南开大学 Nankai University

南　　开　　大　　学

计　算　机　学　院

计算机系统设计实验报告

PA3：穿越时空的旅程——异常控制流

邵琦

年级：2020 级

专业：计算机科学与技术

指导教师：卢冶

2023 年 5 月 15 日

目录

一、 实验目的

1. 理解操作系统概念。
2. 将机器底层和上层应用实现联系。
3. 了解操作系统中的系统调用，并实现中断机制。
4. 了解文件系统的基本内容，实现简易文件系统。
5. 实现支持文件操作的操作系统。

二、 实验内容

1. 学习操作系统的基本概念、系统调用，实现中断机制。
2. 完善系统调用，实现简易文件系统。
3. 将输入输出抽象成文件，并运行仙剑奇侠传。

三、 阶段一

(一) 运行环境

首先我们准备 PA3 所需的环境:

我们打开 nano-lite 中 main.c 文件的 HAS_ASYE 宏定义。修改 Navy-apps 和 nexus-am 中 Makefile 的编译选项，使其 ISA 和 ARCH 均指向 x86。

我们需要在在 navy-apps/tests/dummy 下执行 make，就可以生成 dummy 的可执行文件，并在 nano-lite 下运行 make update，生成 ramdisk 镜像文件 ramdisk.img, 并将可执行文件包含进 Nanos-lite 成为其中的一部分。

(二) 实现 Loader

Loader 是一个用于加载程序的模块，程序中包括代码和数据，他们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，程序就可以开始执行了。

首先我们让 Navy-apps 项目上的程序默认编译到 x86，然后在 navy-apps/tests/dummy 下执行 make，就可以生成 dummy 的可执行文件。

为了避免和 nanos-lite 冲突，pa 约定目前用户程序需要被链接到内存位置 0x4000000 处，也就是我们的 loader 需要将 ramdisk 中从 0 开始所有内容放在 0x4000000，并作为程序的入口返回。要将 ramdisk 复制到指定位置，可以使用 ramdisk_read 函数，代码如下:

实现 Loader

```
#include "common.h"
#include "fs.h"

#define DEFAULT_ENTRY ((void *)0x4000000)


extern uint8_t ramdisk_start;
extern uint8_t ramdisk_end;
```

```
#define RAMDISK_SIZE ((&ramdisk_end)-(&ramdisk_start))
extern void ramdisk_read(void *buf,off_t offset,size_t len);


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

(三) 准备 IDT

我们首先在 nemu/include/cpu/reg.h 添加 IDTR 寄存器，用于存放 IDT 的首地址和长度，为了通过 differential testing，要在 cpu 结构体里添加一个 CS 寄存器，并在 restart() 函数中初始化为 8，EFLAGS 初始化为 0x2。

添加 IDTR 寄存

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

  rtlreg_t cs;
```

添加 IDTR 寄存

```
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));
```

之后，我们实现 lidt 指令。将操作数信息从 eax 寄存器中读出。

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

图 1: lidt 指令

注册执行函数:

实现 lidt 指令

```
make_EHelper(lidt);
```

实现执行函数:

实现 lidt 指令

```
make_EHelper(lidt) {
  t1 = id_dest -> val;
  rtl_lm(&t0, &t1, 2);
  cpu.idtr.limit = t0;

  t1 = id_dest -> val + 2;
  rtl_lm(&t0, &t1, 4);
  cpu.idtr.base = t0;

#ifdef DEBUG
  Log("idtr.limit=0x%x", cpu.idtr.limit);
  Log("idtr.base=0x%x", cpu.idtr.base);
#endif
  print_asm_template1(lidt);
}
```

opcode_table 注册:

实现 lidt 指令

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

(四) 触发异常

我们实现 int 指令。首先需要实现 raise_intr 函数。raise_intr 函数中模拟了异常出现后的硬件操作，具体过程就是寄存器压栈，根据索引取出 IDT 数组信息，处理结构体信息得到跳转目标地址并设置跳转。

执行函数:

实现 int 指令

```
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.
   */

  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);

  vaddr_t gate_addr = cpu.idtr.base + NO * sizeof(GateDesc);
  assert(gate_addr <= cpu.idtr.base + cpu.idtr.limit);

  uint32_t off_15_0 = vaddr_read(gate_addr,2);
  uint32_t off_32_16 = vaddr_read(gate_addr+sizeof(GateDesc)-2,2);
  uint32_t target_addr = (off_32_16 << 16) + off_15_0;
#ifdef DEBUG
  Log("target_addr=0x%x",target_addr);
#endif
  decoding.is_jmp = 1;
  decoding.jmp_eip = target_addr;
}
```

实现执行函数:

实现 int 指令

```
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
}
```

opcode_table 注册:

实现 int 指令

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

(五) 保存现场

我们实现 pusha 指令，该指令需要将所有寄存器进行压栈操作。

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

图 2: pusha 指令

执行函数:

实现 pusha 指令

```
make_EHelper(pusha);
```

实现执行函数:

实现 pusha 指令

```
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");
}
```

opcode_table 注册:

实现 pusha 指令

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

之后，我们需要重新组织 TrapFrame。

重新组织 TrapFrame

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

  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;
};
```

在完成此阶段后，可以看到 HIT BAD TRAP 提示。

（六） 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()。

事件分发

```
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;
}
```

（七） 系统调用处理

首先我们实现正确的 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)

SYSCALL_ARGx() 宏

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

之后，我们添加系统调用，设置系统调用的返回值。（之后添加的系统调用类似，不再赘述）

添加系统调用

```
int sys_none()
{
  return 1;
}

void sys_exit(int a)
{
  _halt(a);
}

int sys_write(int fd,void* buf,size_t len)
```

```
12 {
13   if(fd==1||fd==2)
14   {
15     char c;
16     //Log("buffer:%s",(char*)buf);
17     for(int i=0;i<len;i++)
18     {
19       memcpy(&c,buf+i,1);
20       _putc(c);
21     }
22     return len;
23   }
24   if(fd>=3)
25   {
26     return fs_write(fd,buf,len);
27   }
28   Log("fd<=0");
29
30   return -1;
31 }
32
33 int sys_brk(int addr)
34 {
35   return 0;
36 }
37
38 int sys_open(const char* filename)
39 {
40   return fs_open(filename,0,0);
41 }
42
43 int sys_read(int fd,void*buf,size_t len)
44 {
45   return fs_read(fd,buf,len);
46 }
47
48 int sys_close(int fd)
49 {
50   return fs_close(fd);
51 }
52
53 int sys_lseek(int fd,off_t offset,int whence)
54 {
55   return fs_lseek(fd,offset,whence);
56 }
57
58 _RegSet* do_syscall(_RegSet *r) {
59   uintptr_t a[4];
```

```
60    a[0] = SYSCALL_ARG1(r);
61    a[1] = SYSCALL_ARG2(r);
62    a[2] = SYSCALL_ARG3(r);
63    a[3] = SYSCALL_ARG4(r);
64
65    switch (a[0]) {
66      case SYS_none:
67        SYSCALL_ARG1(r)=sys_none();
68        break;
69      case SYS_exit:
70        sys_exit(a[1]);
71        break;
72      case SYS_write:
73        SYSCALL_ARG1(r)=sys_write(a[1],(void*)a[2],a[3]);
74        break;
75      case SYS_brk:
76        SYSCALL_ARG1(r)=sys_brk(a[1]);
77        break;
78      case SYS_open:
79        SYSCALL_ARG1(r)=sys_open((char*)a[1]);
80        break;
81      case SYS_read:
82        SYSCALL_ARG1(r)=sys_read(a[1],(void*)a[2],a[3]);
83        break;
84      case SYS_close:
85        SYSCALL_ARG1(r)=sys_close(a[1]);
86        break;
87      case SYS_lseek:
88        SYSCALL_ARG1(r)=sys_lseek(a[1],a[2],a[3]);
89        break;
90
91      default: panic("Unhandled syscall ID = %d", a[0]);
92    }
93
94    return NULL;
95  }
```

(八) 恢复现场

之后我们实现 popa 指令，将所有寄存器以一定顺序弹出栈。

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

图 3: popa 指令

执行函数:

popa 指令

```
make_EHelper(popa);
```

实现执行函数:

popa 指令

```
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");
}
```

opcode_table 注册:

popa 指令

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

除此以外，我们还需实现 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:

iret 指令

```
make_EHelper(iret);
```

实现执行函数:

iret 指令

```
make_EHelper(iret) {
  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 注册:

iret 指令

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

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

```
For help, type "help"
(nemu) c
[src/main.c,19,main] 'Hello World!' from Nanos-lite
[src/main.c,20,main] Build time: 18:57:21, May  8 2022
[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
```

图 4: 阶段一结果

四、 阶段二

首先，我们需要把 ramdisk 中的唯一文件换成 hello 程序。

hello 程序

```
OBJCOPY_FILE = $(NAVY_HOME)/tests/hello/build/hello-x86
```

(一) 标准输出

首先我们要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。

辅助函数 _write

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

之后在 do_syscall 函数中加入 SYS_write 系统调用。

我们再实现 sys_write 系统调用。

sys_write

```
int sys_write(int fd,void* buf,size_t len)
{
  if(fd==1||fd==2)
  {
```

```
    char c;
    //Log("buffer:%s",(char*)buf);
    for(int i=0;i<len;i++)
    {
      memcpy(&c,buf+i,1);
      _putc(c);
    }
    return len;
  }
  if(fd>=3)
  {
    return fs_write(fd,buf,len);
  }
  Log("fd<=0");

  return -1;
}
```

(二) 堆区管理

在没有实现堆区时，在 sys_write 函数中通过 Log 查看 write 的调用的情况的时候发现通过 printf 输出的时候是逐个字符输出的。调整堆区大小是通过 _sbrk 库函数来实现的，它的功能是将用户程序的的数据段结束的位置增长 increment 字节来实现。

首先我们实现 sys_brk 系统调用的实现，可以让 SYS_brk 总是返回 0，表示堆区大小的调整总是成功。

sys_brk

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

之后我们在用户层实现 _sbrk()，其主要思路是 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。

实现 _sbrk()

```
void *_sbrk(intptr_t increment){
  //return (void *)-1;
  /**/
  extern 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;

  /*
  extern char _end;
  intptr_t program_break=(intptr_t)&_end;
  void *_sbrk(intptr_t increment);
  int test=_syscall_(SYS_brk,old_pb+increment,0,0);
  if(test==0)
  {
    program_break+=increment;
    return (void*)old_pb;
  }
  else
  {
    return (void *)-1;
  }
  */

}
```

(三) 让 loader 使用文件

首先我们需要对 nano-lite 的 Makefile 进行修改。

Makefile 进行修改

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

之后我们修改文件记录表 file_table 结构体。

file_table 结构体

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

首先需要在 nanos-lite/include/fs.h 中注册需要的函数，并在 nanos-lite/src/fs.c 中实现函数体。同时，还需要在 nanos-lite/src/loader.c 中引入 fs.h 头文件。

实现函数体

```
size_t fs_filesz(int fd)
{
```

```
  assert(fd>=0&&fd<NR_FILES);
  return file_table[fd].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;
}
```

实现 fs_open 根据文件名查找文件描述符 fd。

fs_open

```
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 filename not exist in file_table");
  return -1;
}
```

实现 fs_read 读取 fd 对应的文件。

fs_read

```
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||fd==FD_FB");
    return 0;
  }
  if(fd==FD_EVENTS)
  {
    //Log("fd=%d",fd);
    return events_read(buf,len);
  }
  int n=fs_filesz(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;
}
```

实现 fs_close 关闭文件。

fs_close

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

之后我们重新实现 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。

(四) 完整的文件系统

实现 fs_write 对 fd 对应的文件进行写操作和实现。

fs_write

```
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||fd==FD_DISPINFO");
    return 0;
  }
  int n=fs_filesz(fd)-get_open_offset(fd);
  if(n>len)
  {
    n=len;
  }
  if(fd==FD_FB)
  {
    fb_write(buf,get_open_offset(fd),n);
  }
  else
  {
    ramdisk_write(buf,disk_offset(fd)+get_open_offset(fd),n);
  }
  set_open_offset(fd,get_open_offset(fd)+n);
  return n;
}
```

实现 fs_lseek 修改 fd 对应文件的 open_offset。

fs_lseek

```
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;
  }
}
```

最后我们实现系统调用。

系统调用

```
extern int fs_open(const char* filename,int flags,int mode);
```

```
2  extern ssize_t fs_read(int fd,void* buf,size_t len);
3  extern ssize_t fs_write(int fd,const void* buf,size_t len);
4  extern off_t fs_lseek(int fd,off_t offset,int whence);
5  extern int fs_close(int fd);
6
7  int sys_none()
8  {
9    return 1;
10 }
11
12 void sys_exit(int a)
13 {
14   _halt(a);
15 }
16
17 int sys_write(int fd,void* buf,size_t len)
18 {
19   if(fd==1||fd==2)
20   {
21     char c;
22     //Log("buffer:%s",(char*)buf);
23     for(int i=0;i<len;i++)
24     {
25       memcpy(&c,buf+i,1);
26       _putc(c);
27     }
28     return len;
29   }
30   if(fd>=3)
31   {
32     return fs_write(fd,buf,len);
33   }
34   Log("fd<=0");
35
36   return -1;
37 }
38
39 int sys_brk(int addr)
40 {
41   return 0;
42 }
43
44 int sys_open(const char* filename)
45 {
46   return fs_open(filename,0,0);
47 }
48
49 int sys_read(int fd,void*buf,size_t len)
```

```
50 {
51   return fs_read(fd,buf,len);
52 }
53
54 int sys_close(int fd)
55 {
56   return fs_close(fd);
57 }
58
59 int sys_lseek(int fd,off_t offset,int whence)
60 {
61   return fs_lseek(fd,offset,whence);
62 }
63
64 _RegSet* do_syscall(_RegSet *r) {
65   uintptr_t a[4];
66   a[0] = SYSCALL_ARG1(r);
67   a[1] = SYSCALL_ARG2(r);
68   a[2] = SYSCALL_ARG3(r);
69   a[3] = SYSCALL_ARG4(r);
70
71   switch (a[0]) {
72     case SYS_none:
73       SYSCALL_ARG1(r)=sys_none();
74       break;
75     case SYS_exit:
76       sys_exit(a[1]);
77       break;
78     case SYS_write:
79       SYSCALL_ARG1(r)=sys_write(a[1],(void*)a[2],a[3]);
80       break;
81     case SYS_brk:
82       SYSCALL_ARG1(r)=sys_brk(a[1]);
83       break;
84     case SYS_open:
85       SYSCALL_ARG1(r)=sys_open((char*)a[1]);
86       break;
87     case SYS_read:
88       SYSCALL_ARG1(r)=sys_read(a[1],(void*)a[2],a[3]);
89       break;
90     case SYS_close:
91       SYSCALL_ARG1(r)=sys_close(a[1]);
92       break;
93     case SYS_lseek:
94       SYSCALL_ARG1(r)=sys_lseek(a[1],a[2],a[3]);
95       break;
96
97     default: panic("Unhandled syscall ID = %d", a[0]);
```

```
  }

  return NULL;
}
```

并且我们修改 nano.c。

修改 nano.c

```
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);
}

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);
}
```

至此，阶段二完成。

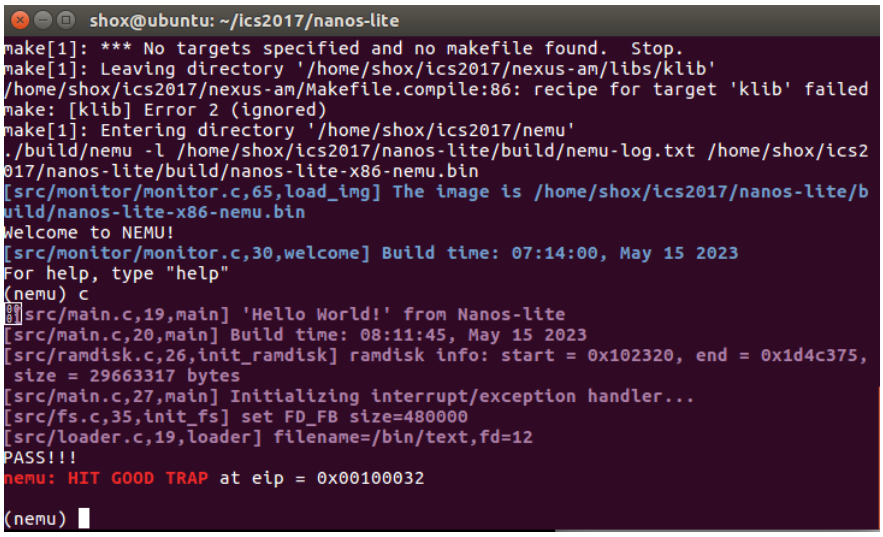

图 5: 阶段二实验结果

五、 阶段三

(一) 把 VGA 显存抽象成文件

首先我们定义 getScreen 函数，返回屏幕大小信息。

getScreen 函数

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

实现 init_fs 函数对文件记录表中的/dev/fb 的大小进行初始化。

init_fs 函数

```
void init_fs() {
  // TODO: initialize the size of /dev/fb

  extern void getScreen(int* p_width,int* p_height);
  int width=0,height=0;
  getScreen(&width,&height);
  file_table[FD_FB].size=width*height*sizeof(uint32_t);
  Log("set FD_FB size=%d",file_table[FD_FB].size);

  /*
  file_table[FD_FB].size=_screen.height*_screen.width*4;
  */
}
```

实现 fb_write 函数绘制屏幕上的像素点。

fb_write 函数

```
extern void getScreen(int* p_width,int* p_height);

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);
  //1
  if(screen_y2==screen_y1)
  {
    _draw_rect(buf,screen_x1,screen_y1,len/4,1);
    return;
  }
  //2
  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;
  }
  //>=3
  _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)
      ;
  */
  assert(offset%4==0 && len%4==0);
  int index,screen_x,screen_y;
  int w=0;
  int h=0;
  getScreen(&w,&h);
  for(int i=0;i<len/4;i++)
  {
    index=offset/4+i;
    screen_y=index/w;
```

```
    screen_x=index%w;
    _draw_rect(buf+i*4,screen_x,screen_y,1,1);
  }
}
```

实现 init_device 函数将长宽信息按格式写入 dispinfo。

init_device 函数

```
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);
  sprintf(dispinfo,"WIDTH:%d\nHEIGHT:%d\n",width,height);
  /*
  strcpy(dispinfo,"WIDTH:400\nHEIGHT:300");
  */
}
```

实现 dispinfo_read。

dispinfo_read

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

补充 fs_write 和 fs_read。

fs_write 和 fs_read

```
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||fd==FD_DISPINFO");
    return 0;
  }
  int n=fs_filesz(fd)-get_open_offset(fd);
  if(n>len)
  {
    n=len;
  }
  if(fd==FD_FB)
  {
```

```
    fb_write(buf,get_open_offset(fd),n);
  }
  else
  {
    ramdisk_write(buf,disk_offset(fd)+get_open_offset(fd),n);
  }
  set_open_offset(fd,get_open_offset(fd)+n);
  return n;
}

void dispinfo_read(void *buf,off_t offset,size_t len);

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

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||fd==FD_FB");
    return 0;
  }
  if(fd==FD_EVENTS)
  {
    //Log("fd=%d",fd);
    return events_read(buf,len);
  }
  int n=fs_filesz(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;
}
```

最终运行结果如下:

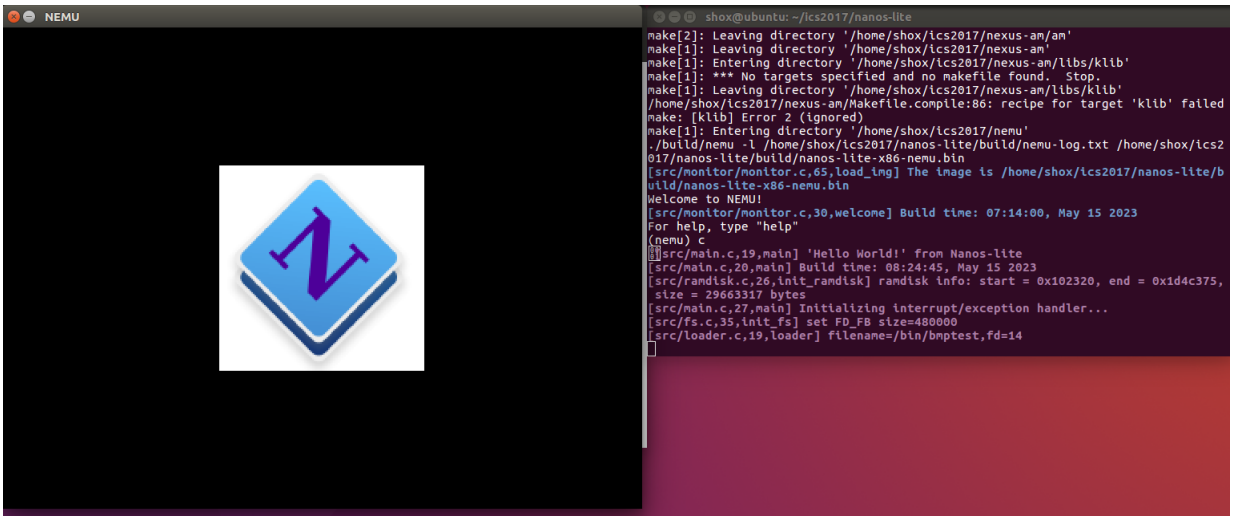

图 6: 最终运行结果

(二) 把设备输入抽象成文件

我们将时钟信息和按键信息抽象成文件/dev/events，并对其进行读取，用户程序可以从中一次读出一个输入事件。首先在 nanos lite/src/device.c 中实现 events_read 函数，将事件写入到 buf 中。

events_read 函数

```
size_t events_read(void *buf, size_t len) {
  char str[40];
  bool down=false;
  int key=_read_key();
  if(key&0x8000)
  {
    key^=0x8000;
    down=true;
  }
  if(key!=_KEY_NONE)
  {
    sprintf(str,"%s %s\n",down?"kd":"ku",keyname[key]);
  }
  else
  {
    //Log("key:%d",key);
    sprintf(str,"t %d\n",_uptime());
  }
  if(strlen(str)<=len)
  {
    strncpy((char*)buf,str,strlen(str));
    return strlen(str);
  }
  Log("strlen(event)>len,return 0");
  return 0;
```

```
}
```

之后我们补充 fs_read:

fs_read

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

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

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||fd==FD_FB");
    return 0;
  }
  if(fd==FD_EVENTS)
  {
    //Log("fd=%d",fd);
    return events_read(buf,len);
  }
  int n=fs_filesz(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;
}
```

最终运行结果如下:

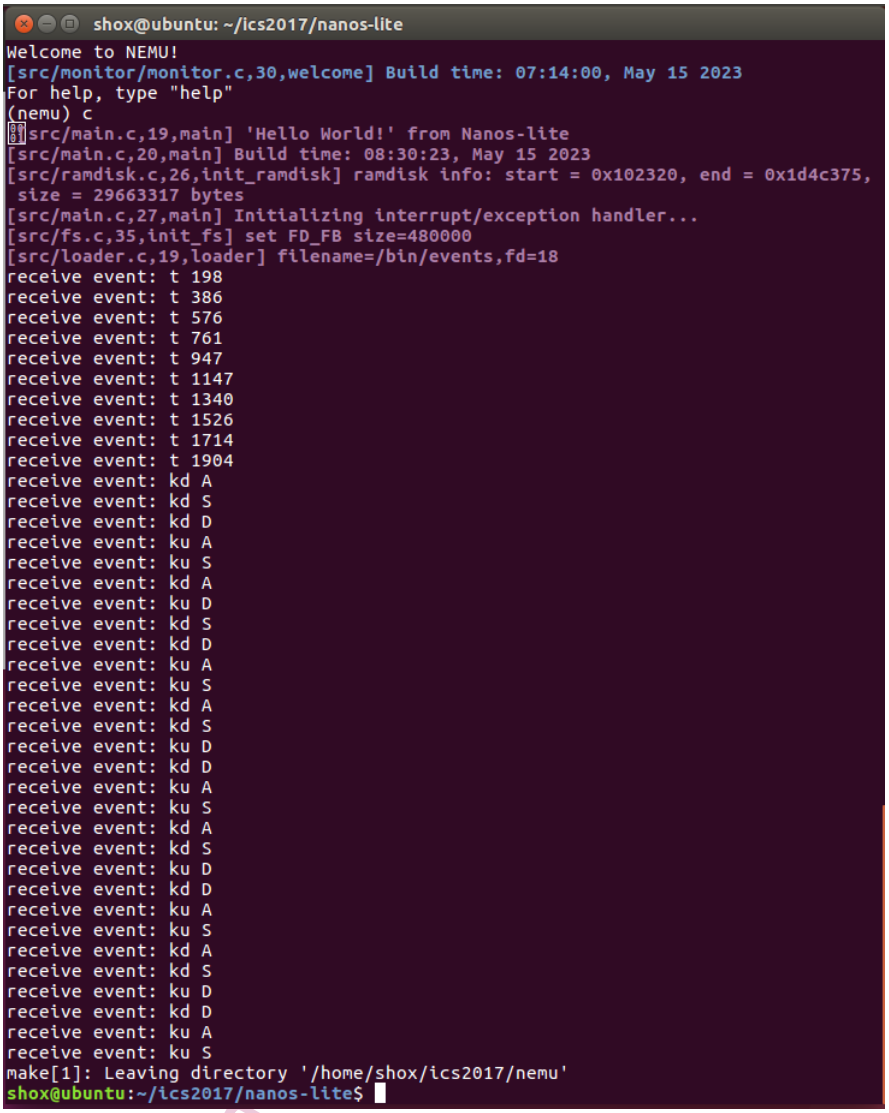

图 7: 最终运行结果

(三) 运行仙剑奇侠传

首先，我们要先实现 cwtl 指令:

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

图 8: cwtl 指令

执行函数:

cwtl 指令

```
1 make_EHelper(cwtl);
```

实现执行函数:

cwtl 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");
}
```

opcode_table 注册:

cwtl 指令

```
  /* 0x98 */     EX(cwtl), EX(cltd), EMPTY, EMPTY,
```

最终运行结果如下:

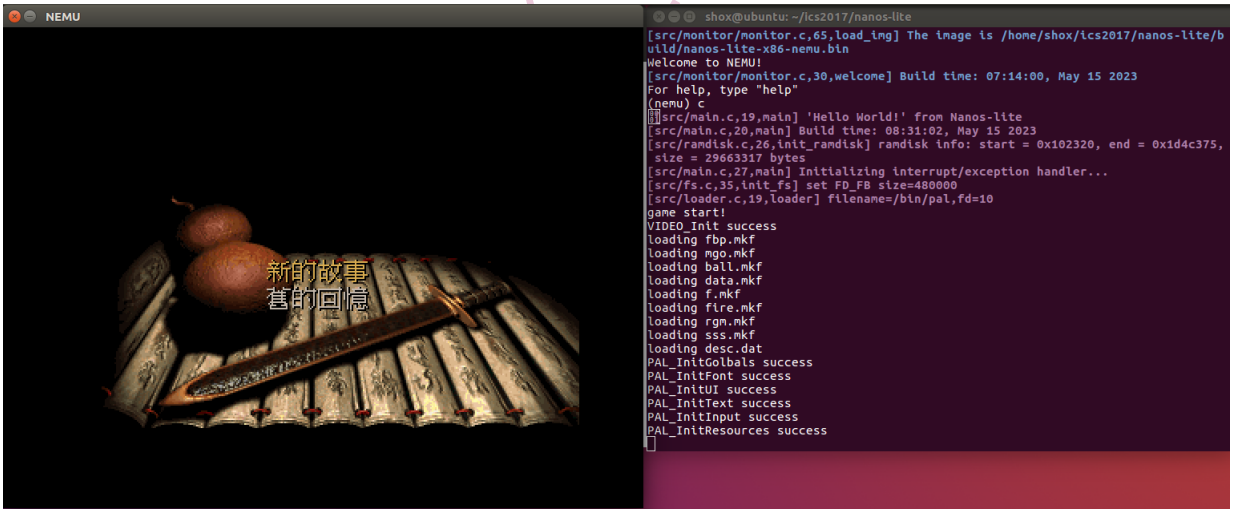

图 9: 最终运行结果

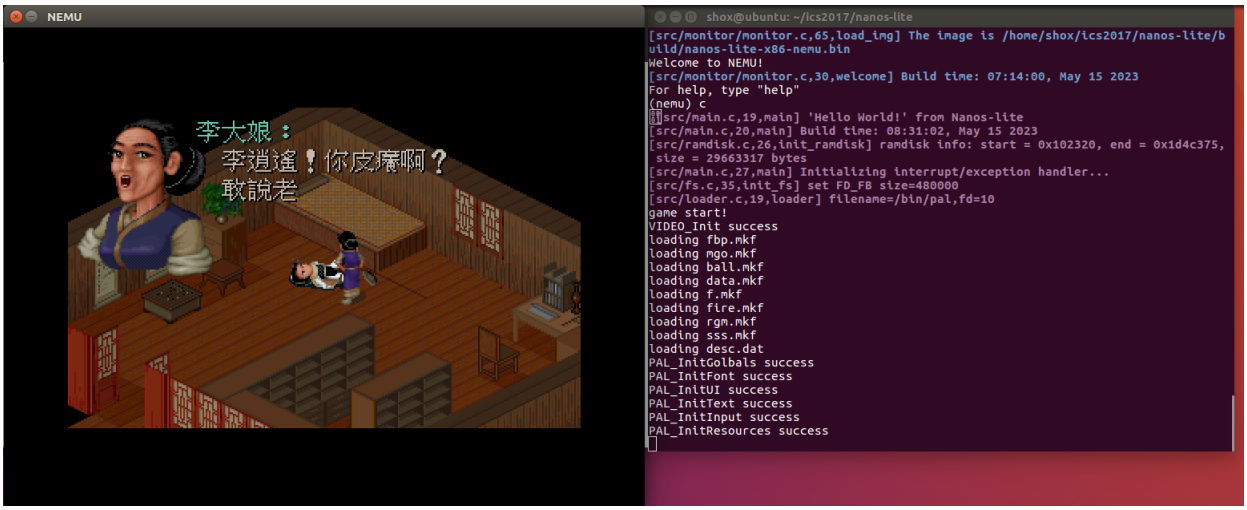

图 10: 最终运行结果

六、 遇到的问题

(一) makefile 问题

问题：在阶段一时，编译运行遇到如下问题：

```
kjy@kjy-virtual-machine:~/NKU_PA/ics2017/nanos-lite$ make run
Building nanos-lite [native]
+ CC src/irq.c
+ AS src/initrd.S
+ CC src/device.c
+ CC src/main.c
+ CC src/syscall.c
src/syscall.c: In function ‘do_syscall’:
src/syscall.c:6:10: error: implicit declaration of function ‘SYSCALL_ARG1’ [-Werror=implicit-function-declaration]
   a[0] = SYSCALL_ARG1(r);
          ^
cc1: all warnings being treated as errors
/home/kjy/NKU_PA/ics2017/nexus-am/Makefile.compile:69: recipe for target '/home/kjy/NKU_PA/ics2017/nanos-lite/build/native//./src/syscall.o' failed
make: *** [/home/kjy/NKU_PA/ics2017/nanos-lite/build/native//./src/syscall.o] Error 1
```

图 11: makefile 问题

解决方式：在 nexus-am 的 makefile.check，将“ARCH ?=-x86-native”改成“ARCH ?=-x86-nemu”即可。

(二) *_sbrk 问题

问题：实现了 *_sbrk 且添加了 SYS_brk 之后，hello world 还是逐字符输出。

解决方式：在 hello 目录重新 make 一下然后再 make update 即可。

(三) 《仙剑》卡住的问题

问题：仙剑奇侠传编译能通过但是一直卡在初始页面。

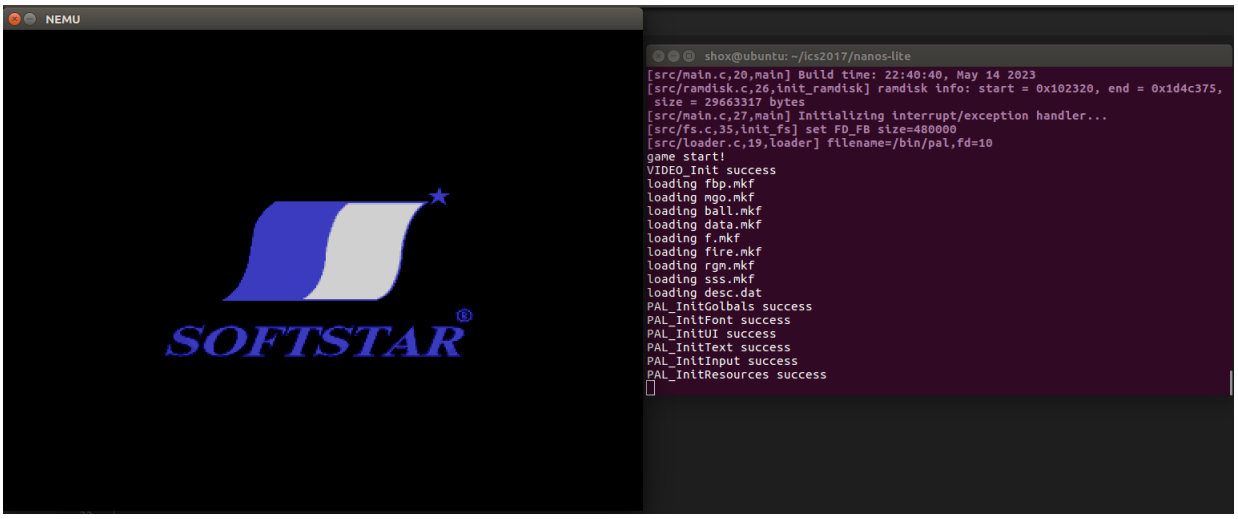

图 12: makefile 问题

解决方式:

对于《仙剑》卡住的问题，经过测试，发现程序一直在重复读取 FD_EVENTS，即读取事件。

仙剑在等待时钟信息到来，因此卡死。

这个错误是由于 Nanos-lite/src/device.c 中打包时钟、键盘事件的时候格式不正确：

- 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

以'\n'结束。

图 13: 解决方式

七、 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.

图 14: 必答题

库函数：是将函数封装入库，供用户使用的一种方式。方法是把一些常用到的函数编完放到一个文件里，供不同的人进行调用。

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。

Nanos-lite：是操作系统 Nanos 的裁剪版，是 MENU 的客户端，运行在 AM 之上。

AM：是计算机的一种抽象模型，用来描述计算机如何构成，在概念上定义了一台抽象计算机，从运行程序的视角刻画了一台计算机应该具备的功能。

NEMU：是一款经过简化的 x86 全系统模拟器。

Nanos-lite 是 NEMU 的客户端，运行在 AM 之上，仙剑是 Nanos-lite 的客户端，运行在 Nanos-lite 之上。

仙剑运行时有时需要调用库函数完成一系列操作，如内存拷贝、打印等，在库函数中有时会进行系统调用，系统调用涉及到支持仙剑游戏的操作系统即 Nanos-lite 提供的 API，Nanos-lite 作为操作系统 Nanos 的裁剪版为仙剑游戏提供简易的运行环境，如文件管理等，而 Nanos-lite 需要运行在 AM 之上，使用 NEMU 模拟出来的 x86 硬件。

仙剑游戏运行的过程中，NEMU 首先模拟出 x86 硬件，让 Nanos-lite 简易操作系统可以运行在模拟出的硬件上，该操作系统会为游戏提供运行时环境并相应系统调用。

(一) 读取游戏存档

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。

PAL_LoadGame

```
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)
    {
```

```
30          return -1;
31      }
32
33      //
34      // Read all data from the file and close.
35      //
36      fread(&s, sizeof(SAVEDGAME), 1, fp);
37      fclose(fp);
38
39      //
40      // Adjust endianness
41      //
42      DO_BYTESWAP(&s, sizeof(SAVEDGAME));
43
44      //
45      // Get all the data from the saved game struct.
46      //
47      gpGlobals->viewport = PAL_XY(s.wViewportX, s.wViewportY);
48      gpGlobals->wMaxPartyMemberIndex = s.nPartyMember;
49      gpGlobals->wNumScene = s.wNumScene;
50      gpGlobals->fNightPalette = (s.wPaletteOffset != 0);
51      gpGlobals->wPartyDirection = s.wPartyDirection;
52      gpGlobals->wNumMusic = s.wNumMusic;
53      gpGlobals->wNumBattleMusic = s.wNumBattleMusic;
54      gpGlobals->wNumBattleField = s.wNumBattleField;
55      gpGlobals->wScreenWave = s.wScreenWave;
56      gpGlobals->sWaveProgression = 0;
57      gpGlobals->wCollectValue = s.wCollectValue;
58      gpGlobals->wLayer = s.wLayer;
59      gpGlobals->wChaseRange = s.wChaseRange;
60      gpGlobals->wChasespeedChangeCycles = s.wChasespeedChangeCycles;
61      gpGlobals->nFollower = s.nFollower;
62      gpGlobals->dwCash = s.dwCash;
63  #ifndef PAL_CLASSIC
64      gpGlobals->bBattleSpeed = s.wBattleSpeed;
65      if (gpGlobals->bBattleSpeed > 5 || gpGlobals->bBattleSpeed == 0)
66      {
67          gpGlobals->bBattleSpeed = 2;
68      }
69  #endif
70
71      memcpy(gpGlobals->rgParty, s.rgParty, sizeof(gpGlobals->rgParty));
72      memcpy(gpGlobals->rgTrail, s.rgTrail, sizeof(gpGlobals->rgTrail));
73      gpGlobals->Exp = s.Exp;
74      gpGlobals->g.PlayerRoles = s.PlayerRoles;
75      memset(gpGlobals->rgPoisonStatus, 0, sizeof(gpGlobals->rgPoisonStatus));
76      memcpy(gpGlobals->rgInventory, s.rgInventory, sizeof(gpGlobals->
            rgInventory));
```

```
77    memcpy(gpGlobals->g.rgScene, s.rgScene, sizeof(gpGlobals->g.rgScene));
78    memcpy(gpGlobals->g.rgObject, s.rgObject, sizeof(gpGlobals->g.rgObject));
79    memcpy(gpGlobals->g.lprgEventObject, s.rgEventObject,
80        sizeof(EVENTOBJECT) * gpGlobals->g.nEventObject);
81
82    gpGlobals->fEnteringScene = FALSE;
83
84    PAL_CompressInventory();
85
86    //
87    // Success
88    //
89    return 0;
90 }
```

而 fread 库函数调用 libc 中的 _read 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。

_read 函数

```
1 int _read(int fd, void *buf, size_t count) {
2   //_exit(SYS_read);
3   return _syscall_(SYS_read,fd,(uintptr_t)buf,count);
4 }
```

nemu 指令系统执行 int 指令，完成异常的硬件处理后，根据 idt 内容切换至 AM 的目标地址。在 AM 中保存现场，将异常封装成事件，并调用 nanos-lite 中的异常处理函数。异常处理函数在事件分发后调用 do_syscall，分发系统调用的具体处理函数，对于 fread 来说就是 sys_read 函数。

调用 do_syscall

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

sys_read 函数

```
1 int sys_read(int fd,void*buf,size_t len)
2 {
3   return fs_read(fd,buf,len);
4 }
```

而 sys_read 调用 fs_read 进行文件系统的操作。根据不同的 fd，执行不同的处理函数。

fs_read

```
1 void dispinfo_read(void *buf,off_t offset,size_t len);
2
3 extern size_t events_read(void *buf, size_t len);
4
5 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||fd==FD_FB");
    return 0;
  }
  if(fd==FD_EVENTS)
  {
    //Log("fd=%d",fd);
    return events_read(buf,len);
  }
  int n=fs_filesz(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;
}
```

之后将时钟信息和按键信息抽象成文件/dev/events，并对其进行读取，用户程序可以从中一次读出一个输入事件。events_read 函数，将该事件写入到 buf 中。

events_read 函数

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

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

最后/proc/dispinfo 的内容已经提前写入将到字符串 dispinfo。dispinfo_read 把字符串 dispinfo 中 offset 开始的 len 字节写到 buf 中。

dispinfo_read

```
static char dispinfo[128] __attribute__((used));

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

(二) 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 为例说明关系。

redraw

```
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();
}
```

$NDL_DrawRectNDL_Render$

```
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];
      }
    }
  }
}

int NDL_Render() {
  if (has_nwm) {
    fflush(stdout);
  } else {
    for (int i = 0; i < canvas_h; i ++) {
      fseek(fbdev, ((i + pad_y) * screen_w + pad_x) * sizeof(uint32_t),
          SEEK_SET);
      fwrite(&canvas[i * canvas_w], sizeof(uint32_t), canvas_w, fbdev);
    }
    fflush(fbdev);
  }
}
```

而 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。

_write 函数

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

nemu 指令系统执行 int 指令，完成异常的硬件处理后，根据 idt 内容切换至 AM 的目标地址。在 AM 中保存现场，将异常封装成事件，并调用 nanos-lite 中的异常处理函数。异常处理函数在事件分发后调用 do_syscall，分发系统调用的具体处理函数，对于 fwrite 来说就是 sys_write 函数。

调用 do_syscall

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

sys_write 函数

```
int sys_write(int fd,void* buf,size_t len)
{
  if(fd==1||fd==2)
  {
    char c;
```

```
    //Log("buffer:%s",(char*)buf);
    for(int i=0;i<len;i++)
    {
      memcpy(&c,buf+i,1);
      _putc(c);
    }
    return len;
  }
  if(fd>=3)
  {
    return fs_write(fd,buf,len);
  }
  Log("fd<=0");

  return -1;
}
```

对于显存的抽象文件/dev/fb，sys_write 调用 fs_write 进行文件系统的操作，因此进入了显存的写函数 fb_write，fb_write 调用 _draw_rect（AM）实现图像的绘制。

fs_write

```
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||fd==FD_DISPINFO");
    return 0;
  }
  int n=fs_filesz(fd)-get_open_offset(fd);
  if(n>len)
  {
    n=len;
  }
  if(fd==FD_FB)
  {
    fb_write(buf,get_open_offset(fd),n);
  }
  else
  {
    ramdisk_write(buf,disk_offset(fd)+get_open_offset(fd),n);
  }
  set_open_offset(fd,get_open_offset(fd)+n);
  return n;
}
```

fb_write

```
extern void getScreen(int* p_width,int* p_height);

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);
  //1
  if(screen_y2==screen_y1)
  {
    _draw_rect(buf,screen_x1,screen_y1,len/4,1);
    return;
  }
  //2
  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;
  }
  //>=3
  _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)
      ;
  */
  assert(offset%4==0 && len%4==0);
  int index,screen_x,screen_y;
  int w=0;
  int h=0;
  getScreen(&w,&h);
  for(int i=0;i<len/4;i++)
  {
    index=offset/4+i;
    screen_y=index/w;
```

```
47      screen_x=index%w;
48      _draw_rect(buf+i*4,screen_x,screen_y,1,1);
49    }
50  }
```

最后，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。

_draw_rect

```
1  void _draw_rect(const uint32_t *pixels, int x, int y, int w, int h) {
2    int cp_bytes = sizeof(uint32_t) * min(w, _screen.width - x);
3    for (int j = 0; j < h && y + j < _screen.height; j ++) {
4      memcpy(&fb[(y + j) * W + x], pixels, cp_bytes);
5      pixels += w;
6    }
7  }
```

其余函数同理。

八、 其他问题

(一) 对比异常与函数调用

对比异常与函数调用

我们知道进行函数调用的时候也需要保存调用者的状态:返回地址,以及调用约定(calling convention)中需要调用者保存的寄存器.而进行异常处理之前却要保存更多的信息.尝试对比它们,并思考两者保存信息不同是什么原因造成的.

图 15: 对比异常与函数调用

函数调用保存的信息包括返回地址和调用约定中需要调用者保存的寄存器。返回地址是指函数调用完成后需要返回到的下一条指令的地址，它保存了函数调用之前的执行位置，以确保程序能够正确地继续执行。调用约定是关于参数传递和寄存器的使用规则，它规定了哪些寄存器的值需要在函数调用过程中保存，以确保函数调用之后的代码能够正确地访问和修改寄存器。异常处理需要保存更多的信息，这是因为异常是在程序执行过程中遇到的意外情况，需要进行特殊处理。为了能够安全地处理异常，需要保存更多的上下文信息。异常处理可能需要保存的信息包括:

返回地址：与函数调用类似，保存异常处理完成后需要返回的位置。

寄存器：除了调用约定中需要调用者保存的寄存器外，异常处理还需要保存当前代码执行的上下文，以便在异常处理完成后能够恢复执行。这包括当前指令的地址、各个寄存器的值等。

异常处理器状态：异常处理可能涉及到多个层级的异常处理器，因此需要保存异常处理器的调用栈信息，以确保能够按照正确的顺序处理异常。

其他上下文信息：根据具体的异常处理机制和编程语言，可能还需要保存其他相关的上下文信息，如异常类型、异常消息等。

分析原因:

1. 异常处理需要保存更多的信息是为了能够正确地处理异常情况，保证程序的稳定性和正确性。相比之下，函数调用只需要保存一些必要的信息来确保函数的调用和返回能够正常进行，不需要额外处理异常情况，因此保存的信息相对较少。

2. 调用子程序过程发生的时间是已知和固定的，即在主程序中的调用指令（CALL）执行时发生；而中断/异常发生的时间一般是随机的：CPU 在执行某一主程序时收到中断源提出的中断申请时，就发生中断过程，而中断申请一般由硬件电路产生，申请提出时间是随机的（软中断发生时间是固定的），也可以说，调用子程序是程序设计者事先安排的，而执行中断服务程序是由系统工作环境随机决定的。

3. 子程序完全为主程序服务的，两者属于主从关系，主程序需要子程序时就去调用子程序，并把调用结果带回主程序继续执行；而中断服务程序与主程序两者一般是无关的，不存在谁为谁服务的问题，两者是平行关系。

(二) 诡异的代码

诡异的代码

trap.S 中有一行 pushl %esp 的代码，乍看之下其行为十分诡异. 你能结合前后的代码理解它的行为吗? Hint: 不用想太多，其实都是你学过的知识.

图 16: 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。