PA3

> 2013551 雷贺奥

1. 实验目的

①梳理操作系统概念

②学习系统调用，并实现中断机制

③了解文件系统的基本内容，实现简易文件系统

④实验最终实现支持文件操作的操作系统，要求能成功运行仙剑奇侠传

2. 实验内容

PA2 实验中主要涉及三大部分。

第一阶段，熟悉操作系统的基本概念、系统调用，实现中断机制。 操作系统的层次比AM要高，在第一阶段中需要完善loader、实现lidt指令、int指令、pusha指令、popa指令，最终成功调用系统调用的相关函数，成功运行dummy。

第二阶段，进一步完善系统调用，实现简易文件系统。让loader加载文件，实现fs_open、fs_read、fs_write、fs_close、fs_lseek，并且添加相应的系统调用，完善辅助函数，最终成功通过`bin\text`。

第三阶段，将输入输出抽象成文件，并运行仙剑奇侠传。

3. 实验过程

阶段一

3.1 准备阶段

一定需要修改两个Makefile.check文件，后续的很多环境错误可能都是由于未修改两个Makefile.check文件。

```
+++ nanos-lite/Makefile
@@ -34,2 +34,2 @@
-update: update-ramdisk-objcopy src/syscall.h
+update: update-ramdisk-fsimg src/syscall.h
```

3.2 实现loader

- 代码实现如下：

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

  需要使用`ramdisk.c`中使用的两个函数`ramdisk_read()`、`get_ramdisk_size()`，将整个ramdisk的大小读入DEFAULT_ENTRY，偏移设置为0对应文件第一个字节。

- 随后，在`navy-apps/tests/dummy/`下执行make命令，在将在`navy-apps/tests/dummy/build`文件夹 下生成 dummy-x86 可执行文件。
- 在`nanos-lite/`目录下执行 make update，将会生成 ramdisk 镜像文件，该镜像文件使用 objcopy 复制了 dummy-x86 可执行文件。
- 同时可以查看dummy的反汇编文件，使用`objdump -s dummy-x86 > dummy.txt`重定向到dummy.txt文件中。
- 运行结果如下：

  ```
  [src/monitor/monitor.c,65,load_img] The image is /home/lha/icslha/nanos-lite/build/nanos-lite-x86-nemu.bin
  Welcome to NEMU!
  [src/monitor/monitor.c,30,welcome] Build time: 00:29:47, Apr 20 2023
  For help, type "help"
  (nemu) c
  [src/main.c,19,main] 'Hello World!' from Nanos-lite
  [src/main.c,20,main] Build time: 18:58:49, May 15 2023
  [src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x102380, end = 0x1d4c299, size = 29663001 bytes
  [src/device.c,99,init_device] WIDTH:400
  HEIGHT:300
  ```

  可以成功加载dummy文件到操作系统中，接下来就需要对操作系统进行不断的完善。

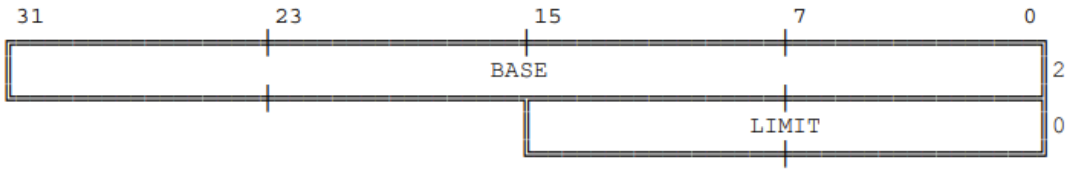
  同时通过报的错误`invalid opcode(eip=0x4001f98): cd 80 5b...`,可以通过反汇编代码定位到`int`指令没有实现,无法进行系统调用。

```
04001f88 <_syscall_>:
 4001f88:   55                      push   %ebp
 4001f89:   89 e5                   mov    %esp,%ebp
 4001f8b:   53                      push   %ebx
 4001f8c:   8b 55 14                mov    0x14(%ebp),%edx
 4001f8f:   8b 4d 10                mov    0x10(%ebp),%ecx
 4001f92:   8b 45 08                mov    0x8(%ebp),%eax
 4001f95:   8b 5d 0c                mov    0xc(%ebp),%ebx
 4001f98:   cd 80                   int    $0x80
 4001f9a:   5b                      pop    %ebx
 4001f9b:   5d                      pop    %ebp
 4001f9c:   c3                      ret
 4001f9d:   66 90                   xchg   %ax,%ax
 4001f9f:   90                      nop
```

3.3 准备IDT

数组IDT是中断描述符表，为了在内存中找到IDT的地址，我们需要新设计一个IDTR寄存器，用于存储IDT的首地址和长度，随后还需要实现lidt在IDTR寄存器中设置好IDT的首地址和长度，中断处理机制就可以正常使用了。

- IDTR寄存器的结构



32bit base即为起始地址，16bit limit 即为IDT长度。在reg.h中设计idtr寄存器。

```
//IDTR寄存器
  struct IDTR_register
  {
    uint32_t base;//基地址
    uint16_t limit;//offset
  }idtr;
```

- 实现lidt指令

lidt_a编码声明

```
make_DHelper(lidt_a);
```

lidt_a编码定义

```
//pa3 add
make_DHelper(lidt_a)
{
    decode_op_a(eip,id_dest,true);
}
```

lidt解码声明

```
make_EHelper(lidt);
```

lidt解码定义

```
make_EHelper(lidt) {
  //TODO();
  //eax 32位操作数，base 32 limit 16
  t1=id_dest->val;
  rtl_lm(&t0,&t1,2);
  cpu.idtr.limit=t0;
  //Log("cpu.idtr.limit=%x",cpu.idtr.limit);
  t2=id_dest->val+2;
  rtl_lm(&t0,&t2,4);
  cpu.idtr.base=t0;
  //Log("cpu.idtr.base=%x",cpu.idtr.base);
  print_asm_template1(lidt);
}
```

由上图可知，将6btye中0-15赋值给IDTR寄存器的limit，将6btye中的16-47赋值给IDTR寄存器的base。

- 在opcode_tabel中注册，这里不再赘述。

  make run，invalid opcode 的 eip 前移，int指令此时还并没有实现。

3.4 int指令触发异常

cs寄存器即代码段寄存器，虽然pa实验中并没有分段机制，但是我们还是在reg.h中加入一个cs寄存器，将其初始化为8。

- cs寄存器的注册和初始化

  cs寄存器本省为16位，由于后续实验中需要使用push、pop指令压栈出栈，并且pa中的push和pop均支持32位寄存器，所以暂时将其声明为32位，只用其低16位

```
//cs寄存器16位
  rtlreg_t cs;//因为使用push指令暂时写为32bit
```

初始化`nemu/src/monitor/monitor.c`

```
static inline void restart() {
  /* Set the initial instruction pointer. */
  cpu.eip = ENTRY_START;
  //进行eflags的初始化，0x0000 0002H
  unsigned int origin =2;
  memcpy(&cpu.eflags,&origin,sizeof(cpu.eflags));
  //cs 初始化为8
  cpu.cs=8;
#ifdef DIFF_TEST
  init_qemu_reg();
#endif
}
```

- int 指令的实现

  `nemu/src/cpu/intr.c` 中实现 `raise_intr` 函数，helper函数中需要调用

```
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.
   */
  //TODO();
  memcpy(&t1,&cpu.eflags,sizeof(cpu.eflags));
  rtl_li(&t0,t1);//赋值给t0
  rtl_push(&t0);//eflags
  rtl_push(&cpu.cs);//cs
  rtl_li(&t0,ret_addr);//返回地址
  rtl_push(&t0);//eip
  //门描述符地址
  vaddr_t read_begin=cpu.idtr.base+NO*sizeof(GateDesc);
  //Log("%x",cpu.idtr.base);
  //读取
  uint32_t offset_0to15 = vaddr_read(read_begin,2);
  //Log("%x",offset_0to15);
  //16-32 btye 7 8
  uint32_t offset_16to32 =vaddr_read(read_begin+sizeof(GateDesc)-2,2);
  //Log("%x",offset_16to32);
  //跳转地址
  uint32_t target_addr=(offset_16to32<<16)+offset_0to15;
  decoding.is_jmp=1;
  decoding.jmp_eip=target_addr;
  //Log("target_addr %x",target_addr);
}
```

（1）将EFLAGS、cs、返回地址、EIP压入堆栈，保存好现在的寄存器状态，（2）从IDRT中读出IDT的首地址，（3）根据NO在IDT中进行索引，找到一个门描述符，（4）将门描述符中的offset域组合成目标地址，（4）is_jmp=1，对目标地址进行跳转。

- Make_EHelper(int)

  在`nemu/src/cpu/exec/system.c`实现执行函数

```
make_EHelper(int) {
  //TODO();idest = imm
  uint8_t NO= (uint8_t) id_dest->val;
  //Log("int NO=%d, seq_eip=0x%x",NO,decoding.seq_eip);
  raise_intr(NO,decoding.seq_eip);//eip下条指令
  print_asm("int %s ", id_dest->str);

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

  调用`raise_intr`函数，保存好现在的寄存器状态。

- opcode_table中进行注册

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

- make run结果，对比反汇编代码发现

  `eip=0x100ad6时遇到 invalid opcode`，此时的指令为pusha，我们并没有实现。

3.5 保存现场

i386提供了 pusha 指令,用于把通用寄存器的值压入堆栈，vecsys()会压入错误码和异常号#irq，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。

```
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
```

- pusha指令

  在 nemu/src/cpu/exec/data-mov.c 中更改执行函数:

```
make_EHelper(pusha) {
  //TODO();
  //Push EAX, ECX, EDX, EBX, original ESP, EBP, ESI, and EDI
  t1=cpu.esp;
  rtl_push(&cpu.eax);
  rtl_push(&cpu.ecx);
  rtl_push(&cpu.edx);
  rtl_push(&cpu.ebx);
  rtl_push(&t1);
  rtl_push(&cpu.ebp);
  rtl_push(&cpu.esi);
  rtl_push(&cpu.edi);
  print_asm("pusha");
}
```

  opcode_table

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

- 需要重新组织 TrapFrame

```
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 i r q ;
 uintptr_t error_code ;
 uintptr_t eip ;
 uintptr_t cs ;
 uintptr_t e f l a g s ;
};
```

  完成这个阶段后，可以看见HIT BAD TRAP 提示。

3.6 事件分发

首先我们在 do_event() 函数中识别出系统调用事件 _EVENT_SYSCALL ， 然后调用 do_syscall()。

```
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;
}
```

需要我们，在接下来的实验中不断补全代码。

3.7 系统调用处理

背景知识，根据异常请求号设置事件，随后调用事件处理函数。

实现正确的 SYSCALL_ARGx() 宏，让它们从作为参数的现场 reg 中获得正确的系统调用参数寄 存器 (*syscall*() 函数以及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
```

- SYS_none函数中添加系统调用，同时设置系统调用的返回值

```
_RegSet* do_syscall(_RegSet *r) {
  uintptr_t a[4];
  a[0] = SYSCALL_ARG1(r);//eax
  a[1] =SYSCALL_ARG2(r);//ebx
  a[2]=SYSCALL_ARG3(r);//ecx
  a[3]=SYSCALL_ARG4(r);//edx
  switch (a[0]) {
    case SYS_none:
        SYSCALL_ARG1(r)=1;//do noting
        break;
    case SYS_exit:
        _halt(a[1]);
        break;
    case SYS_write:
        //a[1]= fd a[2]=buf a[3]=len
        SYSCALL_ARG1(r)=mysys_write(a[1],(void*)a[2],a[3]);
        break;
    case SYS_brk:
        SYSCALL_ARG1(r)=mysys_brk(a[1]);
        break;
    case SYS_read:
        SYSCALL_ARG1(r)=fs_read(a[1],(void*)a[2],a[3]);
        break;
    case SYS_open:
        SYSCALL_ARG1(r)=fs_open((char*)a[1],0,0);
        break;
    case SYS_close:
        SYSCALL_ARG1(r)=fs_close(a[1]);
        break;
```

```
    case SYS_lseek:
        SYSCALL_ARG1(r)=fs_lseek(a[1],a[2],a[3]);
        break;
    default: panic("Unhandled syscall ID = %d", a[0]);
  }

  return NULL;
}
```

ps：之后就不再粘贴代码了

- 实现上述代码后，再运行make run，最终会发现卡在了popa指令。

3.8 恢复现场

popa 指令将所有寄存器以一定顺序弹出栈。

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

代码实现：

- popa执行函数

```
make_EHelper(popa) {
  //TODO();
  rtl_pop(&cpu.edi);
  rtl_pop(&cpu.esi);
  rtl_pop(&cpu.ebp);
  rtl_pop(&t1);//skip esp
  rtl_pop(&cpu.ebx);
  rtl_pop(&cpu.edx);
  rtl_pop(&cpu.ecx);
  rtl_pop(&cpu.eax);
  print_asm("popa");
}
```

- opcode_table

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

- iret 指令

  还需实现 iret 指令对现场进行恢复，其主要作用是从异常处理代码中返回，将栈顶的三个元素来依次 解释成 EIP、CS、EFLAGS，并恢复。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。

  执行函数：

```
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" ) ;
}
```

- opcode_table

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

此时已经完成了阶段一，出现 HIT GOOD TRAP at eip = 0x00100032 字样提示。

```
[src/main.c,19,main] 'Hello World!' from Nanos-lite
[src/main.c,20,main] Build time: 18:58:49, May 15 2023
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x102380, end = 0x1d4c299, size = 29663001 by
[src/device.c,99,init_device] WIDTH:400
HEIGHT:300

[src/main.c,27,main] Initializing interrupt/exception handler...
[src/fs.c,37,init_fs] Init_fs   fd=FD_FB: screan size is 480000
nemu: HIT GOOD TRAP at eip = 0x00100032
```

阶段二

首先需要把程序换成helloword，随后再进行实现。

> 阶段二添加标准输出 和堆区管理功能，并实现一个简易的文件系统。系统调用：（nanos-lite/src/syscall.c），辅助函 数：（navy-apps/libs/libos/src/nanos.c)

4.1 标准输出

首先实现一个辅助函数 _write，其通过调用 _syscall 函数返回 eax 寄存器的值。

运行`man 2 write`，即可查看系统调用，如下图所示：

```
SYNOPSIS
       #include <unistd.h>

       ssize_t write(int fd, const void *buf, size_t count);
```

write()将 buf 中最多 count 个字节写入到文件描述符 fd 对应 的文件中。

PS: 下面要实现的系统调用都用这种方法，进行查阅和理解，下面就不再粘贴具体系统调用的描述了。

- _write()辅助函数

  在 navy-apps/libs/libos/src/nanos.c 中进行修改

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

- do_syscall 函数中加入 SYS_write 系统调用

  在实现SYS_none时已经全部粘贴，这里不再赘述。

- 实现 sys_write 系统调用

```
//ssize_t write(int fd, const void *buf, size_t count);
int mysys_write(int fd,void* buf,size_t len)
{
    //stdout 或 stderr
    if(fd ==1 || fd ==2)
    {
        //Log("buffer:%s",(char*)buf);
        for(int i=0;i<len;i++)
        {
            char temp;//用于传参给串口
            //需要使用char* 强制类型转换
            temp=*(char*)(buf+i);
            _putc(temp);
        }
        return len;
    }
    return -1;
}
```

  使用_putc()串口,将其输出在屏幕上即可。

- 运行结果如下：

```
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 00:29:47, Apr 20 2023
For help, type "help"
(nemu) c
[src/main.c,19,main] 'Hello World!' from Nanos-lite
[src/main.c,20,main] Build time: 21:06:18, May 15 2023
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x102380, end = 0x1d4c299, size = 29663001 bytes
[src/device.c,99,init_device] WIDTH:400
HEIGHT:300

[src/main.c,27,main] Initializing interrupt/exception handler...
[src/fs.c,37,init_fs] Init_fs   fd=FD_FB: screan size is 480000
Hello World!
```

4.2 堆区管理

在 Nanos-lite 的 sys_write 中通过 Log 观察 write 系统调用的情况，发现用户程序使用 printf 输出时，是逐个字节调用 write 来实现的。事实上，用户程序在第一次调用 printf 时，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，就会逐个字符输出。

SYS_brk系统调用

- mysys_brk ()

```
//sbrk(intptr_t increment)
int mysys_brk(int addr)
{
    //单任务总是返回0,
    return 0;
}
```

理论上接收一个参数increment用于指示新的program break的位置，pa3让 SYS_brk 总是返回 0，表示堆区大小的调整总是成功。

- do_syscall()

```
case SYS_brk:
        SYSCALL_ARG1(r)=mysys_brk(a[1]);
        break;
```

用户层实现_sbrk

program break 一开始的位置位于程序的数据段结束的位置（由 _end 标志），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的 program break，若成功返回 0，更新 program break 的位 置，并将旧 program break 的位置作为 _sbrk 的返回值返回；若失败 _sbrk 返回-1。

```
void *_sbrk(intptr_t increment){
  extern end;//阅读man 3 end 得知使用extern 调用
  //记录上一个program break,并且只初始化一次
  static uintptr_t last_break=(uintptr_t)&end;
  //更新 break
  uintptr_t pro_break=last_break+increment;
  //系统调用
  if(_syscall_(SYS_brk,pro_break,0,0)==0)
  {
    uintptr_t temp =last_break;
    last_break = pro_break;
    return (void*) temp;
  }
  else{
    return (void *)-1;
  }
}
```

通过 man 3 end，可以得知使用extern 调用 end，并且用end来初始化last_break，静态变量保证其只被初始化一次。

- 运行结果

```
Hello World for the 3270th time
[src/syscall.c,51,mysys_write] buffer:Hello World for the 3271th time

Hello World for the 3271th time
[src/syscall.c,51,mysys_write] buffer:Hello World for the 3272th time
```

可以发现，log中的hello world 不再是以字符的形式打印出来的，而是以字符串的形式打印出来，由此得出，堆区管理实现成功。

4.3 让loader使用文件

首先，需要对nano-lite 的 Makefile 进行修改。

```
update : update-ramdisk-fsimg src / syscall.h
```

再修改file_table的结构体。

```
typedef struct {
  char *name;
  size_t size;
  off_t disk_offset;
  off_t open_offset;    //文件被打开之后的读写指针
} Finfo;
```

- fs_open 根据文件名查找文件描述符 fd。

```
//fs_open
int fs_open(const char* pathname,int flag,int mode)
{
    //忽略flags、mode
    for(int i=0;i<NR_FILES;i++)
    {
        if(strcmp(pathname,file_table[i].name)==0)
        {
            return i;
        }
    }
    panic("%s is not in file_table",pathname);
    return -1;
}
```

- fs_read 读取 fd 对应的文件。

```
extern void dispinfo_read(void *buf, off_t offset, size_t len);
extern size_t events_read(void *buf, size_t len) ;
//fs_read
ssize_t fs_read(int fd,void* buf,size_t len)
{
    //Log("Debug: cd fs_read");
    assert(fd>0 && fd<NR_FILES);
    if(fd<3)
    {
        Log("fd < 3 when use fs_read/n");
        return 0;
    }
    //注意偏移量不要越过 文件的边界
    int maxbyte=file_table[fd].size-file_table[fd].open_offset;
    if(len>maxbyte)
    {
        //FD_DISPINFO 调用dispinfo_read;
        if(fd==FD_DISPINFO)
        {
            //Log("len>maxbyte fd==FD_DISPINFO");
            dispinfo_read(buf,file_table[fd].open_offset,maxbyte);
```

```
        }
        else if(fd==FD_EVENTS)
        {
            //不需要考虑大小和偏移
            return events_read(buf,len);
        }
        else//其他文件
        {
            ramdisk_read(buf,
                file_table[fd].disk_offset+file_table[fd].open_offset,
                maxbyte);
        }
        //更新偏移量
        file_table[fd].open_offset+=maxbyte;
        return maxbyte;
    }
    else
    {
        if(fd==FD_DISPINFO)
        {
            //Log("len< maxbyte fd==FD_DISPINFO");
            dispinfo_read(buf,file_table[fd].open_offset,len);
        }
        else if(fd==FD_EVENTS)
        {
            return events_read(buf,len);
        }
        else
        {
             ramdisk_read(buf,
             file_table[fd].disk_offset+file_table[fd].open_offset,
             len);
        }
        //更新偏移量
        file_table[fd].open_offset+=len;
        return len;
    }
}
```

代码包括了后面阶段的代码，这里只讲解此阶段的部分代码。

（1）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，所以fd<3,直接return 0。

（2）使用 ramdisk_read()和 ramdisk_write()来进行文件的真正读写。

（3）同时需要注意偏移量不要越过文件的边界，所以len>maxbyte时，需要特殊处理，fs_read最多只能读maxbyte个字节。

- fs_close 关闭文件

```
//fs_close
int fs_close(int fd)
{
    //fs_close()可以直接返回 0,表示总是关闭成功.
    assert(fd>0 && fd<NR_FILES);
    return 0;
}
```

- 重新实现 loader.c 中的 loader 函数。更改 nanos-lite/src/main.c 中的用户程序，让其加载 text 文件。

```
uintptr_t loader(_Protect *as, const char *filename) {
  //TODO();
//  ramdisk_read(DEFAULT_ENTRY,0,get_ramdisk_size());
//  return (uintptr_t)DEFAULT_ENTRY;
    int fd = fs_open(filename,0,0);
    fs_read(fd,DEFAULT_ENTRY,getfile_size(fd));
    fs_close(fd);
    return (uintptr_t)DEFAULT_ENTRY;
}
```

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

4.4实现完整的文件系统

- fs_write 对 fd 对应的文件进行写操作和实现。

```
//fs_write
ssize_t fs_write(int fd,void *buf, size_t len)
{
    //Log("Debug: cd fs_write");
    assert(fd>0 && fd<NR_FILES);
    if(fd<3)
    {
        Log("fd < 3 when use fs_write/n");
        return 0;
    }
    int maxbyte=file_table[fd].size-file_table[fd].open_offset;
    if(len>maxbyte)
    {
        if(fd==FD_FB)//VGA
        {
            fb_write(buf,file_table[fd].open_offset,maxbyte);
        }
        else
        {
            ramdisk_write(buf,

 file_table[fd].disk_offset+file_table[fd].open_offset,
                          maxbyte);
        }
        //更新偏移量
        file_table[fd].open_offset+=maxbyte;
        return maxbyte;
    }
    else
    {
        if(fd==FD_FB)//VGA
        {
            fb_write(buf,file_table[fd].open_offset,len);
        }
        else
        {
            ramdisk_write(buf,
```

```
 file_table[fd].disk_offset+file_table[fd].open_offset,
                                len);
        }
        //更新偏移量
        file_table[fd].open_offset+=len;
        return len;
    }
}
```

代码包括了后面阶段的代码，这里只讲解此阶段的部分代码。

（1）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，所以fd<3,直接return 0。

（2）使用 ramdisk_read()和 ramdisk_write()来进行文件的真正读写。

（3）同时需要注意偏移量不要越过文件的边界，所以len>maxbyte时，需要特殊处理，fs_write最多只能读maxbyte个字节。

- fs_seek 修改 fd 对应文件的 open_offset。

  使用**man 2 lseek**，查出SEEK_SET、SEEK_CUR、SEEK_END对应的offset的设置。

```
//fs_lseek man 2 lseek
off_t fs_lseek(int fd,off_t offset,int whence)
{
    if(whence==SEEK_SET)
    {
        file_table[fd].open_offset=offset;
        return file_table[fd].open_offset;
    }
    else if(whence==SEEK_CUR)
    {
        file_table[fd].open_offset+=offset;
        return file_table[fd].open_offset;
    }
    else if(whence==SEEK_END)
    {
        file_table[fd].open_offset=getfile_size(fd)+offset;
        return file_table[fd].open_offset;
    }
    else{
        panic("whence=%d is not declared",whence);
        return 0;
    }
}
```

实现相关的系统调用

- 引入fs.c中的相关函数#include "fs.h"，即可调用文件系统所需要中断的所有函数
- 加入sys_open、sys_read、sys_close、sys_lseek

```
 case SYS_write:
        //a[1]= fd a[2]=buf a[3]=len
        SYSCALL_ARG1(r)=mysys_write(a[1],(void*)a[2],a[3]);
        break;
    case SYS_read:
        SYSCALL_ARG1(r)=fs_read(a[1],(void*)a[2],a[3]);
```

```
        break;
    case SYS_open:
        SYSCALL_ARG1(r)=fs_open((char*)a[1],0,0);
        break;
    case SYS_close:
        SYSCALL_ARG1(r)=fs_close(a[1]);
        break;
    case SYS_lseek:
        SYSCALL_ARG1(r)=fs_lseek(a[1],a[2],a[3]);
        break;
```

- 修改mysys_write()

```
//ssize_t write(int fd, const void *buf, size_t count);
int mysys_write(int fd,void* buf,size_t len)
{
    //stdout 或 stderr
    if(fd ==1 || fd ==2)
    {
        Log("buffer:%s",(char*)buf);
        for(int i=0;i<len;i++)
        {
            char temp;//用于传参给串口
            //需要使用char* 强制类型转换
            temp=*(char*)(buf+i);
            _putc(temp);
        }
        return len;
    }
    else if(fd>=3)
    {
        return fs_write(fd,buf,len);
    }
    else
    {
        panic("fd = %d is not available\n",fd);
    }
    return -1;
}
```

(1) `fd==1||fd==2` 文件对应stdout 或 stderr，只需要将其连接到串口，将字符输出即可。

(2) `fd>=3` 涉及文件真正的读写，需要调用fs_write()

修改 nano.c中的辅助函数

- 代码部分

```
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);
}
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,fd,(uintptr_t)buf,count);
}
```

- `bin\text` 测试

```
[src/monitor/monitor.c,65,load_img] The image is /home/lha/icslha/nanos-lite/build/nanos-lite-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 00:29:47, Apr 20 2023
For help, type "help"
(nemu) c
[src/main.c,19,main] 'Hello World!' from Nanos-lite
[src/main.c,20,main] Build time: 21:53:46, May 15 2023
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x102380, end = 0x1d4c299, size = 29663001 bytes
[src/device.c,99,init_device] WIDTH:400
HEIGHT:300

[src/main.c,27,main] Initializing interrupt/exception handler...
[src/fs.c,37,init_fs] Init_fs   fd=FD_FB: screan size is 480000
PASS!!!
nemu: HIT GOOD TRAP at eip = 0x00100032
```

阶段三

5.1 把VGA显存抽象为文件

- 在 `nexus-am/am/arch/x86-nemu/src/ioe.c` 文件，为使 nanos-lite 获取 AM 的屏幕信息，在 IOE 中添加的接口。

```
//API 获取屏幕大小
void getscreen(int* width,int* height)
{
    *width=_screen.width;
    *height=_screen.height;
}
```

- init_fs:函数

  修改`nanos-lite/src/fs.c 文件`，文件是字节序列，而 VGA 的一个像素占 32bit(4Byte)，对于 VGA，每次操作以 uint32_t 为基本单位。

```
void init_fs() {
  // TODO: initialize the size of /dev/fb
  int width=0,height=0;
  getscreen(&width,&height);
  file_table[FD_FB].size=width*height*sizeof(uint32_t);//一个像素4b
  Log("Init_fs\tfd=FD_FB: screan size is %d",file_table[FD_FB].size);
}
```

- fb_write函数绘制屏幕上的像素点。改`nanos-lite/src/device.c`

```
//am ioe.c 获取屏幕大小
extern void getscreen(int* width,int* height);
void fb_write(const void *buf, off_t offset, size_t len) {
    //Log("cd fb_write");
    if(offset%4!=0||len%4!=0)
    {
        panic("VGA 1像素 should be 4 byte");
    }
    //屏幕上offset处
    int width=0,height=0;
    getscreen(&width,&height);
    int index = offset/sizeof(uint32_t);//第多少个像素
    //x1 ,y1开始的位置
    int screen_y1 = index/width;
    int screen_x1 = index%width;
    //x2,y2结束的位置
    index=(offset+len)/sizeof(uint32_t);
    int screen_y2=index/width;
    int screen_x2 = index%width;

    if(screen_y2==screen_y1)
    {
        _draw_rect(buf,screen_x1,screen_y1,screen_x2-screen_x1,1);
        return;
    }
    else if(screen_y2-screen_y1==1)
    {
        _draw_rect(buf,screen_x1,screen_y1,width-screen_x1,1);
        _draw_rect(buf+4*(width-screen_x1),0,screen_y2,screen_x2,1);
        return;
    }
    else if(screen_y2-screen_y1>1)
    {
        _draw_rect(buf,screen_x1,screen_y1,width-screen_x1,1);
        int draw_index=width-screen_x1;
        _draw_rect(buf+draw_index*4,0,screen_y1+1,width,screen_y2-screen_y1-
2);
        draw_index += width*(screen_y2-screen_y1-2);
        _draw_rect(buf+draw_index*4,0,screen_y2,screen_x2,1);
        return;
    }
    panic("screen_y2<screen_y1! erro");
}
```

（1）需要接口函数`get_screen()`获取屏幕大小。

（2）像素绘制需要分为三种情况，如代码所示，而不是一个像素一个像素的绘制，否则绘制速度过慢。

- init_device函数将长宽信息按格式写入 dispinfo。

  修改`nanos-lite/src/device.c`文件，这里将 WIDTH 与 HEIGHT 信息按照实验指导书格式 写入 dispinfo。

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

  // TODO: print the string to array `dispinfo` with the format
  // described in the Navy-apps convention
  int width=0,height=0;
  getscreen(&width,&height);
  Log("WIDTH:%d\nHEIGHT:%d\n",width,height);
  sprintf(dispinfo,"WIDTH:%d\nHEIGHT:%d\n",width,height);
}
```

- dispinfo_read:函数

  修改 `nanos-lite/src/device.c` 文件

```
void dispinfo_read(void *buf, off_t offset, size_t len) {
    //用于把字符串 dispinfo 中 offset 开始 的 len 字节写到 buf 中
    Log("dispinfo_read: len=%d",len);
    strncpy(buf,dispinfo+offset,len);
}
```

- fs_read:函数

  修改 nanos-lite/src/fs.c 文件。

```
extern void dispinfo_read(void *buf, off_t offset, size_t len);
extern size_t events_read(void *buf, size_t len) ;
//fs_read
ssize_t fs_read(int fd,void* buf,size_t len)
{
    //Log("Debug: cd fs_read");
    assert(fd>0 && fd<NR_FILES);
    if(fd<3)
    {
        Log("fd < 3 when use fs_read/n");
        return 0;
    }
    //注意偏移量不要越过 文件的边界
    int maxbyte=file_table[fd].size-file_table[fd].open_offset;
    if(len>maxbyte)
    {
        //FD_DISPINFO 调用dispinfo_read;
        if(fd==FD_DISPINFO)
        {
            //Log("len>maxbyte fd==FD_DISPINFO");
            dispinfo_read(buf,file_table[fd].open_offset,maxbyte);
        }
        else if(fd==FD_EVENTS)
        {
            //不需要考虑大小和偏移
            return events_read(buf,len);
        }
        else//其他文件
        {
            ramdisk_read(buf,
                file_table[fd].disk_offset+file_table[fd].open_offset,
                maxbyte);
```

```
        }
        //更新偏移量
        file_table[fd].open_offset+=maxbyte;
        return maxbyte;
    }
    else
    {
        if(fd==FD_DISPINFO)
        {
            //Log("len< maxbyte fd==FD_DISPINFO");
            dispinfo_read(buf,file_table[fd].open_offset,len);
        }
        else if(fd==FD_EVENTS)
        {
            return events_read(buf,len);
        }
        else
        {
             ramdisk_read(buf,
             file_table[fd].disk_offset+file_table[fd].open_offset,
             len);
        }
        //更新偏移量
        file_table[fd].open_offset+=len;
        return len;
    }
}
```

注意，此时由于/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，**需要考虑文件边界，**。

- fs_write：函数

  修改 nanos-lite/src/fs.c 文件

```
//fs_write
ssize_t fs_write(int fd,void *buf, size_t len)
{
    //Log("Debug: cd fs_write");
    assert(fd>0 && fd<NR_FILES);
    if(fd<3)
    {
        Log("fd < 3 when use fs_write/n");
        return 0;
    }
    int maxbyte=file_table[fd].size-file_table[fd].open_offset;
    if(len>maxbyte)
    {
        if(fd==FD_FB)//VGA
        {
            fb_write(buf,file_table[fd].open_offset,maxbyte);
        }
        else
        {
            ramdisk_write(buf,

 file_table[fd].disk_offset+file_table[fd].open_offset,
                          maxbyte);
```

```
        }
        //更新偏移量
        file_table[fd].open_offset+=maxbyte;
        return maxbyte;
    }
    else
    {
        if(fd==FD_FB)//VGA
        {
            fb_write(buf,file_table[fd].open_offset,len);
        }
        else
        {
            ramdisk_write(buf,

 file_table[fd].disk_offset+file_table[fd].open_offset,
                           len);
        }
        //更新偏移量
        file_table[fd].open_offset+=len;
        return len;
    }
}
```

需要考虑文件边界。

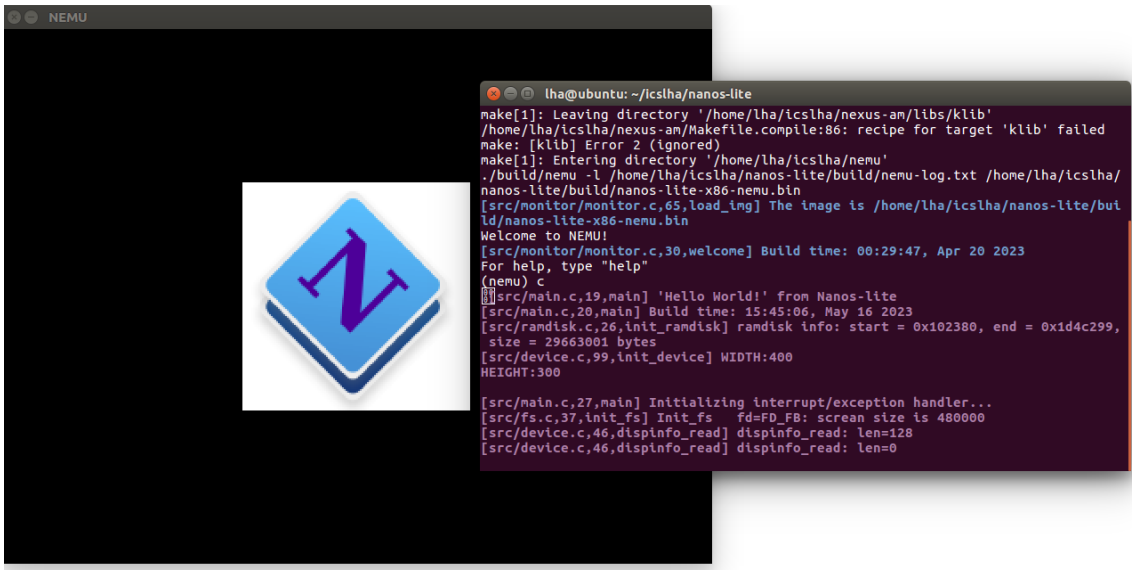
- 运行结果

  在开始运行之前，需要进行一些准备工作 。

  首先修改 nano-lite/Makefile 文件：`OBJCOPY_FILE = $(NAVY_HOME)/tests/text/build/text-x86` 。

  接着修改`nano-lite/src/main.c`文件：`uint32_t entry = loader(NULL, "/bin/bmptest");`

  接着在`navy-apps/tests/text/`下执行make命令。



5.2 将设备输入抽象成文件

输入设备有键盘和时钟，这里将二者封装成事件，使之以文本形式表现出来。

- events_read

  使用read_key()与uptime()读取按键事件与时钟事件，优先读取按键事件。设定函数每次读取一个事件。

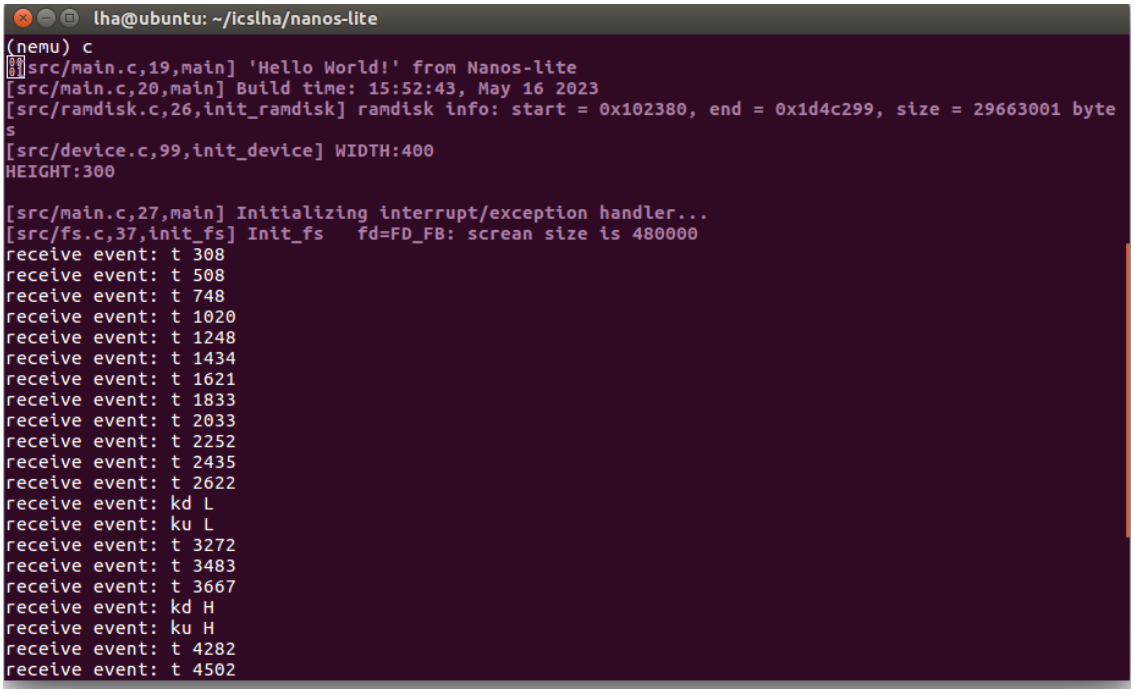
```
extern int _read_key();//key
extern unsigned long _uptime();//time
size_t events_read(void *buf, size_t len) {
  //return 0;
  int key=_read_key();
  bool down = false;
  if (key & 0x8000) {
        key ^= 0x8000;
        down = true;
  }
  char temp[20];
  if (key != _KEY_NONE)
  {
    if(down){
        sprintf(temp,"kd %s\n",keyname[key]);}
    else{
        sprintf(temp,"ku %s\n",keyname[key]);}
  }
  else{
    sprintf(temp,"t %d\n",_uptime());}

  if(strlen(temp)<=len)
  {
    strncpy((char*)buf,temp,len);
    return strlen(temp);
  }
  Log("strlen(event)=%d>len=%d",strlen(temp),len);
  return 0;
}
```

- fs_read函数

  对于/dev/events，file_table 中存储的并不是该输入文件的实体，**无需考虑文件的实际大小和偏移量**，函数的返回值即为 events_read()的返回。

```
else if(fd==FD_EVENTS){
    return events_read(buf,len);
}
```

- 运行结果



运行仙剑奇侠传

cwtl指令在之前运行马里奥时就已经实现，这里简要粘贴一下。

CBW/CWDE — Convert Byte to Word/Convert Word to Doubleword

Opcode	Instruction	Clocks	Description
98	CBW	3	AX ← sign-extend of AL
98	CWDE	3	EAX ← sign-extend of AX

- 执行函数

```
make_EHelper(cwtl) {

  if (decoding.is_operand_size_16) {
    //TODO();
    rtl_lr_b(&t0, R_AX);
    rtl_sext(&t0, &t0, 1);
    rtl_sr_w(R_AX, &t0);
  }
  else {
    //TODO();
    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");
}
```


NEMU
仙劍奇俠傳


NEMU
李大娘：
李逍遙！你皮癢啊？
敢說老娘是什麼鬼婆！

Bug总结

- FD_EVENTS是特殊的文件，**不需要考虑文件边界，即函数中直接输入参数len。**如果考虑文件边界，由于file_table中将其的大小定义为0，会导致读、写的长度一直为0，读写失败。当时调了半天！！！
- 一定要修改Makefile.check文件，很多环境问题全都是没改Makefile文件
- pal资源包的问题，最后使用群里已经编译好的资源包，成功运行仙剑。

手册必答题

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.

温馨提示

PA3 到此结束.

读取存档

对于存档的读取，在 global.c 中的读取存档函数 PAL_LoadGame 调用了 fread 标准 C 语言库函 数。

```
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.

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

fread库函数调用libc中的_read()函数，随后调用libos中的**syscall** 函数

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

*syscall*函数通过 int 0x80 的内联汇编语句。如下所示，通过pa3，我们不难得知int指令就是用来执行系统调用。

```
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;
}
```

所后，nemu指令系统执行int指令，完成异常的硬件处理后，根据IDT中的门描述符，切换到AM的目标地址，在AM中保存现场（即各个寄存器的压栈），同时将异常封装为事件，调用nanos-lite中的异常处理函数，执行do_syscall(),寻找到相应异常的处理办法，此时的处理办法为fs_read(a[1],(void*)a[2],a[3])。

```
case SYS_read:
        SYSCALL_ARG1(r)=fs_read(a[1],(void*)a[2],a[3]);
        break;
```

最后,在文件系统中根据不同的 fd，执行不同的处理函数。

```
extern void dispinfo_read(void *buf, off_t offset, size_t len);
extern size_t events_read(void *buf, size_t len) ;
//fs_read
ssize_t fs_read(int fd,void* buf,size_t len)
{
    //Log("Debug: cd fs_read");
    assert(fd>0 && fd<NR_FILES);
    if(fd<3)
    {
        Log("fd < 3 when use fs_read/n");
        return 0;
    }
    //注意偏移量不要越过 文件的边界
    int maxbyte=file_table[fd].size-file_table[fd].open_offset;
    if(len>maxbyte)
    {
        //FD_DISPINFO 调用dispinfo_read;
        if(fd==FD_DISPINFO)
        {
            //Log("len>maxbyte fd==FD_DISPINFO");
            dispinfo_read(buf,file_table[fd].open_offset,maxbyte);
        }
        else if(fd==FD_EVENTS)
        {
            //不需要考虑大小和偏移
            return events_read(buf,len);
        }
        else//其他文件
        {
            ramdisk_read(buf,
```

```
                file_table[fd].disk_offset+file_table[fd].open_offset,
                maxbyte);
        }
        //更新偏移量
        file_table[fd].open_offset+=maxbyte;
        return maxbyte;
    }
    else
    {
        if(fd==FD_DISPINFO)
        {
            //Log("len< maxbyte fd==FD_DISPINFO");
            dispinfo_read(buf,file_table[fd].open_offset,len);
        }
        else if(fd==FD_EVENTS)
        {
            return events_read(buf,len);
        }
        else
        {
             ramdisk_read(buf,
             file_table[fd].disk_offset+file_table[fd].open_offset,
             len);
        }
        //更新偏移量
        file_table[fd].open_offset+=len;
        return len;
    }
}
```

更新屏幕

对于屏幕的更新，在 hal.c 中调用 NDL_DrawRect 和 NDL_Render 函数进行像素节点的设置和 图像绘制，其中执行了 fwrite、fflush、fseek 等标准 C 语言库函数，下面以 fwrite 为例继续说明关系。

```
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();
}
```

fwrite 库函数调用 libc 中的 _write 函数，最终调用 libos 中的 _syscall__ 函数，该函数通过 int 0x80 的内联汇编语句进行系统调用。

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

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;
}
```

所后，nemu指令系统执行int指令，完成异常的硬件处理后，根据IDT中的门描述符，切换到AM的目标地址，在AM中保存现场（即各个寄存器的压栈），同时将异常封装为事件，调用nanos-lite中的异常处理函数，执行do_syscall(),寻找到相应异常的处理办法，此时的case=SYS_write。

```
 case SYS_write:
        //a[1]= fd a[2]=buf a[3]=len
        SYSCALL_ARG1(r)=mysys_write(a[1],(void*)a[2],a[3]);
        break;
```

```
//ssize_t write(int fd, const void *buf, size_t count);
int mysys_write(int fd,void* buf,size_t len)
{
    //stdout 或 stderr
    if(fd ==1 || fd ==2)
    {
        //Log("buffer:%s",(char*)buf);
        for(int i=0;i<len;i++)
        {
            char temp;//用于传参给串口
            //需要使用char* 强制类型转换
            temp=*(char*)(buf+i);
            _putc(temp);
        }
        return len;
    }
    else if(fd>=3)
    {
        return fs_write(fd,buf,len);
    }
    else
    {
        panic("fd = %d is not available\n",fd);
    }
    return -1;
}
```

fd>3,进一步调用文件系统中的fs_write()。

```
//fs_write
ssize_t fs_write(int fd,void *buf, size_t len)
{
    //Log("Debug: cd fs_write");
    assert(fd>0 && fd<NR_FILES);
    if(fd<3)
    {
        Log("fd < 3 when use fs_write/n");
        return 0;
    }
    int maxbyte=file_table[fd].size-file_table[fd].open_offset;
    if(len>maxbyte)
    {
        if(fd==FD_FB)//VGA
        {
            fb_write(buf,file_table[fd].open_offset,maxbyte);
        }
        else
```

```
        {
            ramdisk_write(buf,
                        file_table[fd].disk_offset+file_table[fd].open_offset,
                        maxbyte);
        }
        //更新偏移量
        file_table[fd].open_offset+=maxbyte;
        return maxbyte;
    }
    else
    {
        if(fd==FD_FB)//VGA
        {
            fb_write(buf,file_table[fd].open_offset,len);
        }
        else
        {
            ramdisk_write(buf,

 file_table[fd].disk_offset+file_table[fd].open_offset,
                            len);
        }
        //更新偏移量
        file_table[fd].open_offset+=len;
        return len;
    }
}
```

fd==FD_FB，进一步调用fb_write()

```
//am ioe.c 获取屏幕大小
extern void getscreen(int* width,int* height);
void fb_write(const void *buf, off_t offset, size_t len) {
    //Log("cd fb_write");
    if(offset%4!=0||len%4!=0)
    {
        panic("VGA 1像素 should be 4 byte");
    }
    //屏幕上offset处
    int width=0,height=0;
    getscreen(&width,&height);
    int index = offset/sizeof(uint32_t);//第多少个像素
    //x1 ,y1开始的位置
    int screen_y1 = index/width;
    int screen_x1 = index%width;
    //x2,y2结束的位置
    index=(offset+len)/sizeof(uint32_t);
    int screen_y2=index/width;
    int screen_x2 = index%width;

    if(screen_y2==screen_y1)
    {
        _draw_rect(buf,screen_x1,screen_y1,screen_x2-screen_x1,1);
        return;
    }
    else if(screen_y2-screen_y1==1)
    {
```

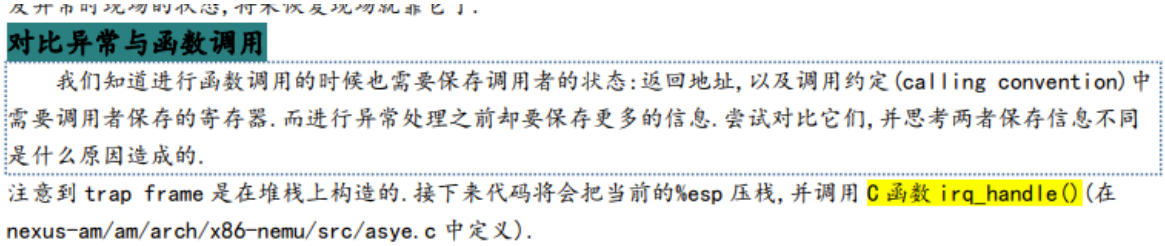
```
        _draw_rect(buf,screen_x1,screen_y1,width-screen_x1,1);
        _draw_rect(buf+4*(width-screen_x1),0,screen_y2,screen_x2,1);
        return;
    }
    else if(screen_y2-screen_y1>1)
    {
        _draw_rect(buf,screen_x1,screen_y1,width-screen_x1,1);
        int draw_index=width-screen_x1;
        _draw_rect(buf+draw_index*4,0,screen_y1+1,width,screen_y2-screen_y1-2);
        draw_index += width*(screen_y2-screen_y1-2);
        _draw_rect(buf+draw_index*4,0,screen_y2,screen_x2,1);
        return;
    }
    panic("screen_y2<screen_y1! erro");
}
```

显而易见，fb_write,将buf中的信息绘制到屏幕上。

VGA 本质上为内存映射的 IO，_draw_rect 的图像绘制实际上是使用 memcpy 修改映射到 video memory 的物理内存（对应 nemu 中的硬件操作）。最终成功实现屏幕的绘制。

其他问题

对比异常和函数调用

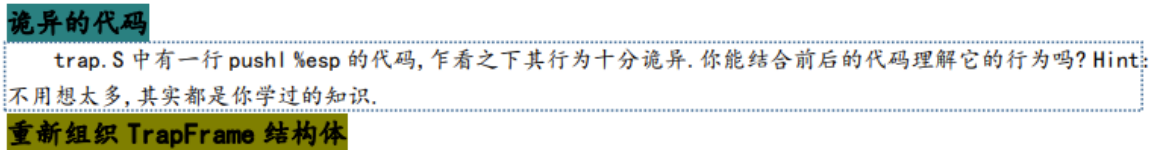


（1）区别：函数调用时需要保存cs和eip。异常处理需要保存所有现场，即eflags、cs、eip、通用寄存器的值。

（2）原因：

- 函数调用发生的时间是已知和固定的；而中断/异常发生的时间一般是随机的。因此异常处理需要保存的内容更多，包括eflags和通用寄存器。
- 函数调用只为主程序服务，异常处理由操作系统负责，与主程序之间没有从属关系，而是并列的。

诡异的代码



pushl %esp 是在执行压入参数的过程，目的是将 TrapFrame 的首地址作为参数传递给 irq_handle 函数的。