1 概述

1.1 实验目的

1. 熟悉Nanos-lite，加深对操作系统的理解
2. 充分理解异常处理过程，中断机制及系统调用
3. 充分理解文件系统并为Nanos-lite实现一个简易的文件系统
4. 充分理解“一切皆文件”的思想，并将设备输入与输出抽象成文件
5. 在NEMU上成功运行《仙剑奇侠传》——“新的故事”

1.2 实验内容

在PA2的阶段的实验中，已经补全了NEMU上的指令译码、执行过程，实现了多条指令并为NEMU添加了IOE，将NEMU打造为了一个冯诺依曼计算机系统，并且已经在AM上运行了许多程序，但为了能够在NEMU上运行仙剑奇侠传，还需要操作系统和文件管理系统的支持。框架代码中的Nanos-lite就是这样一个为PA量身打造的操作系统，但它目前的功能还不完善，不足以支持仙剑奇侠传的运行，所以需要在PA3的实验中为Nanos-lite实现加载程序的模块（loader），实现异常控制流（包括中断机制与系统调用），实现简易文件系统（同时修改loader），将设备输入和输出抽象成文件。最后，在NEMU上成功运行《仙剑奇侠传》。

阶段一：正确实现程序加载模块（loader）及异常控制流（包括中断机制与系统调用），添加SYS_none与SYS_exit系统调用

阶段二：正确实现堆区管理及简易文件系统，修改程序加载模块（loader）以加载文件系统中的特定文件，添加SYS_write、SYS_brk、SYS_open、SYS_read、SYS_close、SYS_lseek系统调用，修改SYS_write系统调用以写入文件系统中的特定文件

阶段三：通过已经实现的简易文件系统将VGA显存和设备输入抽象成文件，并成功运行《仙剑奇侠传》——“新的故事”

2 阶段一

框架代码中Nanos-lite是一个为PA量身订造的操作系统，包含了后续PA所需的所有模块，这些模块可以通过`nanos-lite/src/main.c`中宏来控制，目前这些模块都没有打开，Nanos-lite目前的工作包括：

```
int main() {
#ifdef HAS_PTE
  init_mm();
#endif

  Log("'Hello World!' from Nanos-lite"); //通过Log()打印hello信息
  Log("Build time: %s, %s", __TIME__, __DATE__); //通过Log()打印build时间
  init_ramdisk(); //初始化ramdisk（注意Nanos-lite中的磁盘其实是一段内存，所以必须
                  //将需要运行的程序包含在Nanos-lite中才能在Nanos-lite上运行）
  init_device(); //初始化设备（包括IOE）

#ifdef HAS_ASYE
  Log("Initializing interrupt/exception handler...");
  init_irq();
#endif

  uint32_t entry = loader(NULL, NULL); //从ramdisk中加载用户程序
  ((void (*)(void))entry)(); //跳转到用户程序的入口执行

  panic("Should not reach here");
}
```

为了能够在Nanos-lite上运行想让它运行的用户程序，需要实现程序加载模块（loader）来将可执行文件加载到用户程序的入口处，这样Nanos-lite就可以在完成初始化的工作后执行对应的用户程序。

与PA2中直接在NEMU上运行程序不同，PA3中是在Nanos-lite这一操作系统上运行程序，尽管NEMU中并未加入保护机制，但依然需要操作系统来管理系统中的所有资源，为用户进程提供相应的服务，用户程序也需要通过系统调用才能使用特殊的系统资源（例如输出设备）。为了能让用户程序能够进行系统调用，需要在Nanos-lite上实现异常控制流（系统调用就是一个由程序自己产生的异常），包括准备IDTIDT(Interrupt Descriptor Table，中断描述符表)，添加触发异常的指令，实现对现场的保存，事件（被封装的异常）分发，系统调用以及系统调用完成后的现场恢复。在阶段一中只需要运行`navy-apps/tests`目录下的`dummy`程序，只需要实现SYS_none与SYS_exit系统调用即可。

2.1 实现方法及代码分析

2.1.1 程序加载模块（loader）

程序包括代码和数据，都是存储在可执行文件中。程序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，并且然后跳转到程序入口，程序就可以正常运行了。因为Nanos-lite中的磁盘只是一段内存，将程序在`Navy-apps`下完成编译后，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。因为此时并未实现文件系统，所以磁盘中只包含了一个文件（可执行程序），且`nanos-lite/Makefile`中已经把用户程序运行所需要的代码和静态数据通过objcopy工具从ELF文件中抽取出来了，我们也不需要对文件头做特殊的处理，磁盘中所包含的所有内容即是需要加载的内容（意味着文件在磁盘中存储的偏移量为0），框架代码中也已经提供了`ramdisk`相关的函数（在`nanos-lite/src/ramdisk.c`中定义），

```
/* read `len' bytes starting from `offset' of ramdisk into `buf' */
void ramdisk_read(void *buf, off_t offset, size_t len) {
  assert(offset + len <= RAMDISK_SIZE);
  memcpy(buf, &ramdisk_start + offset, len);
}

/* write `len' bytes starting from `buf' into the `offset' of ramdisk */
void ramdisk_write(const void *buf, off_t offset, size_t len) {
  assert(offset + len <= RAMDISK_SIZE);
  memcpy(&ramdisk_start + offset, buf, len);
}

size_t get_ramdisk_size() {
  return RAMDISK_SIZE;
}
```

在实现程序加载模块（loader）时只需要调用这些函数对磁盘进行操作即可。

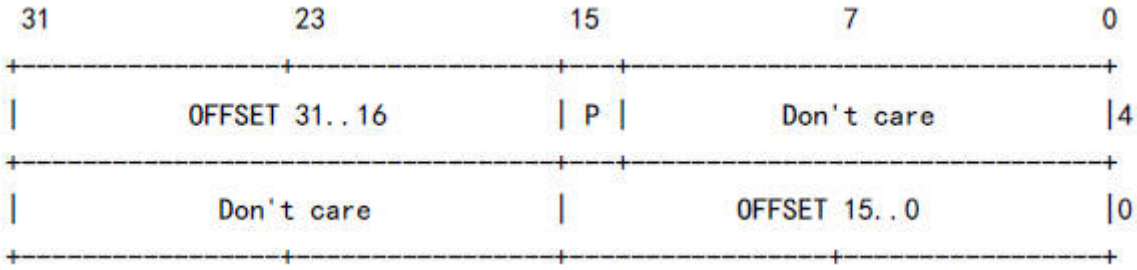
```
#define DEFAULT_ENTRY ((void *)0x4000000)

extern void ramdisk_read(void *buf, off_t offset, size_t len); //调用函数前声明
extern size_t get_ramdisk_size(); //调用函数前声明

uintptr_t loader(_Protect *as, const char *filename) {
  size_t rsize = get_ramdisk_size(); //获取当前磁盘大小（可执行程序大小）
  //将ramdisk中从0开始的rsize大小的字节读入到DEFAULT_ENTRY（用户程序的入口）处
  ramdisk_read(DEFAULT_ENTRY, 0, rsize);

  return (uintptr_t)DEFAULT_ENTRY;
}
```

2.1.2 异常控制流

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，系统调用就是这样一种特殊的异常（在PA3中也没有实现除了系统调用以外的异常的异常处理程序）。为了能成功运行《仙剑奇侠传》，为Nanos-lite这一操作系统实现系统调用是必须的，尽管PA3中并不对除系统调用以外的任何异常做处理，但依然需要搭建完整的异常控制流。

2.1.2.1 准备IDT

CPU检测到异常之后，需要跳转到专门的异常处理程序处来对异常进行处理。在i386中，异常跳转的目标通过门描述符（Gate Descriptor）来指示，门描述符是一个8字节的结构体，NEMU使用的门描述符的结构如图：

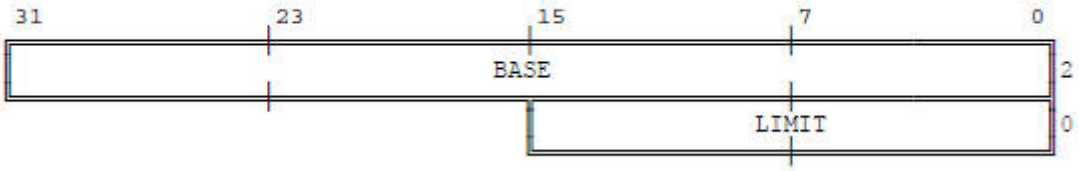


其中存在位P表示这一个门描述符是否有效；偏移量OFFSET指示跳转目标。

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门描述符数组IDT（Interrupt Descriptor Table，中断描述符表），可以通过索引来找到对应的门描述符（索引即异常的编号），并且i386使用IDTR寄存器来存放IDT的首地址和长度。

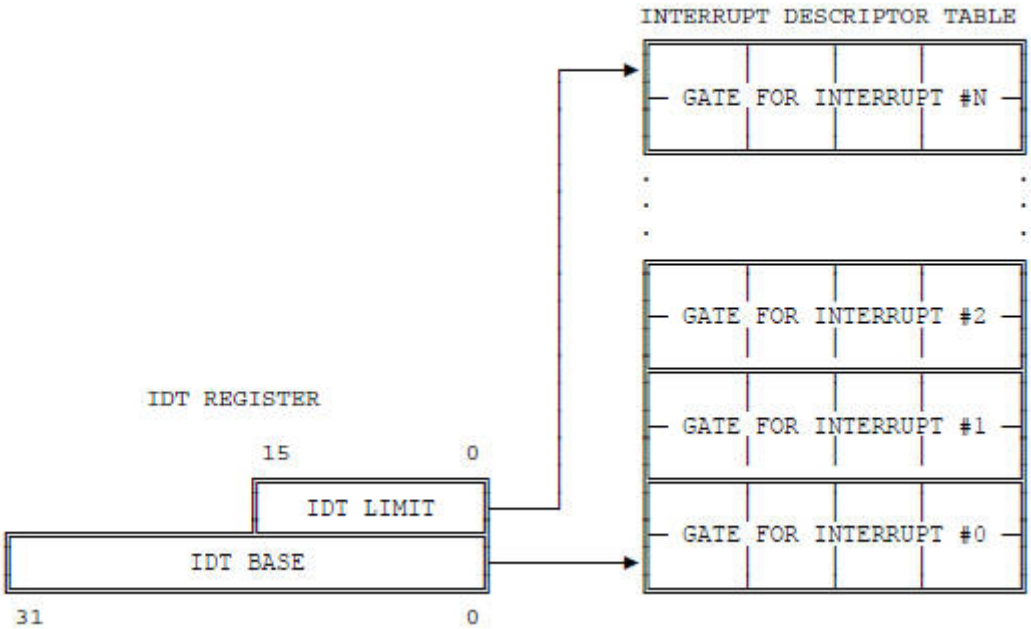
IDTR寄存器结构：

Figure 9-2. Pseudo-Descriptor Format for LIDT and SIDT



其中BASE存储IDT的首地址，LIMIT存储IDT的长度。

Figure 9-1. IDT Register and Table



所以为了构造在程序执行过程中可供使用的IDT，确保触发异常时能跳转到正确的异常处理程序，首先需要在NEMU的CPU中添加IDTR寄存器，

```
struct{
  uint16_t limit;
  uint32_t base;
}idtr;
```

然后在`nanos-lite/src/main.c`中定义宏HAS_ASYE，这样Nanos-lite会在初始化时调用`init_irq()`函数，最终会调用`_asye_init()`函数（`nexus-am/am/arch/x86-nemu/src/asye.c`中）`_asye_init()`函数会初始化IDT：

- 定义门描述符结构体数组idt
- 在相应的数组元素中填写有意义的门描述符
- 在IDTR寄存器中设置idt的首地址和长度

并注册一个事件处理函数`do_event()`，`do_event()`函数会根据事件（异常）类型进行分发。

实现了IDTR寄存器已经为IDT提供了需要的硬件结构，但还需要实现LIDT指令来将限制值limit和基地址base加载到IDTR寄存器中的limit与base部分。

LIDT指令实现

查询LIDT指令的具体细节，

LGDT/LIDT — Load Global/Interrupt Descriptor Table Register

Opcode	Instruction	Clocks	Description
0F 01 /2	LGDT m16&32	11	Load m into GDTR
0F 01 /3	LIDT m16&32	11	Load m into IDTR

```
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;
```

位于Opcode Map中的0F 01/3处，即group7（011）处，具体指令格式为LIDT m16&32。NEMU中框架代码已经选择好了对应的译码函数`make_DHelper(gp7_E)`与操作数宽度（默认的2/4），只需要实现对应的执行函数`make_EHelper(lidt)`，并在`all-instr.h`文件中添加执行函数`make_EHelper(lidt)`的声明即可，

```
make_DHelper(gp7_E) {
  decode_op_rm(eip, id_dest, false, NULL, false);
}

make_EHelper(lidt) {
  rtl_lm(&t0, &id_dest->addr, 2); //读取id_dest->addr处前两个字节内容
  cpu.idtr.limit = t0; //加载到IDTR寄存器中的limit部分
  id_dest->addr = id_dest->addr + 2; //id_dest->addr地址后移两个字节
```

```
  if (id_dest->width == 2)
  { //若操作数宽度为16，将id_dest->addr处三个字节内容加载到IDTR寄存器中的base部分
    rtl_lm(&t1, &id_dest->addr, 3);
    cpu.idtr.base = t1;
  }
  else
  { //否则将id_dest->addr处四个字节内容加载到IDTR寄存器中的base部分
    rtl_lm(&t1, &id_dest->addr, 4);
    cpu.idtr.base = t1;
  }

  print_asm_template1(lidt);
}

//all-instr.h
make_EHelper(lidt);
```

指令打包为`EX(lidt)`，将打包好的指令填入到`opcode_table`的对应位置。

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

2.1.2.2 触发异常

实现了IDTR寄存器和LIDT指令后Nanos-lite就可以准备可使用的IDT了，在NEMU上运行的用户程序就可以使用INT 0x80指令来进行系统调用了，所以还需要实现INT指令。

INT指令实现

查询INT指令的具体细节，

INT/INTO — Call to Interrupt Procedure

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

在NEMU中INT指令的具体操作为：

- 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

在PA3中仅实现CD处的INT指令，具体指令格式为INT imm8，选择译码函数`make_DHelper(I)`与操作数宽度（1），需要实现对应的执行函数`make_EHelper(int)`，并在`all-instr.h`文件中添加执行函数`make_EHelper(int)`的声明即可。这里通过`raise_intr()`函数（在`nemu/src/cpu/intr.c`中定义）来模拟触发异常后的硬件处理过程，并在INT指令的执行函数`make_EHelper(int)`中调用`raise_intr()`函数，在此之前还需要在NEMU的CPU中添加CS寄存器（为了在QEMU中顺利进行Differential testing）并在`restart()`函数中将其初始化为8，将EFLAGS初始化为0x2。

```
//nemu/include/cpu/reg.h
uint16_t cs;

//nemu/src/monitor/monitor.c
#define EFLAGS_START 0x00000002
#define CS_START 0x0008

cpu.eflags_init = EFLAGS_START;
cpu.cs = CS_START;
```

添加了CS寄存器并在`restart()`函数完成初始化后即可实现`raise_intr()`函数，

```
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.
   */
  rtl_push(&cpu.eflags_init); //压栈EFLAGS
  t0 = cpu.cs;
  rtl_push(&t0); //压栈CS，因为CS寄存器为16位，这里先将其内容扩展至32位再压栈
  rtl_push(&ret_addr); //压栈EIP

  vaddr_t addr;
  addr = cpu.idtr.base + 8 * NO; //根据异常（中断）号在IDT中进行索引，找到一个门描述符
  uint32_t offset_l = vaddr_read(addr, 2); //读取门描述符offset域高16位
  uint32_t offset_h = vaddr_read(addr + 6, 2); //读取门描述符offset域低16位
  decoding.jmp_eip = (offset_h << 16) + offset_l; //将offset域组合成目标地址
  decoding.is_jmp = 1; //设置跳转标志
}
```

最后在INT指令的执行函数`make_EHelper(int)`中调用`raise_intr()`函数，并添加执行函数`make_EHelper(int)`的声明。

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

make_EHelper(int) {
  uint8_t NO = id_dest->val & 0x000000ff; //读取NO
  raise_intr(NO, decoding.seq_eip); //调用raise_intr()，保存指令为int指令的下一条指令

  print_asm("int %s", id_dest->str);

#ifdef DIFF_TEST
  diff_test_skip_nemu();
#endif
}

//all-instr.h
make_EHelper(int)
```

指令打包为`IDEXW(I, int, 1)`，将打包好的指令填入到`opcode_table`的对应位置。

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

到此为止异常处理的硬件部分流程已经完成，需要实现异常处理的软件部分了。

2.1.2.3 保存现场

进行异常处理的时候不可避免地需要用到通用寄存器，然而现在的通用寄存器里存放的都是异常触发之前的内容，如果不保存就覆盖，将来就无法恢复异常触发之前的状态了。但硬件并不负责保存这些通用寄存器的内容，因此需要通过软件代码来保存它们的值。

函数`vecsys()`（`nexum-am/am/arch/x86-nemu/src/trap.S`中定义）会压入错误码和异常号#irq，然后跳转到`asm_trap()`，

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

asm_trap:
  pushal

  pushl %esp
  call irq_handle

  addl $4, %esp

  popal
  addl $8, %esp

  iret
```

`asm_trap()`会先使用PUSHA指令将用户进程的通用寄存器保存在堆栈中，然后将当前ESP保存至堆栈中，最后调用`irq_handle`（`nexus-am/am/arch/x86-nemu/src/asye.c`中定义）处理函数。因此需要先实现PUSHA指令来保存通用寄存器的值。

PUSHA指令实现

查询PUSHA指令具体细节，

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;
```

在PA3中只需实现PUSHAD指令，将八个32位通用寄存器的值压入堆栈，不需要译码函数，只需要实现对应的执行函数`make_EHelper(pusha)`，并在`all-instr.h`文件中添加执行函数`make_EHelper(pusha)`的声明即可。

```
make_EHelper(pusha) {
  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");
}

//all-instr.h
make_EHelper(pusha)
```

指令打包为`E(pusha)`，将打包好的指令填入到`opcode_table`的对应位置。

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

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，完整记录了用户进程触发异常时现场的状态。

trap frame（陷阱帧）结构

（栈底）

- EFLAGS
- CS
- EIP
- error_code
- irq
- GPRs

（栈顶）

但为了让`irq_handle()`以及后续代码可以正确地使用trap frame（trap frame中包含了系统调用的参数），还需要重新组织`nexus-am/am/arch/x86-nemu/include/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 frame中保持一致，注意栈由高地址逐渐填充至低地址，而结构体中的内容从低地址开始，因此`_RegSet`结构体中的成员顺序应与trap frame中成员顺序相反。

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

2.1.2.4 事件分发

随后`irq_handle()`函数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`do_event()`（`nanos-lite/src/irq.c`中定义）来处理，`do_event()`根据事件类型再次进行分发。PA3中只需要处理系统调用事件`_EVENT_SYSCALL`（`nexus-am/am/am.h`中定义）。在识别出系统调用事件`_EVENT_SYSCALL`后，调用`do_syscall()`（`nanos-lite/src/syscall.c`中定义）进行处理。

```
extern _RegSet* do_syscall(_RegSet *r); //do_syscall()函数声明

static _RegSet* do_event(_Event e, _RegSet* r) {
  switch (e.event) {
    case _EVENT_SYSCALL: //识别出系统调用事件_EVENT_SYSCALL
      do_syscall(r); //调用do_syscall()处理
      break;
    default: panic("Unhandled event ID = %d", e.event);
  }

  return NULL;
}
```

2.1.2.5 系统调用

为了便于`do_syscall()`函数从保存的现场reg中获取参数并在系统调用完成后设置返回值，还需要在`nexus-am/am/arch/x86-nemu/include/arch.h`中实现正确的`SYSCALL_ARGx()`宏，因为系统调用函数`_syscall_()`（`navy-apps/libs/libos/src/nanos.c`中定义）会将系统调用的参数依次放在EAX，EBX，ECX，EDX四个寄存器中，

```
int _syscall_(int type, uintptr_t a0, uintptr_t a1, uintptr_t a2){
  int ret = -1;
  asm volatile("int $0x80": "=a"(ret): "a"(type), "b"(a0), "c"(a1), "d"(a2));
  return ret;
}
```

所以这里的`SYSCALL_ARGx()`宏也只需要四个即可，分别对应寄存器EAX，EBX，ECX和EDX。

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

到此为止Nanos-lite已经可以在`do_syscall()`函数中首先通过宏`SYSCALL_ARG1(r)`从现场reg中获取用
户进程之前设置好的系统调用参数，识别到用户进程所进行的系统调用了，接下来需要通过第一个参数-系统调用号-进行分发来为不同的系统调用提供不同的处理。添加一个系统调用也需要在分发的过程中添加相应的系统调用号，（如果需要的话）编写相应的系统调用处理函数`sys_xxx()`，然后调用它即可。

为了保证阶段一中`dummy`程序的正常运行，需要添加SYS_none系统调用与SYS_exit系统调用，其中SYS_none系统调用什么都不做，直接返回1；而SYS_exit系统调用会接收一个退出状态的参数，并用这个参数调用`_halt()`。

```
_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:
      a[0] = 1;
      break;

    case SYS_exit:
      _halt(a[1]);
      break;

    default: panic("Unhandled syscall ID = %d", a[0]);
  }
  SYSCALL_ARG1(r) = a[0];

  return NULL;
}
```

实现了系统调用函数后，Nanos-lite就能够实现用户程序的系统调用并在EAX寄存器中保存系统调用函数的返回值了。

2.1.2.6 恢复现场

系统调用处理结束后代码将会返回到`trap.S`的`asm_trap()`中，为了保证用户程序能够继续执行，`asm_trap()`会通过之前保存的trap frame中的内容，恢复用户进程的通用寄存器并执行IRET指令，从异常处理代码中返回，恢复EIP，CS与EFLAGS。用户进程可以从EAX寄存器中获得系统调用的返回值，进而得知系统调用执行的结果。

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

asm_trap:
  pushal
```

```
  pushl %esp
  call irq_handle

  addl $4, %esp

  popal
  addl $8, %esp

  iret
```

所以为了完成整个异常控制流，还需要实现POPA和IRET指令来确保用户程序可以从异常处理中返回并恢复到异常前的状态。

POPA指令实现

查询PUSHA指令具体细节，

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;
```

在PA3中只需实现POPAD指令，将原本压入堆栈的八个32位通用寄存器的值弹出（不需要保存ESP），不需要译码函数，只需要实现对应的执行函数`make_EHelper(popa)`，并在`all-instr.h`文件中添加执行函数`make_EHelper(popa)`的声明即可。

```
make_EHelper(popa) {
  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");
}

//all-instr.h
make_EHelper(popa)
```

指令打包为`E(popa)`，将打包好的指令填入到`opcode_table`的对应位置。

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

IRET指令实现

查询IRET指令具体细节，

IRET/IRETD — Interrupt Return

Opcode	Instruction	Clocks	Description
CF	IRET	22,pm=38	Interrupt return (far return and pop flags)
CF	IRET	pm=82	Interrupt return to lesser privilege
CF	IRET	ts	Interrupt return, different task (NT = 1)
CF	IRETD	22,pm=38	Interrupt return (far return and pop flags)
CF	IRETD	pm=82	Interrupt return to lesser privilege
CF	IRETD	pm=60	Interrupt return to V86 mode
CF	IRETD	ts	Interrupt return, different task (NT = 1)

在NEMU中IRET指令的具体操作为：

- 将栈顶的三个元素来依次解释成EIP，CS，EFLAGS，并恢复它们
- 跳转到EIP寄存指令处继续执行

不需要译码函数，只需要实现对应的执行函数`make_EHelper(int)`，并在`all-instr.h`文件中添加执行函数`make_EHelper(int)`的声明即可。

```
make_EHelper(iret) {
  rtl_pop(&decoding.jmp_eip);
  rtl_pop(&t0);
  cpu.cs = t0 & 0x0000ffff;
  rtl_pop(&cpu.eflags_init);
  decoding.is_jmp = 1;

  print_asm("iret");
}

//all-instr.h
make_EHelper(iret)
```

指令打包为`E(popa)`，将打包好的指令填入到`opcode_table`的对应位置。

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

异常控制流到此搭建完毕。

2.2 运行结果

成功运行`navy-apps/tests`目录下的`dummy`程序（运行结果是在PA3全部完成后截图，所以是从文件系统中加载的`dummy`程序）

```
miaozeyun@Ubuntu32:~/ics2017/nanos-lite$ make run
Building nanos-lite [x86-nemu]
+ CC src/main.c
make[1]: Entering directory '/home/miaozeyun/ics2017/nexus-am'
make[2]: Entering directory '/home/miaozeyun/ics2017/nexus-am/am'
Building am [x86-nemu]
make[2]: Nothing to be done for 'archive'.
make[2]: Leaving directory '/home/miaozeyun/ics2017/nexus-am/am'
make[1]: Leaving directory '/home/miaozeyun/ics2017/nexus-am'
make[1]: Entering directory '/home/miaozeyun/ics2017/nexus-am/libs/klib'
make[1]: *** 没有指明目标并且找不到 makefile。 停止。
make[1]: Leaving directory '/home/miaozeyun/ics2017/nexus-am/libs/klib'
/home/miaozeyun/ics2017/nexus-am/Makefile.compile:86: recipe for target 'klib' f
ailed
make: [klib] Error 2 (ignored)
make[1]: Entering directory '/home/miaozeyun/ics2017/nemu'
./build/nemu -l /home/miaozeyun/ics2017/nanos-lite/build/nemu-log.txt /home/miao
zeyun/ics2017/nanos-lite/build/nanos-lite-x86-nemu.bin
[src/monitor/monitor.c,67,load_img] The image is /home/miaozeyun/ics2017/nanos-l
ite/build/nanos-lite-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,32,welcome] Build time: 21:22:24, May 16 2022
For help, type "help"
(nemu) c
[src/main.c,19,main] 'Hello World!' from Nanos-lite
[src/main.c,20,main] Build time: 21:25:58, May 16 2022
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x102230, end = 0x1d4c1e5,
 size = 29663157 bytes
[src/main.c,27,main] Initializing interrupt/exception handler...
[src/loader.c,14,loader] loader fd:11
nemu: HIT GOOD TRAP at eip = 0x00100032
```

2.3 Bug总结

阶段一中PA实验流程上对程序加载模块（loader）和异常控制流的流程描述的已经非常清楚，所有需要实现的部分也都在PA实验流程上指出，所以基本不会出现有某个部分没有实现导致的Bug，出现的Bug主要还是在指令的实现部分，LIDT，PUSHA，POPA，IRET指令操作都比较简单，按照i386手册或PA实验流程中描述实现即可，INT指令实现稍难，但PA实验流程中也给出了详细的描述和需要注意的点。

还有需要注意的点就是依照trap frame（陷阱帧）的结构进行`nexus-am/am/arch/x86-nemu/include/arch.h`中定义的`_RegSet`结构体成员重排序时要注意地址的问题，否则Nanos-lite就无法从reg中获取正确的参数，会出现报错：

```
[src/irq.c,5,do_event] {kernel} system panic: Unhandled event ID = 3
```

3 阶段二

在阶段一中已经添加了SYS_none系统调用与SYS_exit系统调用，为了让程序能够通过`write()`和`printf()`这样的库函数来实现输出，还需要实现SYS_write系统调用。

但若只实现SYS_write系统调用，用户程序通过`printf()`输出的时候是逐个字符地调用`write()`来输出的，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。所以还需要实现`_sbrk()`函数来调整堆区大小，并提供SYS_brk系统调用以供用户程序使用。

《仙剑奇侠传》的运行需要多个文件的支撑，为了能在NEMU上运行仙剑奇侠传，实现一个简易的文件系统是必须的，PA3中对文件系统的需求比较简单：

- 每个文件的大小是固定的
- 写文件时不允许超过原有文件的大小
- 文件的数量是固定的，不能创建新文件
- 没有目录

需要实现 `fs_open()`，`fs_read()`，`fs_close()`，`fs_write()` 和 `fs_lseek()` 来对文件系统中的文件进行打开、读取、关闭、写入以及偏移量调整的操作。添加SYS_open，SYS_read，SYS_close，SYS_lseek系统调用，修改SYS_write系统调用以供用户程序使用。

3.1 实现方法及代码分析

3.1.1 标准输出

在PA2中已经添加了串口，让程序能够进行输出，但PA2中的程序相当于直接运行在NEMU上，而PA3中的程序运行在Nanos-lite这一操作系统上，所以用户程序想要进行输出需要操作系统来帮忙，需要添加SYS_write系统调用让程序能够通过 `write()` 和 `printf()` 这样的库函数来进行输出。

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），最后设置正确的返回值，如果写入成功，则会返回写入的字节数；否则返回-1。

```
uintptr_t sys_write(int fd, const void *buf, size_t count)
{
  //Log("sys_write function");
  if (fd == 1 || fd == 2) //检查fd值是否为1/2
  {
    for(int i = 0; i < count; i++)
      _putc(((char *)buf)[i]); //调用_putc()将buf为首地址的count字节输出到串口
    return count; //成功返回count
  }
  else
  {
    panic("Unhandled fd=%d in sys_write", fd);
    return -1; //失败返回-1
  }
}

_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:
      a[0] = 1;
      break;

    case SYS_exit:
      _halt(a[1]);
      break;

    case SYS_write:
      //Log("sys_write");
      a[0] = sys_write((int)a[1], (void *)a[2], (size_t)a[3]);
```

```
      break;

    default: panic("Unhandled syscall ID = %d", a[0]);
  }
  SYSCALL_ARG1(r) = a[0];

  return NULL;
}
```

此外还需要在`navy-apps/libs/libos/src/nanos.c`的`_write()`中调用系统调用接口函数，这样通过Navy-apps编译出的程序就可以调用系统接口进行输出了。

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

3.1.2 堆区管理

如果未实现堆区管理，那么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，所以`hello`程序就是逐个字符地调用`write()`来输出，这显然效率非常低下，为了能让用户程序一次调用`printf()`就能够输出所有字符，需要为库函数添加堆区管理功能，并为Nanos-lite添加SYS_brk系统调用。

`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。堆区的使用情况由`libc`来进行管理，而堆区的大小需要通过系统调用向操作系统提出更改。调整堆区大小的时候就是在调整用户程序可用的内存区域。

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。`program break`就是用户程序的数据段（`data segment`）结束的位置。链接的时候`ld`会默认添加一个名为`_end`的符号来指示程序的数据段结束的位置。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。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。

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()`的功能，还需要提供一个用于设置堆区大小的系统调用—SYS_brk，它接收一个参数`addr`，用于指示新的`program break`的位置。

`_sbrk()`通过记录的方式来对用户程序的`program break`位置进行管理，其工作方式如下：

- `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

```
extern char _end;
void *_sbrk(intptr_t increment)
{
  static void *programe_break = (void *)&_end; //获取program break最开始时的位置
  void *old_programe_break = programe_break;
  programe_break += increment; //计算新program break
  //通过SYS_brk系统调用来让操作系统设置新的program break
  if(_syscall_(SYS_brk, (uintptr_t)programe_break, 0, 0) == 0)
    return (void *)old_programe_break; //系统调用成功，返回旧program break的位置
  else
    return (void *)-1; //系统调用失败，返回-1
}
```

在Nanos-lite中添加设置堆区大小的系统调用SYS_brk。

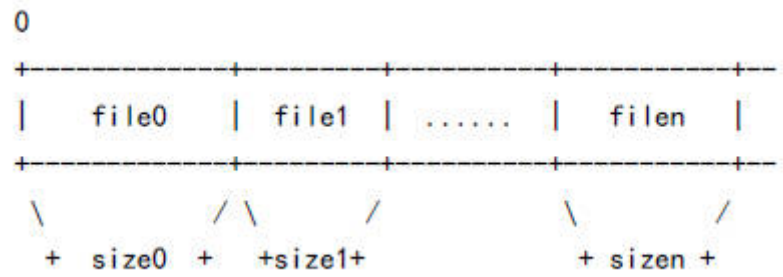
```
_RegSet* do_syscall(_RegSet *r) {
  uintptr_t a[4];
  a[0] = SYSCALL_ARG1(r);
  a[1] = SYSCALL_ARG2(r);
  a[2] = SYSCALL_ARG3(r);
  a[3] = SYSCALL_ARG4(r);

  switch (a[0]) {
    case SYS_none:
      a[0] = 1;
      break;

    case SYS_exit:
      _halt(a[1]);
      break;

    case SYS_write:
      //Log("sys_write");
      a[0] = sys_write((int)a[1], (void *)a[2], (size_t)a[3]);
      break;

    case SYS_brk:
      a[0] = 0;
      break;

    default: panic("Unhandled syscall ID = %d", a[0]);
  }
  SYSCALL_ARG1(r) = a[0];

  return NULL;
}
```

3.1.3 简易文件系统

Nanos-lite对文件系统的需求并不复杂：

- 每个文件的大小是固定的
- 写文件时不允许超过原有文件的大小
- 文件的数量是固定的，不能创建新文件
- 没有目录

因此可以将每一个文件分别固定在ramdisk中的某一个位置，PA3约定文件从ramdisk的最开始一个挨着一个地存放：



为了记录ramdisk中各个文件的名字和大小，还需要一张"文件记录表"，Nanos-lite的`Makefile`已经提供了维护这些信息的脚本，先对`nanos-lite/Makefile`作如下修改：

```
--- nanos-lite/Makefile
+++ nanos-lite/Makefile
@@ -34,2 +34,2 @@
-update: update-ramdisk-objcopy src/syscall.h
+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`。这个文件记录表是一个数组，数组中的每个元素包含了文件名，文件大小和文件在ramdisk中的偏移，并且这三项信息都是固定不变的（因为没有目录所以将目录分隔符/也认为是文件名的一部分）。

```
//nanos-lite/src/fs.c
typedef struct {
  char *name; //文件名
  size_t size; //文件大小
  off_t disk_offset; //文件在ramdisk中的偏移
  off_t open_offset;
} Finfo;

enum {FD_STDIN, FD_STDOUT, FD_STDERR, FD_FB, FD_EVENTS, FD_DISPINFO, FD_NORMAL};

/* This is the information about all files in disk. */
static Finfo file_table[] __attribute__((used)) = {
  {"stdin (note that this is not the actual stdin)", 0, 0},
  {"stdout (note that this is not the actual stdout)", 0, 0},
  {"stderr (note that this is not the actual stderr)", 0, 0},
  [FD_FB] = {"/dev/fb", 0, 0},
  [FD_EVENTS] = {"/dev/events", 0, 0},
  [FD_DISPINFO] = {"/proc/dispinfo", 128, 0},
#include "files.h"
};
```

因为操作系统中确实存在不少"没有名字"的文件（例如先前提到的`stdout`和`stderr`，上述的代码块中的`FD_EVENTS`和`FD_DISPINFO`等）。为了统一管理它们，操作系统通过一个编号—文件描述符（file descriptor）来表示文件，一个文件描述符对应一个正在打开的文件。Nanos-lite中，由于简易文件系统中的文件数目是固定的，所以可以把文件记录表的下标作为相应文件的文件描述符返回给用户程序，并且所有文件操作都通过文件描述符来标识文件。

为了记录目前文件操作的位置，需要为每一个已经打开的文件引入偏移量属性`open_offset`，偏移量可以通过`lseek()`系统调用来调整。

```
typedef struct {
  char *name;
  size_t size;
  off_t disk_offset;
  off_t open_offset; //文件被打开后读写指针的偏移
} Finfo;
```

为了方便用户程序进行标准输入输出，操作系统准备了三个默认的文件描述符：`FD_STDIN`，`FD_STDOUT`和`FD_STDERR`，分别对应标准输入`stdin`，标准输出`stdout`和标准错误`stderr`，`printf`最终调用`write(FD_STDOUT,buf,len)`进行输出；而`scanf`将会通过调用`read(FD_STDIN,buf,len)`进行读入。

在简易文件系统需要实现的文件操作包括：

```
int fs_open(const char *pathname, int flags, int mode);
ssize_t fs_read(int fd, void *buf, size_t len);
ssize_t fs_write(int fd, const void *buf, size_t len);
off_t fs_lseek(int fd, off_t offset, int whence);
int fs_close(int fd);
```

首先在`nanos-lite/src/fs.h`文件中声明需要实现的文件操作函数：

```
int fs_open(const char *pathname, int flgas, int mode);
ssize_t fs_read(int fd, void *buf, size_t len);
ssize_t fs_write(int fd, const void *buf, size_t len);
off_t fs_lseek(int fd, off_t offset, int whence);
int fs_close(int fd);
size_t fs_filesz(int fd);
```

接下来逐个实现这些文件操作函数。

`fs_open()`实现要点：

- 简易文件系统中每一个文件都是固定的，不会产生新文件，因此"`fs_open()`没有找到`pathname`所指示的文件"属于异常情况，需要使用`assertion`终止程序运行
- 为了简化实现，允许所有用户程序都可以对所有已存在的文件进行读写，这样在实现`fs_open()`的时候就可以忽略`flags`和`mode`了

```
int fs_open(const char *pathname, int flags, int mode)
{
  //Log("fs_open pathname:%s",pathname);
  for(int i = 0; i < NR_FILES; i++) //遍历文件记录表
  {
    if (strcmp(pathname, file_table[i].name) == 0) //若pathname匹配
    {
      file_table[i].open_offset = 0; //将对应文件open_offset设为0
      return i; //返回该文件在文件记录表的下标（文件描述符）
    }
  }
  panic("illegal pathname(failure in fs_open)"); //否则终止程序运行
  return -1; //返回-1
}
```

`fs_read()` 实现要点：

- 使用 `ramdisk_read()` 来进行文件的真正读取
- 由于文件的大小是固定的，注意偏移量不要越过文件的边界
- 除了写入 `stdout` 和 `stderr` 之外（用 `_putc()` 输出到串口），其余对于 `stdin`，`stdout` 和 `stderr` 这三个特殊文件的操作可以直接忽略

可以添加辅助函数 `size_t fs_filesz(int fd)` 来获取文件描述符 `fd` 所描述的文件的大小。

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

ssize_t fs_read(int fd, void *buf, size_t len)
{
  //Log("fs_read fd:%d,buf:%d,len:%d,size = %d", fd, buf, len,
file_table[fd].size);
  switch(fd)
  {
    case FD_STDIN: //stdin，stdout和stderr特殊处理
    case FD_STDOUT:
    case FD_STDERR:
      return -1;
    default: //读取磁盘文件
    {
      //获取读取位置在磁盘中总的偏移量
      off_t offset = file_table[fd].disk_offset + file_table[fd].open_offset;
      //确保偏移量不会越过文件的边界
      if(file_table[fd].open_offset + len >= fs_filesz(fd))
        len = fs_filesz(fd) - file_table[fd].open_offset;
      //将ramdisk中从offset开始的len大小的字节读入到buf处
      ramdisk_read((void *)buf, offset, len);
      file_table[fd].open_offset += len; //文件读写指针更新
      break;
    }
  }
  return len; //返回读取的字节数
}

size_t fs_filesz(int fd)
{
  return file_table[fd].size; //获取文件描述符`fd`所描述的文件的大小
}
```

`fs_close()` 实现要点：

- 简易文件系统没有维护文件打开的状态，`fs_close()` 可以直接返回0，表示总是关闭成功

```
int fs_close(int fd)
{
  //Log("fs_close fd:%d",fd);
  return 0;
}
```

修改程序加载模块（loader）以实现加载文件系统中文件，实现之后更换用户程序只需要修改传入 `loader()` 函数的文件名即可，无需更新ramdisk的内容（除非ramdisk上的内容确实需要更新，例如重新编译了Navy-apps的程序）：

```
#include "fs.h"

uintptr_t loader(_Protect *as, const char *filename) {
  //size_t rsize = get_ramdisk_size();
  //ramdisk_read(DEFAULT_ENTRY, 0, rsize);

  int fd = fs_open(filename, 0, 0); //打开文件名称为filename文件，文件标识符设为fd
  Log("loader fd:%d",fd);
  size_t rsize = fs_filesz(fd); //获取文件标识符设为fd的文件大小
  fs_read(fd, DEFAULT_ENTRY, rsize); //读取文件标识符设为fd的文件
  fs_close(fd); //关闭文件标识符设为fd的文件

  return (uintptr_t)DEFAULT_ENTRY;
}
```

`fs_write()` 实现要点：

- 使用 `ramdisk_write()` 来进行文件的真正写入
- 由于文件的大小是固定的，注意偏移量不要越过文件的边界
- 除了写入 `stdout` 和 `stderr` 之外（用 `_putc()` 输出到串口），其余对于 `stdin`，`stdout` 和 `stderr` 这三个特殊文件的操作可以直接忽略

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

ssize_t fs_write(int fd, const void *buf, size_t len)
{
  //Log("fs_write fd:%d,buf:%d,len:%d", fd, buf, len);
  switch(fd)
  {
    case FD_STDIN: //stdin特殊处理
      return -1;
    case FD_STDOUT: //写入stdout和stderr用_putc()输出到串口
    case FD_STDERR:
    {
      for(int i = 0; i < len; i++)
        _putc(((char *)buf)[i]);
      break;
    }
    default: //写入磁盘文件
    {
      //获取写入位置在磁盘中总的偏移量
      off_t offset = file_table[fd].disk_offset + file_table[fd].open_offset;
      //确保偏移量不会越过文件的边界
      if(file_table[fd].open_offset + len >= fs_filesz(fd))
        len = fs_filesz(fd) - file_table[fd].open_offset;
      //将从buf开始的len大小的字节写入到ramdisk中offset处
      ramdisk_write(buf, offset, len);
      file_table[fd].open_offset += len; //文件读写指针更新
      break;
    }
  }
  return len;  //返回写入的字节数
}
```

`fs_lseek()` 实现：

```
//fs.h
enum {SEEK_SET, SEEK_CUR, SEEK_END};

off_t fs_lseek(int fd , off_t offset, int whence)
{
  //Log("fs_lseek fd:%d,offset:%d,whence:%d", fd, offset, whence);
  switch (whence)
  {
    case SEEK_SET: //设置偏移量为offset
    {
      //判断offset是否超出当前操作的文件大小
      if (offset >=0 && offset <= fs_filesz(fd))
      {
        file_table[fd].open_offset = offset; //设置偏移量
        return file_table[fd].open_offset; //设置成功返回当前偏移位置
      }
      else
        return -1; //否则返回-1
    }
    case SEEK_CUR: //设置偏移量为当前偏移量+offset
    {
      //判断当前偏移量+offset是否超出当前操作的文件大小
      if (file_table[fd].open_offset + offset <= fs_filesz(fd))
      {
        file_table[fd].open_offset += offset; //设置偏移量
        return file_table[fd].open_offset; //设置成功返回当前偏移位置
      }
      else
        return -1; //否则返回-1
    }
    case SEEK_END: //设置偏移量为当前操作的文件大小+offset
    {
      file_table[fd].open_offset = fs_filesz(fd) + offset; //设置偏移量
      return file_table[fd].open_offset; //设置成功返回当前偏移位置
    }
    default:
     return -1; //其他情况返回-1
  }
}
```

实现了所有需要的文件操作函数后需要添加SYS_open，SYS_read，SYS_close，SYS_lseek系统调用，并修改SYS_write系统调用以供用户程序使用。

```
#include "fs.h"

_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:
      a[0] = 1;
      break;
```

```
    case SYS_exit:
      _halt(a[1]);
      break;

    case SYS_write:
      //Log("sys_write");
      //a[0] = sys_write((int)a[1], (void *)a[2], (size_t)a[3]);
      a[0] = fs_write((int)a[1], (void *)a[2], (size_t)a[3]);
      break;

    case SYS_brk:
      a[0] = 0;
      break;

    case SYS_open:
      a[0] = fs_open((char *)a[1], (int)a[2], (int)a[3]);
      break;

    case SYS_read:
      a[0] = fs_read((int)a[1], (void *)a[2], (size_t)a[3]);
      break;

    case SYS_close:
      a[0] = fs_close((int)a[1]);
      break;

    case SYS_lseek:
      a[0] = fs_lseek((int)a[1], (off_t)a[2], (int)a[3]);
      break;

    default: panic("Unhandled syscall ID = %d", a[0]);
  }
  SYSCALL_ARG1(r) = a[0];

  return NULL;
}
```

此外还需要在 navy-apps/libs/libos/src/nanos.c 的 _open , _read , _close , _lseek 中调用系统调用接口函数，这样通过Navy-apps编译出的程序就可以调用系统接口进行文件操作了。

```
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){
  //_exit(SYS_write);
  return  _syscall_(SYS_write, (uintptr_t)fd, (uintptr_t)buf, (uintptr_t)count);
}

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

int _close(int fd) {
  //_exit(SYS_close);
```

```
  return _syscall_(SYS_close, (uintptr_t)fd, 0, 0);
}

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

3.2 运行结果

成功运行`navy-apps/tests`目录下的`hello`程序且`printf()`将格式化完毕的字符串通过一次`write`系统调用进行输出，而不是逐个字符地进行输出（运行结果是在PA3全部完成后截图，所以是从文件系统中加载的`hello`程序）。

```
miaozeyun@Ubuntu32:~/ics2017/nanos-lite$ make run
Building nanos-lite [x86-nemu]
+ CC src/syscall.c
+ CC src/main.c
make[1]: Entering directory '/home/miaozeyun/ics2017/nexus-am'
make[2]: Entering directory '/home/miaozeyun/ics2017/nexus-am/am'
Building am [x86-nemu]
make[2]: Nothing to be done for 'archive'.
make[2]: Leaving directory '/home/miaozeyun/ics2017/nexus-am/am'
make[1]: Leaving directory '/home/miaozeyun/ics2017/nexus-am'
make[1]: Entering directory '/home/miaozeyun/ics2017/nexus-am/libs/klib'
make[1]: *** 没有指明目标并且找不到 makefile。 停止。
make[1]: Leaving directory '/home/miaozeyun/ics2017/nexus-am/libs/klib'
/home/miaozeyun/ics2017/nexus-am/Makefile.compile:86: recipe for target 'klib' f
ailed
make: [klib] Error 2 (ignored)
make[1]: Entering directory '/home/miaozeyun/ics2017/nemu'
./build/nemu -l /home/miaozeyun/ics2017/nanos-lite/build/nemu-log.txt /home/miao
zeyun/ics2017/nanos-lite/build/nanos-lite-x86-nemu.bin
[src/monitor/monitor.c,67,load_img] The image is /home/miaozeyun/ics2017/nanos-l
ite/build/nanos-lite-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,32,welcome] Build time: 21:22:24, May 16 2022
For help, type "help"
(nemu) c
[src/main.c,19,main] 'Hello World!' from Nanos-lite
[src/main.c,20,main] Build time: 21:34:37, May 16 2022
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x102270, end = 0x1d4c225,
 size = 29663157 bytes
[src/main.c,27,main] Initializing interrupt/exception handler...
[src/loader.c,14,loader] loader fd:10
[src/syscall.c,38,do_syscall] sys_write
Hello World!
[src/syscall.c,38,do_syscall] sys_write
Hello World for the 2th time
[src/syscall.c,38,do_syscall] sys_write
Hello World for the 3th time
[src/syscall.c,38,do_syscall] sys_write
Hello World for the 4th time
```

通过测试程序`/bin/text`测试。

```
miaozeyun@Ubuntu32:~/ics2017/nanos-lite$ make run
Building nanos-lite [x86-nemu]
+ CC src/syscall.c
+ CC src/main.c
make[1]: Entering directory '/home/miaozeyun/ics2017/nexus-am'
make[2]: Entering directory '/home/miaozeyun/ics2017/nexus-am/am'
Building am [x86-nemu]
make[2]: Nothing to be done for 'archive'.
make[2]: Leaving directory '/home/miaozeyun/ics2017/nexus-am/am'
make[1]: Leaving directory '/home/miaozeyun/ics2017/nexus-am'
make[1]: Entering directory '/home/miaozeyun/ics2017/nexus-am/libs/klib'
make[1]: *** 没有指明目标并且找不到 makefile。 停止。
make[1]: Leaving directory '/home/miaozeyun/ics2017/nexus-am/libs/klib'
/home/miaozeyun/ics2017/nexus-am/Makefile.compile:86: recipe for target 'klib' f
ailed
make: [klib] Error 2 (ignored)
make[1]: Entering directory '/home/miaozeyun/ics2017/nemu'
./build/nemu -l /home/miaozeyun/ics2017/nanos-lite/build/nemu-log.txt /home/miao
zeyun/ics2017/nanos-lite/build/nanos-lite-x86-nemu.bin
[src/monitor/monitor.c,67,load_img] The image is /home/miaozeyun/ics2017/nanos-l
ite/build/nanos-lite-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,32,welcome] Build time: 21:22:24, May 16 2022
For help, type "help"
(nemu) c
[src/main.c,19,main] 'Hello World!' from Nanos-lite
[src/main.c,20,main] Build time: 21:43:32, May 16 2022
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x102230, end = 0x1d4c1e5,
 size = 29663157 bytes
[src/main.c,27,main] Initializing interrupt/exception handler...
[src/loader.c,14,loader] loader fd:13
PASS!!!
nemu: HIT GOOD TRAP at eip = 0x00100032
```

3.3 Bug总结

因为Navy-apps与Nanos-lite是两个不相关的程序，所以在`navy-apps/libs/libos/src/nanos.c`中添加系统调用接口后需要重新编译生成的用户程序才能正常调用Nanos-lite所提供的用户接口。

还有需要注意的就是堆区管理中`_sbrk()`函数的实现，因为在实现`_sbrk()`函数后验证时磁盘内仅有`hello`一个程序，而是实现简易文件系统后磁盘中会有多个文件，所以可能实现的`_sbrk()`函数存在问题但是并没有在堆区管理中发现的问题。

简易文件系统的实现配合PA实验流程中的说明和Linux系统的手册可以非常清楚地查清楚需要实现的函数的参数，具体功能，不同情况下的返回值和对于特殊文件的处理，按照指导和手册一步步实现即可，基本没有什么Bug，注意不要忘记在`navy-apps/libs/libos/src/nanos.c`文件的对应函数中调用系统接口。

4 阶段三

在阶段二已经实现了完整的文件系统，用户程序已经可以读写普通的文件了。但读入按键/查询时钟/更新屏幕也是用户程序的合理需求，操作系统也需要提供支持。一种最直接的方式就是为每个功能单独提供一个系统调用，用户程序可以通过这些系统调用来使用相应的功能，但这样的实现方式不仅需要为不同的设备设计不同的接口，还可能会因为接口间的不一致导致程序间的交互出现问题。

既然文件就是字节序列，那么内存，磁盘，标准输入输出等字节序列都可以看成文件。Unix就是这样做的，因此有"一切皆文件"（Everything is a file）的说法。这种做法最直观的好处就是为不同的事物提供了统一的接口：文件。用户程序可以通过文件的接口来操作计算机上的一切，而不必对不同的设备进行详细的区分。GNU/Linux继承自Unix，也自然继承了这种优秀的特性。为了向用户程序提供统一的抽象，在这一阶段的实验中需要为Nanos-lite将IOE抽象成文件。

输出设备中串口已经被抽象成`stdout`和`stderr`，还需要抽象成为文件的就是VGA的显存。

输入设备包括键盘和时钟，需要将它们的输入包装成事件并抽象为文件。

4.1 实现方法及代码分析

4.1.1 将VGA显存抽象成文件

程序更新屏幕只需要将像素信息写入VGA的显存即可。于是，Nanos-lite需要把显存抽象成文件（显存本身也是一段存储空间，以行优先的方式存储了将要在屏幕上显示的像素）。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；屏幕大小的信息通过`/proc/dispinfo`文件来获得，它需要支持读操作，以便于用户程序获得屏幕大小的信息。`/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，需要在`fs_read()`和`fs_write()`的实现中对它们进行"重定向"。

为了将VGA显存抽象为文件，首先需要在`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；

```
void init_fs() {
  // TODO: initialize the size of /dev/fb
  //使用IOE定义的API来获取屏幕的大小
  file_table[3].size = _screen.width * _screen.height * sizeof(uint32_t);
  file_table[3].open_offset = 0; //将/dev/fb文件内读写指针偏移量初始化为0
}
```

其次实现`fb_write()`（在`nanos-lite/src/device.c`中定义，把`buf`中的`len`字节写到屏幕上`offset`处），使得用户程序可以写VGA显存；

```
void fb_write(const void *buf, off_t offset, size_t len) {
  int size = sizeof(uint32_t); //获取单个像素大小
  offset /= size; //计算像素单位的offset
  len /= size; //计算像素单位的len
  int x = offset % _screen.width; //获取绘制起始点x坐标
  int y = offset / _screen.width; //获取绘制起始点y坐标
  //_draw_rect(buf, x, y, len, 1);
  //return;
  if (len + x <= _screen.width)
  { //单行绘制一次
    _draw_rect(buf, x, y, len, 1);
    return;
  }
  else if (len + x <= 2 * _screen.width)
  { //两行绘制两次
    int first_len = _screen.width - x;
    _draw_rect(buf, x, y, first_len, 1);
    _draw_rect(buf + first_len * size, 0, y + 1, len - first_len, 1);
    return;
  }
  else
  { //大于等于三行绘制三次
    int first_len = _screen.width - x;
    int other_len = (len - first_len) / _screen.width;
    int last_len = len - first_len - other_len * _screen.width;
    _draw_rect(buf, x, y, first_len, 1);
    _draw_rect(buf + first_len * size, 0, y + 1, _screen.width, other_len);
    _draw_rect(buf + (first_len + other_len * _screen.width) * size, 0, y +
other_len, last_len, 1);
    return;
```

```
  }
}
```

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；

```
void init_device() {
  _ioe_init();

  // TODO: print the string to array `dispinfo` with the format
  // described in the Navy-apps convention
  //Log("WIDTH:%d,HEIGHT:%d.", _screen.width, _screen.height);
  //使用IOE定义的API来获取实际屏幕的大小
  sprintf(dispinfo, "WIDTH:%d\nHEIGHT:%d\n", _screen.width, _screen.height);
}
```

实现`dispinfo_read()`（在`nanos-lite/src/device.c`中定义，把字符串`dispinfo`中`offset`开始的`len`字节写到`buf`中），使得用户程序可以获得屏幕大小的信息；

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

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

```
extern void fb_write(const void *buf, off_t offset, size_t len);
extern void dispinfo_read(void *buf, off_t offset, size_t len);

ssize_t fs_read(int fd, void *buf, size_t len)
{
  //Log("fs_read fd:%d,buf:%d,len:%d,size = %d", fd, buf, len,
file_table[fd].size);
  switch(fd)
  {
    case FD_STDIN:
    case FD_STDOUT:
    case FD_STDERR:
      return -1;
    case FD_DISPINFO: //读取/proc/dispinfo文件
    {
      //确保偏移量不会越过文件的边界
      if(file_table[fd].open_offset + len >= fs_filesz(fd))
        len = fs_filesz(fd) - file_table[fd].open_offset;
      //调用dispinfo_read()进行读取
      dispinfo_read((void *)buf, file_table[fd].open_offset, len);
      file_table[fd].open_offset += len;  //文件读写指针更新
      break;
    }
    default:
    {
      off_t offset = file_table[fd].disk_offset + file_table[fd].open_offset;
      if(file_table[fd].open_offset + len >= fs_filesz(fd))
        len = fs_filesz(fd) - file_table[fd].open_offset;
      ramdisk_read((void *)buf, offset, len);
      file_table[fd].open_offset += len;
```

```
      break;
    }
  }
  return len;
}

ssize_t fs_write(int fd, const void *buf, size_t len)
{
  //Log("fs_write fd:%d,buf:%d,len:%d", fd, buf, len);
  switch(fd)
  {
    case FD_STDIN:
      return -1;
    case FD_STDOUT:
    case FD_STDERR:
    {
      for(int i = 0; i < len; i++)
        _putc(((char *)buf)[i]);
      break;
    }
    case FD_FB: //写入/dev/fb文件
    {
      //确保偏移量不会越过文件的边界
      if(fs_filesz(fd) <= len + file_table[fd].open_offset)
        len = fs_filesz(fd) - file_table[fd].open_offset;
      //调用fb_write()进行写入
      fb_write(buf, file_table[fd].open_offset, len);
      file_table[fd].open_offset += len; //文件读写指针更新
      break;
    }
    default:
    {
      off_t offset = file_table[fd].disk_offset + file_table[fd].open_offset;
      if(file_table[fd].open_offset + len >= fs_filesz(fd))
        len = fs_filesz(fd) - file_table[fd].open_offset;
      ramdisk_write(buf, offset, len);
      file_table[fd].open_offset += len;
      break;
    }
  }
  return len;
}
```

4.1.2 将设备输入抽象成文件

输入设备有键盘和时钟，需要把它们的输入包装成事件。一种简单的方式是把事件以文本的形式表现出来，定义以下事件，一个事件以换行符`\n`结束：

- t 1234：返回系统启动后的时间，单位为毫秒
- kd RETURN/ku A：按下/松开按键，按键名称全部大写，使用AM中定义的按键名

这样的文本方式使得用户程序可以容易地解析事件的内容，Nanos-lite和Navy-app约定，上述事件抽象成文件`/dev/events`，它需要支持读操作，用户程序可以从中一次读出一个输入事件（由于时钟事件可以任意时刻进行读取，需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，否则用户程序将永远无法读到按键事件）。

为了将设备输入抽象成文件，首先需要实现`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），使得用户程序可以获取事件内容；

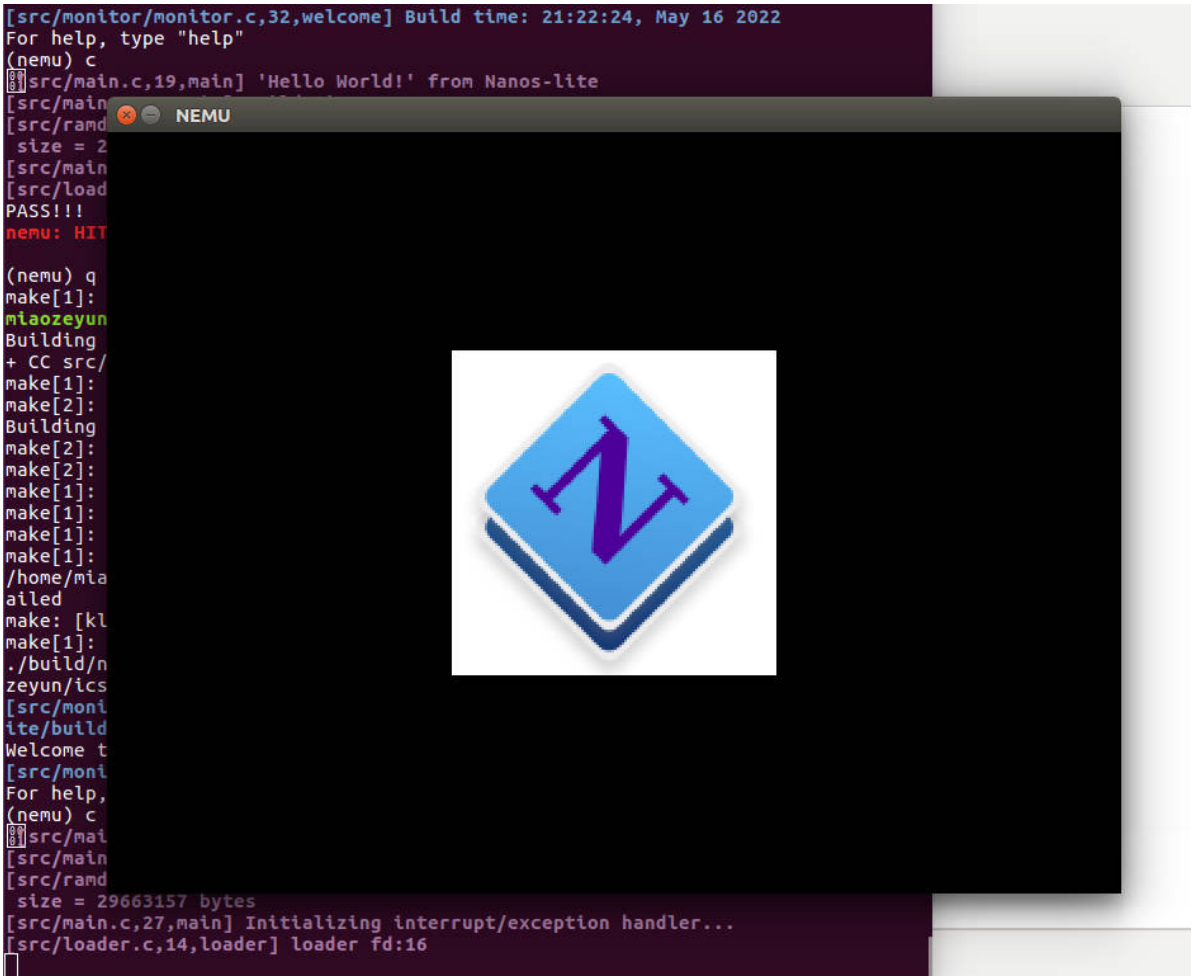
```
size_t events_read(void *buf, size_t len) {
  int key = _read_key();
  if(key == _KEY_NONE) //不存在按键事件的时候写入时钟事件
    sprintf(buf, "t %d\n", _uptime());
  else if ((uint32_t)key & 0x8000) //松开按键事件判断
    sprintf(buf, "ku %s\n", keyname[key & ~0x8000]); //写入松开按键事件
  else //按下按键事件
    sprintf(buf, "kd %s\n", keyname[key]); //写入按下按键事件
  return strlen(buf); //返回写入长度
}
```

最后在文件系统中添加对`/dev/events`的支持。

```
extern size_t events_read(void *buf, size_t len);

ssize_t fs_read(int fd, void *buf, size_t len)
{
  //Log("fs_read fd:%d,buf:%d,len:%d,size = %d", fd, buf, len,
file_table[fd].size);
  switch(fd)
  {
    case FD_STDIN:
    case FD_STDOUT:
    case FD_STDERR:
      return -1;
    case FD_EVENTS: //读取/dev/events文件
    {
      len = events_read((void *)buf, len); //调用events_read()函数读取
      break;
    }
    case FD_DISPINFO:
    {
      if(file_table[fd].open_offset + len >= fs_filesz(fd))
        len = fs_filesz(fd) - file_table[fd].open_offset;
      dispinfo_read((void *)buf, file_table[fd].open_offset, len);
      file_table[fd].open_offset += len;
      break;
    }
    default:
    {
      off_t offset = file_table[fd].disk_offset + file_table[fd].open_offset;
      if(file_table[fd].open_offset + len >= fs_filesz(fd))
        len = fs_filesz(fd) - file_table[fd].open_offset;
      ramdisk_read((void *)buf, offset, len);
      file_table[fd].open_offset += len;
      break;
    }
  }
  return len;
}
```

4.2 运行结果

成功运行`/bin/bmptest`程序，在屏幕上显示ProjectN的Logo。



成功运行`/bin/events`程序可以看到程序输出时间事件的信息并在敲击按键时会输出按键事件的信息。

```
miaozeyun@Ubuntu32:~/ics2017/nanos-lite$ make run
Building nanos-lite [x86-nemu]
make[1]: Entering directory '/home/miaozeyun/ics2017/nexus-am'
make[2]: Entering directory '/home/miaozeyun/ics2017/nexus-am/am'
Building am [x86-nemu]
make[2]: Nothing to be done for 'archive'.
make[2]: Leaving directory '/home/miaozeyun/ics2017/nexus-am/am'
make[1]: Leaving directory '/home/miaozeyun/ics2017/nexus-am'
make[1]: Entering directory '/home/miaozeyun/ics2017/nexus-am/libs/klib'
make[1]: *** 没有指明目标并且找不到 makefile。 停止。
make[1]: Leaving directory '/home/miaozeyun/ics2017/nexus-am/libs/klib'
/home/miaozeyun/ics2017/nexus-am/Makefile.compile:86: recipe for target 'klib' f
ailed
make: [klib] Error 2 (ignored)
make[1]: Entering directory '/home/miaozeyun/ics2017/nemu'
./build/nemu -l /home/miaozeyun/ics2017/nanos-lite/build/nemu-log.txt /home/miao
zeyun/ics2017/nanos-lite/build/nanos-lite-x86-nemu.bin
[src/monitor/monitor.c,67,load_img] The image is /home/miaozeyun/ics2017/nanos-l
ite/build/nanos-lite-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,32,welcome] Build time: 21:22:24, May 16 2022
For help, type "help"
(nemu) c
[src/main.c,19,main] 'Hello World!' from Nanos-lite
[src/main.c,20,main] Build time: 21:48:08, May 16 2022
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x102230, end = 0x1d4c1e5,
 size = 29663157 bytes
[src/main.c,27,main] Initializing interrupt/exception handler...
[src/loader.c,14,loader] loader fd:12
receive event: t 185
receive event: t 366
receive event: t 540
receive event: t 708
receive event: t 883
receive event: t 1068
receive event: t 1241
receive event: ku E
receive event: ku W
receive event: ku G
receive event: kd E
receive event: kd W
receive event: kd G
receive event: ku T
```

成功运行《仙剑奇侠传》——“新的故事”

4.3 Bug总结

本阶段中所有需要实现的部分和注意事项PA实验指导中都已经说明，并且将VGA显存和设备输入抽象成文件只是在文件系统中对三个特殊的文件做特殊的处理，因此如果阶段二中简易文件系统的实现正确的话，阶段三的实验基本上不会遇到什么很麻烦的Bug。

5 手册必答题

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。

《仙剑奇侠传》对存档的读取本质上就是读取文件，`navy-apps/apps/pal/src/global/global.c`的`PAL_LoadGame()`中通过`fread()`读取游戏存档，最后会调用`libos`库中的`_syscall`函数通过`int 0x80`的内联汇编语句进行系统调用，然后NEMU会执行该语句，NEMU会完成异常处理流程的硬件部分，并通过IDT（Interrupt Descriptor Table，中断描述符表）进入目标地址`vecsys`；而AM会压入`error_code`和`#irq`并调用`trap.s`中的`asm_trap`，`asm_trap`中通过PUSHA指令和PUSH ESP指令保存现场和trap frame（陷阱帧）的首地址以供后续函数使用，最终调用`irq_handle`；`irq_handle`将异常包装为事件并通过`do_event()`函数进行处理，Nanos_lite中`do_event()`函数通过异常号来区分不同异常，对于系统调用则通过`do_syscall()`函数处理，`do_syscall()`函数为不同的系统调用执

行不同的操作并将返回值写入EAX，这里因为是读文件操作所以会执行`fs_read()`；然后返回到`asm_trap`中通过POPA指令和IRET指令恢复现场，程序就可以继续执行了。

对屏幕的更新流程基本与上述内容相同，只是更新屏幕的操作申请的是SYS_write与SYS_lseek系统调用。

6 感悟与体会

1. PA3的PA实验流程中按顺序说明了所有需要实现的部分，并且几乎所有的注意事项和NEMU中的特殊处理也都进行了说明，配合`man`命令查询手册中对函数的详细描述，PA3的实验流程相较于PA2顺利了许多。
2. 由于PA中各个阶段的实验都是由简到难的实现某个功能，所以先前所实现的函数不能够确保在之后的实验中不会出现问题，即使是经过了PA框架代码中测试程序的测试也是如此，所以解决Bug的时候如果在刚实现的函数或功能中没有找到Bug，也不要忘了查看先前所写的代码中是否还隐藏着Bug。
3. PA3中除了指令实现的部分，其他部分因为是为Nanos-lite这一操作系统添加功能，所以Differential testing基本上已经没有办法帮助定位代码中的Bug，而且PA3中的报错很多都是**HIT BAD TRAP**，或者是更麻烦的内存越界，Bug定位和Debug都比PA2中困难一些。
4. 在运行《仙剑奇侠传》的时候一定要将先前Debug时在系统调用中添加的的Log注释掉，并关闭Debug和Differential testing，否则《仙剑奇侠传》会运行的非常慢，可能会误以为是某个部分的实现有Bug导致程序卡住，但其实只是运行的太慢还没有加载出来。