

计算机系统设计

P32 实验报告

专业：计算机科学与技术

学号：2013210

姓名：郑凌伟

一、实验目的

实现一个足够简单的操作系统，来支持游戏“仙剑奇侠传”的运行

二、实验内容

阶段 1: 完成

三、阶段一

首先我们按照要求实现 loader，就是利用 ramdisk 提供的函数即可

```
1.  #include "common.h"
2.
3.  #define DEFAULT_ENTRY ((void *)0x4000000)
4.
5.  extern void ramdisk_read(void *buf, off_t offset, size_t len);
6.
7.  extern size_t get_ramdisk_size();
8.
9.  uintptr_t loader(_Protect *as, const char *filename) {
10.    ramdisk_read((void*)0x4000000, 0, get_ramdisk_size());
11.    return (uintptr_t)DEFAULT_ENTRY;
12. }
```

接下来就是实现中断机制了,首先需要实现 lidt 和 int 指令,将 main.c 中的宏定义解除注释,然后在 decode 中加入对 int 的支持并完成 idtr 结构体的构建.其中比较值得注意的是 intr.c 的实现,这就是我们对于中断的处理,具体来说,它的参数有两个: NO 是中断号,标识出要触发的特定中断或异常; ret_addr 是返回地址,也就是中断处理程序完成后,程序应该继续执行的地址。然后它完成了如下步骤:

1. 首先,保存当前的处理器状态。这包括标志寄存器 (cpu.eflags), 代码段寄存器 (cpu.cs) 和返回地址 (ret_addr)。这些都被压入栈,以便在中断处理程序完成后可以恢复。

2. 然后,计算中断描述符表 (IDT) 中对应中断号的中断门的地址。中断描述符表是一个表格,存储着每个中断号对应的中断门的信息。中断门的信息分成两个部分,分别在 temp1

和 temp2 中。

3. 接着，从 temp1 和 temp2 中提取出中断处理程序的起始地址 (jumptarget)。

4. 最后，设置 decoding.is_jump 为 1，表示要进行跳转，然后设置 decoding.jump_eip 为中断处理程序的起始地址。这样，下一步执行的就是中断处理程序了。

```
1. void raise_intr(uint8_t NO, vaddr_t ret_addr) {
2.     /* TODO: Trigger an interrupt/exception with ``NO``.
3.      * That is, use ``NO`` to index the IDT.
4.      */
5.     uint32_t temp1;
6.     uint32_t temp2;
7.     vaddr_t jumptarget;
8.
9.     t0 = cpu.eflags;
10.    rtl_push(&t0);
11.    t0 = cpu.cs;
12.    rtl_push(&t0);
13.    t0 = ret_addr;
14.    rtl_push(&t0);
15.
16.    temp1 = vaddr_read(cpu.idtr.base + 8 * NO, 4);
17.    temp2 = vaddr_read(cpu.idtr.base + 8 * NO + 4, 4);
18.    jumptarget = ((temp1 & 0x0000FFFF) | (temp2 & 0xFFFF0000));
19.
20.    decoding.is_jump = 1;
21.    decoding.jump_eip = jumptarget;
22. }
```

之后我们需要实现 pusha 命令并重新租着 TrapFrame 结构体。

我们需要先填充 iret 指令的内容,然后完成内容:

1. rtl_pop(&t0);: 从栈中弹出一个值，存储到临时变量 t0 中。这个值是中断发生前的程序计数器值，也就是中断发生前处理器将要执行的下一个指令的地址。

2. decoding.jump_eip = t0; 和 decoding.is_jump = 1;: 设置程序计数器为 t0，并标记要进行跳转。这样在 iret 指令执行完毕后，处理器将会继续执行 t0 指向的指令。

3. rtl_pop(&t0); 和 cpu.cs = t0;: 从栈中弹出一个值，存储到代码段寄存器中。代码段寄存器是用来存储当前执行的代码的内存区段的基址的。

4. rtl_pop(&t0); 和 cpu.eflags = t0;: 从栈中弹出一个值，存储到标志寄存器中。标志寄存器是用来存储处理器状态的寄存器，如进位标志、零标志、符号标志等。

```
1. make_EHelper(iret) {
```

```

2.     rtl_pop(&t0);
3.     decoding.jump_eip = t0;
4.     decoding.is_jump = 1;
5.     rtl_pop(&t0);
6.     cpu.cs = t0;
7.     rtl_pop(&t0);
8.     cpu.eflags = t0;
9.     print_asm("iret");
10. }

```

这是当前阶段完成的效果

```

+ LD build/nemu
./build/nemu -l /home/leo/SystemDesign/nemu_2017/nanos-lite/build/nemu-log.txt /
home/leo/SystemDesign/nemu_2017/nanos-lite/build/nanos-lite-x86-nemu.bin
[src/monitor/monitor.c,65,load_img] The image is /home/leo/SystemDesign/nemu_201
7/nanos-lite/build/nanos-lite-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 21:09:00, May 15 2023
For help, type "help"
(nemu) c
[src/main.c,19,main] 'Hello World!' from Nanos-lite
[src/main.c,20,main] Build time: 21:09:00, May 15 2023
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x100cec, end = 0x1052c8,
size = 17884 bytes
invalid opcode(eip = 0x04001f98): cd 80 5b 5d c3 66 90 90 ...

There are two cases which will trigger this unexpected exception:
1. The instruction at eip = 0x04001f98 is not implemented.
2. Something is implemented incorrectly.
Find this eip(0x04001f98) in the disassembling result to distinguish which case
it is.

If it is the first case, see

```

在第一阶段的最后我们需要实现系统调用,按照实验指导书的步骤,首先是在 do_event()中识别出系统调用事件_EVENT_SYSCALL, 然后调用 do_syscall().

```

1.     #include "common.h"
2.
3.     _RegSet* do_syscall(_RegSet *r);
4.
5.     static _RegSet* do_event(_Event e, _RegSet* r) {
6.         switch (e.event) {
7.             case _EVENT_SYSCALL: return do_syscall(r);
8.             default: panic("Unhandled event ID = %d", e.event);
9.         }
10.
11.     return NULL;
12. }
13.
14. void init_irq(void) {

```

```
15.     _asye_init(do_event);
16. }
```

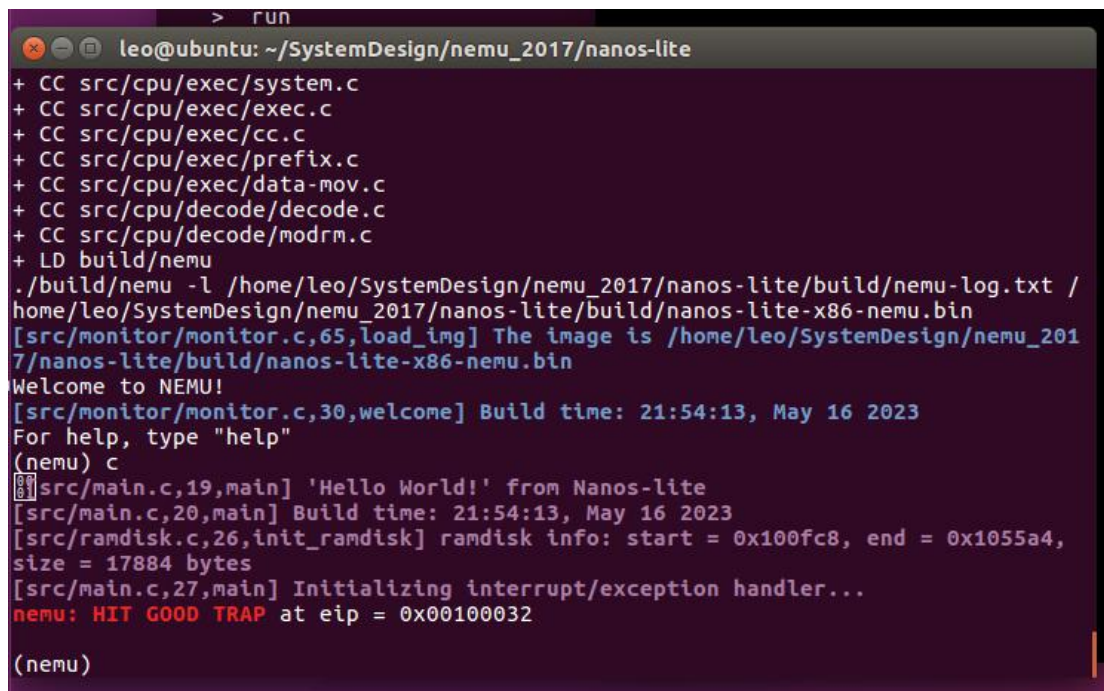
它接收一个事件 `e` 和寄存器集 `r` 作为参数。这个函数首先通过一个 `switch` 语句判断事件的类型。如果事件是一个系统调用事件（即 `_EVENT_SYSCALL`），那么它就调用 `do_syscall` 函数处理这个事件，并返回处理结果。如果事件的类型是其他未处理的类型，那么这个函数就会触发一个 `panic`，表示发生了一个未处理的事件。

接着在 `arch` 中设置对应的宏，在修改前，`SYSCALL_ARGn(r)`宏始终返回 0，这意味着系统调用的参数无法正确获取。在修改后，这些宏被重新定义以返回对应的寄存器值（`eax`，`ebx`，`ecx` 和 `edx`），使得系统调用可以接收到正确的参数。

然后在 `main` 中解除 `ASYE` 的宏限制，另一个比较关键的点就是 `raise_intr`，具体实现如下功能

1. 首先，检查中断向量号（`NO`）是否超过中断描述符表（`IDT`）的限制。如果超过，则触发断言错误。
2. 之后，把 `cpu.eflags`，`cpu.cs` 以及 `ret_addr` 压入堆栈。这些值保存了发生中断前的状态，以便于在中断处理完成后能恢复到中断发生前的状态。
3. 计算中断描述符在 `IDT` 中的地址，然后读取中断描述符的低 32 位和高 32 位。中断描述符的具体结构可以在 `i386` 手册中找到，这里简单地将低 16 位和高 16 位组合起来，形成了中断服务例程的地址。
4. 最后，设置跳转地址为中断服务例程的地址，并标记为需要跳转。

修改完成之后，我们再次运行，即可看到我们的程序正确运行并 `hit good trap` 了



```
> run
leo@ubuntu: ~/SystemDesign/nemu_2017/nanos-lite
+ CC src/cpu/exec/system.c
+ CC src/cpu/exec/exec.c
+ CC src/cpu/exec/cc.c
+ CC src/cpu/exec/prefix.c
+ CC src/cpu/exec/data-mov.c
+ CC src/cpu/decode/decode.c
+ CC src/cpu/decode/modrm.c
+ LD build/nemu
./build/nemu -l /home/leo/SystemDesign/nemu_2017/nanos-lite/build/nemu-log.txt /
home/leo/SystemDesign/nemu_2017/nanos-lite/build/nanos-lite-x86-nemu.bin
[src/monitor/monitor.c,65,load_img] The image is /home/leo/SystemDesign/nemu_201
7/nanos-lite/build/nanos-lite-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 21:54:13, May 16 2023
For help, type "help"
(nemu) c
[src/main.c,19,main] 'Hello World!' from Nanos-lite
[src/main.c,20,main] Build time: 21:54:13, May 16 2023
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x100fc8, end = 0x1055a4,
size = 17884 bytes
[src/main.c,27,main] Initializing interrupt/exception handler...
nemu: HIT GOOD TRAP at eip = 0x00100032
(nemu)
```

四、 阶段二

在第二阶段,我们按照实验指导书首先要完成 write 的系统调用并运行 helloworld 小程序. 首先修改 makefile 指向 helloworld,然后就是在 syscall.c 中实现 sys_write 函数,关键主体如下所示:

```
1.  static inline _RegSet* sys_write(_RegSet *r) {
2.      int fd = (int)SYSCALL_ARG2(r);
3.      const char *buf = (const char *)SYSCALL_ARG3(r);
4.      size_t count = (size_t)SYSCALL_ARG4(r);
5.      if (fd != 1 && fd != 2)
6.          TODO();
7.      int i;
8.      for (i = 0; i < count; ++i)
9.          _putc(buf[i]);
10.     SYSCALL_ARG1(r) = i + 1;
11.     return NULL;
12. }
```

在这个函数中,它首先从寄存器中获取参数,包括文件描述符 (fd),写入的数据的地址 (buf) 和数据的长度 (count)。然后,它检查 fd 是否为 1 或 2,这两个值分别表示标准输出和标准错误。如果 fd 是其他值,那么函数暂时无法处理,因此调用 TODO()。

接下来,函数使用一个循环将 buf 指向的数据逐个字符地写入到输出中,这是通过调用 _putc 函数实现的。最后,它将写入的字符数加 1 赋值给 SYSCALL_ARG1(r) (即 eax 寄存器),这是 write 系统调用的返回值,表示写入的字符数。

至此 sys_write 就实现好了。


```
leo@ubuntu: ~/SystemDesign/nemu_2017/nanos-lite
make[1]: *** No targets specified and no makefile found. Stop.
make[1]: Leaving directory '/home/leo/SystemDesign/nemu_2017/nexus-am/libs/klib'
/home/leo/SystemDesign/nemu_2017/nexus-am/Makefile.compile:86: recipe for target
'klib' failed
make: [klib] Error 2 (ignored)
make[1]: Entering directory '/home/leo/SystemDesign/nemu_2017/nemu'
./build/nemu -l /home/leo/SystemDesign/nemu_2017/nanos-lite/build/nemu-log.txt /
home/leo/SystemDesign/nemu_2017/nanos-lite/build/nanos-lite-x86-nemu.bin
[src/monitor/monitor.c,65,load_img] The image is /home/leo/SystemDesign/nemu_201
7/nanos-lite/build/nanos-lite-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 21:54:13, May 16 2023
For help, type "help"
(nemu) c
[src/main.c,19,main] 'Hello World!' from Nanos-lite
[src/main.c,20,main] Build time: 21:54:13, May 16 2023
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x1010a0, end = 0x10577c,
size = 18140 bytes
[src/main.c,27,main] Initializing interrupt/exception handler...
Hello World!
Hello World for the 2th time
Hello World for the 3th time
```

紧接着我们需要实现堆区管理,具体来说在 Nanos-lite 中实现 SYS_brk 系统调用,然后在用户层实现_sbrk().

我们先完成 sys_brk,在 syscall 中首先将 SYSCALL_ARG2(r)的值赋给了_heap.end, 这是改变数据段大小的操作。然后, 它将 0 赋值给 SYSCALL_ARG1(r), 这是 brk 系统调用的返回值, 表示操作成功

```
1. static inline _RegSet* sys_brk(_RegSet *r) {
2.     _heap.end = (void *)SYSCALL_ARG2(r);
3.     SYSCALL_ARG1(r) = 0;
4.     return NULL;
5. }
```

然后是 sbrk 的实现,首先, 代码引入了一个全局变量 program_break, 该变量初始化为 end 的地址。end 是一个链接时定义的符号, 表示程序数据段的末尾。

_sbrk 函数通过 _syscall_ 函数发送 SYS_brk 系统调用来改变程序的数据段大小。_syscall_ 的第二个参数为新的 program_break 的地址。

如果 _syscall_ 返回值为 0, 表示系统调用成功, 那么更新 program_break, 并返回原来的 program_break。否则返回 (void *)-1, 表示分配内存失败。

```
1. extern char end;
2. intptr_t program_break = (intptr_t)&end;
3.
4. void *_sbrk(intptr_t increment){
5.     // char buf[100];
6.     // sprintf(buf, "%x\n", &end);
7.     // write(1, buf, strlen(buf));
8.     intptr_t old_program_break = program_break;
9.     intptr_t addr = program_break + increment;
10.
```

```

11.     if(_syscall_(SYS_brk, addr, 0, 0) != 0)
12.         return (void *)-1;
13.     program_break = addr;
14.     return (void *)old_program_break;
15. }

```

接着就是实现完整的程序系统了,代码难度主要体现在 fs.c,具体来说我们分门别类的实现了指导书中的若干 fs 函数:

Finfo 结构体被增加了一个 open_offset 成员,用于记录文件的当前读写位置。

fs_open 函数通过遍历 file_table 来查找文件名对应的文件描述符 (fd)。如果找到,就将文件的 open_offset 重置为 0,并返回该文件描述符。

fs_read 函数通过文件描述符 (fd) 从文件的当前位置 (由 open_offset 决定) 开始读取指定长度的数据。如果文件剩余的数据长度不足,那么只读取剩余的数据。读取数据后,open_offset 要增加相应的长度。函数返回实际读取的数据长度。

fs_write 函数和 fs_read 类似,只是它是将数据写入文件。如果文件的剩余空间不足,那么只写入剩余的空间。写入数据后,open_offset 也要增加相应的长度。函数返回实际写入的数据长度。

fs_lseek 函数用于改变文件的当前读写位置。它有三种模式: SEEK_SET 模式将读写位置设置为文件开头的偏移位置; SEEK_CUR 模式将读写位置增加一个偏移量; SEEK_END 模式将读写位置设置为文件末尾加上一个偏移量。

fs_close 函数在这里并没有实现任何功能,只是返回 0 表示成功。

fs_filesz 函数返回指定文件的大小。

实现了文件系统以后,就要让 loader 使用文件系统了,原始版本的加载器直接从 RAM 磁盘中读取整个可执行文件到默认入口地址 (DEFAULT_ENTRY)。RAM 磁盘的读取通过 ramdisk_read 实现,并且整个 RAM 磁盘的大小是通过 get_ramdisk_size 获取的。

修改后的版本改为从文件系统中读取文件。加载器首先打开文件,然后读取文件的全部内容到默认入口地址,最后关闭文件。这些操作分别通过 fs_open, fs_read 和 fs_close 实现。文件的大小是通过 fs_filesz 获取的。

然后修改 syscall,SYS_open 调用会打开一个文件,返回一个文件描述符 (fd)。这个调用需要三个参数: 文件路径 (pathname), 打开方式 (flags), 和权限模式 (mode)。

SYS_read 调用会从一个已打开的文件 (通过文件描述符指定) 中读取数据。这个调用需要三个参数: 文件描述符 (fd), 读取数据的缓冲区 (buf), 和读取的字节数 (len)。

SYS_close 调用会关闭一个已打开的文件。这个调用需要一个参数: 文件描述符 (fd)。

SYS_lseek 调用会改变文件的当前读/写位置。这个调用需要三个参数: 文件描述符 (fd), 偏移量 (offset), 和起始位置 (whence)。起始位置可以是文件开始 (SEEK_SET), 当前位置 (SEEK_CUR), 或文件结束 (SEEK_END)。

最后可以看到我们的程序 pass 了样例

```
leo@ubuntu: ~/SystemDesign/nemu_2017/nanos-lite
make[1]: *** No targets specified and no makefile found. Stop.
make[1]: Leaving directory '/home/leo/SystemDesign/nemu_2017/nexus-am/libs/klib'
/home/leo/SystemDesign/nemu_2017/nexus-am/Makefile.compile:86: recipe for target
'klib' failed
make: [klib] Error 2 (ignored)
make[1]: Entering directory '/home/leo/SystemDesign/nemu_2017/nemu'
./build/nemu -l /home/leo/SystemDesign/nemu_2017/nanos-lite/build/nemu-log.txt /
home/leo/SystemDesign/nemu_2017/nanos-lite/build/nanos-lite-x86-nemu.bin
[src/monitor/monitor.c,65,load_img] The image is /home/leo/SystemDesign/nemu_201
7/nanos-lite/build/nanos-lite-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 02:05:45, May 17 2023
For help, type "help"
(nemu) c
[src/main.c,19,main] 'Hello World!' from Nanos-lite
[src/main.c,20,main] Build time: 02:03:24, May 17 2023
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x1019b0, end = 0x3703ba,
size = 2550282 bytes
[src/main.c,27,main] Initializing interrupt/exception handler...
PASS!!!
nemu: HIT GOOD TRAP at eip = 0x00100032
(nemu)
```

五、 阶段三

第三阶段是完善文件系统,首先是将 VGA 显存抽象成文件,我们从 device 开始入手。
dispinfo_read 函数用于读取显示设备的信息。修改后的版本使用 memcpy 函数,将 dispinfo 数组从 offset 开始的 len 字节的数据复制到 buf 中。

fb_write 函数用于将图形数据写入帧缓冲区 (frame buffer), 以在屏幕上显示。修改后的版本首先确认 offset 和 len 是 4 的倍数, 然后计算出写入的起始位置 (x 和 y), 最后调用 _draw_rect 函数将图形数据写入帧缓冲区。

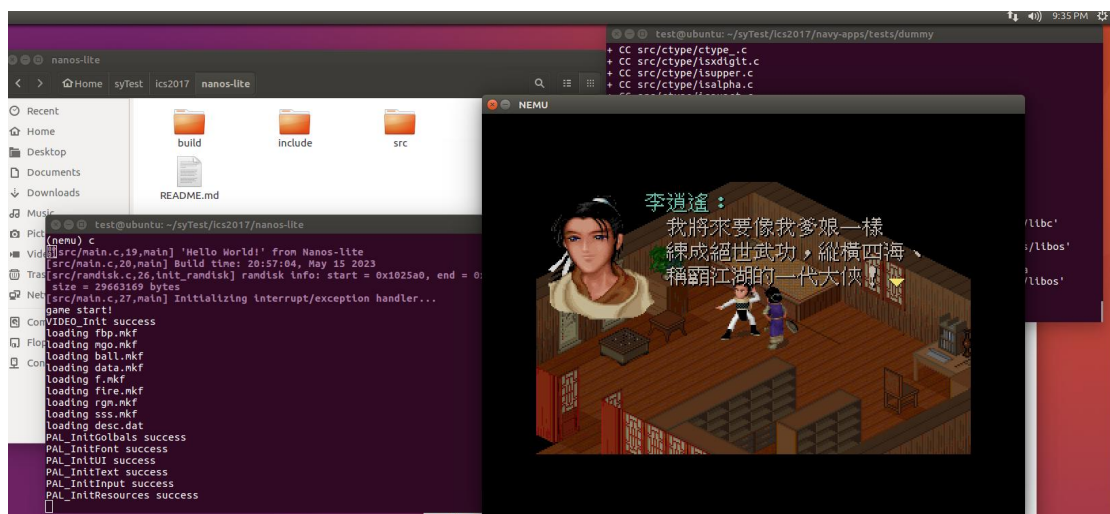
init_device 函数用于初始化设备。修改后的版本首先调用 _ioe_init 函数初始化输入输出设备, 然后使用 sprintf 函数将屏幕的宽度和高度写入 dispinfo 数组。

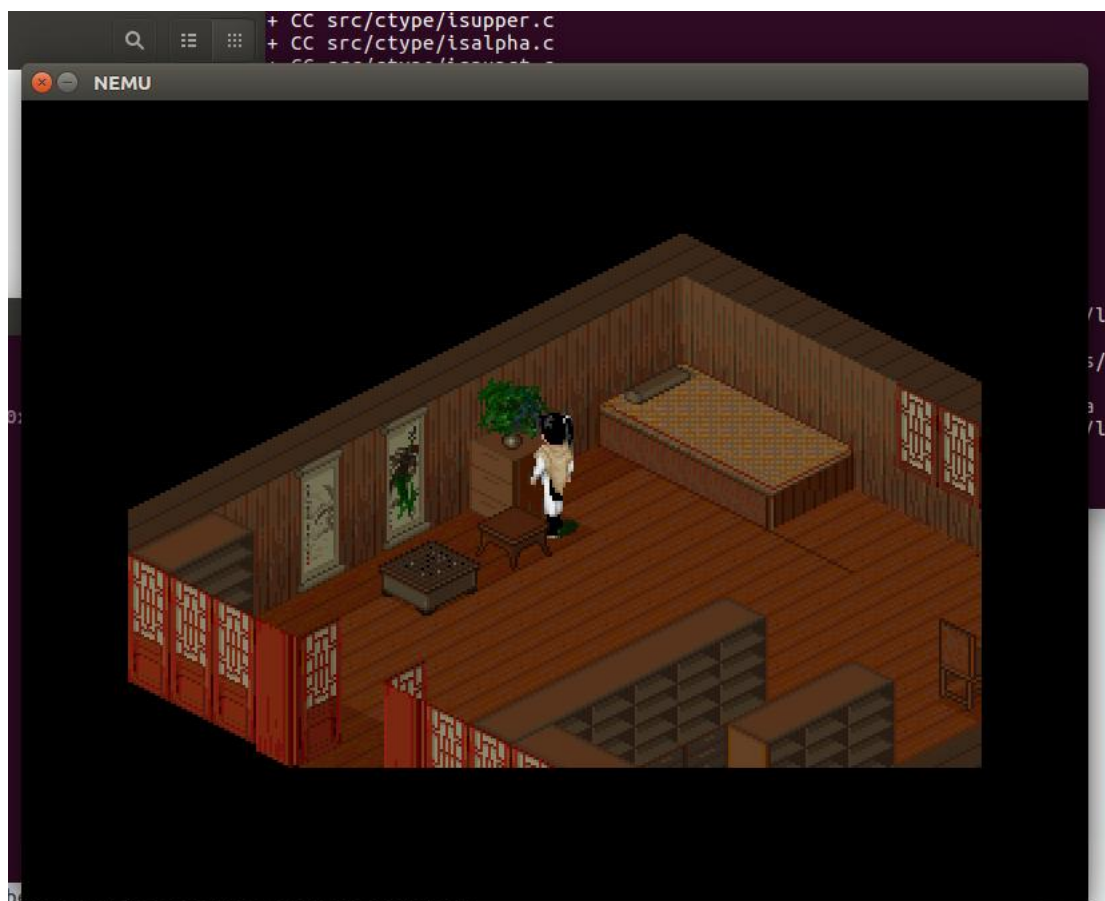
接着我们来完善 fs 文件系统,完成了如下功能: init_fs 函数中, 使用 _screen.width * _screen.height * sizeof(uint32_t) 的方式来计算并初始化 /dev/fb 的大小。这个大小应该等于屏幕的像素数乘以每个像素的字节数。

fs_read 函数中, 如果文件描述符是 FD_DISPINFO, 则调用 dispinfo_read 函数读取显示设备信息, 否则调用 ramdisk_read 函数读取普通文件的内容。

fs_write 函数中, 如果文件描述符是 FD_FB, 则调用 fb_write 函数将图形数据写入帧缓冲区, 否则调用 ramdisk_write 函数写入普通文件的内容。

之后我们还对一些细枝末节的代码进行了修改。
最后就是激动人心的运行了“仙剑奇侠传”:





六、 难点和总结

本次 ICS2017 PA3 实验主要涉及异常控制流和简易文件系统的实现，需要通过 `int 0x80` 中断机制将 NEMU 的实现，AM 实现，操作系统的实现串联起来。其中，涉及的问题包括符号扩展、操作数长度，以及通过系统调用实现的打印函数等。此外，调试过程中可能会遇到较多的 bug，需要仔细分析和解决。实验过程中，我加深了对于系统调用生命周期的理解，例如，通过 `navy-apps/tests/hello` 例子来理解从 `printf` 到系统调用 `_write` 的过程。此外，还更加认识到理解 NEMU 和 AM 如何协同工作，以及在不同情况下 `printf` 的实现。

七、 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。

答:对于在仙剑奇侠传中涉及的文件读取和屏幕更新,这两个过程都涉及到多个部分的协同工作。在 navy-apps/apps/pal/src/global/global.c 的 PAL_LoadGame()函数中,数据的加载过程最终都依赖于 fread()函数,这是一个在 navy-apps/libs/libc/src/stdio/fread.c 中实现的 C 语言库函数。在 fread()函数中,通过调用 memcpy()函数来读取文件内容,而文件的内容在调用 fopen()函数时进行处理。

在 navy-apps/apps/pal/src/hal/hal.c 的 redraw()函数中,利用 NDL_DrawRect()函数来刷新屏幕。NDL_DrawRect()函数在 navy-apps/libs/libndl/src/ndl.c 中定义,它会调用 fwrite()函数,这也是一个在 navy-apps/libs/libc/src/stdio/fwrite.c 中实现的 C 语言库函数。fwrite()函数通过一系列的函数调用,最终在 navy-apps/libs/libos/src/nanos.c 中的 _write()函数,由此将调用链路接到了我们实现的中断处理部分。此时, _syscall()执行 int 0x80 参数中断,并设定好相应参数,随后操作系统通过 nanos-lite/src/syscall.c 中定义的 _RegSet* do_syscall(_RegSet *r)捕获中断,判断出其为 SYS_write,然后调用 fs_write()。在 nanos-lite/src/fs.c 中,我们通过参数 fd 判断出文件类型为 FD_fb,并通过 fb_write()更新屏幕。而 fb_write()则是调用了之前在 nexus-am/am/arch/x86-nemu/src/ioe.c 中实现 _draw_rect()来在更新显示内容所对用的内存。

总结一下,应用程序的运行流程如下:在 nanos-lite 执行 make run 时,首先程序编译 navy-apps 路径下我们指定的应用程序,而在编译应用程序时使用的系统调用有关的实现均在 navy-apps/libs/ 下代码中实现,这部分代码会通过 nexus-am/am/arch/x86-nemu/src/ 中定义的与硬件紧密相关的代码来实现对硬件的具体读写等操作,最终在应用程序、操作系统、lib、am 等完成编译后生成镜像,在交给 nemu 执行。