PA3实验报告

1811***_秦凧

Part1

1.1 实验目的

1. 了解简单操作系统的基本构成
2. 了解操作系统加载程序的过程
3. 了解操作系统的特权级和中断
4. 了解操作系统对异常和系统调用的处理

1.2 实验内容

1. 实现 loader 并加载 `dummy` 程序
2. 实现中断机制
3. 重新组织 `TrapFrame` 结构体
4. 实现系统调用

1.3 实验步骤

1.3.1 实现 loader 并加载 `dummy` 程序

loader 是一个用于加载程序的模块，它的工作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。

为了使 loader 能够加载 `dummy` ，我们首先需要让 Navy-apps 项目上的程序默认编译到 x86 中，在 `Navy-app/Makefile` 中将 `ISA ?= native` 改为 `ISA ?= x86` ，之后在 然后在 `navy-apps/tests/dummy` 下执行 make，就会在 `navy-apps/tests/dummy/build/` 目录下生成 `dummy` 的可执行文件。

同时为了避免和 `Nanos-lite` 的内容产生冲突，pa约定目前用户程序需要被链接到内存位置 0x4000000 处，也就是说我们的 `loader` 需要将 `ramdisk` 中从 0 开始的所有内容放置在 0x4000000，并把这个地址作为程序的入口返回。要将 `ramdisk` 复制到指定位置可以使用框架中已经编写好的 `ramdisk_read` 函数，代码如下：

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

实现 loader 后再 nanos-lite 目录下执行 `make update` 和 `make run` 在 NEMU 上运行带有最新版 ramdisk 的 Nanos-lite ，之后会看到 `dummy` 执行了一条未实现的 int 指令，说明 loader 已成功加载并跳转到 `dummy` 中执行。

1.3.2 实现中断机制

异常是指 CPU 在执行过程中检测到的不正常事件， CPU 检测到异常之后会首先保存现场，之后跳转到处理该异常的函数处执行已成处理程序，处理完异常之后根据之前保存的地址和其他寄存器值回复现场并跳转回原来停下的地方。

实现中断处理机制我们首先需要在 NEMU 中添加 IDTR 寄存器并实现 lidt 指令。

对于 IDTR 寄存器，需要修改 `nemu\include\cpu\reg.h` 中的 `CPU_state` 结构体，在其中增加对 IDTR 寄存器的定义，根据实验指导书和 i386 手册可知 IDTR 寄存器结构如下：

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

实现 lidt 指令的流程与pa2中大致流程相同，首先在 `opcode_table` 对应位置填写指令函数名，之后在 `nemu\src\cpu\exec\all-instr.h` 中定义函数，最后实现 lidt 指令的函数。 lidt 指令用于在 IDTR 中设置 IDT 的首地址和长度，首先调用 `vaddr_read` 函数，从给出的地址中读出长度，之后判断操作数宽度，根据不同的操作数宽度设置 idtr 寄存器的首地址：

```
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);
}
```

之后需要在 `nanos-lite/src/main.c` 中定义宏 `HAS_ASYE` ， 并实现 `raise_intr` 函数。

`raise_intr` 函数用于处理中断机制，首先需要保存当前一部分寄存器的值，之后设置跳转，跳转地址需要通过查询 idtr 寄存器，并进行一些列计算来获取：

```
//nanos-lite/src/main.c 中定义宏 HAS_ASYE
#define HAS_ASYE

//raise_intr
void raise_intr(uint8_t NO, vaddr_t ret_addr) {
    rtl_push(&(cpu.eflags.eflagsVal));
    rtl_push((rtlreg_t*)&(cpu.cs));
    rtl_push(&ret_addr);//保存现场

    uint32_t idtr_base = cpu.idtr.base;//读首地址
    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;//索引
    offset = eip_low | eip_high;//目标地址
    decoding.jmp_eip = offset;//设置跳转
    decoding.is_jmp = true;
}
```

除此之外，我们还需要实现在pa2中没有实现的 `int` 指令，实现该指令较为简单，只需要在函数中调用 `raise_intr` 即可：

```
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
}
```

全部实现后重新运行 `dummy` 会看到出现为实现的指令的提示。

1.3.3 重新组织 `TrapFrame` 结构体

实现系统调用之前首先需要保存异常出现之前的通用寄存器内容，我们可以使用 `pusha` 指令将通用寄存器的值压入堆栈。除此之外，还需要重新组织 `nexus-am/am/arch/x86-nemu/include/arch.h` 中定义的 `_RegSet` 结构体的成员，使其声明的顺序和 `nexus-am/am/arch/x86-nemu/src/trap.S` 中构造的 `trap frame` 保持一致，使得代码可以正确使用 `trap frame` 。

对于 `pusha` 指令，我们只需要按照 i386 手册的描述按下魂虚将寄存器依次入栈即可：

```
make_EHelper(pusha) {
  t2=cpu.esp;
  rtl_push(&cpu.eax);
  rtl_push(&cpu.ecx);
  rtl_push(&cpu.edx);
  rtl_push(&cpu.ebx);

  rtl_push(&t2);

  rtl_push(&cpu.ebp);
  rtl_push(&cpu.esi);
  rtl_push(&cpu.edi);
  print_asm("pusha");
}
```

对于 `_RegSet` 结构体，通过查看 `trap.S` 可知重构后的顺序如下：

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

实现上述内容后，再次运行 `dummy` ，会看到触发了未处理的 8 号事件。

1.3.4 实现系统调用

根据实验指导书，实现系统系统调用有以下几个步骤：

在 do_event 中识别出系统调用事件 `_EVENT_SYSCALL` ，然后调用 `do_syscall` ：

实现该部分时我们只需要在 `switch` 语句中对 `_EVENT_SYSCALL` 事件进行处理，调用 `do_syscall` 并将 r 作为参数传入即可：

```
static _RegSet* do_event(_Event e, _RegSet* r) {
  switch (e.event) {
    case _EVENT_SYSCALL:
      do_syscall(r);
      break;

    default: panic("Unhandled event ID = %d", e.event);
  }
  return NULL;
}
```

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_ARGx` 宏定义在 `nexus-am\am\arch\x86-nemu\include\arch.h` 中，只需要返回参数对应的寄存器值即可：

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

添加 `SYS_none` 系统调用，设置系统调用的返回值：

根据实验指导书， `SYS_none` 需要返回 1 ， `SYSCALL_ARG1(r)` 用于保存返回值，故只需将 1 赋给 `SYSCALL_ARG1(r)` 即可：

```
case SYS_none:
    SYSCALL_ARG1(r) = 1;
    break;
```

实现 popa 和 iret 指令：

popa 指令与 pusha 指令过程相反，大体上只需要按照相反的顺序将寄存器出栈即可， iret 指令需要按照与 raise_intr 相反的顺序将三个寄存器出栈，并设置返回时的跳转地址：

```
make_EHelper(popa) {
  rtl_pop(&cpu.edi);
  rtl_pop(&cpu.esi);
  rtl_pop(&cpu.ebp);
  rtl_pop(&t2);

  rtl_pop(&cpu.ebx);
  rtl_pop(&cpu.edx);
  rtl_pop(&cpu.ecx);
  rtl_pop(&cpu.eax);
  print_asm("popa");
}

make_EHelper(iret) {
  rtl_pop(&t2);
  decoding.jmp_eip=t2;
  rtl_pop((rtlreg_t *)&cpu.cs);
  rtl_pop(&(cpu.eflags.eflagsVal));//恢复现场
  decoding.is_jmp=1;//设置跳转
  print_asm("iret");
}
```

实现 SYS_exit 系统调用：

该系统调用接收一个退出状态的参数，用这个参数调用_halt()即可：

```
case SYS_exit:
    _halt(a[1]);
    break;
```

pa3-part1 全部实现之后，再次运行 dummy 程序，会看到 GOOD TRAP 的信息：

```
[src/monitor/monitor.c,30,welcome] Build time: 16:37:06, May  3 2021
For help, type "help"
(nemu) c
[src/main.c,19,main] 'Hello World!' from Nanos-lite
[src/main.c,20,main] Build time: 21:49:27, May  3 2021
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x100fd0, end = 0x1055ac,
size = 17884 bytes
[src/main.c,27,main] Initializing interrupt/exception handler...
nemu: HIT GOOD TRAP at eip = 0x00100032
```

1.4 问答

1.4.1 对比异常与函数调用

相同点：异常和函数调用都会进行跳转，跳转之前都需要保存跳转前的指令运行到的地址。

不同点：函数保存的跳转前指令运行到的地址时保存的是 eip 或 eip 和 cs 寄存器的值，而异常除了要保存 cs 寄存器的值，还需要保存 EFLAG 寄存器的值。

不同的原因：函数调用跳转的距离有时远有时近，故有时需要保存 cs 寄存器，有时不需要；异常处理跳转的距离通常较远，故需要保存 cs 寄存器；对于 EFLAG 寄存器，函数调用之后执行的内容与之前的内容时相关的，即使影响了 EFLAG 寄存器也没关系，但异常处理与之前正在执行的任务具体内容无关，如果改变了 EFALG 会影响原来进程的正确执行，所以异常处理需要保存原来的 EFALGS 寄存器。

1.4.2 pushl %esp 的行为

类比函数调用，计算机会对战进行一系列操作，首先将调用函数的参数存储到 eax 等寄存器中，并将这些寄存器压栈，之后将此时下一条要执行指令的地址也就是 eip 寄存器的压栈，然后进入被调用的函数。进入被调用的函数之后，计算机会继续将原来的栈底即 ebp 寄存器的值压栈，目的是当返回时能找到原来的函数的栈底，接着将 esp 的值赋给 ebp ，生成新的栈底，这是 ebp 指向的位置临近被调用的函数需要的参数的位置。而对于 irq_handle ，他在运行时除了基本的参数还需要用户进程触发异常时现场的状态，该状态存储在 trap frame 中，这时候就需要在额外将 trap frame 的地址传给 irq_handle ， pushl %esp 则是用来将 trap frame 的地址压入栈，提供给 irq_handle 做参数使用。

---

part2

2.1 实验目的

1. 了解标准输出的处理过程
2. 了解堆区管理的方式
3. 了解简易文件系统的管理方式

2.2 实验内容

1. 在 Nanos-lite 上运行 Hello world
2. 实现堆区管理
3. 让 loader 使用文件
4. 实现完整的文件系统

2.3 实验步骤

2.3.1 在 Nanos-lite 上运行 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。

实现的具体流程如下：

首先在在 do_syscall 中编写对系统调用号 SYS_write 的处理，并编写处理输出的函数。注意此处的 sys_write 自己添加的函数，用来临时处理输出，后续会将该函数合并到之后编写的 fs_write 中，其中的 log 语句用于下一内容中对堆区管理实现的调试：

```
//do_syscall 中编写对系统调用号 SYS_write 的处理
case SYS_write:
  SYSCALL_ARG1(r) = sys_write((int)a[1],(void *)a[2],(size_t)a[3]);
  break;

//sys_write 函数
int sys_write(int fd, const void *buf, size_t len){
  Log("sys_write:fd %d  len %d\n",fd,len);
  if (fd == 1 || fd == 2) {
    int i = 0;
    for(; i < len; i++){
      _putc(((char*)buf)[i]);
    }
    return i;
  }
  return -1;
}
```

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：

1. 切换到 navy-apps/tests/hello/ 目录下执行 make 编译 hello 程序
2. 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。

```
[src/main.c,19,main] 'Hello World!' from Nanos-lite
[src/main.c,20,main] Build time: 21:49:27, May  3 2021
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x1010a8, end = 0x105784,
size = 18140 bytes
[src/main.c,27,main] Initializing interrupt/exception handler...
[src/syscall.c,7,sys_write] sys_write:fd 1  len 13

Hello World!
[src/syscall.c,7,sys_write] sys_write:fd 1  len 1

H[src/syscall.c,7,sys_write] sys_write:fd 1  len 1

e[src/syscall.c,7,sys_write] sys_write:fd 1  len 1

l[src/syscall.c,7,sys_write] sys_write:fd 1  len 1

l[src/syscall.c,7,sys_write] sys_write:fd 1  len 1

o[src/syscall.c,7,sys_write] sys_write:fd 1  len 1
```

2.3.2 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:
  SYSCALL_ARG1(r) = 0;
  break;
```

之后在 `navy-app` 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，其使用方法可以通过 `man 3 end` 查阅；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 end = (intptr_t)&_end;
void *_sbrk(intptr_t increment){
  intptr_t old = end;
  if(_syscall_(SYS_brk, old + increment, 0, 0) == 0){
    end += increment;
    return (void*)old;
  }
  return (void *)-1;
}
```

实现以上内容后再次运行 Hello world ，可以看到此时系统不再是逐个字符输出，而是一次性输出完整的语句：

```
(nemu) c
[src/main.c,19,main] 'Hello World!' from Nanos-lite
[src/main.c,20,main] Build time: 15:08:29, May  5 2021
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x1010c0, end = 0x105800,
size = 18240 bytes
[src/main.c,27,main] Initializing interrupt/exception handler...
[src/syscall.c,9,sys_write] sys_write:fd 1  len 13

Hello World!
[src/syscall.c,9,sys_write] sys_write:fd 1  len 29

Hello World for the 2th time
[src/syscall.c,9,sys_write] sys_write:fd 1  len 29

Hello World for the 3th time
[src/syscall.c,9,sys_write] sys_write:fd 1  len 29

Hello World for the 4th time
[src/syscall.c,9,sys_write] sys_write:fd 1  len 29
```

2.3.3 让 loader 使用文件

在前面的内容中 loader 来直接调用 `ramdisk_read` 来加载用户程序，在这里我们需要让 loader 使用文件系统，为此我们需要实现文件的打开、读取和关闭，这也是 2.3.4 中文件系统的一部分。实现文件的打开、读取和关闭涉及到三个系统调用，实现系统调用的过程与上面类似，都是首先 `nanos-lite` 中实现对系统调用号的处理，之后在 `navy-app` 中调用接口函数，需要注意的是在处理系统调用号时为了使代码更加更加规整， pa 定义了三个函数供处理系统调用号时调用 `fs_open` `fs_read` 和 `fs_close` ，其在 `nanos-lite\include\fs.h` 中定义，在 `nanos-lite\src\fs.c` 中实现。

首先，为了保存文件的状态，我们需要补全 `Finfo` 结构体，包括文件名、文件大小、文件在磁盘上的偏移和文件读取到的位置：

```
typedef struct {
  char *name;
  size_t size;
  off_t disk_offset;
  off_t open_offset;//文件打开后的读写指针
} Finfo;
```

之后是对系统调用号的处理，只需要在 `do_syscall` 的 `switch` 语句中添加相关内容，调用函数即可：

```
case SYS_open:
  SYSCALL_ARG1(r) = fs_open((char *)a[1],(int)a[2],(int)a[3]);
  break;
case SYS_read:
  SYSCALL_ARG1(r) = fs_read((int)a[1],(void *)a[2],(size_t)a[3]);
  break;
case SYS_close:
  SYSCALL_ARG1(r) = fs_close((int)a[1]);
  break;
```

同样在 navy-app 中调用接口函数也比较简单，只需要传入相关参数即可：

```
int _open(const char *path, int flags, mode_t mode) {
  _syscall_(SYS_open, (uintptr_t)path, flags, mode);
}
int _read(int fd, void *buf, size_t count) {
  _syscall_(SYS_read, fd, (uintptr_t)buf, count);

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

重点在与对 fs_open fs_read 和 fs_close 的实现。首先是 fs_open 和 fs_close ，根据实验指导手册，因为简易文件系统没有维护文件打开的状态，故 fs_close 可以直接返回 0，表示总是关闭成功；而 fs_open 需要在所有的文件中查找要打开的文件，如果找到返回该文案件对应的索引下标，否则使用断言停下程序：

```
int fs_open(const char *pathname, int flags, int mode) {
  for (int i = 0; i < NR_FILES; i++) {//查找文件
    if (strcmp(pathname, file_table[i].name) == 0) {
      file_table[i].open_offset = 0;
      return i;
    }
  }
  assert(0);//没有该文件
  return -1;
}

int fs_close(int fd) {
  //没有维护文件打开的操作，直接返回0
  return 0;
}
```

对于 fs_read ，该函数需要根据文件的不同类型进行不同操作，但在 part2 中我们只需要处理 FD_STDOUT 和 FD_STDERR 类型的文件即可，其余类型会在后续试验中补充。 fs_read 首先计算需要读取的字节数，如果调用时要读取的字节数超过文件中的剩余的字节数，则读取到文件结尾，否则按要求读取，读取文件内容可以通过调用 ramdisk_read 实现，读取之后需要修改记录文件读取偏移的 open_offset 。

```
ssize_t fs_read(int fd, void *buf, size_t len) {
  int toRead = (file_table[fd].size - file_table[fd].open_offset > len) ? len :
file_table[fd].size - file_table[fd].open_offset;//要读出的长度
  if (toRead > 0) {
    if (fd == FD_STDOUT || fd == FD_STDERR) {
      ramdisk_read(buf, file_table[fd].disk_offset + file_table[fd].open_offset,
toRead);//读文件
      file_table[fd].open_offset += toRead;//修改偏移
    }
```

```
  }
  return toRead;
}
```

最后，实验指导书中还提到我们需要实现获取文件大小的函数 `fs_filesz` ，该函数不会在 part2 中使用到，但会在后续 part3 中用到，对于该函数只要根据文件号找到其对应的 `file_table` 成员，并返回该成员的 size 即可：

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

至此，文件系统还未全部实现， 但 loader 已经可以通过文件名读取文件了，只需要在 `nanos-lite\src\loader.c` 中调用 `fs_open` 并记录返回的文件号，再通过 `fs_read` 就可以对该文件进行读取。

2.3.4 实现完整的文件系统

2.3.3 中已经实现了文件系统的一部分，在此只需要补充对读取和设置偏移的实现即可，流程与上述相同。

首先在 `do_syscall` 中对系统调用号进行处理，需要注意的是在 2.3.1 输出 Hello World 时我们编写了 `sys_write` 函数，此时可以将该函数合并到 `fs_read` 中，即在处理 `SYS_write` 调用号时调用新编写的 `fs_read` 而不是之前的 `sys_write` 。

```
case SYS_write:
  // SYSCALL_ARG1(r) = sys_write((int)a[1],(void *)a[2],(size_t)a[3]);
  SYSCALL_ARG1(r) = fs_write((int)a[1],(void *)a[2],(size_t)a[3]);
  break;
case SYS_lseek:
  SYSCALL_ARG1(r) = fs_lseek((int)a[1],(off_t)a[2],(int)a[3]);
  break;
```

之后，在 `navy-app` 中调用接口函数：

```
int _write(int fd, void *buf, size_t count){
  _syscall_(SYS_write, fd, (uintptr_t)buf, count);
}
off_t _lseek(int fd, off_t offset, int whence) {
  _syscall_(SYS_lseek, fd, offset, whence);
}
```

最后实现 `fs_write` 和 `fs_lseek`。 `fs_lseek` 根据参数 `whence` 选择不同的修改偏移量方式，具体内容可通过 `man 2 lseek` 查阅。对于 `fs_write` ，我们在 part2 中 `sys_write` 的中实现了 `FD_STDOUT` 和 `FD_STDERR` 类型的文件，现在需要补充对一般文件的读写并将 `sys_write` 合并到其中：

```
off_t fs_lseek(int fd, off_t offset, int whence){
  int newOfset = 0;
  switch (whence) {
    case SEEK_SET: {
      newOfset = offset;
      break;
    }
    case SEEK_CUR: {
      newOfset = file_table[fd].open_offset + offset;
      break;
    }
    case SEEK_END: {
      newOfset = file_table[fd].size + offset;
      break;
    }
  }
  if (newOfset < 0){
    newOfset = 0;
  }
  else if (newOfset > file_table[fd].size){
    newOfset = file_table[fd].size;
  }
  file_table[fd].open_offset = newOfset;//更新offset
  return newOfset;
}

ssize_t fs_write(int fd, const void *buf, size_t len) {
  int toWrite = (file_table[fd].size - file_table[fd].open_offset > len) ? len :
file_table[fd].size - file_table[fd].open_offset;//要写的长度
  switch(fd) {
    case FD_STDOUT:
    case FD_STDERR: {//输出到串口
      toWrite = len;
      for(int i = 0; i < len; i++){
        _putc(((char*)buf)[i]);
      }
      break;
    }
    case FD_NORMAL: {//输出文件内容
      if (toWrite > 0) {
        ramdisk_write(buf, file_table[fd].disk_offset +
file_table[fd].open_offset, toWrite);//写文件
      }
      break;
    }
    default: {

ramdisk_write(buf,file_table[fd].disk_offset+file_table[fd].open_offset,toWrite);
      break;
    }
  }
  file_table[fd].open_offset += toWrite;//修改偏移
```

```
  return toWrite;
}
```

上述全部实现之后修改 `loader.c` ，运行 `/bin/text` ：

```
uintptr_t loader(_Protect *as, const char *filename) {
  // pa3 part2
  int fd=fs_open("/bin/text",0,0);
  int size=fs_filesz(fd);
  fs_read(fd,DEFAULT_ENTRY,size);
  return (uintptr_t)DEFAULT_ENTRY;
}
```

程序输出 PASS!!! 信息：

```
[src/main.c,19,main] 'Hello World!' from Nanos-lite
[src/main.c,20,main] Build time: 15:08:29, May  5 2021
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x101880, end = 0x370366,
size = 2550502 bytes
[src/main.c,27,main] Initializing interrupt/exception handler...
PASS!!!
nemu: HIT GOOD TRAP at eip = 0x00100032
```

part3

3.1 实验目的

1. 了解特殊文件的处理方式
2. 了解如何将输入设备包装成文件

3.2 实验内容

1. 把 VGA 显存抽象成文件
2. 把设备输入抽象成文件
3. 在 NEMU 中运行仙剑奇侠传

3.3 实验步骤

3.3.1 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() {
  file_table[FD_FB].size=_screen.width*_screen.height*sizeof(uint32_t);
}
```

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

首先需要计算屏幕上的坐标，屏幕上一个像素所占的空间为一个 uint32_t 类型变量的大小，故直接用 offset 除以 `sizeof(uint32_t)` 即可得到屏幕上的坐标；之后调用 `_draw_rect` 将 buff 中的信息写到屏幕上：

```
void fb_write(const void *buf, off_t offset, size_t len) {
  offset /= sizeof(uint32_t);//计算屏幕上的坐标
  //把 buf 中的 len 字节写到屏幕上 offset 处
  _draw_rect(buf, offset % _screen.width, offset / _screen.width, len /
sizeof(uint32_t), 1);
}
```

在 `init_device` 中将 `/proc/dispinfo` 的内容提前写入到字符串 `dispinfo` 中，在 `Nanos-lite` 中获取实际的屏幕大小信息：

实验指导书中已经给出了 `/proc/dispinfo` 内容的例子，只需要按照例子的格式将屏幕信息复制到 `dispinfo` 中即可：

```
void init_device() {
  _ioe_init();
  sprintf(dispinfo, "WIDTH: %d\nHEIGHT: %d", _screen.width, _screen.height);
}
```

实现 `dispinfo_read`，用于把字符串 `dispinfo` 中 offset 开始的 len 字节写到 buf 中：

根据实验指导书的描述，只需要将对应内容复制到 buf 中即可：

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

在文件系统中添加对 `/dev/fb` 和 `/proc/dispinfo` 这两个特殊文件的支持：

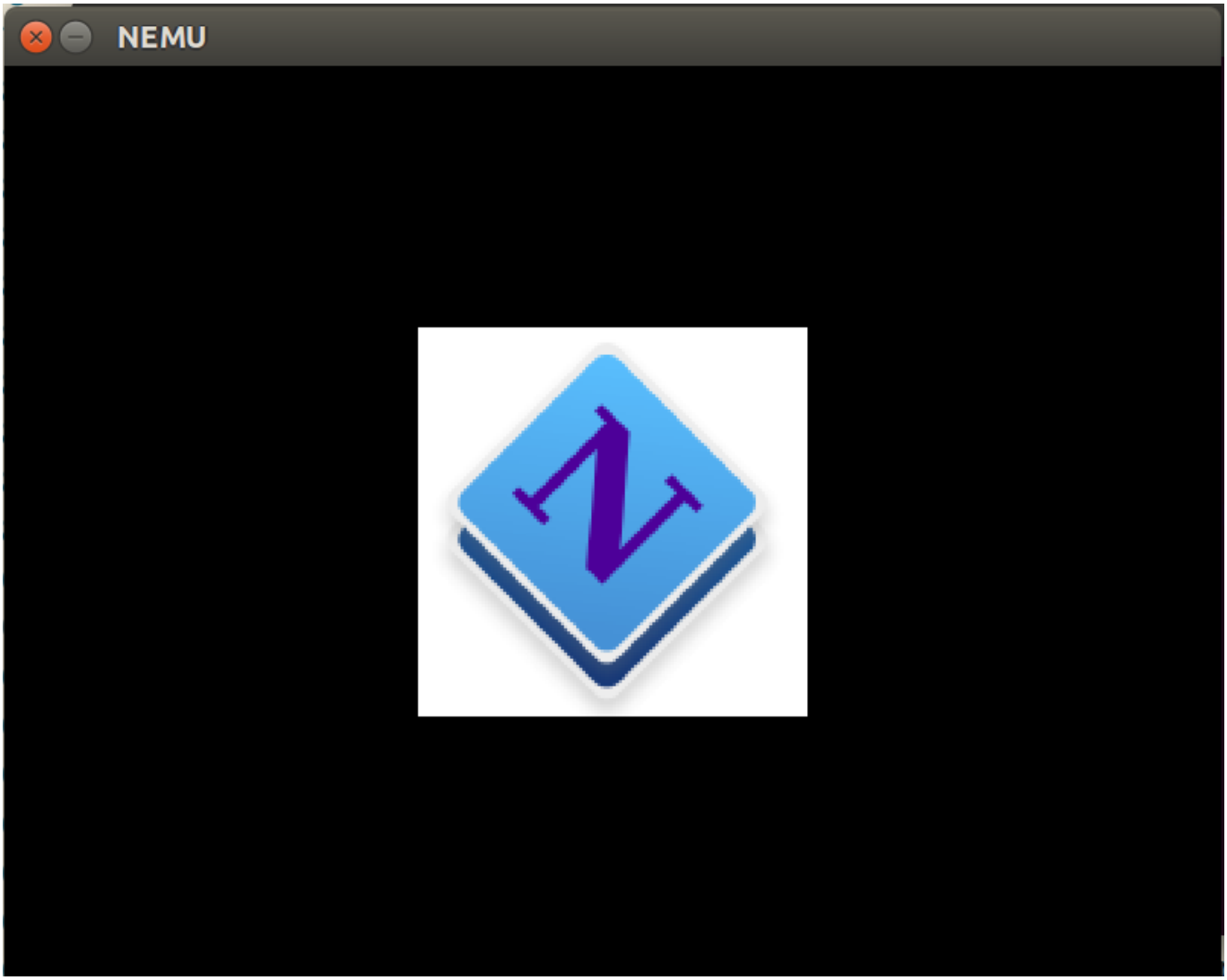
添加对这两个文件的支持需要在 `fs_write` 和 `fs_read` 中添加相应代码，在 `fs_read` 中添加 `FD_DISPINFO` 相关内容用于读取屏幕信息，在 `fs_write` 中添加 `FD_FB` 相关内容用于将信息写到屏幕上：

```
// fs_read 添加内容
if (fd == FD_DISPINFO) {
  dispinfo_read(buf, file_table[fd].disk_offset + file_table[fd].open_offset,
```

```
toRead);//读文件
}

// fs_write 添加内容
case FD_FB: {
  fb_write(buf, file_table[fd].disk_offset + file_table[fd].open_offset, toWrite);
  break;
}
```

实现上述内容之后修改 loader，让 `Nanos-lite` 加载 `/bin/bmptest` 并运行，会看到屏幕上显示 ProjectN 的 Logo。



3.3.2 把设备输入抽象成文件

把输入包装成事件一种简单的方法是把事件以文本的形式表现出来，实验手册中给出了以文本表示事件的方式，我们需要首先实现 `events_read`，之后在文件系统中添加对 `/dev/events` 的支持。

`events_read` 负责将事件写入文件，如果有键盘事件则首先写入键盘事件，没有则写入时钟事件。对于键盘事件和时钟事件可以使用 IOE 定义的 API 获取，通过 `通码=断码+0x8000` 的规则判断区分键盘的断码和通码：

```
size_t events_read(void *buf, size_t len) {
  // return 0;
```

```
  int key = _read_key();
  char keydown_char = (key&0x8000 ? 'd' : 'u');//通码=断码+0x8000
  int keyid = key & ~0x8000;
  if(keyid != _KEY_NONE) {//优先处理按键事件
    snprintf(buf, len, "k%c %s\n", keydown_char, keyname[keyid]);//写入按键事件
    return strlen(buf);
  }
  else {
    unsigned long time_ms=_uptime();
    return snprintf(buf, len, "t %d\n", time_ms) - 1;//写入时间
  }
  return 0;
}
```

在文件系统中添加对 /dev/events 的支持需要在 fs_write 和 fs_read 中添加内容，在 fs_read 中调用 events_read 将事件写入文件，在 fs_write 中只需要修改文件的偏移地址即可，至此 pa3 中的简易文件系统已经全部实现，补充完整的的 fs_write 和 fs_read 代码如下：

```
ssize_t fs_write(int fd, const void *buf, size_t len) {
  int toWrite = (file_table[fd].size - file_table[fd].open_offset > len) ? len :
file_table[fd].size - file_table[fd].open_offset;//要写的长度
  switch(fd) {
    case FD_STDOUT:
    case FD_STDERR: {//输出到串口
      toWrite = len;
      for(int i = 0; i < len; i++){
        _putc(((char*)buf)[i]);
      }
      break;
    }
    case FD_NORMAL: {//输出文件内容
      if (toWrite > 0) {
        ramdisk_write(buf, file_table[fd].disk_offset +
file_table[fd].open_offset, toWrite);//写文件
      }
      break;
    }
    case FD_FB: {
      fb_write(buf, file_table[fd].disk_offset + file_table[fd].open_offset,
toWrite);
      break;
    }
    case FD_EVENTS: {
      toWrite = len;
      break;
    }
    default: {

ramdisk_write(buf,file_table[fd].disk_offset+file_table[fd].open_offset,toWrite);
      break;
    }
```

```
  }
  file_table[fd].open_offset += toWrite;//修改偏移
  return toWrite;
}

ssize_t fs_read(int fd, void *buf, size_t len) {
  if (fd == FD_EVENTS){
    return events_read(buf, len);
  }
  int toRead = (file_table[fd].size - file_table[fd].open_offset > len) ? len :
file_table[fd].size - file_table[fd].open_offset;//要读出的长度
  if (toRead > 0) {
    if (fd == FD_DISPINFO) {
      dispinfo_read(buf, file_table[fd].disk_offset + file_table[fd].open_offset,
toRead);//读文件
    }
    else {
      ramdisk_read(buf, file_table[fd].disk_offset + file_table[fd].open_offset,
toRead);//读文件
    }
    file_table[fd].open_offset += toRead;//修改偏移
  }
  return toRead;
}
```

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。

```
receive event: t 2831
receive event: t 3225
receive event: kd D
receive event: ku D
receive event: t 4431
receive event: kd D
receive event: ku D
receive event: t 5447
receive event: kd SPACE
receive event: t 6146
receive event: t 6549
receive event: t 6925
receive event: t 7298
receive event: ku SPACE
receive event: t 8008
receive event: t 8358
```

3.4.3 在 NEMU 中运行仙剑奇侠传

这部分不需要进行代码编写，只需要获取游戏文件将其放在 `navy-apps/fsimg/share/games/pal/` 目录下，并修改 loader 让其运行 `/bin/pal` 即可，需要注意的是，因为浮点数还未实现，故游戏中不能进行战斗。

4 必答题

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。

- 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。

4.1 游戏读档

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。

4.2 屏幕更新

屏幕更新涉及到将信息输出到屏幕上，因为 pa 将 VGA 显存抽象成文件，所以这一步同样涉及文件操作，对于文件操作与游戏独当的流程类似，在最后通过屏幕输出信息时模拟硬件的 AM 中定义了 `_draw_rect` 用于将屏幕信息输出

5 遇到的问题和解决方法

5.1

第一次进行 make 是出现错误，报错：

```
ln: 无法创建符号链接'src/syscall.h': 不支持的操作
```

通过查阅网络资料得知是因为将项目放在了共享文件夹下，在共享的文件夹内，不可以创建到linux本地目录的链接，导致无法成功 make。

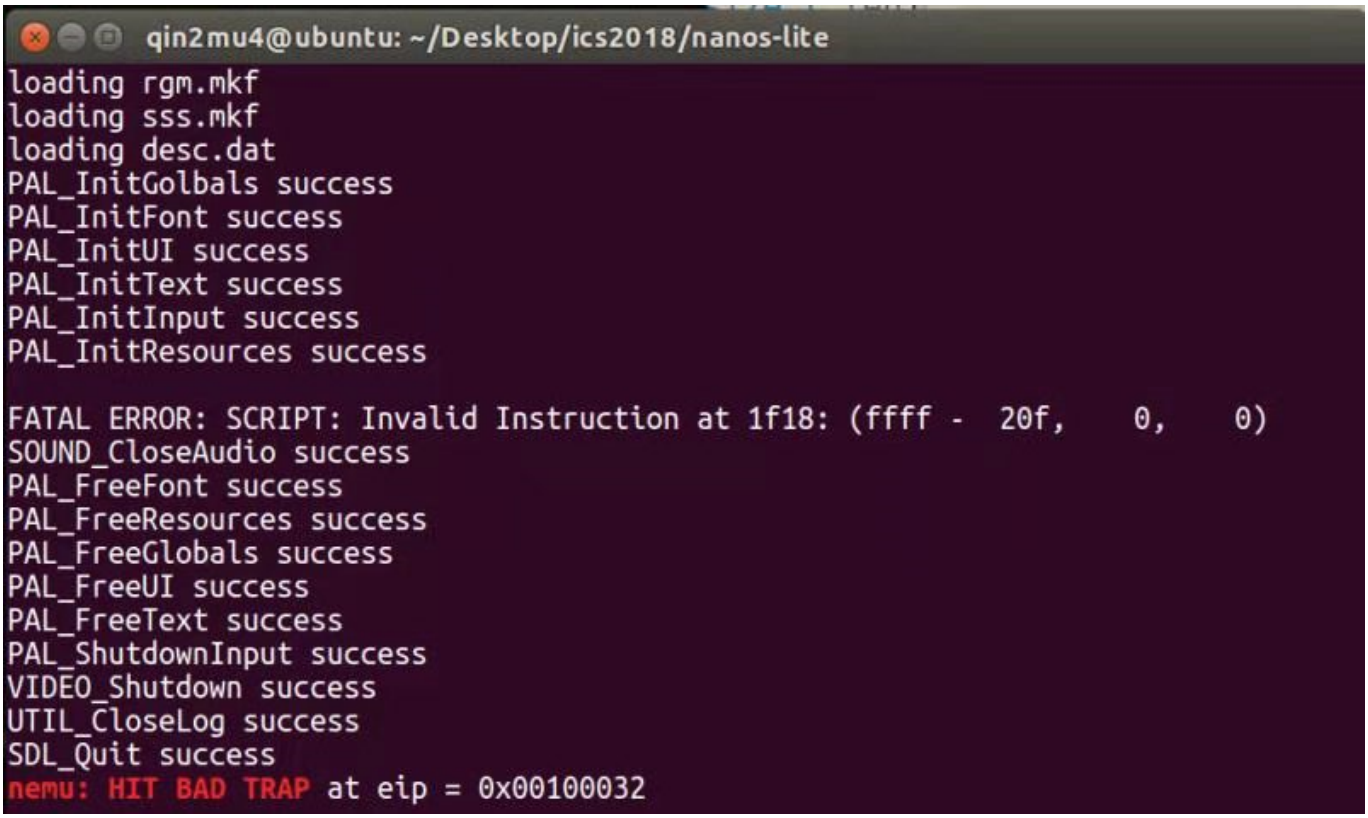
5.2

实现 part2-2 时一直不成功，无法一次性输出完整的语句

经过多次尝试之后发现在 `navy-app` 下执行 make 之后才可以成功，仅在修改文件的目录下即 `navy-apps/libs/libos` 下 make 不可以。

5.3

part3-3 游戏运行到一半，人物要对话时崩溃，错误如下：



请教助教并借鉴另一位大佬的经验，将 `Makefile.compile` 里的 O2 优化改成 O1 ， 游戏成功运行。