南开大学

Nankai University

南　　开　　大　　学

计　算　机　学　院

计算机系统实验报告

PA3

年级：2022 级

专业：计算机科学与技术

姓名：姚知言

学号：2211290

指导教师：卢冶

2025 年 4 月 17 日

目录

一、 实验目的

1. 熟悉操作系统的基本概念，实现系统调用及中断机制。
2. 了解并实现简易的文件系统。
3. 实现 IOE 抽象文件操作，以运行《仙剑奇侠传》等简单游戏。

二、 实验重要内容

PA3 的实验共分为 3 个阶段，包括以下要点：

- 第 1 阶段：了解操作系统相关概念。
- 第 1 阶段：理解 Nanos-lite 的文件架构。

```
nanos-lite
├── include
│   ├── common.h
│   ├── debug.h
│   ├── fs.h
│   ├── memory.h
│   └── proc.h
├── Makefile
└── src
    ├── device.c   # 设备抽象
    ├── fs.c       # 文件系统
    ├── initrd.S   # ramdisk设备
    ├── irq.c      # 中断异常处理
    ├── loader.c   # 加载器
    ├── main.c
    ├── mm.c       # 存储管理
    ├── proc.c     # 进程调度
    ├── ramdisk.c  # ramdisk驱动程序
    └── syscall.c  # 系统调用处理
```

图 1: Nanos-lite 文件架构

- 第 1 阶段：了解并实现中断异常机制，实现相关指令。
- 第 1，2 阶段：实现系统调用 syscall。
- 第 2 阶段：实现简易文件系统，构建文件系统调用接口。

```
--- nanos-lite/src/fs.c
+++ nanos-lite/src/fs.c
@@ -3,5 +3,6 @@
 typedef struct {
   char *name;          // 文件名
   size_t size;         // 文件大小
   off_t disk_offset;   // 文件在ramdisk中的偏移
+  off_t open_offset;   // 文件被打开之后的读写指针
 } Finfo;
```

图 2: 文件记录表

- 第 3 阶段：将 VGA 显存和设备抽象成文件。
- 第 3 阶段：运行《仙剑奇侠传》。

三、 阶段 1 实验方法与结果

(一) 编码内容及测试结果

1. 实现 loader

在实现 loader 之前，首先首先让 Navy-apps 项目上的程序默认编译到 x86 中，在 navy-apps/Makefile.check 中修改。

```
ISA ?= x86
```

然后在 navy-apps/tests/dummy 下执行 make，生成 dummy 的可执行文件。

在 nanos-lite/ 目录下执行 make update，生成 ramdisk 镜像文件 ramdisk.img, 并包含进 Nanos-lite 成为其中的一部分。

在 loader 中, 利用 ramdisk 相关函数, 把用户程序整个搬到 DEFAULT_ENTRY(0x4000000) 处。

```
// 从ramdisk中`offset`偏移处的`len`字节读入到`buf`中
void ramdisk_read(void *buf, off_t offset, size_t len);

// 把`buf`中的`len`字节写入到ramdisk中`offset`偏移处
void ramdisk_write(const void *buf, off_t offset, size_t len);

// 返回ramdisk的大小, 单位为字节
size_t get_ramdisk_size();
```

图 3: ramdisk 相关函数

具体实现如下:

```
void ramdisk_read(void *buf, off_t offset, size_t len);
size_t get_ramdisk_size();

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

需要在文件中添加函数的声明，否则会找不到函数。

然后再 nanos-lite 根目录下 make update，后 make run。可以发现 i386 捕捉到了一条无法识别的指令 (int)，说明 loader 实现已经没有问题，已经成功跳转到 dummy 中执行。

```
src/main.c,19,main] 'Hello World!' from Nanos-lite
[src/main.c,20,main] Build time: 16:44:42, Apr  7 2025
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x100ce8, end = 0x1052c4, 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
 _ ____   ___    __    __ __                          _
(_)___ \ / _ \  / /   |  \/  |                        | |
  __) | (_) |/ /_   | \  / | __ _ _ __   ___  ___ ___| |
 |__ < > _ <| '_ \  | |\/| |/ _` | '_ \ / _ \/ __/ __| |
 |___) | (_) | (_) | | |  | | (_| | | | | (_) \__ \__ \ |
 |____/ \___/ \___/  |_|  |_|\__,_|_| |_|\___/|___/___/_|

for more details.

If it is the second case, remember:
* The machine is always right!
* Every line of untested code is always wrong!
```

图 4: loader 实现验证

2. 实现中断机制

为实现中断机制及 lidt 指令，首先需要实现 idtr 寄存器，以存储中断描述符表的首地址及长度。

在 CPU_state 结构体中添加:

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

手册中 LIDT 指令的说明如图5所示。

```
Operation

IF instruction = LIDT
THEN
    IF OperandSize = 16
    THEN IDTR.Limit:Base ← m16:24 (* 24 bits of base loaded *)
    ELSE IDTR.Limit:Base ← m16:32
    FI;
ELSE (* instruction = LGDT *)
    IF OperandSize = 16
    THEN GDTR.Limit:Base ← m16:24 (* 24 bits of base loaded *)
    ELSE GDTR.Limit:Base ← m16:32;
    FI;
FI;
```

图 5: LIDT

据此，实现 LIDT 指令，并实现对应 opcode_table 的填写。

```
make_group(gp7,
    EMPTY, EMPTY, EMPTY, EX(lidt),
    EMPTY, EMPTY, EMPTY, EMPTY)

make_EHelper(lidt);

make_EHelper(lidt) {
```

```
  cpu.idtr.limit = vaddr_read(id_dest->addr,2);
  cpu.idtr.base = vaddr_read(id_dest->addr+2,(decoding.is_operand_size_16 ? 3
      : 4));
  print_asm_template1(lidt);
}
```

并且，开启对应的宏定义：

```
#define HAS_ASYE
```

然后，实现 int 指令以触发中断。

首先实现 raise_intr 函数，然后再 int 指令中调用。因为后续还有其他函数需要调用此函数，所以不能直接在 int 指令中进行设计。

中断处理流程如图6所示。

- 依次将 EFLAGS, CS, EIP 寄存器的值压入堆栈
- 从 IDTR 中读出 IDT 的首地址
- 根据异常(中断)号在 IDT 中进行索引，找到一个门描述符
- 将门描述符中的 offset 域组合成目标地址
- 跳转到目标地址

图 6: 中断处理流程

据此，实现如下：

```
void raise_intr(uint8_t NO, vaddr_t ret_addr) {
  rtl_push(&cpu.eflags);
  rtl_push(&cpu.cs);
  rtl_push(&ret_addr);

  decoding.jmp_eip = (vaddr_read(cpu.idtr.base + 8 * NO, 4) & 0xFFFF)|(
      vaddr_read(cpu.idtr.base + 4 + 8 * NO, 4) & 0xFFFF0000);
  decoding.is_jmp = 1;
}
```

然后，实现 int 指令。

```
make_EHelper(int);

 /* 0xcc */      EMPTY, IDEXW(I,int,1), EMPTY, EMPTY,

make_EHelper(int) {
  //TODO();
  raise_intr(id_dest->val, decoding.seq_eip);
  print_asm("int %s", id_dest->str);

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

虽然在 NEMU 中并未使用，但为配合 differential testing，需要在 CPU_state 添加 cs 寄存器，并初始化为 0x00000008。

```
//reg.h
uint32_t cs;

//monitor.c:restart
cpu.cs = 0x00000008;
```

测试结果如图7所示，显示未实现的指令经查表为 pusha，对应 trap.S 中的汇编指令，实现成功！

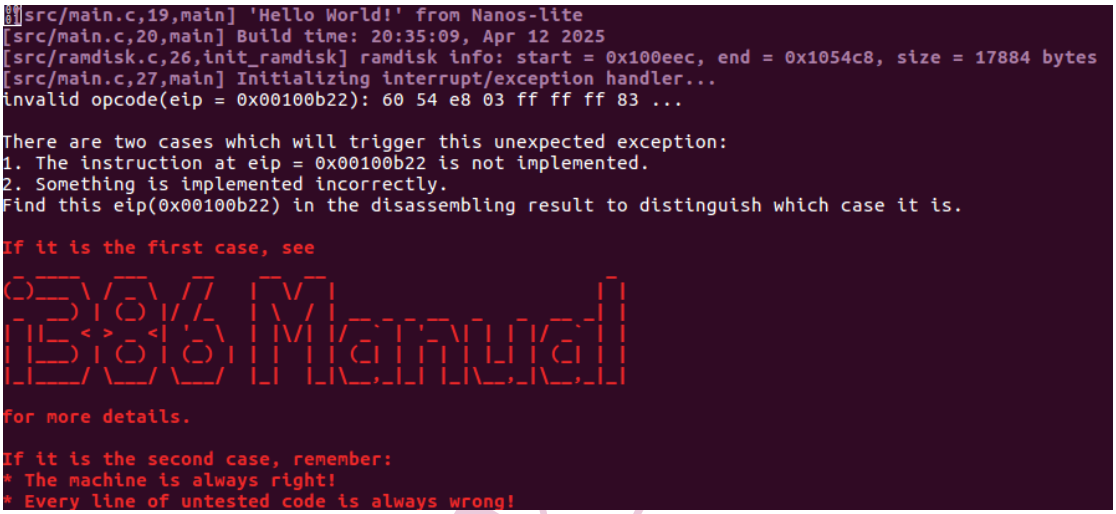


图 7: 中断机制测试结果

3. 重新组织 Trapframe 结构体

根据图8中，PUSHA 中寄存器的保存顺序保存寄存器，实现 pusha 指令。

```
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;
```

图 8: PUSHA

```
make_EHelper(pusha);
```

```
/* 0x60 */      EX(pusha), EMPTY, EMPTY, EMPTY,

make_EHelper(pusha) {
  t1=cpu.esp;
  rtl_push(&cpu.eax);
  rtl_push(&cpu.ecx);
  rtl_push(&cpu.edx);
  rtl_push(&cpu.ebx);
  rtl_push(&t1);
  rtl_push(&cpu.ebp);
  rtl_push(&cpu.esi);
  rtl_push(&cpu.edi);
  print_asm("pusha");
}
```

然后重新组织 _Regset 结构体，nemu 中栈从高地址向低地址，因此先入栈的后声明，后入栈的先声明。首先是一些硬件层的保存，包括 eflags，cs，eip 等。然后转向 pusha 中，根据刚刚实现的 pusha 函数实现排列。

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

测试结果如图9所示，HIT BAD TRAP，Unhandled event ID = 8，测试成功。

```
[src/main.c,19,main] 'Hello World!' from Nanos-lite
[src/main.c,20,main] Build time: 22:59:03, Apr 12 2025
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x100eec, end = 0x1054c8, size = 17884 bytes
[src/main.c,27,main] Initializing interrupt/exception handler...
[src/irq.c,5,do_event] system panic: Unhandled event ID = 8
nemu: HIT BAD TRAP at eip = 0x00100032
```

图 9: Trapframe 测试结果

4. 实现系统调用

首先在 do_event() 中识别出系统调用事件 _EVENT_SYSCALL，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;
}
```

然后实现 SYSCALL_ARGx() 宏，在 i386 中，系统调用的参数会依次保存在 EAX，EBX，ECX，EDX，ESI，EDI.... 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 do_syscall 中实现 SYS_none。什么都不做，直接返回 1。

```
switch (a[0]) {
    case SYS_none:
      SYSCALL_ARG1(r) = 1;
      break;
    default: panic("Unhandled syscall ID = %d", a[0]);
}
```

然后实现 POPA 和 IRET 指令。

POPA 在 i386 手册中的说明如图10所示，和 PUSHA 相反的顺序即可，注意有原先的 ESP 要丢弃。

```
IF OperandSize = 16 (* instruction = POPA *)
THEN
   DI ← Pop();
   SI ← Pop();
   BP ← Pop();
   throwaway ← Pop (); (* Skip SP *)
   BX ← Pop();
   DX ← Pop();
   CX ← Pop();
   AX ← Pop();
ELSE (* OperandSize = 32, instruction = POPAD *)
   EDI ← Pop();
   ESI ← Pop();
   EBP ← Pop();
   throwaway ← Pop (); (* Skip ESP *)
   EBX ← Pop();
   EDX ← Pop();
   ECX ← Pop();
   EAX ← Pop();
FI;
```

图 10: POPA

```
make_EHelper(popa);

  /* 0x60 */     EX(pusha), EX(popa), EMPTY, EMPTY,

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

IRET 的实现如图11所示，会把前三个参数解释为 eip，cs 和 eflags。同时需要标记指令需要跳转。

```
IF PE = 0
THEN (* Real-address mode *)
   IF OperandSize = 32 (* Instruction = IRETD *)
   THEN EIP ← Pop();
   ELSE (* Instruction = IRET *)
      IP ← Pop();
   FI;
   CS ← Pop();
   IF OperandSize = 32 (* Instruction = IRETD *)
   THEN EFLAGS ← Pop();
   ELSE (* Instruction = IRET *)
      FLAGS ← Pop();
   FI;
```

图 11: IRET

```
make_EHelper(iret);

  /* 0xcc */     EMPTY, IDEXW(I,int,1), EMPTY, EX(iret),

make_EHelper(iret) {
  rtl_pop(&decoding.jmp_eip);
  rtl_pop(&cpu.cs);
  rtl_pop(&cpu.eflags);
  decoding.is_jmp = 1;
  print_asm("iret");
}
```

执行程序，发现此处缺失一个编号为 4 的调用，说明之前的实现已经没有问题。

```
[src/monitor/diff-test/diff-test.c,96,init_difftest] Connect to QEMU successfully
[src/monitor/monitor.c,65,load_img] The image is /home/latesoon/ics2017/nanos-lite/build/nanos-lite-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 04:20:44, Apr 13 2025
For help, type "help"
(nemu) c
[src/main.c,19,main] 'Hello World!' from Nanos-lite
[src/main.c,20,main] Build time: 03:41:11, Apr 13 2025
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x100fb0, end = 0x10558c, size = 17884 bytes
[src/main.c,27,main] Initializing interrupt/exception handler...
[src/syscall.c,12,do_syscall] system panic: Unhandled syscall ID = 4
nemu: HIT BAD TRAP at eip = 0x00100032
```

图 12: 验证结果

然后完成这个调用,即完成 sys_exit,它会接收一个退出状态的参数, 用这个参数调用 _halt() 即可。在 do_syscall 中添加。

```
case SYS_exit:
    _halt(SYSCALL_ARG2(r));
    break;
```

再次测试，HIT GOOD TRAP，成功。

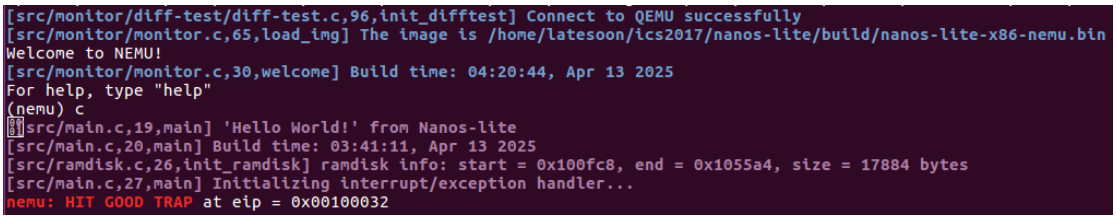


图 13: 第一阶段测试结果

(二) 问答题

1. 对比异常与函数调用

- Q:

  我们知道进行函数调用的时候也需要保存调用者的状态：返回地址，以及调用约定 (calling convention) 中需要调用者保存的寄存器，而进行异常处理之前却要保存更多的信息。尝试对比它们，并思考两者保存信息不同是什么原因造成的。

- A:

  函数调用的保存内容包括：返回地址，调用者保存的寄存器，参数和局部变量等。

  异常处理的保存内容包括：完整上下文，异常原因和状态，还需要保存一些硬件保存和内核态信息等。

  这样的原因是函数调用是一个较为确定的过程，而异常处理是异步的，涉及到特权态的转换，可能发生在任何指令执行时，且异常处理程序无法预知被中断代码的状态，因此必须保存全部寄存器。

2. 诡异的代码

- Q:

  trap.S 中有一行 pushl %esp 的代码，乍看之下其行为十分诡异。你能结合前后的代码理解它的行为吗？Hint: 不用想太多, 其实都是你学过的知识。

- A:

  将当前栈指针的值（即%esp 寄存器的内容）压入栈中。

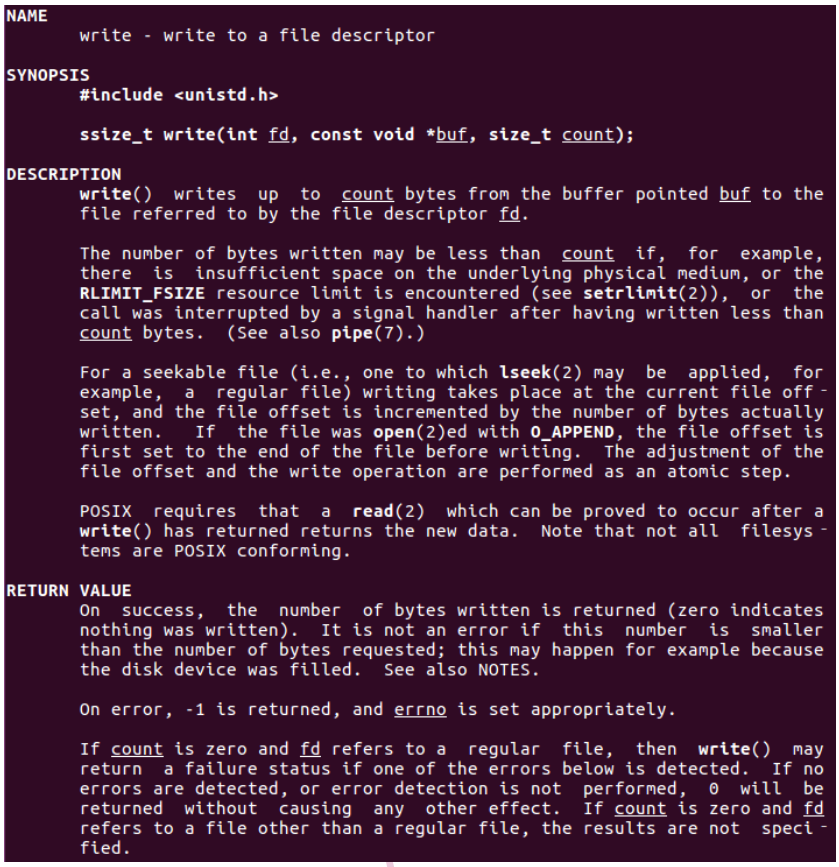
  pushl %esp 压入的是当前内核栈的栈顶地址，此时栈顶恰好指向 pushal 保存的寄存器结构（trapframe）的起始位置。以便传递给后续 irq_handle 函数。

四、 阶段 2 实验方法与结果

(一) 编码内容及测试结果

1. 在 Nanos-lite 上运行 Hello world

查阅 man 2 write 参数和返回值信息如图14。包括三个参数 fd，buffer，count。如果 fd 是 1 或者 2(分别代表 stdout 和 stderr)，则打印 buffer 指向位置的至多 count 个字符。正确则返回打印的字符数量，错误则返回-1。



```
NAME
       write - write to a file descriptor

SYNOPSIS
       #include <unistd.h>

       ssize_t write(int fd, const void *buf, size_t count);

DESCRIPTION
       write()  writes  up  to  count bytes from the buffer pointed buf to the
       file referred to by the file descriptor fd.

       The number of bytes written may be less than  count  if,  for  example,
       there  is  insufficient space on the underlying physical medium, or the
       RLIMIT_FSIZE resource limit is encountered (see setrlimit(2)),  or  the
       call was interrupted by a signal handler after having written less than
       count bytes.  (See also pipe(7).)

       For a seekable file (i.e., one to which lseek(2) may  be  applied,  for
       example,  a  regular file) writing takes place at the current file off-
       set, and the file offset is incremented by the number of bytes actually
       written.   If  the file was open(2)ed with O_APPEND, the file offset is
       first set to the end of the file before writing.  The adjustment of the
       file offset and the write operation are performed as an atomic step.

       POSIX  requires  that  a  read(2)  which can be proved to occur after a
       write() has returned returns the new data.  Note that not all  filesys-
       tems are POSIX conforming.

RETURN VALUE
       On  success,  the  number  of bytes written is returned (zero indicates
       nothing was written).  It is not an error if  this  number  is  smaller
       than the number of bytes requested; this may happen for example because
       the disk device was filled.  See also NOTES.

       On error, -1 is returned, and errno is set appropriately.

       If count is zero and fd refers to a  regular  file,  then  write()  may
       return  a failure status if one of the errors below is detected.  If no
       errors are detected, or error detection is not  performed,  0  will  be
       returned  without  causing  any  other effect.  If count is zero and fd
       refers to a file other than a regular file, the results are not  speci-
       fied.
```

图 14: man 2 write

据此，实现 SYS_write 调用。

```
1  int do_syswrite(int fd,const void* buf,size_t len){
2    if(fd == 1 || fd == 2){
3      for(int i=0;i<len;i++)
4        _putc(((char*)(buf))[i]);
5    }
6    return len;
7  }
8
9  _RegSet* do_syscall(_RegSet *r) {
10   //...
11   a[1] = SYSCALL_ARG2(r);
12   a[2] = SYSCALL_ARG3(r);
13   a[3] = SYSCALL_ARG4(r);
14
15   switch (a[0]) {
16     //...
17     case SYS_write:
18       SYSCALL_ARG1(r) = do_syswrite(((int)(a[1])), ((char*)(a[2])),((size_t)(
             a[3])));
19       break;
20     default: panic("Unhandled syscall ID = %d", a[0]);
```

```
  }
  return NULL;
}
```

在 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){
  _syscall_(SYS_write, fd, buf, count);
}
```

在 hello 目录下运行 make，然后修改 nanos-lite/Makefile 以运行 hello 文件。

```
OBJCOPY_FILE = $(NAVY_HOME)/tests/hello/build/hello-x86
```

运行结果如图15所示，正确打印。

```
[src/main.c,19,main] 'Hello World!' from Nanos-lite
[src/main.c,20,main] Build time: 03:41:11, Apr 13 2025
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x101074, end = 0x105750, 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
Hello World for the 4th time
Hello World for the 5th time
Hello World for the 6th time
Hello World for the 7th time
```

图 15: hello 测试结果

2. 实现堆区管理

开始实验之前，先在 SYS_write 分支中增加一行 Log，并重新运行程序，用于后续对比。

```
case SYS_write:
  Log("SYS_write!");
  SYSCALL_ARG1(r) = do_syswrite(((int)(a[1])), ((char*)(a[2])),((size_t)(a
      [3])));
  break;
```

可以看到，由于没有实现堆区管理，只能每输出一次字符就调用一次 SYS_write。

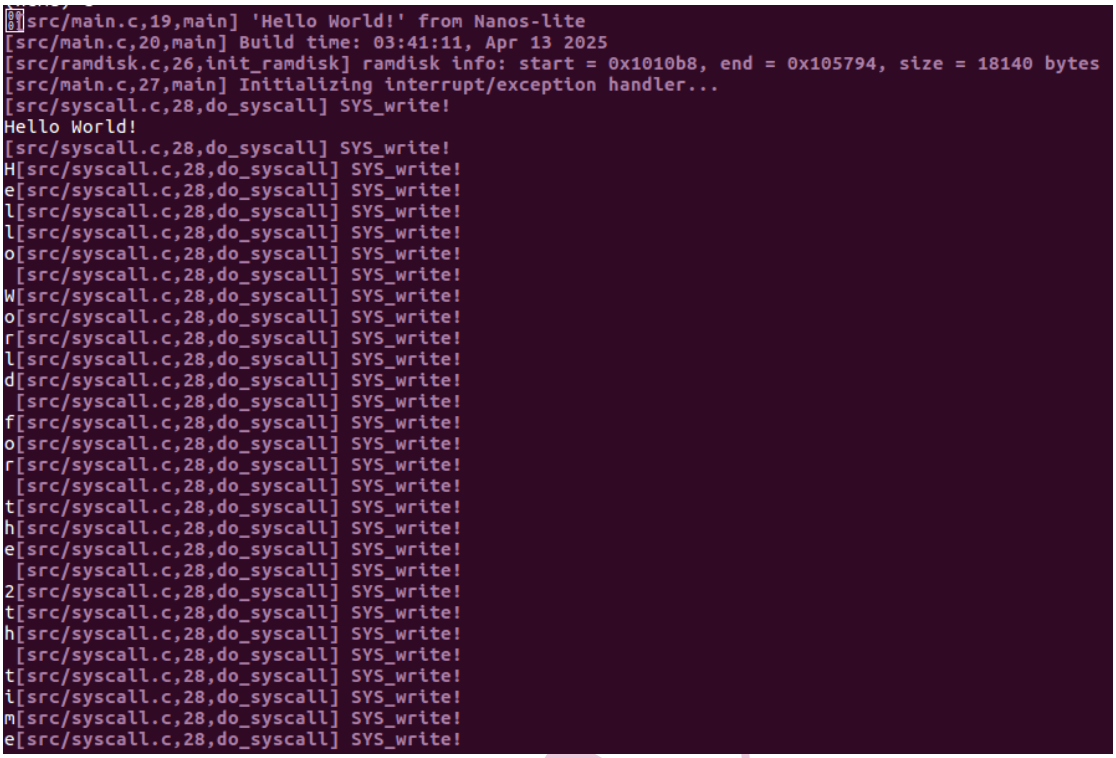


图 16: 实现堆区管理之前

查阅 man 2 sbrk 信息，可以知道两个函数用于调整断点（即未初始化数据段的末尾）：

- brk(void *addr)：直接设置断点为 addr。
- sbrk(intptr_t increment)：按增量 increment 调整断点。

图 17: man 2 sbrk

查询 man 3 end 信息，可以知道 end 表示的是程序断点（program break）是未初始化数据段（BSS 段）末尾之后的第一个地址。指示进程数据段的当前结束位置，决定了堆（heap）的起始边界。

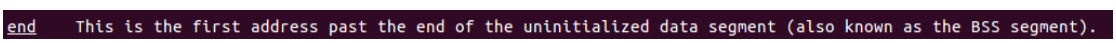


图 18: man 3 end

实现的总体思路为:

- program break 一开始的位置位于 _end
- 被调用时, 根据记录的 program break 位置和参数 increment, 计算出新 program break
- 通过 SYS_brk 系统调用来让操作系统设置新 program break
- 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
- 若该系统调用失败,_sbrk() 会返回-1

暂时让 SYS_brk 一直返回 0 就可以。

```
case SYS_brk:
  SYSCALL_ARG1(r) = 0;
  break;
```

然后实现 _sbrk 函数, 首先需要保存 _end 变量的初始值, 记录在 pb 中。然后根据 increment 更新 pb，如果 syscall 执行成功则更新，并返回原来的 pb，否则返回-1。

```
extern char _end;
static intptr_t pb = (intptr_t)& _end;

void *_sbrk(intptr_t increment){
  intptr_t update_pb = pb + increment;
  if(!_syscall_(SYS_brk,update_pb,0,0)){
    pb = update_pb;
    return (void *)(update_pb - increment);
  }
  else
    return (void *)-1;
}
```

堆区管理测试结果如图19所示，在 Nanos-lite 中看到，printf() 将格式化完毕的字符串通过一次 write 系统调用进行输出，说明堆区管理的实现已经完成。

```
(nemu) c
[src/main.c,19,main] 'Hello World!' from Nanos-lite
[src/main.c,20,main] Build time: 03:41:11, Apr 13 2025
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x1010d0, end = 0x1057f0, size = 18208 bytes
[src/main.c,27,main] Initializing interrupt/exception handler...
[src/syscall.c,28,do_syscall] SYS_write!
Hello World!
[src/syscall.c,28,do_syscall] SYS_write!
Hello World for the 2th time
[src/syscall.c,28,do_syscall] SYS_write!
Hello World for the 3th time
[src/syscall.c,28,do_syscall] SYS_write!
Hello World for the 4th time
[src/syscall.c,28,do_syscall] SYS_write!
Hello World for the 5th time
[src/syscall.c,28,do_syscall] SYS_write!
Hello World for the 6th time
[src/syscall.c,28,do_syscall] SYS_write!
Hello World for the 7th time
```

图 19: 堆区管理测试结果

3. 让 loader 使用文件

首先对 nanos-lite/Makefile 进行修改。

```
update: update-ramdisk-fsimg src/syscall.h
```

修改后运行 make update 就会自动编译 Navy-apps 里面的所有程序, 并把 navy-apps/fsimg/目录下的所有内容整合成 ramdisk 镜像。

然后，在文件记录结构体 Finfo 中增加一个属性 open_offset 指示文件当前打开的位置指针。

```
typedef struct {
  char *name;
  size_t size;
  off_t disk_offset;
  off_t open_offset;
} Finfo;
```

为让 loader 使用文件，首先需要实现 fs_open，fs_read，fs_close 函数。

如图20所示，OPEN 函数的作用是根据文件路径返回一个文件描述符 (小的非负整数)，用于后续的系统调用 (如 read, write 等)。

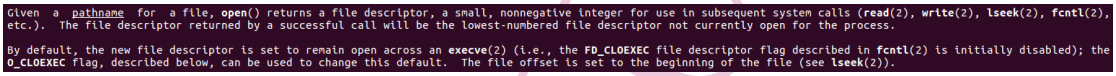


图 20: OPEN

由于简易文件系统中不会产生新文件，”fs_open() 没有找到 pathname 所指示的文件” 属于异常情况，使用 assertion 终止程序运行。

为了简化实现，允许所有用户程序都可以对所有已存在的文件进行读写，这样以后，我们在实现 fs_open() 的时候就可以忽略 flags 和 mode 了。

```
int fs_open(const char* pathname, int flags, int mode){
  for(int fd = 0; fd < NR_FILES; fd++){
    if(!strcmp(pathname,file_table[fd].name))
      return fd;
  }
  assert(0);//should not reach here
  return -1;
}
```

然后是 fs_close 函数。由于简易文件系统没有维护文件打开的状态,fs_close() 可以直接返回 0，表示总是关闭成功。

```
int fs_close(int fd){
  return 0;
}
```

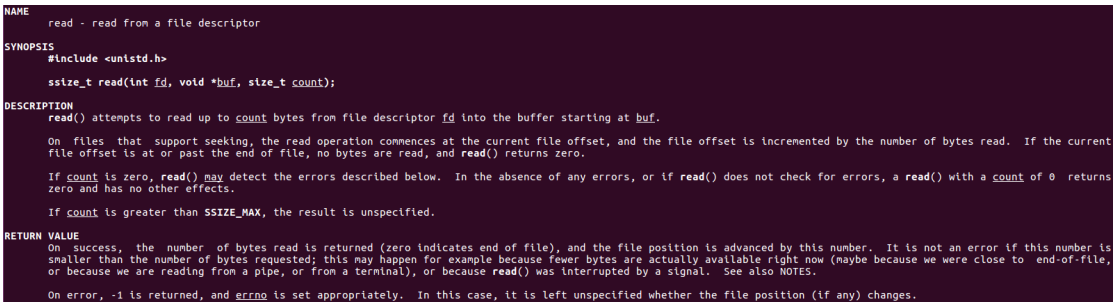
在实现后续函数之前，我先实现几个辅助函数，在后续函数中调用以简化流程。由于 c 语言不能够实现函数参数的默认值以及函数重载操作，update_offset 和 update_offset3 进行了分别实现，update_offset 直接调用 update_offset3。

函数功能分别为:

1. fs_filesz：返回文件大小。
2. tot_offset：返回文件当前位置的总偏移量（文件偏移量 + 打开位置偏移量）。
3. fs_offset：返回文件打开位置的偏移量。
4. update_offset3：能够接受模式的更新文件当前打开位置偏移量，用于后续 lseek 扩展，此外也被 update_offset 直接调用。
5. update_offset：根据参数增减文件当前打开位置偏移量，用于 read，及后续要实现的 write 完成后更新。

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

static inline off_t tot_offset(int fd){
  return file_table[fd].disk_offset + file_table[fd].open_offset;
}

static inline off_t fs_offset(int fd){
  return file_table[fd].open_offset;
}

static inline off_t update_offset3(int fd, int len, int mode){
  if(mode == SEEK_SET)
    file_table[fd].open_offset = 0;
  else if(mode == SEEK_END)
    file_table[fd].open_offset = file_table[fd].size;
  file_table[fd].open_offset += len;
  file_table[fd].open_offset = ((file_table[fd].open_offset > file_table[fd].
      size) ? file_table[fd].size : file_table[fd].open_offset);
  file_table[fd].open_offset = ((file_table[fd].open_offset < 0) ? 0 :
      file_table[fd].open_offset);
  return file_table[fd].open_offset;
}

static inline void update_offset(int fd, int len){
  update_offset3(fd, len, SEEK_CUR);
}
```

如图21所示，READ 函数的作用是从文件当前偏移量处，开始取最多 len 个字符，存到 buf 中。成功时返回实际读取的字节数 (可能小于 len)，返回 0 表示已到达文件末尾 (EOF)，出错时返回-1。



```
NAME
       read - read from a file descriptor

SYNOPSIS
       #include <unistd.h>

       ssize_t read(int fd, void *buf, size_t count);

DESCRIPTION
       read() attempts to read up to count bytes from file descriptor fd into the buffer starting at buf.

       On  files  that  support seeking, the read operation commences at the current file offset, and the file offset is incremented by the number of bytes read.  If the current
       file offset is at or past the end of file, no bytes are read, and read() returns zero.

       If count is zero, read() may detect the errors described below.  In the absence of any errors, or if read() does not check for errors, a read() with a count of 0  returns
       zero and has no other effects.

       If count is greater than SSIZE_MAX, the result is unspecified.

RETURN VALUE
       On  success,  the  number  of bytes read is returned (zero indicates end of file), and the file position is advanced by this number.  It is not an error if this number is
       smaller than the number of bytes requested; this may happen for example because fewer bytes are actually available right now (maybe because we were close to  end-of-file,
       or because we are reading from a pipe, or from a terminal), or because read() was interrupted by a signal.  See also NOTES.

       On error, -1 is returned, and errno is set appropriately.  In this case, it is left unspecified whether the file position (if any) changes.
```

图 21: READ

read 函数实现如下，要声明其中使用的 ramdisk_read。

```c
void ramdisk_read(void *buf, off_t offset, size_t len);

ssize_t fs_read(int fd, void* buf, size_t len){
  switch(fd){
    case FD_STDIN:
    case FD_STDOUT:
    case FD_STDERR:
      break;
    default:
      len = (fs_filesz(fd) - fs_offset(fd) >= len) ? len : (fs_filesz(fd) -
          fs_offset(fd));
      if(len <= 0)
        return 0;
      ramdisk_read(buf, tot_offset(fd), len);
      update_offset(fd, len);
  }
  return len;
}
```

将实现的函数声明添加在 fs.h 中。

```c
//fs.h
int fs_open(const char* pathname, int flags, int mode);
ssize_t fs_read(int fd, void* buf, size_t len);
int fs_close(int fd);
size_t fs_filesz(int fd);
```

在 syscall.c 中引入 fs.h，并实现新 case。在 nanos.c 中对新实现函数进行修改。

```c
//syscall.c
#include "fs.h"

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

```
case SYS_close:
  SYSCALL_ARG1(r) = fs_close((int)(a[1]));
  break;

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

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

在 loader.c 中引入 fs.h，更新 loader，使其不再直接调用 ramdisk_read，而是通过文件实现更加方便的加载。实现之后，以后更换用户程序只需要修改传入 loader() 函数的文件名即可。

```
//loader.c
#include "fs.h"
uintptr_t loader(_Protect *as, const char *filename) {
  //PA3.2
  int fd = fs_open(filename, 0, 0);
  fs_read(fd, DEFAULT_ENTRY, fs_filesz(fd));
  fs_close(fd);
  return (uintptr_t)DEFAULT_ENTRY;
}
```

通过在 nanos-lite/src/main.c 修改，就可以调整加载的文件。

```
uint32_t entry = loader(NULL, "/bin/hello");
```

重新测试结果如图22，通过，已经成功让 loader 使用文件。（我的猜测是，由于这一步测试的是之前测试过的文件，应该不会用到还未实现的 syscall，所以这一步不实现 syscall 和 nanos 的修改可能也可以通过。但是无论如何实现完整文件系统都要实现。）

```
(nemu) c
[src/main.c,19,main] 'Hello World!' from Nanos-lite
[src/main.c,20,main] Build time: 02:09:40, Apr 16 2025
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x101900, end = 0x370306, size = 2550278 bytes
[src/main.c,27,main] Initializing interrupt/exception handler...
Hello World!
Hello World for the 2th time
Hello World for the 3th time
Hello World for the 4th time
Hello World for the 5th time
Hello World for the 6th time
```

图 22: loader 使用文件

4. 实现完整的文件系统

虽然在 i386 中，LSEEK 允许偏移超过文件边界形成文件空洞，但是由于在 NEMU 中文件的大小是固定的，所以不考虑超过边界的情况。

```
NAME
       lseek - reposition read/write file offset

SYNOPSIS
       #include <sys/types.h>
       #include <unistd.h>

       off_t lseek(int fd, off_t offset, int whence);

DESCRIPTION
       The lseek() function repositions the offset of the open file associated with the file descriptor fd to the argument offset according to the directive whence as follows:

       SEEK_SET
              The offset is set to offset bytes.

       SEEK_CUR
              The offset is set to its current location plus offset bytes.

       SEEK_END
              The offset is set to the size of the file plus offset bytes.

       The lseek() function allows the file offset to be set beyond the end of the file (but this does not change the size of the file).  If data is later written at this point,
       subsequent reads of the data in the gap (a "hole") return null bytes ('\0') until data is actually written into the gap.
```

图 23: LSEEK

基本上把功能都封装在之前的辅助函数中了。但因为辅助函数实现较简单，所以增加一条 assert 以保证 whence 是我们期望的值。

```
off_t fs_lseek(int fd, off_t offset, int whence){
  assert(whence == SEEK_CUR || whence == SEEK_END || whence == SEEK_SET);
  return update_offset3(fd, offset, whence);
}
```

如图24所示，write 的主体思想与 read 类似。

```
NAME
       write - write to a file descriptor

SYNOPSIS
       #include <unistd.h>

       ssize_t write(int fd, const void *buf, size_t count);

DESCRIPTION
       write() writes up to count bytes from the buffer pointed buf to the file referred to by the file descriptor fd.

       The  number  of bytes written may be less than count if, for example, there is insufficient space on the underlying physical medium, or the RLIMIT_FSIZE resource limit is
       encountered (see setrlimit(2)), or the call was interrupted by a signal handler after having written less than count bytes.  (See also pipe(7).)

       For a seekable file (i.e., one to which lseek(2) may be applied, for example, a regular file) writing takes place at the current file  offset,  and  the  file  offset  is
       incremented  by  the  number of bytes actually written.  If the file was open(2)ed with O_APPEND, the file offset is first set to the end of the file before writing.  The
       adjustment of the file offset and the write operation are performed as an atomic step.

       POSIX requires that a read(2) which can be proved to occur after a write() has returned returns the new data.  Note that not all filesystems are POSIX conforming.

RETURN VALUE
       On success, the number of bytes written is returned (zero indicates nothing was written).  It is not an error  if  this  number  is  smaller  than  the  number  of  bytes
       requested; this may happen for example because the disk device was filled.  See also NOTES.

       On error, -1 is returned, and errno is set appropriately.

       If  count  is  zero and fd refers to a regular file, then write() may return a failure status if one of the errors below is detected.  If no errors are detected, or error
       detection is not performed, 0 will be returned without causing any other effect.  If count is zero and fd refers to a file other than a regular file, the results are  not
       specified.
```

图 24: WRITE

唯一的区别在于 read 不考虑 stdout 和 stderr 文件，而 write 中将其视为 putc 操作。

```
void ramdisk_write(const void *buf, off_t offset, size_t len);

ssize_t fs_write(int fd, const void* buf, size_t len){
  switch(fd){
    case FD_STDIN:
      break;
    case FD_STDOUT:
    case FD_STDERR:
      for(int i=0;i<len;i++)
      _putc(((char*)(buf))[i]);
      break;
    default:
      len = (fs_filesz(fd) - fs_offset(fd) >= len) ? len : (fs_filesz(fd) -
          fs_offset(fd));
      if(len <= 0)
```

```
        return 0;
      ramdisk_write(buf, tot_offset(fd), len);
      update_offset(fd, len);
  }
  return len;
}
```

补充函数声明以及 syscall 调用逻辑。

```
//fs.h
off_t fs_lseek(int fd, off_t offset, int whence);
ssize_t fs_write(int fd, const void* buf, size_t len);

//nanos.c
off_t _lseek(int fd, off_t offset, int whence) {
  _syscall_(SYS_lseek, fd, offset, whence);
}

//syscall.c
case SYS_write://3.1中已实现, 但此处应该更新为文件系统版本, 其中兼容之前的
    putc
  //PA3.2
  SYSCALL_ARG1(r) = fs_write(((int)(a[1])), ((char*)(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;
```

测试 text，PASS，通过测试。

```
uint32_t entry = loader(NULL, "/bin/text");
```

```
(nemu) c
[src/main.c,19,main] 'Hello World!' from Nanos-lite
[src/main.c,20,main] Build time: 02:24:43, Apr 16 2025
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x101920, end = 0x370326, size = 2550278 bytes
[src/main.c,27,main] Initializing interrupt/exception handler...
PASS!!!
nemu: HIT GOOD TRAP at eip = 0x00100032
```

图 25: 文件系统测试

(二) 问答题：缓冲区与系统调用开销

- Q:

  你已经了解系统调用的过程了. 事实上, 如果通过系统调用千辛万苦地陷入操作系统只是为了输出区区一个字符, 那就太不划算了. 于是有了 batching 的技术: 将一些简单的任务累积起来, 然后再一次性进行处理. 缓冲区是 batching 技术的核心,libc 中的输入输出函数正是通过缓冲区来将输入输出累积起来, 然后再通过一次系统调用进行处理. 例如通过一个 1024 字节的缓冲区, 就可以通过一次系统调用直接输出 1024 个字符, 而不需要通过 1024 次系统调用来逐个字符地输出. 显然, 后者的开销比前者大得多.

在 GNU/Linux 上编写相应的程序, 来粗略测试一下一次 write 系统调用的开销, 然后和这篇文章对比一下.

- A:

编写的测试程序如下，丢弃控制台输出以减少输出到控制台造成的误差，同时使用 stderr 输出测试结果，避免被重定向影响。对比了无缓冲，有自行实现缓冲和使用 stdio 自带缓冲的三种情况。

```c
#include <unistd.h>
#include <time.h>
#include <stdio.h>
#include <string.h>

#define TRIES 100000
#define BUF_SIZE 1024

void test_unbuffered() {
    char c = 'A';
    struct timespec start, end;

    clock_gettime(CLOCK_MONOTONIC, &start);
    for (int i = 0; i < TRIES; i++) {
        write(STDOUT_FILENO, &c, 1);
    }
    clock_gettime(CLOCK_MONOTONIC, &end);

    long time_ns = (end.tv_sec - start.tv_sec) * 1000000000 + (end.tv_nsec - start.tv_nsec);
    fprintf(stderr, "无缓冲 - 每次write()平均耗时: %ld ns\n", time_ns / TRIES);
}

void test_buffered() {
    char buf[BUF_SIZE];
    struct timespec start, end;

    memset(buf, 'A', BUF_SIZE);

    clock_gettime(CLOCK_MONOTONIC, &start);
    for (int i = 0; i < TRIES / BUF_SIZE; i++) {
        write(STDOUT_FILENO, buf, BUF_SIZE);
    }
    clock_gettime(CLOCK_MONOTONIC, &end);

    long time_ns = (end.tv_sec - start.tv_sec) * 1000000000 + (end.tv_nsec - start.tv_nsec);
    fprintf(stderr, "有缓冲(%dB) - 每次write()平均耗时: %ld ns\n",
            BUF_SIZE, time_ns / TRIES);
}
```

```c

void test_stdio_buffered() {
    struct timespec start, end;

    clock_gettime(CLOCK_MONOTONIC, &start);
    for (int i = 0; i < TRIES; i++) {
        putchar('A');
    }
    clock_gettime(CLOCK_MONOTONIC, &end);

    long time_ns = (end.tv_sec - start.tv_sec) * 1000000000 + (end.tv_nsec - start.tv_nsec);
    fprintf(stderr, "stdio带缓冲 - 每次putchar()平均耗时: %ld ns\n", time_ns / TRIES);
}

int main() {
    fprintf(stderr, "系统调用开销测试 (每次测试%d次操作)\n", TRIES);
    fprintf(stderr, "==================================\n");

    test_unbuffered();
    test_buffered();
    test_stdio_buffered();

    return 0;
}
```

测试结果如图26所示，可以看出缓冲区减小系统调用开销还是非常显著的。

```
~/Desktop ❯ gcc -o pa3 pa3.c -lrt
~/Desktop ❯ ./pa3 > /dev/null
系统调用开销测试 (每次测试100000次操作)
==================================
无缓冲 - 每次write()平均耗时: 399 ns
有缓冲(1024B) - 每次write()平均耗时: 0 ns
stdio带缓冲 - 每次putchar()平均耗时: 15 ns
```

图 26: 缓冲区对比

五、 阶段 3 实验方法与结果

(一) 编码内容及测试结果

1. 把 VGA 显存抽象成文件

首先在 init_fs()(在 nanos-lite/src/fs.c 中) 中对文件记录表中/dev/fb 的大小进行初始化。将其初始化为长宽之积乘上每一个点的数据大小（即 uint32_t)。

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

实现 fb_write()(在 nanos-lite/src/device.c 中)，把 buf 中的 len 字节写到屏幕上 offset 处。

根据 offset 计算横纵坐标，注意数据都需要除以 uint32 大小。为确保程序的正确性，要检查 offset 和 len 是否是 uint32 大小，即 4 的倍数。

```
void fb_write(const void *buf, off_t offset, size_t len) {
  assert(!(offset%sizeof(uint32_t) | len%sizeof(uint32_t)));
  _draw_rect((uint32_t *)buf, (offset/sizeof(uint32_t))%_screen.width, (
      offset/sizeof(uint32_t))/_screen.width, len/sizeof(uint32_t),1);
}
```

在 init_device()(在 nanos-lite/src/device.c 中) 中将/proc/dispinfo 的内容提前写入到字符串 dispinfo 中。

通过 sprintf，构建格式化字符串，传入 _screen.width 和 _screen.height，并写入 dispinfo。

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

然后实现 dispinfo_read()(在 nanos-lite/src/device.c 中)，把字符串 dispinfo 中 offset 开始的 len 字节写到 buf 中。

直接调用 strncpy，当前位置是 dispinfo 偏移 offset 位置，读取 len 长度的数据。

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

然后在 fs_read 和 fs_write 中补充 FB 和 DISPINFO 的分支。

```
void dispinfo_read(void *buf, off_t offset, size_t len);

ssize_t fs_read(int fd, void* buf, size_t len){
  switch(fd){
    //...
    case FD_FB:
      break;
    case FD_DISPINFO:
      len = (fs_filesz(fd) - fs_offset(fd) >= len) ? len : (fs_filesz(fd) -
          fs_offset(fd));
      if(len <= 0)
        return 0;
      dispinfo_read(buf, tot_offset(fd), len);
      update_offset(fd, len);
      break;
    //...
  }
  return len;
}

void fb_write(const void *buf, off_t offset, size_t len);

```

```
ssize_t fs_write(int fd, const void* buf, size_t len){
  switch(fd){
    //...
    case FD_DISPINFO:
      break;
    case FD_FB:
      fb_write(buf, fs_offset(fd), len);
      update_offset(fd,len);
      break;
    //...
  }
  return len;
}
```

运行结果如图27所示，成功打印，运行成功。

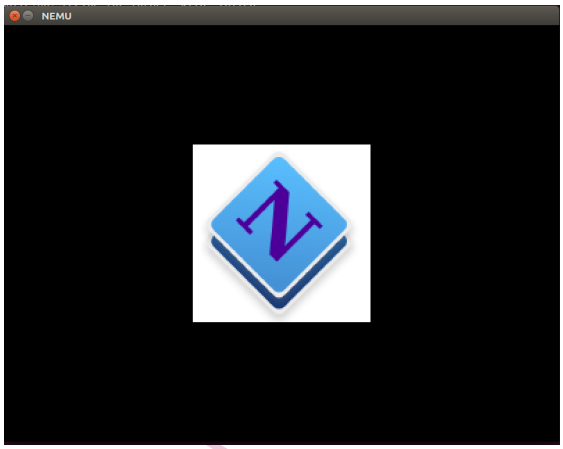


图 27: VGA 显存运行结果

2. 把设备输入抽象成文件

实现 events_read 函数，在没有键盘操作的时候返回程序运行的时间（通过 uptime），在有操作的时候通过 readkey 打印键盘操作。

```
size_t events_read(void *buf, size_t len) {
  int key = _read_key();
  if(key == _KEY_NONE)
    sprintf(buf,"t %d\n",_uptime());
  else
    sprintf(buf,"%s %s\n",(key & 0x8000) ? "kd" : "ku", keyname[key & 255]);
  return strlen(buf);
}
```

然后在 fs.c 中增加对应分支。

```
//增加函数声明
size_t events_read(void *buf, size_t len);

//fs_read
```

```
case FD_EVENTS:
  len = events_read(buf, len);
  break;

//fs_write
case FD_EVENTS:
  break;
```

测试结果如图28所示，我依次键入 ILOVENANKAI，成功显示了输出，正确。

```
receive event: t 12182
receive event: t 12358
receive event: t 12530
receive event: kd I
receive event: ku I
receive event: kd L
receive event: ku L
receive event: kd O
receive event: ku O
receive event: kd V
receive event: ku V
receive event: t 14125
receive event: t 14297
receive event: kd E
receive event: ku E
receive event: t 14822
receive event: t 14995
receive event: t 15170
receive event: kd N
receive event: ku N
receive event: kd A
receive event: ku A
receive event: kd N
receive event: ku N
receive event: kd K
receive event: ku K
receive event: kd A
receive event: ku A
receive event: kd I
receive event: ku I
```

图 28: 时钟/输入测试结果

3. 在 NEMU 中运行仙剑奇侠传

下载课程群里的压缩包,解压缩后将整个 pal 文件夹放在 navy-apps/fsimg/share/games/pal/目录下。重新 make update，make run，成功运行，如图29和30所示。

图 29: 仙剑奇侠传

图 30: 仙剑奇侠传 2

(二) 问答题：不再神秘的秘技

- Q:

  网上流传着一些关于仙剑奇侠传的秘技, 其中的若干条秘技如下:

  很多人到了云姨那里都会去拿三次钱, 其实拿一次就会让钱箱爆满! 你拿了一次钱就去买剑把钱用到只剩一千多, 然后去道士那里, 先不要上楼, 去掌柜那里买酒, 多买几次你就会发现钱用不完了。

不断使用乾坤一掷 (钱必须多于五千文) 用到财产低于五千文, 钱会暴增到上限, 如此一来就有用不完的钱了。

当李逍遥等级到达 99 级时, 用 5 10 只金蚕王, 经验点又跑出来了, 而且升级所需经验会变回初期 5 10 级内的经验值, 然后去打敌人或用金蚕王升级, 可以学到灵儿的法术 (从五气朝元开始); 升到 199 级后再用 5 10 只金蚕王, 经验点再跑出来, 所需升级经验也是很低, 可以学到月如的法术 (从一阳指开始); 到 299 级后再用 10 30 只金蚕王, 经验点出来后继续升级, 可学到阿奴的法术 (从万蚁蚀象开始)。

假设这些上述这些秘技并非游戏制作人员的本意, 请尝试解释这些秘技为什么能生效。

- A:

这些秘技能够生效的主要原因可能是：游戏存储数值变量溢出，多个功能代码复用引发意外副作用，测试覆盖不足等。

六、 必答题

- Q:

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。

- A:

游戏存档的读取步骤如下:

1. PAL_LoadGame 调用 fread 从存档文件中读取数据。
2. 库函数 fread 会触发在 navy_apps 中封装在 libos 中的系统调用 _read，最后传递到 do_syscall 中。
3. nanos-lite 收到调用后，调用 fs_read 读取数据。
4. 读取到的数据经由 nemu 映射的内存存放在内存中。

屏幕的更新步骤如下:

1. redraw 函数调用 NDL_DrawRect 更新屏幕。

```
int NDL_DrawRect(uint32_t *pixels, int x, int y, int w, int h) {
  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];
      }
    }
  }
}
```

2. 借助把显存抽象成文件的思想，可以看到该函数调用了 fwrite，printf 等函数，同样会触发系统调用，经由 ioe 中的 _draw_rect 函数更新屏幕（本质上是 memcpy）。
3. 输出的内容经由 nemu 抽象的接口显示出来。

七、 所遇 BUG 及解决思路

(一) 阶段 1 中的段错误

在第一阶段中，运行中发生了段错误，且在段错误之前并没有发生 difftest 错误。

```
src/main.c,19,main] 'Hello World!' from Nanos-lite
[src/main.c,20,main] Build time: 03:41:11, Apr 13 2025
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x100fb0, end = 0x10558c, size = 17884 bytes
[src/main.c,27,main] Initializing interrupt/exception handler...
Makefile:46: recipe for target 'run' failed
make[1]: *** [run] 段错误 (core dumped)
make[1]: Leaving directory '/home/latesoon/ics2017/nemu'
qemu-system-i386: terminating on signal 15 from pid 13020
/home/latesoon/ics2017/nexus-am/Makefile.app:35: recipe for target 'run' failed
make: *** [run] Error 2
```

图 31: 段错误

一开始想要 si 方式查找，但是因为代码量过大而放弃。后来通过移除 opcode_table 中的指令来检查（因为在那一部分主要就是完成了 iret 和 popa 两条指令）。发现在删除 iret 的实现后，出现了熟悉的指令未实现的报错。

```
invalid opcode(eip = 0x00100b90): cf 00 00 00 73 72 63 2f ...

There are two cases which will trigger this unexpected exception:
1. The instruction at eip = 0x00100b90 is not implemented.
2. Something is implemented incorrectly.
Find this eip(0x00100b90) in the disassembling result to distinguish which case it is.

If it is the first case, see
 _____   ___     __    __  __                          _
(_)___ \ / _ \   / /   |  \/  |                        | |
  __) | (_) |/ /_   | \  / | __ _ _ __ _ __   _   _  _| |
 |__ < > _ <| '_ \  | |\/| |/ _` | '__| '_ \| | | |/ _` |
 |___) | (_) | (_) | | |  | | (_| | |  | | | | |_| | (_| |
 |____/ \___/ \___/  |_|  |_|\__,_|_|  |_| |_|\__,_|\__,_|

for more details.

If it is the second case, remember:
* The machine is always right!
* Every line of untested code is always wrong!
```

图 32: 报错

因此重新检查该函数，发现是 decoding 的跳转地址未能正确赋值（先前错误直接给 cpu.eip 赋值，修改为 decoding.jmp_eip，以期在指令结束后完成跳转。

修复后完成该步骤，继续实验。

(二) 阶段 2 中的 make 问题

一开始实现在 Nanos-lite 上运行 Hello world 的时候，一开始第一次运行出了 bug，后来怎么修改 make run/make update 也一直 HIT BAD TRAP。最后意识到问题是因为修改了 navy-app 中的文件，需要回到 hello 目录下重新 make，后修复了该问题。

(三) 阶段 2 中 DEBUG 思路

在阶段 2 中，往往 HIT BAD TRAP 后也没有明显提示。

通过添加 LOG 逐行分析输出，可以成功查找到问题。

```
1 Log("fd%d len%d mode%d size%d",fd,len,mode,fs_filesz(fd));
2 Log(" newoffset %d\n",fs_offset(fd));
```

```
[src/fs.c,74,fs_read] 11:size 28448,len 28448,offset 0
[src/fs.c,42,update_offset3] fd11 len28448 mode1 size28448
[src/fs.c,47,update_offset3]  newoffset 0

[src/fs.c,49,update_offset3]  newoffset 28448

[src/fs.c,51,update_offset3]  newoffset 28448

[src/fs.c,53,update_offset3]  newoffset 28448

[src/fs.c,87,fs_read]  newoffset 28448

[src/fs.c,42,update_offset3] fd31 len0 mode2 size5000
[src/fs.c,47,update_offset3]  newoffset 5000

[src/fs.c,49,update_offset3]  newoffset 5000
```

图 33: LOG

（四） 阶段 3 运行仙剑奇侠传的显示 BUG

一开始，我的仙剑奇侠传显示如图34所示。

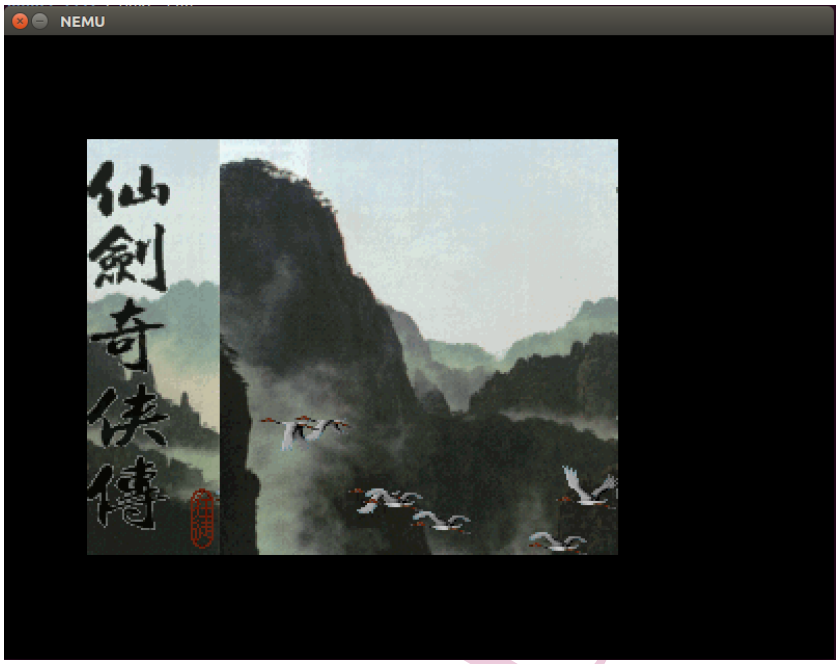


图 34: 错误的仙剑奇侠传

我首先对我的 _draw_rect 和 fb_write 函数进行了检查，没有发现逻辑性错误。于是我对 fb_write 进行 log，发现每相邻两次的偏移量是一样的，这是不合理的。最终我发现我没有在 fs_write 该 case 下更新偏移量，修复后运行成功。

```
Log("%d %d\n",offset,len);
```

```
update_offset(fd, len);
```

```
[src/device.c,28,fb_write] 350560 1024
[src/device.c,28,fb_write] 350560 256
[src/device.c,28,fb_write] 352160 1024
[src/device.c,28,fb_write] 352160 256
[src/device.c,28,fb_write] 353760 1024
[src/device.c,28,fb_write] 353760 256
```

图 35: Log 结果