PA3-穿越时空的旅程：批处理系统

遗留问题

上次实验，时钟间隔不是1s输出，问题在于在写in指令的执行函数时，少写了`s0=`

```
case 4:
    s0=pio_read_l(id_src->val);
```

修改后时钟正常。键盘输入，所有的字符键还不能识别，而shift、F1等按键可以识别。

概述

实验目的

1. 梳理操作系统概念

2. 学习系统调用，并实现中断机制
3. 了解文件系统的基本内容，实现简易的文件系统
4. 最终实现支持文件系统的操作系统，要求能成功运行仙剑奇侠传

2.2 实验内容

1. 实现自陷操作`_yield()`及其过程
2. 实现用户程序的加载和系统调用, 支撑TRM程序的运行
3. 行仙剑奇侠传并展示批处理系统

3 阶段一

3.1 最简单的操作系统

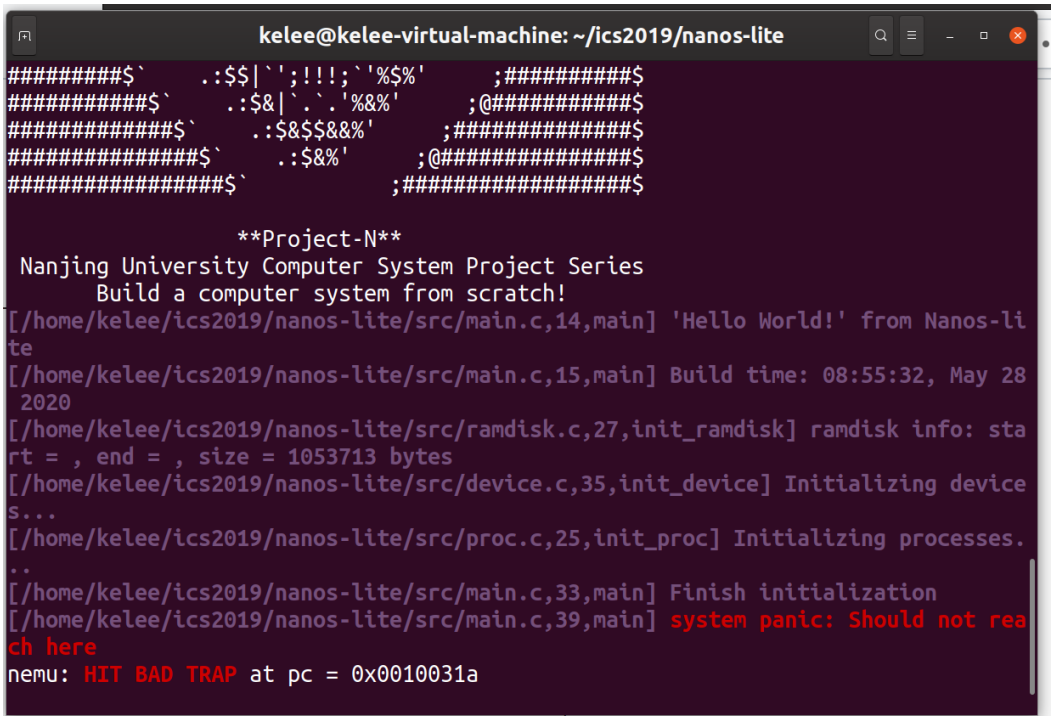
目前Nanos-lite运行的主函数如下：打印logo，输出hello信息和编译时间。然后进行磁盘初始化、设备初始化、文件系统初始化、进程初始化、最后输出panic信息标志结束。

```
int main() {
  printf("%s", logo);
  Log("'Hello World!' from Nanos-lite");
  Log("Build time: %s, %s", __TIME__, __DATE__);

#ifdef HAS_VME
  init_mm();
#endif

  init_ramdisk();

  init_device();

#ifdef HAS_CTE
  init_irq();
#endif

  init_fs();

  init_proc();

  Log("Finish initialization");

#ifdef HAS_CTE
  _yield();
#endif

  panic("Should not reach here");
}
```

在`nanos-lite/`目录下执行

```
make ARCH=$ISA-nemu run
```

执行结果如下：



目前的系统什么事都没有做，要想实现最简单的操作系统，就要实现：

1. 当一个程序执行结束后，可以跳转到操作系统的代码继续执行
2. 操作系统可以加载一个新的用户程序来执行

3.2 等级森严的制度

i386有四个特权级，级数越低，权限越高，权限低的不能访问权限高的资源，反之则可。操作系统运行在0级，用户进程运行在3级。

i386中：

- DPL(Descriptor Privilege Level)属性描述了一段数据所在的特权级
- RPL(Requestor's Privilege Level)属性描述了请求者所在的特权级
- CPL(Current Privilege Level)属性描述了当前进程的特权级,

一次数据的访问操作是合法的, 当且仅当

```
data.DPL >= requestor.RPL        # <1>
data.DPL >= current_process.CPL     # <2>
```

两式同时成立, 注意这里的>=是数值上的(numerically greater).

<1>式表示请求者有权限访问目标数据, <2>式表示当前进程也有权限访问目标数据. 如果违反了上述其中一式, 此次操作将会被判定为非法操作, CPU将会抛出异常信号, 并跳转到一个和操作系统约定好的内存位置, 交由操作系统进行后续处理.

而NEMU中考虑到能力有限，并未实现该机制。

3.3 穿越时空的旅程

硬件提供一种可以限制入口的执行流切换方式. 这种方式就是自陷指令, 程序执行自陷指令之后, 就会陷入到操作系统预先设置好的跳转目标. 这个跳转目标也称为异常入口地址.也就是中断/异常相应机制。

x86中提供`int`指令作为自陷指令，异常入口地址由门描述符来指示。x86提供一个叫IDT的数组来管理门描述符，而IDT的首地址和长度，由IDTR寄存器来指示。因此操作系统事先把IDT准备好，然后调用`lidt`指令，在IDTR中设置好相应值，当程序执行到自陷指令或者触发异常，就会按照设置好的IDT跳转。

当触发异常后硬件的相应过程如下：

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

当异常处理结束后恢复到触发异常前的状态。x86通过`iret`指令从异常处理过程中返回，将栈顶3个元素解释为eip,cs,eflags并恢复。

3.4 将上下文管理抽象为CTE

操作系统在上下文管理上需要知道：

1. 首先当然是引发这次执行流切换的原因, 是程序除0, 非法指令, 还是触发断点, 又或者是程序自愿陷入操作系统? 根据不同的原因, 操作系统都会进行不同的处理.
2. 然后就是程序的上下文了, 在处理过程中, 操作系统可能会读出上下文中的一些寄存器, 根据它们的信息来进行进一步的处理. 例如操作系统读出PC所指向的非法指令, 看看其是否能被模拟执行. 事实上, 通过这些上下文, 操作系统还能实现一些神奇的功能, 你将会在PA4中了解更详细的信息.

上面两点用数据结构`_Event`和`_Context`来表示。

还有两个API：

1. `int _cte_init(_Context* (*handler)(_Event ev, _Context *ctx))` 用于进行CTE相关的初始化操作. 其中它还接受一个来自操作系统的事件处理回调函数的指针, 当发生事件时, CTE将会把事件和相关的上下文作为参数, 来调用这个回调函数, 交由操作系统进行后续处理.
2. `void _yield()` 用于进行自陷操作, 会触发一个编号为_EVENT_YIELD事件.

3.4.1 设置异常入口地址

首先开启`nanos-lite/include/common.h`中的`HAS_CTE`宏

然后跳转到`_cte_init`方法，通过`set_idt`方法李的`lidt`指令来初始化IDTR

3.4.1.1 IDTR

在CPU的寄存器中添加IDTR寄存器

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

3.4.1.2 `lidt`

`lidt`执行函数如下

```
make_EHelper(lidt)
{
  // TODO();
  cpu.IDTR.limit = vaddr_read(id_dest->addr, 2);
  cpu.IDTR.base = vaddr_read(id_dest->addr + 2, 4);
  print_asm_template1(lidt);
}
```

3.4.2 触发自陷操作

3.4.2.1 `raise_intr`

根据异常响应机制步骤如下：

```
void raise_intr(uint32_t NO, vaddr_t ret_addr)
{
  /* TODO: Trigger an interrupt/exception with ``NO''.
   * That is, use ``NO'' to index the IDT.
   */
  uint32_t hi, lo;
  assert(NO <= cpu.IDTR.limit);
  rtl_push((rtlreg_t *)&cpu.eflags);
  rtl_push(&cpu.CS);
  rtl_push(&ret_addr);
  lo = vaddr_read(cpu.IDTR.base + 8 * NO, 4) & 0x0000ffffu;
  hi = vaddr_read(cpu.IDTR.base + 8 * NO + 4, 4) & 0xffff0000u;
#ifdef DEBUG
  uint32_t target_addr = hi | lo;
  Log("Target_Addr=0x%x", target_addr);
#endif
  rtl_j(hi | lo);
}
```

3.4.2.2 寄存器

在CPU的寄存器中添加CS寄存器

```
  rtlreg_t CS;
```

3.4.2.3 寄存器初始化

在`nemu/src/isa/x86/init.c`中的`restart`方法中添加初始化代码

```
static void restart() {
  /* Set the initial program counter. */
  cpu.pc = PC_START;
  cpu.CS=8;
  cpu.eflags.val=0x2;
}
```

3.4.2.4 `int`

在实现自陷的`int`指令中调用`raise_intr`函数

```

make_EHelper(int)
{
  // TODO();
extern void raise_intr(uint32_t NO, vaddr_t ret_addr);
  raise_intr(id_dest->val, decinfo.seq_pc);
  print_asm("int %s", id_dest->str);

  difftest_skip_dut(1, 2);
}
```

3.4.3 保存上下文

3.4.3.1 `pusha`

需要实现`pusha`指令

```
make_EHelper(pusha) {
  // TODO();
  s0=cpu.esp;
  rtl_push(&cpu.eax);
  rtl_push(&cpu.ecx);
  rtl_push(&cpu.edx);
  rtl_push(&cpu.ebx);
  rtl_push(&s0);
  rtl_push(&cpu.ebp);
  rtl_push(&cpu.esi);
  rtl_push(&cpu.edi);
  print_asm("pusha");
}
```

3.4.3.2 `_Context`

按顺序修改结构体

```
struct _Context
{
  struct _AddressSpace *as;
  uint32_t edi, esi, ebp, esp, ebx, edx, ecx, eax;
  uint32_t irq;
```

```
  uint32_t eip, cs;
  union {
    struct
    {
      uint32_t CF : 1;
      uint32_t dummy0 : 1;
      uint32_t PF : 1;
      uint32_t dummy1 : 1;
      uint32_t AF : 1;
      uint32_t dummy2 : 1;
      uint32_t ZF : 1;
      uint32_t SF : 1;
      uint32_t TF : 1;
      uint32_t IF : 1;
      uint32_t DF : 1;
      uint32_t OF : 1;
      uint32_t OLIP : 2;
      uint32_t NT : 1;
      uint32_t dummy3 : 1;
      uint32_t RF : 1;
      uint32_t VM : 1;
      uint32_t dummy4 : 14;
    };
    uint32_t val;
  } eflags;
};
```

问题一：理解上下文结构体的前世今生

你会在`__am_irq_handle()`中看到有一个上下文结构指针c, c指向的上下文结构究竟在哪里? 这个上下文结构又是怎么来的? 具体地, 这个上下文结构有很多成员, 每一个成员究竟在哪里赋值的?`$ISA-nemu.h`, `trap.S`, 上述讲义文字, 以及你刚刚在NEMU中实现的新指令, 这四部分内容又有什么联系?

答：该指针作为参数传入`__am_irq_handle()`，该函数在`trap.S`中被调用，那么指针是以压栈的形式传入的，结构体是在`nexus-am/am/include/arch/x86-nemu.h`中声明的。赋值是在`trap.S`中`pushal`负责压入寄存器，`pushl $0`是给AS赋值，`pushl %esp`是给`_Context *`参数赋值，然后调用函数`__am_irq_handle()`。

```
__am_asm_trap:
pushal

pushl $0

pushl %esp
call __am_irq_handle

addl $4, %esp

addl $4, %esp
popal
addl $4, %esp
```

```
iret
```

3.4.4 事件分发

3.4.4.1 `__am_irq_handle()`

通过该函数识别出异常，并打包为`_EVENT_TIELD`的自陷事件。

```
_Context *__am_irq_handle(_Context *c)
{
  _Context *next = c;
  if (user_handler)
  {
    _Event ev = {0};
    switch (c->irq)
    {
    case 0x20:
      ev.event = _EVENT_IRQ_TIMER;
      break;
    case 0x80:
      ev.event = _EVENT_SYSCALL;
      break;
    case 0x81:
      ev.event = _EVENT_YIELD;
      break;
    default:
      ev.event = _EVENT_ERROR;
      break;
    }

    next = user_handler(ev, c);
    if (next == NULL)
    {
      next = c;
    }
  }
  return next;
}
```

3.4.4.2 `do_event`

根据事件类型识别出事件，然后输出一句话

```
static _Context *do_event(_Event e, _Context *c)
{
  switch (e.event)
  {
```

```
  case _EVENT_YIELD:
    Log("This is yield.");
    break;
  default:
    panic("Unhandled event ID = %d", e.event);
  }

  return NULL;
}
```

3.4.4.3 实验结果

```
[/home/kelee/ics2019/nanos-lite/src/irq.c,8,do_event] This is yield.
```

3.4.5 恢复上下文

3.4.5.1 `popa`

恢复时，在`trap.S`里面取出上下文和AS，然后通过`popal`指令来恢复寄存器。

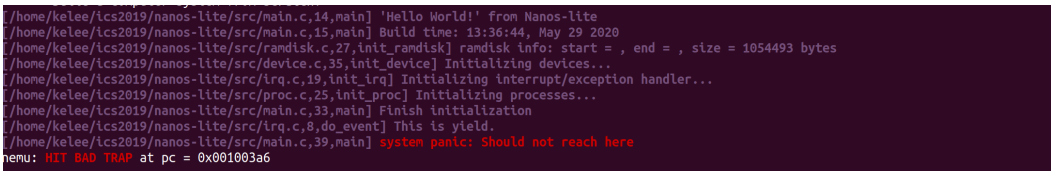
```
make_EHelper(popa) {
  // TODO();
rtl_pop(&cpu.edi);
rtl_pop(&cpu.esi);
rtl_pop(&cpu.ebp);
rtl_pop(&s0);
rtl_pop(&cpu.ebx);
rtl_pop(&cpu.edx);
rtl_pop(&cpu.ecx);
rtl_pop(&cpu.eax);
  print_asm("popa");
}
```

3.4.5.2 `iret`

```
make_EHelper(iret)
{
  // TODO();
  rtl_pop(&s0);
  rtl_j(s0);
  rtl_pop(&s0);
  cpu.CS=s0;
  rtl_pop(&s0);
rtl_li((void *)&cpu.eflags,s0);
  print_asm("iret");
}
```

> 问题二：理解穿越时空的旅程
>
> 从Nanos-lite调用`_yield()`开始, 到从`_yield()`返回的期间, 这一趟旅程具体经历了什么? 软(AM, Nanos-lite)硬(NEMU)件是如何相互协助来完成这趟旅程的? 你需要解释这一过程中的每一处细节, 包括涉及的每一行汇编代码/C代码的行为, 尤其是一些比较关键的指令/变量. 事实上, 上文的必答题"理解上下文结构体的前世今生"已经涵盖了这趟旅程中的一部分, 你可以把它的回答包含进来.
>
> 答：调用`_yield()`后实际上调用的是`int`指令，`int`指令在执行的时候调用了函数`raise_intr()`，`raise_intr ()`函数通过传入的NO确定了下一步跳转的地址，该地址在之前就已经在`_cet_init()`中初始化，然后通过`set_idt`指令把idt的地址放入IDTR寄存器里。这时就找到了跳转地址，通过`__am_irq_handle()`处理函数，根据地址不同封装不同的事件，通过`init_irq`函数，最终调用`user_handle`就是函数`do_event`，处理完返回到`__am_asm_trap`，通过`iret`指令恢复了`raise_intr`压入的参数。

3.5 实验结果



4 阶段二

4.1 加载第一个用户程序

用户程序在`Navy-apps`上进行编译，第一个程序是`dummy`在nanos-lite/目录下执行

```
make ARCH=$ISA-nemu
```

生成`ramdisk.img`镜像文件，可执行文件位于`ramdisk`偏移为0处，访问它就可以得到用户程序的第一个字节。

4.1.1 `loader`

读取的时候把磁盘里的内容读取到内存去，判断类型是否需要加载

```
static uintptr_t loader(PCB *pcb, const char *filename)
{
  // TODO();

  Elf_Ehdr elf;
  //ramdisk_read((void *)0x3000000,0, get_ramdisk_size());
  ramdisk_read((void *)&elf, 0, sizeof(elf));
  Elf_Phdr phdr[elf.e_phnum];
  for (size_t i = 0; i < elf.e_phnum; i++)
  {
    ramdisk_read((void
                      *)&phdr[i],
                 elf.e_phoff + i * elf.e_phentsize,
elf.e_phentsize);
```

```
    if (phdr[i].p_type == PT_LOAD)
    {
      void *va = (void *)phdr[i].p_vaddr;
      int32_t fsize = phdr[i].p_filesz;
      ramdisk_read(va, phdr[i].p_offset, fsize);
    }
  }
  return elf.e_entry;
}
```

4.2 系统调用

4.2.1 do_event

添加系统调用的处理分支

```
case _EVENT_SYSCALL:
    return do_syscall(c);
```

4.2.2 GPR?

```
#define GPR1 eax
#define GPR2 ebx
#define GPR3 ecx
#define GPR4 edx
#define GPR5 esi
#define GPR6 edi
#define GPR7 ebp
```

4.2.3 do_syscall

修改函数让之识别出系统调用，然后进行相应的操作。

```
_Context *do_syscall(_Context *c)
{
  uintptr_t a[4];
  a[0] = c->GPR1;
  a[1] = c->GPR2;
  a[2] = c->GPR3;
  a[3] = c->GPR4;
  switch (a[0])
  {
  case SYS_exit:
    c->GPRx = 0;
    sys_exit(a[1]);
    break;
  case SYS_yield:
    _yield();
    c->GPRx = 0;
    break;
  default:
```

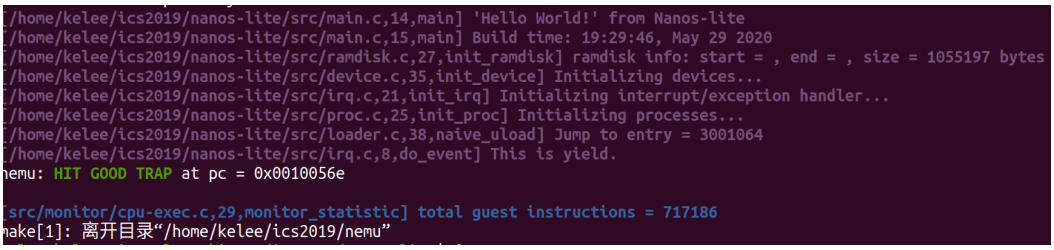
```
    panic("Unhandled syscall ID = %d", a[0]);
  }

  return NULL;
}
```

4.2.4 `sys_exit`

新增系统调用`sys_exit`方法，直接调用`_halt(0)`

```
static inline int32_t sys_exit(int32_t status){
  _halt(0);
  return 0;
}
```

4.2.5 实验结果



4.3 操作系统之上的TRM

4.3.1 标准输出

4.3.1.1 `do_syscall`

新增识别`SYS_write`的分支

```
  case SYS_write:
    c->GPRx = sys_write(a[1], (void *)a[2], a[3]);
    break;
```

4.3.1.2 `sys_write`

增加处理方法

```
static inline int32_t sys_write(int fd, void *buf, size_t len)
{
  switch (fd)
  {
  case 1:
  case 2:
  char c;
  for (int i=0;i<len;i++){
    memcpy(&c,buf+i,1);
    _putc(c);
  }
```

```
    break;
  default:
  panic("Unhandled fd =%d in sys_write",fd);
    return -1;
    break;
  }
  return len;
}
```

4.3.1.3 `_write`

修改`navy-apps/libs/libos/src/nanos.c`的`_write()`中调用系统调用接口函数.

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

4.3.1.4 实验结果

```
[/home/kelee/ics2019/nanos-lite/src/irq.c,21,init_irq] Initializing i
nterrupt/exception handler...
[/home/kelee/ics2019/nanos-lite/src/proc.c,25,init_proc] Initializing
 processes...
[/home/kelee/ics2019/nanos-lite/src/loader.c,38,naive_uload] Jump to
entry = 30010c0
Hello World!
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!
```

4.3.2 堆区管理

4.3.2.1 `do_syscall`

增加系统调用`SYS_brk`

查询手册只需要传入一个地址参数即可

```
  case SYS_brk:
    c->GPRx = sys_brk(a[1]);
    break;
```

4.3.2.2 `sys_brk`

因为系统允许用户自由使用空闲的内存，因此让`SYS_brk`总是返回0。

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

4.3.2.3 `_sbrk`

其工作方式如下:

1. program break一开始的位置位于_end
2. 被调用时, 根据记录的program break位置和参数increment, 计算出新program break
3. 通过SYS_brk系统调用来让操作系统设置新program break
4. 若SYS_brk系统调用成功, 该系统调用会返回0, 此时更新之前记录的program break的位置, 并将旧program break的位置作为_sbrk()的返回值返回
5. 若该系统调用失败, _sbrk()会返回-1

```
void *_sbrk(intptr_t increment)
{
  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;
}
```

4.3.2.4 实验结果

代码实现 前

```
[/home/kelee/ics2019/nanos-lite/src/syscall.c,15,sys_write] buf:H
H[/home/kelee/ics2019/nanos-lite/src/syscall.c,15,sys_write] buf:e
e[/home/kelee/ics2019/nanos-lite/src/syscall.c,15,sys_write] buf:l
l[/home/kelee/ics2019/nanos-lite/src/syscall.c,15,sys_write] buf:l
l[/home/kelee/ics2019/nanos-lite/src/syscall.c,15,sys_write] buf:o
o[/home/kelee/ics2019/nanos-lite/src/syscall.c,15,sys_write] buf:
 [/home/kelee/ics2019/nanos-lite/src/syscall.c,15,sys_write] buf:W
W[/home/kelee/ics2019/nanos-lite/src/syscall.c,15,sys_write] buf:o
o[/home/kelee/ics2019/nanos-lite/src/syscall.c,15,sys_write] buf:r
r[/home/kelee/ics2019/nanos-lite/src/syscall.c,15,sys_write] buf:l
l[/home/kelee/ics2019/nanos-lite/src/syscall.c,15,sys_write] buf:d
d[/home/kelee/ics2019/nanos-lite/src/syscall.c,15,sys_write] buf:
 [/home/kelee/ics2019/nanos-lite/src/syscall.c,15,sys_write] buf:f
f[/home/kelee/ics2019/nanos-lite/src/syscall.c,15,sys_write] buf:r
r[/home/kelee/ics2019/nanos-lite/src/syscall.c,15,sys_write] buf:o
o[/home/kelee/ics2019/nanos-lite/src/syscall.c,15,sys_write] buf:m
m[/home/kelee/ics2019/nanos-lite/src/syscall.c,15,sys_write] buf:
 [/home/kelee/ics2019/nanos-lite/src/syscall.c,15,sys_write] buf:N
N[/home/kelee/ics2019/nanos-lite/src/syscall.c,15,sys_write] buf:a
a[/home/kelee/ics2019/nanos-lite/src/syscall.c,15,sys_write] buf:v
v[/home/kelee/ics2019/nanos-lite/src/syscall.c,15,sys_write] buf:y
y[/home/kelee/ics2019/nanos-lite/src/syscall.c,15,sys_write] buf:-
-[/home/kelee/ics2019/nanos-lite/src/syscall.c,15,sys_write] buf:a
a[/home/kelee/ics2019/nanos-lite/src/syscall.c,15,sys_write] buf:p
p[/home/kelee/ics2019/nanos-lite/src/syscall.c,15,sys_write] buf:p
p[/home/kelee/ics2019/nanos-lite/src/syscall.c,15,sys_write] buf:s
s[/home/kelee/ics2019/nanos-lite/src/syscall.c,15,sys_write] buf:
 [/home/kelee/ics2019/nanos-lite/src/syscall.c,15,sys_write] buf:f
f[/home/kelee/ics2019/nanos-lite/src/syscall.c,15,sys_write] buf:o
o[/home/kelee/ics2019/nanos-lite/src/syscall.c,15,sys_write] buf:r
```

代码实现后

```
Hello World from Navy-apps for the 449th time!
[/home/kelee/ics2019/nanos-lite/src/syscall.c,15,sys_write] buf:Hello World from Navy-apps for the 450th time!

Hello World from Navy-apps for the 450th time!
[/home/kelee/ics2019/nanos-lite/src/syscall.c,15,sys_write] buf:Hello World from Navy-apps for the 451th time!

Hello World from Navy-apps for the 451th time!
[/home/kelee/ics2019/nanos-lite/src/syscall.c,15,sys_write] buf:Hello World from Navy-apps for the 452th time!

Hello World from Navy-apps for the 452th time!
[/home/kelee/ics2019/nanos-lite/src/syscall.c,15,sys_write] buf:Hello World from Navy-apps for the 453th time!

Hello World from Navy-apps for the 453th time!
[/home/kelee/ics2019/nanos-lite/src/syscall.c,15,sys_write] buf:Hello World from Navy-apps for the 454th time!

Hello World from Navy-apps for the 454th time!
[src/monitor/cpu-exec.c,29,monitor_statistic] total guest instructions = 31635618
make[1]: 离开目录“/home/kelee/ics2019/nemu”
```

> 问题三：hello程序是什么, 它从而何来, 要到哪里去
>
> 我们知道`navy-apps/tests/hello/hello.c`只是一个C源文件, 它会被编译链接成一个ELF文件. 那么, hello程序一开始在哪里? 它是怎么出现内存中的? 为什么会出现在目前的内存位置? 它的第一条指令在哪里? 究竟是怎么执行到它的第一条指令的? hello程序在不断地打印字符串, 每一个字符又是经历了什么才会最终出现在终端上?
>
> 答：最开始是位于`build/hello_x86`中，通过`Nanos-lite`的Makefile初始化到了ramdisk里，然后`loader`使用`ramdisk_read`来读取elf内容，最开始偏移为0，把内容读取出来后写入内存，返回入口以后就可运行`hello`了。进入hello后程序调用了`write`和`printf`两个命令，不过都是系统调用`write`，调用`_putc`，再调用`outb`来进行端口输出，`out`指令实际调用了`pio_write_common`来把数据传入端口，然后通过`map_write`函数来把数据拷贝到map里面，然后就可以在vga上显示了。

5 阶段三

5.1 简易的文件系统

5.1.1 `Finfo`

增加偏移量

```
typedef struct {
  char *name;
  size_t size;
  size_t disk_offset;
  size_t open_offset;
  ReadFn read;
  WriteFn write;
} Finfo;
```

5.1.2 s_putc

进行字符串输出

```
static inline size_t s_putc(const void *buf, size_t offset,
size_t len)
{
  char *start = (char *)(buf + offset);
  for (int i = 0; i < len; i++)
  {
    _putc(start[i]);
  }
  return len;
}
```

5.1.3 fd_check

主要用于检查文件是否存在

```
static inline void fd_check(int fd)
{
  assert(fd >= 0 && fd < NR_FILES);
  return;
}
```

5.1.4 fs_open

打开文件，其实就是找到文件

```
int fs_open(const char *pathname, int flags, int mode)
{
  for (int i = 0; i < NR_FILES; i++)
  {
    if (strcmp(pathname, file_table[i].name) == 0)
      return i;
  }
  printf("%s\n", pathname);
  panic("Could't find the file");
  return -1;
}
```

5.1.5 fs_read

注意偏移量不能越过文件边界，使用`ramdisk_read`进行文件写入。

```
size_t fs_read(int fd, void *buf, size_t len)
{
  fd_check(fd);
  size_t sz;
  if (file_table[fd].read == NULL)
  {
    sz = file_table[fd].open_offset + len < +file_table[fd].size
? len : file_table[fd].size - file_table[fd].open_offset;
    sz = ramdisk_read(buf, file_table[fd].disk_offset +
file_table[fd].open_offset, sz);
    file_table[fd].open_offset += sz;
    return sz;
  }
  else
  {
    sz = len;
    if (file_table[fd].size && file_table[fd].open_offset + len >
file_table[fd].size)
    {
      sz = file_table[fd].size - file_table[fd].open_offset;
    }
    sz = file_table[fd].read(buf, file_table[fd].open_offset,
sz);
    file_table[fd].open_offset += sz;
    return sz;
  }
}
```

5.1.6 fs_write

注意偏移量不能越过文件边界，使用`ramdisk_write`进行文件写入。

```
size_t fs_write(int fd, const void *buf, size_t len)
{
  fd_check(fd);
  size_t sz;
  if (file_table[fd].write == NULL)
  {
    sz = file_table[fd].open_offset + len < +file_table[fd].size
? len : file_table[fd].size - file_table[fd].open_offset;
    sz = ramdisk_write(buf, file_table[fd].disk_offset +
file_table[fd].open_offset, sz);
    file_table[fd].open_offset += sz;
    return sz;
  }
  else
```

```
  {
    sz = len;
    if (file_table[fd].size && file_table[fd].open_offset + len >
file_table[fd].size)
    {
      sz = file_table[fd].size - file_table[fd].open_offset;
    }
    sz = file_table[fd].write(buf, file_table[fd].open_offset,
sz);
    file_table[fd].open_offset += sz;
    return sz;
  }
}
```

5.1.7 fs_lseek

根据设置的不同来确定位置。

```
size_t fs_lseek(int fd, size_t offset, int whence)
{
  fd_check(fd);
  switch (whence)
  {
  case SEEK_SET:
    file_table[fd].open_offset = offset;
    break;
  case SEEK_CUR:
    file_table[fd].open_offset += offset;
    break;
  case SEEK_END:
    file_table[fd].open_offset = file_table[fd].size + offset;
    break;
  default:
    panic("lseek whence error!");
    break;
  }
  return file_table[fd].open_offset;
}
```

5.1.8 loader

修改loader可以加载文件

```
static uintptr_t loader(PCB *pcb, const char *filename)
{
  int fd = fs_open(filename, 0, 0);
  if (fd == -1)
  {
    panic("loader: can't open file %s!", filename);
  }
  Elf_Ehdr elf_header;
  fs_read(fd, (void *)&elf_header, sizeof(Elf_Ehdr));
```

```
  if (memcmp(elf_header.e_ident, ELFMAG, SELFMAG))
    panic("file %s ELF format error!", filename);
  for (size_t i = 0; i < elf_header.e_phnum; ++i)
  {
    Elf_Phdr phdr;
    fs_lseek(fd, elf_header.e_phoff + elf_header.e_phentsize * i,
SEEK_SET);
    fs_read(fd, (void *)&phdr, elf_header.e_phentsize);
    if (phdr.p_type == PT_LOAD)
    {
      fs_lseek(fd, phdr.p_offset, SEEK_SET);
      fs_read(fd, (void *)phdr.p_vaddr, phdr.p_filesz);
      memset((void *)(phdr.p_vaddr + phdr.p_filesz), 0,
phdr.p_memsz - phdr.p_filesz);
    }
  }
  fs_close(fd);
  return elf_header.e_entry;
}
```

5.1.9 nanos.c

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

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

void *_sbrk(intptr_t increment)
{
  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;
}

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

```
}

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

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

5.1.10 syscall.c

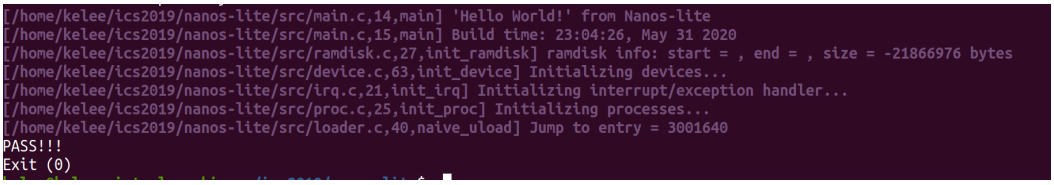
让系统调用转化为文件操作。

```
static inline int32_t sys_open(const char *path, int flags, int
mode) {
  return fs_open(path, flags, mode);
}

static inline int32_t sys_read(int fd, void *buf, size_t len) {
  return fs_read(fd, buf, len);
}

static inline int32_t sys_write(int fd, void *buf, size_t len) {
  return fs_write(fd, buf, len);
}

static inline int32_t sys_lseek(int fd, uint32_t offset, int
whence) {
  return fs_lseek(fd, offset, whence);
}

static inline int32_t sys_close(int fd) {
  return fs_close(fd);
}
```

5.1.11 do_syscall

```
 case SYS_open:
    c->GPRx=sys_open((void*)a[1],a[2],a[3]);
    break;
     case SYS_read:
      c->GPRx = sys_read(a[1], (void *)a[2], a[3]);
      break;
      case SYS_close:
      c->GPRx = sys_close(a[1]);
      break;
      case SYS_lseek:
      c->GPRx = sys_lseek(a[1], a[2], a[3]);
      break;
```

5.1.12 实验结果



5.2 操作系统之上的IOE

5.2.1 `serial_write`

```
size_t serial_write(const void *buf, size_t offset, size_t len)
{
  char c;
  for (size_t i = 0; i < len; ++i)
  {
    memcpy(&c,buf+i,1);
    _putc(c);
  }
  return len;
}
```

5.2.2 `events_read`

```
size_t events_read(void *buf, size_t offset, size_t len)
{
  int key = read_key();
  if (key & 0x8000)
  {
    sprintf(buf, "kd %s\n", keyname[key & ~0x8000]);
  }
  else if ((key & ~0x8000) != _KEY_NONE)
  {
    sprintf(buf, "ku %s\n", keyname[key & ~0x8000]);
  }
  else
  {
```

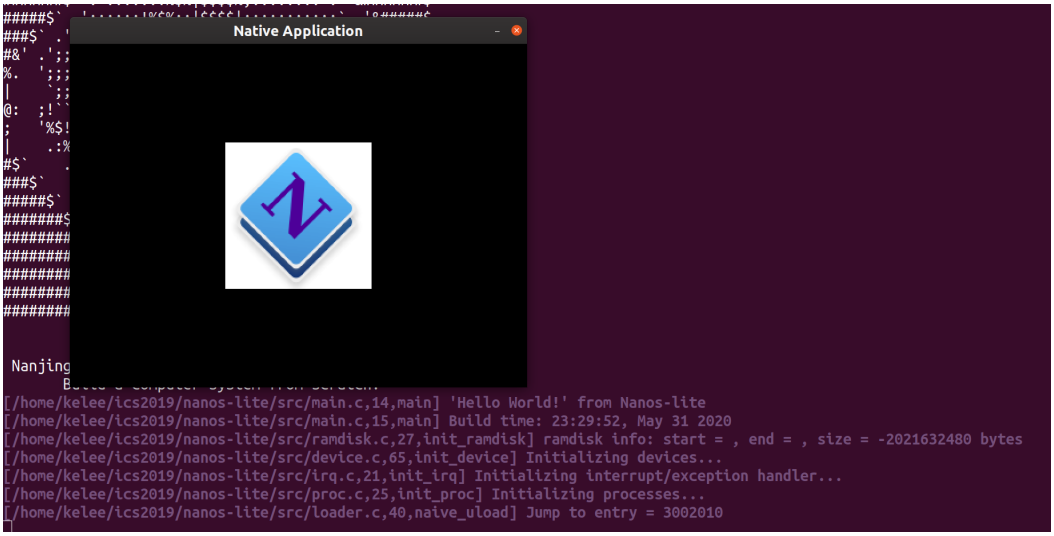
```
    sprintf(buf, "t %d\n", uptime());
  }
  return strlen(buf);
}
```

5.2.3 VGA

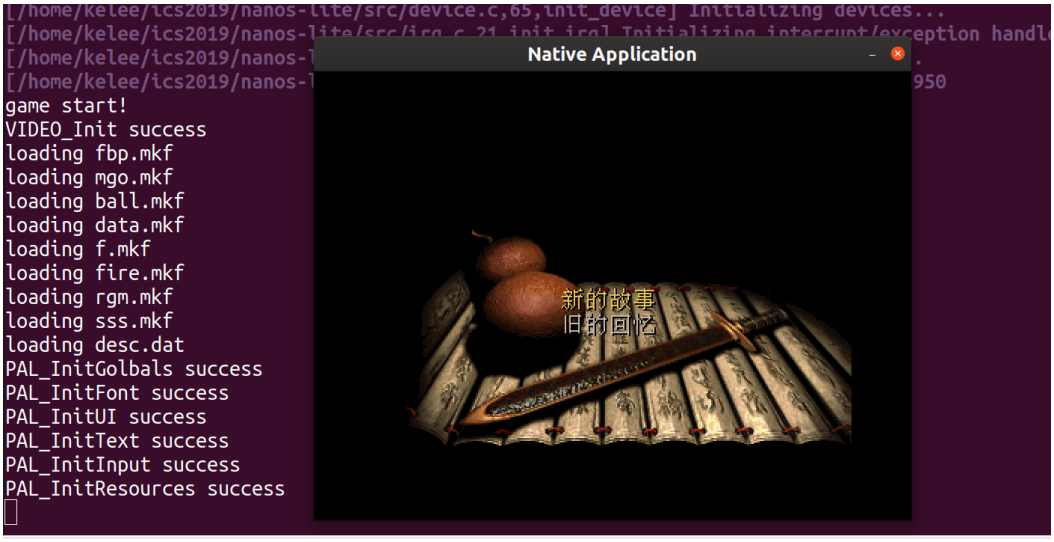
```
static char dispinfo[128] __attribute__((used)) = {};
static int screen_h, screen_w;
size_t get_dispinfo_size()
{
  return strlen(dispinfo);
}
size_t dispinfo_read(void *buf, size_t offset, size_t len)
{
  strncpy(buf, dispinfo + offset, len);
  return len;
}
size_t fb_write(const void *buf, size_t offset, size_t len)
{
  int x, y;
  assert(offset + len <= (size_t)screen_h * screen_w * 4);
  x = (offset / 4) % screen_w;
  y = (offset / 4) / screen_w;
  assert(x + len < (size_t)screen_w * 4);
  draw_rect((void *)buf, x, y, len / 4, 1);
  return len;
}
size_t fbsync_write(const void *buf, size_t offset, size_t len)
{
  draw_sync();
  return len;
}
void init_device()
{
  Log("Initializing devices...");
  _ioe_init();
  // TODO: print the string to array `dispinfo` with the format
  // described in the Navy-apps convention
  screen_h = screen_height();
  screen_w = screen_width();
  sprintf(dispinfo, "WIDTH:%d\nHEIGHT:%d\n", screen_w, screen_h);
}
```

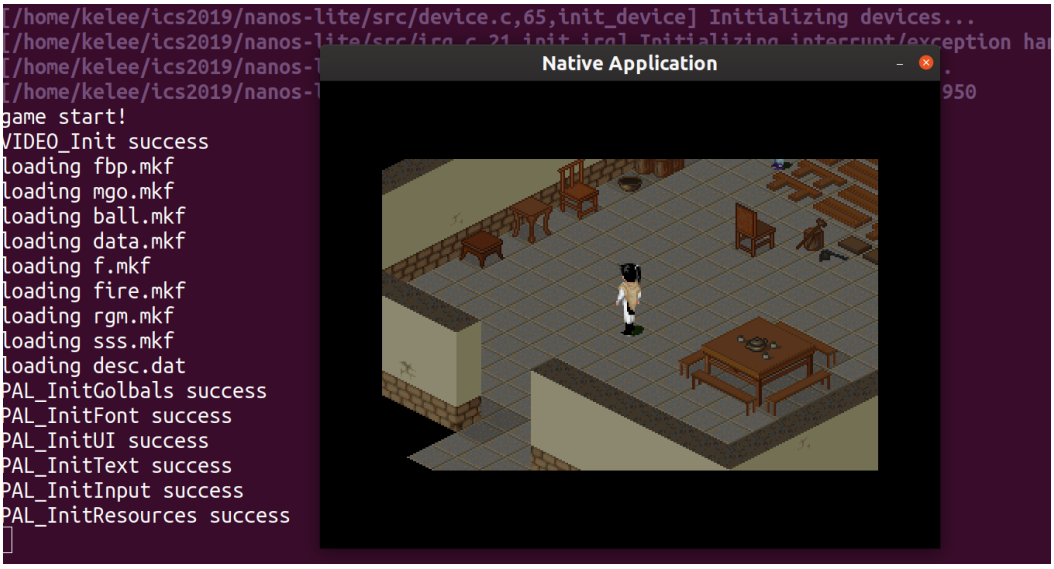
5.2.4 实验结果

```
[/home/kelee/ics2019/nanos-lite/src/main.c,14,main] 'Hello World!' from Nanos-lite
[/home/kelee/ics2019/nanos-lite/src/main.c,15,main] Build time: 23:41:51, May 31 2020
[/home/kelee/ics2019/nanos-lite/src/ramdisk.c,27,init_ramdisk] ramdisk info: start = , end = , size = -853838304 bytes
[/home/kelee/ics2019/nanos-lite/src/device.c,65,init_device] Initializing devices...
[/home/kelee/ics2019/nanos-lite/src/irq.c,21,init_irq] Initializing interrupt/exception handler...
[/home/kelee/ics2019/nanos-lite/src/proc.c,25,init_proc] Initializing processes...
[/home/kelee/ics2019/nanos-lite/src/loader.c,40,naive_uload] Jump to entry = 3001110
Start to receive events...
receive time event for the 1024th time: t 258
receive time event for the 2048th time: t 259
receive time event for the 3072th time: t 260
receive time event for the 4096th time: t 261
receive time event for the 5120th time: t 262
```



5.3 运行仙剑奇侠传





5.4 批处理系统

5.4.1 do_syscall

```
 case SYS_execve:
    c->GPRx = sys_execve((void *)a[1], (char *const *)(void
*)a[2], (char *const *)(void *)a[3]);
  break;
```

5.4.2 sys_execve

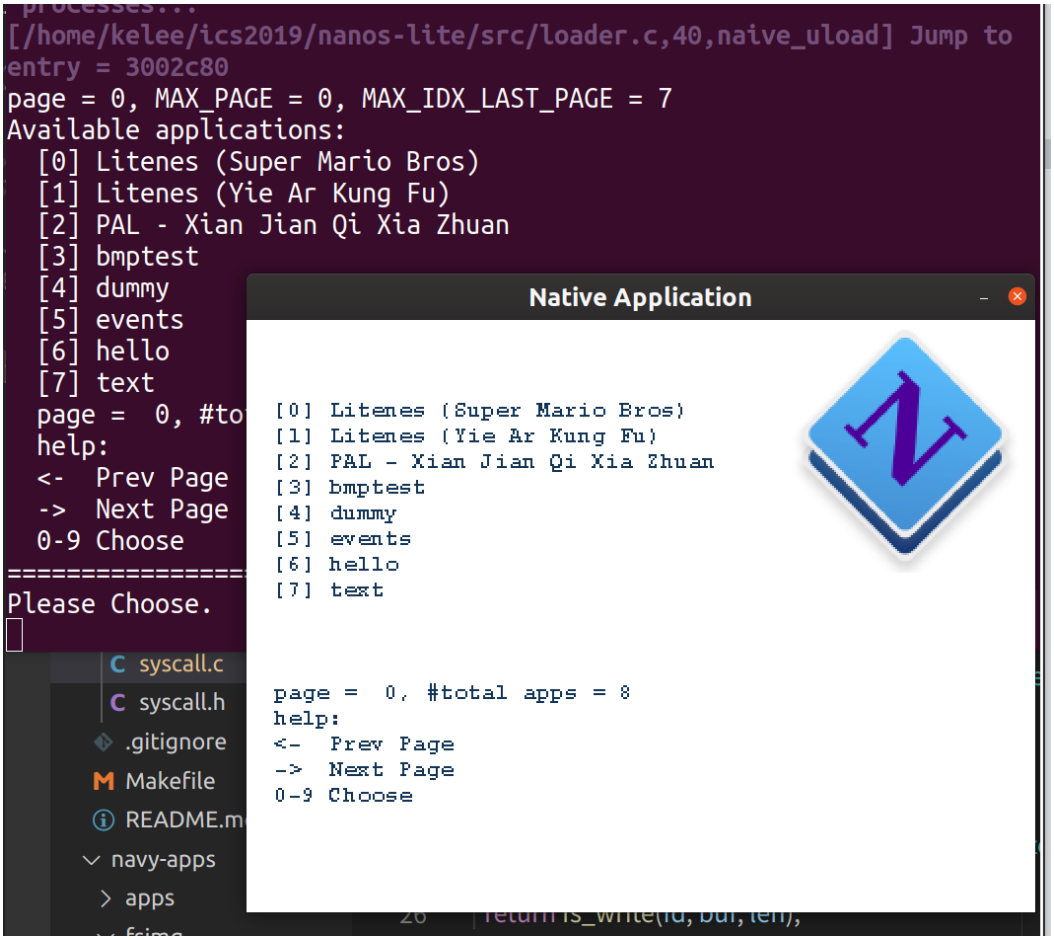
```
static inline int32_t sys_execve(const char *pathname, char
*const argv[], char *const envp[]) {
  return naive_uload(NULL, pathname);
}
```

5.4.3 sys_exit

```
static inline int32_t sys_exit(int32_t status)
{
  // _halt(0);
return sys_execve("/bin/init", NULL, NULL);
}
```

注意两个函数有先后顺序。

5.4.4 实验结果



问题四：仙剑奇侠传究竟如何运行

运行仙剑奇侠传时会播放启动动画, 动画中仙鹤在群山中飞过. 这一动画是通过`navy-apps/apps/pal/src/main.c`中的`PAL_SplashScreen()`函数播放的. 阅读这一函数, 可以得知仙鹤的像素信息存放在数据文件`mgo.mkf`中. 请回答以下问题: 库函数, libos, Nanos-lite, AM, NEMU是如何相互协助, 来帮助仙剑奇侠传的代码从`mgo.mkf`文件中读出仙鹤的像素信息, 并且更新到屏幕上? 换一种PA的经典问法: 这个过程究竟经历了些什么?

答：在加载`mgo.mkf`时，实际上调用了`fopen`库函数，库函数通过系统调用打开了文件，然后调用`PAL_MKFReadChunk`读取文件内容。调用的是libc中的`fseek`，`fread`函数而实际就是调用了系统调用了。然后通过`Decompress`函数来实现信息解压读取内容，然后通过一个循环来实现渐变，然后调用`VIDEO_SetPlatte`，其内部实际上调用了`void SDL_SetPalette`，把图像拷贝到了渲染的接口，然后调用`NDL_DrawRect`，这是系统的库函数，该函数通过`fseek`，`fwrite`把内容写入内存，然后通过文件指定的写出方式进行写出，其实就是`{"/dev/fbsync", 0, 0, 0, invalid_read, fbsync_write},`