计算机系统设计PA3

> 1911533
>
> 朱昱函

实验目的

- 学习系统调用并实现中断机制
- 了解文件系统的基本内容，实现简易的文件系统
- 实验最终以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

实验内容

- 熟悉OS的基本概念、系统调用、实现中断机制
- 进一步完善系统调用、实现简易文件系统
- 输入输出抽象成文件、并运行仙剑奇侠传

PA3.1

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

```
+ AR /home/anjin/ics2017/navy-apps/libs/libc/build/libc-x86.a
make[1]: Leaving directory '/home/anjin/ics2017/navy-apps/libs/libc'
make -C /home/anjin/ics2017/navy-apps/libs/libos
make[1]: Entering directory '/home/anjin/ics2017/navy-apps/libs/libos'
+ CC src/nanos.c
+ AR /home/anjin/ics2017/navy-apps/libs/libos/build/libos-x86.a
make[1]: Leaving directory '/home/anjin/ics2017/navy-apps/libs/libos'
+ CC dummy.c
+ LD /home/anjin/ics2017/navy-apps/tests/dummy/build/dummy-x86
```

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：

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

准备IDT

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寄存器

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

unsigned int cs;
```

```
static inline void restart() {
  /* Set the initial instruction pointer. */
  cpu.eip = ENTRY_START;
  cpu.cs=0x8;
  cpu.eflags.one=0x2;
```

实现lidt指令

```
make_group(gp7,
    EMPTY, EMPTY, EMPTY, EX(lidt),
    EMPTY, EMPTY, EMPTY, EMPTY)
```

```
make_EHelper(lidt) {
  cpu.idtr.limit=vaddr_read(id_dest->addr,2);
  if(decoding.is_operand_size_16)
    cpu.idtr.base=vaddr_read(id_dest->addr+2,4) & 0x00ffffff;
  else
    cpu.idtr.base=vaddr_read(id_dest->addr+2,4);

  print_asm_template1(lidt);
}
```

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

```
/* Uncomment these macros to enable corresponding functionality. */
#define HAS_ASYE
//#define HAS_PTE
```

触发异常

实现int指令

```
/* 0xc8 */  EMPTY, EX(leave), EMPTY, EMPTY,
/* 0xcc */  EMPTY, IDEXW(I,int,1), EMPTY, EMPTY,
```

```
extern void raise_intr(uint8_t NO, vaddr_t ret_addr);
make_EHelper(int) {
  raise_intr(id_dest->val, decoding.seq_eip);
  print_asm("int %s", id_dest->str);
#ifdef DIFF_TEST
  diff_test_skip_nemu();
#endif
}
```

实现raise_intr函数，出发异常之后的硬件处理。需要经过5个步骤帮助CPU从5状态从S状态跳转到新地方，并保存当前状态。

```
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.
   */

  //模拟i386中断机制处理过程
    //1、当前状态压栈
    rtl_push((rtlreg_t *)&cpu.eflags);
    cpu.eflags.IF=0;
    rtl_push((rtlreg_t *)&cpu.cs);
    rtl_push((rtlreg_t *)&ret_addr);

    //2、从idtr中读出首地址
    uint32_t idtr_base = cpu.idtr.base;

    //3、索引，找到门描述符
    uint32_t eip_low, eip_high, offset;
    eip_low = vaddr_read(idtr_base + NO * 8, 4) & 0x0000ffff;
    eip_high = vaddr_read(idtr_base + NO * 8 + 4, 4) & 0xffff0000;

    //4、将门描述符中的offset域组合成目标地址
    offset = eip_low | eip_high;

    //5、跳转到目标地址
    decoding.jmp_eip = offset;
    decoding.is_jmp = 1;
}
```

保存现场

实现pusha，popa

```
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");
}

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");
}
```

实现RegSet：

nemu中栈从高地址向低地址，struct_RegSet 结构中的内容从低地址向高地址延伸，所以先入栈的后声明，后入栈的先声明，重新定义RegSet 结构体的成员，使得这些成员声明的顺序和 nexus-am/am/arch/x86-nemu/src/trap.S 中构造的 trap frame 保持一致

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

事件分发

在 do_event()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;
}
```

系统调用处理

在 nexus-am/am/arch/x86-nemu/include/arch.h 中实现正确的 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

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

添加 SYS_none 系统调用。这个系统调用什么都不用做，直接返回

```
_RegSet* do_syscall(_RegSet *r) {
  uintptr_t a[4], neweax=-1;
  a[0] = SYSCALL_ARG1(r);

  switch (a[0]) {
    case SYS_none:
        neweax = 1;
        break;
    default: panic("Unhandled syscall ID = %d", a[0]);
  }

  return NULL;
}
```

恢复现场

实现iret 指令，恢复 eip，cs，eflags 指令，并修改 decoding 中的跳转 eip 信息

```
  make_EHelper(iret) {
    rtl_pop(&decoding.jmp_eip);
    rtl_pop(&cpu.cs);
    rtl_pop(&cpu.eflags.eflags_init);
    decoding.is_jmp=1;
    print_asm("iret");
  }
```

make run之后hit good trap，PA3.1结束

```
./build/nemu -l /home/anjin/ics2017/nanos-lite/build/nemu-log.txt /home/anjin/ics2017/nanos-lite/build/nanos-lite-x86-nemu.bin
[src/monitor/diff-test/diff-test.c,96,init_difftest] Connect to QEMU successfully
[src/monitor/monitor.c,65,load_img] The image is /home/anjin/ics2017/nanos-lite/build/nanos-lite-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 18:38:46, May  8 2022
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
```

PA3.2

标准输出与堆区管理的实现，运行Hello World

- 为了运行 Hello world 我们需要实现 SYS_write 系统调用。首先需要在 do_syscall 中识别出系统调用号是SYS_write ，之后检查 fd 的值，如果是 1 或 2 则使用 _putc 将以 buf 为首地址的 len 字节输出到串口，最后设置正确的返回值，要返回的内容可以通过 man 2 write 指令查询

```
uintptr_t sys_write(int fd, const voide *buf, size_t count){
    uintptr_t i=0;
    if(fd==1||fd==2){
      for(;count>0;count--){
        _putc(((char*)buf)[i]);
        i++;
      }
    }
}

_RegSet* do_syscall(_RegSet *r) {
  uintptr_t a[4], neweax=-1;
  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:
        neweax = 1;
        break;
    case SYS_exit:
        _halt(a[1]);
        break;
    case SYS_write:
        neweax=sys_write(a[1],(void*)a[2],a[3]);
        break;
    default: panic("Unhandled syscall ID = %d", a[0]);
  }
```

- 在 nanos-lite\src\syscall.c 中添加以上内容之后，还需要在 navy-apps\libs\libos\src\nanos.c 的 _write 调用系统调用接口函数并传入相应参数

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

实现以上两步之后对于输出 Hello world 的系统调用已经基本实现了，但为了最终输出成功还需要进行一些其他操作：

- 切换到 navy-apps/tests/hello/ 目录下执行 make 编译 hello 程序
- 修改 nanos-lite/Makefile 中 ramdisk 的生成规则，把 ramdisk 中的唯一的文件换成 hello 程序

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

最后在 nanos-lite 执行 make update 和 make run 即可实现 Hello world 的输出，此时 Hello world 是逐一输出字符的，即每输出一个字符都会产生一次系统调用

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

实现堆区管理

实现堆区管理主要需要实现两部分，首先要在 nanos-lite 中实现对系统调用号 SYS_brk 的处理，此处我们目前只需要设置返回值为 0 表示堆区调整成功即可：

```
case SYS_brk:
    neweax = 0;
    break;
```

之后在 navy-apps的 _sbrk 中实现相关逻辑并调用系统调用接口函数：首先使用 _end 获得用户程序的 program break 即数据段结束的位置,之后根据传入的参数计算出新的 program break 并将其作为参数进行系统调用，如果系统成功则将原来的 program break 改为增加后的program break 并返回其原来的值，否则返回 -1 ，表示堆区调整失败：

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

```
Hello World for the 118th time
[src/syscall.c,5,sys_write] sys_write:fd 1  len 31

Hello World for the 119th time
[src/syscall.c,5,sys_write] sys_write:fd 1  len 31

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

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

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

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

简易文件系统

对 nanos-lite/Makefile 作如下修改。然后运行 make update 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 nanoslite/src/files.h。

```
.PHONY: update update-ramdisk-objcopy update-ramdisk-fsimg update-fsimg
```

让loader使用文件

为 Finfo 添加一个成员 open_offset（偏移量属性）来记录目前文件操作的位置。每次对文件读写了多少个字节，偏移量就前进多少。

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

- 实现 fs_open()、fs_read()、fs_close()、fs_filesz()函数。
  - **fs_open()**：open 的第一个参数是需要打开的文件名字，需要遍历查找 file_table，如果查找到目标文件则返回文件描述符。
  - **fs_read()**：读文档函数。需要注意不能越过边界，以及使用 ramdisk_read 进行文件读。
  - **fs_filesz()**：read 和 write 的辅助函数，用于返回文件描述符 fd 所描述的文件的大小。直接到数组file_table 中读取。

```
#include "common.h"

#define DEFAULT_ENTRY ((void *)0x4000000)

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

extern ssize_t fs_read(int fd, void *buf, size_t len);
extern size_t fs_filesz(int fd);
extern int fs_open(const char *pathname, int flags, int mode);
extern int fs_close(int fd);

uintptr_t loader(_Protect *as, const char *filename) {
    //功能：将ramdisk中从0开始的所有内容放置到0x4000000,并返回
    /*size_t len = get_ramdisk_size();
    ramdisk_read(DEFAULT_ENTRY,0,len);
    return (uintptr_t)DEFAULT_ENTRY;*/
    int fd = fs_open(filename, 0, 0);           //打开文件
    printf("fd = %d\n", fd);
    fs_read(fd, DEFAULT_ENTRY, fs_filesz(fd));  //读文件
    fs_close(fd);                               //关闭文件
    return (uintptr_t)DEFAULT_ENTRY;
}
```

```c
void init_fs() {
  // TODO: initialize the size of /dev/fb
  file_table[FD_FB].size = _screen.height * _screen.width * 4;
}

int fs_open(const char *pathname, int flags, int mode) {
  Log("pathname %s\n", pathname);
  for (int i = 0; i < NR_FILES; i++) {
    //printf("file name: %s\n", file_table[i].name);
    if (strcmp(file_table[i].name, pathname) == 0)
      //找到了目标文件
      return i; //返回文件描述符
  }

  //没找到目标文件
  assert(0);
  return -1;
}
```

```c
ssize_t fs_read(int fd, void *buf, size_t len) {
  ssize_t fs_size = fs_filesz(fd);
  //Log("in the read, fd = %d, file size = %d, len = %d, file open_offset = %d\n", fd, fs_size, len, file_table[fd].open_offset);

  switch(fd) {
    case FD_STDOUT:
    case FD_FB:
      break;
    case FD_EVENTS:
      len = events_read((void *)buf, len);
      break;
    case FD_DISPINFO:
      if (file_table[fd].open_offset >= file_table[fd].size)
        return 0;
      if (file_table[fd].open_offset + len > file_table[fd].size)
        len = file_table[fd].size - file_table[fd].open_offset;
      dispinfo_read(buf, file_table[fd].open_offset, len);
      file_table[fd].open_offset += len;
      break;
    default:
      if(file_table[fd].open_offset >= fs_size || len == 0)
        return 0;
      if(file_table[fd].open_offset + len > fs_size)
        len = fs_size - file_table[fd].open_offset;
      ramdisk_read(buf, file_table[fd].disk_offset + file_table[fd].open_offset, len);
      file_table[fd].open_offset += len;
      break;
  }
  return len;
}
```

```c
int fs_close(int fd) {
  return 0;
}

size_t fs_filesz(int fd) {
  return file_table[fd].size;
}

ssize_t fs_write(int fd, const void *buf, size_t len) {
  ssize_t fs_size = fs_filesz(fd);
  switch(fd) {
    case FD_STDOUT:
    case FD_STDERR:
      for(int i = 0; i < len; i++) {
          _putc(((char*)buf)[i]);
      }
      break;
    case FD_FB:
      fb_write(buf, file_table[fd].open_offset, len);
      file_table[fd].open_offset += len;
      break;
    default:
      if(file_table[fd].open_offset >= fs_size)
        return 0;
      if(file_table[fd].open_offset + len > fs_size)
        len = fs_size - file_table[fd].open_offset;
      ramdisk_write(buf, file_table[fd].disk_offset + file_table[fd].open_offset, len);
      file_table[fd].open_offset += len;
      break;
  }
  return len;
}
```

```c
off_t fs_lseek(int fd, off_t offset, int whence) {
  off_t result = -1;

  switch(whence) {
    case SEEK_SET:
      if (offset >= 0 && offset <= file_table[fd].size) {
        file_table[fd].open_offset = offset;
        result = file_table[fd].open_offset = offset;
      }
      break;
    case SEEK_CUR:
      if ((offset + file_table[fd].open_offset >= 0) &&
          (offset + file_table[fd].open_offset <= file_table[fd].size)) {
        file_table[fd].open_offset += offset;
        result = file_table[fd].open_offset;
      }
      break;
    case SEEK_END:
      file_table[fd].open_offset = file_table[fd].size + offset;
      result = file_table[fd].open_offset;
      break;
  }

  return result;
}
```

完善 sys_call.c 和 nanos.c 的各个函数和系统调用。

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

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

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

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

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

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

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

运行bin/text **HIT GOOD TRAP**

```
(nemu) c
[src/main.c,19,main] 'Hello World!' from Nanos-lite
[src/main.c,20,main] Build time: 18:43:07, May 10 2022
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x1017c0, end = 0x36e8ca, size = 2543882 bytes
[src/main.c,27,main] Initializing interrupt/exception handler...
[src/fs.c,39,fs_open] pathname /bin/text

fd = 10
loader end!
[src/fs.c,39,fs_open] pathname /share/texts/num

PASS!!!
nemu: HIT GOOD TRAP at eip = 0x00100032

(nemu)
```

PA3.3

把 VGA 显存抽象成文件

在 pa 中，显存被抽象成文件 /dev/fb ，根据实验指导书，具体步骤如下：

- **在 init_fs中对文件记录表中 /dev/fb 的大小进行初始化，使用 IOE 定义的 API 来获取屏幕的大小：**
- IOE 定义的 API 在 nexus-am\am\arch\x86-nemu\src\ioe.c 中，在 init_fs 中我们需要设置 File_table[FD_FB] 的 size ，具体应为显示屏幕大小的图形所需要的字节数，即屏幕长度宽度的乘积再乘以32 位数据的大小：

```
void init_fs() {
  // TODO: initialize the size of /dev/fb
  file_table[FD_FB].size = _screen.height * _screen.width * 4;
}
```

实现 fb_write()，用于把 buf 中的 len 字节写到屏幕上 offset 处。需要先从 offset 计算出屏幕上的坐标，然后调用 IOE 的_draw_rect()接口

```
void fb_write(const void *buf, off_t offset, size_t len) {
    int x, y;
    int len1, len2, len3;
    offset = offset >> 2;
    y = offset / _screen.width;
    x = offset % _screen.width;

    len = len >> 2;
    len1 = len2 = len3 = 0;

    len1 = len <= _screen.width - x ? len : _screen.width - x;
    _draw_rect((uint32_t *)buf, x, y, len1, 1);

    if (len > len1 && ((len - len1) > _screen.width))   {
        len2 = len - len1;
        _draw_rect((uint32_t *)buf + len1, 0, y + 1, _screen.width, len2 / _screen.width);
    }

    if (len - len1 - len2 > 0) {
        len3 = len - len1 - len2;
        _draw_rect((uint32_t *)buf + len1 + len2, 0, y + len2 / _screen.width + 1, len3, 1);
    }
}
```

在 init_device()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
  strcpy(dispinfo, "WIDTH:400\nHEIGHT:300");
}
```

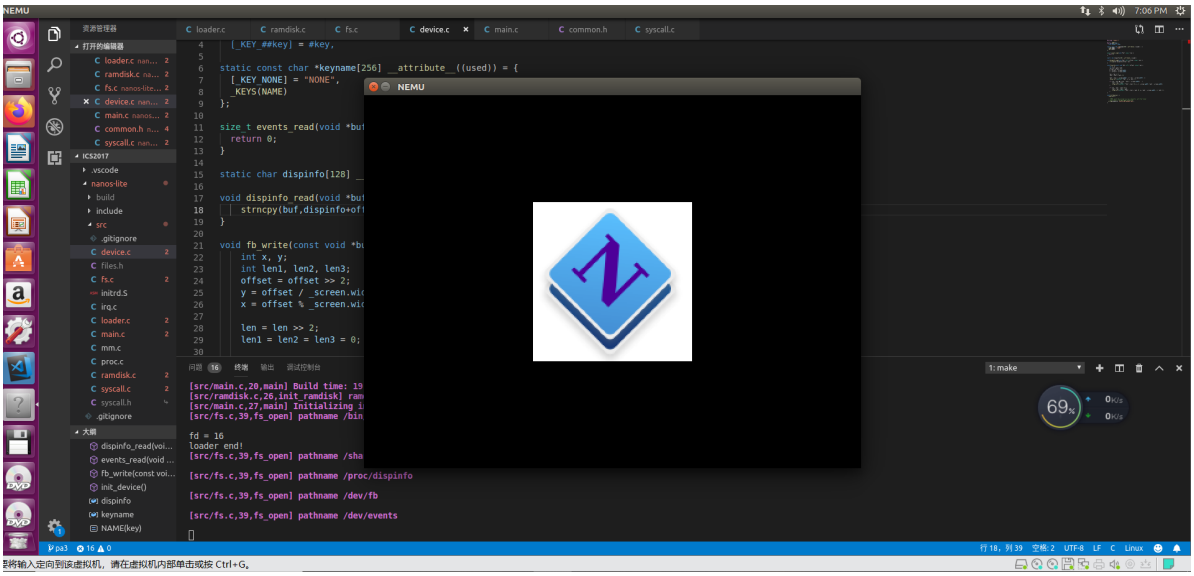
在 fs_read()的实现中对 FD_DISPINFO 进行"重定向"。

```
    break;
  case FD_DISPINFO:
    if (file_table[fd].open_offset >= file_table[fd].size)
      return 0;
    if (file_table[fd].open_offset + len > file_table[fd].size)
      len = file_table[fd].size - file_table[fd].open_offset;
    dispinfo_read(buf, file_table[fd].open_offset, len);
    file_table[fd].open_offset += len;
    break;
```

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

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

运行显示NEMU LOGO，说明没有问题



把设备输入抽象成文件

现在要把设备输入抽象成文件。实现 events_read()，把事件写入到 buf 中，最长写入 len 字节，并返回写入的实际长度。需要借助 IOE 的 API 来获得设备的输入。

```
size_t events_read(void *buf, size_t len) {
  //return 0;
  int key = _read_key();
  bool down = false;
  if(key & 0x0800){
    key^=0x0800;
    down=true;
  }
  if(key==_KEY_NONE){
    unsigned long t = _uptime();
    sprintf(buf,"t %d\n",t);
  }
  else{
    sprintf(buf, "%s %s\n", down ? "kd" : "ku", keyname[key]);
  }
  return strlen(buf);
}
```

在文件系统的 fs_read 函数中添加对/dev/events 的支持。

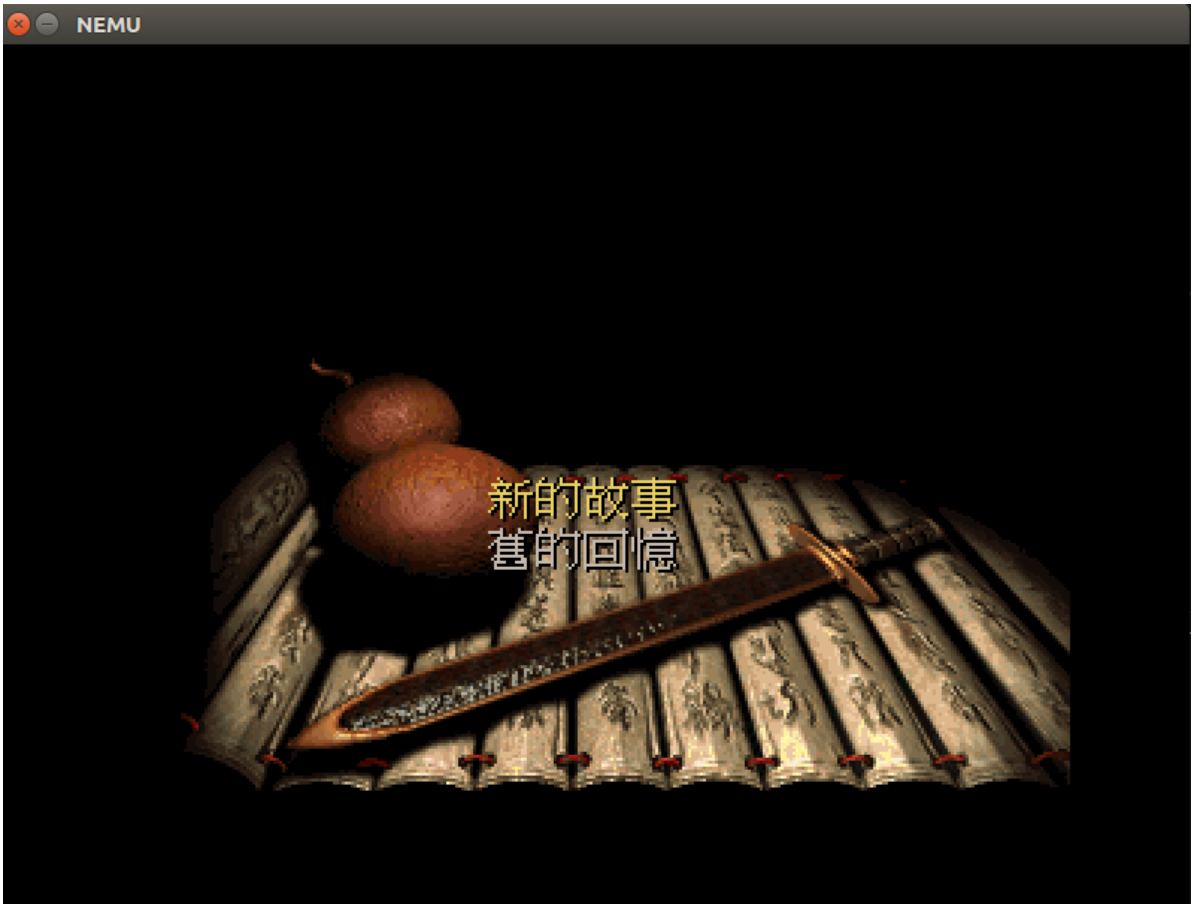
```
      break;
    case FD_EVENTS:
      len = events_read((void *)buf, len);
      break;
```

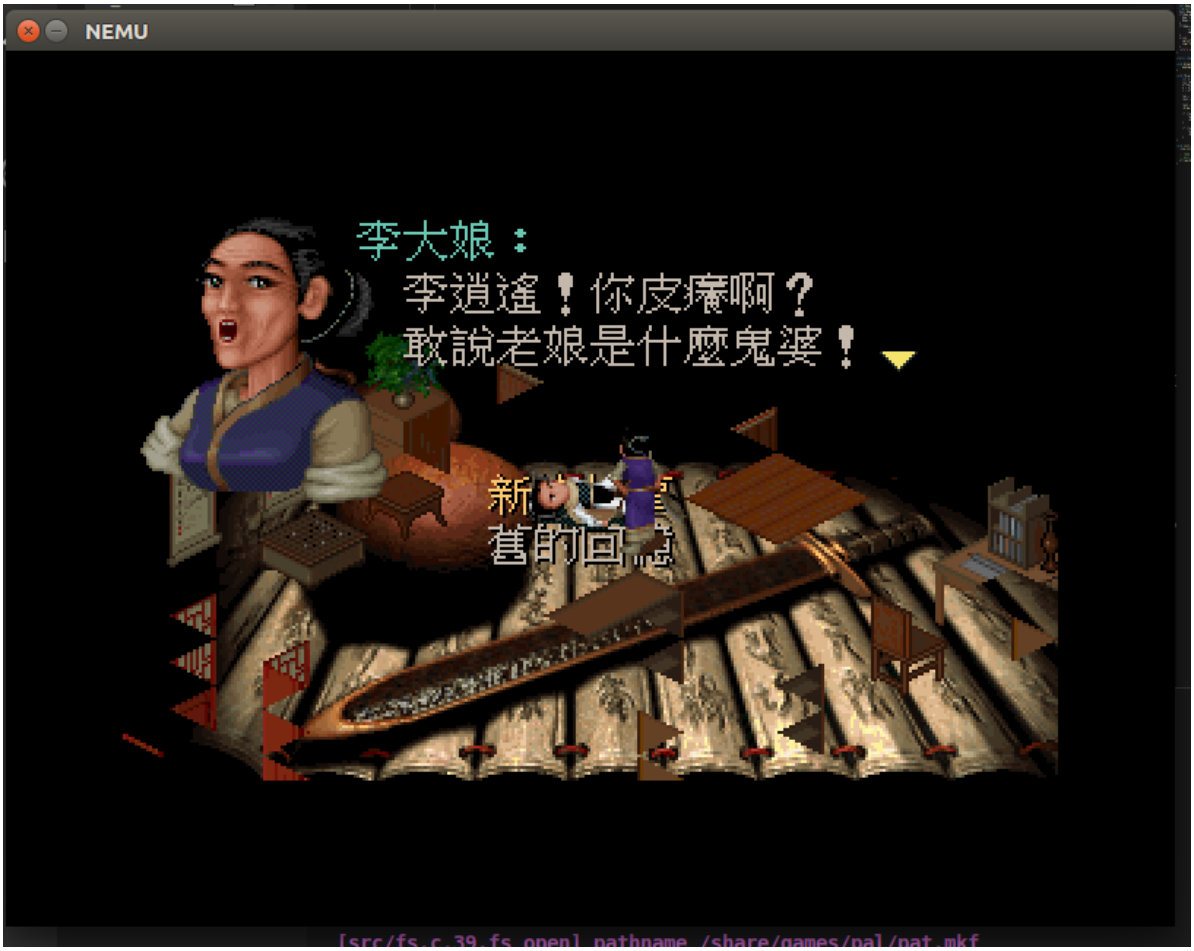
修改 loader 加载 /bin/events ， 程序输出时间事件的信息，敲击按键时会输出按键事件的信息

```
loader end!
[src/fs.c,39,fs_open] pathname /dev/events

receive event: t 242
receive event: t 414
jfreceive event: t 587
eiworeceive event: t 771
jfeireceive event: t 943
receive event: t 1110
dncreceive event: t 1284
ncreceive event: t 1457
creceive event: t 1626
nureceive event: t 1796
nuireceive event: t 1969
ewreceive event: t 2139
freceive event: t 2309
```

运行仙剑奇侠传





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。

加载存档

游戏存档可以以文件的方式储存在 navy-apps 下，在游戏读档时客户仙剑作为客户程序，在程序代码中使用的文件库函数，读文件库函数请求读文件的系统调用， Nanos-lite 响应该调用，其下一层的 NEMU 为其提供模拟出的硬件支持。

更新屏幕

在 redraw()函数中先用 palette 给 fb 对应元素赋值。Palette 是 256 色调色板，fb 和 vmen 都是 size为 W*H 的数组。将 fb 作为第一个参数传入 NDL_DrawRect 中。

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

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

在 ndl.c 中实现了 NDL_DrawRect 函数。这个 c 文件包含了<stdio.h>头文件，函数中的 printf、putchar 和 fwrite 都出自该头文件。在调用他们时都会像上文描述的 fread 一样陷入操作系统内核态然后进行一系列相关操作。

```
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];
      }
    }
  }
}
```

仙剑属于应用程序，在 PA 实验中，它直接运行在操作系统上（我们把 pal 放在了 Navy-apps 中而不是 nexus-am 中）。在调用库函数 fread、fwrite 等时，标准 IO 操作会陷入 OS 内核进行操作（在普通计算机上，大量的存档资料应当存储于硬盘而不是内存中，故需要陷入而不只是读写缓存），也就是 PA中的轻量级操作系统 Nanos-lite 中进行文件读写操作，而 libos 中的 Nanos.c 中提供了 PA3 中系统调用的接口（sys_call 函数），在这里根据不同的系统调用号进行不同的中断操作。在 fwrite 时，内核操作中会调用 AM 的 IOE 接口进行屏幕信息更新，NEMU 提供硬件支持。

遇到的问题和解决方法

- git出现了诸如此类错误

```
error: object file .git/objects/3a/60046cdd45cf3e943d1294b3cb251a63fb9552 is
empty
error: object file .git/objects/3a/60046cdd45cf3e943d1294b3cb251a63fb9552 is
empty
fatal: loose object 3a60046cdd45cf3e943d1294b3cb251a63fb9552 (stored
in.git/objects/3a/60046cdd45cf3e943d1294b3cb251a63fb9552) is corrupt
```

应该是由于虚拟机异常关机导致git的head指针丢失所致，由于急于完成作业，将原来的git文件备份了出来，然后暴力git init，因此提交时会提交三份git记录（运气差的时候就是极致的非...orz），但大部分的git记录都是保留的。

- 实现pa3.2时始终是单个字母输出，无法输出完整句子。

经过网上查阅资料后发现每次都需要重新去文件目录make一下，然后再make update最后make run即可解决。

- pa3.3时检测不到键盘输入

I/O大法调试了好久然后发现把0x8000写成了0x0800.....