

## PA3 目录-1613415 潘巧巧

加载用户程序与系统调用.....	2
入口代码分析.....	2
流程分析.....	2
Main () .....	2
Loader () .....	3
OS 保护机制：系统调用 .....	3
PA3.1 具体操作（运行 dummy.c） .....	4
标准输出与文件系统.....	10
标准输出.....	10
堆区管理.....	11
关于 sbrk()和 brk() .....	11
文件系统.....	12
文件抽象 .....	12
输出设备 .....	13
输入设备 .....	13
仙剑奇侠传.....	13
PA3.2 具体操作（运行 hello.c+/bin/text） .....	14
PA3.3 具体操作（运行/bin/bmptest） .....	20
回答问题.....	25
遇到的坑.....	28
复习的知识和学到的经验 .....	29

# 加载用户程序与系统调用

## 入口代码分析

在 nanos-lite/src/main.c 中开始进行程序加载，可以看到除了 Log 信息，有几个重要步骤，即 init\_ramdisk(), init\_device(), init\_irq(), init\_fs(), loader(), 其中 loader 是用于加载程序的核心模块。

```
int main() {
#ifdef HAS_PTE
    init_mm();
#endif

    Log("'Hello World!' from Nanos-lite");
    Log("Build time: %s, %s", __TIME__, __DATE__);

    init_ramdisk();

    init_device();

#ifdef HAS_ASYE
    Log("Initializing interrupt/exception handler...");
    init_irq();
#endif

    init_fs();

    uint32_t entry = loader(NULL, NULL);
    ((void (*)(void))entry)();

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

## 流程分析

### Main ()

加载用户程序，即把可执行文件中的**代码和数据**放置在正确的内存位置，然后跳转到程序入口，程序开始执行。

关于上述代码的运行：

- 1、Log 输出 hello 信息和编译时间
- 2、Init\_ramdisk(): 初始化 ramdisk, ramdisk 是一个暂时的磁盘，目前先把 Nanos-lite 中的一段内存作为磁盘来使用。
- 3、Init\_device(): 初始化设备
- 4、Init\_fs(): 目前暂无操作
- 5、Loader(): 加载用户程序,函数会返回用户程序的入口地址。此函数是 PA3.1 的核心。

## Loader ()

- 1、功能：加载用户程序（可执行程序）
- 2、raw program loader（要做的事）：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。
- 3、可执行程序的位置：目前的 ramdisk 中，可执行程序只有一个，故可执行文件位于 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、框架代码提供的可以用于实现 Loader()的函数

```
// 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();
```

## OS 保护机制：系统调用

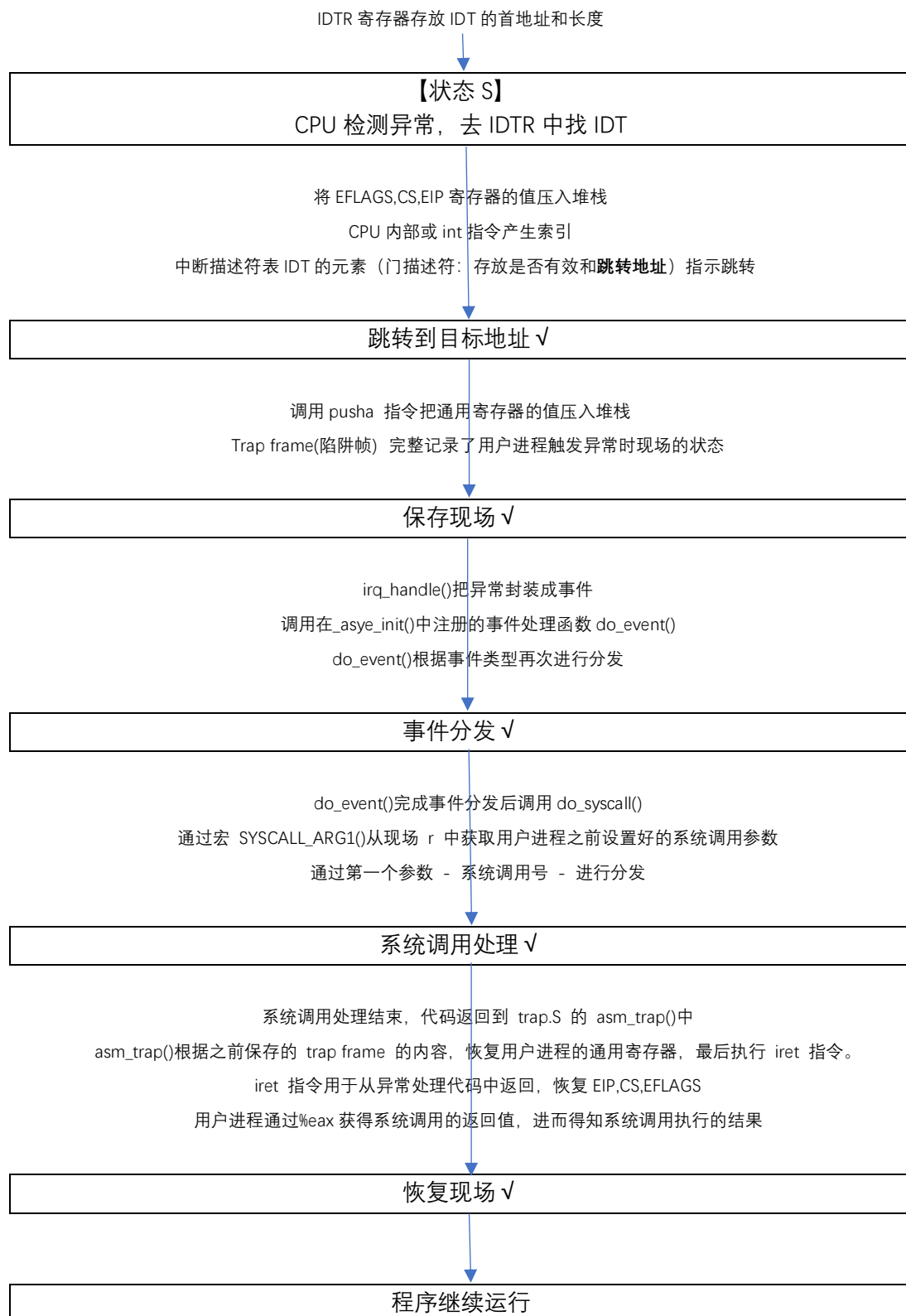
- 1、4 个特权级，0 特权级最高，3 特权级最低。
- 2、高特权级的进程才能去执行一些系统级别的操作，如果一个特权级低的进程尝试执行它没有权限执行的操作，CPU 将会抛出一个异常。操作系统拥有最高特权级（ring0），故 OS 需要管理系统中的所有资源，为用户进程提供相应的服务。
- 3、特权级相关概念：
  - DPL：一段数据所在的特权级
  - RPL：请求者所在的特权级
  - CPL：当前进程的特权级一次数据的访问操作是合法的,当且仅当：

$\begin{aligned} \text{data.DPL} &\geq \text{requestor.RPL} \\ \text{data.DPL} &\geq \text{current\_process.CPL} \end{aligned}$
---

- 4、普通 OS 中用户进程在 ring3，本实验中让所有用户进程都运行在 ring 0。
- 5、保护机制的本质：在硬件中加入一些与特权级检查相关的门电路，如果发现了非法操作就会抛出一个异常，CPU 跳转到一个固定的地方，并进行后续处理。
- 6、为了让 OS 注意到用户进程提交的申请，系统调用通常都会触发一个异常，然后陷入操作系统。在 GNU/Linux 中，系统调用产生的异常通过 `int $0x80` 指令触发。本实验中需要实现 int 操作。
- 7、**系统调用接口**：内联汇编\_syscall\_函数

```
int _syscall_(int type, uintptr_t a0, uintptr_t a1, uintptr_t a2){
    int ret = -1;
    asm volatile("int $0x80": "=a"(ret): "a"(type), "b"(a0), "c"(a1), "d"(a2));
    return ret;
}
```

## 8、异常处理过程



## PA3.1 具体操作（运行 dummy.c）

- 1、修改 Nave-app/Makefile.check，让 Navy-apps 项目上的程序默认编译到 x86 中

```
Makefile.check (~/桌面/PA/ics2018/navy-apps) - gedit
打开(O) [icon]

#ISA ?= native
ISA ?= x86
ifeq ($(NAVY_HOME), )
    $(error Must set NAVY_HOME environment variable)
endif

$(shell mkdir -p $(NAVY_HOME)/fsimg/bin/ $(NAVY_HOME)/fsimg/dev/)
```

- 2、在 navy-apps/tests/dummy 下执行 `make`，在 navy-apps/tests/dummy/build/ 目录下生成 dummy 的可执行文件。
- 3、在 nanos-lite/目录下执行 `make update`，nanos-lite/Makefile 中会将其生成 ramdisk 镜像文件 ramdisk.img，并包含进 Nanos-lite 成为其中的一部分。
- 4、利用框架代码提供的函数实现 Loader()：根据实验指导和框架代码的提示，利用 `get_ramdisk_size()` 函数获取需要读取的内容的 size，再调用 `ramdisk_read()` 从 ramdisk 中的参数 2 偏移处 (0) 的 len 字节读入参数 1 中。根据已知的 Loader 的目的，需要放置在 0x4000000 处，当前文件头已经定义好一个宏 `DEFAULT_ENTRY`，故 `ramdisk_read()` 函数的参数 1 设为这个宏，然后返回这个地址值。此时传入的参数均为 NULL，故不必对参数进行操作。

```
#define DEFAULT_ENTRY ((void *)0x4000000)
```

```
uintptr_t loader(_Protect *as, const char *filename) {
    //TODO();
    //功能：将ramdisk中从0开始的所有内容放置到0x4000000,并返回
    size_t len = get_ramdisk_size();
    ramdisk_read(DEFAULT_ENTRY, 0, len);

    return (uintptr_t)DEFAULT_ENTRY;
}
```

- 5、在 nanos-lite/目录下手动执行 `make update`，更新 Nanos-lite 中的 ramdisk 内容，然后再通过 `make run`，在 NEMU 上运行带有最新版 ramdisk 的 Nanos-lite。
- 6、遇到一个一条待实现的 int 指令，程序终止。
- 7、根据上节《OS 保护机制：系统调用》中提到的机制和实验指导，此时任务是实现 IDT。首先是在 nemu/include/cpu/reg.h 添加 IDTR 寄存器，用于存放 IDT 的首地址 (base) 和长度 (limit)。根据实验指导，为了通过 differential testing，需要在 cpu 结构体中添加一个 CS 寄存器，并在 `restart()` 函数中将其初始化为 8，EFLAGS 初始化为 0x2。

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

```
unsigned int cs;
```

```
static inline void restart() {
    /* Set the initial instruction pointer. */
    cpu.oip = ENTRY_START;
    cpu.cs = 0x8;
    cpu.eflags.eflags_init = 0x2;
}
```

- 8、实现 lidt 指令。实现指令的详细步骤如 PA2 所示。

```
/* 0x0f 0x01*/
make_group(gp7,
    EMPTY, EMPTY, EMPTY, EX(lidt),
    EMPTY, EMPTY, EMPTY, EMPTY)
```

```
make_EHelper(lidt) {
    //TODO();
    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);
}
```

- 9、定义宏 HAS\_ASYE。定义后，程序加载的入口函数 main() 中会运行 init\_irq() 函数，该函数会调用 \_asye\_init() 函数，主要功能是初始化 IDT 和注册一个事件处理函数。

```
/* Uncomment these ma
#define HAS_ASYE
// #define HAS_PTE
```

```
int main() {
#ifdef HAS_PTE
    init_mm();
#endif

    Log("Hello World!" from Nanos-lite");
    Log("Build time: %s, %s", __TIME__, __DATE__);

    init_ramdisk();

    init_device();

#ifdef HAS_ASYE
    Log("Initializing interrupt/exception handler...");
    init_irq();
#endif
    init_ts();

    //uint32_t entry = loader(NULL, NULL);
    //uint32_t entry = loader(NULL, "/bin/text");
    uint32_t entry = loader(NULL, "/bin/bmptest");
    ((void (*)(void))entry)();

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

- 10、实现 int 指令。实现指令的详细步骤如 PA2 所示。根据讲义要求，需要调用 raise\_intr() 函数实现中断机制，而不能在此实现。

```
/* 0xc0 */ IDEXW(gp2_1b2E, gp2, 1), ID
/* 0xc4 */ EMPTY, EMPTY, IDEXW(mov_I2E,
/* 0xc8 */ EMPTY, EX(leave), EMPTY EMP
/* 0xcc */ EMPTY, IDEXW(I, int, 1) EMP
/* 0xd0 */ IDEXW(gp2_1-E, gp2, 1), IDEX
/* 0xd4 */ EMPTY, EMPTY, EX(nemu_trap),
/* 0xd8 */ EMPTY, EMPTY, EMPTY, EMPTY,
```

```
extern void raise_intr(uint8_t NO, vaddr_t ret_addr);
make_EHelper(int) {
    //TODO(); 中断指令
    raise_intr(id_dest->val, decoding.seq_eip);
    print_asm("int %s", id_dest->str);

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

- 11、实现 raise\_intr() 函数。根据讲义和上节描述实现该函数，需要经过 5 个步骤帮助

CPU 从 S 状态跳转到新的地方，并保存当前状态。

```
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();
    //模拟i386中断机制处理过程
    //1、当前状态压栈
    rtl_push((rtlreg_t *)&cpu.eflags);
    cpu.eflags.IF=0;
    rtl_push((rtlreg_t *)&cpu.cs);
    rtl_push((rtlreg_t *)&ret_addr);

    //2、从idtr中读出首地址
    uint32_t idtr_base = cpu.idtr.base;

    //3、索引，找到门描述符
    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;

    //4、将门描述符中的offset域组合成目标地址
    offset = eip_low | eip_high;

    //5、跳转到目标地址
    decoding.jmp_eip = offset;
    decoding.is_jmp = 1;
}
```

- 12、 键入 make update 和 make run，再次运行 dummy 程序遇到与讲义相符的未触发指令。
- 13、 为了保存现场，实现 pusha 指令。实现指令流程见 PA2。注意查阅 i386 手册注意压栈顺序

```
Operation

IF OperandSize = 16 (* PUSHAX 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
```

```
/* 0x58 */ IDEX(r, pop
/* 0x5c */ IDEX(r, pop
/* 0x60 */ EX(pusha),
/* 0x64 */ EMPTT, EMPT
/* 0x68 */ IDEX(push_S
```

```
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");
}
```

- 14、 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。

```
#----|-----entry-----|-----errorcode-----|---irq id---|---handler---|
.globl vecsys;   vecsys:   pushl $0;   pushl $0x80; jmp asm_trap
.globl vecnull;  vecnull:  pushl $0;   pushl $-1;  jmp asm_trap

asm_trap:
    pushal

    pushl %esp
    call irq_handle

    addl $4, %esp

    popal
    addl $8, %esp

    iret
```

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

- 15、 重新运行 dummy.c 程序，触发一个 ID=8 的 BAD TRAP
- 16、 实现系统调用：在 do\_event()中识别出系统调用事件\_EVENT\_SYSCALL，然后调用 do\_syscall()。

```
extern _RegSet* do_syscall(_RegSet *r);

//时间分发
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;
}
```

- 17、 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

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

- 18、 添加 SYS\_none 系统调用。根据手册提示，这个系统调用什么都不用做，直接返回 1。



```
switch (a[0]) {
case SYS_none:
    neweax = 1;
    break;
}
```

19、 设置系统调用的返回值。

```
SYSCALL_ARG1(r) = neweax;
return NULL;
```

20、 实现 popa。具体步骤详见 PA2。实现 make\_EHelper 函数的时候注意查看 i386 手册查看 pop 顺序。

```
/* 0x58 */ IDEX(r, pop), IDEX(r, pop), I
/* 0x5c */ IDEX(r, pop), IDEX(r, pop), I
/* 0x60 */ EX(pusha), EX(popa), EMPTY, E
/* 0x64 */ EMPTY, EMPTY, EX(operand size
/* 0x68 */ IDEX(push SI, push), EMPTY, I
```

```
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;
```

```
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");
}
```

21、 实现 iret 指令。具体步骤详见 PA2。

```
v_12r, mov), IDEX(mov
EX(gp2_1b2E, gp2), EM
, mov, 1), IDEX(mov_1
PTY
PTY EX(iret),
X(gp2_1.E_gp2), IDEX
, EMPTY,
```

```
make_EHelper(iret) {
    //TODO();
    rtl_pop(&decoding.jump_eip);
    rtl_pop(&cpu.cs);
    rtl_pop(&cpu.eflags.eflags_init);
    decoding.is_jump = 1;
    print_asm("iret");
}
```

- 22、 重新运行 dummy 程序，看到 dummy 程序触发一个号码为 4 的系统调用 (SYS\_exit)。此时实现 SYS\_exit。根据手册提示，实现时只需接收一个退出状态的参数 (a[1])，用这个参数调用\_halt()即可。

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

- 23、 重新运行 dummy 程序，看到 GOOD TRAP 的信息。PA3.1 结束。

```
[src/monitor/diff-test/diff-test.c,96,init_difftest] Connect to QEMU successfully
[src/monitor/monitor.c,67,load_img] The image is /home/pqq/桌面/PA/ics2018/nanos-lite/build/nanos-lite-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,32,welcome] Build time: 09:41:59, Jun 3 2019
For help, type "help"
(nemu) c
[src/main.c,19,main] 'Hello World!' from Nanos-lite
[src/main.c,20,main] Build time: 10:06:38, Jun 3 2019
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x101c40, end = 0x106220, size = 17952 bytes
[src/main.c,27,main] Initializing interrupt/exception handler...
loader end
nemu: HIT GOOD TRAP at eip = 0x00100032
```

## 标准输出与文件系统

### 标准输出

在 hello.c 中，首先通过 write()来输出一句话，然后通过 printf()来不断输出。在实现了实现 SYS\_write 系统调用后可以执行。但在 sys\_write()函数中使用 Log 打印信息，会发现每个字母都需要进入 SYS\_write。我们加载的用户程序在第一次调用 printf()的时候会尝试通过 malloc()申请一片缓冲区来存放格式化的内容。若申请失败会逐个字符进行输出。

```
[src/monitor/monitor.c,32,welcome] Build time: 09:41:59, Jun 3
For help, type "help"
(nemu) c
[src/main.c,19,main] 'Hello World!' from Nanos-lite
[src/main.c,20,main] Build time: 10:06:38, Jun 3 2019
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x101c40,
[src/main.c,27,main] Initializing interrupt/exception handler...
loader end
[src/syscall.c,13,sys_write] enter sys_write
Hello World!
[src/syscall.c,13,sys_write] enter sys_write
H[src/syscall.c,13,sys_write] enter sys_write
e[src/syscall.c,13,sys_write] enter sys_write
l[src/syscall.c,13,sys_write] enter sys_write
l[src/syscall.c,13,sys_write] enter sys_write
o[src/syscall.c,13,sys_write] enter sys_write
[src/syscall.c,13,sys_write] enter sys_write
W[src/syscall.c,13,sys_write] enter sys_write
o[src/syscall.c,13,sys_write] enter sys_write
e[src/syscall.c,13,sys_write] enter sys_write
f[src/syscall.c,13,sys_write] enter sys_write
f[src/syscall.c,13,sys_write] enter sys_write
o[src/syscall.c,13,sys_write] enter sys_write
r[src/syscall.c,13,sys_write] enter sys_write
[src/syscall.c,13,sys_write] enter sys_write
l[src/syscall.c,13,sys_write] enter sys_write
d[src/syscall.c,13,sys_write] enter sys_write
[src/syscall.c,13,sys_write] enter sys_write
e[src/syscall.c,13,sys_write] enter sys_write
e[src/syscall.c,13,sys_write] enter sys_write
t[src/syscall.c,13,sys_write] enter sys_write
e[src/syscall.c,13,sys_write] enter sys_write
e[src/syscall.c,13,sys_write] enter sys_write
t[src/syscall.c,13,sys_write] enter sys_write
n[src/syscall.c,13,sys_write] enter sys_write
n[src/syscall.c,13,sys_write] enter sys_write
e[src/syscall.c,13,sys_write] enter sys_write
```

# 堆区管理

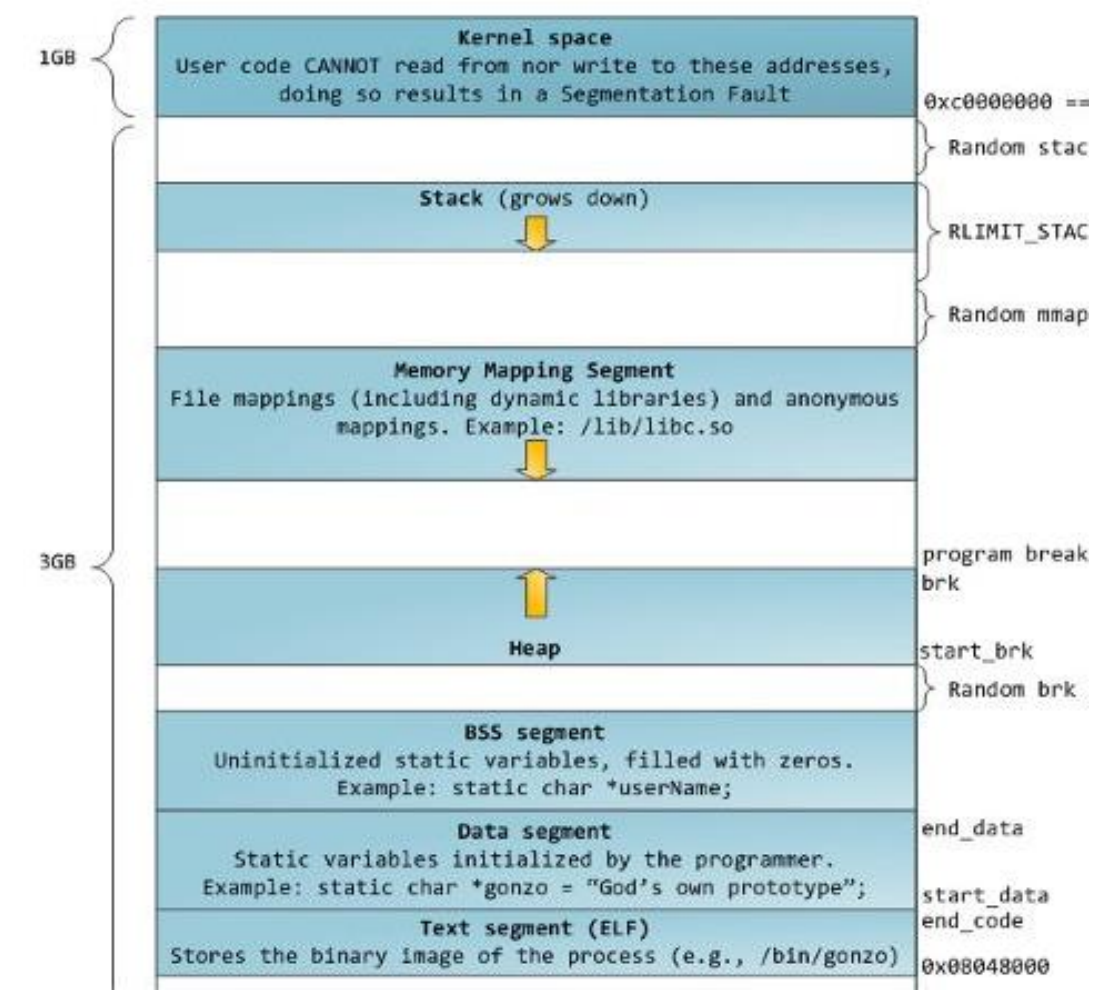
操作系统的任务之一是进行内存区域的管理，而堆区的本质就是一片内存区域。OS 中调整堆区大小的函数是 **sbrk()**，这个函数用于将用户程序的 program break（用户程序的数据段 data segment 结束的位置）增长 increment（可正可负）字节。

可执行文件里面有**代码段**和**数据段**，链接的时候 ld 会默认添加一个名为 **\_end** 的符号来指示程序的数据段结束的位置。用户程序开始运行的时候 program break 会位于 **\_end** 所指示的位置，意味着此时堆区的大小为 0。

malloc()被第一次调用的时候会通过 sbrk(0)来查询用户程序当前 program break 的位置，之后就可以通过后续的 sbrk()调用来动态调整用户程序 program break 的位置。当前 program break 和其初始值之间的区间就可以作为用户程序的堆区，由 malloc()/free()进行管理。

## 关于 sbrk()和 brk()

这两个函数都用来改变 "program break" (程序中断点)的位置，这个位置可参考下图：



我们在 terminal 中键入“man 2 sbrk”查看信息

```
DESCRIPTION
brk() and sbrk() change the location of the program break, which
defines the end of the process's data segment (i.e., the program break
is the first location after the end of the uninitialized data segment).
Increasing the program break has the effect of allocating memory to the
process; decreasing the break deallocates memory.

brk() sets the end of the data segment to the value specified by addr,
when that value is reasonable, the system has enough memory, and the
process does not exceed its maximum data size (see setrlimit(2)).

sbrk() increments the program's data space by increment bytes. Calling
sbrk() with an increment of 0 can be used to find the current location
of the program break.
```

根据该描述知：

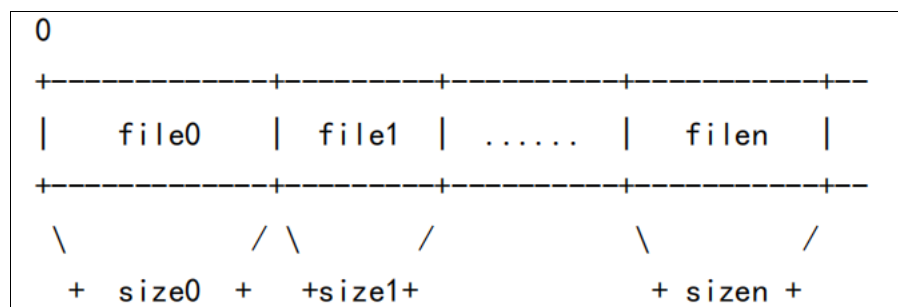
**sbrk(increment)** 每次让堆往栈的方向增加 increment 个字节的大小空间。

**brk()** 的参数是一个地址，假如你已经知道了堆的起始地址和堆的大小就可以据此修改 brk() 中的地址参数已达到调整堆的目的。

## 文件系统

### 文件抽象

- 1、文件的数量增加之后，我们需要知道哪个文件在 ramdisk 的什么位置。故操作系统还需要在存储介质的驱动程序之上为用户程序提供一种更高级的抽象——文件。文件的本质就是字节序列+一些额外属性。
- 2、在 PA 实验中，我们约定文件的数量和大小固定，故可以把每一个文件分别固定在 ramdisk 中的某一个位置，即约定文件从 ramdisk 的最开始一个挨着一个地存放。



- 3、为了记录文件相关信息，我们需要文件记录表 **file\_table**，文件记录表的每一个元素是一个 **struct Finfo**，包含以下信息。

```
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;    //文件被打开后的读写指针(新添加的成员)
} Finfo;
```

- 4、文件系统不是通过文件名来保存文件(有的文件没有名字且检索复杂)，而是通过“编号”，即**文件描述符**。一个文件描述符对应一个正在打开的文件，由操作系统来维护文件描述符到具体文件的映射。在 PA 实验中，由于简易文件系统中的文件数目是固定的，我们在用 open 函数打开一个文件时，可以简单地把文件记录表的下标作为相应文件的文件

描述符返回给用户程序。

```
//文件描述符
enum {FD_STDIN, FD_STDOUT, FD_STDERR, FD_FB, FD_EVENTS, FD_DISPINFO, FD_NORMAL};
```

- 5、由于文件的本质是字节序列，故我们可以把一切抽象成文件。如内存、管道、磁盘、网络套接字等，于是我们可以利用文件为不同的事物提供了统一的接口。

## 输出设备

- 1、对于 PA 实验暂时不需要考虑串口，目前只需要把显存抽象成文件。显存本身也是一段存储空间，以行优先的方式存储了将要在屏幕上显示的像素。
- 2、PA 中把显存抽象成文件/dev/fb，即 Frame Buffer，需要获得 fb.c 中 write 和 lseek 函数的支持。
- 3、用户程序还需要获得屏幕大小的信息，通过/proc/dispinfo 文件来获得，需要 fb.c 中 read 函数的支持。

## 输入设备

- 1、把输入设备有**键盘**和**时钟**的输入包装成事件，即把事件以文本的形式表现出来。文本是一种字节序列，这使得事件很容易抽象成文件，且文本方式使得用户程序可以容易可读地解析事件的内容。
- 2、我们定义以下事件，一个事件以换行符\n 结束:  
**t 1234**: 返回系统启动后的时间，单位为毫秒  
**kd RETURN/ku A**: 按下/松开按键，按键名称全部大写，使用 AM 中定义的按键名。
- 3、上述事件抽象成文件/dev/events，需要支持读操作，用户程序可以从其中一次读出一个输入事件。由于时钟事件可以任意时刻进行读取，故需要优先处理按键事件，当不存在按键事件的时候才返回时钟事件，否则用户程序将永远无法读到按键事件。

## 仙剑奇侠传

- 1、仙剑奇侠传是 Nanos-lite 的用户程序，Nanos-lite 是 NEMU 的用户程序
- 2、移植的主要工作就是把应用层之下提供给仙剑奇侠传的所有 API 重新实现一遍，因为这些 API 大多都依赖于操作系统提供的运行时环境，我们需要根据 Navy-apps 提供的运行时环境重写它们。主要包括三部分内容: C 标准库、浮点数、SDL 库。

## PA3.2 具体操作（运行 hello.c+/bin/text）

- 1、把 Nanos-lite 上运行的用户程序切换成 hello 程序。在 hello 目录下运行 make，再到 Nanos-lite 下运行 make update 和 make run

```
RAMDISK_FILE = build/ramdisk.img

OBJCOPY_FLAG = -S --set-section-flags .bss=alloc,contents -O binary
OBJCOPY_FILE = $(NAVY_HOME)/tests/hello/build/hello-x86

.PHONY: update update-ramdisk-objcopy update-ramdisk-fsimg update-fsimg
update-ramdisk-objcopy:
```

- 2、实现 SYS\_write。包括在 do\_syscall 函数中添加 SYS\_write 的 case，实现 sys\_write 函数，根据手册提示，在 terminal 中输入 man 2 write 查看该函数相关信息以及返回值怎么写。以及根据手册提示在 navy-apps/libs/libos/src/nanos.c 的 \_write() 中调用系统调用接口函数。

```
case SYS_write:
    neweax = sys_write(a[1], (void *)a[2], a[3]);
    break;
```

### 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.

```
uintptr_t sys_write(int fd, const void *buf, size_t count) {
    uintptr_t i = 0;
    if (fd == 1 || fd == 2) {
        for(; count > 0; count--) {
            _putc(((char*)buf)[i]);
            i++;
        }
    }
    return i;
}
```

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

- 3、在 Nanos-lite 中实现 SYS\_brk 系统调用 case。根据手册提示直接返回 0 即可。

```
case SYS_brk:
    neweax = 0;
    break;
```

- 4、实现 \_sbrk()。根据手册提示可以通过 man 3 end 来查阅如何使用\_end 符号，根据上节对 sbrk 的查询和学习实现函数。



```

END(3)                                     Linux Programmer's Manual                                     END(3)

NAME
  etext, edata, end - end of program segments

SYNOPSIS
  extern etext;
  extern edata;
  extern end;

DESCRIPTION
  The addresses of these symbols indicate the end of various program segments:
  etext This is the first address past the end of the text segment (the program code).
  edata This is the first address past the end of the initialized data segment.
  end This is the first address past the end of the uninitialized data segment (also known as the BSS segment).

CONFORMING TO
  Although these symbols have long been provided on most UNIX systems, they are not standardized; use with caution.

NOTES
  The program must explicitly declare these symbols; they are not defined in any header file.
  On some systems the names of these symbols are preceded by underscores, thus: _etext, _edata, and _end. These symbols are also defined for programs compiled on Linux.
  At the start of program execution, the program break will be somewhere near end (perhaps at the start of the following page). However, the break will change as memory is allocated via brk(2) or malloc(3). Use sbrk(2) with an argument of zero to find the current value of the program break.

EXAMPLE
  When run, the program below produces output such as the following:

  $ ./a.out
  First address past:
    program text (etext)      0x8048568
    initialized data (edata)  0x804a01c
    uninitialized data (end)  0x804a024

  Program source
  #include <stdio.h>
  #include <stdlib.h>

  extern char etext, edata, end; /* The symbols must have some type,
                                or 'gcc -Wall' complains */

```

```

extern char _end;
intptr_t program_break = (intptr_t)&_end;

```

```

void *_sbrk(intptr_t increment){
    intptr_t old_pb = program_break;
    int test=_syscall_(SYS_brk, old_pb + increment, 0, 0);
    if (test == 0)
    {
        program_break += increment;
        return (void *)old_pb;
    }
    else
        return (void *)-1;
}

```

- 5、再次运行 hello.c，记住这里要先到 hello.c 的目录 make，然后再 make update 和 make run，否则无法更新 `sbrk()` 的实现。如果成功实现，则可以看到此时 `printf` 打印的时候不是一个一个字符打印，而是把需要打印的一整句话加载完毕后再调用一次 `SYS_write` 进行打印。Hello.c 运行成功，当前任务暂时结束。

```

pqq@pqq-virtual-machine:~/桌面/PA/ics2018/navy-apps/tests/hello$ make
make -C /home/pqq/桌面/PA/ics2018/navy-apps/libs/libc
make[1]: Entering directory '/home/pqq/桌面/PA/ics2018/navy-apps/libs/libc'
make[1]: Nothing to be done for 'archive'.
make[1]: Leaving directory '/home/pqq/桌面/PA/ics2018/navy-apps/libs/libc'
make -C /home/pqq/桌面/PA/ics2018/navy-apps/libs/libos
make[1]: Entering directory '/home/pqq/桌面/PA/ics2018/navy-apps/libs/libos'
+ CC src/nanos.c
+ AR /home/pqq/桌面/PA/ics2018/navy-apps/libs/libos/build/libos-x86.a
make[1]: Leaving directory '/home/pqq/桌面/PA/ics2018/navy-apps/libs/libos'
+ LD /home/pqq/桌面/PA/ics2018/navy-apps/tests/hello/build/hello-x86

```

```

loader end
[src/syscall.c,13,sys_write] enter sys_write
Hello World!
[src/syscall.c,13,sys_write] enter sys_write
Hello World for the 2th time
[src/syscall.c,13,sys_write] enter sys_write
Hello World for the 3th time
[src/syscall.c,13,sys_write] enter sys_write
Hello World for the 4th time
[src/syscall.c,13,sys_write] enter sys_write
Hello World for the 5th time
[src/syscall.c,13,sys_write] enter sys_write
Hello World for the 6th time
[src/syscall.c,13,sys_write] enter sys_write
Hello World for the 7th time
[src/syscall.c,13,sys_write] enter sys_write
Hello World for the 8th time
[src/syscall.c,13,sys_write] enter sys_write
Hello World for the 9th time
[src/syscall.c,13,sys_write] enter sys_write
Hello World for the 10th time
[src/syscall.c,13,sys_write] enter sys_write
Hello World for the 11th time

```

- 6、现在进入文件系统实现阶段，第一个任务是对 nanos-lite/Makefile 作如下修改。然后运行 `make update` 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，同时生成这个 ramdisk 镜像的文件记录表 nanos-lite/src/files.h。

```

src/syscall.n: $(NAVY_HOME)/LLDS/LLDOS/src/sysc
ln -sf $^ $@

update: update-ramdisk-fsimg src/syscall.h
@touch src/incl.d.3

```

- 7、为 Finfo 添加一个成员 `open_offset`（偏移量属性）来记录目前文件操作的位置。每次对文件读写了多少个字节，偏移量就前进多少。

```

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;    //文件被打开后的读写指针(新添加的成员)
} Finfo;

```

- 8、修改 `main()` 和 `loader()`，让 `loader` 直接调用 `ramdisk_read()` 来加载用户程序，享受文件系统的便利性，此后更换用户程序只需要修改传入 `loader()` 函数的文件名而无需手动更新 ramdisk 的内容。

```

int main() {
#ifdef HAS_PTE
    init_mm();
#endif

    Log("'Hello World!' from Nanos-lite");
    Log("Build time: %s, %s", __TIME__, __DATE__);

    init_ramdisk();
    init_device();

#ifdef HAS_ASSE
    Log("Initializing interrupt/exception handler...");
    init_irq();
#endif
    init_fs();

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

    printf("loader end\n");
    ((void (*)(void))entry)();

    panic("Should not reach here");
}

```



```

uintptr_t loader(_Protect *as, const char *filename) {
    //TODO();
    //功能: 将ramdisk中从0开始的所有内容放置到0x4000000,并返回
    /*size_t len = get_ramdisk_size();
    ramdisk_read(DEFAULT_ENTRY,0,len);
    return (uintptr_t)DEFAULT_ENTRY;*/
    int fd = fs_open(filename, 0, 0);           //打开文件
    printf("fd = %d\n", fd);
    fs_read(fd, DEFAULT_ENTRY, fs_filesz(fd)); //读文件
    fs_close(fd);                             //关闭文件
    return (uintptr_t)DEFAULT_ENTRY;
}

```

9、实现上图中的 fs\_open()、fs\_read()、fs\_close()、fs\_filesz()函数。

- 1) **fs\_open()**: 根据 man 2 open 知, open 的第一个参数是需要打开的文件名字, 需要遍历查找 file\_table, 如果查找到目标文件则返回文件描述符。根据流程文档, 在 PA 实验中可以打开所以已有文件, 故省略对 flags 和 mode 的操作。

```

int fs_open(const char *pathname, int flags, int mode) {
    for (int i = 0; i < NR_FILES; i++)
        if (strcmp(file_table[i].name, pathname) == 0)
            //找到了目标文件
            return i; //返回文件描述符

    //没找到目标文件
    assert(0);
    return -1;
}

```

- 2) **fs\_read()**: 读文档函数。根据文档提示需要注意不能越过文件边界, 以及使用 ramdisk\_read 进行文件读。

```

ssize_t fs_read(int fd, void *buf, size_t len) {
    ssize_t fs_size = fs_filesz(fd);

    //不能越过文件边界
    if(file_table[fd].open_offset >= fs_size || len == 0)
        return 0;
    if(file_table[fd].open_offset + len > fs_size)
        len = fs_size - file_table[fd].open_offset;
    ramdisk_read(buf, file_table[fd].disk_offset + file_table[fd].open_offset, len);
    file_table[fd].open_offset += len;

    return len;
}

```

```

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

```

- 3) **fs\_close()**: 直接返回 0, 表示总是关闭成功。

```

int fs_close(int fd) {
    return 0;
}

```

- 4) **fs\_filesz()**: read 和 write 的辅助函数, 用于返回文件描述符 fd 所描述的文件的大小。直接到数组 file\_table 中读取。

```

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

```

10、完善 sys\_call.c 和 nanos.c 的各个函数和系统调用。

```

switch (a[0]) {
case SYS_none:
    result = 1;
    break;
case SYS_exit:
    halt(a[1]);
    break;
case SYS_write:
    //result = sys_write(a[1], (void *)a[2], a[3]);
    result = fs_write(a[1], (void *)a[2], a[3]);
    break;
case SYS_read:
    result = fs_read(a[1], (void *)a[2], a[3]);
    break;
case SYS_brk:
    result = 0;
    break;
case SYS_open:
    result = fs_open((char *)a[1], a[2], a[3]);
    break;
case SYS_close:
    result = fs_close(a[1]);
    break;
case SYS_lseek:
    result = fs_lseek(a[1], a[2], a[3]);
    break;
default: panic("Unhandled syscall ID = %d", a[0]);
}

```

```

void _exit(int status) {
    _syscall_(SYS_exit, status, 0, 0);
}

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

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

void *_sbrk(intptr_t increment) {
    intptr_t old_pb = program_break;
    if (_syscall_(SYS_brk, old_pb + increment, 0, 0) == 0) {
        program_break += increment;
        return (void *)old_pb;
    }
    else {
        return (void *)-1;
    }
}

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

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

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

```

- 11、 再次 make update 和 make run 运行程序 hello.c, 通过, 运行成功。

```

[src/monitor/monitor.c,32,welcome] Build time: 09:41:59, Jun  3 2019
For help, type "help"
(nemu) c
[src/main.c,19,main] 'Hello World!' from Nanos-lite
[src/main.c,20,main] Build time: 21:57:43, Jun  3 2019
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x101bc0, end = 0x3705ea,
size = 2550314 bytes
[src/main.c,27,main] Initializing interrupt/exception handler...
fd = 17
loader end
Hello World!
Hello World for the 2th time
Hello World for the 3th time

```

- 12、 修改 main()函数使其准备运行/bin/text。这个测试程序用于进行一些简单的文件读写和定位操作。

```

int main() {
#ifdef HAS_PTE
    init_mm();
#endif

    Log("Hello World!' from Nanos-lite");
    Log("Build time: %s, %s", __TIME__, __DATE__);

    init_ramdisk();

    init_device();

#ifdef HAS_ASYNC
    Log("Initializing interrupt/exception handler...");
    init_irq();
#endif
    init_fs();

    //uint32_t entry = loader(NULL, NULL);
    //uint32_t entry = loader(NULL, "/bin/hello");
    uint32_t entry = loader(NULL, "/bin/text");
    printf("loader end\n");
    ((void (*)(void))entry)();

    panic("Should not reach here");
}

```

13、 实现 fs\_write()和 fs\_lseek()。

- 1) **fs\_write()**: 写文档函数。根据文档提示需要注意不要越过文件边界，并且使用 ramdisk\_write 函数实现写入功能。且写入 stdout 和 stderr 的时候用\_putc()输出到串口。

```

ssize_t fs_write(int fd, const void *buf, size_t len) {
    ssize_t fs_size = fs_filesz(fd);
    // 写入ramdisk
    if(file_table[fd].open_offset >= fs_size)//注意偏移量不要越过文件边界
        return 0;
    if(file_table[fd].open_offset + len > fs_size)
        len = fs_size - file_table[fd].open_offset;

    ramdisk_write(buf, file_table[fd].disk_offset + file_table[fd].open_offset, len);
    file_table[fd].open_offset += len;

    return len;
}

```

- 2) **fs\_lseek()**: 调整偏移量的函数。

```

off_t fs_lseek(int fd, off_t offset, int whence) {
    off_t result = -1;

    switch(whence) {
        case SEEK_SET:
            if (offset >= 0 && offset <= file_table[fd].size) {
                file_table[fd].open_offset = offset;
                result = file_table[fd].open_offset;
            }
            break;
        case SEEK_CUR:
            if ((offset + file_table[fd].open_offset >= 0) &&
                (offset + file_table[fd].open_offset <= file_table[fd].size)) {
                file_table[fd].open_offset += offset;
                result = file_table[fd].open_offset;
            }
            break;
        case SEEK_END:
            file_table[fd].open_offset = file_table[fd].size + offset;
            result = file_table[fd].open_offset;
            break;
    }

    return result;
}

```

14、 运行/bin/text。显示“PASS!!!”和“GOOD TRAP”，PA3.2 结束。

```

For help, type "help"
(nemu) c
[src/main.c,19,main] 'Hello World!' from Nanos-lite
[src/main.c,20,main] Build time: 16:11:40, Jun  2 2019
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x101920, end = 0x37034a,
size = 2550314 bytes
[src/main.c,27,main] Initializing interrupt/exception handler...
fd = 14
PASS!!!
nemu: HIT GOOD TRAP at eip = 0x00100032

```

## PA3.3 具体操作（运行/bin/bmptest）

- 1、现在的任务是把 VGA 显存抽象成文件。在 `init_fs()` 中对文件记录表中 `/dev/fb` 的大小进行初始化，且需要使用 IOE 定义的 API 来获取屏幕的大小。

```

void init_fs() {
    // TODO: initialize the size of /dev/fb
    file_table[FD_FB].size = _screen.height * _screen.width * 4;
}

```

- 2、在 `fs_write()` 的实现中对 `FD_FB` 进行"重定向"。

```

ssize_t fs_write(int fd, const void *buf, size_t len) {
    ssize_t fs_size = fs_filesz(fd);
    switch(fd) {
        case FD_STDOUT:
        case FD_STDERR:
            for(int i = 0; i < len; i++) {
                _putc(((char*)buf)[i]);
            }
            break;
        case FD_FB:
            fb_write(buf, file_table[fd].open_offset, len);
            file_table[fd].open_offset += len;
            break;
        default:
            if(file_table[fd].open_offset >= fs_size)
                return 0;
            if(file_table[fd].open_offset + len > fs_size)
                len = fs_size - file_table[fd].open_offset;
            ramdisk_write(buf, file_table[fd].disk_offset + file_table[fd].open_offset, len);
            file_table[fd].open_offset += len;
            break;
    }
    return len;
}

```

- 3、实现 `fb_write()`，用于把 `buf` 中的 `len` 字节写到屏幕上 `offset` 处。需要先从 `offset` 计算出屏幕上的坐标，然后调用 IOE 的 `_draw_rect()` 接口。

```

void fb_write(const void *buf, off_t offset, size_t len) {
    int x, y;
    int len1, len2, len3;
    offset = offset >> 2;
    y = offset / _screen.width;
    x = offset % _screen.width;

    len = len >> 2;
    len1 = len2 = len3 = 0;

    len1 = len <= _screen.width - x ? len : _screen.width - x;
    _draw_rect((uint32_t *)buf, x, y, len1, 1);

    if (len > len1 && ((len - len1) > _screen.width)) {
        len2 = len - len1;
        _draw_rect((uint32_t *)buf + len1, 0, y + 1, _screen.width, len2 / _screen.width);
    }

    if (len - len1 - len2 > 0) {
        len3 = len - len1 - len2;
        _draw_rect((uint32_t *)buf + len1 + len2, 0, y + len2 / _screen.width + 1, len3, 1);
    }
}

```

- 4、在 `init_device()` 中将 `/proc/dispinfo` 的内容提前写入到字符串 `dispinfo` 中

```

void init_device() {
    _ioe_init();

    // TODO: print the string to array `dispinfo` with the format
    // described in the Navy-apps convention
    strcpy(dispinfo, "WIDTH:400\nHEIGHT:300");
}

```

- 5、在 fs\_read()的实现中对 FD\_DISPINFO 进行"重定向"。

```

ssize_t fs_read(int fd, void *buf, size_t len) {
    ssize_t fs_size = fs_filesz(fd);
    switch(fd) {
        case FD_STDOUT:
        case FD_FB:
            break;
        case FD_DISPINFO:
            if (file_table[fd].open_offset >= file_table[fd].size)
                return 0;
            if (file_table[fd].open_offset + len > file_table[fd].size)
                len = file_table[fd].size - file_table[fd].open_offset;
            dispinfo_read(buf, file_table[fd].open_offset, len);
            file_table[fd].open_offset += len;
            break;
        default:
            if (file_table[fd].open_offset >= fs_size || len == 0)
                return 0;
            if (file_table[fd].open_offset + len > fs_size)
                len = fs_size - file_table[fd].open_offset;
            ramdisk_read(buf, file_table[fd].disk_offset + file_table[fd].open_offset, len);
            file_table[fd].open_offset += len;
            break;
    }
    return len;
}

```

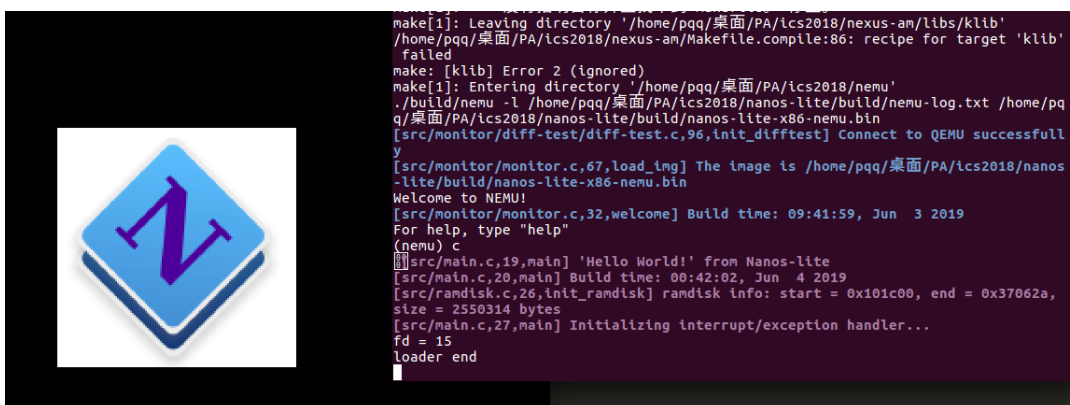
- 6、实现 dispinfo\_read(), 用于把字符串 dispinfo 中 offset 开始的 len 字节写到 buf 中。

```

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

```

- 7、让 Nanos-lite 加载/bin/bmptest, 实现正确, 看到屏幕上显示 ProjectN 的 Logo。VGA 抽象成功。



- 8、现在要把设备输入抽象成文件。实现 events\_read(), 把事件写入到 buf 中, 最长写入 len 字节, 并返回写入的实际长度。需要借助 IOE 的 API 来获得设备的输入。

```

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

```

9、在文件系统的 fs\_read 函数中添加对/dev/events 的支持。

```

ssize_t fs_read(int fd, void *buf, size_t len) {
    ssize_t fs_size = fs_filesz(fd);
    switch(fd) {
        case FD_STDOUT:
        case FD_FB:
            break;
        case FD_EVENTS:
            len = events_read((void *)buf, len);
            break;
        case FD_DISPINF0:
            if (file_table[fd].open_offset >= file_table[fd].size)
                return 0;
            if (file_table[fd].open_offset + len > file_table[fd].size)
                len = file_table[fd].size - file_table[fd].open_offset;
            dispinfo_read(buf, file_table[fd].open_offset, len);
            file_table[fd].open_offset += len;
            break;
        default:
            if (file_table[fd].open_offset >= fs_size || len == 0)
                return 0;
            if (file_table[fd].open_offset + len > fs_size)
                len = fs_size - file_table[fd].open_offset;
            ramdisk_read(buf, file_table[fd].disk_offset + file_table[fd].open_offset, len);
            file_table[fd].open_offset += len;
            break;
    }
    return len;
}

```

10、运行/bin/event

```

pqq@pqq-virtual-machine: ~/桌面/PA/ics2018/nanos-lite
./build/nemu -l /home/pqq/桌面/PA/ics2018/nanos-lite/build/nemu
q/桌面/PA/ics2018/nanos-lite/build/nanos-lite-x86-nemu.bin
[src/monitor/diff-test/diff-test.c,96,init_diffptest] Connect to
y
[src/monitor/monitor.c,67,load_img] The image is /home/pqq/桌面
-lite/build/nanos-lite-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,32,welcome] Build time: 09:41:59, Jun 3
For help, type "help"
(nemu) c
[src/main.c,19,main] 'Hello World!' from Nanos-lite
[src/main.c,20,main] Build time: 14:12:55, Jun 4 2019
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x102110,
size = 29663105 bytes
[src/main.c,27,main] Initializing interrupt/exception handler..
[src/fs.c,44,fs_open] pathname /bin/events

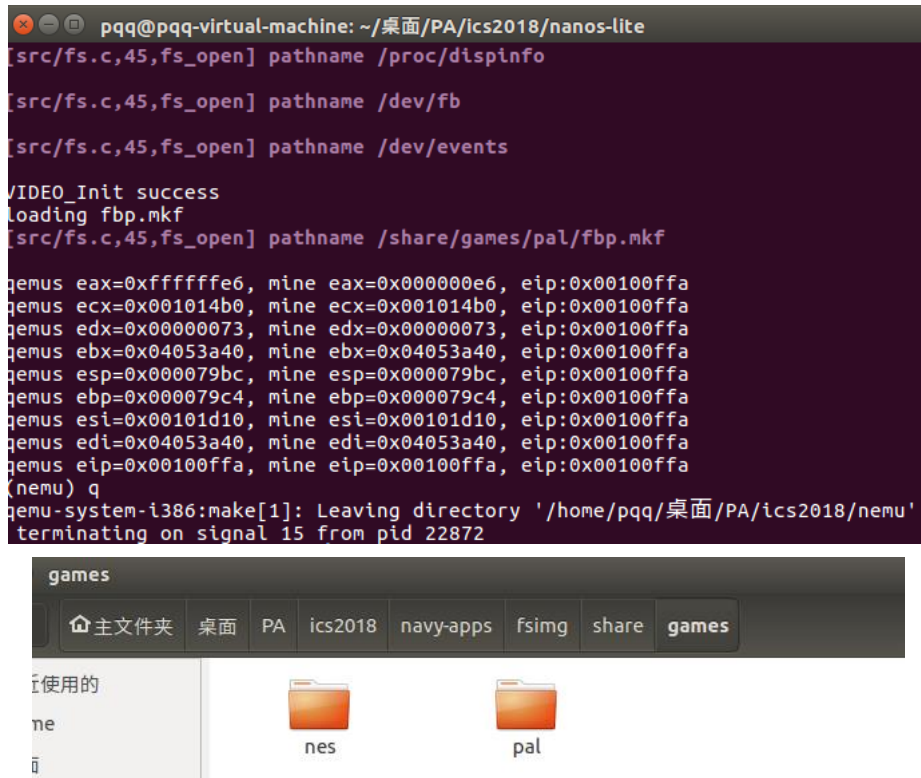
fd = 6
loader end
[src/fs.c,44,fs_open] pathname /dev/events

enter_read
receive event: t 2540401

```

(输出太慢了，这句话跑了1.5h)

- 11、 找助教或到 github 上下载仙剑奇侠传的数据文件，并放到 navy-apps/fsimg/share/games/pal/目录下。否则会加载失败。



The top part of the image shows a terminal window with the following output:

```
pqq@pqq-virtual-machine: ~/桌面/PA/ics2018/nanos-lite
[src/fs.c,45,fs_open] pathname /proc/dispinfo
[src/fs.c,45,fs_open] pathname /dev/fb
[src/fs.c,45,fs_open] pathname /dev/events
VIDEO_Init success
loading fbp.mkf
[src/fs.c,45,fs_open] pathname /share/games/pal/fbp.mkf
qemus eax=0xffffffff, mine eax=0x000000e6, eip:0x00100ffa
qemus ecx=0x001014b0, mine ecx=0x001014b0, eip:0x00100ffa
qemus edx=0x00000073, mine edx=0x00000073, eip:0x00100ffa
qemus ebx=0x04053a40, mine ebx=0x04053a40, eip:0x00100ffa
qemus esp=0x000079bc, mine esp=0x000079bc, eip:0x00100ffa
qemus ebp=0x000079c4, mine ebp=0x000079c4, eip:0x00100ffa
qemus esi=0x00101d10, mine esi=0x00101d10, eip:0x00100ffa
qemus edi=0x04053a40, mine edi=0x04053a40, eip:0x00100ffa
qemus eip=0x00100ffa, mine eip=0x00100ffa, eip:0x00100ffa
(nemu) q
qemu-system-i386:make[1]: Leaving directory '/home/pqq/桌面/PA/ics2018/nemu'
terminating on signal 15 from pid 22872
```

The bottom part of the image shows a file manager window titled 'games'. The breadcrumb path is '主文件夹 > 桌面 > PA > ics2018 > navy-apps > fsimg > share > games'. The left sidebar shows '使用的' and 'ne'. The main area shows two folders: 'nes' and 'pal'.

- 12、 修改 main(), 运行/bin/events

```
int main() {
#ifdef HAS_PTE
    init_mm();
#endif

    Log("'Hello World!' from Nanos-lite");
    Log("Build time: %s, %s", __TIME__, __DATE__);

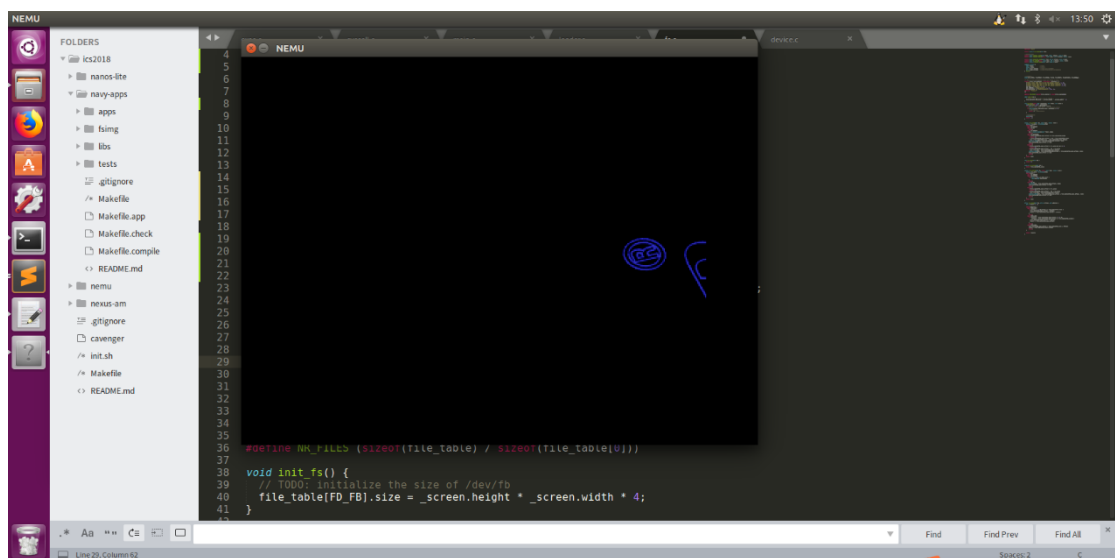
    init_ramdisk();

    init_device();

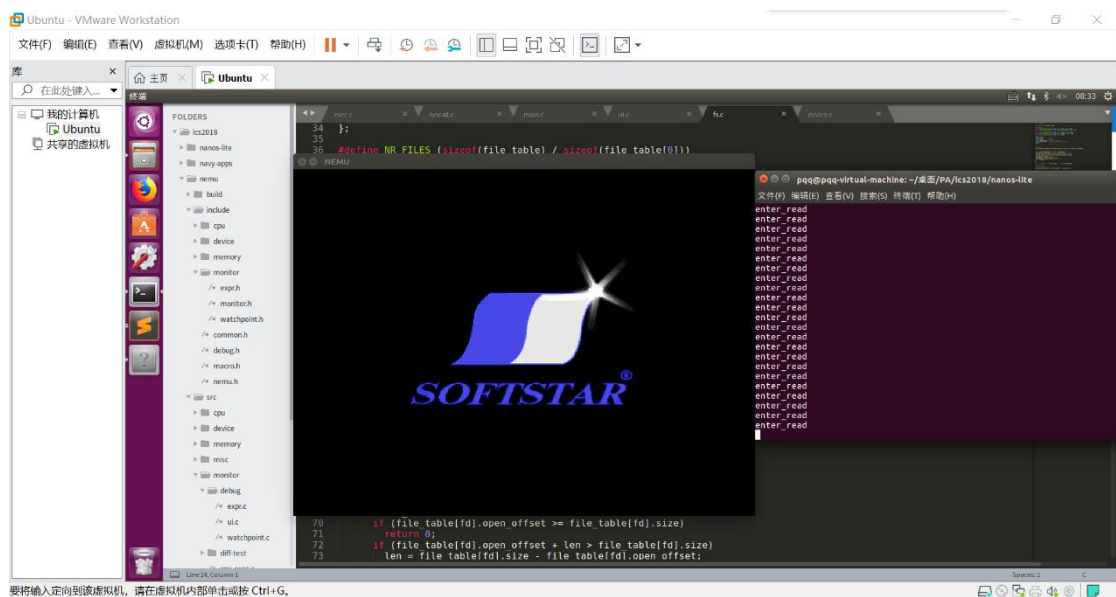
#ifdef HAS_ASYE
    Log("Initializing interrupt/exception handler...");
    init_irq();
#endif
    init_fs();

    //uint32_t entry = loader(NULL, NULL);
    //uint32_t entry = loader(NULL, "/bin/hello");
    //uint32_t entry = loader(NULL, "/bin/text");
    //uint32_t entry = loader(NULL, "/bin/bmtest");
    //uint32_t entry = loader(NULL, "/bin/events");
    uint32_t entry = loader(NULL, "/bin/pal");
    printf("loader end\n");
    ((void (*)(void))entry)();

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



(这个界面加载了 3h+)



(这个界面加载了 12h+)  
(我尽力了)



# 回答问题

1、

## 对比异常与函数调用

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。

函数调用需要保存的状态:用栈保存返回地址、寄存器

异常处理需要保存的状态:EFLAGS, CS, EIP 等

对比:为了返回到离开的地方,函数调用和异常处理都需要保存当前位置 EIP。

原因:异常处理需要陷入内核态,函数调用不需要。

2、

## 诡异的代码

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。

```
trap.S (~/.桌面/PA/ics2018/nexus-am/am/arch/x86-nemu/src) - gedit
打开(O)  图标
#----|-----entry-----|-----errorcode-|---irq id---|---handler---|
.globl vecsys;   vecsys:  pushl $0;   pushl $0x80; jmp asm_trap
.globl vecnull;  vecnull: pushl $0;   pushl $-1;  jmp asm_trap

asm_trap:
    pushal

    pushl %esp
    call irq_handle

    addl $4, %esp

    popal
    addl $8, %esp

    iret
```

代码: trap frame 是在堆栈上构造的,接下来代码将会把当前的%esp 压栈,并调用 C 函数 `irq_handle()`。

原因:将 esp 当前的内容入栈,使其指向陷阱帧,作为 `irq_handle()`的参数传递

## 必答题

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。

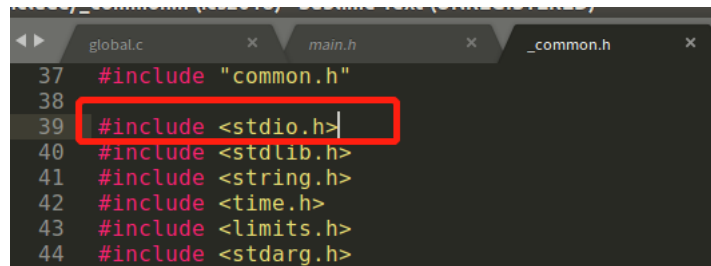
## ● 存档读取

1. Global.c 中 `PAL_LoadGame()`函数的参数是 `LPCSTR szFileName`,在 `common.h` 中显示是个 `const char`,即参数为需要读取的存档名。函数中先是调用了 `fopen()`打开目的存档的文件,再使用 `fread()`函数将存档中的信息读取到 `static SAVEDGAME` 类型的变量 `s` 中,该类型在 `global.h` 中定义(如下图)。然后关闭打开的文件,并对

保存了存档的变量 s 进行相关操作。

```
typedef struct tagSAVEDGAME
{
    WORD          wSavedTimes;           // saved times
    WORD          wViewportX, wViewportY; // viewport location
    WORD          nPartyMember;          // number of members in party
    WORD          wNumScene;             // scene number
    WORD          wPaletteOffset;        // 
    WORD          wPartyDirection;       // party direction
    WORD          wNumMusic;             // music number
    WORD          wNumBattleMusic;       // battle music number
    WORD          wNumBattleField;       // battle field number
    WORD          wScreenWave;           // level of screen waving
    WORD          wBattleSpeed;          // battle speed
    WORD          wCollectValue;         // value of "collected" items
    WORD          wLayer;
    WORD          wChaseRange;
    WORD          wChasespeedChangeCycles;
    WORD          nFollower;
    WORD          rgwReserved2[3];        // unused
    DWORD         dwCash;                // amount of cash
    PARTY         rgParty[MAX_PLAYABLE_PLAYER_ROLES]; // player party
    TRAIL         rgTrail[MAX_PLAYABLE_PLAYER_ROLES]; // player trail
    ALLEXPERIENCE Exp;                   // experience data
    PLAYERROLES   PlayerRoles;
    POISONSTATUS  rgPoisonStatus[MAX_POISONS][MAX_PLAYABLE_PLAYER_ROLES]; // poison status
    INVENTORY     rgInventory[MAX_INVENTORY]; // inventory status
    SCENE         rgScene[MAX_SCENES];
    OBJECT        rgObject[MAX_OBJECTS];
    EVENTOBJECT   rgEventObject[MAX_EVENT_OBJECTS];
} SAVEDGAME, *LPSAVEDGAME;
```

2. Global.c 包含了头文件 main.h，打开 main.h 看的它包含了\_common.h，而\_common.h 中又包含了 C 标准库 stdio.h，fread()函数出自该标准库。



```
37 #include "common.h"
38
39 #include <stdio.h>
40 #include <stdlib.h>
41 #include <string.h>
42 #include <time.h>
43 #include <limits.h>
44 #include <stdarg.h>
```

fread 函数属于标准 IO（带缓冲的）。标准 IO 提供缓冲的目的是尽可能减少使用 read 和 write 调用的次数（也就是用户态和内核态切换的次数），也就是在调用 fwrite 的时候，先把数据放在缓冲区。不直接使用系统调用，切换到内核态。而是等到缓冲区满等条件满足时，再一次性调用，把数据拷贝到内核空间。

## ● 屏幕更新

- 1、在 redraw()函数中先用 palette 给 fb 对应元素赋值。Palette 是 256 色调色板，fb 和 vmem 都是 size 为 W\*H 的数组。将 fb 作为第一个参数传入 NDL\_DrawRect 中。

```
static void redraw() {
    for (int i = 0; i < W; i++)
        for (int j = 0; j < H; j++)
            fb[i + j * W] = palette[vmem[i + j * W]];

    NDL_DrawRect(fb, 0, 0, W, H);
    NDL_Render();
}
```

- 2、在 ndl.c 中实现了 NDL\_DrawRect 函数。这个 c 文件包含了<stdio.h>头文件，函数中的 printf、putchar 和 fwrite 都出自该头文件。在调用他们时都会像上文描述的 fread 一样陷入操作系统内核态然后进行一系列相关操作。

```

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;Y%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];
            }
        }
    }
}

```

## ● 解释

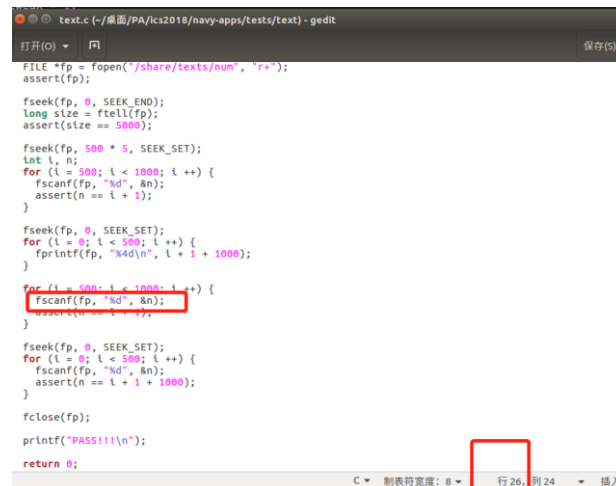
仙剑属于应用程序，在 PA 实验中，它直接运行在操作系统上（我们把 pal 放在了 Navy-apps 中而不是 nexus-am 中）。在调用库函数 `fread`、`fwrite` 等时，标准 IO 操作会陷入 OS 内核进行操作（在普通计算机上，大量的存档资料应当存储于硬盘而不是内存中，故需要陷入而不只是读写缓存），也就是 PA 中的轻量级操作系统 `Nanos-lite` 中进行文件读写操作，而 `libos` 中的 `Nanos.c` 中提供了 PA3 中系统调用的接口（`_sys_call_` 函数），在这里根据不同的系统调用号进行不同的中断操作。在 `fwrite` 时，内核操作中会调用 `AM` 的 IOE 接口进行屏幕信息更新，`NEMU` 提供硬件支持。

# 遇到的坑

- 1、实现 PA3.2 的时候修改完只进行 make update 和 make run, 没有去 hello.c 目录里 make (毕竟手册里说 make 是编译, 我还以为只需要编译一次呜呜呜), 导致一直没法更新 sbrk 的实现, 打印的时候总是直接进入 SYS\_write 逐个字母打印, 这个智障问题耽搁了一整周进度。
- 2、完成 PA3.2 后, 为了写实验报告又把设置都调回去, 把 SYS\_write 的 fs\_write 改回 sys\_write, 然后忘记改回来了! 运行 /bin/text 的时候疯狂报错, 于是循着报错去找 text.c 文件第 26 行。

```
size = 2550314 bytes
[src/main.c,27,main] Initializing interrupt/exception ha
fd = 14
loader end
Assertion failed: n == i + 1, file text.c, line 26
nemu: HIT BAD TRAP at eip = 0x00100032
(nemu) q
```

好吧 26 行显示是一个 fscanf



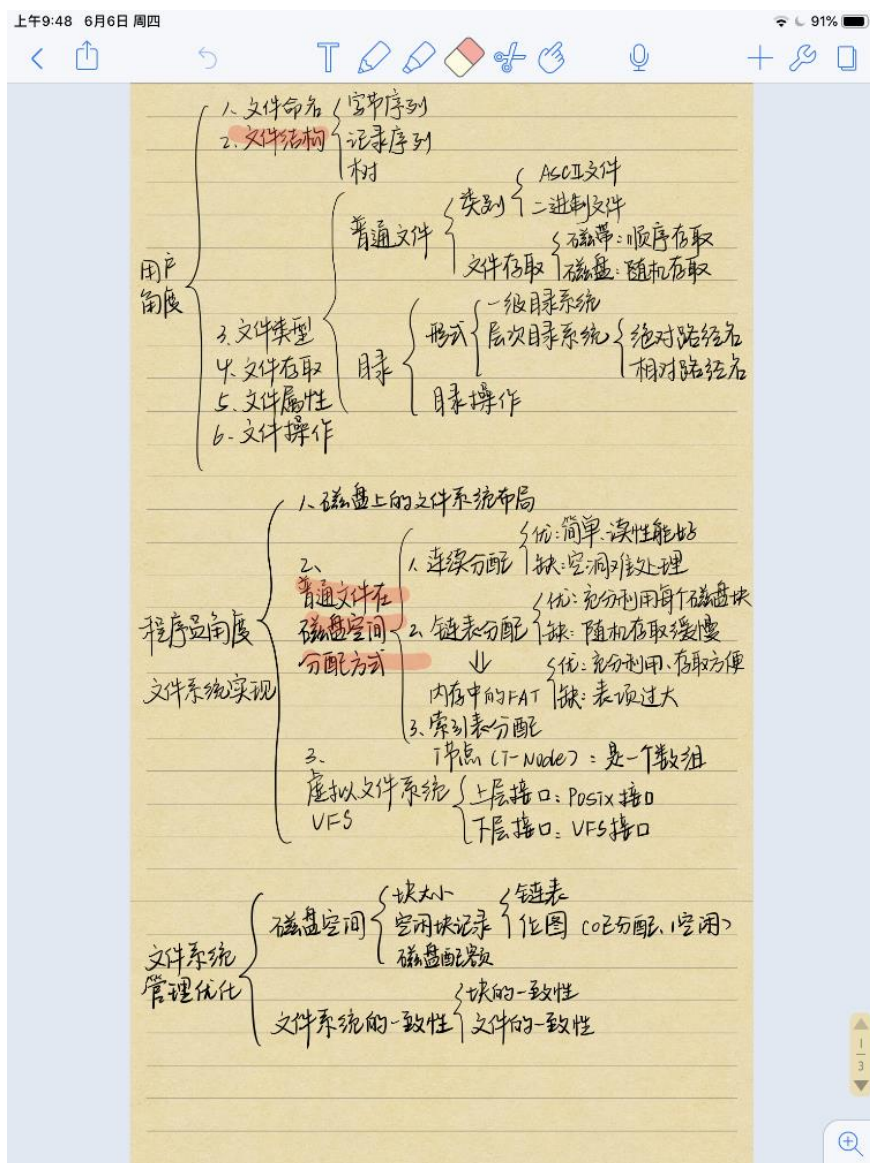
于是找跟 read (输入) 相关的代码 (?), 查到眼睛。而且虚拟机上的虚拟机上的程序运行很慢, 五分钟才出一次结果, 终于找到了这个地方, 大概是因为对于用户程序而言的“读”是计算机的“输出”吧。现在是凌晨 00:40, 深夜使人脑子不清醒, 为了 PA 本人发际线疯狂往后退, 微笑.jpg 😊

0:40  
2019/6/4

- 3、以为 /bin/bmptest 的输出在 terminal 上, 然后 2 天每天为 bmptest 傻等一小时……结果鼠标误触 NEMU……
- 4、实现 text 的时候在 Main 里写成 /bin/test, 于是 open 失败, 去 open 里面挨个儿打印文件名查 (……)
- 5、/bin/text 运行得巨慢, 一度以为是自己实现有错, 然后查了半天啥也没查出来。放他自己在那儿跑了 2 个小时之后报错! fine, 漏了 35 号指令, 回去补完再等 2 小时。终于看见了正常输出。
- 6、到最后一步后来跑仙剑的时候, 本人已经佛了, PA 最大收获之使我变佛。

# 复习的知识和学到的经验

- 1、 开始之前复习了 OS 课中的文件管理相关知识，然后发现没啥关系。



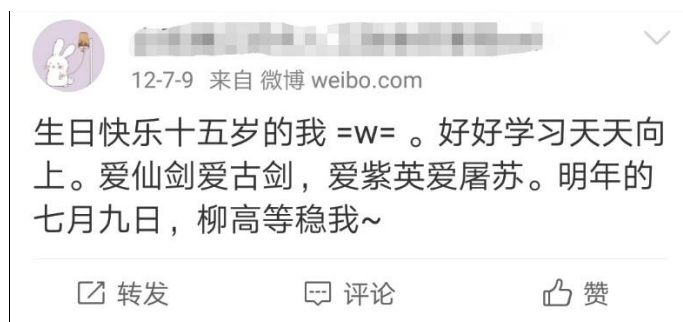
- 2、 重新学习了 OS 中文件系统的读写相关知识，以及为什么要区分内核态和用户态，操作系统的各制度等级
- 3、 对“抽象”、“接口”等概念有了更深入的理解
- 4、 复习了中断的时候保存现场到处理中断到继续运行的过程。想起腾讯面试的时候被总监问了一个问题：简述调用库函数和系统调用的区别，当时答得模模糊糊，被 diss 了一波，现在做完 PA 实验（大概）可以回答上来了（吧）。
- 5、 作为一名

真情实感 · 出钱出感情 · 深爱十几年 · 真 · 仙剑系列铁粉

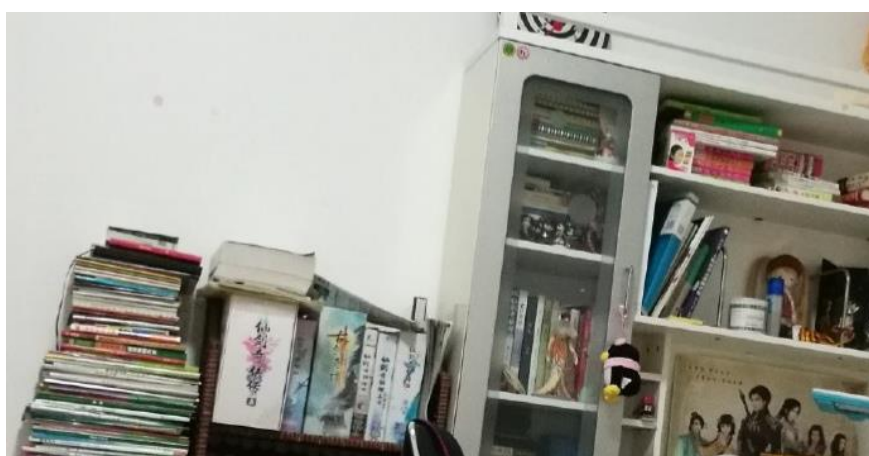
没想到能有机会被我仙以这种方式虐得死去活来  
菜到流下了没有技术含量的泪水



实验报告最后一条  
献给陪伴我成长十几年的情怀吧。



(12 年 15 岁表白仙剑的微博纪实)



(全套正版)



(手办小说)