1 概述

1.1 实验目的

1. 充分理解虚拟内存机制实现方式，并在NEMU中实现i386分页机制
2. 实现从虚拟地址到物理地址的映射，让不同程序可以运行在独立的虚拟地址空间上
3. 充分理解上下文切换机制实现方式，将Nanos-lite完善为分时多任务操作系统
4. 充分理解中断机制实现方式，并在NEMU中添加时钟中断

1.2 实验内容

在PA3中凭借着Nanos-lite的支持以及系统调用和文件系统的实现，已经能够成功在NEMU上运行《仙剑奇侠传》，但此时的Nanos-lite只能同时支持单个程序的运行，是一个单任务操作系统，想要在Nanos-lite上同时运行多个程序暂时还是不可能的，因此为了将Nanos-lite完善为分时多任务操作系统，在PA4的实验中还需要在NEMU和AM中添加虚拟内存机制，使Nanos-lite上的程序可以运行在虚拟地址空间上；在AM中实现内核自陷并让Nanos-lite可以通过内核自陷来进行上下文切换以实现多个程序在Nanos-lite上的分时运行；在NEMU中添加时钟中断并让Nanos-lite通过时钟中断来进行进程切换。

阶段一：在NEMU中实现i386分页机制，在AM中实现虚拟内存空间中虚拟地址到物理地址的映射，并修改Nanos-lite程序加载模块（loader）使《仙剑奇侠传》可以在分页机制上运行

阶段二：在AM中添加内核自陷机制使得Nanos-lite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》，并可以实现《仙剑奇侠传》和`hello`程序之间的分时运行

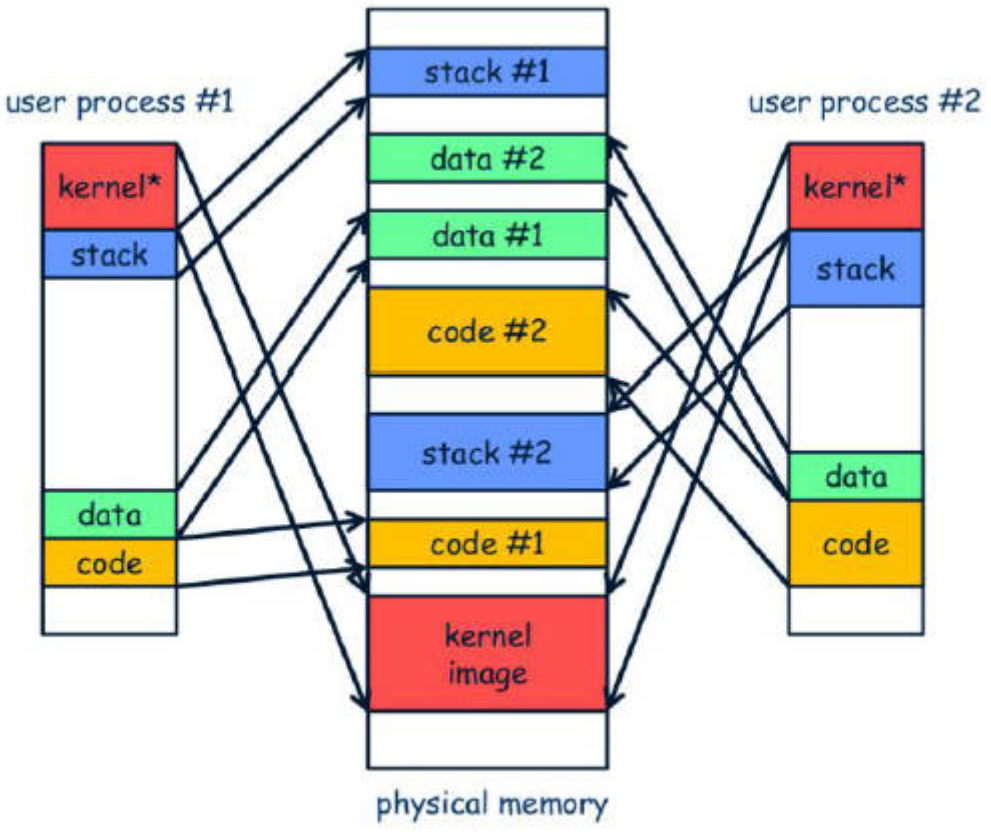
阶段三：在NEMU中实现硬件中断机制，并使Nanos-lite通过时钟中断来进行进程切换，最后添加手动切换进程的功能（在按下F12时，Nanos-lite上的进程在《仙剑奇侠传》和`videotest`间切换）

2 阶段一

通过Nanos-lite的支持以及PA3中系统调用和文件系统的实现，《仙剑奇侠传》已经可以成功地在NEMU上运行了，但现在的Nanos-lite仅能同时运行单个程序，如果想要在NEMU上运行不同的程序，就必须终止现有程序的运行，修改`nanos-lite/src/main.c`中加载的文件内容，再重新启动NEMU，原本运行的程序的状态也无法保存，这显然与我们对一个操作系统的期望有很大差距，为了能让Nanos-lite也能同时运行多个程序，首先要让不同的程序在运行时能够拥有独立的存储空间，否则程序的正确性就无法得到保证。在PA4中采用虚拟内存机制来解决这一问题，即在在真正的内存（物理内存）之上建立一层专门给程序使用的抽象，这样程序只需要认为自己运行在虚拟地址上，真正运行的时候，再把虚拟地址映射到物理地址，就可以同时满足在让程序认为自己在某个固定的内存位置和把程序加载到不同的内存位置去执行的需求。

引入了虚拟内存机制后，指令地址EIP就是一个虚拟地址了，因此需要在访问存储器之前完成从虚拟地址到物理地址的转换CPU才能取到正确的指令。但操作系统无法干涉指令的执行，必须在硬件实现从虚拟地址到物理地址的转换，即在处理器和存储器之间添加一个新的硬件模块MMU（Memory Management Unit，内存管理单元）来实现地址的映射，但只有操作系统才知道具体要把虚拟地址映射到哪些物理地址上。所以，虚拟内存机制是一个软硬协同才能生效的机制：操作系统负责进行物理内存的管理，加载程序的时候决定要把程序的虚拟地址映射到哪些物理地址；等到程序真正运行之前，还需要配置MMU，把之前决定好的映射落实到硬件上，程序运行的时候，MMU就会进行地址转换，把程序的虚拟地址映射到操作系统决定的物理地址上，程序就可以正常运行了。

NEMU中虚拟地址空间管理采用分页机制：把连续的存储空间分割成小片段，以这些小片段为单位进行组织，分配和管理，这些小片段称为页面，在虚拟地址空间和物理地址空间中也分别称为虚拟页和物理页。分页机制需要把虚拟页分别映射到相应的物理页上，这一映射通过“页表”的结构实现，页表中的每一个表项记录了一个虚拟页到物理页的映射关系，来把不必连续的页面重新组织成连续的虚拟地址空间。



NEMU中使用i386分页机制，采用二级页表结构，并且设计了专门的CR3（control register 3）寄存器来存放页目录的基地址。

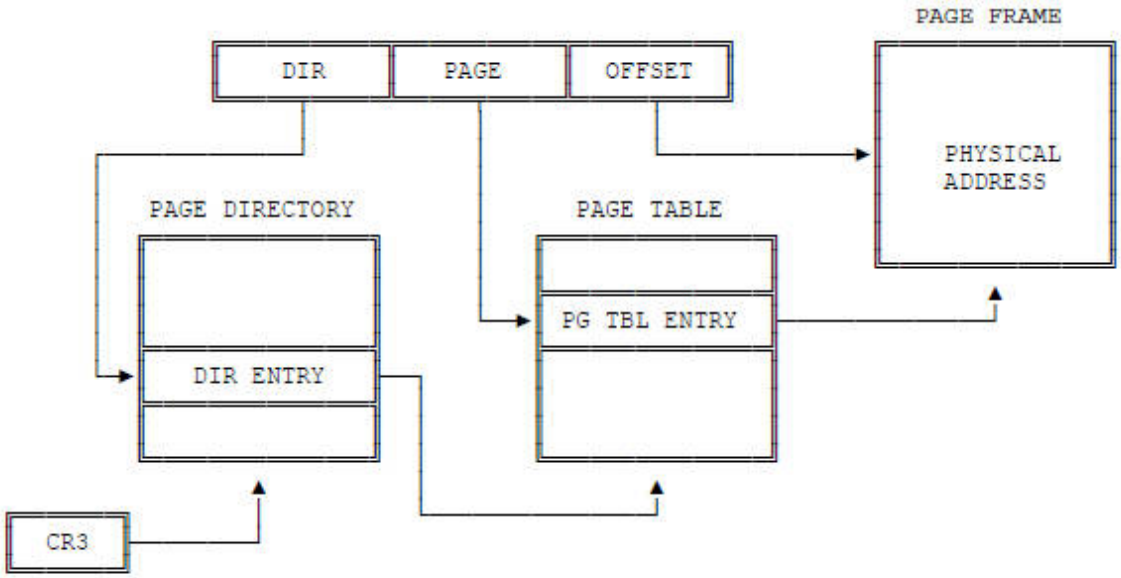
在阶段一中需要在NEMU和AM中添加PTE的支持，包括完善操作系统准备内核页表的工作，修改`nemu/src/memory/memory.c`目录下`vaddr_read()`和`vaddr_write()`函数以实现对虚拟地址的读写，修改Nanos-lite程序加载模块（loader）将用户程序加载到操作系统为其分配的虚拟地址空间，在AM中实现将虚拟地址空间中的虚拟地址映射到物理地址，最后再为堆区中新申请的空间建立到虚拟地址空间的映射，就可以成功在分页机制上运行仙剑奇侠传了。

2.1 实现方法及代码分析

在分页模式下，操作系统首先需要以物理页为单位对内存进行管理。每当加载程序的时候，就给程序分配相应的物理页（可以不连续），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。等到程序运行的时候，操作系统就把之前为这个程序填写好的页表设置到MMU中，MMU就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原本设置的物理地址空间。

NEMU中使用i386分页机制，它把物理内存划分成以4KB为单位的页面，同时采用了二级页表的结构。其中第一级页表称为“页目录”，第二级页表称为“页表”。每一张页目录和页表都有1024个表项，每个表项的大小都是4字节，除了包含页表（物理页）的基地址，还包含一些标志位信息来记录当前页表的状态。因此，一张页目录或页表的大小是4KB，无法存放在寄存器中，需要放在内存中。i386设计了专门的CR0（control register 0）寄存器来标记分页机制是否开启，CR3（control register 3）寄存器来存放页目录的基地址。页级地址转换就从CR3开始到页目录再到页表最后到真正的物理地址。

Figure 5-9. Page Translation

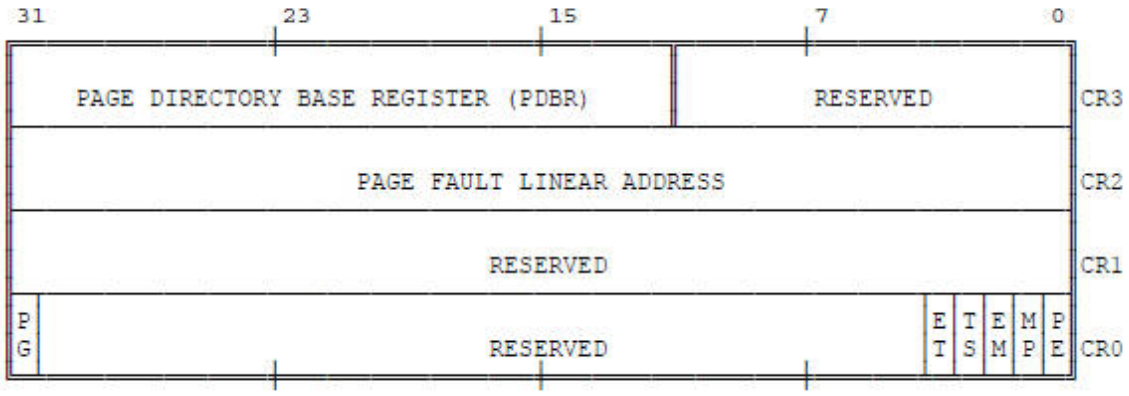


32位线性地址也被分为三部分，前10位记录在页目录内的偏移（DIR），用于检索页目录；中间10位记录在页表内的偏移（PAGE），用于检索页表；最后12位记录在物理页内的偏移（OFFSET），用于获取所对应的物理地址。

Figure 5-8. Format of a Linear Address

控制寄存器CR0（control register 0）与CR3（control register 3）结构：

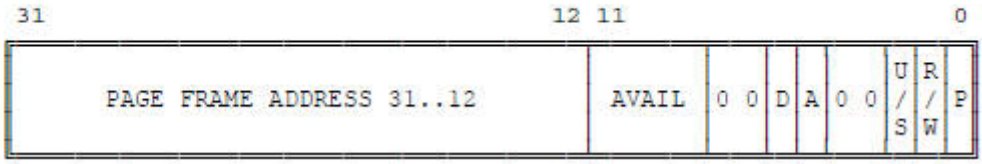
Figure 4-2. Control Registers



NEMU中只需要使用到CR0与CR3寄存器，CR0寄存器只需要关注PG位，PG=1时意味着开启分页机制；CR3是页目录基址寄存器，在开启分页机制后使用。前20位存放页目录的基地址（物理地址），后12位在NEMU中不必关心。

页目录PDE（Page Directory Entry）与页表PTE（Page Table Entry）结构：

Figure 5-10. Format of a Page Table Entry




```
P       - PRESENT
R/W     - READ/WRITE
U/S     - USER/SUPERVISOR
D       - DIRTY
AVAIL   - AVAILABLE FOR SYSTEMS PROGRAMMER USE

NOTE: 0 INDICATES INTEL RESERVED. DO NOT DEFINE.
```

因为4K对齐的原因只需要20位空间存储基地址信息，剩余空间中有一些为标志位，NEMU中使用到的标志位包括`present`位，`accessed`位和`dirty`位：

- `present`位表示物理页是否可用，不可用的时候分两种情况：
  - 物理页面由于交换技术被交换到磁盘中，需要通知操作系统内核将目标页面换回来才能继续执行
  - 进程试图访问一个未映射的线性地址，并没有实际的物理页与之相对应
- `accessed`位表示该物理页被访问（包括写操作和读操作）
- `dirty`位表示该物理页被写（仅PTE中包含此标志位）

到此NEMU在实现虚拟内存机制中所需的结构已全部说明，并且NEMU框架代码中已经实现了CR0，CR3，PTE和PDE（在`nemu/include/memory/mmu.h`文件中），在PA4中NEMU部分实现虚拟内存机制时只需要包含这一头文件并直接使用定义好的联合体即可。

```
/* the Control Register 0 */
typedef union CR0 {
  struct {
    uint32_t protect_enable      : 1;
    uint32_t dont_care           : 30;
    uint32_t paging              : 1;
  };
  uint32_t val;
```

```
} CR0;

/* the Control Register 3 (physical address of page directory) */
typedef union CR3 {
  struct {
    uint32_t pad0                : 3;
    uint32_t page_write_through  : 1;
    uint32_t page_cache_disable  : 1;
    uint32_t pad1                : 7;
    uint32_t page_directory_base : 20;
  };
  uint32_t val;
} CR3;


/* the 32bit Page Directory(first level page table) data structure */
typedef union PageDirectoryEntry {
  struct {
    uint32_t present             : 1;
    uint32_t read_write          : 1;
    uint32_t user_supervisor     : 1;
    uint32_t page_write_through  : 1;
    uint32_t page_cache_disable  : 1;
    uint32_t accessed            : 1;
    uint32_t pad0                : 6;
    uint32_t page_frame          : 20;
  };
  uint32_t val;
} PDE;

/* the 32bit Page Table Entry(second level page table) data structure */
typedef union PageTableEntry {
  struct {
    uint32_t present             : 1;
    uint32_t read_write          : 1;
    uint32_t user_supervisor     : 1;
    uint32_t page_write_through  : 1;
    uint32_t page_cache_disable  : 1;
    uint32_t accessed            : 1;
    uint32_t dirty               : 1;
    uint32_t pad0                : 1;
    uint32_t global              : 1;
    uint32_t pad1                : 3;
    uint32_t page_frame          : 20;
  };
  uint32_t val;
} PTE;
```

2.1.1 准备内核页表

由于页表位于内存中，但计算机启动时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。而是由操作系统来启动分页机制，这需要先需要准备一些内核页表。框架代码已经实现了这一功能（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就会在初始化的时候调用`init_mm()`函数(在`nanos-lite/src/mm.c`中定义)来初始化MM（Memory Manager，存储管理器）模块，由MM专门负责分页相关的存储管理。

初始化MM的工作有两项：

- 将TRM提供的堆区起始地址作为空闲物理页的首地址，在之后的程序运行过程中会通过`new_page()`函数来分配空闲的物理页
- 调用AM的`_pte_init()`函数，填写内核的页目录和页表，然后设置CR3寄存器，最后通过设置CR0寄存器来开启分页机制。

```
//nanos-lite/src/main.c
int main() {
#ifdef HAS_PTE
  init_mm();
#endif

  ......
}

//nanos-lite/src/mm.c
void init_mm() {
  //将堆区起始地址作为空闲物理页的首地址，并且按页对齐
  pf = (void *)PGROUNDUP((uintptr_t)_heap.start);
  Log("free physical pages starting from %p", pf);

  _pte_init(new_page, free_page); //调用_pte_init()
}

//nexus-am/am/arch/x86-nemu/src/pte.c
void _pte_init(void* (*palloc)(), void (*pfree)(void*)) {
  palloc_f = palloc;
  pfree_f = pfree;

  int i;

  // make all PDEs invalid
  for (i = 0; i < NR_PDE; i ++) {
    kpdirs[i] = 0;
  }

  //填写内核的页目录和页表
  PTE *ptab = kptabs;
  for (i = 0; i < NR_KSEG_MAP; i ++) {
    uint32_t pdir_idx = (uintptr_t)segments[i].start / (PGSIZE * NR_PTE);
    uint32_t pdir_idx_end = (uintptr_t)segments[i].end / (PGSIZE * NR_PTE);
    for (; pdir_idx < pdir_idx_end; pdir_idx ++) {
      // fill PDE
      kpdirs[pdir_idx] = (uintptr_t)ptab | PTE_P;

      // fill PTE
      PTE pte = PGADDR(pdir_idx, 0, 0) | PTE_P;
      PTE pte_end = PGADDR(pdir_idx + 1, 0, 0) | PTE_P;
      for (; pte < pte_end; pte += PGSIZE) {
        *ptab = pte;
        ptab ++;
      }
    }
  }

  set_cr3(kpdirs); //设置CR3寄存器
```

```
  set_cr0(get_cr0() | CR0_PG); //设置CR0寄存器来开启分页机制
}
```

这样以后，Nanos-lite就运行在分页机制之上了。调用`_pte_init()`函数的时候还需要提供物理页的分配和回收两个回调函数，用于在AM中获取/释放物理页。NEMU中为了简化实现，采用顺序的方式对物理页进行分配，而且分配后无需回收。

为了在NEMU中实现分页机制，首先需要在CPU中添加CR0寄存器和CR3寄存器（`nemu/include/cpu/reg.h`），

```
#include "memory/mmu.h"

typedef struct {

  ......

  CR0 cr0;
  CR3 cr3;

} CPU_state;
```

其次需要添加操作CR0寄存器和CR3寄存器的指令，查阅i386手册可知，需要实现的指令为0F 20和0F 22处的MOV指令。

MOV — Move to/from Special Registers

Opcode	Instruction	Clocks	Description
0F 20 /r	MOV r32,CR0/CR2/CR3	6	Move (control register) to (register)
0F 22 /r	MOV CR0/CR2/CR3,r32	10/4/5	Move (register) to (control register)
0F 21 /r	MOV r32,DR0 -- 3	22	Move (debug register) to (register)
0F 21 /r	MOV r32,DR6/DR7	14	Move (debug register) to (register)
0F 23 /r	MOV DR0 -- 3,r32	22	Move (register) to (debug register)
0F 23 /r	MOV DR6/DR7,r32	16	Move (register) to (debug register)
0F 24 /r	MOV r32,TR6/TR7	12	Move (test register) to (register)
0F 26 /r	MOV TR6/TR7,r32	12	Move (register) to (test register)

Operation

DEST ← SRC;

Description

The above forms of MOV store or load the following special registers in or from a general purpose register:

- Control registers CR0, CR2, and CR3
- Debug Registers DR0, DR1, DR2, DR3, DR6, and DR7
- Test Registers TR6 and TR7

32-bit operands are always used with these instructions, regardless of the operand-size attribute.

0F 20处MOV指令格式为MOV r32，CR0/CR2/CR3，译码函数选择`make_Dhelper(G2E)`，操作数宽度为默认的4，执行函数选择`make_EHelper(mov_cr2r)`，实现对应的执行函数并在`all-instr.h`文件中添加执行函数`make_EHelper(mov_cr2r)`的声明，

```
/* Eb <- Gb
```

```
 * Ev <- Gv
 */
make_DHelper(G2E) {
  decode_op_rm(eip, id_dest, true, id_src, true);
}

make_EHelper(mov_cr2r) {
  switch(id_src->reg) //判断操作的CR寄存器
  {
    case 0: //若为CR0
      operand_write(id_dest, &cpu.cr0.val); //dest<-CR0
      break;
    case 3: //若为CR3
      operand_write(id_dest, &cpu.cr3.val); ////dest<-CR3
      break;
    default: //否则报错并终止程序运行
      panic("error in make_Ehelper(mov_cr2r), cr%d", id_src->reg);
      break;
  }
  print_asm("movl %%cr%d,%%%s", id_src->reg, reg_name(id_dest->reg, 4));

#ifdef DIFF_TEST
  diff_test_skip_qemu();
#endif
}

//all-instr.h
make_EHelper(mov_cr2r);
```

指令打包为`IDEX(G2E, mov_cr2r)`，将打包好的指令填入到`opcode_table`的对应位置。

```
/*2 byte_opcode_table */

  /* 0x20 */    IDEX(G2E, mov_cr2r), EMPTY, IDEX(E2G, mov_r2cr), EMPTY,
```

0F 22处MOV指令格式为MOV CR0/CR2/CR3，r32，译码函数选择`make_Dhelper(E2G)`，操作数宽度为默认的4，执行函数选择`make_EHelper(mov_r2cr)`，实现对应的执行函数并在`all-instr.h`文件中添加执行函数`make_EHelper(mov_r2cr)`的声明，

```
/* Gb <- Eb
 * Gv <- Ev
 */
make_DHelper(E2G) {
  decode_op_rm(eip, id_src, true, id_dest, true);
}

make_EHelper(mov_r2cr) {
  switch(id_dest->reg) //判断操作的CR寄存器
  {
    case 0: //若为CR0
      cpu.cr0.val = id_src->val; //CR0<-dest
      break;
    case 3: //若为CR3
      cpu.cr3.val = id_src->val; //CR3<-dest
      break;
    default: //否则报错并终止程序运行
```

```
      panic("error in make_Ehelper(mov_r2cr), cr%d", id_dest->reg);
      break;
  }

  print_asm("movl %%%s,%%cr%d", reg_name(id_src->reg, 4), id_dest->reg);
}

//all-instr.h
make_EHelper(mov_r2cr);
```

指令打包为`IDEX(E2G, mov_r2cr)`，将打包好的指令填入到`opcode_table`的对应位置。

```
/*2 byte_opcode_table */

  /* 0x20 */    IDEX(G2E, mov_cr2r), EMPTY, IDEX(E2G, mov_r2cr), EMPTY,
```

为了使differential testing机制能够正确工作，还需要在`restart()`函数（`nemu/src/monitor/monitor.c`）中将CR0寄存器初始化为0x60000011，

```
#define CR0_START 0x60000011

static inline void restart() {
  ......
  cpu.cr0.val = CR0_START;

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

对`vaddr_read()`和`vaddr_write()`函数（`nemu/src/memory/memory.c`）进行修改以真正实现对虚拟地址的读写，编写`page_translate()`函数来实现分页地址转换的过程，PA4中不需要实现保护机制，因此在`page_translate()`函数的实现中需要检查页目录项和页表项的`present`位，如果发现了一个
无效的表项就需要及时终止NEMU的运行。并且确保必须进入保护模式并开启分页机制之后NEMU才会进行页级地址转换。为了让differential testing机制正确工作，需要实现分页机制中`accessed`位和`dirty`位的功能。

此外还需要注意可能会出现的数据跨越虚拟页边界的情况，当出现数据跨越虚拟页边界情况时要进行特殊处理，分两次进行读取和写入。

```
#include "memory/mmu.h"

//声明page_translate()函数，其中形参bool write用于传递是否为写入操作
paddr_t page_translate(vaddr_t addr, bool write);

uint32_t vaddr_read(vaddr_t addr, int len) {
  if ((addr & 0xfffff000) != ((addr + len - 1) & 0xfffff000)) //如果数据跨越虚拟页边界
  {
    //panic("data cross page boundary(in vaddr_read())");
    int first_pg_len = 0x1000 - (int)((uint32_t)addr & 0xfff); //第一页字节数
    int second_pg_len = len - first_pg_len; //第二页字节数
    uint32_t data = 0;
```

```
    paddr_t first_paddr = page_translate(addr, false); //第一页地址转换
    data = paddr_read(first_paddr, first_pg_len); //读取第一页数据

    vaddr_t second_vaddr = addr + first_pg_len; //计算第二页虚拟地址首地址
    paddr_t second_paddr = page_translate(second_vaddr, false); //第二页地址转换
    data = data | (paddr_read(second_paddr, second_pg_len) << (first_pg_len <<
3)); //读取第二页数据并将结果组合

    return data;
  }
  else //否则只需进行单页的地址转换和数据读取
  {
    //printf("vaddr = 0x%08x(vaddr_read()) ", addr);
    paddr_t paddr = page_translate(addr, false);
    //printf("paddr = 0x%08x(vaddr_read())\n", paddr);
    return paddr_read(paddr, len);
  }
  //return paddr_read(addr, len);
}

void vaddr_write(vaddr_t addr, int len, uint32_t data) {
  if ((addr & 0xfffff000) != ((addr + len - 1) & 0xfffff000)) //如果数据跨越虚拟页边
界
  {
    //panic("data cross page boundary(in vaddr_write())");
    int first_pg_len = 0x1000 - (int)(addr & 0xfff); //第一页字节数
    int second_pg_len = len - first_pg_len; //第二页字节数

    uint32_t first_pg_data = (data << (second_pg_len << 3)) >> (second_pg_len <<
3); //将需要写入的数据拆分，这里拆出写入第一页的数据
    paddr_t first_paddr = page_translate(addr, true);  //第一页地址转换
    paddr_write(first_paddr, first_pg_len, first_pg_data); //将数据写入第一页

    uint32_t second_pg_data = data >> (first_pg_len << 3); //拆分出写入第二页的数据
    vaddr_t second_vaddr = addr + first_pg_len; //计算第二页虚拟地址首地址
    paddr_t second_paddr = page_translate(second_vaddr, true); //第二页地址转换
    paddr_write(second_paddr, second_pg_len, second_pg_data); //将数据写入第二页
  }
  else //否则只需进行单页的地址转换和数据写入
  {
    //printf("vaddr = 0x%08x(vaddr_write()) ", addr);
    paddr_t paddr = page_translate(addr, true);
    //printf("paddr = 0x%08x(vaddr_write())\n", paddr);
    paddr_write(paddr, len, data);
  }
  //paddr_write(addr, len, data);
}

paddr_t page_translate(vaddr_t addr, bool write)
{
  //在进入保护模式并开启分页机制之后进行页级地址转换
  if (cpu.cr0.protect_enable && cpu.cr0.paging)
  {
    PDE pde;
    PTE pte;

    //从CR3寄存器获取PDE基址
    PDE *pg_dic_base = (PDE*)((uint32_t)cpu.cr3.val & 0xfffff000);
```

```
    //获取线性地址前10位（页目录内的偏移，DIR）
    uint32_t pg_dic_index = ((uint32_t)addr >> 22) & 0x3ff;
    //获取对应的页目录PDE
    pde.val = paddr_read((paddr_t) &pg_dic_base[pg_dic_index], 4);
    //检查该PDEpresent位是否为0，若表项无效报错并终止NEMU运行
    Assert(pde.present, "pde.present = 0");
    pde.accessed = 1; //该PDEaccessed位置为1

    //从PDE获取PTE基址
    PTE *pg_tab_base = (PTE*)((uint32_t)pde.val & 0xfffff000);
    //获取线性地址中间10位（页表内的偏移，PAGE）
    uint32_t pg_tab_index = ((uint32_t)addr >> 12) & 0x3ff;
    //获取对应的页表PTE
    pte.val = paddr_read((paddr_t) &pg_tab_base[pg_tab_index], 4);
    //检查该PTEpresent位是否为0，若表项无效报错并终止NEMU运行
    Assert(pte.present, "pte.present = 0");
    pte.accessed = 1; //该PTEaccessed位置为1

    if(write)
      pte.dirty = 1; //如果是写入操作将该PTEdirty位置为1
    //将对应PTE与线性地址最后12位（物理页内的偏移，OFFSET）组合获得对应的物理地址
    paddr_t paddr = ((uint32_t)pte.val & 0xfffff000) | ((uint32_t)addr & 0xfff);
    return paddr;
  }
  else //否则不进行页级地址转换
    return addr;
}
```

2.1.2 让用户程序运行在分页机制上

实现了分页机制后，尽管《仙剑奇侠传》已经可以运行在虚拟地址空间之中了，但因为在`_asye_init()`中创建了内核的虚拟地址空间之后就再也没有进行过切换，所以《仙剑奇侠传》运行在内核的虚拟内存空间上，但实际上用户程序应该运行在操作系统为其分配的虚拟地址空间之上，为了实现这一目的，首先需要将`navy-apps/Makefile.compile`中的链接地址`-Ttext`参数改为`0x8048000`，以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。

```
  LDFLAGS += -Ttext 0x8048000
```

同时修改`nanos-lite/src/loader.c`中的DEFAULT_ENTRY为`0x8048000`，

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

然后，让Nanos-lite通过`load_prog()`函数（`nanos-lite/src/proc.c`中定义）来进行用户程序的加载，

```
int main() {
  ......

  //uint32_t entry = loader(NULL, "/bin/pal");
  //((void (*)(void))entry)();
  load_prog("/bin/pal");
  ......
}
```

`load_prog()`函数首先会通过`_protect()`函数（在`nexus-am/am/arch/x86-nemu/src/pte.c`中定义）创建一个用户进程的虚拟地址空间，然后`load_prog()`会调用`loader()`函数加载用户程序（调用`_protect()`的时候使用的PCB的结构体指进程控制块（PCB，process control block），在阶段二中会详细说明，此处只需要知道虚拟地址空间的信息存放它的`as`成员中即可）。此时`loader()`需要获取用户程序的大小之后，以页为单位进行加载：

- 申请一页空闲的物理页
- 把这一物理页映射到用户程序的虚拟地址空间中
- 从文件中读入一页的内容到这一物理页上

所以为了`loader()`能够加载用户进程并为用户进程准备映射，所以需要实现将一物理页映射到用户进程的虚拟地址空间的功能，即实现在AM中的`_map()`函数（`nexus-am/am/arch/x86-nemu/src/pte.c`中定义），功能是将虚拟地址空间p中的虚拟地址va映射到物理地址pa，因为AM中已经提供了相应的宏来操作PDE和PTE（在`nexus-am/am/arch/x86-nemu/include/x86.h`文件中），在实现`_map()`函数时直接使用即可。

```
//nexus-am/am/arch/x86-nemu/include/x86.h
// Page directory and page table constants
#define NR_PDE    1024    // # directory entries per page directory
#define NR_PTE    1024    // # PTEs per page table
#define PGSHFT    12      // log2(PGSIZE)
#define PTXSHFT   12      // Offset of PTX in a linear address
#define PDXSHFT   22      // Offset of PDX in a linear address

// Page table/directory entry flags
#define PTE_P     0x001     // Present
#define PTE_W     0x002     // Writeable
#define PTE_U     0x004     // User
#define PTE_PWT   0x008     // Write-Through
#define PTE_PCD   0x010     // Cache-Disable
#define PTE_A     0x020     // Accessed
#define PTE_D     0x040     // Dirty

// +--------10------+-------10-------+---------12----------+
// | Page Directory |   Page Table   | Offset within Page  |
// |      Index     |      Index     |                     |
// +----------------+----------------+---------------------+
//  \--- PDX(va) --/ \--- PTX(va) --/\------ OFF(va) ------/
typedef uint32_t PTE;
typedef uint32_t PDE;
#define PDX(va)     (((uint32_t)(va) >> PDXSHFT) & 0x3ff)
#define PTX(va)     (((uint32_t)(va) >> PTXSHFT) & 0x3ff)
#define OFF(va)     ((uint32_t)(va) & 0xfff)

// construct virtual address from indexes and offset
#define PGADDR(d, t, o) ((uint32_t)((d) << PDXSHFT | (t) << PTXSHFT | (o)))

// Address in page table or page directory entry
#define PTE_ADDR(pte)   ((uint32_t)(pte) & ~0xfff)

//nexus-am/am/arch/x86-nemu/src/pte.c
void _map(_Protect *p, void *va, void *pa) {
  PDE *pg_dic_base = (PDE*)(p->ptr); //获取PDE基址
  uint32_t pg_dic_index = PDX(va); //从va中获取页目录内的偏移，DIR
  uint32_t pg_tab_index = PTX(va); //从va中获取页表内的偏移，PAGE
```

```
  if (!(pg_dic_base[pg_dic_index] & PTE_P)) //如果当前页表无效
  {
    //通过回调函数palloc_f()向Nanos-lite获取一页空闲的物理页
    PTE *pg_tab_base =  (PTE*)palloc_f();
    pg_dic_base[pg_dic_index] = (PDE)pg_tab_base | PTE_P;
    pg_tab_base = (PTE*)PTE_ADDR(pg_dic_base[pg_dic_index]); //获取页表首地址
    pg_tab_base[pg_tab_index] = (PTE)pa | PTE_P; //添加从页表到物理页的映射
  }
  else //否则直接建立映射即可
  {
    PTE *pg_tab_base = (PTE*)PTE_ADDR(pg_dic_base[pg_dic_index]); //获取页表首地址
    pg_tab_base[pg_tab_index] = (PTE)pa | PTE_P; //添加从页表到物理页的映射
  }
  //pg_tab_base[pg_tab_index] = (PTE)pa | PTE_P;
}
```

然后根据上述需求修改`loader()`函数，申请空闲的物理页需要使用的`new_page()`函数通过包含其头文件`memory.h`来使用。。

```
#include "memory.h"

extern void _map(_Protect *p, void *va, void *pa);

uintptr_t loader(_Protect *as, const char *filename) {
  //size_t rsize = get_ramdisk_size();
  //ramdisk_read(DEFAULT_ENTRY, 0, rsize);

  int fd = fs_open(filename, 0, 0);
  Log("loader fd:%d",fd);
  size_t rsize = fs_filesz(fd);
  //fs_read(fd, DEFAULT_ENTRY, rsize);
  //fs_close(fd);

  int page_num = rsize / PGSIZE; //计算需要的物理页数
  int last_bytes = rsize % PGSIZE; //计算是否能刚好分配到page_num个物理页上
  void *page;
  int i;
  for(i = 0; i < page_num; i++)
  {
    page = new_page(); //申请一页空闲的物理页
    //把这一物理页映射到用户程序的虚拟地址空间中
    _map(as, DEFAULT_ENTRY + i * PGSIZE, page);
    fs_read(fd, page, PGSIZE); //从文件中读入一页的内容到这一物理页上
  }
  if(last_bytes != 0) //如果还有剩余的字节，则再申请一页空闲的物理页并重复以上操作
  {
    page = new_page();
    _map(as, DEFAULT_ENTRY + i * PGSIZE, page);
    fs_read(fd, page, last_bytes);
  }
  fs_close(fd);

  return (uintptr_t)DEFAULT_ENTRY;
}
```

从`loader()`返回后，`load_prog()`会调用`_switch()`函数（在`nexus-am/am/arch/x86-nemu/src/pte.c`中定义），切换到刚才为用户程序创建的地址空间。最后跳转到用户程序的入口，这样用户程序就能运行在分页机制上了。

```
//nexus-am/am/arch/x86-nemu/src/pte.c
void _switch(_Protect *p) {
  set_cr3(p->ptr);
}

//nanos-lite/src/proc.c
void load_prog(const char *filename) {
  int i = nr_proc ++; //新进程创建在现有所有进程之后
  _protect(&pcb[i].as); //创建一个用户进程的虚拟地址空间

  uintptr_t entry = loader(&pcb[i].as, filename); //加载用户程序

  // TODO: remove the following three lines after you have implemented _umake()
  _switch(&pcb[i].as); //切换到刚才为用户程序创建的地址空间
  current = &pcb[i];
  ((void (*)(void))entry)(); //跳转到用户程序的入口

  _Area stack;
  stack.start = pcb[i].stack;
  stack.end = stack.start + sizeof(pcb[i].stack);

  pcb[i].tf = _umake(&pcb[i].as, stack, stack, (void *)entry, NULL, NULL);
}
```

但为了能让《仙剑奇侠传》也能够在分页机制上正确运行，还需要修改原本直接返回0的`mm_brk()`函数来把新申请的堆区映射到虚拟地址空间中，不需要实现堆区的回收功能，并且在实现时还需要注意按页对齐的问题。这里可以使用`memory.h`文件中定义的宏来简化实现，

```
//nanos-lite/include/memory.h
#ifndef PGSIZE
#define PGSIZE 4096
#endif

#define PGMASK          (PGSIZE - 1)    // Mask for bit ops
#define PGROUNDUP(sz)   (((sz)+PGSIZE-1) & ~PGMASK)
#define PGROUNDDOWN(a)  (((a)) & ~PGMASK)

//nanos-lite/src/mm.c
/* The brk() system call handler. */
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)
    {
      //获取需要新增映射的第一个页面的虚拟地址首地址
      uint32_t first_page = PGROUNDUP(current->max_brk);
      //获取需要新增映射的最后一个页面的虚拟地址首地址
      uint32_t last_page = PGROUNDDOWN(new_brk);
```

```
      if((new_brk & 0xfff) == 0) //如果new_brk页对齐，则需要新增映射的页面数减一
        last_page -= PGSIZE;
      //按顺序申请新的物理页并为其建立映射
      for(uint32_t vaddr = first_page; vaddr <= last_page; vaddr += PGSIZE)
        _map(&current->as, (void *)vaddr, new_page());

      current->max_brk = new_brk; //更新current->max_brk为new_brk
    }
    current->cur_brk = new_brk;
  }
  return 0;
}
```

同时原本直接返回0的SYS_brk系统调用也需要做相应的修改。

```
    case SYS_brk:
          //a[0] = 0;
          a[0] = mm_brk((uint32_t)a[1]);
          break;
```

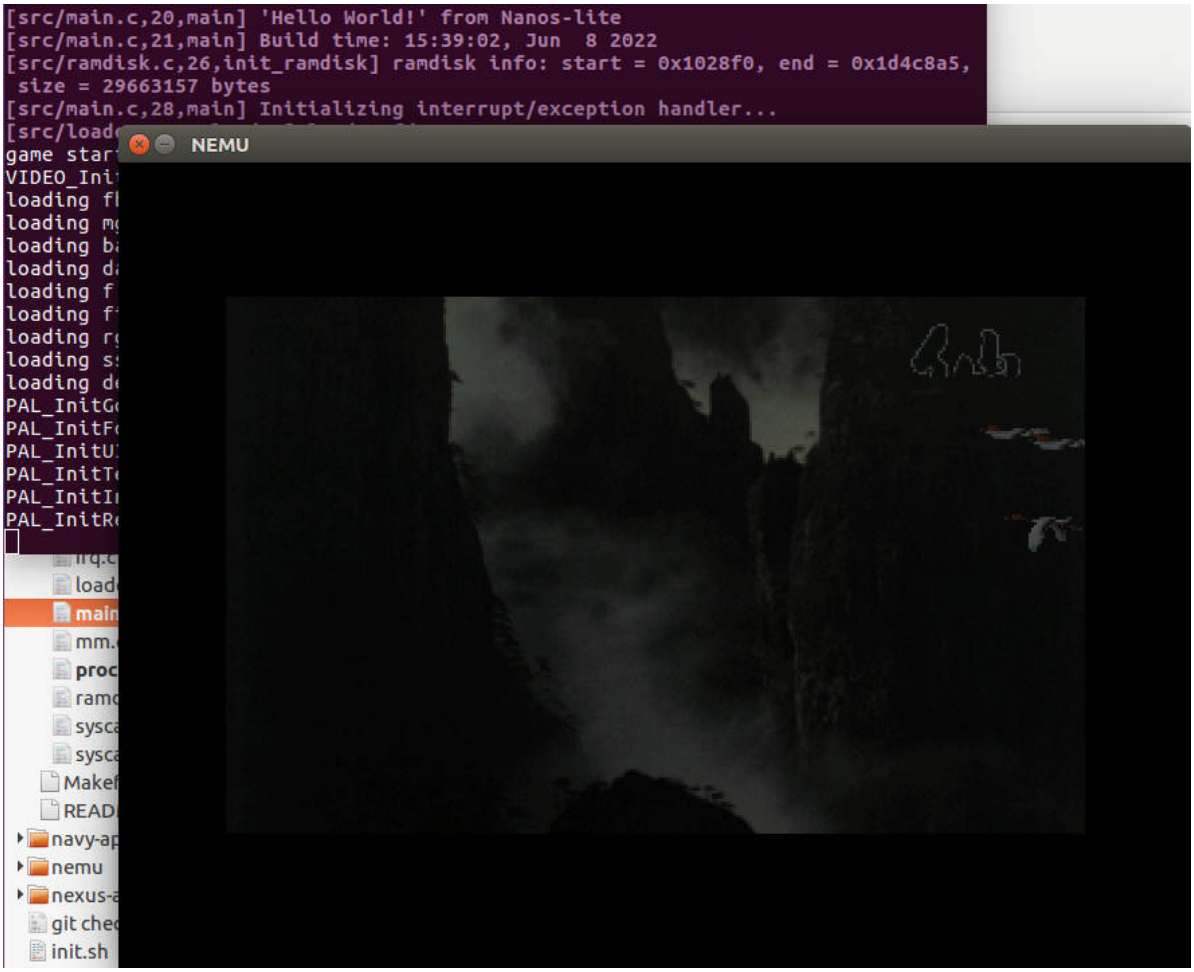
这样《仙剑奇侠传》就可以正确在分页机制上运行了。

2.2 运行结果

在分页机制下运行`dummy`。

```
[src/monitor/monitor.c,68,load_img] The image is /home/miaozeyun/ics2017/nanos-l
ite/build/nanos-lite-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,33,welcome] Build time: 15:29:38, Jun  8 2022
For help, type "help"
(nemu) c
[src/mm.c,43,init_mm] free physical pages starting from 0x1d91000
[src/main.c,20,main] 'Hello World!' from Nanos-lite
[src/main.c,21,main] Build time: 15:33:51, Jun  8 2022
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x1028f0, end = 0x1d4c8a5,
 size = 29663157 bytes
[src/main.c,28,main] Initializing interrupt/exception handler...
[src/loader.c,18,loader] loader fd:11
nemu: HIT GOOD TRAP at eip = 0x00100032

(nemu)
```

在分页机制下运行《仙剑奇侠传》。



2.3 Bug总结

阶段一的遇到的Bug比较少，并且PA实验流程中对于分页机制的描述已经非常清楚地讲明了关键点，再配合查询i386手册，分页机制的实现方式就能够完全理解了，并且PA实验流程中也对所有简化的部分做了详细的说明。在最开始实现`page_translate()`函数的时候对线性地址的位数操作出了问题，不过很快就排查到了（到AM部分实现`_map()`函数时才发现AM中有实现好的宏可以拿来用）；其次就是对`mm_brk()`函数做修改的时候要注意如果`new_brk`页对齐，那么需要新增映射的页面数就要减一；还有就是不要忘记修改SYS_brk系统调用，否则Nanos-lite还是无法把新申请的堆区映射到虚拟地址空间中。

3 阶段二

在阶段二中需要加入上下文切换的机制来让Nanos-lite成为一个真正的分时多任务操作系统，具体的实现方式与PA3中使用的的陷阱帧非常类似，当程序A运行的过程中触发了系统调用，陷入到内核，A的陷阱帧将会被保存到A的堆栈上，这时如果栈顶指针切换到另一个程序B的堆栈上并将B的现场恢复，那么接下来运行的就是程序B。所以，上下文切换其实就是不同程序之间的堆栈切换。对于刚刚加载完的程序，则需要在程序的堆栈上人工初始化一个陷阱帧使得将来切换的时候可以根据这个人工陷阱帧来正确地恢复现场。

在阶段二中需要实现上下文切换，包括在用户程序的堆栈上初始化陷阱帧和添加内核自陷并包装为_EVENT_TRAP事件使得Nanos-lite可以通过内核自陷来启动第一个用户程序；实现分时多任务，让Nanos-lite可以在系统调用后进行进程切换。

3.1 实现方法及代码分析

3.1.1 上下文切换与内核自陷

为了方便对进程进行管理，Nanos-lite使用了进程控制块（PCB,process control block）的数据结构来记录进程相关信息，包括陷阱帧的位置指针、虚拟地址空间以及用户进程堆区的位置，并且为每一个进程维护一个单独的PCB。Nanos-lite的框架代码中已经定义了所使用的PCB结构：

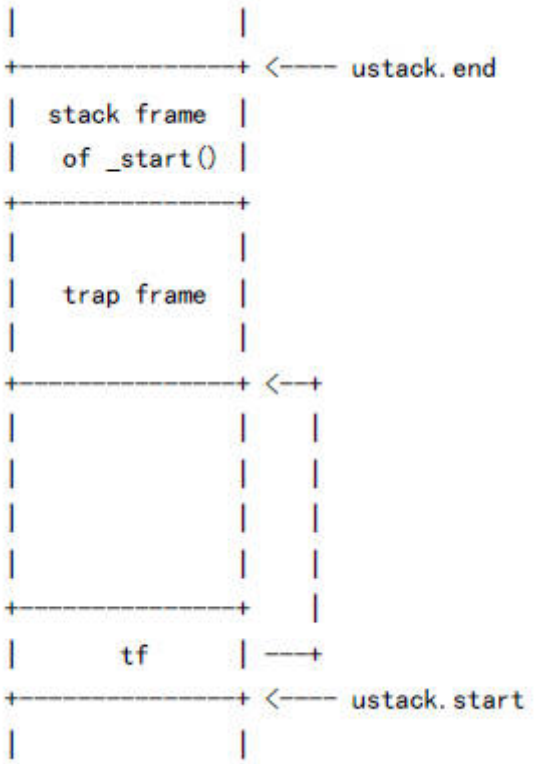
```
//nanos-lite/include/proc.h

typedef union {
  uint8_t stack[STACK_SIZE] PG_ALIGN;
  struct {
    _RegSet *tf; //陷阱帧的位置指针
    _Protect as; //进程所处虚拟地址空间
    uintptr_t cur_brk; //进程堆区的位置
    // we do not free memory, so use `max_brk' to determine when to call _map()
    uintptr_t max_brk;
  };
} PCB;
```

在进行上下文切换的时候，只需要把PCB中的`tf`指针返回给ASYE的`irq_handle()`函数即可。

为了能使刚加载完的程序能够正常运行，需要在用户进程的堆栈上初始化一个陷阱帧，这项工作通过PTE提供的`_umake()`函数（`nexus-am/am/arch/x86-nemu/src/pte.c`中定义）来实现，其原型为：

```
_RegSet *_umake(_Protect *p, _Area ustack, _Area kstack,
  void *entry, char *const argv[], char *const envp[]);
```

因为Nanos-lite对用户进程作了一些简化，所以`_umake()`函数只需要在`ustack`的底部初始化一个以`entry`为返回地址的陷阱帧，又因为Navy-apps中程序的入口函数是`navy-apps/libs/libc/src/start.c`中的`_start()`函数，因此还需要在陷阱帧之前设置好`_start()`函数的栈帧，确保`_start()`可以正确执行。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。



同时为了保证differential testing的正确运行，人为初始化的陷阱帧中的cs需要设置为8。

```
_RegSet *_umake(_Protect *p, _Area ustack, _Area kstack, void *entry, char
*const argv[], char *const envp[]) {
  uint32_t *stack_pointer = (uint32_t*)(ustack.end); //将指针指向ustack的底部
  //将_start()的三个参数和eip入栈，这里都设置为0
  for(int i = 0; i < 4; i++)
    *(stack_pointer--) = 0x0;

  //人为构建陷阱帧，其中eflags设置为2，cs设置为8
  //irq设置为0x81，因为是通过内核自陷的方式来触发第一次上下文切换，具体机制在接下来的报告中说明
  *(stack_pointer--) = 0x2; //eflags
  *(stack_pointer--) = 0x8; //cs
  *(stack_pointer--) = (uint32_t)entry; //eip
  *(stack_pointer--) = 0x0; //error_code
  *(stack_pointer--) = 0x81; //irq

  for(int i = 0; i < 8; i++) //eax->edi
    *(stack_pointer--) = 0x0;

  stack_pointer++;
  return (_RegSet *)stack_pointer; //返回陷阱帧的指针
  //return NULL;
}
```

当操作系统初始化工作结束之后，应该通过自陷指令来触发一次上下文切换，从而切换到第一个用户程序中来执行，内核自陷的功能与ISA相关，由ASYE的`_trap()`函数所提供，在`x86-nemu`的AM中内核自陷通过指令`int $0x81`触发。在ASYE的`irq_handle()`函数发现触发了内核自陷之后，会将其包装成一个_EVENT_TRAP事件。Nanos-lite收到这个事件之后，就可以返回第一个用户程序的现场了。

实现内核自陷首先需要对Nanos-lite进行简单的修改，

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

  _trap();

  panic("Should not reach here");
}

//nanos-lite/src/pro.c
void load_prog(const char *filename) {
  int i = nr_proc ++;
  _protect(&pcb[i].as);

  uintptr_t entry = loader(&pcb[i].as, filename);

  // TODO: remove the following three lines after you have implemented _umake()
  //_switch(&pcb[i].as);
  //current = &pcb[i];
  //((void (*)(void))entry)();

  _Area stack;
  stack.start = pcb[i].stack;
  stack.end = stack.start + sizeof(pcb[i].stack);
```

```
  pcb[i].tf = _umake(&pcb[i].as, stack, stack, (void *)entry, NULL, NULL);
}
```

然后在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，这里的实现可以参考AM框架代码中已经实现并且在PA3阶段已经使用过的_EVENT_SYSCALL系统调用事件，

```
//nexus-am/am/arch/x86-nemu/src/asye.c
void vectrap(); //定义内核自陷函数vecself()

_RegSet* irq_handle(_RegSet *tf) {
  _RegSet *next = tf;
  if (H) {
    _Event ev;
    switch (tf->irq) {
      case 0x80: ev.event = _EVENT_SYSCALL; break;
      //irq_handle()函数将0x81异常封装成_EVENT_TRAP事件
      case 0x81: ev.event = _EVENT_TRAP; break;
      default: ev.event = _EVENT_ERROR; break;
    }

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

  return next;
}

void _asye_init(_RegSet*(*h)(_Event, _RegSet*)) {
  // initialize IDT
  for (unsigned int i = 0; i < NR_IRQ; i ++) {
    idt[i] = GATE(STS_TG32, KSEL(SEG_KCODE), vecnull, DPL_KERN);
  }

  // -------------------- system call ---------------------------
  idt[0x80] = GATE(STS_TG32, KSEL(SEG_KCODE), vecsys, DPL_USER);
  //填写0x81的门描述符
  idt[0x81] = GATE(STS_TG32, KSEL(SEG_KCODE), vectrap, DPL_USER);

  set_idt(idt, sizeof(idt));

  // register event handler
  H = h;
}

void _trap() {
  asm volatile("int $0x81"); //_trap()函数通过int 0x81指令触发内核自陷
}
```

```
#nexus-am/am/arch/x86-nemu/src/trap.S
#----|-------entry-------|-errorcode-|---irq id---|---handler---|
.globl vecsys;    vecsys:  pushl $0;  pushl $0x80; jmp asm_trap
#定义内核自陷入口函数vectrap，功能为压入error_code和irq（0x81），之后跳转到asm_trap中
.globl vectrap;  vectrap:  pushl $0;  pushl $0x81; jmp asm_trap
.globl vecnull;  vecnull:  pushl $0;  pushl   $-1; jmp asm_trap
```

因为实际中是由Nanos-lite来决定具体切换到哪个进程的上下文，所以还需要为Nanos-lite添加进程调度的功能，这项工作由`schedule()`函数（`nanos-lite/src/proc.c`中定义）来完成，它用于返回将要调度的进程的上下文，并且通过`current`指针（`nanos-lite/src/proc.c`中定义）来记录当前运行进程（指向当前运行进程的PCB）这样就可以在`schedule()`中通过`current`来决定接下来要调度哪一个进程，不过在调度之前，还需要把当前进程的上下文信息的位置保存在PCB当中，现在的`schedule()`只需要总是切换到第一个用户进程（pcb[0]）即可。

```
_RegSet* schedule(_RegSet *prev) {
  //Log("schedule");
  current->tf = prev; //保存当前进程上下文信息

  current = &pcb[0]; //选择第一个用户进程（pcb[0]）为新的进程

  _switch(&current->as); //切换至第一个用户进程（pcb[0]）
  return current->tf; //返回第一个用户进程的上下文
  //return NULL;
}
```

同时要在`do_event()`函数中对_EVENT_TRAP事件进行识别与处理，这里直接调用`schedule()`并返回其现场即可。

```
//nanos-lite/src/irq.c
extern _RegSet* schedule(_RegSet *prev);

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

  return NULL;
}
```

最后修改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。

```
#nexus-am/am/arch/x86-nemu/src/trap.S
asm_trap:
  pushal

  pushl %esp
  call irq_handle

  #addl $4, %esp
  movl %eax, %esp #将栈顶指针切换到新进程的陷阱帧，irq_handle()返回值存放在eax寄存器中

  popal
  addl $8, %esp

  iret
```

3.1.2 分时多任务

实现了虚拟内存和上下文切换机制之后，Nanos-lite已经可以支持分时多任务了，在Nanos-lite中加载第二个用户程序`hello`，

```
int main() {
  ......
  load_prog("/bin/pal");
  load_prog("/bin/hello");

  _trap();

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

修改调度的代码，让`schedule()`轮流返回仙剑奇侠传和`hello`的现场，

```
_RegSet* schedule(_RegSet *prev) {
  //Log("schedule");
  current->tf = prev;

  current = (current == &pcb[0] ? &pcb[1] : &pcb[0]);

  _switch(&current->as);
  return current->tf;
  //return NULL;
}
```

并且让在处理系统调用之后调用`schedule()`函数来进行系统调度。

```
static _RegSet* do_event(_Event e, _RegSet* r) {
  switch (e.event) {
    case _EVENT_SYSCALL:
      do_syscall(r);
      return schedule(r);
    ......

  return NULL;
}
```

最后为了提高《仙剑奇侠传》的优先级，调整《仙剑奇侠传》和`hello`程序调度的频率比例，让《仙剑奇侠传》调度若干次后`hello`程序才调度1次。

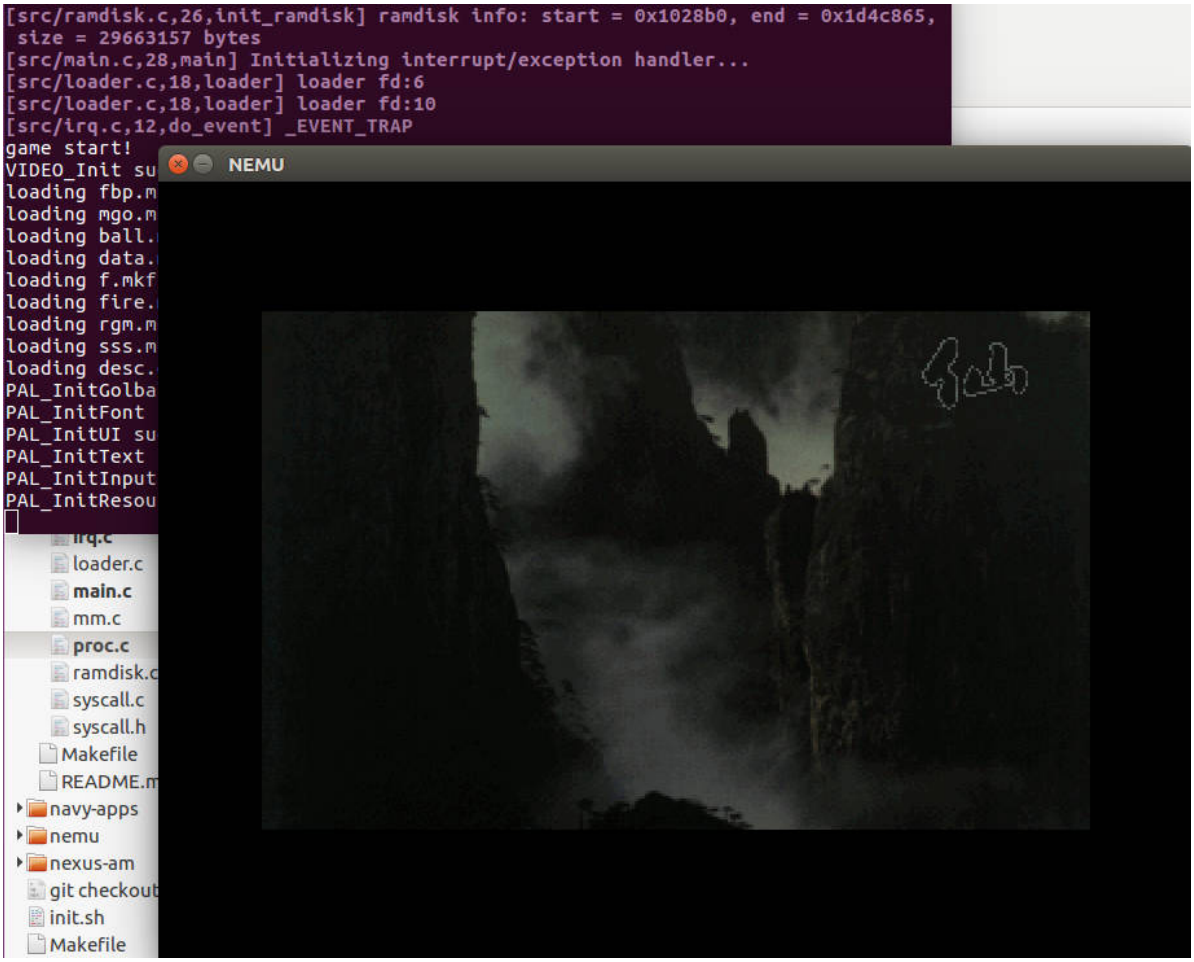
```
int count = 0;
extern int current_game;
_RegSet* schedule(_RegSet *prev) {
  //Log("schedule");
  current->tf = prev;
  //current = (current == &pcb[0] ? &pcb[1] : &pcb[0]);

  current = &pcb[0];
  if(count % 500 == 0)
    current = &pcb[1];
```

```
    count++;

    _switch(&current->as);
    return current->tf;
    //return NULL;
}
```

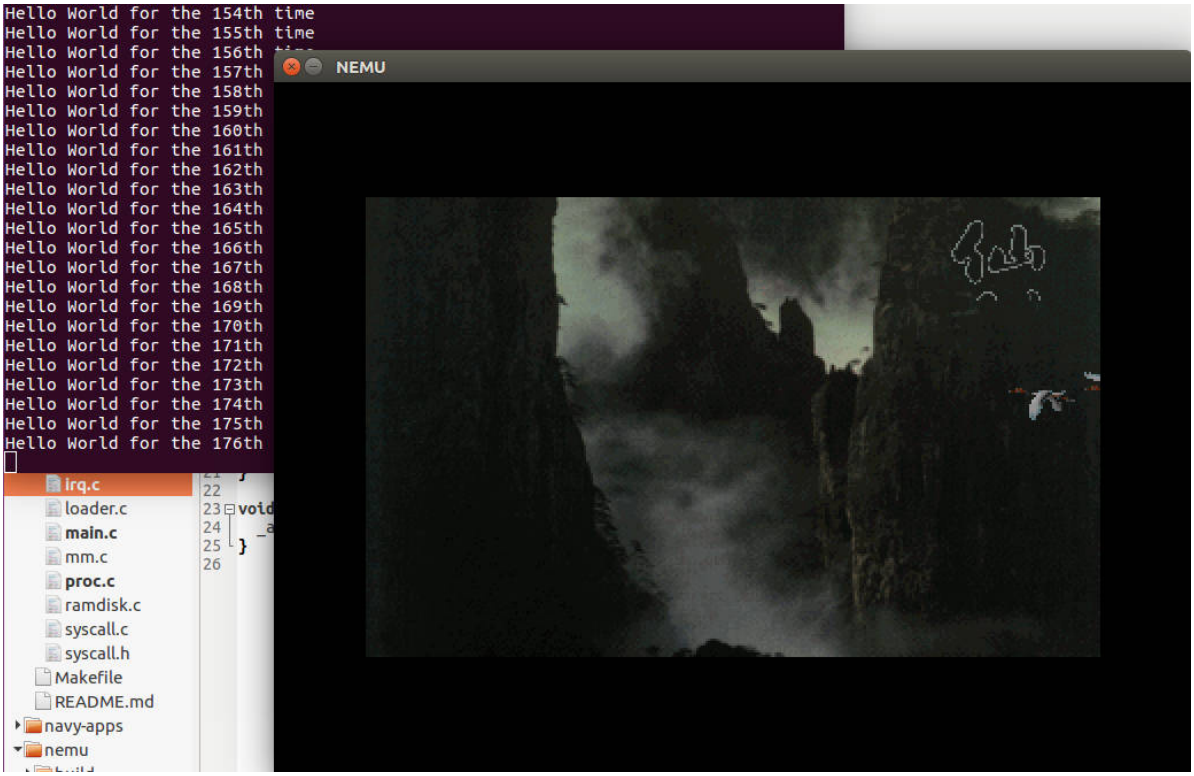
然后就可以在NEMU上在确保《仙剑奇侠传》运行较为流程的同时分时运行《仙剑奇侠传》与`hello`了。

3.2 运行结果

通过内核自陷启动《仙剑奇侠传》。



《仙剑奇侠传》与`hello`分时运行。



3.3 Bug总结

阶段二需要实现的功能较少，需要注意的就是在实现`_umake()`函数，人为构建陷阱帧时注意陷阱帧内各部分的数值和顺序；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事件时不要遗漏某个部分的实现；在`do_event()`函数中对_EVENT_TRAP事件进行识别与处理时要注意是直接调用`schedule()`并返回其现场，而不能调用`schedule()`后返回其他的值。

4 阶段三

实现阶段二后的Nanos-lite已经可以分时多任务地运行运行用户程序了，但现有的系统调度依赖于用户程序所触发的系统调用，这并不安全，所以要将触发系统调用的信号改为时钟，这样才能够确保所有的用户进程依然处于操作系统的控制之下，并且为了不浪费CPU的性能，原本实现的IOE中让CPU轮询设备的状态的方式就不那么合适了，要让时钟能够主动地通知处理器。

这样的通知机制，在计算机中称为硬件中断，在没有触发中断的时间里CPU可以正常工作，只有当触发中断后CPU才会执行与中断相关的代码。硬件中断的实质是一个数字信号，当设备有事件需要通知CPU的时候，就会发出中断信号。这个信号最终会传到CPU中，由CPU进行接下来的处理。

在阶段三中需要在NEMU中添加时钟中断，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，并让Nanos-lite在受到_EVENT_IRQ_TIME 事件之后进行进程切换，最后对系统做一点简单的修改，让Nanos-lite加载第3 个用户程序`videotest`并让用户可以通过按下F12来切换正在运行的用户程序。

4.1 实现方法及代码分析

支持中断机制的设备控制器都有一个中断引脚，这个引脚会和CPU的INTR引脚相连，当设备需要发出中断请求的时候，设备中断引脚置为高电平，中断信号就会一直传到CPU的INTR引脚中多个设备的中断处理可以通过（Programmable Interrupt Controller，可编程中断控制器）来控制。CPU每次执行完一条指令的都会看看INTR引脚是否有设备的中断请求到来。如果此时CPU没有处在关中断状态，它就会马上响应到来的中断请求。中断控制器会生成一个中断号，CPU会保存中断现场，然后根据这个中断号在IDT中进行索引，找到并跳转到入口地址，进行一些和设备相关的处理；如果CPU处于关中断状态，那么CPU就不会响应中断。并且为了中断处理的流程过于复杂或原本的进程信息丢失，中断嵌套是不被允许

的，这就意味着在处理一个中断时需要将CPU设置为关中断状态，在这个中断的处理过程结束后才会处理下一个中断。

4.1.1 时钟中断

在NEMU 中只需要添加时钟中断这一种中断，所以直接让时钟中断连接到CPU的INTR引脚即可，约定时钟中断的中断号是32。时钟中断通过`nemu/src/device/timer.c`中的`timer_intr()`触发，每10ms触发一次。触发后，会调用`dev_raise_intr()`函数（`nemu/src/cpu/intr.c`中定义）。

添加时钟中断，硬件部分首先要在cpu结构体中添加一个bool成员INTR，

```
//nemu/include/cpu/reg.h
typedef struct {
  ......

  bool INTR;

} CPU_state;
```

然后在`dev_raise_intr()`中将INTR引脚设置为高电平，

```
//nemu/src/cpu/intr.c
void dev_raise_intr() {
  cpu.INTR = true;
}
```

接着在`exec_wrapper()`的末尾添加轮询INTR引脚的代码，

```
//nemu/src/cpu/exec/exec.c
#define TIMER_IRQ 0x32
extern void raise_intr(uint8_t NO, vaddr_t ret_addr);

void exec_wrapper(bool print_flag) {
  ......
  //printf("cpu.INTR: 0x%x, cpu.eflags.IF: 0x%x\n",cpu.INTR, cpu.eflags.IF);
  if (cpu.INTR & cpu.eflags.IF) //在开中断且INTR为高电平情况下触发时钟中断
  {
    cpu.INTR = false;
    raise_intr(TIMER_IRQ, cpu.eip);
    //printf("TIMER_IRQ\n");
    update_eip();
  }
}
```

最后修改`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。

```
//nemu/src/cpu/intr.c
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.
   */
  rtl_push(&cpu.eflags_init);
  cpu.eflags.IF = 0; //关中断

  ......
}
```

软件部分首先需要在ASYE中添加时钟中断的支持，将时钟中断打包成_EVENT_IRQ_TIME事件，这部分的实现可以参考阶段二中内核自陷的实现，

```
//nexus-am/am/arch/x86-nemu/src/asye.c
void vectime(); //定义时钟中断函数

_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;
      //irq_handle()函数将0x32异常封装成_EVENT_IRQ_TIME事件
      case 0x32: ev.event = _EVENT_IRQ_TIME; break;
      default: ev.event = _EVENT_ERROR; break;
    }

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

  return next;
}

void _asye_init(_RegSet*(*h)(_Event, _RegSet*)) {
  // initialize IDT
  for (unsigned int i = 0; i < NR_IRQ; i ++) {
    idt[i] = GATE(STS_TG32, KSEL(SEG_KCODE), vecnull, DPL_KERN);
  }

  // ------------------- system call --------------------------
  idt[0x80] = GATE(STS_TG32, KSEL(SEG_KCODE), vecsys, DPL_USER);
  idt[0x81] = GATE(STS_TG32, KSEL(SEG_KCODE), vectrap, DPL_USER);
  //填写0x32的门描述符
  idt[0x32] = GATE(STS_TG32, KSEL(SEG_KCODE), vectime, DPL_USER);

  set_idt(idt, sizeof(idt));

  // register event handler
  H = h;
}
```

```
#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 vectrap;  vectrap:  pushl $0;  pushl $0x81; jmp asm_trap
.globl vecnull;  vecnull:  pushl $0;  pushl   $-1; jmp asm_trap
#定义时钟中断入口函数vectime，功能为压入error_code和irq（0x32），之后跳转到asm_trap中
.globl vectime;  vectime:  pushl $0;  pushl $0x32; jmp asm_trap
```

其次让Nanos-lite在收到_EVENT_IRQ_TIME事件之后直接调用schedule()进行进程调度，同时去掉系统调用之后调用的schedule()代码，

```
//nanos-lite/src/irq.c
static _RegSet* do_event(_Event e, _RegSet* r) {
  switch (e.event) {
    case _EVENT_SYSCALL:
      return do_syscall(r);
      //return schedule(r);
    case _EVENT_TRAP:
      Log("_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;
}
```

最后为了可以让处理器在运行用户进程的时候响应时钟中断，需要修改`_umake()`的代码，在构造现场的时候设置正确的EFLAGS。

```
//nexus-am/am/arch/x86-nemu/include/x86.h
// Eflags register
#define FL_IF     0x00000200  // Interrupt Enable

//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 *stack_pointer = (uint32_t*)(ustack.end);
  for(int i = 0; i < 4; i++)
    *(stack_pointer--) = 0x0;

  *(stack_pointer--) = 0x2 | FL_IF; //eflags，开中断
  *(stack_pointer--) = 0x8; //cs
  *(stack_pointer--) = (uint32_t)entry; //eip
  *(stack_pointer--) = 0x0; //error_code
  *(stack_pointer--) = 0x81; //irq

  for(int i = 0; i < 8; i++) //eax->edi
    *(stack_pointer--) = 0x0;

  stack_pointer++;
  return (_RegSet *)stack_pointer;
  //return NULL;
}
```

这样NEMU就可以在时钟中断的控制下分时运行《仙剑奇侠传》和`hello`了。

4.1.2 手动切换用户程序

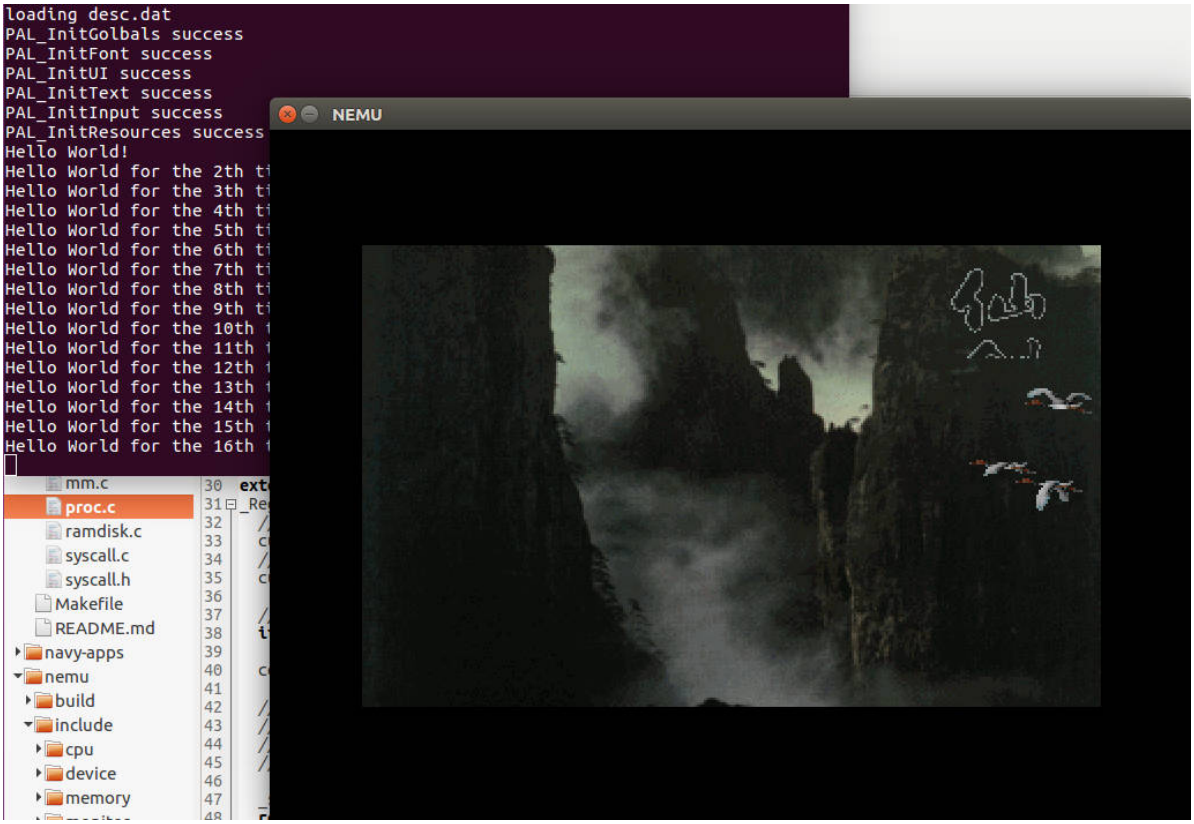
让Nanos-lite加载第3个用户程序`/bin/videotes`，并在Nanos-lite的`events_read()`函数中添加以下功能：当发现按下F12的时候，让游戏在仙剑奇侠传和`videotest`之间切换。为了实现这一功能，还需要修改`schedule()`的代码：通过一个变量`current_game`来维护当前的游戏，在`current_game`和`hello`程序之间进行调度。

```
//nanos-lite/src/device.c
int current_game = 0;

size_t events_read(void *buf, size_t len) {
  int key = _read_key();
  if(key == _KEY_NONE)
    sprintf(buf, "t %d\n", _uptime());
  else if ((uint32_t)key & 0x8000)
    sprintf(buf, "ku %s\n", keyname[key & ~0x8000]);
  else
  {
    sprintf(buf, "kd %s\n", keyname[key]);
    if (key == _KEY_F12) //在按下F12时切换current_game
    {
      current_game = (current_game == 0 ? 2 : 0);
      Log("switch current game to %d", current_game);
    }
  }
  return strlen(buf);
}

//nanos-lite/src/pro.c
int count = 0;
extern int current_game;
_RegSet* schedule(_RegSet *prev) {
  //Log("schedule");
  current->tf = prev;
  //current = (current == &pcb[0] ? &pcb[1] : &pcb[0]);
  current = (current_game == 0 ? &pcb[0] : &pcb[2]);

  //current = &pcb[0];
  if(count % 500 == 0)
    current = &pcb[1];
  count++;

  _switch(&current->as);
  return current->tf;
  //return NULL;
}
```

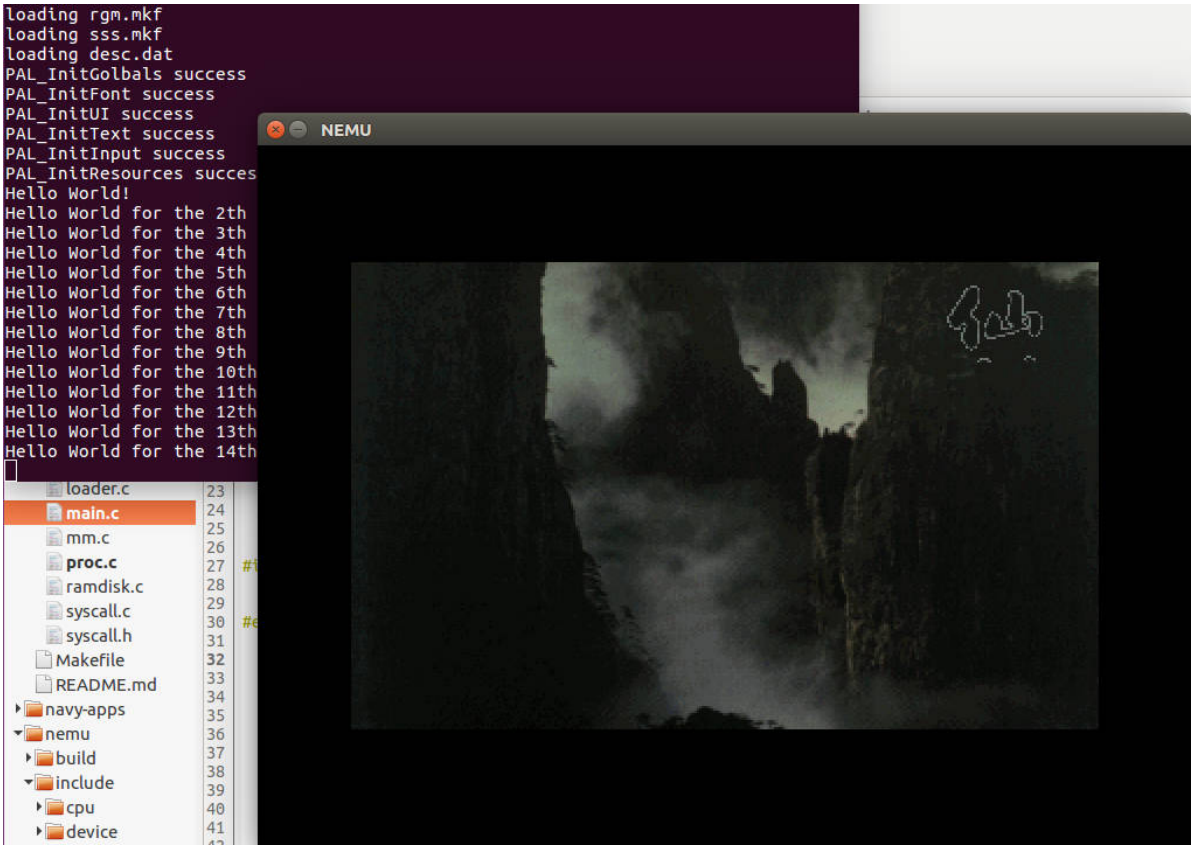
这样就可以通过按下F12来控制当前运行的用户程序了。

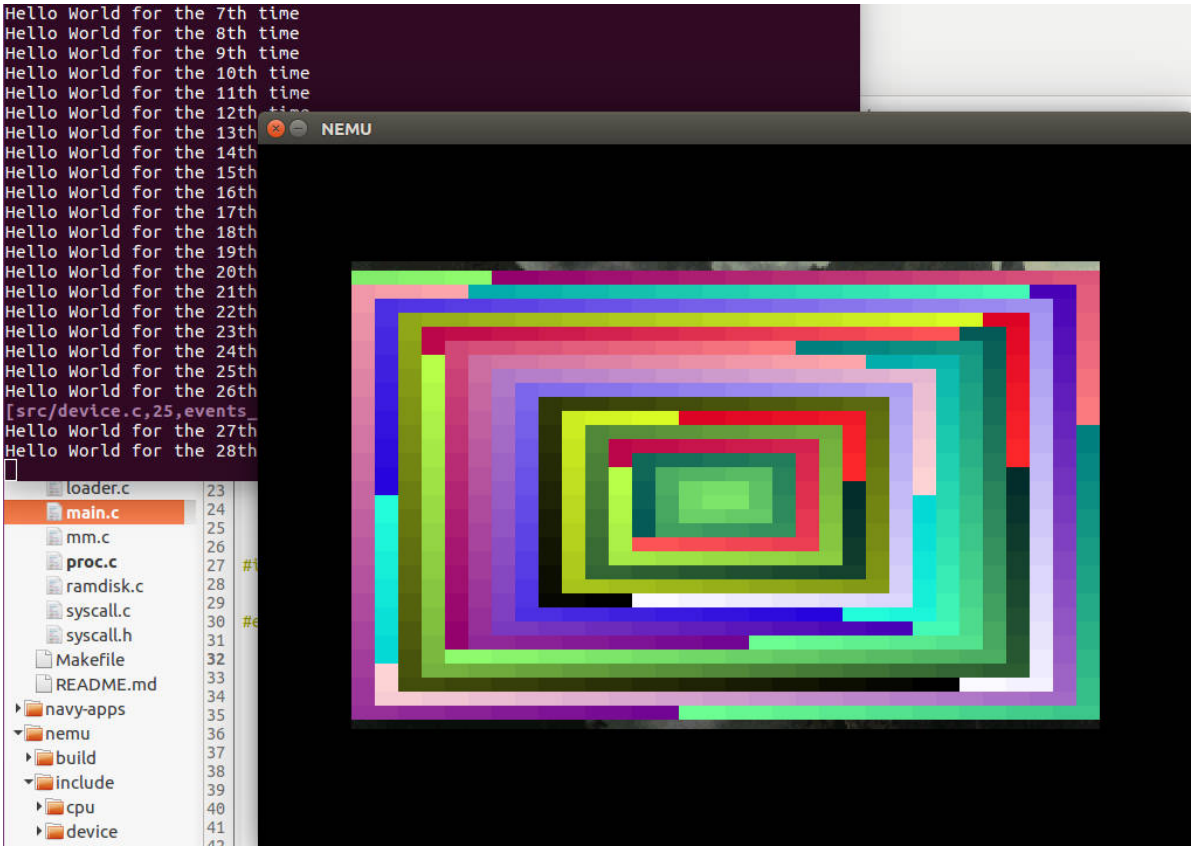
4.2 运行结果

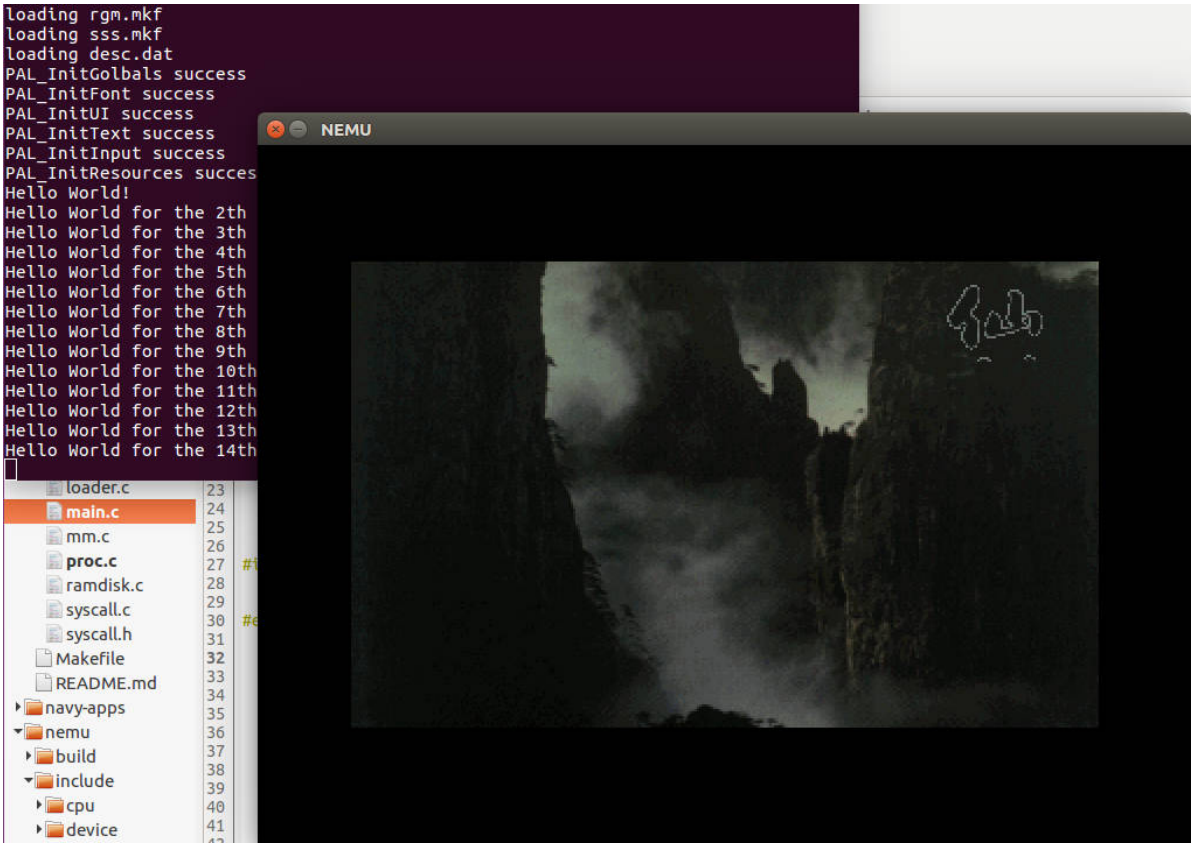
Nanos-lite通过时钟中断来进行用户进程调度。



通过F12手动切换用户进程。







4.3 Bug总结

阶段三需要实现的功能比较少，按照PA实验流程一步步完成，在实现过程中没有遇到什么难解决的Bug。

5 手册必答题

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。

多个用户进程的分时运行首先需要虚拟内存机制的支持，PA中虚拟内存机制采用i386分页机制实现，这需要NEMU提供CR0与CR3寄存器，并且在NEMU上实现从虚拟地址到物理地址的转换（`vaddr_read()`与`vaddr_write()`），Nanos-lite与AM共同实现内核页表的准备，其中Nanos-lite将TRM提供的堆区起始地址作为空闲物理页首地址，并注册物理页的分配函数new_page()与回收函数free_page()；AM通过`_pte_init()`来准备内核页表。

其次需要让Nanos-lite通过`load_prog()`来加载用户程序，`load_prog()`在此过程中会调用`_protect()`函数创建虚实地址映射；然后调用`loader()`加载程序（需要按页加载）最后通过`_umake()`函数创建进程的上下文，这样Nanos-lite就可以通过`_trap()`函数出发内核自陷来加载第一个用户程序，其他的用户进程的上下文同时也被创建，只是此时尚未被使用。

最后NEMU的`exec_wrapper`每执行完一条指令，就会查看INTR引脚是否有设备的中断请求到来（这里只可能是时钟中断），如果此时没有处在关中断状态，就会在AM中将时钟中断打包成_EVENT_IRQ_TIME事件，Nanos-lite收到该事件后就会调用`schedule()`进行进程调度，通过`_switch()`切换进程的虚拟内存空间，将新的进程的上下文传递给AM，在`asm_trap()`恢复新的进程的现场，接下来所运行的就是新的进程了。

6 感悟与体会

1. PA4的内容总体来说不是很多，并且有了先前的经验，Debug的速度比起最开始做PA已经快了很多，所以总体实现起来还是比较顺利的，没有在某个Bug上卡住导致浪费时间。
2. 还是要注意多看框架代码，阶段一中NEMU已经提供了CR0，CR3，PDE和PTE的实现，最开始没有注意到，差点自己给自己增加工作量。
3. 经过了PA1-PA4的实验，对于计组、操作系统的理解都更加深刻，也对NEMU，AM和Nanos-lite之间的协作以支撑《仙剑奇侠传》运行的工作机制有了充分的理解。
4. 尽管从PA1-PA4的却走了很多弯路，也踩了很多坑，在有的Bug上浪费了几个小时，或者因为忽略了实验流程中的一句话而Debug几个小时，但我依然认为PA依然是一个设计非常优秀的实验，配套的实验流程和要点指导的质量也都非常高，配合完善的框架代码和分阶段的设计，使得每个小阶段的实验复杂度都不会特别高，但整体完成之后再回看又的确完成了原本觉得相当困难的工作。