PA3:穿越时空的旅程： 批处理系统

张祎扬 匡亚明学院 181840326

实验进度：我完成了所有内容

必答题

- 游戏存档的读取

在函数`PAL_LoadGame()`中调用了函数`fread()`来读取游戏存档，代码如下：

```
70
71    //
72    // Read all data from the file and close.
73    //
74    fread(&s, sizeof(SAVEDGAME), 1, fp);
75    fclose(fp);
76
```

`fread()`又调用了nanos.c中的`_read()`函数，这就实现了和libos的协助。`_read()`函数中又调用了`_syscall_()`,调用`SYS_read`,这又实现了和nanos-lite中系统调用的交互。

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

`SYS_read`又调用了函数`fs_read()`，而函数`fs_read()`中通过调用`Ramdisk_read()`来对磁盘进行读取。`ramdisk_read()`通过调用`memcpy()`来实现，这是由AM提供的库函数。最终由NEMU来读写内存，取出一条一条指令，并且译码执行。

```
size_t ramdisk_read(void *buf, size_t offset, size_t len) {
  assert(offset + len <= RAMDISK_SIZE);
  memcpy(buf, &ramdisk_start + offset, len);
  return len;
}
```

- 屏幕的更新

在`navy-apps/apps/pal/src/hal/hal.c`的`redraw()`中调用`NDL_DrawRect()`。`NDL_DrawRect()`在ndl.c中定义。它调用了`fwrite()`，这又是nanos-lite中的系统调用。在系统调用中调用了`ramdisk_write()`函数，它同样调用了`memcpy()`,这是由AM提供的库函数。

```
int NDL_DrawRect(uint32_t *pixels, int x, int y, int w, int h) {
  //printf("hi\n");
  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];
      }
    }
  }
```

通过以上操作，把VGA显存抽象成文件，最后NEMU通过AM提供的I/O接口把文件内容显示到屏幕上。

最简单的操作系统

穿越时空的旅程

实现异常响应机制

在PA3的开始，我花了很多时间RTFSC，我打算在这里做一个记录。

当定义了宏`HAS_CTE`之后，就会调用初始化函数`init_irq()`，并且最终调用`_cte_init()`。

在`_cte_init()`中，首先使用了一些宏，我在x86.h中找到了一些定义。

```
// Gate descriptors for interrupts and traps
typedef struct GateDesc {
  uint32_t off_15_0 : 16;   // Low 16 bits of offset in segment
  uint32_t cs : 16;         // Code segment selector
  uint32_t args : 5;        // # args, 0 for interrupt/trap gates
  uint32_t rsv1 : 3;        // Reserved(should be zero I guess)
  uint32_t type : 4;        // Type(STS_{TG,IG32,TG32})
  uint32_t s : 1;           // Must be 0 (system)
  uint32_t dpl : 2;         // Descriptor(meaning new) privilege level
  uint32_t p : 1;           // Present
  uint32_t off_31_16 : 16;  // High bits of offset in segment
} GateDesc;

#define GATE(type, cs, entry, dpl) (GateDesc)               \
{  (uint32_t)(entry) & 0xffff, (cs), 0, 0, (type), 0, (dpl), \
  1, (uint32_t)(entry) >> 16 }
```

`GateDesc`定义了门描述符的结构。而`GATE`通过几个参数定义了一个新的门描述符。通过阅读这些代码，尝试将宏代入，编号为0x81的门描述符中包含了自陷操作的入口地址为`__am_vectrap`。搜索`__am_vectrap`,发现它在文件`trap.S`中。

```
.globl __am_vectrap;   __am_vectrap: pushl $0x81; jmp __am_asm_trap
```

```
7   __am_asm_trap:
8     pushal
9
10    pushl $0
11
12    pushl %esp
13    call __am_irq_handle
14
15    addl $4, %esp
16
17    addl $4, %esp
18    popal
19    addl $4, %esp
20
21    iret
```

它的作用是先将`0x81`入栈，然后调转至`__am_asm_trap`，这是一串汇编代码，用来保存上下文，最后再返回。在调用`__am_irq_handle`时，将保存的上下文作为参数，然后该函数读出上下文中的irq值，识别出相应的值，并且给event附上对应的值。

为了实现异常响应机制，主要要实现两个指令：`lidt`和`int`.`lidt`的指令是加载idtr，所以我在cpu中添加了idtr，它由base和limit组成。再根据手册的描述，分别加载对应的数值到idtr。调用`vaddr_read()`读取对应位置的数据，并且保存到idtr的相应位置即可。

根据讲义的介绍，要实现`raise_intr()`函数，并且在`int`指令中调用它。讲义的这一部分已经对这一个过程做了一个介绍：

1. 依次将eflags, cs(代码段寄存器), eip(也就是PC)寄存器的值压栈
2. 从IDTR中读出IDT的首地址
3. 根据异常号在IDT中进行索引, 找到一个门描述符
4. 将门描述符中的offset域组合成异常入口地址
5. 跳转到异常入口地址

所以首先用`rtl_push()`指令分别把eflags，cs和返回地址压栈，然后根据idtr中的首地址和异常号进行索引，找到门描述符。根据讲义对门描述符的描述，这里可以很轻松地读取offset，然后组合出最后的跳转地址。根据这个地址来设置decinfo，最终调用`rtl_j()`来跳转到目标地址即可。

通过输出调试法，最终确实跳转到了找到的异常入口地址。

在这一过程中，出现了一个未实现指令a4，我发现框架代码并没有提供这个指令的辅助函数，我阅读手册后尝试实现这个指令，却一直不能正确实现它。最后我在汇编代码中定位到这个指令，发现它在函数`my_memcpy`中，所以我修改了自己在PA2中编写的`memcpy()`函数，重新编译，避开了这个指令。

诡异的x86代码

`push %esp`在`call __am_irq_handle`之前，而`__am_irq_handle()`函数只有一个参数，是一个指向上下文首地址的指针。而在`push %esp`之前，栈中保存的刚好是上下文，所以此时%esp的值就是上下文开始的首地址，所以`push %esp`的行为其实就是向函数传递了一个参数，这个参数是通过栈传递的，是上下文的首地址，也就是一个指向上下文的指针了。

重新组织_Context结构体

观察trap.S中的入栈顺序。首先在跳转到`__am_asm_trap`之前先push了irq，然后调用了pusha指令，最后push0为地址空间占位，由这些在栈上的顺序可以得出上下文的顺序。在`raise_intr()`函数中，也定义了push eflags，cs和eip的顺序。由这些信息在栈上的分布可以得出最后上下文的顺序如下：

```
struct _Context {
  struct _AddressSpace *as;
  uintptr_t edi, esi, ebp, esp, ebx, edx, ecx, eax;
  int irq;
  uintptr_t eip,cs,eflags;
};
```

实现正确的事件分发

这一步并不是很难，首先在`__am_irq_handle()`中添加相应的switch case，然后在`do_event()`中添加相应的switch case，在这里我调用`Log()`输出"yield event"。

恢复上下文

由于我是做完PA3.2之后统一回来码的报告（因为这一次想先集中看代码，不想边看边码），所以我有点忘记了。这里应该是实现了popa指令和iret指令，并且在实现iret的时候按一定顺序出栈即可。

用户程序和系统调用

实现loader

在这里我花费了差不多一整天的时间阅读代码和各种宏的定义，才差不多理解了这个过程。这里其实就是一个对ELF的正确解读。只要找到我们需要的数据，就可以利用`ramdisk_read()`函数把要加载的程序读出来，并且用`memcpy()`把它复制到正确的内存地址（也需要从phdr中读出）。

一开始我有点无从下手，不知道该怎么用`ramdisk_read()`函数。如果把绝对地址作为参数，那太蠢了。后来我想到既然已经定义了Elf_Ehdr和Elf_Phdr这两个结构，我就可以利用结构来通过名称直接获得我想要的数据，而不需要通过绝对地址的索引了。

弄懂了这一点之后，其实就很简单了，最后的代码也并没有很长，但是我留下了非常多的调试痕迹，所以显得有点乱。核心过程就是用memcpy把该加载的内容加载到正确的内存位置，再使用memset把filesz到memsz区间清零即可。

为什么要清零?

我自己的理解是清零可以保证程序正确执行并跳转，不然假如该内存区间原先有其他数据的话，加载程序的时候只是复制而没有清零，那么该内存空间的原数据就不会被覆盖，那么程序执行到本应该结束的地方也许会继续执行下去，超出了应该执行的范围，造成出错。

识别和实现系统调用

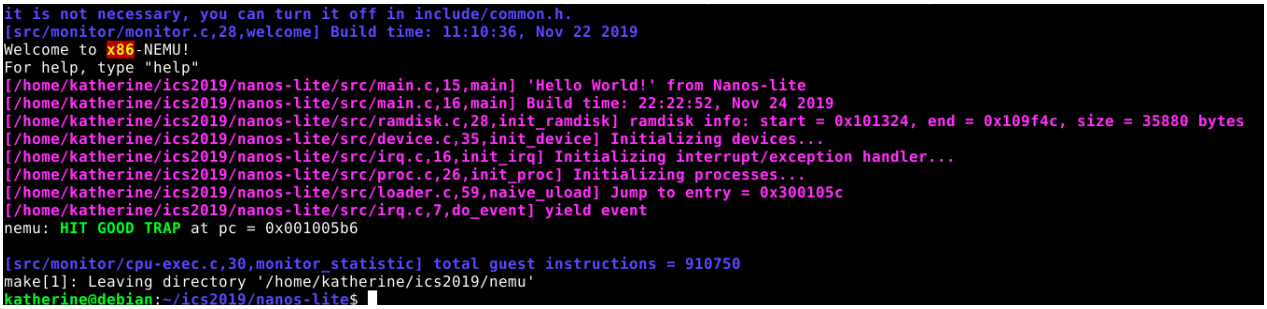
这些过程总的来说不是很复杂，主要就是在各处添加switch case和函数调用。由于时间有点久远，所以细节我记不太清了……

在这其中，一个**关键**是确定GPRx的值。

```
// ISA-depedent definitions
#if defined(__ISA_X86__)
# define ARGS_ARRAY ("int $0x80", "eax", "ebx", "ecx", "edx", "eax")
#elif defined(__ISA_MIPS32__)
# define ARGS_ARRAY ("syscall", "v0", "a0", "a1", "a2", "v0")
#elif defined(__ISA_RISCV32__)
# define ARGS_ARRAY ("ecall", "a7", "a0", "a1", "a2", "a0")
#elif defined(__ISA_AM_NATIVE__)
# define ARGS_ARRAY ("call *0x100000", "rax", "rdi", "rsi", "rdx", "rax")
#else
#error syscall is not supported
#endif

intptr_t _syscall_(intptr_t type, intptr_t a0, intptr_t a1, intptr_t a2) {
  register intptr_t _gpr1 asm (GPR1) = type;
  register intptr_t _gpr2 asm (GPR2) = a0;
  register intptr_t _gpr3 asm (GPR3) = a1;
  register intptr_t _gpr4 asm (GPR4) = a2;
  register intptr_t ret asm (GPRx);
  asm volatile (SYSCALL : "=r" (ret) : "r"(_gpr1), "r"(_gpr2), "r"(_gpr3), "r"(_gpr4));
  return ret;
}
```

这是nanos.c中的代码和宏，由此可以知道，在我选择的ISA x86中，GPR1-4分别为eax，ebx，ecx，edx，返回值GPRx为eax，由此就可以修改这些宏的定义。

在实现完`SYS_exit`系统调用之后，出现了hit good trap。



在nanos-lite上运行Hello World

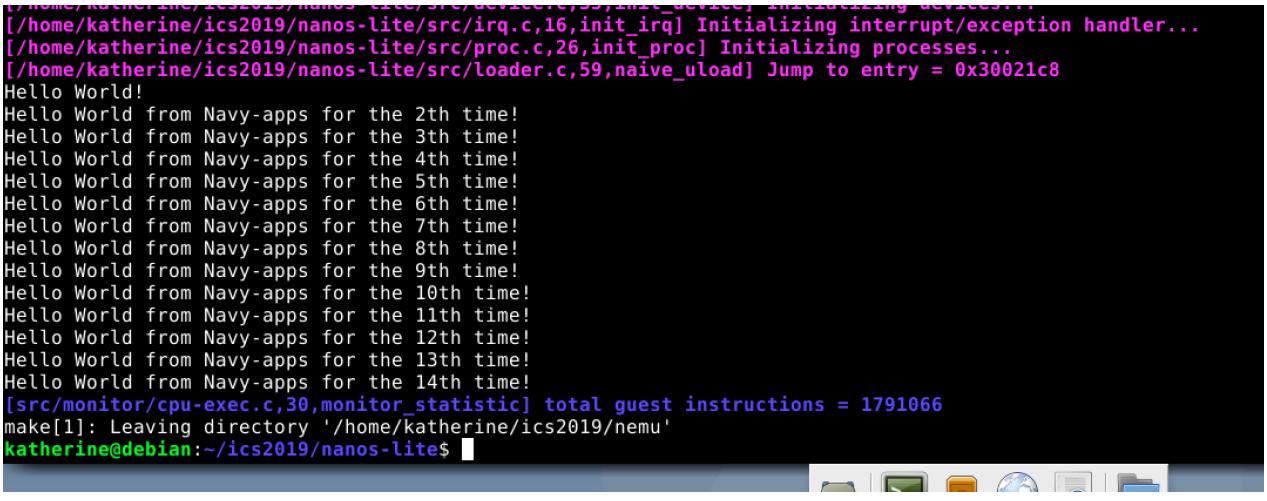
在实现write系统调用时，只需要参考之前实现系统调用的例子，在适当的地方添加代码即可。在man write之后，可以很清楚地了解到write函数的具体行为，从而可以实现它。

实现堆区管理

参照之前的系统调用过程布置好SYS_brk的大致框架，实现_sbrk()的过程参考了manual，代码如下：

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

在实现完之后，确实可以观察到字符串是一次性输出的，同时程序是一个死循环，不停保持输出：



写在这里：PA3.1，PA3.2到这里结束了。在以往的PA过程中，我都选择了一边写代码，一边即时记录感想和思考过程，所以实验报告比较详细，细节也比较多。这一次我怕码报告的过程会影响我的思考进程，所以选择了一次性码完3.1和3.2的代码，再补充实验报告。但是在写实验报告的时候，我意识到，很多我踩过的坑，和四处寻找需要补充的代码的细节都已经遗忘了，（曾经我觉得我在第二遍看讲义的时候可以回忆起来）。所以，在补充实验报告的时候，我只记得一个大概的流程，并且也讲不出什么细节的东西，也不记得自己踩过的坑，也没有梳理一下自己的思考过程，只是很自然的来到了结果。

这让我意识到，在实验的过程中一边踩坑一边记录自己的思考过程是一件很重要的事。当我做完全部的过程再回过头来看的时候，如果没有当初的记录，我会遗忘大量的细节，而这些对于我以后的学习都有很大的帮助。比如在loader的实现过程中，我很痛苦地对着代码思考了一天，仔细阅读了好多遍讲义和宏定义，也stfw去寻找和elf相关的信息，并且在如何用代码实现上进行了大量的思考，也进行了很多输出调试。但由于时间有点久远，在最终的报告上，我不过呈现了寥寥数行字，读起来也没有什么可以引起我的回忆和思考，真的十分可惜。

所以接下来我还是一遍做PA一边记录吧。

文件系统

让loader使用文件

根据讲义的提示，我需要先实现`fs_open()`,`fs_read()`和`fs_close()`这三个函数。

首先实现`fs_open()`,通过`man 2 open`知道函数的返回值是打开的文件的编号。因为文件数量和文件名都是固定的，所以只需要遍历file table中的所有文件，如果文件名相同，则返回文件的编号，否则如果没有找到文件，则根据讲义提示用assertion终止程序。通过观察源代码，我发现fs.c中定义了宏NR_FILES，用file_table的大小除以file_table[0]的大小，就是file_table中文件的数量。

接下来实现`fs_read()`。根据讲义，除了写入`stdout`和`stderr`之外(用`_putc()`输出到串口), 其余对于`stdin`, `stdout`和`stderr`这三个特殊文件的操作可以直接忽略.。所以我打算用一个switch case来区分fd。如果是这三种就什么也不做。同时还要判断偏移量有没有超过文件的边界。以及`fs_read()`返回的是实际读入的长度，所以还要对参数len和可以读的大小即文件大小减去偏移量进行比较，返回比较小的那个值。

接下来实现`fs_close()`，根据讲义要求，直接return 0即可。

最后，为了实现在loader用用文件名加载文件，我还需要改变loader中原来直接使用ramdisk_read()的代码，变成对fs.c中函数的调用。因为可以调用`fs_read()`函数，就可以不必考虑文件在磁盘中的偏移量了。但是在这里我遇到了一个问题，由于`fs_read()`中文件指针`open_offset`只能往后顺延，所以不能在读完phdr之后又马上读对应的程序。也许可以选择一次性读完所有的程序头表，再往后顺延，但我不喜欢这样。所以我决定写一个函数`fs_offset()`返回文件在磁盘中的偏移量，在最后一次读取程序时调用`ramdisk_read()`,直接与程序在文件中的偏移量相加即可。

我还考虑到可以用`fs_lseek()`函数来调整偏移量，可以在每读完一次就调整一次偏移量。但是由于在这个板块中另外两个函数还没有实现，所以也先不考虑这种做法。最后我编译运行，文件名输入"/bin/dummy"和"/bin/hello",全部成功运行。

```
[/home/katherine/ics2019/nanos-lite/src/main.c,15,main] 'Hello World!' from Nanos-lite
[/home/katherine/ics2019/nanos-lite/src/main.c,16,main] Build time: 23:41:17, Nov 28 2019
[/home/katherine/ics2019/nanos-lite/src/ramdisk.c,28,init_ramdisk] ramdisk info: start = 0x1019b0, en
[/home/katherine/ics2019/nanos-lite/src/device.c,35,init_device] Initializing devices...
[/home/katherine/ics2019/nanos-lite/src/irq.c,16,init_irq] Initializing interrupt/exception handler..
[/home/katherine/ics2019/nanos-lite/src/proc.c,26,init_proc] Initializing processes...
[/home/katherine/ics2019/nanos-lite/src/loader.c,81,naive_uload] Jump to entry = 0x300105c
[/home/katherine/ics2019/nanos-lite/src/irq.c,7,do_event] yield event
nemu: HIT GOOD TRAP at pc = 0x00100722
```

```
Hello World from Navy-apps for the 2th time!
Hello World from Navy-apps for the 3th time!
Hello World from Navy-apps for the 4th time!
Hello World from Navy-apps for the 5th time!
Hello World from Navy-apps for the 6th time!
Hello World from Navy-apps for the 7th time!
Hello World from Navy-apps for the 8th time!
Hello World from Navy-apps for the 9th time!
Hello World from Navy-apps for the 10th time!
Hello World from Navy-apps for the 11th time!
Hello World from Navy-apps for the 12th time!
[src/monitor/cpu-exec.c,30,monitor_statistic] total guest instructions = 1663246
make[1]: Leaving directory '/home/katherine/ics2019/nemu'
katherine@debian:~/ics2019/nanos-lite$
```

(因为loader的代码老是改来改去的所以我这里贴一下核心代码)

```
static uintptr_t loader(PCB *pcb, const char *filename) {
  Elf_Ehdr elfheader;
  Elf_Phdr programheader;
  int fd = fs_open(filename,0,0);
  if(fd!=-1){
    fs_read(fd,&elfheader,sizeof(Elf_Ehdr));
    for(uint16_t i=0;i<elfheader.e_phnum;i++){
      fs_read(fd,&programheader,sizeof(Elf_Phdr));
      if(programheader.p_type == PT_LOAD){
        uint8_t buf[programheader.p_filesz];

 ramdisk_read(&buf,programheader.p_offset+fs_offset(fd),programheader.p_filesz);
        memcpy((void*)programheader.p_vaddr,&buf,programheader.p_filesz);
        memset((void*)(programheader.p_vaddr+programheader.p_filesz),0,(programheader.p_memsz-programheader.p_filesz));
      }
    }
  }
  fs_close(fd);
  return elfheader.e_entry;
}
```

一点总结：一开始的loader都是用`ramdisk_read()`实现的，在实现了一些fs函数之后，就要对loader进行改动。一开始我总是出现out of bounds的报错，于是我检查了很多遍loader的实现，发现跳转地址是正确的，也就是说我的`fs_read()`函数应该问题不大，问题应该出在后面读程序段的过程中，然后我意识到了文件指针的偏移问题。既然还没有实现`fs_lseek()`函数，还不能在每次读完程序头表之后调整偏移量。我也不想选择一次性读完全部程序头表，所以只能出此下策，写了一个返回disk_offset的函数，并且在读程序段的时候调用`ramdisk_read()`，在偏移量参数中加上offset。在实现loader的过程中，通过不断的调试，我对程序头表、文件在ramdisk中的关系都有了非常深刻的理解。

实现完整的文件系统

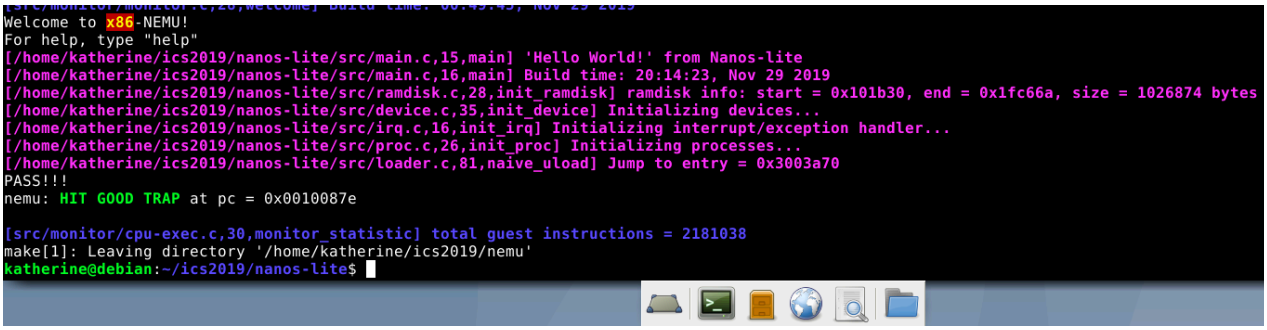
首先实现`fs_write()`.根据讲义提示，写入stdout和stderr是用`_putc()`输出到串口，并且调用`Ramdisk_write()`进行真正的写操作。在这里我仍然模仿之前的使用一个switch case。特殊情况时使用`_putc()`进行输出，default情况时模仿`fs_read()`，首先检查有没有越界，然后通过比较确定写入数据的长度，最后调用`ramdisk_write()`进行写入，并且更新offset。

接着实现`fs_lseek()`.它的作用是调整文件指针的偏移量。通过man 2 lseek我知道whence有三种类型，分别是SEEK_SET,SEEK_CUR,SEEK_END,在fs.h中已经对它们进行了定义，所以我还是打算使用一个switch case，对这些情况分别进行处理。并且它的返回值是指针距文件开头的偏移量。实现的过程并不难，按照man的指示即可。

实现完之后我尝试运行"/bin/test",发现一直是hit bad trap.我用输出调试法调试了很久，发现只要和文件有关的操作都会hit bad trap.....我陷入了僵局，然后就半夜一点了，于是我就上床睡觉了。

第二天醒来之后，我继续僵局了很久，直到我看见了讲义上的一句话.....**最后你还需要在Nanos-lite和Navy-apps的libos中添加相应的系统调用, 来调用相应的文件操作**，dbq是我眼瞎我面壁我该打！（跑去实现系统调用去了）。

实现了系统调用之后，我再次编译运行，发现了一个invalid opcode，是一个inc指令没有实现。我填好了opcode table之后，再次编译运行，看到了pass！！并且hit good trap了。



把串口抽象成文件

首先我完成函数`serial_write()`,即输出到串口，再把相应的函数填入file table 中的stdout和stderr中。然后我修改了我的`fs_write()`函数，先判断文件的函数指针是否为空，如果为空，则默认进行文件写操作，如果不为空，则调用相应的写函数。

把设备输入抽象成文件

读完讲义后，我找到了以下几个注意点。

- 优先处理按键事件，再处理时钟事件。
- 实现events_read(),返回写入的实际长度
- 借助IOE的API来获得设备的输入。
- 在VFS中添加支持。

首先我打算实现`events_read()`函数。我用vscode搜索关键词key，在am的klib中找到了c文件`io.c`，在里面找到了调用`read_key()`的接口，它返回一个int型数据，是key在数组中的编号。
在`events_read()`中调用这个函数，就可以获得按下的按键。同时`io.c`中还有`uptime()`函数，调用它可以获得当前的时间。在`events_read()`中，我优先实现对按键事件的处理。最后在VFS中添加支持。

我尝试编译运行，发现并没有输出时间信息，只能在我按下按键的时候输出按键信息。我用native编译运行，发现看上去似乎很正常，于是我以为是我的nemu的问题，我一路排查到了nemu-input.c，还是没有找到问题所在。不知道为什么同样是

把VGA显存抽象成文件

- 在`init_fs()`中对`/dev/fb`的大小初始化，需要调用io.c中的接口`screen_width()`和`screen_height()`。
- 实现`fb_write()`.因为`draw_rect()`只能画矩形，所以要针对能不能整除进行分类。保证每次写入的形状都是矩形。
- 实现`fbsync_write()`.直接调用API`draw_sync()`.
- `init_device()`直接调用`sprintf()`即可。将宽和高按照指定格式输入。
- `dispinfo()`本质也不复杂，我对len进行了一个判断，然后调用了`strncpy()`.

这些都实现完之后，问题就很大了，我并不能跑起`/bin/bmptest`。而是直接出现了out of bounds。后来通过屈道涵助教的点拨，我发现还是我的loader实现有问题。在实现了文件系统以后，我并没有完全在loader中使用fs函数，而是保留了一个`ramdisk_read()`,这就需要我定义一个filesz大小的数组，当filesz很大的时候，就溢出了。所以我重新写了loader函数，全部调用文件系统的函数，并且用`fs_lseek()`来设置offset，改进后的代码如下：

```
static uintptr_t loader(PCB *pcb, const char *filename) {
  Elf_Ehdr elfheader;
  Elf_Phdr programheader;
  int fd = fs_open(filename,0,0);
  if(fd!=-1){
    fs_read(fd,&elfheader,sizeof(Elf_Ehdr));
    fs_lseek(fd,elfheader.e_phoff,SEEK_SET);
    for(uint16_t i=0;i<elfheader.e_phnum;i++){
      fs_read(fd,&programheader,sizeof(Elf_Phdr));
      size_t opset=fs_openoffset(fd);
      if(programheader.p_type == PT_LOAD){
        fs_lseek(fd,programheader.p_offset,SEEK_SET);
        fs_read(fd,(void *)programheader.p_vaddr,programheader.p_filesz);
        memset((void*)(programheader.p_vaddr+programheader.p_filesz),0,
(programheader.p_memsz-programheader.p_filesz));
      }
      fs_lseek(fd,opset,SEEK_SET);
    }
```

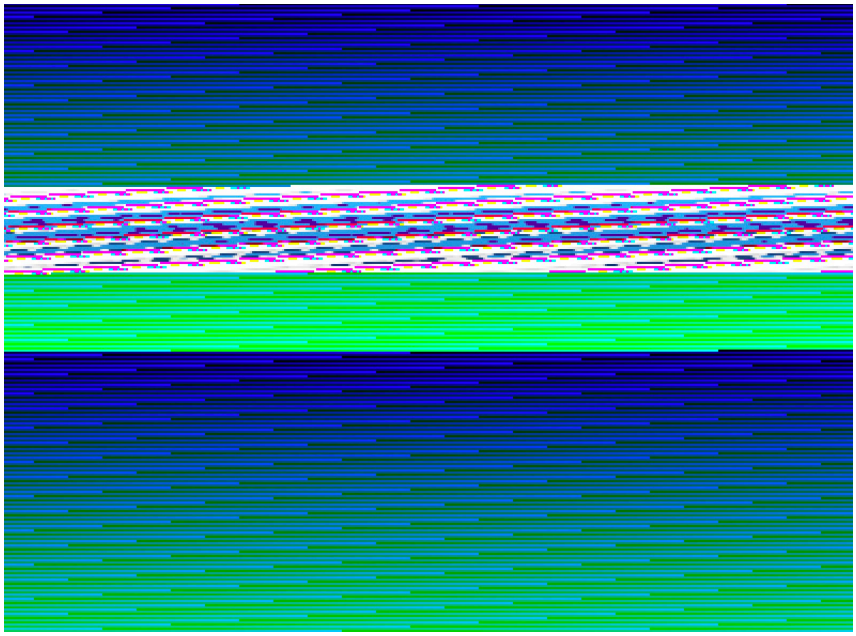
```
  }
  fs_close(fd);
  return elfheader.e_entry;
}
```

实现完之后，我重新跑了hello和dummy和text，都通过了。

解决了buffer过大的问题以后，就可以开始debug了。首先我碰到了一些还没有实现的指令，比如a4的movsb，它和段寄存器有关，我尝试忽略段寄存器按照手册实现了它。接着又触发了几个invalid opcode，只需要完善opcode_table即可。实现指令后，我通过添加Log发现在read死循环了，然后找到原因是read完之后没有设置offset。在实现文件系统以后可以通过判断文件的函数指针是否为空来决定是调用特定的函数还是默认函数，因为一开始还没有实现函数指针的功能，在添加了以后我不小心把设置offset的语句放进了else里，这就导致如果函数指针不为空的话，就缺少了设置offset的语句，所以一直进行死循环。

同理，我也修改了`fs_write()`里设置offset的语句。然后我突然想到，events的异常可能就是写函数的问题，于是我重新跑了events，发现它果然正常输出时间信息了。

接下来就是调bmptest了。我在nemu上跑，发现输出的图案是扭曲的，如下图



但是我尝试在native里跑，却出现了segmentation fault。为了这个segmentation fault我实在是耗费了大量时间精力……我阅读了bmptest的main.c的代码，又根据讲义的指示阅读了`navy-apps/libs/libndl/`目录下的代码. 我一步步地用printf法输出，想要尝试找到出现segmentation fault的语句。这一个过程花了很久。最后我定位到了`ndl.c`中的这样的代码：

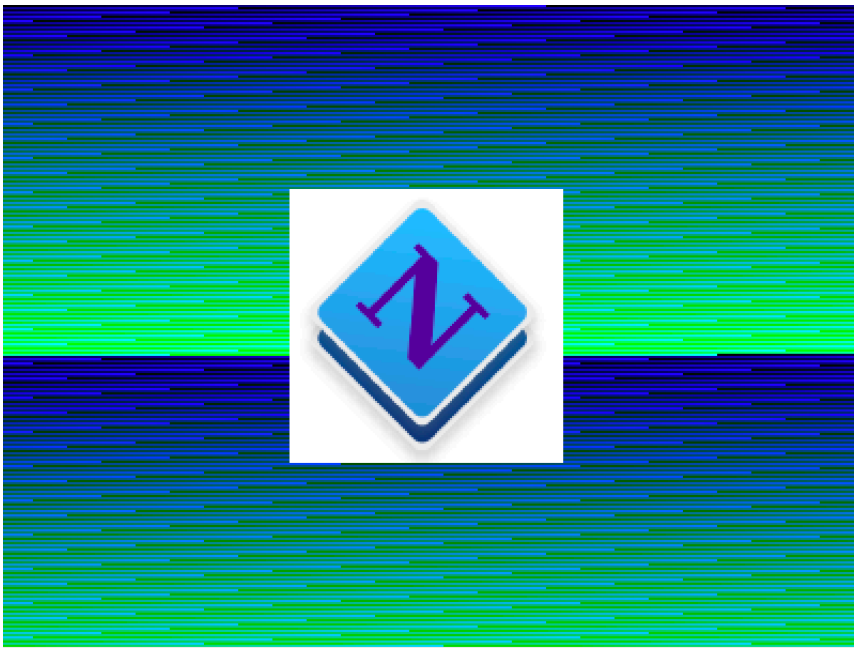
```c
static void get_display_info() {
  printf("before open dispinfo\n");
  FILE *dispinfo = fopen("/proc/dispinfo", "r");
  printf("after open dispinfo\n");
  assert(dispinfo);
  screen_w = screen_h = 0;
  char buf[128], key[128], value[128], *delim;
  printf("buf success\n");
  while (fgets(buf, 128, dispinfo)) {
    printf("close\n");
    *(delim = strchr(buf, ':')) = '\0';
    printf("hi\n");
    sscanf(buf, "%s", key);
    printf("buf\n");
    sscanf(delim + 1, "%s", value);
    //printf("close\n");
    if (strcmp(key, "WIDTH") == 0) sscanf(value, "%d", &screen_w);
    if (strcmp(key, "HEIGHT") == 0) sscanf(value, "%d", &screen_h);

  }
```

通过输出调试法我发现，while循环的作用是读取dispinfo中的数据，第一次循环读取了宽度，第二次循环读取了高度。按照道理不应该开始第三次循环，因为在init中只写入了这两个数据。但是输出调试法却显示它开始了第三次循环，并且在while循环第一个语句就开始segmentation fault，所以我怀疑读取到的dispinfo文件有问题，它可能不能正常的结束，于是我回过头去看了我的`dispinfo_read()`函数，如下图：

```c
size_t dispinfo_read(void *buf, size_t offset, size_t len) {
  //Log("dispinfo_read\n");
  //if(len+offset>128)len=128-offset;
  if(len+offset>strlen(dispinfo))len=strlen(dispinfo)-offset;
  strncpy(buf, &dispinfo[offset], len);
  return len;
}
```

本来我写的是被注释掉的语句，因为数组的大小规定好了是128，但我想了想这样也许要出问题，就把它改为了数组的真实大小，也就是`strlen(dispinfo)`.**改完之后，native确实可以显示图像了，和nemu中的结果一样。**

那么接下来我就有理由相信问题出在draw上了，我反复检查了很多遍draw函数，都觉得逻辑正确。但后来我考虑到最终显示的图像在尺寸上完全斜过来了，而且刚好是4倍的关系，于是我发现了返回值的错误。在`fb_write()`中我写了`len = len>>2`这样的语句，可是我的返回值还是len，但它其实已经无形中被修改为参数len的$\frac{1}{4}$,所以最终呈现的图像是斜过来并且压扁的。我重新设置了一个新的变量用来存放`len>>2`值，图像最终显示正常了。



一点感想：最后的VGA和eventbug百出，让我经过了无数次调试编译运行。虽然它最后在讲义中看似轻描淡写，但是我差不多在上面耗费了一天两夜的时间。大多数时候，我都是在徒劳地改一些并不关键的代码。在调这几个bug的过程中，我大量使用了Log和printf，并体会到了输出调试法对于定位问题的帮助。同时，我也感受到了一个bug de好几天的绝望的感觉。但不管是多难的事，多深的bug，只要一步步去调试，总是可以慢慢接近问题的本质。在调这些bug的时候，我经历了一段非常黑暗和低落的过程，对自己陷入了很深的自我怀疑，甚至在毫无头绪的时候怀疑过自己的整个构思框架是不是不对，并且十分焦虑。好在后来定位到bug之后，证明了都是一些非常细节的小错误，至少我的大思路还是对的。（之前read死循环的时候一直怀疑是自己文件系统的实现框架就不对）但这恰恰也证明了，**The machine is always right,**计算机世界是非常严谨的，哪怕一个微小的错误都可能引发巨大的后果（比如我在native中的segmentation fault），甚至可能让你很多天的时间一无所获。但正因为计算机的世界是非常严谨的，机器永远是对的，所以无论你在debug的时候多么没有头绪，多么自我怀疑，结果都证明了不管多难找，bug是一定存在的，所以当遇到问题的时候，找bug就对了。

致谢：感谢屈道涵助教的指点，让我学到了很多debug的技巧和心态方面的感悟。

批处理系统

在NEMU中运行仙剑奇侠转

按照讲义步骤配置好文件并更新ramdisk，run的时候又出现了invalid opcode a5，我仿照实现a4的过程实现了它，就可以运行仙剑奇侠传了！！！我玩的不亦乐乎（因为以前从来没玩过类似游戏），惊叹于它的强大。

添加开机菜单

根据讲义的介绍，首先在VFS中添加对文件`/dev/tty`的支持，并且将它的写函数设置为串口写入函数`serial_write()`.然后在`syscall.c`的switch case中添加case `SYS_execve`，并且调用`naive_uload()`,第一个参数设置为`NULL`,第二个参数由寄存器传入。

之后在nanos.c中完成`_execve()`函数，调用`_syscall_()`函数并正确传入参数即可。

实现完之后就可以成功实现程序的自由选择了。

展示你的批处理系统

根据讲义提示，只需修改syscall.c中`SYS_exit`的内容，将它改为调用`naive_uload(NULL,"/bin/init")`即可。

写在最后：~~一开始信子讲义的邪~~，以为PA3真的超级无敌简单，然而发现并不是.....不过PA3是目前为止我最喜欢的一部分，它的工作更有挑战性。此外，无论是系统调用的实现，还是一开始让我想破脑袋的loader，其实都让我对计算机系统有了更加深入的了解。（尤其是一开始实现loader，简直是让我对elf和程序头表有了非常无比深刻的认识）。我非常喜欢实现文件系统中的函数指针部分，通过判断指针是否为空并进行不同的函数调用，真的是一个让我惊叹的想法！