南開大學
Nankai University

南　开　大　学

计 算 机 学 院

计算机系统设计

PA **实验三报告**

朱浩泽 1911530

年级：2019 级

专业：计算机科学与技术

指导教师：卢冶

2022 年 5 月 16 日

目录

一、 **概述** 1

（一） 实验目的 1

（二） 实验内容 1

二、 **阶段一 加载操作系统的第一个用户程序** 1

（一） 环境变量的修改 1

（二） 实现 loader 1

（三） 准备 IDT 1

（四） cs 寄存器的实现 3

（五） int 指令的实现 3

（六） pusha 和 popa 指令实现 4

（七） 系统调用 6

（八） 实验结果 7

三、 bug **总结** 7

（一） 实现 loader 后出现段错误 7

（二） HAS ASYE 宏开启 8

一、 概述

(一) 实验目的

- 理解操作系统概念
- 将机器底层和上层应用实现联系
- 了解操作系统中最常见的系统调用，并实现中断机制
- 了解文件系统的基本内容，实现简易文件系统

(二) 实验内容

实现一个简易的系统，主要实现系统的中断过程，尤其是系统调用，于此同时，实现文件系统（通过系统调用写入文件系统）。**在不听劝告的情况下使用 64 位系统，折腾 gcc 以及各种环境问题，提升工程能力**。

二、 阶段一 加载操作系统的第一个用户程序

(一) 环境变量的修改

首先，我们需要在 NEMU 上运行一个裁剪版本的 Nanos 操作系统，然后在这个操作系统上运行我们的程序。为了在 Nanos 上加载我们的可执行程序，我们需要更改 app 的编译路径，将 navy-apps 的 makefile 中的 ISA 路径指向 x86，然后我们编译测试程序生成可执行文件，等待后续的加载。

(二) 实现 loader

loader 将 ramdisk 的内容移动到正确的内存位置 0x4000000(通过 navy-apps/Makefile.compile 中的 LDFLAGS 变量可以知道)，将比特串放在正确的位置后可实现程序的运行。

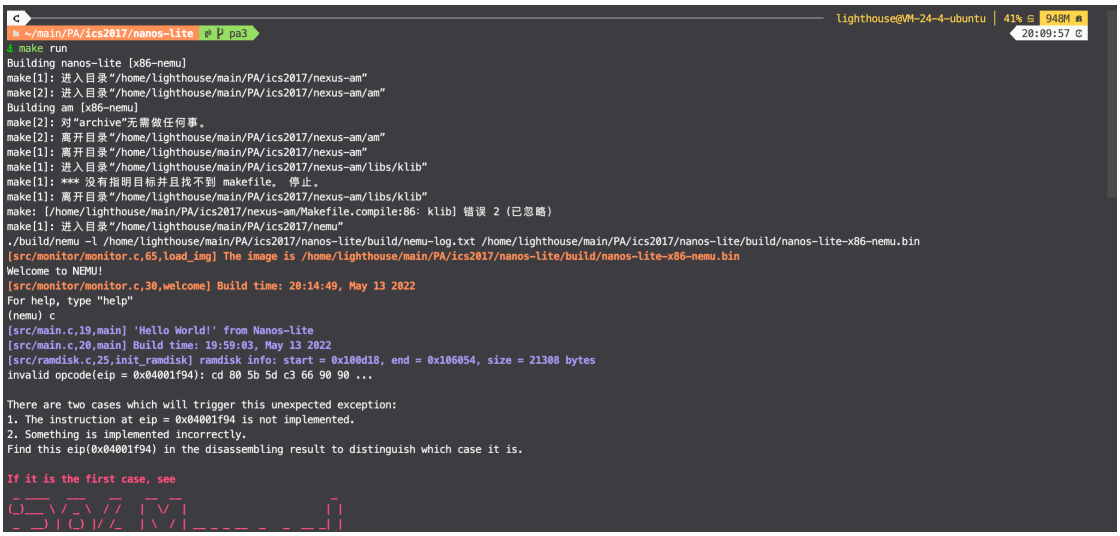
修改 nanos-lita/src/loader.c 中的 loader 函数

```
uintptr_t loader(_Protect *as, const char *filename) {
  // TODO();
  ramdisk_read(DEFAULT_ENTRY, 0, RAMDISK_SIZE);
  return (uintptr_t)DEFAULT_ENTRY;
}
```

此处出现了第一个离谱的 bug，见 bug 总结/实现 loader 后出现段错误

(三) 准备 IDT

在实现了建议的 loader 函数后，我们对 dummy.c 程序进行测试，结果发现有指令没有实现，通过查阅 i386 手册可以知道是 int 指令没有实现。但是，我们在实现 int 指令前，我们先要实现中断描述符表（IDT），因为实际的问题来源于调用中断信号 0x80 时使用了 int 指令。



在 IDT 准备过程中，首先需要一个寄存器存放 IDT 的首地址和长度，所以我们引入了 IDTR 寄存器存放这些信息。该寄存器在 nemu/include/cpu/reg.h 中实现，代码如下：

```
struct IDTR
{
  /* data */
  uint32_t base;
  uint16_t limit;
} idtr;
```

该寄存器会在 _asye_init() 函数中初始化设置 idt 的首地址和长度，传入对应的数据。

然后我们要实现 lidt 指令，该指令会将 idtr 的首地址和长度写入寄存器中，然后调用 _asm_lidt() 函数来实现 IDT 的设置。

然后我们需要实现 lidt 指令，将操作数信息从 eax 寄存器中读出。该指令首先还是需实现翻码函数 lidt_a，该译码函数在 nemu/include/cpu/decode.h 中进行注册，然后在 nemu/src/decode/decode.c 中进行实现

```
make_DHelper(lidt_a) {
  decode_op_a(eip, id_dest, true);
}
```

然后我们在 nemu/src/cpu/exec/all-instr.h 中对 lidt 指令进行注册，并在 nemu/src/cpu/exec/system.c 中进行实现，将操作数信息从 eax 寄存器中读出。

```
make_EHelper(lidt) {
  // TODO();
  t1 = id_dest -> val;
  rtl_lm(&t0, &t1, 2);
  cpu.idtr.limit = t0;

  t1 = id_dest -> val + 2;
  rtl_lm(&t0, &t1, 4);
  cpu.idtr.base = t0;

#ifdef DEBUG
  Log("idtr.limit=0x%x", cpu.idtr.limit);
  Log("idtr.base=0x%x", cpu.idtr.base);
```

```
#endif
  print_asm_template1(lidt);
}
```

最后我们在 exec.c 中进行 optable 的注册

```
make_group(gp7,
  EMPTY, EMPTY, EMPTY, IDEX(lidt_a, lidt),
  EMPTY, EMPTY, EMPTY, EMPTY)
```

（四） cs 寄存器的实现

我们在 nemu/include/cpu/reg.h 文件中对 cs 寄存器进行初始化，将其放在 cpu_state 中，并在 nemu/src/monitor/monitor.c 中进行初始化，赋初值为 8。

（五） int 指令的实现

在我们实现了 IDT 和 cs 寄存器后，我们在 nemu/src/cpu/intr.c 文件中的 raise_intr 函数中模拟了异常出现后的硬件操作，具体过程就是寄存器压栈，根据索引取出 IDT 数组信息，处理结构体信息得到跳转目标地址并设置跳转。

```
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.
   */

  // TODO();
  memcpy(&t1, &cpu.eflags, sizeof(cpu.eflags));
  rtl_li(&t0, t1);
  rtl_push(&t0);
  rtl_push(&cpu.cs);
  rtl_li(&t0, ret_addr);
  rtl_push(&t0);
  vaddr_t gate_addr = cpu.idtr.base + NO * sizeof(GateDesc);
  // Log("%d %d %d\n", gate_addr, cpu.idtr.base, cpu.idtr.limit);
  assert(gate_addr <= cpu.idtr.base + cpu.idtr.limit);

  uint32_t off_15_0 = vaddr_read(gate_addr,2);
  uint32_t off_32_16 = vaddr_read(gate_addr+sizeof(GateDesc)-2,2);
  uint32_t target_addr = (off_32_16 << 16) + off_15_0;
#ifdef DEBUG
  Log("target_addr=0x%x",target_addr);
#endif
  decoding.is_jmp = 1;
  decoding.jmp_eip = target_addr;
}
```

在这些都实现后，我们便可以着手进行 int 指令的实现。我们 int 执行函数的主体实现在了 nemu/src/cpu/exec/system.c 中，并在 exe.c 中对 int 指令进行了注册

```
make_EHelper(int) {
   // TODO();

   uint8_t NO = id_dest -> val & 0xff;
```

```
  raise_intr(NO, decoding.seq_eip);
  print_asm("int %s", id_dest->str);

#ifdef DIFF_TEST
  diff_test_skip_nemu();
#endif
}
```

（六） pusha 和 popa 指令实现

实现完 int 指令之后，本以为没问题行了，进行测试，结果发现仍然后 invalid code。查阅 i386 手册，可以知道是 pusha 指令没有实现，经过思考可以得知，popa 指令一定也没有实现。查看手册的说明，pusha 的作用是将所有的寄存器分别压栈和出栈。

PUSHA/PUSHAD — Push all General Registers

Opcode	Instruction	Clocks	Description
60	PUSHA	18	Push AX, CX, DX, BX, original SP, BP, SI, and DI
60	PUSHAD	18	Push EAX, ECX, EDX, EBX, original ESP, EBP, ESI, and EDI

Operation

```
IF OperandSize = 16 (* PUSHA instruction *)
THEN
   Temp ← (SP);
   Push(AX);
   Push(CX);
   Push(DX);
   Push(BX);
   Push(Temp);
   Push(BP);
   Push(SI);
   Push(DI);
ELSE (* OperandSize = 32, PUSHAD instruction *)
   Temp ← (ESP);
   Push(EAX);
   Push(ECX);
   Push(EDX);
   Push(EBX);
   Push(Temp);
   Push(EBP);
   Push(ESI);
   Push(EDI);
FI;
```

我们在 nemu/src/cpu/exec/data-mov.c 中实现 pusha 指令如下

```
make_EHelper(pusha) {
  // TODO();
  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");
```

```
}
```

popa 的作用是将所有的寄存器分别出栈和压栈。

PUSHA/PUSHAD — Push all General Registers

Opcode	Instruction	Clocks	Description
60	PUSHA	18	Push AX, CX, DX, BX, original SP, BP, SI, and DI
60	PUSHAD	18	Push EAX, ECX, EDX, EBX, original ESP, EBP, ESI, and EDI

Operation

```
IF OperandSize = 16 (* PUSHA instruction *)
THEN
   Temp ← (SP);
   Push(AX);
   Push(CX);
   Push(DX);
   Push(BX);
   Push(Temp);
   Push(BP);
   Push(SI);
   Push(DI);
ELSE (* OperandSize = 32, PUSHAD instruction *)
   Temp ← (ESP);
   Push(EAX);
   Push(ECX);
   Push(EDX);
   Push(EBX);
   Push(Temp);
   Push(EBP);
   Push(ESI);
   Push(EDI);
FI;
```

我们在 nemu/src/cpu/exec/data-mov.c 中实现 popa 指令如下

```
make_EHelper(popa) {
  // TODO();
  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");
}
```

在这之后，我们需要重新组织 TrapFrame（定义在 nexus-am/am/arch/x86-nemu/include/arch.h 的 _ResSet 结构体中），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 esi, ebx, eax, eip, edx, error_code, eflags, ecx, cs, esp, edi, ebp;
  uintptr_t edi, esi, ebp, esp, ebx, edx, ecx, eax;
  int irq;
  uintptr_t error_code;
  uintptr_t eip;
  uintptr_t cs;
```

```
  uintptr_t eflags;
};
```

在这里的时候，我们又出现了段错误的 bug，见 bug 总结/HAS ASYE 宏开启

（七） 系统调用

接下来，我们要实现系统调用。首先我们在 nanos-lite/src/irq.c 文件中的 do_event() 函数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;
}
```

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（_syscall_() 函数以及将系统调用的参数依次放入%eax, %ebx, %ecx, %edx 四个寄存器中）。

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

在 nanos-lite/src/syscall.c 的 SYS_none 函数中添加系统调用，设置系统调用的返回值。

```
_RegSet* do_syscall(_RegSet *r) {
   uintptr_t a[4];
   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:
       SYSCALL_ARG1(r) = sys_none();
       break;
     default: panic("Unhandled syscall ID = %d", a[0]);
   }

   return NULL;
 }
```

然后，我们还需要在 nemu/src/cpu/exec/system.c 中实现 iret 指令对现场进行恢复，其主要作用是从异常处理代码中返回，将栈顶的三个元素来依次解释成 EIP、CS、EFLAGS，并恢复。用户进程可以通过%eax 寄存器获得系统调用的返回值，进而得知系统调用执行的结果。

```
make_EHelper(iret) {
  // TODO();
  rtl_pop(&cpu.eip);
  rtl_pop(&cpu.cs);
```

```
  rtl_pop(&t0);
  memcpy(&cpu.eflags, &t0, sizeof(cpu.eflags));
  decoding.jmp_eip = 1;
  decoding.seq_eip = cpu.eip;
  print_asm("iret");
}
```

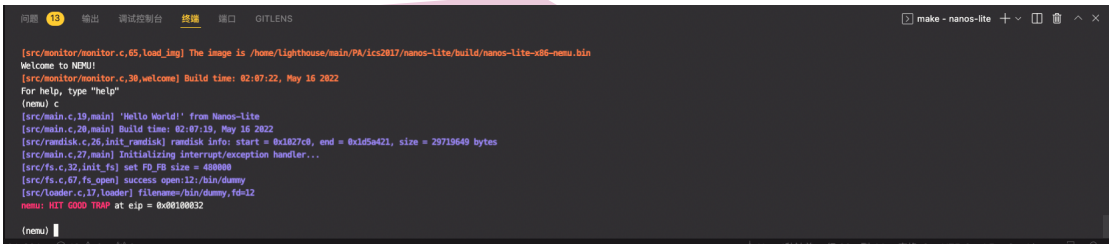
在这之后，我们在 nanos-lite/src/syscall.c 中实现 SYS_exit 系统调用，其目的是接收一个退出状态的参数，用这个参数调用 _halt() 即可。

```
_RegSet* do_syscall(_RegSet *r) {
   uintptr_t a[4];
   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:
       SYSCALL_ARG1(r) = sys_none();
       break;
     case SYS_exit:
       sys_exit(a[1]);
       break;
     default: panic("Unhandled syscall ID = %d", a[0]);
   }

   return NULL;
 }
```

（八） 实验结果



三、 bug 总结

这一部分，基本都是因为我们自己不听劝告，执意要使用 64 位系统，折腾 gcc 以及各种环境问题，导致了各种各样意想不到的问题。

（一） 实现 loader 后出现段错误

在我们最开始实现 loader 时，运行出现了段错误的问题，第一反应是肯定是代码书写有问题，心想坏了，这刚刚开始指定是以前代码的问题，顾花了一下午找之前代码中的问题，没找到，第一次破防。

然后想起来，可以查看反汇编代码，看出了端倪，生成了一堆没啥用的不知道是干啥的东西，导致程序过大越界了。意识到是因该是编译器的问题，上 stackoverflow 查找，在 nano 下的 makefile 中加入如下参数，问题解决。

```
1 OBJCOPY_FLAG = -S --remove-section .note.gnu.property --set-section-flags .bss=alloc,
    contents -O binary
```

(二) HAS ASYE 宏开启

在我们实现完 popa 和 pusha 并正确组织了 trap frame 之后，我们进行了运行测试，嘿嘿，出现了段错误。

可以看到是因为 gate_addr 的值大于了 cpu.idtr.base + cpu.idtr.limit 的值。我们利用 Log 进行输出，得到的结果比较出乎意料，cpu.idtr.base 和 cpu.idtr.limit 的值都是 0。总以为是哪里写错了，又从头到尾撸了一遍代码，没啥问题，最终发现是 nanos-lite/src/main.c 中的 HAS ASY 宏没打开，这意味着根本没有调用 init_irq() 函数，也意味着没有调用位于 nexus-am/am/arch/x86-nemu/src/asye.c 中的 _asye_init() 函数，导致 IDT 根本都没初始化，加上了这个宏之后，错误解决。