PA4-虚实交错的魔法：分时多任务

1 概述

1.1 实验目的

1. 学习虚拟内存映射，并实现分页机制
2. 学习上下文切换的基本原理并实现上下文切换、进程调度与分时多任务
3. 学习硬件中断并现实时钟中断

1.2 实验内容

1. 第一阶段，实现基本的多道程序系统
2. 第二阶段，实现支持虚存管理的多道程序系统
3. 第三阶段，实现抢占式分时多任务系统

2 阶段一

2.1 创建上下文

2.1.1 `_kcontext`

上下文的创建是通过CTE的`_kcontext()`方法来创建，在`stack`的底部创建一个以`entry`为返回地址的上下文结， 然后返回这一结构的指针. 改函数位于`nexus-am/am/src/x86/nemu/cte.c`中

```
_Context *_kcontext(_Area stack, void (*entry)(void *), void
*arg)
{
  _Context *context =stack.end-sizeof(_Context);
  memset(context,0x00,sizeof(_Context));
  context->cs=8;
  context->eip=(uint32_t)entry;
  return context;
}
```

2.2 进程调度

2.2.1 `schedule`

进程调度是通过`nanos-lite/src/proc.c`中的`schedule`函数来完成的，首先要保存当前进程的上下文指针，记录目前在运行哪个进程，然后运行新的进程。

```
_Context* schedule(_Context *prev) {
// save the context pointer
current->cp = prev;

// always select pcb[0] as the new process
current = &pcb[0];

// then return the new context
return current->cp;

}
```

2.2.2 do_event

修改`do_event`函数，当收到`_EVENT_YIELD`事件后，调用`schedule()`并返回新的上下文

```
case _EVENT_YIELD:
    // Log("This is yield.");
    return schedule(c);
    break;
```

2.2.3 trap.S

修改`__am_asm_trap`的汇编代码，使得从`__am_irq_handle()`返回后，先将栈顶指针切换到新进程的上下文结构，然后才恢复上下文，从而完成上下文切换的本质操作

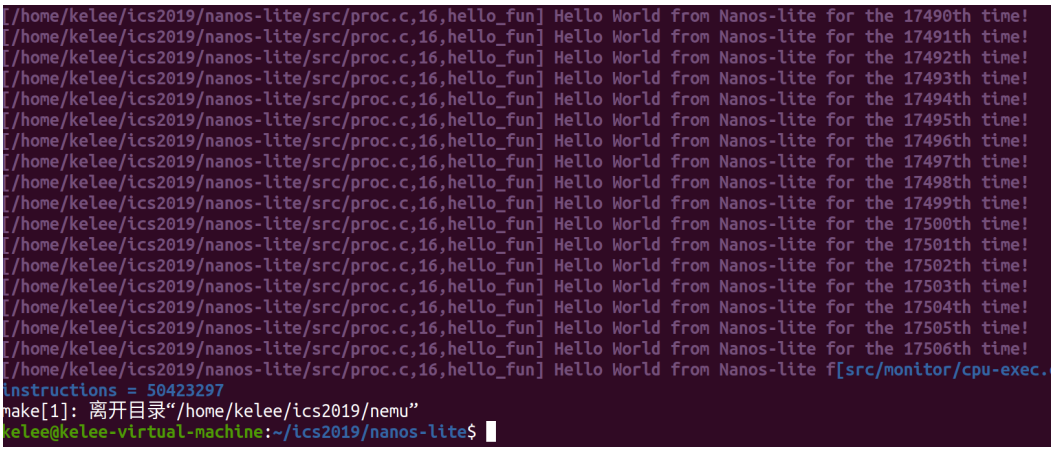
```
__am_asm_trap:
  pushal

  pushl $0

  pushl %esp
  call __am_irq_handle

  addl $4, %esp
  movl %eax, %esp # 进行上下文切换，eax保存了调度程序返回的上下文指针
  addl $4, %esp
  popal
  addl $4, %esp

  iret
```

2.2.4 init_proc

创建一个以`hello_fun`为返回地址的上下文

```
void init_proc() {
  context_kload(&pcb[0], (void *)hello_fun);
  switch_boot_pcb();

  Log("Initializing processes...");

  // load program here
  // naive_uload(NULL,"/bin/pal");
}
```

2.2.5 实验结果



2.3 创建用户进程上下文

2.3.1 `_ucontext`

创建用户进程的上下文需要调用`ucontext`方法，位于`nexus-am/am/src/x86/nemu/vme.c`
在创建的时候需要创建一个栈帧，因此代码如下

```
_Context *_ucontext(_AddressSpace *as, _Area ustack, _Area
kstack, void *entry, void *args)
{
  _Context *context = ustack.end - sizeof(_Context) - 0x20;
  memset(context, 0x00, sizeof(_Context) + 0x20);
  context->cs = 8;
  context->eip = (uint32_t)entry;
  return context;
}
```

2.3.2 `init_proc`

修改进程初始化代码，把仙剑奇侠传的进程添加到进程管理器中

```
  void init_proc()
{
  context_kload(&pcb[0], (void *)hello_fun);
  context_uload(&pcb[1], "/bin/pal");
  switch_boot_pcb();

  Log("Initializing processes...");

  // load program here
  // naive_uload(NULL,"/bin/pal");
}
```

2.3.3 `schedule`

修改调度代码，让当前进程在`pcb[0]`与`pcb[1]`之间切换

```
_Context *schedule(_Context *prev)
{
  // save the context pointer
  current->cp = prev;

  // always select pcb[0] as the new process
  // current = &pcb[0];
  current = (current == &pcb[0] ? &pcb[1] : &pcb[0]);

  // then return the new context
  return current->cp;
}
```

2.3.4 添加`_yield`

在serial_write(), events_read() 和fb_write()的开头调用_yield(), 来模拟设备访问缓慢的情况.

2.3.5 实验结果

3 阶段二

3.1 在分页机制上运行Nanos-lite

3.1.1 增加寄存器

首先开启`HAS_VME`宏定义，然后添加CR3和CR0寄存器。

```
CR0 cr0;
CR3 cr3;
```

3.1.2 `mov_cr2r/r2cr`

在`system.c`中实现这两个指令，指令的实现步骤和以前一样。

```
make_EHelper(mov_r2cr)
{
  // TODO();
  if (id_dest->reg == 0) {
    rtl_li(&cpu.cr0.val, id_src->val);
  } else if (id_dest->reg == 3) {
    rtl_li(&cpu.cr3.val, id_src->val);
  } else {
    Assert(0, "movr2cr!");
  }

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

make_EHelper(mov_cr2r)
{
```

```
  // TODO();
  if (id_src->reg == 0) {
    rtl_li(&s0, cpu.cr0.val);
  } else if (id_src->reg == 3) {
    rtl_li(&s0, cpu.cr3.val);
  } else {
    Assert(0, "movcr2r!");
  }

  operand_write(id_dest, &s0);

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

  difftest_skip_ref();
}
```

3.1.3 isa_vaddr_read/write

虚存访问都需要经过分页地址的转换。

```
uint32_t isa_vaddr_read(vaddr_t addr, int len)
{

  if (cpu.cr0.paging)
  {
    /* this is a special case, you can handle it later. */
    assert(0);
  }
  else
  {
    paddr_t paddr = page_translate(addr);
    return paddr_read(paddr, len);
  }
}

void isa_vaddr_write(vaddr_t addr, uint32_t data, int len)
{
  if (cpu.cr0.paging)
  {
    /* this is a special case, you can handle it later. */
    assert(0);
  }
  else
  {
    paddr_t paddr = page_translate(addr);
    paddr_write(paddr, len);
  }
}
```

3.1.4 `mmu.h`

在`mmu.h`中引入

```
#define PTXSHFT         12      // Offset of PTX in a linear
address
#define PDXSHFT         22      // Offset of PDX in a linear
address
#define PTE_P           0x001   // Present
#define PDX(va)           (((uint32_t)(va) >> PDXSHFT) & 0x3ff)
#define PTX(va)           (((uint32_t)(va) >> PTXSHFT) & 0x3ff)
#define OFF(va)           ((uint32_t)(va) & 0xfff)
#define PTE_ADDR(pte)     ((uint32_t)(pte) & ~0xfff)
```

3.1.5 `page_translate`

实现分页地址转换函数，使用assertion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, 及时终止NEMU的运行。

```
inline paddr_t page_translate(vaddr_t addr)
{
  paddr_t PDT_base = PTE_ADDR(cpu.cr3.val);
  assert(paddr_read(PDT_base + PDX(addr) * sizeof(PDE),
sizeof(PDE)) & PTE_P);
  paddr_t PTE_base = PTE_ADDR(paddr_read(PDT_base + PDX(addr) *
sizeof(PDE), sizeof(PDE)));
  assert(paddr_read(PTE_base + PTX(addr) * sizeof(PTE),
sizeof(PTE)) & PTE_P);
  paddr_t PF_base = PTE_ADDR(paddr_read(PTE_base + PTX(addr) *
sizeof(PTE), sizeof(PTE)));
  paddr_t paddr = PF_base | OFF(addr);
  return paddr;
}
```

3.1.6 数据跨页

由于x86并没有严格要求数据对齐, 因此可能会出现数据跨越虚拟页边界的情况, 例如一条很长的指令的首字节在一个虚拟页的最后, 剩下的字节在另一个虚拟页的开头. 在判断出数据跨越虚拟页边界的情况之后, 先使用assert(0)终止NEMU

```
uint32_t isa_vaddr_read(vaddr_t addr, int len)
{

  if (cpu.cr0.paging)
  {
    /* this is a special case, you can handle it later. */
    if (PTE_ADDR(addr) != PTE_ADDR(addr + len - 1))
    {
      printf("data cross the page boundary!");
      assert(0);
```

```
    }

    else
    {
      paddr_t paddr = page_translate(addr);
      return paddr_read(paddr, len);
    }
  }
  else
  {
    return paddr_read(addr, len);
  }
}

void isa_vaddr_write(vaddr_t addr, uint32_t data, int len)
{
  if (cpu.cr0.paging)
  {
    /* this is a special case, you can handle it later. */
    if (PTE_ADDR(addr) != PTE_ADDR(addr + len - 1))
    {
      printf("data cross the page boundary!");
      assert(0);
    }

    else
    {
      paddr_t paddr = page_translate(addr);
      paddr_write(paddr, len);
    }
  }
  else
  {
    paddr_write(addr, len);
  }
}
```

3.2 在分页机制上运行用户进程

3.2.1 `context_uload`

在开启`HAS_VME`宏之后，需要在加载用户进程之前为其创建地址空间。在`context_uload()`的开头调用`_protect()`就可以实现地址空间的创建

```
void context_uload(PCB *pcb, const char *filename)
{
#ifdef HAS_VME
  _protect(&pcb->as); // 建立用户进程虚存空间中的内核映射
#endif
  uintptr_t entry = loader(pcb, filename);

  _Area stack;
  stack.start = pcb->stack;
  stack.end = stack.start + sizeof(pcb->stack);

  pcb->cp = _ucontext(&pcb->as, stack, stack, (void *)entry,
NULL);
}
```

3.2.2 loader

`loader()`要做的事情是, 获取程序的大小之后, 以页为单位进行加载:

- 申请一页空闲的物理页
- 通过`_map()`把这一物理页映射到用户进程的虚拟地址空间中
- 从文件中读入一页的内容到这一物理页上

注意第一页可能不对齐，且要对进程剩下的地址空间赋值为0。

```
static uintptr_t loader(PCB *pcb, const char *filename)
{
  int fd = fs_open(filename, 0, 0);
  if (fd == -1)
  {
    panic("loader: can't open file %s!", filename);
  }
  Elf_Ehdr elf_header;
  fs_read(fd, (void *)&elf_header, sizeof(Elf_Ehdr));
  if (memcmp(elf_header.e_ident, ELFMAG, SELFMAG))
    panic("file %s ELF format error!", filename);
  for (size_t i = 0; i < elf_header.e_phnum; ++i)
  {
    Elf_Phdr phdr;
    fs_lseek(fd, elf_header.e_phoff + elf_header.e_phentsize * i,
SEEK_SET);
    fs_read(fd, (void *)&phdr, elf_header.e_phentsize);
    if (phdr.p_type == PT_LOAD)
    {
      fs_lseek(fd, phdr.p_offset, SEEK_SET);
#ifdef HAS_VME
      void *vaddr = (void *)phdr.p_vaddr;
      void *paddr;
      int32_t left_file_size = phdr.p_filesz;

      //处理第一页
      paddr = new_page(1);
      _map(&pcb->as, vaddr, paddr, 0);
```

```
      uint32_t page_write_size = min(left_file_size,
PTE_ADDR((uint32_t)vaddr + PGSIZE) - (uint32_t)vaddr);
      fs_read(fd, (void *)(PTE_ADDR(paddr) | OFF(vaddr)),
page_write_size);
      left_file_size -= page_write_size;
      vaddr += page_write_size;
      while (left_file_size > 0)
      {
        assert(((uint32_t)vaddr & 0xfff) == 0);
        paddr = new_page(1);
        _map(&pcb->as, vaddr, paddr, 0);
        page_write_size = min(left_file_size, PGSIZE);
        fs_read(fd, paddr, page_write_size);
        left_file_size -= page_write_size;
        vaddr += page_write_size;
      }
      // 将进程剩下的地址空间赋值为0
      left_file_size = phdr.p_memsz - phdr.p_filesz;
      if (((uint32_t)vaddr & 0xfff) != 0)
      {
        page_write_size = min(left_file_size,
PTE_ADDR((uint32_t)vaddr + PGSIZE) - (uint32_t)vaddr);
        memset((void *)(PTE_ADDR(paddr) | OFF(vaddr)), 0,
page_write_size);
        left_file_size -= page_write_size;
        vaddr += page_write_size;
      }
      for (page_write_size = PGSIZE; left_file_size > 0;
left_file_size -= page_write_size, vaddr += page_write_size)
      {
        assert(((uint32_t)vaddr & 0xfff) == 0);
        paddr = new_page(1);
        _map(&pcb->as, vaddr, paddr, 0);
        memset(paddr, 0, page_write_size);
      }

#else
      fs_read(fd, (void *)phdr.p_vaddr, phdr.p_filesz);
      memset((void *)(phdr.p_vaddr + phdr.p_filesz), 0,
phdr.p_memsz - phdr.p_filesz);
#endif
    }
  }
  fs_close(fd);
  return elf_header.e_entry;
}
```

3.2.3 `_map`

通过`as->ptr`获取页目录的基地址. 若在映射过程中发现需要申请新的页表, 可以通过回调函数`pgalloc_usr()`向Nanos-lite获取一页空闲的物理页，如果要映射的页已经存在，则报错。

```
// 将va虚拟地址映射到pa物理地址，将该映射关系写入页表中
int _map(_AddressSpace *as, void *va, void *pa, int prot)
{
  PDE *pdir = (PDE *)as->ptr;
  PDE *pgtab = NULL;
  PDE *pde=pdir+PDX(va);
  if (!(*pde&PTE_P))
  {

    pgtab=(PTE *)(pgalloc_usr(1));
    *pde=(uintptr_t)pgtab|PTE_P;
  }
  pgtab=(PTE*)PTE_ADDR(*pde);
  PTE *pte=pgtab+PTX(va);
  *pte=(uintptr_t)pa|PTE_P;
  return 0;
}
```

3.2.4 _ucontext

创建的用户进程上下文中设置地址空间相关的指针`as`

```
_Context *_ucontext(_AddressSpace *as, _Area ustack, _Area
kstack, void *entry, void *args)
{
  _Context *context = ustack.end - sizeof(_Context) - 0x20;
  memset(context, 0x00, sizeof(_Context) + 0x20);
  context->cs = 8;
  context->eip = (uint32_t)entry;
  context->as=as;
  return context;
}
```

3.2.5 __am_irq_handle()

在`__am_irq_handle()`的开头调用`__am_get_cur_as()`(在`nexus-am/am/src/$ISA/nemu/vme.c`中定义), 来将当前的地址空间描述符指针保存到上下文中。

在`__am_irq_handle()`返回前调用`__am_switch()`(`nexus-am/am/src/$ISA/nemu/vme.c`中定义) 来切换地址空间, 将调度目标进程的地址空间落实到MMU中

```
_Context *__am_irq_handle(_Context *c)
{
  extern void __am_get_cur_as(_Context * c);
  __am_get_cur_as(c);
  _Context *next = c;
  if (user_handler)
  {
    _Event ev = {0};
    switch (c->irq)
    {
```

```
    case 0x20:
      ev.event = _EVENT_IRQ_TIMER;
      break;
    case 0x80:
      ev.event = _EVENT_SYSCALL;
      break;
    case 0x81:
      ev.event = _EVENT_YIELD;
      break;
    default:
      ev.event = _EVENT_ERROR;
      break;
    }

    next = user_handler(ev, c);
    if (next == NULL)
    {
      next = c;
    }
  }
  // 改变CR3
  extern void __am_switch(_Context * c);
  __am_switch(next);

  return next;
}
```

3.2.6 init_proc

单独运行`dummy`(别忘记修改调度代码), 并先在`sys_exit`的实现中调用`_halt()`结束系统的运行。

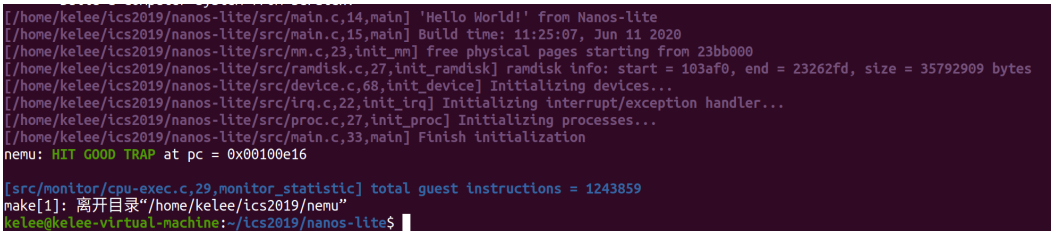
```
void init_proc()
{
  Log("Initializing processes...");
  context_uload(&pcb[0],"/bin/dummy");
  // context_kload(&pcb[0], (void *)hello_fun);
  // context_uload(&pcb[1], "/bin/pal");
  // switch_boot_pcb();


  // load program here
  // naive_uload(NULL,"/bin/pal");
}
```

3.2.7 修正数据跨页

```
uint32_t isa_vaddr_read(vaddr_t addr, int len)
{
  if (cpu.cr0.paging)
  { // 开启分页
```

```
    if (PTE_ADDR(addr) != PTE_ADDR(addr + len - 1))
    { // 数据跨页
      uint8_t byte[4];
      for (int i = 0; i < len; i++)
        byte[i] = isa_vaddr_read(addr + i, 1);
      if (len == 2)
        return *(uint16_t *)byte;
      else
        return *(uint32_t *)byte;
    }
    else
    {
      paddr_t paddr = page_translate(addr);
      return paddr_read(paddr, len);
    }
  }
  else
  {
    return paddr_read(addr, len);
  }
}

void isa_vaddr_write(vaddr_t addr, uint32_t data, int len)
{
  if (cpu.cr0.paging)
  { // 开启分页
    if (PTE_ADDR(addr) != PTE_ADDR(addr + len - 1))
    { // 数据跨页
      uint8_t byte[4];
      if (len == 2)
        *(uint16_t *)byte = data;
      else
        *(uint32_t *)byte = data;
      for (int i = 0; i < len; i++)
        isa_vaddr_write(addr + i, byte[i], 1);
    }
    else
    { // 数据没有跨页
      paddr_t paddr = page_translate(addr);
      paddr_write(paddr, data, len);
    }
  }
  else
  { // 不分页
    paddr_write(addr, data, len);
  }
}
```

3.2.8 实验结果



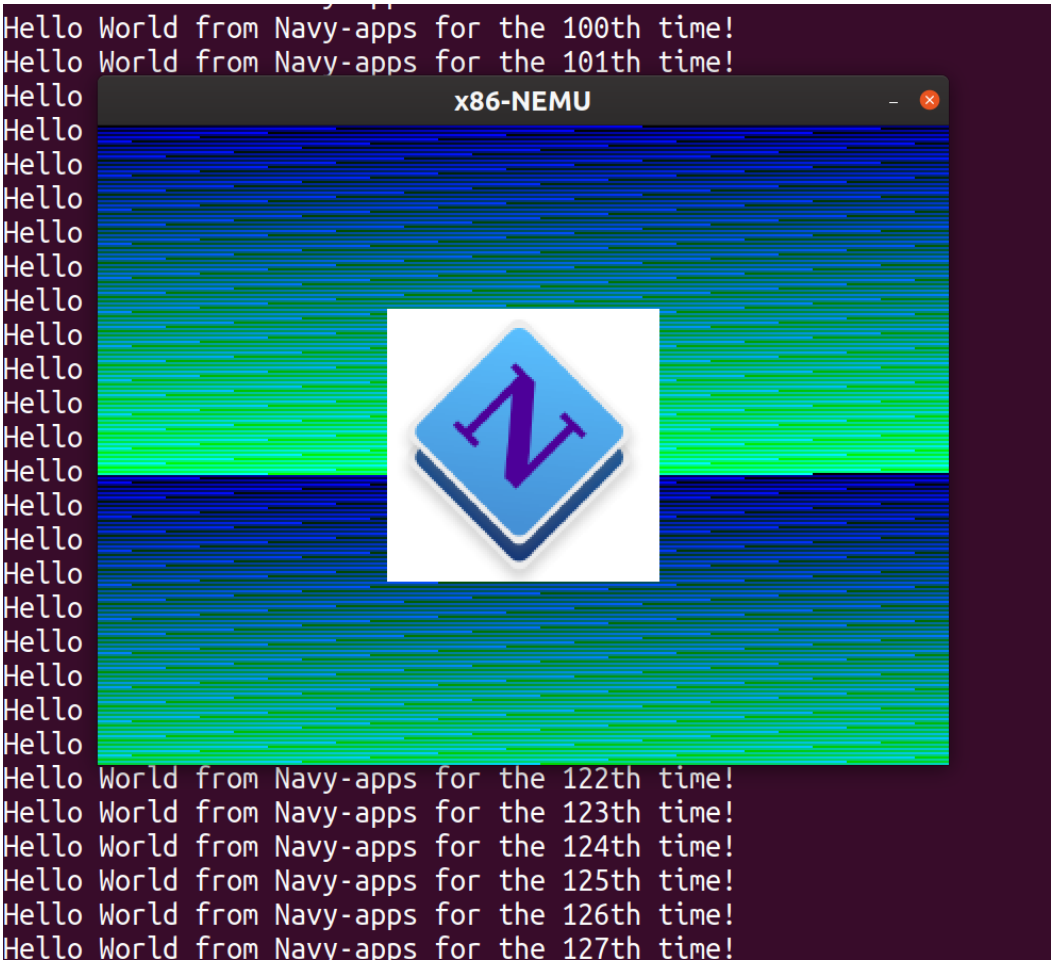
3.2.9 mm_brk

```
int mm_brk(uintptr_t new_brk)
{
#ifdef DEBUG
  Log("mm_brk: %x, max_brk: %x\n", (uint32_t)new_brk,
(uint32_t)current->max_brk);
#endif
  // 首次调用记录max_brk
  if (current->max_brk == 0)
  {
    current->max_brk = (new_brk & 0xfff) ? ((new_brk & ~0xfff) +
PGSIZE) : new_brk;
    return 0;
  }

  for (; current->max_brk < new_brk; current->max_brk += PGSIZE)
  {
    _map(&current->as, (void *)current->max_brk, new_page(1), 0);
  }

  return 0;
}
```

3.2.10 sys_brk

```
static inline int sys_brk(uint32_t brk)
{
#ifdef HAS_VME
  extern int mm_brk(uintptr_t new_brk);
  return mm_brk(brk);
#else
  return 0;
#endif
}
```

3.2.11 实验结果



3.2.12 问题

不能运行仙剑奇侠传，原因在编译APP时候系统调用出现问题。

```
void *_sbrk(intptr_t increment) {
  extern uint32_t _end;
  static uint32_t program_break = 0;
  if (program_break == 0) {
    program_break = &_end;
    _syscall_(SYS_brk, program_break, 0, 0);  // 第一次调用
  }
  if (_syscall_(SYS_brk, program_break + increment, 0, 0) == 0) {
    uint32_t old_break = program_break;
    program_break += increment;
    return old_break;
  } else {
    return -1;
  }
}
```

4 阶段三

4.1 抢占多任务

4.1.1 设置`INTR`

```
bool INTR;
} CPU_state;
```

在`nemu/src/device/intr.c`中

```
void dev_raise_intr() {
    cpu.INTR = true;
}
```

4.1.2 exec_once

```
vaddr_t exec_once(void)
{
  decinfo.seq_pc = cpu.pc;
  isa_exec(&decinfo.seq_pc);
  update_pc();
  extern bool isa_query_intr(void);
  if (isa_query_intr())
    update_pc();
  return decinfo.seq_pc;
}
```

4.1.3 isa_query_intr

```

bool isa_query_intr(void)
{
  if (cpu.eflags.IF & cpu.INTR)
  {
    cpu.INTR = false;
    raise_intr(IRQ_TIMER, cpu.pc);
    return true;
  }
  return false;
}
```

然后修改`raise_intr`增加一个`cpu.eflags.IF = 0;`

4.1.4 时钟中断

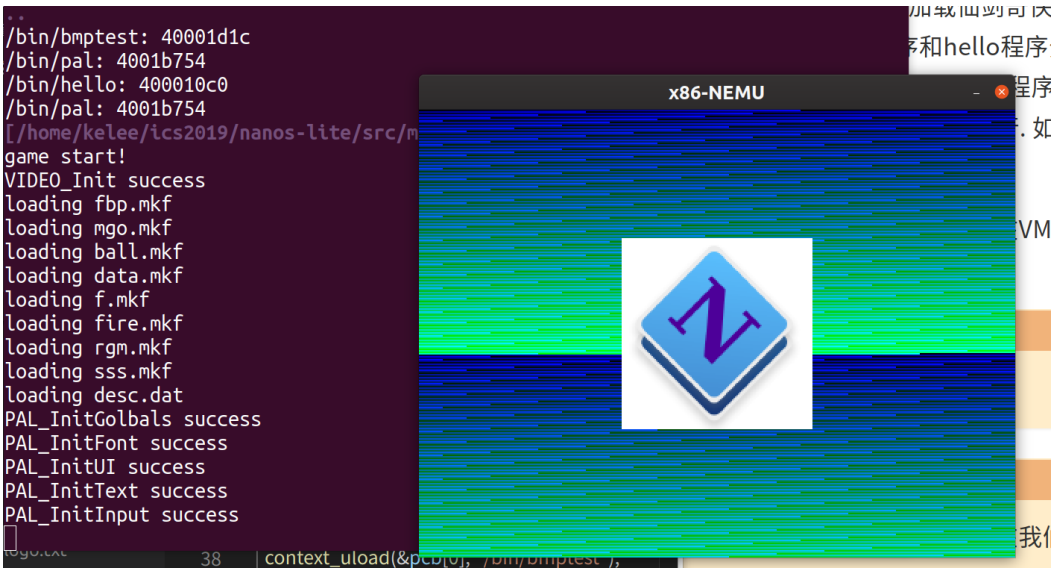
将时钟中断打包成_EVENT_IRQ_TIMER事件，然后再`do_events`里面调用`__yield`，然后再`ucontext`里面增加IF的赋值。

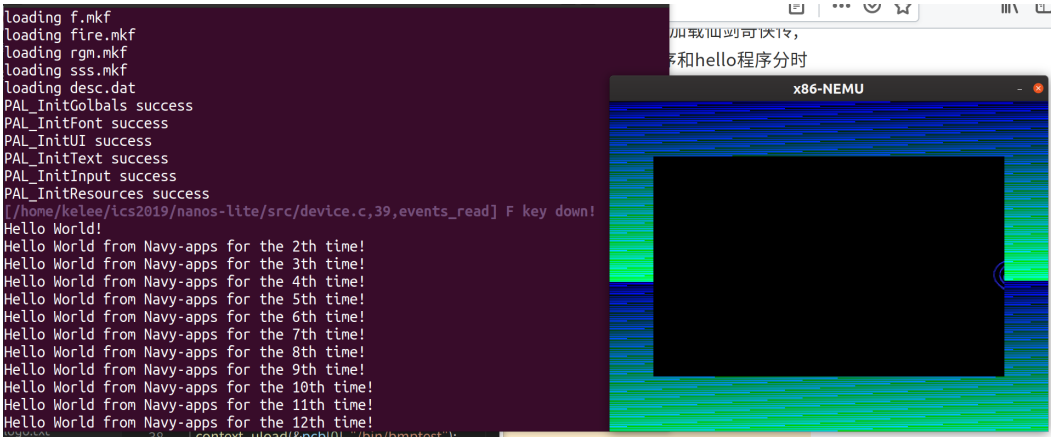
修改`device.c`里面的`events_read`可以识别`F`按键

```
size_t events_read(void *buf, size_t offset, size_t len)
{
  _yield(); // 模拟IO慢，进行调度

  int keycode = read_key();
  if ((keycode & ~KEYDOWN_MASK) == _KEY_NONE)
```

```
  {
    sprintf(buf, "t %d\n", uptime());
  }
  else if (keycode & KEYDOWN_MASK)
  {
    sprintf(buf, "kd %s\n", keyname[keycode & ~KEYDOWN_MASK]);
    if (keyname[keycode & ~KEYDOWN_MASK][0] == 'F')
    {
      Log("F key down!");
      switch (keyname[keycode & ~KEYDOWN_MASK][1])
      {
      case '1':
        fg_pcb = 1;
        break;
      case '2':
        fg_pcb = 2;
        break;
      case '3':
        fg_pcb = 3;
        break;
      default:
        break;
      }
    }
  }
  else
  {
    sprintf(buf, "ku %s\n", keyname[keycode & ~KEYDOWN_MASK]);
  }
  return strlen(buf);
}
```

4.2 实验结果





> 问题一：分时多任务的具体过程 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.
>
> 答：首先CR0、CR3寄存器掌管分页机制，cr0开启，cr3记录根页表基地址，然后交给MMU转换，也就是`vaddr_read/write`，为启动分页机制，操作系统还需要准备内核页表，这一过程由 Nanos-lite 与 AM 协作实现：Nanos-lite 实现存储管理器的初始化：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()`，调用AM 的`vme_init()`来准备内核页表。`vme_init()`函数为 AM 的准备内核页表的基本框架，该函数填写内核的二级页表并设置 CR0 与 CR3 寄存器，程序通过`context_uload`进行加载，通过`_protect`创建地址空间和内核映射。然后调用`loader`进行读取程序，并通过`_map`分配相应的物理页，通过`_ucontext`创建相应的上下文。
>
> 中断就是当CPU执行完一条指令后查看寄存器是否开中断和有中断，如果触发时钟中断，就会引发`_EVENT_IRQ_TIMER`事件，就会自陷，然后引发`_EVENT_YIELD`事件，通过`schedule`进行进程调度。调度时候通过`trap.S`切换上下文，执行下一条指令时就可以开始新进程的运行。