

# PA4实验报告

1811\*\*\*\_秦夙

## Part1

### 1.1 实验目的

1. 了解虚拟化地址空间相关知识
2. 学习虚拟地址空间转换的原理
3. 学习i386的分段和分页机制，以及页表相关内容

### 1.2 实验内容

1. 在 NEMU 中实现分页机制
2. 让用户程序运行在分页机制上
3. 在分页机制上运行仙剑奇侠传

### 1.3 实验步骤

#### 1.3.1 在 NEMU 中实现分页机制

分页机制是基于虚拟内存机制产生的，实现了按需分配的虚存管理机制。分页机制将一个个的虚拟页分别映射到相应的物理页上，把不连续的页面重新组织成连续的虚拟地址空间。每当加载程序的时候，操作系统就给程序分配相应的物理页，并为程序准备一个新的页表，在页表中填写程序用到的虚拟页到分配到的物理页的映射关系，到程序运行的时候，操作系统把之前为这个程序填写好的页表设置到 MMU 中，MMU 会根据页表的内容进行地址转换，把程序的虚拟地址空间映射到操作系统所希望的物理地址空间上。

按照实验指导书，实现分页机制首先需要在 NEMU 中添加 CR3 和 CR0 寄存器，CR3 寄存器用于存放页目录基地址，CR0 中包含表示分页机制是否开启的 PG 位。PA 中已经提供了用于定义 CR3 和 CR0 的数据结构（在 `nemu/include/memory/mmu.h` 中），我们只需要在 `nemu/include/cpu/reg.h` 的 `CPU_state` 中声明这两个寄存器，并在 `nemu/src/monitor/monitor.c` 的 `restart` 函数中初始化 CR0 即可：

```
//nemu/include/cpu/reg.h
typedef struct {
    ...
    CR0 cr0;
    CR3 cr3;
    ...
} CPU_state;

//nemu/src/monitor/monitor.c
static inline void restart() {
    ...
    cpu.cr0.val = 0x60000011;
```

```
...
}
```

之后我们需要实现操作 CR0 和 CR3 的指令，可以直接通过查找 i386 手册获得其指令码，也可以运行一遍程序，通过指令未实现的报错获取其指令码，操作 CR0 和 CR3 的指令是两字节宽的，指令码为 0x20 和 0x22，在 `opcode_table` 中对应位置填写指令处理信息：

```
...
/* 0x20 */ IDEX(mov_G2E,mov_cr2r), EMPTY, IDEX(mov_E2G,mov_r2cr), EMPTY,
...
```

之后在 `nemu\src\cpu\exec\system.c` 中实现操作这两个寄存器的具体指令内容，`mov_r2cr` 将寄存器中的信息存入 CR0 或 CR3 中，`mov_cr2r` 将 CR0 或 CR3 中的信息存入寄存器中，在指令执行时，首先 `id_dest->reg` 判断要处理的是 CR0 还是 CR3，之后根据具体的指令将其中的信息存入寄存器或将寄存器的信息存入 CR0 或 CR3：

```
make_EHelper(mov_r2cr) {
    switch(id_dest->reg){
        case 0: cpu.cr0.val=reg_l(id_src->reg); break;
        case 3: cpu.cr3.val=reg_l(id_src->reg); break;
        default: assert(0);
    }
    print_asm("movl %%s,%%cr%d", reg_name(id_src->reg, 4), id_dest->reg);
}

make_EHelper(mov_cr2r) {
    switch(id_src->reg) {
        case 0: reg_l(id_dest->reg)=cpu.cr0.val; break;
        case 3: reg_l(id_dest->reg)=cpu.cr3.val; break;
        default: assert(0);
    }
    print_asm("movl %%cr%d,%%s", id_src->reg, reg_name(id_dest->reg, 4));

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

这之后还需要修改 `nemu\src\memory\memory.c` 中的 `vaddr_read` 和 `vaddr_write`，让其根据分页机制是否打开进行不同的操作，如果分页机制未打开，直接按照原来的方式进行读取写入即可，否则需要先将虚拟地址转换为物理地址再读取写入，还要注意在读取和写入时，数据可能跨越页边界，按照实验指导书，在这里我们暂时不需要考虑这种情况，但在 part1 的最后要让仙剑运行起来还是要实现对于跨越页边界的数据的处理。而对于虚拟地址转换为物理地址，则需要通过额外定义函数 `page_translate` 实现。

在实现 `vaddr_read` 和 `vaddr_write` 时，首先检查 CR0 的 `paging` 判断分页机制是否开启，没有开启则按照原的方式直接调用 `paddr_read` 或 `paddr_write` 进行读取写入，否则进行虚拟地址到物理地址的转换。在转

换时。同样要向判断数据是否跨越页边界，没有跨域直接调用 `page_translate` 进行转换，否则将要读取或写入的数据在分页出划分成两部分，分别转换并读取或写入：

```
uint32_t vaddr_read(vaddr_t addr, int len) {
    if(cpu.cr0.paging){//分页机制开启
        if(((addr & 0xffff) + len) > PAGE_SIZE){//数据跨越页边界
            paddr_t paddr,low,high;
            int x;
            x=(int)((addr&0xffff)+len-4096);

            paddr=page_translate(addr,false);
            low=paddr_read(paddr,len-x);//读前一个页

            paddr=page_translate(addr+len-x,false);
            high=paddr_read(paddr,x);//读后一个页

            paddr=(high<<((len-x)<<3))+low;//拼接

            return paddr;
        }
        else {
            paddr_t paddr = page_translate(addr, false);
            return paddr_read(paddr, len);
        }
    }
    else{//分页机制关闭
        return paddr_read(addr, len);
    }
}

void vaddr_write(vaddr_t addr, int len, uint32_t data) {
    if(cpu.cr0.paging){//分页机制开启
        if(((addr & 0xffff) + len) > PAGE_SIZE){//数据跨越页边界
            paddr_t paddr,low,high;
            int x;
            x=(int)((addr&0xffff)+len-0x1000);

            low=(data<<(x<<3))>>(x>>3);//前一个页中写入的数据长度
            paddr=page_translate(addr,true);//转换
            paddr_write(paddr,len-x,low);//在前一个页中写入

            high=data>>((len-x)<<3);//后一个页中写入的数据长度
            paddr=page_translate(addr+len-x,true);//转换
            paddr_write(paddr,x,high);//在后一个页中写入
        }
        else{
            paddr_t paddr = page_translate(addr, true);
            paddr_write(paddr, len, data);
        }
    }
    else{//分页机制关闭
        paddr_write(addr, len, data);
    }
}
```

```

}
}

```

对于用于将虚拟地址转换为物理地址的函数 `page_translate`，该函数首先从 CR3 中获取页目录基地址，之后根据页目录基地址获取页目录地址，进而找到页表地址和要读取或写入的具体位置，并调用 `paddr_read` 或 `paddr_write` 进行读取写入，最后还要为该页面设置 `accessed` 和 `dirty` 位。需要注意的是，在查找到页目录或页表后，还需要检查其 `present` 位是否为有效，如果无效需要使用 `assert(0)` 终止程序：

```

paddr_t page_translate(vaddr_t addr, bool iswrite){
    paddr_t pde; // 页目录
    paddr_t pte; // 页
    paddr_t base1 = cpu.cr3.val; // 页目录基地址
    paddr_t pde_address = base1 + ((addr >> 22) << 2); // 页目录地址
    pde = paddr_read(pde_address, 4); // 页目录
    if(!(pde & PTE_P)) { // 检查 present 位，是无效页目录
        assert(0);
    }
    paddr_t base2 = pde & 0xfffff000; // 页表基地址
    paddr_t pte_address = base2 + ((addr & 0x003ff000) >> 10); // 页表地址
    pte = paddr_read(pte_address, 4); // 页表
    if(!(pte & PTE_P)) { // 检查 present 位，是无效页表
        assert(0);
    }

    paddr_t paddress = (pte & 0xfffff000) + (addr & 0xfff); // 实际地址

    // 设置 accessed 和 dirty 位
    pde = pde | PTE_A;
    pte = pte | PTE_A;
    if (iswrite) {
        pde = pde | PTE_D;
        pte = pte | PTE_D;
    }
    paddr_write(pde_address, 4, pde);
    paddr_write(pte_address, 4, pte);
    return paddress; // 返回实际地址
}

```

至此，我们已在 NEMU 中实现了 i386 分页机制。

### 1.3.2 让用户程序运行在分页机制上

实现 i386 分页机制之后，仙剑奇侠传可以顺利运行，但此时游戏并没有真正地运行在分页机制上，而是运行在内核的虚拟地址空间之上，依然在 0x4000000 的位置，如果将来 MM 把 0x4000000 所在的物理页分配出去，仙剑奇侠传的内容将会被覆盖，因此要让用户程序运行在操作系统为其分配的虚拟地址空间之上。

首先需要将 `navy-apps/Makefile.compile` 中的链接地址 `-Ttext` 参数和 `nanos-lite/src/loader.c` 中的 `DEFAULT_ENTRY` 改为 0x8048000，以避免用户程序的虚拟地址空间与内核相互重叠：

```
//navy-apps/Makefile.compile
LD_FLAGS += -Ttext 0x8048000

//nanos-lite/src/loader.c
#define DEFAULT_ENTRY ((void *)0x8048000)
```

接下来在 `nanos-lite/src/main.c` 中进行用户程序的加载，加载 `dummy` 程序：

```
int main() {
    ...
    init_fs();
    // uint32_t entry = loader(NULL, "/bin/pal");
    // ((void (*)(void))entry)();
    load_prog("/bin/dummy");
}
```

之后要实现 `nexus-am/am/arch/x86-nemu/src/pte.c` 中的函数 `_map`，该函数用将虚拟地址空间中的虚拟地址映射到物理地址，在 `_map` 函数中，首先找到页目录基地址，之后检查该目录项的 `present` 位是否有效，有效则直接设置到物理地址的映射，否则现申请一个空闲的物理页并将该页目录项 `present` 位设为有效，再设置映射：

```
void _map(_Protect *p, void *va, void *pa) {
    PDE* basePDE = (PDE*) p->ptr; //基地址
    uint32_t dir = PDX(va);
    uint32_t page = PTX(va);
    if(!(basePDE[dir] & PTE_P)){//检查present位是否有效
        PTE* basePTE = (PTE*)palloc_f(); //申请空闲物理页
        basePDE[dir] = (uint32_t)basePTE | PTE_P; //设置present位有效
        basePTE = (PTE*)(basePDE[dir] & 0xfffff000);
        basePTE[page] = (uint32_t)pa | PTE_P; //映射到pa
    }
    else{
        PTE* basePTE = (PTE*)(basePDE[dir] & 0xfffff000);
        basePTE[page] = (uint32_t)pa | PTE_P; //映射到pa
    }
}
```

最后，还需要修改 `nanos-lite/src/loader.c` 中定义的 `loader` 的内容，让其以页为单位加载用户程序，`loader` 在循环中读取文件内容，没读取一页大小的内容就申请一个新的页面，并将其映射到物理地址：

```
uintptr_t loader(_Protect *as, const char *filename) {
    int fd=fs_open(filename,0,0);
    int size=fs_filesz(fd);
    void *page;

    for(int i=0;i<size;i+=PGSIZE){
```

```

    page = (void*)new_page();
    _map(as, DEFAULT_ENTRY + i, page); //映射到物理地址
    fs_read(fd, page, PGSIZE); //读一页文件
}
fs_close(fd);

return (uintptr_t)DEFAULT_ENTRY;
}

```

全部实现后，运行 `dummy` 会看到输出 GOOD TRAP 的信息：

```

[09]src/mm.c,24,init_mm] free physical pages starting from 0x1d8d000
[src/main.c,21,main] 'Hello World!' from Nanos-lite
[src/main.c,22,main] Build time: 14:20:52, Jun 14 2021
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x1023e0, end = 0x1d47611,
size = 29643313 bytes
[src/main.c,29,main] Initializing interrupt/exception handler...
nemu: HIT GOOD TRAP at eip = 0x00100032

```

### 1.3.3 在分页机制上运行仙剑奇侠传

根据实验指导书，我们在pa3中实现的 `mm_brk` 函数直接返回0，表示堆区大小总是修改成功，现在因为 0x4000000 之上的内存不再是都可以让用户程序自由使用，因此我们需要修改 `mm_brk` 函数，新申请的堆区映射到虚拟地址空间中，从而使仙剑奇侠传能运行在分页机制上。

在pa给出的框架中，我们需要填补第二层 if 语句的内容，首先计算要增加的空间大小，之后在循环中调用 `_map` 函数将页面依次映射到虚拟地址：

```

if(new_brk > current->max_brk){
    // TODO: map memory region [current->max_brk, new_brk)
    // into address space current->as
    int size = new_brk - current->max_brk; //要增加的大小
    void *page;
    void* va = (void*)PGROUNDUP(current->max_brk);
    for(int i=0;i<size;i+=PGSIZE){
        page = (void*)new_page();
        _map(&current->as, va + i, page); //映射到虚拟地址
    }
}

```

全部实现后，运行仙剑奇侠传，可以正常运行：



## 1.4 问答

### 1.4.1 一些问题

#### 1.i386 不是一个 32 位的处理器吗,为什么表项中的基地址信息只有 20 位,而不是 32 位?

页表项中除了地址信息还有一些其他标志位，如表示页表是否有效的标志位等。而物理页是按照页大小对齐的，x86的页大小为4KB，即 $2^{12}\text{B}$ ，正好可以用  $32-12=20$  位表示基地址信息。

#### 2.手册上提到表项(包括 CR3)中的基地址都是物理地址,物理地址是必须的吗?能否使用虚拟地址?

不可以，因为表项（包括 CR3）是用来将虚拟地址装环岛物理地址的，如果用虚拟地址表示他们，就不能实现表项的虚拟地址到物理地址的转换。

#### 3.为什么不采用一级页表?或者说采用一级页表会有什么缺点?

储存一级页表需要  $1\text{M} \times 4\text{B} = 4\text{MB}$  空间，对内存要求较大而多级页表只需要在进程运行时将其使用的页表调入内容即可，内存占用小。同时，一级页表需要使用连续的空间存储，而多级页表因为添加了索引项，可以实现分开存储，对连续空间的要求较低。

### 1.4.2 空指针真的是"空"的吗?

查阅资料知，C语言中的空指针 NULL 是一个定义在 `stddef.h` 宏定义：



```
# define NULL ((void*)0)
```

根据上述代码，我们可以知道空指针指向进程的最小地址，通常为0，而线性地址0处没有映射，如果将线性地址0所在的虚拟也找到一个物理页与它建立映射，就可以使用线性地址0，空指针也就不会为“空”了。

### 1.4.3 内核映射的作用

根据实验指导书，注释掉用于拷贝内核映射的代码后再运行，出现了 `assert(0)` 报错：

```
(nemu) C
[0] src/mm.c,24,init_mm] free physical pages starting from 0x1d8d000
[src/main.c,21,main] 'Hello World!' from Nanos-lite
[src/main.c,22,main] Build time: 14:20:52, Jun 14 2021
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x1023e0, end = 0x1d47611,
size = 29643313 bytes
[src/main.c,29,main] Initializing interrupt/exception handler...
nemu: src/memory/memory.c:48: page_translate: Assertion `0' failed.
Makefile:47: recipe for target 'run' failed
make[1]: *** [run] 已放弃 (core dumped)
make[1]: Leaving directory '/home/qin2mu4/Desktop/ics2018/nemu'
/home/qin2mu4/Desktop/ics2018/nexus-am/Makefile.app:35: recipe for target 'run'
failed
make: *** [run] Error 2
```

检查报错处的代码，发现是在 `page_translate` 函数中因为 `pte` 的 `present` 位无效而报错：

```
47 | if(!(pde & 0x1)) { //检查present位，是无效页目录
48 | | assert(0);
49 | }
```

报错原因是因为进程的页表包括两部分，进程单独的页表和内核页表，前者是每个进程都不同的，后者所映射到的位置都是相同的（因为内核进程只有一个），被注释掉的代码是用于拷贝内核映射的，注释后导致进程的页目录表未能完全初始化，`page_translate` 函数在进行地址和转换时，内核部分的虚拟地址与物理地址间的映射关系就不能正确装换，进程访问的与内核相关的虚拟地址没有物理地址与其映射，因此会报错。

---

## Part2

---

### 2.1 实验目的

1. 学习上下文切换的过程
2. 学习上下文切换时进程选择的实现方法
3. 学习分时多任务的实现原理

### 2.2 实验内容

1. 实现内核自陷
2. 实现上下文切换
3. 分时运行仙剑奇侠传和 hello 程序
4. 优先级调度



## 2.3 实验步骤

### 2.3.1 实现内核自陷

操作系统进行上下文切换时，首先会保存当前进程的现场，再陷入内核，之后恢复要切入的进程现场，最后从内核陷入状态返回，因此要实现上下文切换需要实现内核自陷。

在pa中，内核自陷通过指令 `int $0x81` 触发，之后由 `irq_handle` 函数包装成 `_EVENT_TRAP` 事件，再由 `Nanos-lite` 返回第一个用户程序的现场。

实现时，我们首先需要在 `nanos-lite\src\main.c` 中调用 `_trap` 函数触发内核自陷，并注释掉 `nanos-lite\src\proc.c` 中的三行代码：

```
//nanos-lite\src\main.c
int main() {
    ...
    load_prog("/bin/pal");
    _trap();

    panic("Should not reach here");
}

//nanos-lite\src\proc.c
void load_prog(const char *filename) {
    ...
    // TODO: remove the following three lines after you have implemented _umake()
    // _switch(&pcb[i].as);
    // current = &pcb[i];
    // ((void (*)(void))entry)();
    ...
}
```

之后我们要在 `nexus-am\am\arch\x86-nemu\src\asye.c` 中添加代码，使 `irq_handle` 函数可以识别内核自陷并包装成 `_EVENT_TRAP` 事件。 `nexus-am\am\arch\x86-nemu\src\asye.c` 已经给出了系统调用的处理样例，只需要按照样例进行添加代码即可，在第一个函数中进行 case 0x81 的事件分发，在第二个函数中添加 idt 表项，并定义内核自陷的函数：

```
void vectrap();

_RegSet* irq_handle(_RegSet *tf) {
    ...
    switch (tf->irq) { //事件分发
        case 0x80: ev.event = _EVENT_SYSCALL; break;
        case 0x81: ev.event = _EVENT_TRAP; break;
        default: ev.event = _EVENT_ERROR; break;
    }
    ...
}

void _asye_init(_RegSet*(*h)(_Event, _RegSet*)) {
```

```

...
// ----- system call -----
idt[0x80] = GATE(STS_TG32, KSEL(SEG_KCODE), vecsys, DPL_USER);
idt[0x81] = GATE(STS_TG32, KSEL(SEG_KCODE), vectrap, DPL_USER); //添加idt表项
set_idt(idt, sizeof(idt));
...
}

```

除此之外，还需要在汇编代码文件 `nexus-am\am\arch\x86-nemu\src\trap.S` 中添加对内核自陷的支持，同样仿照已有的 `0x80` 系统调用，定义内核自陷的入口函数即可：

```

//nexus-am\am\arch\x86-nemu\src\trap.S
#----|-----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
.globl vectrap;   vectrap: pushl $0;  pushl  $0x81; jmp asm_trap
...

```

之后，根据实验指导书，在 `nanos-lite\src\irq.c` 中修改 `do_event` 函数，让其能够识别 `_EVENT_TRAP` 事件，并输出一句话，之后直接返回：

```

static _RegSet* do_event(_Event e, _RegSet* r) {
    switch (e.event) {
        case _EVENT_SYSCALL:
            return do_syscall(r);
        case _EVENT_TRAP:
            print("_EVENT_TRAP");
            return NULL;
        default: panic("Unhandled event ID = %d", e.event);
    }
    return NULL;
}

```

实现后运行程序，会看到 `Nanos-lite` 触发了 `main` 函数中最后的 `panic`：

```

[0]src/mm.c,44,init_mm] free physical pages starting from 0x1d8d000
[src/main.c,21,main] 'Hello World!' from Nanos-lite
[src/main.c,22,main] Build time: 19:48:31, Jun 16 2021
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x1024a0, end = 0x1d476d1,
size = 29643313 bytes
[src/main.c,29,main] Initializing interrupt/exception handler...
[src/main.c,44,main] system panic: Should not reach here
nemu: HIT BAD TRAP at eip = 0x00100032

```

### 2.3.2 实现上下文切换

根据实验指导书，实现上下文切换首先需要实现 PTE 的 `_umake` 函数，该函数定义在 `nexus-am\am\arch\x86-nemu\src\pte.c` 中，负责在加载程序时创建用户进程的上下文，`_umake` 函数需要在栈上进行初始化，然后返回陷阱帧的指针，再由 `Nanos-lite` 把这一指针记录到用户进程 PCB 的 `tf` 中。实现 `_umake` 时，首先设置好

`_start` 函数的栈帧，之后获取用户进程的 `tf`，将其 `cs` 寄存器设为8，以保证 `diff-test` 正常运行，最后将 `tf` 的 `eip` 设为 `entry` 并返回 `tf`：

```
_RegSet *_umake(_Protect *p, _Area ustack, _Area kstack, void *entry, char *const
argv[], char *const envp[]) {
    // 初始化陷阱帧
    uint32_t* ptr = (uint32_t*)(ustack.end-4);
    *(ptr--)=0;
    *(ptr--)=0;
    *(ptr--)=0;
    *(ptr--)=0; // 设置_start栈帧

    _RegSet* tf = (void*)(ptr-sizeof(_RegSet));
    tf->cs=8; // 使diff-test正常运行
    tf->eip=(uintptr_t)entry;

    return tf;
}
```

之后实现 `nanos-lite\src\proc.c` 中的 `schedule` 函数，并在 `nanos-lite\src\irq.c` 的 `do_event` 函数中调用。`schedule` 函数用于进程调度，在当前阶段该函数只需要总是切换到第一个用户进程即 `pcb[0]` 即可，首先保存当前进程的上下文指针，之后选择下一个要调度的进程 (`pcb[0]`)，最后调用 `_switch` 函数（定义在 `nexus-am\am\arch\x86-nemu\src\pte.c` 中）进行进程切换，代用 `_switch` 函数时传入了 `&current->as` 作为参数，`as` 是 `_Protect` 类型（`nanos-lite\include\proc.h`）的变量，在切换时主要使用到其中的 `ptr` 指针：

```
//nanos-lite\src\proc.c
_RegSet* schedule(_RegSet *prev) {
    current->tf = prev; //保存当前上下文指针
    current = &pcb[0]; //选择下一个要执行的进程
    _switch(&current->as); //切换
    return current->tf;
}

//nanos-lite\src\irq.c
static _RegSet* do_event(_Event e, _RegSet* r) {
    switch (e.event) {
        case _EVENT_SYSCALL:
            return do_syscall(r);
        case _EVENT_TRAP:
            return schedule(r);
        default: panic("Unhandled event ID = %d", e.event);
    }
    return NULL;
}
```

最后要修改 `nexus-am\am\arch\x86-nemu\src\trap.S` 中 `asm_trap` 函数的实现，使得从 `irq_handle` 返回后，先将栈顶指针切换到新进程的陷阱帧，然后才根据陷阱帧的内容恢复现场，`do_event` 调用 `schedule` 并

返回新进程的现场后，`irq_handle` 会将新进程陷阱帧位置作为函数返回值存放在 `eax` 寄存器中，因此只要将 `eax` 寄存器的值送入 `esp` 寄存器即可：

```
...
asm_trap:
    pushal

    pushl %esp
    call irq_handle

    movl %eax, %esp

    popal
    addl $8, %esp

    iret
```

全部实现后，`Nanos-lite` 就可以通过内核自陷触发上下文切换的方式运行仙剑奇侠传：



### 2.3.3 分时运行仙剑奇侠传和 hello 程序

在实现了了虚拟内存和上下文切换机制之后，`Nanos-lite` 就可以支持分时多任务了，即可以支持仙剑奇侠传和 `hello` 程序分时运行，但根据实验指导书的说明，多个需要更新画面的进程参与调度进行分时运行会导致画面被相互覆盖，因此我们不能让多个有画面的进程分时运行。

实现仙剑奇侠传和 `hello` 程序分时运行首先需要在 `nanos-lite\src\main.c` 中加载 `hello` 程序，之后修改 `nanos-lite\src\proc.c` 中的 `schedule` 函数，让其在调度时轮流返回两个程序的现场：

```
//nanos-lite\src\main.c
int main() {
    ...
    init_fs();
    load_prog("/bin/pal");
    load_prog("/bin/hello");
    _trap();
    panic("Should not reach here");
}

//nanos-lite\src\proc.c
_RegSet* schedule(_RegSet *prev) {
    current->tf = prev; //保存当前上下文指针
    current=(current == &pcb[0] ? &pcb[1] : &pcb[0]); //轮流返回两个进程的现场
    _switch(&current->as); //切换
    return current->tf;
}
```

这之后还需要一个时机来触发进程调度，实验指导书中给出的时机是在处理系统调用之后，调用 `schedule` 函数并返回其现场，因此我们还需要修改 `nanos-lite\src\irq.c` 中的 `do_event` 函数：

```
static _RegSet* do_event(_Event e, _RegSet* r) {
    switch (e.event) {
        case _EVENT_SYSCALL:
            do_syscall(r);
            return schedule(r);
        case _EVENT_TRAP:
            print("_EVENT_TRAP");
            return schedule(r);
        default:
            panic("Unhandled event ID = %d", e.event);
    }
    return NULL;
}
```

实现上述内容后，理论上我们可以看到仙剑奇侠传运行的同时 `hello` 程序也在输出，但实际上因为两个程序并不是真正的同时运行，而是轮流使用处理器，两个程序的运行速度都会下降，在实际测试中 `hello` 程序可以一直输出，但仙剑奇侠传的运行在原有的基础上更慢了，很长时间都没能出现画面。

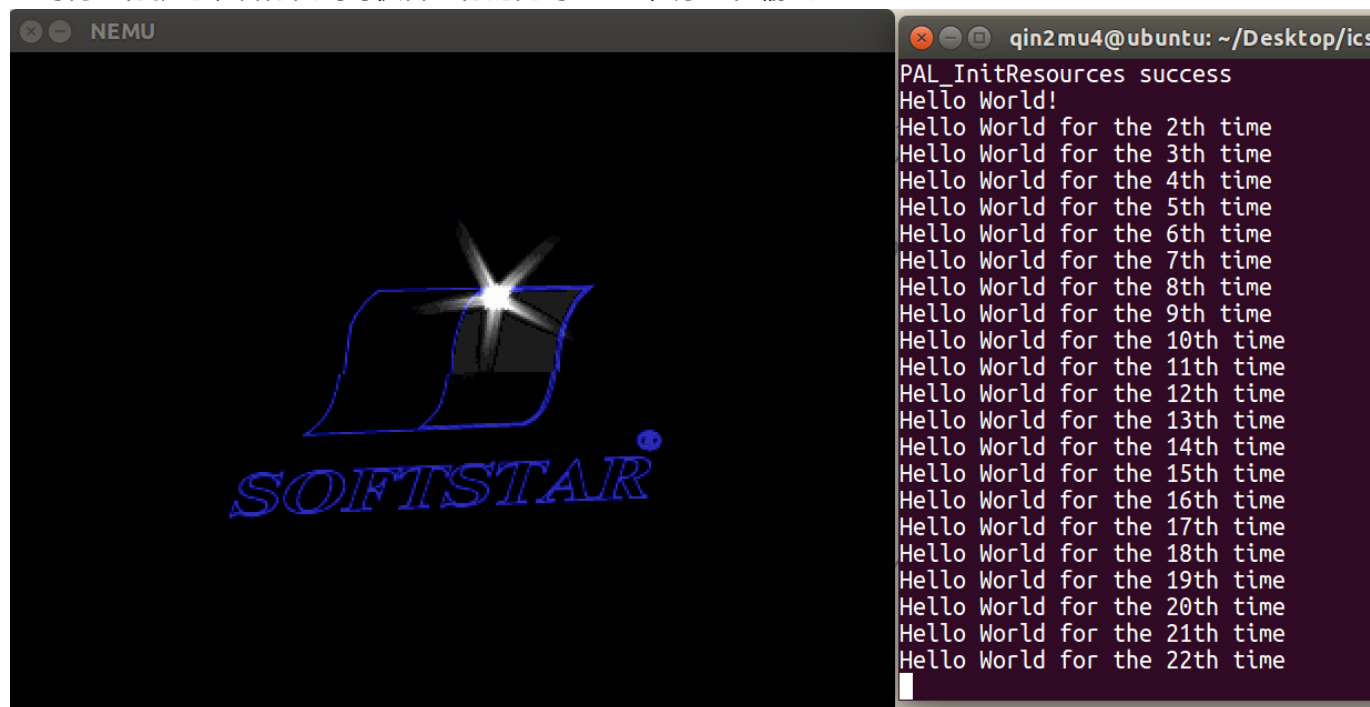
### 2.3.4 优先级调度

为了解决上述仙剑奇侠传运行过慢的问题，使其尽量保持原有性能，我们需要在进程调度时进行一些修改，让系统每调度 1000 次仙剑奇侠传才调度一次 hello 程序。修改 `schedule` 函数，添加一个用于计数仙剑奇侠传调度次数的变量 `count`，当该变量值小于 1000 时让 `schedule` 函数返回仙剑奇侠传的现场，当其值达到 1000 时 `schedule` 函数返回 hello 的现场，并将 `count` 清零：

```
_RegSet* schedule(_RegSet *prev) {
    static int count = 0;
    current->tf = prev; //保存当前上下文指针

    if (count < 1000) { // pa4-part2-4
        current = &pcb[0];
        count++;
    }
    else {
        current = &pcb[1];
        count = 0;
    }
    _switch(&current->as); //切换
    return current->tf;
}
```

此时再运行就可以看待仙剑奇侠传运行的同时 hello 程序也在输出：



## Part3

### 3.1 实验目的

1. 了解硬件中断相关内容
2. 了解时钟中断内容

## 3.2 实验内容

### 添加时钟中断

## 3.3 实验步骤：添加时钟中断添加时钟中断

在 pa 中，目前还存在一个致命的问题，即我们将上下文切换的触发条件寄托在程序的行为之上，只有程序主动触发了系统调用，才能触发上下文切换，但如果被调度的程序意外陷入死循环或是恶意程序，就会造成严重后果。为了解决这个问题，我们需要找到一种机制使得操作系统可以在不考虑程序的情况下强制进行上下文切换，这种机制是硬件中断机制，在 pa 中我们只需要实现时钟中断。

首先需要在 `nemu\include\cpu\reg.h` 中的 `cpu` 结构体中添加一个 `bool` 成员 `INTR` 作为 `cpu` 的 `INTR` 引脚，表示是否有中断信号到来，并在 `nemu\src\cpu\intr.c` 的 `dev_raise_intr` 函数中将 `INTR` 引脚设置为高电平，表示中断信号到来；之后在 `nemu\src\cpu\exec\exec.c` 的 `exec_wrapper` 的末尾对 `INTR` 引脚进行轮询，每次执行完一条指令就查看是否有硬件中断到来；最后修改 `nemu\src\cpu\intr.c` 中 `raise_intr` 函数的代码，在保存 `EFLAGS` 寄存器后，将其 `IF` 位置为 0，让处理器进入关中断状态，避免中断嵌套：

```
//nemu\include\cpu\reg.h
typedef struct {
    ...
    bool INTR;
} CPU_state;

//nemu\src\cpu\intr.c
void dev_raise_intr() {
    cpu.INTR = true; //设为高电平
}

void raise_intr(uint8_t NO, vaddr_t ret_addr) {
    ...
    cpu.eflags.IF = 0; //关中断
}

//nemu\src\cpu\exec\exec.c
void exec_wrapper(bool print_flag) {
    ...
    //pa4-part3 轮询INTR引脚
    #define TIMER_IRQ 32
    if (cpu.INTR & cpu.eflags.IF) {
        cpu.INTR = false;
        raise_intr(TIMER_IRQ, cpu.eip);
        update_eip();
    }
}
```

除了上述内容外，还要在软件上进行修改。

第一，要在 `ASYS` 中添加时钟中断的支持，并将时钟中断打包成 `_EVENT_IRQ_TIME` 事件，这一部分和之前做的内核自陷大同小异，首先在 `asys.c` 中添加对时钟中断的事件分发、添加 `idt` 表项，并定义时钟中断函数，之后 `trap.S` 中添加时钟中断函数：



```
//nexus-am\am\arch\x86-nemu\src\asye.c
void vectime();
_RegSet* irq_handle(_RegSet *tf) {
    ...
    switch (tf->irq) { //事件分发
        case 0x80: ev.event = _EVENT_SYSCALL; break;
        case 0x81: ev.event = _EVENT_TRAP; break;
        case 0x20: ev.event = _EVENT_IRQ_TIME; break;
        default: ev.event = _EVENT_ERROR; break;
    }
    ...
}

void _asye_init(_RegSet*(*h)(_Event, _RegSet*)) {
    ...
    // ----- system call -----
    idt[0x80] = GATE(STS_TG32, KSEL(SEG_KCODE), vecsys, DPL_USER);
    idt[0x81] = GATE(STS_TG32, KSEL(SEG_KCODE), vectrap, DPL_USER); //添加idt表项
    idt[0x20] = GATE(STS_TG32, KSEL(SEG_KCODE), vectime, DPL_USER); //添加idt表项
    set_idt(idt, sizeof(idt));
    ...
}

//nexus-am\am\arch\x86-nemu\src\trap.S
#----|-----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
.globl vectrap;   vectrap:   pushl $0;    pushl    $0x81; jmp asm_trap
.globl vectime;   vectime:   pushl $0;    pushl    $0x20; jmp asm_trap
...
```

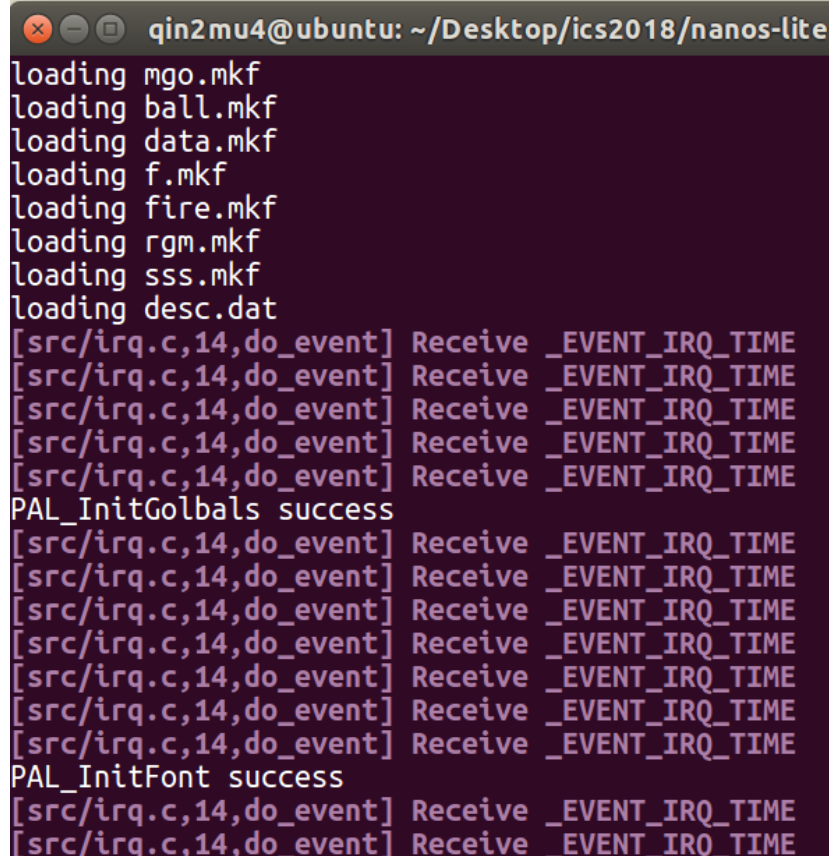
第二，修改 `irq.c` 中的 `do_event` 函数，让其能够识别 `_EVENT_IRQ_TIME` 事件，并在识别后直接调用 `schedule` 进行进程调度，同时去掉 part3 中系统调用 之后调用的 `schedule` 代码：

```
static _RegSet* do_event(_Event e, _RegSet* r) {
    switch (e.event) {
        case _EVENT_SYSCALL:
            return do_syscall(r);
        case _EVENT_TRAP:
            print("_EVENT_TRAP");
            return schedule(r);
        case _EVENT_IRQ_TIME:
            // Log("_EVENT_IRQ_TIME");
            return schedule(r);
        default: panic("Unhandled event ID = %d", e.event);
    }
    return NULL;
}
```

最后，需要修改 `pte.c` 中的 `_umake` 函数，在构造现场时设置正确的 `EFLAGS` 打开用户进程的中断：

```
_RegSet *_umake(_Protect *p, _Area ustack, _Area kstack, void *entry, char *const
argv[], char *const envp[]) {
    ...
    tf->eflags = 0x20 | FL_IF; // pa4-part3
    return tf;
}
```

实现之后再次运行程序，可以看到触发了时钟中断：



```
loading mgo.mkf
loading ball.mkf
loading data.mkf
loading f.mkf
loading fire.mkf
loading rgm.mkf
loading sss.mkf
loading desc.dat
[src/irq.c,14,do_event] Receive _EVENT_IRQ_TIME
[src/irq.c,14,do_event] Receive _EVENT_IRQ_TIME
[src/irq.c,14,do_event] Receive _EVENT_IRQ_TIME
[src/irq.c,14,do_event] Receive _EVENT_IRQ_TIME
[src/irq.c,14,do_event] Receive _EVENT_IRQ_TIME
PAL_InitGolbals success
[src/irq.c,14,do_event] Receive _EVENT_IRQ_TIME
[src/irq.c,14,do_event] Receive _EVENT_IRQ_TIME
[src/irq.c,14,do_event] Receive _EVENT_IRQ_TIME
[src/irq.c,14,do_event] Receive _EVENT_IRQ_TIME
[src/irq.c,14,do_event] Receive _EVENT_IRQ_TIME
[src/irq.c,14,do_event] Receive _EVENT_IRQ_TIME
[src/irq.c,14,do_event] Receive _EVENT_IRQ_TIME
PAL_InitFont success
[src/irq.c,14,do_event] Receive _EVENT_IRQ_TIME
[src/irq.c,14,do_event] Receive _EVENT_IRQ_TIME
```

### 3.4 必答题：解释分页机制和硬件中断是如何支撑仙剑奇侠传和 hello 程序在我们的计算机系统中分时运行的。

#### 3.4.1 分页机制

分页机制的作用主要是提供连续线性地址到不连续物理地址的映射，以及用大小相等的页代替大小不等的段。每加载一个进程，操作系统按照进程中各段的起始范围，在进程自己的4GB虚拟地址空间中寻找可用空间分配内存段。接着操作系统开始为这些虚拟内存分配真实的物理内存页，它查找物理内存中可用的页，然后在页中登记这些物理页地址，这样就完成了分页机制下虚拟页到物理页的映射，每个进程都以为自己独享4GB地址空间。

在 `pa` 中，分页机制涉及到了 `NEMU` 和 `nanos-lite`。首先，`NEMU` 中提供存放页目录地址的 `CR3` 寄存器和包含指示分页机制是否开启位的 `CR0` 寄存器，同时还提供了可以用于转换虚拟地址的内存读写函数。而 `nanos-lite` 模拟了简易操作系统，提供了操作系统对分页机制的支持。

#### 3.4.2 硬件中断

硬件中断是由与系统相连的外设(比如网卡 硬盘 键盘等)自动产生的，每个设备或设备集都有他自己的IRQ(中断请求)，基于IRQ，CPU可以将相应的请求分发到相应的硬件驱动上，处理中断的驱动是需要运行在CPU上的，因此，当中断产生时，CPU会暂时停止当前程序的程序转而执行中断请求。

分页机制使得多个程序可以同时运行，但此时操作系统的控制权在运行着的程序手中，如果该程序出现bug或为恶意程序，一直没有主动陷入内核进行上下文切换，将会造成严重后果，此时使用硬件中断中的时钟中断可以保证强制切换上下文，及保证了多个程序的正常运行，不会造成某一程序长时间占有资源。除此之外，硬件中断还可以接收其他设备的信号并及时做出处理。

在 pa 中，硬件中断是否到来主要通过 INTR 是否为高位判断，在 `nemu\src\cpu\exec\exec.c` 中的 `exec_wrapper` 最末尾的代码保证了每执行完一条指令就轮询一遍 INTR，如果有中断发生则切换到中断处理程序中。对于发生的中断，则可以在 `nanos-lite\src\irq.c` 的 `do_event` 函数中进行捕获和处理：

```
#define TIMER_IRQ 32

if (cpu.INTR & cpu.eflags.IF) {
    cpu.INTR = false;
    raise_intr(TIMER_IRQ, cpu.eip);
    update_eip();
}
```

```
static _RegSet* do_event(_Event e, _RegSet* r) {
    switch (e.event) {
        case _EVENT_SYSCALL:
            return do_syscall(r);
            // return schedule(r); //pa4-part2-3
        case _EVENT_TRAP:
            print("_EVENT_TRAP");
            // return NULL;
            return schedule(r);
        case _EVENT_IRQ_TIME:
            // Log("_EVENT_IRQ_TIME");
            return schedule(r);
        default: panic("Unhandled event ID = %d", e.event);
    }

    return NULL;
}
```

# 编写不朽的传说

根据实验指导书，我们需要运行第三个程序 `videotest`，并使用 F12 键切换仙家奇侠传和 `videotest` 程序。

首先在 `main.c` 中添加第三个程序：

```
int main() {
    ...
    init_fs();
    load_prog("/bin/pal");
    load_prog("/bin/hello");
    load_prog("/bin/videotest");
    _trap();
    panic("Should not reach here");
}
```

之后修改 `proc.c` 中的 `schedule` 函数，让其能够调度三个进程。因为现在有两个游戏需要使用图形输出，因此我们需要添加一个变量 `current_game` 来维护当前的游戏。我们依旧沿用之前的策略，每运行 1000 次游戏，运行一次 `hello` 程序，运行的游戏是 `current_game` 指向的游戏，而 `current_game` 具体指向哪个游戏由 F12 按下的次数决定，对于 `current_game` 指向游戏的切换，将在下面处理 F12 按下事件时说明：

```
static PCB *current_game = &pcb[0];
_RegSet* schedule(_RegSet *prev) {
    static int count = 0;
    current->tf = prev; //保存当前上下文指针
    if (count < 1000) {
        current = current_game;
        count++;
    }
    else {
        current = &pcb[1];
        count = 0;
    }
    _switch(&current->as); //切换
    return current->tf;
}
```

最后，我们要修改 `nanos-lite\src\device.c` 中的 `events_read` 函数，使其能够识别出 F12 键盘事件并做出相应操作。为了让 `events_read` 函数能够修改 `current_game` 指向的游戏进程，我们还需要在 `schedule` 函数中添加一个可以操作 `current_game` 的函数 `switch_game` 供 `events_read` 调用，该函数会切换 `current_game` 指向的游戏进程。之后在按键事件的处理中添加对 F12 按键的识别，并在识别出该事件后调用 `switch_game` 函数：

```
//nanos-lite\src\proc.c
void switch_game() {
```

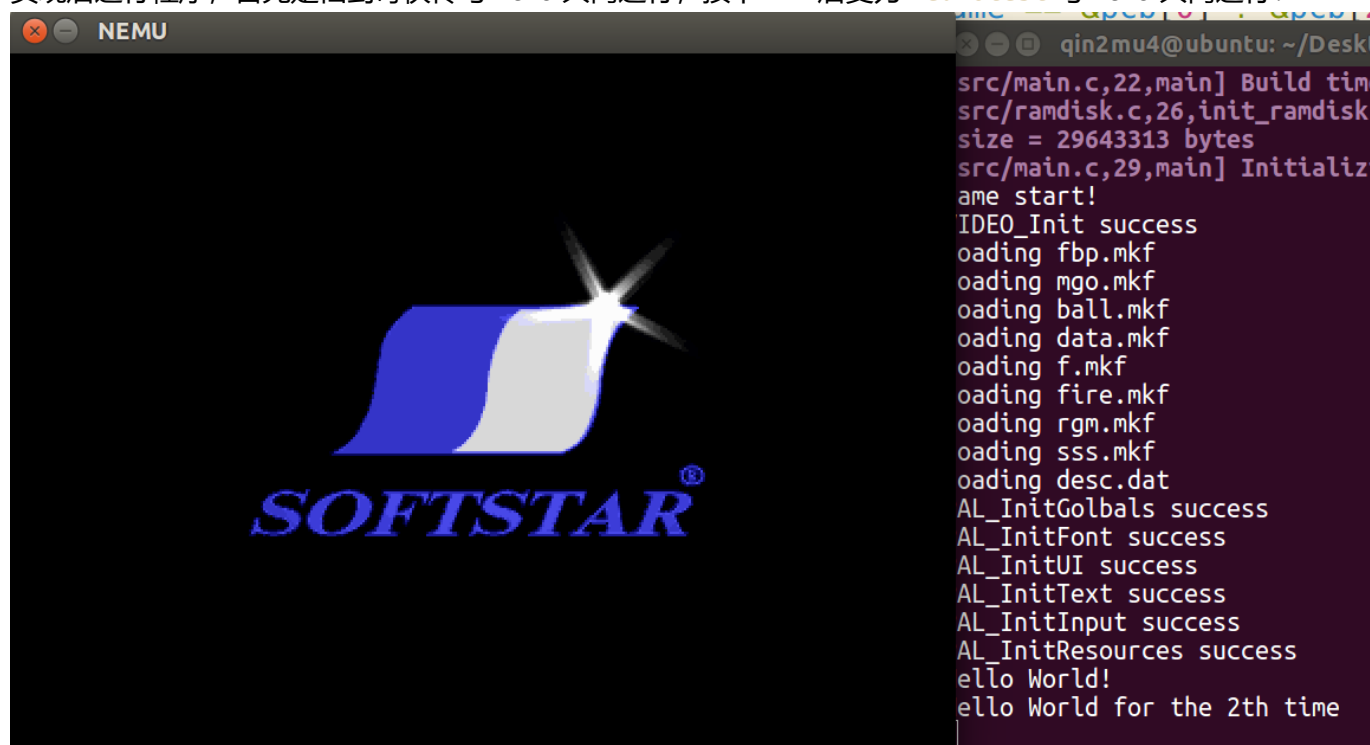
```

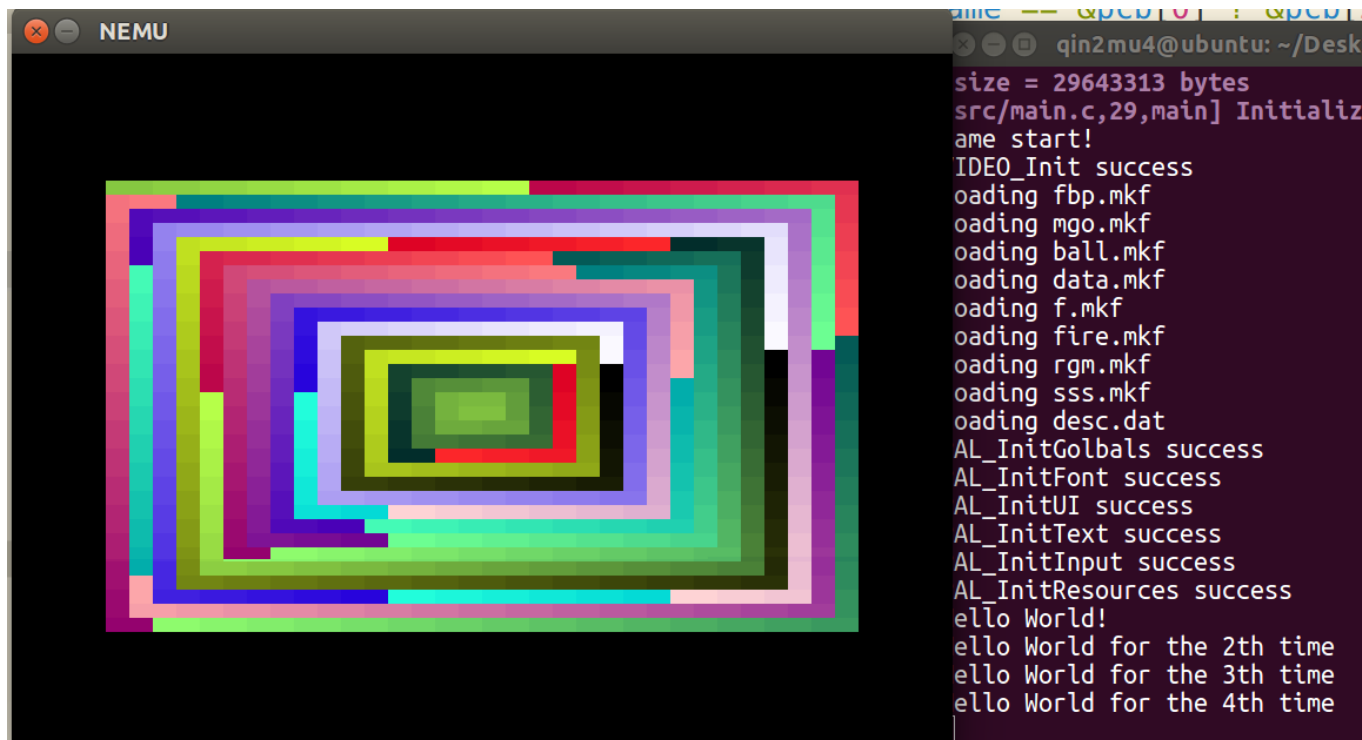
    current_game = (current_game == &pcb[0] ? &pcb[2] : &pcb[0]);
}

//nanos-lite\src\device.c
size_t events_read(void *buf, size_t len) {
    int key = _read_key();
    char keydown_char = (key & 0x8000 ? 'd' : 'u'); //通码=断码+0x8000
    int keyid = key & ~0x8000;
    if(keyid != _KEY_NONE) { //优先处理按键事件
        snprintf(buf, len, "k%c %s\n", keydown_char, keyname[keyid]); //写入按键事件
        if ((key & 0x8000) && (keyid == _KEY_F12)) {
            switch_game(); //切换游戏
        }
        return strlen(buf);
    }
    else {
        unsigned long time_ms = _uptime();
        return snprintf(buf, len, "t %d\n", time_ms) - 1; //写入时间
    }
    return 0;
}

```

实现后运行程序，首先是仙剑奇侠传与 hello 共同运行，按下 F12 后变为 vediotest 与 hello 共同运行：





## 遇到的问题

在 part1 中，实现全部要求的内容后运行仙剑奇侠传一直无法成功，会因为 `present` 为 0 而报 `assert(0)` 的错误，尝试了很久都没改好，后来想到可能是实现的 `mm_brk` 函数没有被调用，pa3 在实现 `nanos-lite/src/syscall.c` 的 `do_syscall` 函数时，对于 `SYS_brk` 时间是直接将 0 作为了返回值赋给 `SYSCALL_ARG1(r)`，在 pa4 中应该调用 `mm_brk` 函数进行堆区的修改和地址映射。更改这里后可以正常运行了。

```
case SYS_brk:
    // SYSCALL_ARG1(r) = 0; // pa3
    SYSCALL_ARG1(r) = mm_brk(a[1]); // pa4
    break;
```