PA4 实验报告

1711346-李明旗

目录

实验内容 **1**
PA4 阶段一 **2**
在 NEMU 中实现分页机制 2
CR3 与 CR0 的实现 2
虚拟地址的转换 3
让用户程序运行在分页机制上 4
在分页机制上运行仙剑奇侠传 7
PA4 阶段二 **8**
实现内核自陷 9
PCB 进程控制块 9
事件分发 9
实现上下文切换 10
分时多任务 12
优先级调度 12
运行结果 13
PA4 阶段三 **13**
添加时钟中断 13
软件处理时钟中断 14
必答题 15
编写不朽的传奇 **15**
PA 实验感想 **17**

实验内容

PA4 实验中主要涉及三个阶段：

第一阶段：熟悉虚拟内存、保护机制的概念，完成虚实地址转换

第二阶段，熟悉多道程序系统的基本思想，把批处理的 Nanos-lite 改造成一个多道程序操作系统：内存中可以同时存在多个进程；进程之间可以相互切换；

第三阶段：处理外部中断

PA4 阶段一

在 NEMU 中实现分页机制

由于页表位于内存中,但计算机启动的时候,内存中并没有有效的数据,因此我们不可能让计算机启动的时 候就开启分页机制.操作系统为了启动分页机制,首先需要准备一些内核页表.框架代码已经为我们实现好这一 功能了(见 nexus-am/am/arch/x86-nemu/src/pte.c 的 _pte_init() 函 数). 只 需 要 在 nanos-lite/src/main.c 中 定义宏 HAS_PTE,Nanos-lite 在初始化的时候首先就会调用 init_mm()函数(在 nanos-lite/src/mm.c 中定义)来 初始化 MM.这里的 MM 是指存储管理器(Memory Manager)模块,它专门负责分页相关的存储管理.

CR3 与 CR0 的实现

代码：nemu/include/cpu/reg.h

```
//添加CR0,CR3
CR0 cr0;
CR3 cr3;
```

nemu/src/monitor/monitor.c 中

```
//进行cr0的初始化
cpu.cr0.val=0x60000011;
```

nemu/include/cpu/rtl.h rtl 指令实现对 CR3 与 CR0 的访问与修改操作

```
static inline void rtl_lr(rtlreg_t* dest, int r, int width) {
  switch (width) {
    case 4: rtl_lr_l(dest, r); return;
    case 1: rtl_lr_b(dest, r); return;
    case 2: rtl_lr_w(dest, r); return;
    default: assert(0);
  }
}

static inline void rtl_sr(int r, int width, const rtlreg_t* src1) {
  switch (width) {
    case 4: rtl_sr_l(r, src1); return;
    case 1: rtl_sr_b(r, src1); return;
    case 2: rtl_sr_w(r, src1); return;
    default: assert(0);
  }
}
```

nemu/src/cpu/decode/decode.c 中

```
make_DHelper(out_a2I) {
  decode_op_a(eip, id_src, true);
  id_dest->width = 1;
  decode_op_I(eip, id_dest, true);
}

make_DHelper(out_a2dx) {
  decode_op_a(eip, id_src, true);

  id_dest->type = OP_TYPE_REG;
  id_dest->reg = R_DX;
  rtl_lr_w(&id_dest->val, R_DX);
#ifdef DEBUG
  sprintf(id_dest->str, "(%%dx)");
#endif
```

虚拟地址的转换

为了实现 vaddr_read()和 vaddr_write()函数，我们还需要一个把虚拟地址转 换成物理地址的函数 page_translate()。

代码实现：

nemu/src/memory/memory.c 增加 mmu.h 头文件的引用：

```
#define PDX(va)     (((uint32_t)(va) >> 22) & 0x3ff)
#define PTX(va)     (((uint32_t)(va) >> 12) & 0x3ff)
#define OFF(va)     ((uint32_t)(va) & 0xfff)
#define PTE_ADDR(pte)   ((uint32_t)(pte) & ~0xfff)
```

page_translate 增加参数 iswrite 来判断读或写操作，据此修改对应的 Accessed 与 Dirty 位。

```
paddr_t page_translate(vaddr_t addr, bool w1r0) {
    //Log("before:addr=%x",addr);
    //aka page_walk
    PDE pde, *pgdir;
    PTE pte, *pgtab;
    // 只有进入保护模式并开启分页机制后才会进行页级地址转换。。。。。。。。。
    if (cpu.cr0.protect_enable && cpu.cr0.paging) {
            pgdir = (PDE *)(PTE_ADDR(cpu.cr3.val)); //cr3存放20位的基址作为页目录入口
            pde.val = paddr_read((paddr_t)&pgdir[PDX(addr)], 4);
            assert(pde.present);
            pde.accessed = 1;

            pgtab = (PTE *)(PTE_ADDR(pde.val));  //页目录存放20位的基址作为页表入口
            pte.val = paddr_read((paddr_t)&pgtab[PTX(addr)], 4);
            assert(pte.present);
            pte.accessed = 1;
            pte.dirty = w1r0 ? 1 : pte.dirty; //写则置脏位

            //pte高20位和线性地址低12位拼接成真实地址
            //Log("cr3=0x%x,cr0=0x%x",cpu.cr3.val,cpu.cr0.val);
            //Log("after=%x",PTE_ADDR(pte.val) | OFF(addr));
```

vaddr_read 与 vaddr_write

```
uint32_t vaddr_read(vaddr_t addr, int len) {
  //PAGE_MASK = 0xfff
  if ((((addr) + (len) - 1) & ~PAGE_MASK) != ((addr) & ~PAGE_MASK)) {
        //data cross the page boundary
        uint32_t data = 0;
        for(int i=0;i<len;i++){
                paddr_t paddr = page_translate(addr + i, false);
                data += (paddr_read(paddr, 1))<<8*i;
        }
        return data;
        //assert(0);
  } else {
    paddr_t paddr = page_translate(addr, false);
    return paddr_read(paddr, len);
  }
}

void vaddr_write(vaddr_t addr, int len, uint32_t data) {
  if ((((addr) + (len) - 1) & ~PAGE_MASK) != ((addr) & ~PAGE_MASK)) {
        //data cross the page boundary
        for(int i=0;i<len;i++){ //len 最大为4
                paddr_t paddr = page_translate(addr + i,true);
                paddr_write(paddr,1,data>>8*i);
        }
        //assert(0);
  } else {
```

让用户程序运行在分页机制上

我们应该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.为此,我们需要对工程 作一些变动.首先需要将 navy-apps/Makefile.compile 中的链接地

址-Ttext 参数改为 0x8048000,这是为了避免 用户程序的虚拟地址空间与内核相互重叠,从而产生非预期的错误.同样的,nanos-lite/src/loader.c 中的 DEFAULT_ENTRY 也需要作相应的修改.

让用户程序运行在分页机制上

根据上述的讲义内容,在 PTE 中实现_map(),然后修改 loader()的内容,通过_map()在用户程序的虚拟地址空间中创建虚拟页,并把用户程序加载到虚拟地址空间上.

实现正确后,你会看到 dummy 程序最后输出 GOOD TRAP 的信息,说明它确实在虚拟地址空间上成功运行了.

要将 navy-apps/Makefile.compile 中的链接地址 -Ttext 参数改为 0x8048000

```
ifeq ($(LINK), dynamic)
  CFLAGS   += -fPIE
  CXXFLAGS += -fPIE
  LDFLAGS  += -fpie -shared
else
  LDFLAGS += -Ttext 0x8048000
endif
```

修改 nanos-lite/src/loader.c

```
#define DEFAULT_ENTRY ((void *)0x8048000)
```

修改 nanos-lite/src/main.c

```
load_prog("/bin/dummy");
```

为了让用户进程在将来可以正确地运行:用户进程在将来使用虚拟地址访问内存,在 loader 为 用户进程准备的映射下,虚拟地址被转换成物理地址,通过这一物理地址访问到的物理内存,恰好就是用户进程 想要访问的数据.为了提供映射一页的功能,你需要在 AM 中实现_map()函数

nexus-am/am/arch/x86-nemu/src/pte.c

```
void _map(_Protect *p, void *va, void *pa) {
        PDE *pgdir = p->ptr;
        PDE *pde = &pgdir[PDX(va)];
        PTE *pgtab;
        if (*pde & PTE_P) { //present
                pgtab = (PTE *)PTE_ADDR(*pde);
        } else {
                //映射过程中发现需要申请新的页表
                pgtab = (PTE *)palloc_f();
                *pde = PTE_ADDR(pgtab) | PTE_P;
        }
        pgtab[PTX(va)] = PTE_ADDR(pa) | PTE_P;
}
```

nanos-lite/src/loader.c loader()，获取用户程序的大小之后，以页为单位进行加载：

```
uintptr_t loader(_Protect *as, const char *filename) {
  int fd = fs_open(filename, 0, 0);
  int bytes = fs_filesz(fd); //出错在之前为size_t

  Log("Load [%d] %s with size: %d", fd, filename, bytes);

  void *pa,*va = DEFAULT_ENTRY;
  while(bytes>0){
        pa = new_page(); //申请空闲物理页
        _map(as, va, pa);//该物理页映射到用户程序虚拟地址空间
        fs_read(fd, pa, PGSIZE);  //读一页文件到该物理页

        va += PGSIZE;
        bytes -= PGSIZE;
  }
  //fs_read(fd,DEFAULT_ENTRY,bytes);
  fs_close(fd);
  return (uintptr_t)DEFAULT_ENTRY;
}
```

运行结果：

```
(nemu) c
[src/mm.c,47,init_mm] free physical pages starting from 0x2cd8000
[src/main.c,20,main] 'Hello World!' from Nanos-lite
[src/main.c,21,main] Build time: 10:28:00, Jun 11 2020
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x103280, end = 0x2c93ef8,
 size = 45681784 bytes
[src/main.c,28,main] Initializing interrupt/exception handler...
[src/fs.c,38,init_fs] set FD_FB size=480000
[src/loader.c,18,loader] Load [50] /bin/dummy with size: 17952
nemu: HIT GOOD TRAP at eip = 0x00100032

(nemu)
```

在分页机制上运行仙剑奇侠传


在分页机制上运行仙剑奇侠传

之前我们让 mm_brk()函数直接返回 0,表示用户程序的堆区大小修改总是成功,这是因为在实现分页机制之前,0x4000000 之上的内存都可以让用户程序自由使用.现在用户程序运行在虚拟地址空间之上,我们还需要在 mm_brk()中把新申请的堆区映射到虚拟地址空间中:

```
int mm_brk(uint32_t new_brk) {
  if (current->cur_brk == 0) {
    current->cur_brk = current->max_brk = new_brk;
  }
  else {
    if (new_brk > current->max_brk) {
      // TODO: map memory region [current->max_brk, new_brk)
      // into address space current->as


      current->max_brk = new_brk;
    }

    current->cur_brk = new_brk;
  }

  return 0;
}
```

你需要填充上述 TODO 处的代码,其中 current 是一个特殊的指针,我们会在后面介绍它.你需要注意 _map()参数是否需要按页对齐的问题(这取决于你的_map()实现).为了简化,我们也不实现堆区的回收功能了.实现正确后,仙剑奇侠传就可以正确在分页机制上运行了.

为了在分页机制上运行仙剑奇侠传，我们还需要考虑堆区的问题，之前我们 让 mm_brk()函数直接返回 0,表示用户进程对堆区大小的修改总是成功的。这是 因为在分页机制之前，0x30000000/0x83000000 之上的内存都可以让程序自由使用，现在用户程序运行在虚拟地址空间之上,我们需要在 mm_brk()中把新申请的堆区映射到虚拟地址空间中

nanos-lite/src/mm.c

```
int mm_brk(uint32_t new_brk)
{
        if (current->cur_brk == 0)
        {
                current->cur_brk = current->max_brk = new_brk;
        }
        else
        {
                if (new_brk > current->max_brk)
                {
                        // TODO: map memory region [current->max_brk, new_brk)
                        // into address space current->as
                        uintptr_t pa,va;
                        for(va=(current->max_brk+0xfff) & ~0xfff;va<new_brk;va
+=PGSIZE)
                        {
                           pa=(uintptr_t)new_page();
                           _map(&current->as,(void *)va,(void *)pa);
                        }
                        current->max_brk = new_brk; // =va ?
                }
        current->cur_brk = new_brk;
        }
        return 0;
}
```

成功运行仙剑奇侠传。

PA4 阶段二

我们已经可以让用户程序运行在相互独立的虚拟地址空间上了,我们只需要再加入上下文切换的机制,就可以实现一个真正的分时多任务操作系统了

此阶段的目标就是让 nanos-lite 加载仙剑奇侠传和 hello 这两个用户进程。如果 你的实现正确，你将可以一边运行仙剑奇侠传的同时，一边输出 hello 信息

实现内核自陷

实现内核自陷

修改 Nanos-lite 的如下代码:

```
--- nanos-lite/src/main.c
+++ nanos-lite/src/main.c
@@ -33,3 +33,5 @@
   load_prog("/bin/pal");

+  _trap();
+
   panic("Should not reach here");
--- nanos-lite/src/proc.c
+++ nanos-lite/src/proc.c
@@ -17,4 +17,4 @@
   // TODO: remove the following three lines after you have implemented _umake()
-  _switch(&pcb[i].as);
-  current = &pcb[i];
-  ((void (*)(void))entry)();
+  // _switch(&pcb[i].as);
+  // current = &pcb[i];
+  // ((void (*)(void))entry)();
```

并在 ASYE 添加相应的代码,使得 irq_handle()可以识别内核自陷并包装成_EVENT_TRAP 事件,Nanos-lite 接收到_EVENT_TRAP 之后可以输出一句话,然后直接返回即可,因为真正的上下文切换还需要正确实现_umake()之后才能实现.实现正确之后,你会看到 Nanos-lite 触发了 main()函数中最后的 panic.如果你不知道应该怎么做,请参考你对 PA3 必答题中关于系统调用部分的回答.

PCB 进程控制块

为了方便对进程进行管理,操作系统使用一种 叫进程控制块(PCB,process control block)的数据结构,为每一个进程维护一个 PCB.Nanos-lite 的框架代码中已经定义了我们所需要使用的 PCB 结构(在 nanos-lite/include/proc.h 中定义):

```
typedef union {
  uint8_t stack[STACK_SIZE] PG_ALIGN;
  struct {
    _RegSet *tf;
    _Protect as;
    uintptr_t cur_brk;
    // we do not free memory, so use `max_b
    uintptr_t max_brk;
  };
} PCB;
```

事件分发

在 asye.c 添加相应的代码,使得 irq_handle()可以识别内核自陷并包装成

_EVENT_TRAP 事件,Nanos-lite 接收到_EVENT_TRAP 之后可以输出一句话,

然后直接返回即可。

```
_RegSet* irq_handle(_RegSet *tf) {
  _RegSet *next = tf;
  if (H) {
    _Event ev;
    switch (tf->irq) {
      case 0x80: ev.event = _EVENT_SYSCALL; break;
      case 0x81: ev.event = _EVENT_TRAP; break;
      case 32: ev.event=_EVENT_IRQ_TIME;break;
      default: ev.event = _EVENT_ERROR; break;
    }

    next = H(ev, tf);
    if (next == NULL) {
      next = tf;
    }
  }
```

do_event 根据事件再次分发 对于_EVENT_TRAP，输出提示信息，直接返回。

nanos-lite/src/irq.c

```
static _RegSet* do_event(_Event e, _RegSet* r) {

  switch (e.event) {
    case _EVENT_SYSCALL:
       do_syscall(r);
       //return schedule(r);
    case _EVENT_TRAP: return schedule(r);
      //printf("receive an event trap!!!");break;
    case _EVENT_IRQ_TIME:
      //Log("event:IRQ_TIME");
      return schedule(r);
    default: panic("Unhandled event ID = %d", e.event);
  }

  return NULL;
}
```

实现上下文切换

实现上下文切换

根据讲义的上述内容，实现以下功能：

- PTE 的_umake()函数
- Nanos-lite 的 schedule()函数，Nanos-lite 收到_EVENT_TRAP 事件后，调用 schedule()并返回其现场
- 修改 ASYE 中 asm_trap()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，从而完成上下文切换的本质操作

实现成功后，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了.

umake 函数，首先将_start()的三个参数和 eip 入栈，这里简单将参数和 eip 内容设置为 0 或 NULL 即可。 随后初始化陷阱帧，为通过 differentialtesting，初

始化 cs 为 8，eflags 为 2，并设置返回值 eip 为 entry。 最后，返回陷阱帧的指针，load_prog()将会将这一指针记录在用户进程 PCB 的 tf 中。

nexus-am/am/arch/x86-nemu/src/pte.c

```
_RegSet *_umake(_Protect *p, _Area ustack, _Area kstack, void *entry, char *const
argv[], char *const envp[]) {
 uint32_t *ptr = ustack.end;
  //navyapps程序入口函数_start的 栈帧，即8个通用寄存器
  for (int i = 0; i < 8; i++) {
        *ptr = 0x0;
         ptr--;
  }
  //陷阱帧，包括栈帧的8个通用寄存器
  *ptr = 0x02 | FL_IF;    ptr--; //eflags,即IF置1即可
  *ptr = 0x8;             ptr--; //cs 为了diff test
  *ptr = (uint32_t)entry; ptr--; //eip
  *ptr = 0x0;             ptr--; //error code
  *ptr = 0x81;            ptr--; //irq id
  for (int i = 0; i < 8; i++) {
        *ptr = 0x0;
         ptr--;
  }
  ptr++;
  return (_RegSet *)ptr; //将会记录到tf
}
```

schedule 函数用于进程调度。

schedule 做的事就是把上下文指针给保存在 current->tf(陷阱帧中)， 以便之后能够恢复之前的上下文信息，然后把当前上下文设置为 pcb[0]中的，最 后，调用_switch()函数完成上下文的切换。

```
_RegSet* schedule(_RegSet *prev) {
  //save the context pointer


  if(current!=NULL)
    current->tf = prev;
  else
    current=&pcb[current_game];
  if(count<400)
  {
  current = &pcb[current_game];
  count++;
  }
  else
  {
   current = &pcb[1];
   count=0;
  }
```

修改 asm_trap，我们在进行上下文切换时，本质上是把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，最后 iret 从异常中返回，执行另一个进程。

```
asm_trap:
  pushal

  pushl %esp
  call irq_handle

  addl $4, %esp
  movl %eax, %esp
  popal
  addl $8, %esp

  iret
```

分时多任务

分时运行仙剑奇侠传和 hello 程序

根据讲义的上述内容，添加相应的代码来实现仙剑奇侠传和 hello 程序之间的分时运行. 实现正确后，你会看到仙剑奇侠传一边运行的同时，hello 程序也会一边输出.

代码：nanos-lite/src/main.c

```
load_prog("/bin/pal");
load_prog("/bin/hello");
```

do_event()函数

设置处理完系统调用之后，并不返回原进程，而是进程调度切换进程

```
case _EVENT_SYSCALL:
   do_syscall(r);
   //return schedule(r);
```

优先级调度

优先级调度

我们可以修改 schedule() 的代码，来调整仙剑奇侠传和 hello 程序调度的频率比例，使得仙剑奇侠传调度若干次，才让 hello 程序调度 1 次. 这是因为 hello 程序做的事情只是不断地输出字符串，我们只需要让 hello 程序偶尔进行输出，以确认它还在运行就可以了.

设置频率比例，当仙剑奇侠传运行次数达到该频次时，切换运行一次 hello 程序

nanos-lite/src/proc.c

```
_RegSet* schedule(_RegSet *prev) {
  //save the context pointer

  if(current!=NULL)
    current->tf = prev;
  else
    current=&pcb[current_game];
  if(count<1000)
  {
  current = &pcb[current_game];
  count++;
  }
  else
  {
   current = &pcb[1];
   count=0;
  }
```

运行结果

PA4 阶段三

添加时钟中断

添加时钟中断

根据讲义的上述内容，添加相应的代码来实现真正的分时多任务. 为了证明时钟中断确实在工作，你可以在 Nanos-lite 收到_EVENT_IRQ_TIME 事件后用 Log()输出一句话.

需要注意的是，添加时钟中断之后，differential testing 机制就无法正确工作了. 这是因为，我们无法给 QEMU 注入时钟中断，无法保证 QEMU 与 NEMU 处于相同的状态. 不过，differential testing 作为一个强大的工具用到这时候，指令实现的正确性也基本上得到相当大的保证了.

按着指导一步一步来，cpu 结构中增加 INTR 引脚

然后 dev_raise_intr()设置 INTR 为高电平

在 exec_wrapper()末尾添加轮询 INTR 引脚的代码

```
#define TIME_IRQ 32
  if(cpu.INTR & cpu.eflags.IF)
  {
     cpu.INTR=false;
     extern void raise_intr(uint8_t NO,vaddr_t ret_addr);
     raise_intr(TIME_IRQ,cpu.eip);
     update_eip();
  }
}
```

修改 raise_intr()中的代码，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.

```
  memcpy(&t1,&cpu.eflags,sizeof(cpu.eflags));
  rtl_li(&t0,t1);//*t0 = t1
  rtl_push(&t0);
  cpu.eflags.IF=0;
  rtl_push(&cpu.cs);
  rtl_li(&t0,ret_addr);
  rtl_push(&t0);
```

软件处理时钟中断

在 ASYE 中添加时钟中断的支持,将时钟中断打包成_EVENT_IRQ_TIME 事件。

vectime 把中断号 0x32 压栈后，调用 asm_trap：

```
.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 $32; jmp asm_trap
```

irq_handler 识别出中断号为 32 后，将时钟中断打包成_EVENT_IRQ_TIME 事件。

```
_RegSet* irq_handle(_RegSet *tf) {
  _RegSet *next = tf;
  if (H) {
    _Event ev;
    switch (tf->irq) {
      case 0x80: ev.event = _EVENT_SYSCALL; break;
      case 0x81: ev.event = _EVENT_TRAP; break;
      case 32: ev.event=_EVENT_IRQ_TIME;break;
      default: ev.event = _EVENT_ERROR; break;
    }

    next = H(ev, tf);
    if (next == NULL) {
      next = tf;
    }
  }
```

为了可以让处理器在运行用户进程的时候响应时钟中断,你还需要修改 _umake()的代码,

必答题

必答题

请结合代码,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(Nanos-lite, AM, NEMU)中分时运行的.

(1) 分页机制保证了不同进程拥有独立的存储空间。为实现多进程分时运行，引入了虚拟内存的概念，让程序链接到固定的虚拟地址的同时，加载都不同的物理位置去执行，因此产生了分页机制。分页机制由 Nanos-lite、AM 和 NEMU 配合实现。对于仙剑奇侠传与 hello，Nanos-lite 通过 load_prog()实现用户程序的加载。load_prog 通过 AM 中提供的_protect()函数创建虚实地址映射；随后调用 loader() 加载程序，最后 load_prog()通过 umake()函数创建进程的上下文，为进程切换打下基础。

(2) 硬件中断与上下文切换保证程序的分时运行，NEMU 的 exec_wrapper 每执行完一条指令，当触发时钟中断时，将在 AM 中将时钟中断打包成 IRQ_TIME 事件， Nanos-lite 收到该事件后调用 schedule()进行进程调度。schedule()进行进程调度时，通过 AM 的_switch()切换进程的虚拟内存空间，并将进程的上下文传递给 AM，AM 的 asm_trap()恢复这一现场。NEMU 执行下一条指令时，便开始新进程的运行。

编写不朽的传奇

展示你的计算机系统

让 Nanos-lite 加载第 3 个用户程序/bin/videotest,并在 Nanos-lite 的 events_read() 函数中添加以下功能:当发现按下 F12 的时候,让游戏在仙剑奇侠传和 videotest 之间切换.为了实现这一功能,你还需要修改 schedule() 的代码:通过一个变量 current_game 来维护当前的游戏,在 current_game 和 hello 程序之间进行调度.例如,一开始是仙剑奇侠传和 hello 程序分时运行,按下 F12 之后,就变成 videotest 和 hello 程序分时运行.

代码实现:

nanos-lite/src/main.c

```
load_prog("/bin/pal");
load_prog("/bin/hello");
load_prog("/bin/videotest");
```

nanos-lite/src/proc.c，设置 current_game 维护当前运行游戏的进程号，提供接口函数 switch_current_game 进行切换

```
void switch_current_game()
{
   current_game=2-current_game;
   Log("current_game=%d",current_game);
}
```

schedule()在 current_game 与 hello 中切换：

```
_RegSet* schedule(_RegSet *prev) {
  //save the context pointer


  if(current!=NULL)
    current->tf = prev;
  else
    current=&pcb[current_game];
  if(count<1000)
  {
  current = &pcb[current_game];
  count++;
  }
  else
  {
   current = &pcb[1];
   count=0;
  }
  //TODO: switch to the new address space,
  //then return the new context
  _switch(&current->as);
  return current->tf;
```

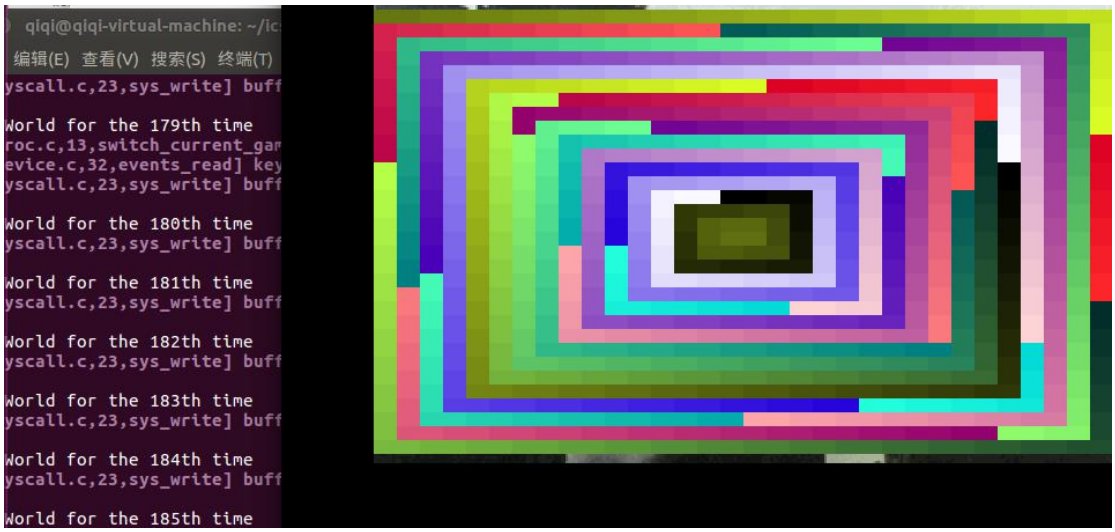
nanos-lite/src/device.c 当按下 F12 时，切换 current_game。则在下一次时钟中断触发进程调度时，变为另一 game 与 hello 分时运行

```
if(down && key==_KEY_F12)
{
  extern void switch_current_game();
  switch_current_game();
  Log("key down:_KEY_F12,switch current game");
}
```

运行结果：

开始仙剑奇侠传与 hello 程序分时运行：

按下 F12 后变为 videotest 与 hello 分时运行:



PA 实验感想

从 pa1 到 pa4 最终能成功运行仙剑奇侠传，中间经历了许多莫名奇妙的 bug，之前没有深入学习的一些课程，特别是编译原理，操作系统，计算机组成原理等，现在在 PA 中为此付出了惨痛的代价。非常感谢学姐的帮助和老师详细的实验指导，要是没有指导的话不知道自己能不能做出来（有指导也差点做不出来，留下了菜鸡的泪水。。。），PA 的实验可以说很好的将之前学的重要课程做了一个很好的贯通。转眼间大三已经结尾，由于疫情原因还没能面对面见一下老师，也是有

点小遗憾。最后希望 PA 这门课越来越好，能够帮助到越来越多的学弟学妹!