

# 2012543-刘沿辰-PA3报告

学号：2012543

姓名：刘沿辰

专业：计算机科学与技术

指令集：RISC-V 32

日期：2023.5.30

## 实验目的

- 了解系统调用操作原理，实现异常响应机制
- 实现loader以及相关函数，运行程序
- 实现丰富的库函数，运行批处理系统下的各个程序

## 实验内容

- 第一阶段：实现自陷操作 `yield()`
- 第二阶段：实现用户程序加载与系统调用，支持TRM的运行
- 第三阶段：运行仙剑奇侠传并展示批处理系统，提交完整报告

## 实验过程

### 第一阶段

#### 实现异常响应机制

想要实现异常响应机制，存储程序状态的寄存器必不可少。此处将存储相关内容的寄存器扩展称为系统寄存器System Register，并将其添加到寄存器结构中，具体实现如下：

```

typedef struct {
    // CSR registers
    vaddr_t mepc;
    word_t mstatus;
    word_t mcause;

    // Other registers
    vaddr_t mtvec;
} riscv32_System_Registers;

typedef struct {
    word_t gpr[32];
    vaddr_t pc;
    riscv32_System_Registers sr;
} riscv32_CPU_state;

```

此外，想要实现这个机制，我们还需要实现3条相关的指令，实现如下：

```

INSTRPAT("??????? ???? ???? 010 ????? 11100 11", csrrs , I,
t = *CSR(imm); *CSR(imm) = t | src1; gpr(dest) = t);

INSTRPAT("??????? ???? ???? 001 ????? 11100 11", csrrw , I,
t = *CSR(imm); *CSR(imm) = src1; gpr(dest) = t);

INSTRPAT("0000000 00000 00000 000 00000 11100 11", ecall , I,
s->dnpc = isa_raise_intr(11, s->pc));

```

其中 **ecall** 指令调用的 **isa\_raise\_intr** 函数定义如下：

```

word_t isa_raise_intr(word_t NO, vaddr_t epc) {
    cpu.sr.mcause = NO;
    cpu.sr.mepc = epc;
    return cpu.sr.mtvec;
}

```

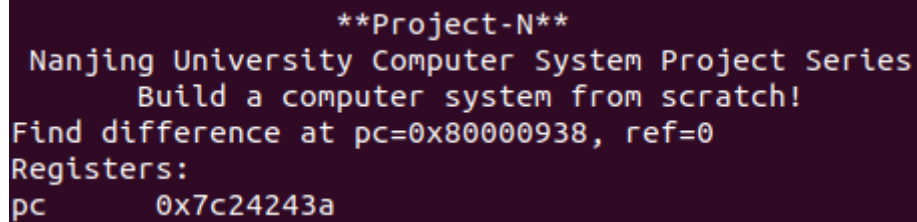
值得注意的是，此处为了方便，我提前定义了宏CSR来方便我们找到系统寄存器，如下：

```
word_t *get_csr(word_t imm) {
    switch (imm) {
        case 0x305: return &cpu.sr.mtvec;
        case 0x300: return &cpu.sr.mstatus;
        case 0x341: return &cpu.sr.mepc;
        case 0x342: return &cpu.sr.mcause;
        default: Log("Invalid csr code!"); assert(0);
    }
}

#define CSR(i) get_csr(i)
```

## 愚蠢的错误

然而，程序实际运行时出现了如下错误：



```

**Project-N**
Nanjing University Computer System Project Series
Build a computer system from scratch!
Find difference at pc=0x80000938, ref=0
Registers:
pc      0x7c24243a
```

PC跑到了界外？？？

这个bug花费了我一整天的时间来寻找，我只能说穷极抽象。最后成功找到了bug：错误的原因是printf函数实现时的缓冲区不足，将缓冲区从1024改为4096后解决问题。这个问题完全无法溯源，能找到只能说是运气太好。。。。我闲极无聊试图估算南京大学Project-N的大小，大约是 $50 * 50 = 2500$ 左右，突然意识到1024的缓冲区是否不够？这才通过修改缓冲区大小成功解决了问题。

于是我在互联网上查找了C语言库中printf缓冲区的大小，看到一位博主做了[关于标准printf缓冲区大小的测试实验](#)，才知道标准库中的printf缓冲区就是4096。某种意义上，这个bug是我自己不仔细RTFM而造成的，认栽。

## 重新组织Context结构体

观察Trap代码如下：

80000558:	f7010113	addi	sp,sp,-144
8000055c:	00112223	sw	ra,4(sp)
80000560:	00312623	sw	gp,12(sp)
80000564:	00412823	sw	tp,16(sp)
80000568:	00512a23	sw	t0,20(sp)
8000056c:	00612c23	sw	t1,24(sp)
80000570:	00712e23	sw	t2,28(sp)
80000574:	02812023	sw	s0,32(sp)
80000578:	02912223	sw	s1,36(sp)
8000057c:	02a12423	sw	a0,40(sp)
80000580:	02b12623	sw	a1,44(sp)
80000584:	02c12823	sw	a2,48(sp)
80000588:	02d12a23	sw	a3,52(sp)
8000058c:	02e12c23	sw	a4,56(sp)
80000590:	02f12e23	sw	a5,60(sp)
80000594:	05012023	sw	a6,64(sp)
80000598:	05112223	sw	a7,68(sp)
8000059c:	05212423	sw	s2,72(sp)
800005a0:	05312623	sw	s3,76(sp)
800005a4:	05412823	sw	s4,80(sp)
800005a8:	05512a23	sw	s5,84(sp)
800005ac:	05612c23	sw	s6,88(sp)
800005b0:	05712e23	sw	s7,92(sp)
800005b4:	07812023	sw	s8,96(sp)
800005b8:	07912223	sw	s9,100(sp)
800005bc:	07a12423	sw	s10,104(sp)
800005c0:	07b12623	sw	s11,108(sp)
800005c4:	07c12823	sw	t3,112(sp)
800005c8:	07d12a23	sw	t4,116(sp)
800005cc:	07e12c23	sw	t5,120(sp)
800005d0:	07f12e23	sw	t6,124(sp)
800005d4:	342022f3	csrr	t0,mcause
800005d8:	30002373	csrr	t1,mstatus
800005dc:	341023f3	csrr	t2,mepc
800005e0:	08512023	sw	t0,128(sp)

800005e4:	08612223	sw	t1,132(sp)
800005e8:	08712423	sw	t2,136(sp)

因此我应该将Context结构初始化为：

```
uintptr_t gpr[32], mcause, mstatus, mepc;  
void *pdir; // 不要漏了
```

## 理解上下文结构体

在文件 `trap.S` 中，汇编语言将寄存器压栈之后（压栈顺序见上文），使用jal指令调用函数 `__am_irq_handle()`。此时的结构体Context作为一个函数调用的参数，实际上是被压在了程序的栈中，并保存有函数调用时的上下文环境。有意思的来了，切换过程在此处和函数调用体现出了高度的一致性，甚至此处的实现就是用函数调用的形式实现的。

发表一个暴论：上下文切换可以当作一个翻版的函数调用来理解，就是参数比较多

此时我产生了一个问题，这么多寄存器压栈那不会分分钟给栈干爆？噢不过上下文切换这个行为并不会经常发生，更不会在栈上递归般地多次发生，所以问题不大

## 事件分发与上下文恢复

想实现事件分发，首先我们要知道自陷异常的处理方式。使用指令 `man 2 syscall` 查阅可知，我们的yield语句使用寄存器a7存储异常号码

Arch/ABI	Instruction	System call #	Ret val	Ret val2	Error	Notes
alpha	callsys	v0	v0	a4	a3	1, 6
arc	trap0	r8	r0	-	-	
arm/OABI	swi NR	-	r0	-	-	2
arm/EABI	swi 0x0	r7	r0	r1	-	
arm64	svc #0	w8	x0	x1	-	
blackfin	excpt 0x0	P0	R0	-	-	
i386	int \$0x80	eax	eax	edx	-	
ia64	break 0x100000	r15	r8	r9	r10	1, 6
m68k	trap #0	d0	d0	-	-	
microblaze	brki r14,8	r12	r3	-	-	
mips	syscall	v0	v0	v1	a3	1, 6
nios2	trap	r2	r2	-	r7	
parisc	ble 0x100(%sr2, %r0)	r20	r28	-	-	
powerpc	sc	r0	r3	-	r0	1
powerpc64	sc	r0	r3	-	cr0.SO	1
riscv	ecall	a7	a0	a1	-	

观察yield函数的内联汇编语句，也能验证上述结论：

```
void yield() {
    asm volatile("li a7, -1; ecall");
}
```

接着在\_\_am\_irq\_handle函数内部添加处理函数，然后在Nanos的irq.c中输出一个语句即可。

```
Context* __am_irq_handle(Context *c) {
    if (user_handler) {
        Event ev = {0};
        switch (c->mcause) {
            case 11: c->mepc += 4; ev.event = EVENT_YIELD; break;
            default: ev.event = EVENT_ERROR; break;
        }

        c = user_handler(ev, c);
        assert(c != NULL);
    }

    return c;
}
```

此处需要补充另一条汇编指令mret，用于恢复由于程序打断而切换的上下文。mret指令实现如下：

```
INSTPAT("0011000 00010 00000 000 00000 11100 11", mret    , N,
s->dnpc = cpu.sr.mepc);
```

由此，程序便可以顺利执行到末尾的panic部分。

## 从Nanos调用yield()开始，发生了什么？

首先，在Nanos的任何定义中都找不到yield()这个函数，说明这是运行时环境（am）的一部分，直接在abstract-machine的cte.c中定义。此处的yield()函数的行为是直接执行内联汇编语句asm volatile("li a7, -1; ecall")，将事件编号0xFFFFFFFF传入寄存器a7后，使用ecall指令完成自陷。

ecall指令的实现见上文isa\_raise\_intr函数，它干的事情就是将事件编号放到mcause寄存器中，把当前pc存储到mepc寄存器中，最后返回一个处理函数的入口地址。这个处理函数就是trap.S，而它的地址早在cte\_init函数中就被放入了mtvec寄存器中。

再然后，程序将会执行trap.S函数，将所有上下文压栈存储，然后跳转到\_\_am\_irq\_handle函数。

在 `__am_irq_handle` 函数中，程序将会根据事件编号进行相应的处理，然后调用 `user_handler` 函数对上下文进行恢复。那么这个恢复函数又是什么呢？答案就是Nanos自己编写的 `do_event` 函数，也就是说硬件在处理完事件后，会直接将上下文丢给操作系统进行处理。

我在实现这个地方的时候偷懒了，把pc加4这个操作放在了硬件层面实现，但这实际上是符合CISC哲学的做法，而非RISC的哲学。

最后Nanos返回了上下文（我在这里没有修改上下文的内容），然后把上下文恢复，调用 `mret` 语句回到原来的地方继续执行就可以了。

经过一段时间的纠结，我还是把这个地方改了，将pc加4这个操作放到了操作系统中。一方面，这更符合RISC设计的初衷：简化硬件设计，加速硬件单指令执行；另一方面也有助于我在操作系统层面进行调节时采取不同的措施。修改后的代码如下：

```
// in irq.c
static Context* do_event(Event e, Context* c) {
    switch (e.event) {
        case 1: Log("Yield handle successfully!"); c->mepc += 4; break;
        default: panic("Unhandled event ID = %d", e.event);
    }

    return c;
}
```

PA3阶段1到此结束

---

## 第二阶段

### 实现loader

loader的简易实现如下：



```

Elf_Ehdr ehdr;
ramdisk_read(&ehdr, 0, sizeof(Elf_Ehdr));
assert((* (uint32_t *)ehdr.e_ident == 0x464c457f));

Elf_Phdr phdr[ehdr.e_phnum];
ramdisk_read(phdr, ehdr.e_phoff,
sizeof(Elf_Phdr)*ehdr.e_phnum);
for (int i = 0; i < ehdr.e_phnum; i++) {
    if (phdr[i].p_type == PT_LOAD) {
        ramdisk_read((void*)phdr[i].p_vaddr,
phdr[i].p_offset, phdr[i].p_memsz);
        memset((void*)
(phdr[i].p_vaddr+phdr[i].p_filesz), 0, phdr[i].p_memsz -
phdr[i].p_filesz);
    }
}
}

```

只需要检查elf文件的magic\_number即可

## 识别并实现系统调用

首先定义好上下文的相关宏：

```

#define GPR1 gpr[17] // a7
#define GPR2 gpr[10]
#define GPR3 gpr[11]
#define GPR4 gpr[12]
#define GPRx gpr[10]

```

由于上文实现的yield未识别调用，所以还需要修改\_\_am\_irq\_handle函数如下：

```

Context* __am_irq_handle(Context *c) {
    if (user_handler) {
        Event ev = {0};
        switch (c->mcause) {
            case 11:
                switch(c->GPR1){
                    case -1:ev.event=EVENT_YIELD ;break;
                    default:ev.event=EVENT_SYSCALL;break;
                }
            default: ev.event = EVENT_ERROR; break;
        }
        c = user_handler(ev, c);
        assert(c != NULL);
    }
    return c;
}

```

最后在 **syscall.c** 文件中补全相关内容即可：

```

void do_syscall(Context *c) {
    uintptr_t a[4];
    a[0] = c->GPR1;
    a[1] = c->GPR2;
    a[2] = c->GPR3;
    a[3] = c->GPR4;

    switch (a[0]) {
        case SYS_yield: yield(); break;
        case SYS_exit: Log("sys_call:exit"); halt(0); break;
        default: panic("Unhandled syscall ID = %d", a[0]);
    }
}

```

此时，dummy程序运行成功！

# Hello World!

只要在 `syscall` 中不断地 `putch` 就可以了呢!

```
for (int i = 0; i < c->GPR4; ++i){
    putch(*(((char *)c->GPR3) + i));
}
```

## 堆区管理

首先在navy中修改\_sbrk函数为:

```
extern char _end;
static void* program_end = NULL;

void *_sbrk(intptr_t increment) {
    if (program_end == NULL) {
        program_end = &_amp;_end;
    }
    void *res = program_end;

    int ret = _syscall_(SYS_brk, (intptr_t)program_end +
increment, 0, 0)
    if (ret == 0)
        program_end = program_end + increment;
    else
        return (void *)-1;
    return res;
}
```

然后在操作系统中永远返回0（即永远成功）即可。

```
[/home/yyy/Documents/ics2022/nanos-lite/src/loader.c,51,naive_uload] Jump to entry = C6D40038
Hello World!
Hello World from Navy-apps for the 2th time!
Hello World from Navy-apps for the 3th time!
Hello World from Navy-apps for the 4th time!
Hello World from Navy-apps for the 5th time!
Hello World from Navy-apps for the 6th time!
Hello World from Navy-apps for the 7th time!
Hello World from Navy-apps for the 8th time!
Hello World from Navy-apps for the 9th time!
Hello World from Navy-apps for the 10th time!
Hello World from Navy-apps for the 11th time!
Hello World from Navy-apps for the 12th time!
Hello World from Navy-apps for the 13th time!
Hello World from Navy-apps for the 14th time!
Hello World from Navy-apps for the 15th time!
Hello World from Navy-apps for the 16th time!
Hello World from Navy-apps for the 17th time!
Hello World from Navy-apps for the 18th time!
Hello World from Navy-apps for the 19th time!
Hello World from Navy-apps for the 20th time!
```

可以看到程序不断hello，执行成功。

## hello程序从何而来，到哪里去？

Hello程序本质上只是一个可以在计算机上执行的C代码。由于我的操作系统现在还非常简陋，只支持执行一个程序，所以这个hello程序会被我直接放到操作系统的镜像中，作为第一个（也是唯一一个）程序存在于入口处。

当我们开始执行Nanos时，操作系统首先会从ELF文件标准结构中获得程序入口地址，然后通过上下文切换来执行对应程序。这样hello程序就可以正常的执行了。（这也启示了我，实际上无论是操作系统还是简单的程序，从CPU的角度来看都只是一条条指令而已，**计算机时刻不停的计算这个属性永远不会改变！**）

而程序自身在执行时，由于需要使用SY\_write等操作的支持，程序本身又会通过上下文切换来进行系统调用，此时操作系统便会识别到这个系统调用，将调用交给 `do_syscall` 函数处理，并成功在硬件层面输出内容。

以上一切的一切都是以上下文为媒介进行的切换，也就是说：CPU本身只知道计算，不管前方是什么，它都是一台不断计算的机器；而上下文作为进程切换的媒介，帮助计算机在需要的时刻切换CPU的状态，是系统与程序的交互媒介。

这样看来，多线程似乎也没有那么困难了，毕竟CPU的任务只有计算，而只要通过合理的上下文切换，我们就可以保存任何程序的执行状态，也可以切换到任何程序来执行。

*PA3阶段2到此结束*

---

## 第三阶段

刚刚说到多程序的事情，目前我们的loader依然是在手动load，接下来就要使用新的接口来进行自动操作！

### 实现五个读写相关函数

首先实现五个读写相关函数如下：

```

int fs_open(const char *path){
    if(strcmp(path, file_table[FD_EVENT].name) == 0) return
    FD_EVENT;
    if(strcmp(path, file_table[FD_DISPINFO].name) == 0) return
    FD_DISPINFO;
    if(strcmp(path, file_table[FD_FB].name) == 0) return FD_FB;
        for (int i = FD_FB + 1; i < FD_SIZE; i++) {
            if (strcmp(path, file_table[i].name) == 0) {
                file_table[i].open_offset = 0;
                return i;
            }
        }
    return -1;
}

```

```

size_t fs_read(int fd,void *buf,size_t len){
    size_t actual_len;
    if (file_table[fd].open_offset + len > file_table[fd].size)
    {
        actual_len = file_table[fd].size -
file_table[fd].open_offset;
    }
    else {
        actual_len = len;
    }

    if (file_table[fd].read) {
        file_table[fd].read(buf, file_table[fd].open_offset,
actual_len);
    }
    else {
        ramdisk_read(buf, file_table[fd].disk_offset +
file_table[fd].open_offset, actual_len);
    }
    file_table[fd].open_offset += actual_len;
    return actual_len;
}

```

```

}

size_t fs_lseek(int fd, size_t offset, int whence){
    switch(whence) {
        case SEEK_SET: file_table[fd].open_offset = offset;
        break;
        case SEEK_CUR: file_table[fd].open_offset += offset;
        break;
        case SEEK_END: file_table[fd].open_offset =
file_table[fd].size + offset; break;
        default: assert(0);
    }

    return file_table[fd].open_offset;
}

size_t fs_write(int fd,const void *buf,size_t len){
    size_t actual_len;
    if (file_table[fd].open_offset + len > file_table[fd].size)
    {
        panic("Wrong writing!");
        assert(0);
    }
    else {
        actual_len = len;
    }

    if (file_table[fd].write) {
        file_table[fd].write(buf, file_table[fd].open_offset,
actual_len);
    }
    else {
        ramdisk_write(buf, file_table[fd].disk_offset +
file_table[fd].open_offset, actual_len);
    }
    file_table[fd].open_offset += actual_len;
    return actual_len;
}

```

```

}

int fs_close(int fd){
    file_table[fd].open_offset=0;
    return 0;
}

```

然后据此修改loader函数如下：

```

static uintptr_t loader(PCB *pcb, const char *filename) {
    Elf32_Ehdr header;
    int fd = fs_open(filename);
    fs_read(fd, &header, sizeof(Elf32_Ehdr));
    assert(*(uint32_t*)header.e_ident==0x464c457f);

    Elf_Phdr Phdr;
    for (int i = 0; i < header.e_phnum; i++) {
        fs_lseek(fd, header.e_phoff + i*header.e_phentsize,
        SEEK_SET);
        fs_read(fd, &Phdr, sizeof(Phdr));
        if (Phdr.p_type == PT_LOAD) {
            fs_lseek(fd, Phdr.p_offset, SEEK_SET);
            fs_read(fd, (void*)Phdr.p_vaddr, Phdr.p_filesz);
            for(unsigned int i = Phdr.p_filesz; i <
            Phdr.p_memsz;i++){
                ((char*)Phdr.p_vaddr)[i] = 0;
            }
        }
    }
    return header.e_entry;
}

```

完善syscall之后运行程序，通过file测试



```
[/home/yyy/Documents/ics2022/nanos-lite/src/proc.c,27,init_proc] Initializing processes...  
[/home/yyy/Documents/ics2022/nanos-lite/src/loader.c,50,naive_upload] Jump to entry = 02D70038  
PASS!!!  
[src/cpu/cpu-exec.c:143 cpu_exec] nemu: HIT GOOD TRAP at pc = 0x80000af4  
[src/cpu/cpu-exec.c:103 statistic] host time spent = 13,689,230 us  
[src/cpu/cpu-exec.c:104 statistic] total guest instructions = 1,126,533  
[src/cpu/cpu-exec.c:105 statistic] simulation frequency = 82,293 inst/s
```

## 串口抽象

串口的实现只需要在device中putch就可以了，如下：

```
size_t serial_write(const void *buf, size_t offset, size_t len) {  
    for(int i = 0; i < len; i++){  
        putch(((char*)buf)[i]);  
    }  
    return len;  
}
```

## 时间与NDL时钟

实现时钟的代码很简单：

```
int sys_gettimeofday(struct timeval *value){  
    value->tv_usec = (io_read(AM_TIMER_UPTIME).us % 1000000);  
    value->tv_sec = (io_read(AM_TIMER_UPTIME).us / 1000000);  
    return 0;  
}
```

然后用上下文的GPR2作为参数调用函数即可。同时框架还提供了NDL库，我们在NDL库中直接调用 **sys\_gettimeofday** 函数即可：

```
uint32_t NDL_GetTicks() {
    struct timeval tv;
    struct timezone tz;
    gettimeofday(&tv, &tz);
    return tv.tv_sec * 1000 + tv.tv_usec / 1000;
}
```

测试通过：

```
[/home/yyy/Documents/ics2022/nanos-lite/src/loader.c,38,naive_upload] Jump to entry = CFD40038
now time: 2.500s
now time: 3.0s
now time: 3.500s
now time: 4.0s
now time: 4.500s
now time: 5.0s
now time: 5.500s
now time: 6.0s
now time: 6.500s
now time: 7.0s
now time: 7.500s
now time: 8.0s
```

## 按键抽象

首先我们需要实现 `events_read` 函数：

```
size_t events_read(void *buf, size_t offset, size_t len) {
    size_t actual_len = 1;
    AM_INPUT_KEYBRD_T ev = io_read(AM_INPUT_KEYBRD);
    if (ev.keycode == AM_KEY_NONE) {
        return 0;
    }
    actual_len = sprintf(buf, "%s %s\n", ev.keydown ? "kd":
"ku", keyname[ev.keycode]);
    return actual_len;
}
```

在 `fs.c` 的表中添加一行按键支持，最后用NDL封装：

```
int NDL_PollEvent(char *buf, int len) {  
    int fd = open("/dev/events", 0, 0);  
    return read(fd, buf, len);  
}
```

运行成功：

```
receive event: kd A  
receive event: ku A  
receive event: kd LSHIFT  
receive event: ku LSHIFT  
receive event: kd F  
receive event: ku F
```

## 遇到的问题

在完成这一部分时，最开始我的终端一直不输出信息，后来查bug得知此时读的长度为0（但是使用的是printf，所以能正常输出），导致返回值为0，于是就一直没有输出信息，哪怕信息已经被存到缓冲区中。

## VGA显存

首先实现特殊的读写函数：

```

size_t dispinfo_read(void *buf, size_t offset, size_t len) {
    int actual_len = snprintf((char*)buf, len,
        "WIDTH:%d\nHEIGHT: %d\n",
        io_read(AM_GPU_CONFIG).width,
        io_read(AM_GPU_CONFIG).height);
    Log("display infomation: %s", (char*)buf);
    return actual_len;
}

size_t fb_write(const void *buf, size_t offset, size_t len) {
    int x, y, w, h, actual_len;
    w = io_read(AM_GPU_CONFIG).width;
    h = io_read(AM_GPU_CONFIG).height;
    x = (offset / 4) % w;
    y = (offset / 4) / w;
    actual_len = offset + len > w * h * 4 ? w * h * 4 - offset
: len;
    io_write(AM_GPU_FBDRAW, x, y, (uint32_t*)buf,
        actual_len / 4, 1, true);
    return actual_len;
}

```

然后实现NDL中的两个函数：

```

void NDL_OpenCanvas(int *w, int *h) {
    if (getenv("NWM_APP")) {
        .....
    }
    else{
        int fp = open("/proc/dispinfo", 0, 0);
        if(*w == 0 && *h == 0){
            char buf[64];
            read(fp, buf, 64);
            sscanf(buf, "screen width: %d, height: %d\n",
&canvas_w, &canvas_h);
            *w = canvas_w;
            *h = canvas_h;
        }
        else{
            canvas_w = *w;
            canvas_h = *h;
        }
        printf("w=%d\th=%d\n", canvas_w, canvas_h);
    }
}

void NDL_DrawRect(uint32_t *pixels, int x, int y, int w, int
h) {
    int fp = open("/dev/fb", 2, 0);
    for (int i = 0; i < h; i++) {
        lseek(fp, ((y + i) * screen_w + x) * 4, SEEK_SET);
        write(fp, (pixels + i * w), w * 4);
    }
}

// 还需要初始化屏幕大小
int NDL_Init(uint32_t flags) {
    if (getenv("NWM_APP")) {
        evtdev = 3;
    }
}

```

```
int fp = open("/proc/dispinfo", 0, 0);
char buf[64];
read(fp, buf, 64);
sscanf(buf, "screen width: %d, height: %d\n", &screen_w,
&screen_h);

return 0;
}
```

运行测试程序，显示成功：



## 实现更多的fixedptc API

加减乘除和绝对值比较简单，此处主要展示 `fixedpt_floor` 与 `fixedpt_ceil` 的实现。

```

static inline fixedpt fixedpt_floor(fixedpt A) {
    if ((int)A == 0x7fffffff || (int)A == 0xffffffff ||
(int)A == 0) return A;
    else if ((int)A < 0) return A | 0xff;
    else if ((int)A > 0) return A & 0xffffffff00;
    assert(0);
}

static inline fixedpt fixedpt_ceil(fixedpt A) {
    if ((int)A == 0x7fffffff || (int)A == 0xffffffff ||
(int)A == 0) return A;
    if ((int)A < 0) return (fixedpt)(-((- (int)A) &
0xffffffff00));
    if ((int)A > 0) return (fixedpt)(-((- (int)A) |
0xff));
    assert(0);
}

```

## 运行丰富多彩的程序

由于库函数过多，此处不再赘述，可查看 [libminiSDL](#) 内部。以下为各程序运行展示：

[menu](#) 程序：

```
[0] NJU Terminal
[1] NSlider
[2] FCEUX (Super Mario Bros)
[3] FCEUX (100 in 1)
[4] Flappy Bird
[5] PAL - Xian Jian Qi Xia Zhuan
[6] NPlayer
[7] coremark
[8] dhrystone
[9] typing-game
```

```
page = 0, #total apps = 11
help:
<- Prev Page
-> Next Page
0-9 Choose
```



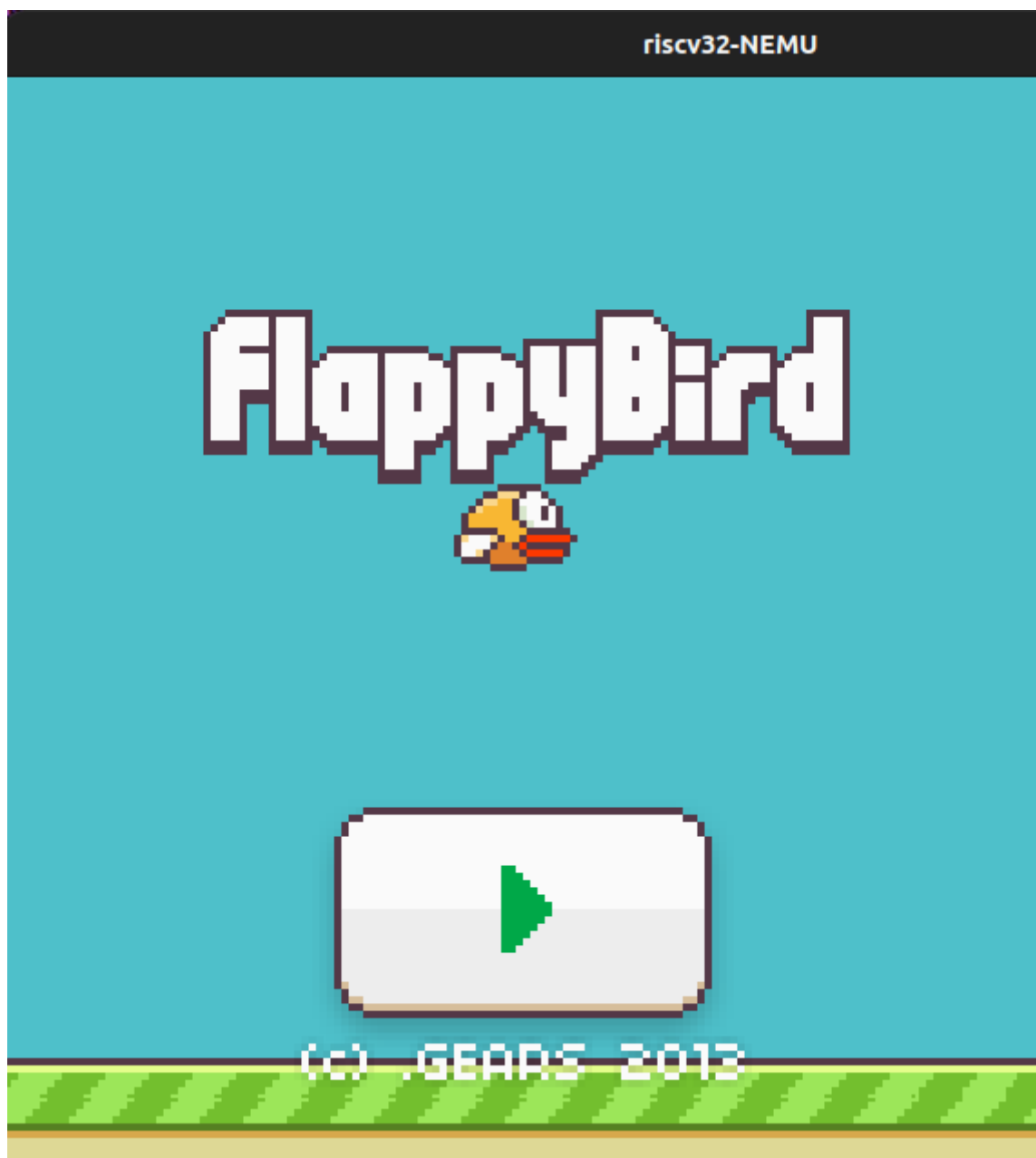
nterm程序:

Built-in Shell in NTerm (NJU Terminal)

sh> █



Flappy Bird程序：



仙剑奇侠传程序：

```
ln -sf ../data ./build/  
./build/sdlpal  
make: *** [Makefile:49: run] Segmentation fault (core dumped)
```

好耶！是好久不见的段错误！

根据编译原理带来的经验，这大概率会是一个空指针之类的错误。经过检查，是出现了一处io错误，需要手动编写 `sdlpal.cfg` 并一起放到 `./data` 文件夹中。这个 `sdlpal.cfg` 需要包括如下内容：

```
GamePath = ./data
OPLSampleRate
SampleRate
WindowHeight
WindowWidth
```

这样真机就可以直接make run了，如下：



但是在我们自己的操作系统中运行会出现问题，发现 **fbp.mkf** 文件打开失败！经过对仙剑奇侠传游戏的RTFSC，我发现这个文件是第一个被open的文件，于是首先怀疑自己的open实现有问题。

检查了一万遍open后，找到了问题所在：

```
[/home/yyy/Documents/ics2022/nanos-lite/src/fs.c,56,fs_open] Open file: /share/games/pal/sdlpal.cfg
[/home/yyy/Documents/ics2022/nanos-lite/src/fs.c,56,fs_open] Open file: ./data/fbp.mkf
[/home/yyy/Documents/ics2022/nanos-lite/src/fs.c,66,fs_open] Wrong open!

FATAL ERROR: 2 file open, name: fbp.mkf, mode: rb
FATAL ERROR: 2 file open, name: fbp.mkf, mode: rb
```

不是open的问题，是路径的问题。之前在真机上跑时的IO错误正是由于配置文件填写导致的，于是赶紧查看 **file.h** 文件进行对比：

```
{"/share/games/pal/rgm.mkf", 453202, 159897},
{"/share/games/pal/2.rpg", 188864, 613099},
{"/share/games/pal/wor16.asc", 5374, 801963},
{"/share/games/pal/5.rpg", 188864, 807337},
{"/share/games/pal/rng.mkf", 4546074, 996201},
{"/share/games/pal/3.rpg", 188864, 5542275},
{"/share/games/pal/abc.mkf", 1022564, 5731139},
{"/share/games/pal/4.rpg", 188864, 6753703},
{"/share/games/pal/desc.dat", 16027, 6942567},
{"/share/games/pal/voc.mkf", 1997044, 6958594},
{"/share/games/pal/m.msg", 188232, 8955638},
{"/share/games/pal/mgo.mkf", 1577442, 9143870},
{"/share/games/pal/fire.mkf", 834728, 10721312},
{"/share/games/pal/gop.mkf", 11530322, 11556040},
{"/share/games/pal/fbp.mkf", 1128064, 23086362},
{"/share/games/pal/f.mkf", 186966, 24214426},
{"/share/games/pal/wor16.fon", 82306, 24401392},
{"/share/games/pal/map.mkf", 1496578, 24483698},
{"/share/games/pal/1.rpg", 188864, 25980276},
{"/share/games/pal/word.dat", 5650, 26169140},
{"/share/games/pal/ball.mkf", 134704, 26174790},
{"/share/games/pal/sss.mkf", 557004, 26309494},
{"/share/games/pal/mus.mkf", 331284, 26866498},
{"/share/games/pal/sdlpal.cfg", 7364, 27197782},
{"/share/games/pal/data.mkf", 66418, 27205146},
{"/share/games/pal/pat.mkf", 8488, 27271564},
```

果然，结合上文的调试信息，发现是文件路径不对。而这个问题的答案就在 **GamePath** 中。于是修改配置文件如下：

```
# GamePath = ./data
GamePath=/share/games/pal
```

成功运行仙剑奇侠传



感动！

## 展示批处理系统

首先完善系统调用如下：

```
// nanos-lite/src/syscall.c
case SYS_exit: naive_uload(NULL, "/bin/nterm"); break;
case SYS_execve: c->GPRx = 0; naive_uload(NULL, (char*)a[1]);
break;

// navy-apps/libs/libos/src/syscall.c
int _execve(const char *fname, char * const argv[], char
*const envp[]) {
    return _syscall_(SYS_execve, (intptr_t)fname,
(intptr_t)argv, (intptr_t)envp);
}
```

然后在nterm中完成指令处理函数：

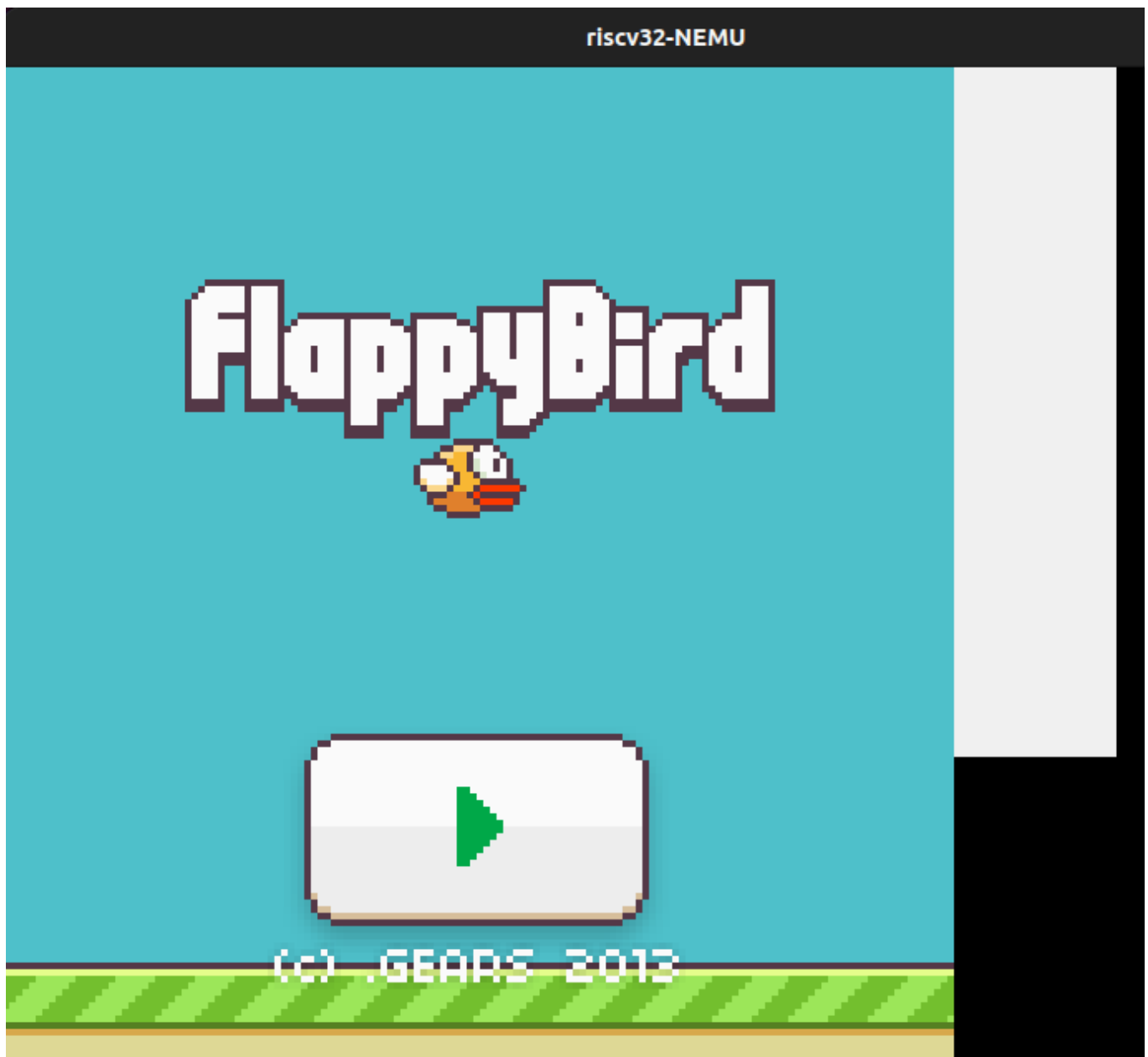
```
static void sh_handle_cmd(const char *cmd) {
    char inst[64];
    strcpy(inst, cmd);
    inst[strlen(inst) - 1] = '\0';

    char *token = strtok(inst, " ");
    char *argv[16];
    int argc = 0;

    while(token) {
        argv[argc++] = token;
        token = strtok(NULL, " ");
    }
    argv[argc] = NULL;

    execvp(argv[0], argv);
}
```

这样一来，在游戏界面输入 `/bin/bird` 或者 `/bin/pal` 等文件路径即可运行！（白色的背景就是我们的nterm）如果之后能添加退出按钮，展示效果将会更好



## 仙剑奇侠传到底如何运行？

显示屏幕无疑是一个硬件，而在硬件的角度，一帧画面的显示无非就是：

1. 刷新显示缓冲区内容；
2. 软件通知硬件模拟器刷新屏幕。

刷新的速度决定了程序的流畅程度，从硬件角度做的事情就这么点。剩下的刷新缓冲区和发出刷新信号就是软件的事情了。

软件方面主要负责寻找信息并放入“缓冲区”中，等待程序的输出。在这个过程中，操作系统首先得到画面的像素信息并抽象到 `dispinfo` 文件中。我们的特殊的库函数 `fs_write` 在执行中通过系统调用向画面文件的对应位置书写像素信息，并通知硬件刷新屏幕。由此，仙剑奇侠传的一帧帧画面便可以活灵活现地展现在我们面前。

---

PA3到此结束。