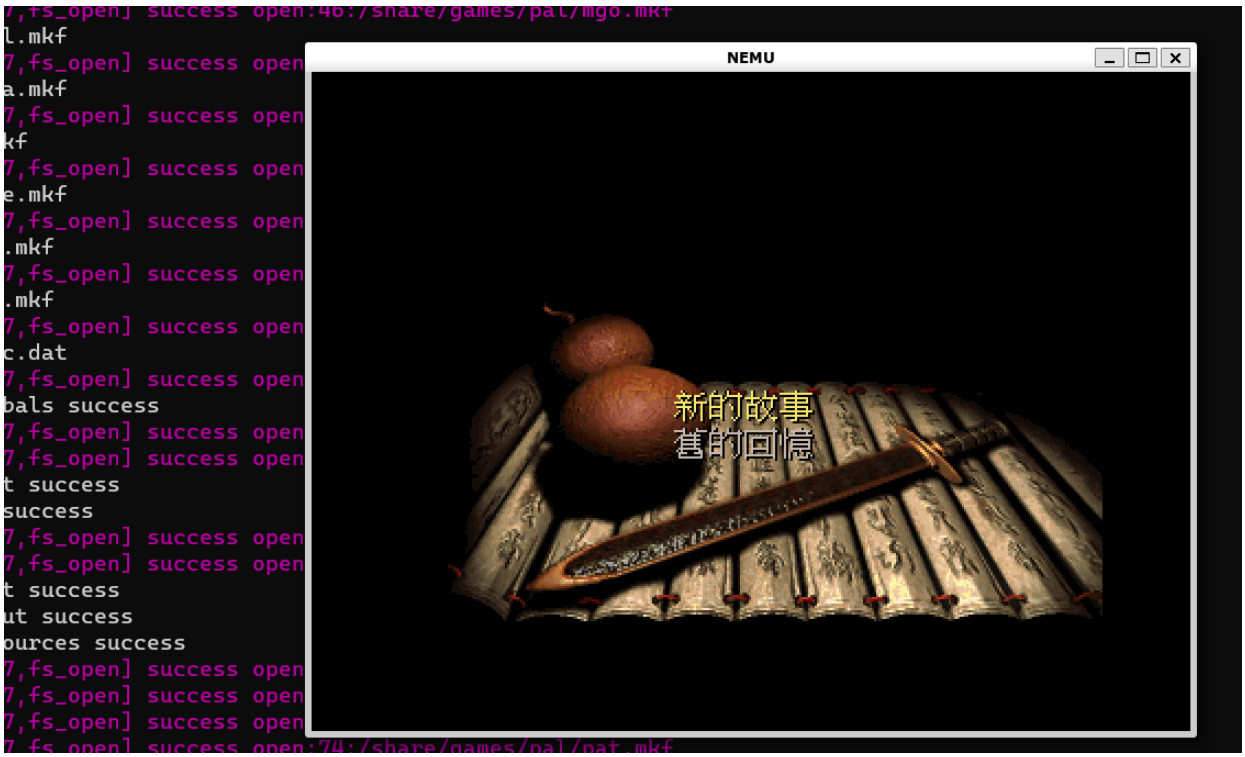
4.5 运行仙剑奇侠传

在运行仙剑奇侠传之前，我们需要在`nemu/src/cpu/exec/data-mov.c`文件中补充指令`cwtl`。`cwtl`是一个AT&A格式的符号扩展指令。

```
make_EHelper(cwtl) {
  if (decoding.is_operand_size_16) {
      rtl_lr_b(&t0, R_AX);
      rtl_sext(&t0, &t0, 1);
      rtl_sr_w(R_AX, &t0);
    }
    else {
      rtl_lr_w(&t0, R_AX);
      rtl_sext(&t0, &t0, 2);
      rtl_sr_l(R_EAX, &t0);
    }
  print_asm(decoding.is_operand_size_16 ? "cbtw" : "cwtl");
}
```

下载`pal.tar.bz2`，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`.


fs.c,67,fs_open] success open:46:/share/games/pal/mgo.mkf
ng ball.mkf
fs.c,67,fs_open] success open:37:
ng data.mkf
fs.c,67,fs_open] success open:38:
ng f.mkf
fs.c,67,fs_open] success open:40:
ng fire.mkf
fs.c,67,fs_open] success open:42:
ng rgm.mkf
fs.c,67,fs_open] success open:75:
ng sss.mkf
fs.c,67,fs_open] success open:77:
ng desc.dat
fs.c,67,fs_open] success open:39:
nitGolbals success
fs.c,67,fs_open] success open:79:
fs.c,67,fs_open] success open:80:
nitFont success
nitUI success
fs.c,67,fs_open] success open:44:
fs.c,67,fs_open] success open:81:
nitText success
nitInput success
nitResources success
fs.c,67,fs_open] success open:74:
fs.c,67,fs_open] success open:76:
fs.c,67,fs_open] success open:74:
NEMU
仙劍奇俠


7,fs_open] success open:46:/share/games/pal/mgo.mkf
l.mkf
7,fs_open] success open
a.mkf
7,fs_open] success open
kf
7,fs_open] success open
e.mkf
7,fs_open] success open
.mkf
7,fs_open] success open
.mkf
7,fs_open] success open
c.dat
7,fs_open] success open
bals success
7,fs_open] success open
7,fs_open] success open
t success
success
7,fs_open] success open
7,fs_open] success open
t success
ut success
ources success
7,fs_open] success open
7,fs_open] success open
7,fs_open] success open
7,fs_open] success open:74:/share/games/pal/pat.mkf
NEMU
新的故事
舊的回憶


```
oading sss.mkf
[src/fs.c,67,fs_open] success open:77:/share/games/pal/sss.mkf
oading desc.dat
[src/fs.c,67,fs_open] success open:39:/share/games/pal/desc.dat
AL_InitGolbals success
[src/fs.c,67,fs_open] success open:79:/share/games/pal/wor16.asc
[src/fs.c,67,fs_open] success open:80:/share/games/pal/wor16.fon
AL_InitFont success
AL_InitUI success
[src/fs.c,67,fs_open] success open:44:/share/games/pal/m.msg
[src/fs.c,67,fs_open] success open:81:/share/games/pal/word.dat
AL_InitText success
AL_InitInput success
AL_InitResources success
[src/fs.c,67,fs_open] success open:74:/share/games/pal/pat.mkf
[src/fs.c,67,fs_open] success open:76:/share/games/pal/rng.mkf
[src/fs.c,67,fs_open] success open:74:/share/games/pal/pat.mkf
[src/fs.c,67,fs_open] success open:74:/share/games/pal/pat.mkf
[src/fs.c,67,fs_open] success open:45:/share/games/pal/map.mkf
[src/fs.c,67,fs_open] success open:43:/share/games/pal/gop.mkf
[src/fs.c,67,fs_open] success open:74:/share/games/pal/pat.mkf
[src/fs.c,67,fs_open] success open:74:/share/games/pal/pat.mkf
[src/fs.c,67,fs_open] success open:74:/share/games/pal/pat.mkf
[src/fs.c,67,fs_open] success open:74:/share/games/pal/pat.mkf
[src/fs.c,67,fs_open] success open:74:/share/games/pal/pat.mkf
```

同时我们可以看到，在运行的过程中，会不断地打开mkf文件。

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

> 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。

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\libs` 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中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。

他们之间的协作关系为：

- `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。

5.1 查看存档

查看 `global.c` 中的代码：

```
static INT
PAL_LoadGame(
   LPCSTR         szFileName
)
/*++
  Purpose:

    Load a saved game.

  Parameters:
```

```
    [IN]  szFileName - file name of saved game.

  Return value:

    0 if success, -1 if failed.

--*/
{
   FILE                     *fp;
   PAL_LARGE SAVEDGAME       s;
   UINT32                    i;

   //
   // Try to open the specified file
   //
   fp = fopen(szFileName, "rb");
   if (fp == NULL)
   {
      return -1;
   }

   //
   // Read all data from the file and close.
   //
   fread(&s, sizeof(SAVEDGAME), 1, fp);
   fclose(fp);
   ...
}
```

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。

5.2 更新屏幕

查看`hal.c`的代码，如下所示：

```
static void redraw() {
  for (int i = 0; i < W; i ++)
    for (int j = 0; j < H; j ++)
      fb[i + j * W] = palette[vmem[i + j * W]];

  NDL_DrawRect(fb, 0, 0, W, H);
  NDL_Render();
}
```

在 `redraw` 函数中，`Palette` 是 256 色调色板，`fb` 和 `vmen` 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，如下所示：

```
int NDL_DrawRect(uint32_t *pixels, int x, int y, int w, int h) {
  if (has_nwm) {
    for (int i = 0; i < h; i ++) {
      printf("\033[X%d;%d", x, y + i);
      for (int j = 0; j < w; j ++) {
        putchar(';');
        fwrite(&pixels[i * w + j], 1, 4, stdout);
      }
      printf("d\n");
    }
  } else {
    for (int i = 0; i < h; i ++) {
      for (int j = 0; j < w; j ++) {
        canvas[(i + y) * canvas_w + (j + x)] = pixels[i * w + j];
      }
    }
  }
}
```

总体来说，`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。

6 问题与解决办法

6.1 实现loader后出现段错误

查阅Stack Overflow之后，发现这个问题是由于越界造成的，修改Nanos-lite的 `makefile` 后得以解决：

```
OBJCOPY_FLAG = -S --remove-section .note.gnu.property --set-section-flags
.bss=alloc,contents -O binary
```

该语句的含义为：

- `-S`：用于剥离目标文件中的所有符号信息。剥离符号信息可以减小目标文件的大小，并且在某些情况下可能有安全性的考虑。
- `--remove-section .note.gnu.property`：这是 `objcopy` 工具的选项之一，用于移除目标文件中名为 `.note.gnu.property` 的节（section）。这个节通常包含了GNU属性相关的信息。

- `--set-section-flags .bss=alloc,contents`：设置目标文件中名为`.bss`的节的属性标志（flags）。这里设置`.bss`节的属性为`alloc,contents`，表示将该节标记为可分配内存并包含内容。
- `-O binary`：用于将目标文件转换为二进制格式。通过这个选项，`objcopy`将目标文件的内容提取出来，以纯二进制的形式保存在文件中。

进行如下修改之后，问题即可解决。

6.2 未在`exec.c`中注册pusha和popa导致错误

排查了数个小时，才发现漏改了一行代码，裂开

6.3 BAD HIT和单字母输出

网上查资料后发现，每次都需要重新make clean，然后make update，最后make run，即可解决。（这个错误太窒息了，困扰了我好几天，之前还有一次BAD HIT也是这个问题，当时都想摆烂了，结果ddl前三天看别人的博客正好看到了这个问题，make clean就出来了，喜极而泣）

6.4 bmptest后出错

```
[src/main.c,20,main] Build time: 17:39:44, May 15 2023
[src/ramdisk.c,25,init_ramdisk] ramdisk info: start = 0x1030e0, end = 0x4681f97, size = 72871607 bytes
[src/main.c,27,main] Initializing interrupt/exception handler...
[src/fs.c,32,init_fs] set FD_FB size = 480000
[src/fs.c,67,fs_open] success open:15:/bin/pal
[src/loader.c,17,loader] filename=/bin/pal,fd=15
game start!
[src/fs.c,67,fs_open] success open:5:/proc/dispinfo
invalid opcode(eip = 0x040210f8): a4 39 d7 75 fb 5e 5f 5d ...

There are two cases which will trigger this unexpected exception:
1. The instruction at eip = 0x040210f8 is not implemented.
2. Something is implemented incorrectly.
Find this eip(0x040210f8) in the disassembling result to distinguish which case it is.

If it is the first case, see
 _ ____   ___    __    __ __                       _
(_)___ \ / _ \  / /   |  \/  |                     | |
 _  __) | (_) |/ /_   | \  / | __ _ _ __ _   _  __ _| |
| ||__ < > _ <| '_ \  | |\/| |/ _` | '_ \ | | |/ _` | |
| |___) | (_) | (_) | | |  | | (_| | | | | |_| | (_| | |
|_|____/ \___/ \___/  |_|  |_|\__,_|_| |_|\__,_|\__,_|_|

for more details.

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

(nemu)
```

这个！！也是个坑！参考这篇博客：https://zhuanlan.zhihu.com/p/522797589

运行`/bin/bmptest`时，遇到提示`a4`这条指令没有实现的情况，在`navy-apps/Makefile.compile`修改`o2`为`o0`即可，这是关闭了代码优化，避免了一些数据未保存。然后重新`make clean`，`make update`，`make run`，即可大功告成！