PA4

> 2013551 雷贺奥

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学习硬件中断并实现时钟中断

实验内容

PA4实验中主要涉及四大部分：

第一阶段，虚拟地址空间的作用，并实现分页机制，并让用户程序运行在分页机制上。

第二阶段，实现内核自陷、上下文切换与分时多任务。

第三阶段，解决阶段二分时多任务的隐藏bug：改为使用时钟中断来进行进程调度。

最后，实现当前运行游戏的切换，使不用的游戏与hello程序分时运行。

实验过程

阶段一

实现CR0与CR3

在编写代码前，我们需要在 common.h 中打开 HAS_PTE 宏定义。

- 在 reg.h 中修改寄存器结构体，增加 CR0 和 CR3 寄存器。

  （1） CR0，PA 中只实现PG 位，PG=1 时意味着咱们开启分页机制。

  （2）CR3 是页目录基址寄存器，在开启分页机制后使用，保存页目录表的物理地址。

```
uint32_t CR0;
uint32_t CR3;
```

- monitor.c 中的 restart 函数初始化 CR0 值。

```
static inline void restart() {
  cpu.eip = ENTRY_START;
  //进行eflags的初始化, 0x0000 0002H
  unsigned int origin =2;
  memcpy(&cpu.eflags,&origin,sizeof(cpu.eflags));
  //cs 初始化为8
  cpu.cs=8;
  // CR0 初始化
  cpu.CR0=0x60000011;
}
```

CR0初始化为PG=1 ，意味着咱们开启分页机制。

此时运行程序，发现指令`mov %eax,%cr3`报错，查看 nanos-lite-x86-nemu.txt 发现是与 CR3 相关的指令。需要实现新的mov指令函数，来完成对 两个控制寄存器的操作。

mov指令

新增的rtl指令实现对二者的访问与修改，阅读i386手册可以知道，load操作，在获取到CR寄存器编号和操作数值时，只需要使用原有的mov指令，将CR寄存器中的值mov到通用寄存器；而store操作，需要重新定义执行函数，将通用寄存器中的值存在CR寄存器中。

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

```
static inline void rtl_load_cr(rtlreg_t* dest,int r)
{
    if(r==0)
        *dest=cpu.CR0;
    else if(r==3)
        *dest=cpu.CR3;
    Log("load CR%d\n",r);
}
static inline void rtl_store_cr(int r,const rtlreg_t* src)
{
    if(r==0)
        cpu.CR0=*src;
    else if(r==3)
        cpu.CR3=*src;
    //Log("store CR%d\n",r);
}
```

- 修改 `nemu/include/cpu/decode.h` 和 `decode.c` 文件

```
//pa4 添加
make_DHelper(mov_load_cr);
make_DHelper(mov_store_cr);
//pa4 add
make_DHelper(mov_load_cr)
{
    decode_op_rm(eip,id_dest,false,id_src,false);
    //load cr
    rtl_load_cr(&id_src->val,id_src->reg);

}
make_DHelper(mov_store_cr)
{
    //store cr
    decode_op_rm(eip,id_src,true,id_dest,false);
}
```

- 在 all_instr.h 中注册指令、在 data-move.c 中实现执行函数

```
make_EHelper(mov_store_cr);//pa 4 add
//pa 4 add
make_EHelper(mov_store_cr)
{
    rtl_store_cr(id_dest->reg,&id_src->val);
    print_asm_template2(mov);
}
```

- 修改 `nemu/src/cpu/exec/exec.c` 文件，更改的是 `2 Byte_opcode_table` 部分

```
/* 0x20 */ IDEX(mov_load_cr,mov) , EMPTY, IDEX( mov_store_cr , mov_store_cr )
, EMPTY,
```

- 观察执行到 `movl %cr0,%eax`，两个CR寄存器中的值

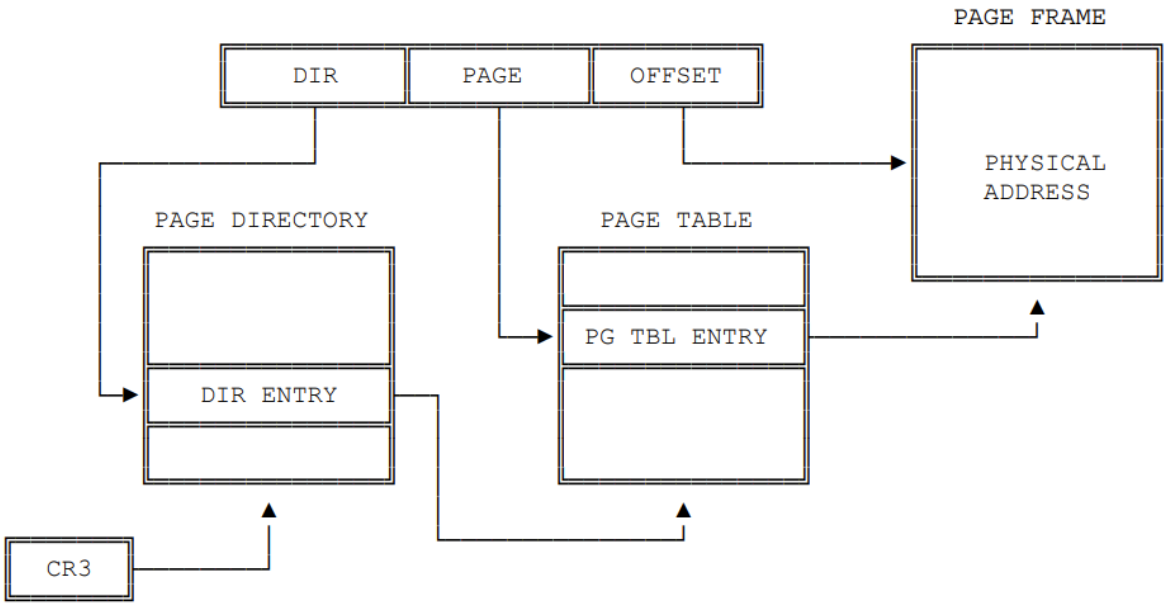
```
(nemu) si
  1017ee:   0f 20 c0                                movl %cr0,%eax
[src/monitor/debug/expr.c,104,make_token] match rules[3] = "\$(eax|
```

watch point： CR0 = 0xe0000011 CR3=0x1d6e000

虚拟地址的转换

只需要简单地阅读i386手册，找到虚拟地址和物理地址之间的转化关系。

Figure 5-9. Page Translation



其中，可以简单得知DIR 10bits、PAGE 10bits、OFFSET 10bits。

- 修改 `nemu/src/memory/memory.c` 文件，在 page_translate 中将会使用到

```
//取前20位
#define PTE_ADDR(pte) ((uint32_t)(pte)& ~0xfff)
//页目录
#define PDX(va) (((uint32_t)(va)>>22) & 0x3ff)
//二级页表
#define PTX(va) (((uint32_t)(va)>>12) & 0x3ff)
//offset
#define OFF(va) ((uint32_t)(va) & 0xfff)
```

- 修改 `nemu/src/memory/memory.c` 文件实现page_translate

  此时还需注意，page_translate 增加参数flag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 flag)
{
    //CR0转为mmu中定义的结构体
    CR0 cr0=(CR0)cpu.CR0;
    if(!(cr0.paging&&cr0.protect_enable))
        return addr;
    //CR3转为mmu中定义的结构体
    CR3 cr3=(CR3)cpu.CR3;
    //页目录
    //CR3中base，页目录表基址
    PDE* base=(PDE*) PTE_ADDR(cr3.val);//
    PDE pde=(PDE)paddr_read((uint32_t)(base+PDX(addr)),4);
    assert(pde.present);
    //二级页表
    PTE* ptab=(PTE*)PTE_ADDR(pde.val);
    PTE pte=(PTE)paddr_read((uint32_t)(ptab+PTX(addr)),4);
    assert(pte.present);
    //access、dirty
    pde.accessed=1;
    pte.accessed=1;
```

```
    if(flag)
        pte.dirty=1;
    //物理地址
    paddr_t real=PTE_ADDR(pte.val)|OFF(addr);
    //Log("virtual addr=%x\t real address:%x",addr,real);
    return real;
}
```

- 编写 vaddr_read 和 vaddr_write 函数，实现读写地址时的虚拟地址转换。

```
uint32_t vaddr_read(vaddr_t addr, int len) {
  //return paddr_read(addr, len);
  if(PTE_ADDR(addr)!=PTE_ADDR(addr+len-1))
  {
    Log("read跨页了，需要拼接\n");
    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) {
  //paddr_write(addr, len, data);
    if(PTE_ADDR(addr)!=PTE_ADDR(addr+len-1))
    {
      Log("write跨页了，需要拼接\n");
    }
    else
    {
      paddr_t paddr=page_translate(addr,true);
      paddr_write(paddr,len,data);
    }
}
```

这个函数需要依次判断页目录、页表是否存在，并根据读写情况对页面进行脏位标记。如果页面 不存在，需要呈现错误信息并结束程序。运行程序时可以发现虚拟地址和物理地址是相同的。

现在，运行程序，可以看出dummy可以运行，并且此时的物理地址和虚拟地址相同，运行仙剑时，log报错，“跨页，需要拼接”。

```
vaddr=0x100032, paddr=0x100032
nemu: HIT GOOD TRAP at eip = 0x00100032
```

数据跨越虚拟页边界的实现

为了解决上述的跨页问题，需要重写read和write函数，在其跨页时，对其数据进行拼接。

- 在 vaddr_read 中将两次读取的字节进行整合，实现时注意模拟器为小端结构。

  由于是小端结构，所以，paddr2需要放置在高位，paddr1需要放置在低位。

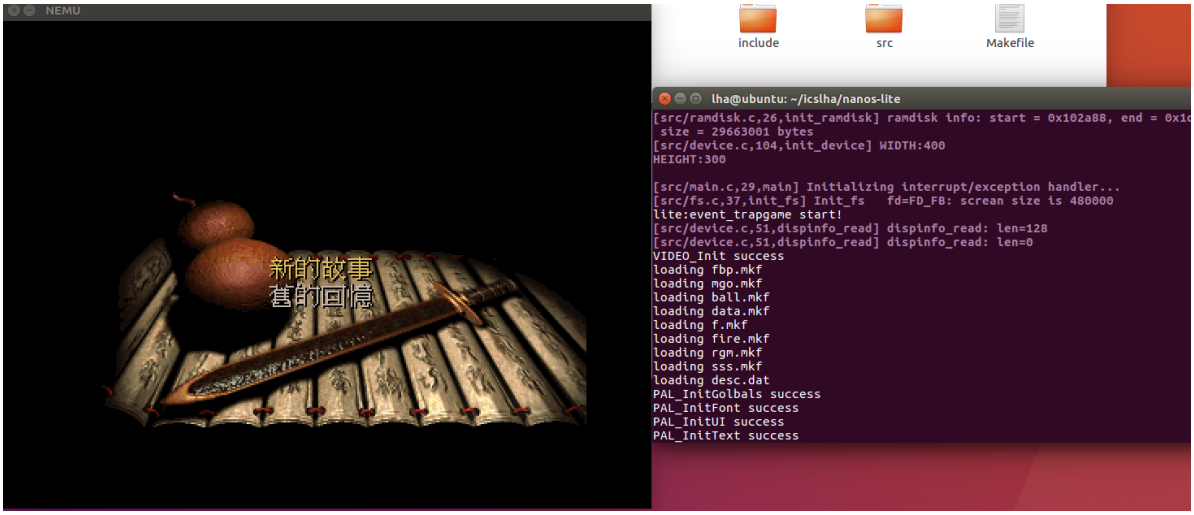
```
uint32_t vaddr_read(vaddr_t addr, int len) {
  //return paddr_read(addr, len);
  if(PTE_ADDR(addr)!=PTE_ADDR(addr+len-1))
```

```
  {
    //Log("read跨页了，需要拼接\n");
    int len1=0x1000-OFF(addr);
    int len2=len-len1;
    //分别映射
    paddr_t paddr1=page_translate(addr,false);
    paddr_t paddr2=page_translate(addr+len1,false);
    //组合返回值
    uint32_t low=paddr_read(paddr1,len1);
    uint32_t high=paddr_read(paddr2,len2);
    uint32_t result=high<<(len1*8)|low;
    //Log("high:%x\t low:%x\t result:%x\n",high,low,result);
    return result;
  }
  else
  {
    paddr_t paddr=page_translate(addr,false);
    return paddr_read(paddr,len);
  }
}
```

- 在 vaddr_write 中将需要写入的字节进行拆分并分 别写入两个页面。实现时注意模拟器为小端结构。与read类似。

```
void vaddr_write(vaddr_t addr, int len, uint32_t data) {
  //paddr_write(addr, len, data);
    if(PTE_ADDR(addr)!=PTE_ADDR(addr+len-1))
    {
      //Log("write跨页了，需要拼接\n");
      int len1=0x1000-OFF(addr);
      int len2=len-len1;
      //分别映射
      paddr_t paddr1=page_translate(addr,true);
      paddr_t paddr2=page_translate(addr+len1,true);
      //组合
      uint32_t low=data&(~0u>>((4-len1)<<3));
      uint32_t high=data>>((4-len2)<<3);
      paddr_write(paddr1,len1,low);
      paddr_write(paddr2,len2,high);
    }
    else
    {
      paddr_t paddr=page_translate(addr,true);
      paddr_write(paddr,len,data);
    }
}
```

即可成功运行仙剑奇侠传：



让用户程序运行在分页机制上

- 修改 Makefile.compile

  如指导手册所示，这里不再赘述。

- 修改 `nexus-am/am/arch/x86-nemu/src/pte.c` 文件

  实现 _map 函数，该函数将虚拟地址空间 p 中的虚拟地址 va 映射到物理地址 pa，通过 p->ptr 可以获取页目录的基地址；如果映射过程中发现需要申请新的页，则调用 palloc_f 函数;如果va 已经被映射到其他页，则取消原映射。

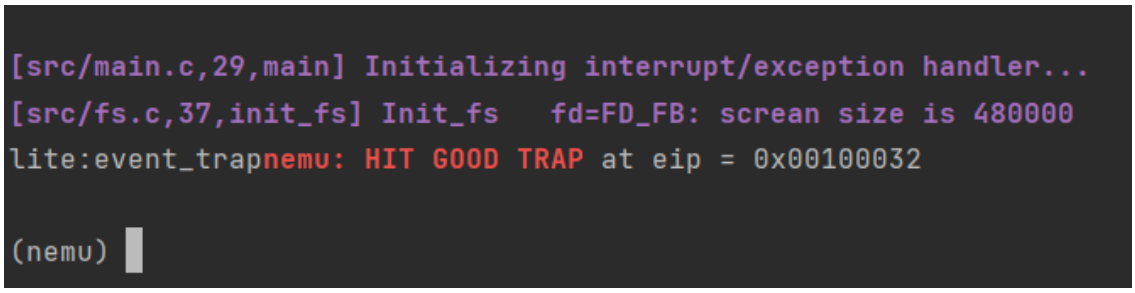
```
void _map(_Protect *p, void *va, void *pa) {
    if(OFF(va)||OFF(pa))
    {
        return;
    }
    //页目录表基地址
    PDE *base=(PDE*)p->ptr;
    //二级页表首地址
    PTE *pgtab=NULL;

    PDE *pde=base+PDX(va);//页目录表中entry
    if((*pde)&PTE_P)//present
    {
        pgtab=(PTE*)PTE_ADDR(*pde);//二级页表首地址
    }
    else
    {
        //!present 申请新页
        pgtab=(PTE*)(palloc_f());
        *pde=(uintptr_t)pgtab|PTE_P;//set present
        pgtab=(PTE*)PTE_ADDR(*pde);//二级页表首地址
    }
    PTE *pte=pgtab+PTX(va);//二级页表项
    *pte=(uintptr_t)pa|PTE_P;
}
```

- 修改 `nanos-lite/src/loader.c` 文件

```
uintptr_t loader(_Protect *as, const char *filename) {
/* PA4*/
    int fd = fs_open(filename,0,0);
```

```
    int size= getfile_size(fd);
    int page_num=size/PGSIZE;//页数量
    if(size%PGSIZE!=0)
        page_num++;
    void *pa=NULL;
    void *va=DEFAULT_ENTRY;
    for(int i=0;i<page_num;i++)
    {
        //申请空闲页
        pa=new_page();
        //va->pa
        _map(as,va,pa);
        fs_read(fd,pa,PGSIZE);
        va+=PGSIZE;
    }
    fs_close(fd);
    return (uintptr_t)DEFAULT_ENTRY;
}
```

- 最终dummy可以成功运行



在分页机制上运行仙剑奇侠传

- 修改 `nanos-lite/src/mm.c` 文件实现mm_brk函数,实现堆内存的映射

```
int mm_brk(uint32_t new_brk) {
  //return 0;
  if(current->cur_brk==0)
  {
    current->cur_brk=current->max_brk=new_brk;
  }
  else
  {
    if(new_brk>current->max_brk)
    {
        uint32_t begin=PGROUNDUP(current->max_brk);
        uint32_t end=PGROUNDDOWN(new_brk);
        //new_brk = 4k
        if((new_brk&0xfff)==0)
        {
            //无需映射
            end-=PGSIZE;
        }
        for(uint32_t va=begin;va<=end;va+=PGSIZE)
        {
            void* pa=new_page();
            _map(&(current->as),(void*)va,pa);
        }
        current->max_brk=new_brk;
```

```
    }
    current->cur_brk=new_brk;
  }
  return 0;
}
```

- 修改 `nanos-lite/src/syscall.c` 文件实现sys_brk函数

  在 PA3 中只是简单的return 0，但它实际上需要调用mm_brk 来完成功能。

```
int mysys_brk(int addr)
{
    //单任务总是返回0,
    //return 0;
    return mm_brk(addr);
}
```

- 运行结果

阶段二

实现内核自陷

- 准备工作，修改main.c 和proc.c

```
--- 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)();
```

- _trap触发 int 0x81指令

  修改`nexus-am/am/arch/x86-nemu/src/asye.c`文件

```
void _trap()
{
    asm volatile("int $0x81");
}
```

- 定义内核自陷的入口函数 vecself

  该函数压入错误码和异常号 irq（0x81），并跳转到 asm_trap 中

  修改`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 vecself;  vecself:  pushl $0;  pushl $0x81; jmp asm_trap
```

- ASYE 中定义 vecself 函数

  修改`nexus-am/am/arch/x86-nemu/src/asye.c`文件

```
void vecself();
```

- 设置IDT

  _asye_init()函数中填写 0x81 的门描述符

  修改`nexus-am/am/arch/x86-nemu/src/asye.c`文件

```
  // ------------------- system call --------------------------
  idt[0x80] = GATE(STS_TG32, KSEL(SEG_KCODE), vecsys, DPL_USER);
  idt[0x81] = GATE(STS_TG32, KSEL(SEG_KCODE), vecself, DPL_USER);
```

- irq_handle 将 0x81 异常封装成_EVENT_TRAP 事件

  修改`nexus-am/am/arch/x86-nemu/src/asye.c`文件

```
_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;
    }
  }
  return next;
}
```

- do_event 根据事件再次分发

  对于_EVENT_TRAP，输出提示信息，直接返回。修改 `nanos-lite/src/irq.c` 文件

```
 case 0x81 : ev.event = _EVENT_TRAP; break ;
```

- 运行结果

```
event:self-trapped
[src/main.c,35,main] system panic: Should not reach here
nemu: HIT BAD TRAP at eip = 0x00100032
```

  输出“event：self trapped”的提示信息，触发main()函数最后的panic。这是由于此时只实现了内核自陷，而没有实现上下文切换。

实现上下文切换

- umake函数, `nexus-am/am/arch/x86-nemu/src/pte.c`

  如实验指导手册所示：

  首先将start()的三个参数和eip入栈，实际上start()不会使用这些参数，也不会从_start()返回， 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 differential testing，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。

```
_RegSet *_umake(_Protect *p, _Area ustack, _Area kstack, void *entry, char
*const argv[], char *const envp[]) {
      extern void* memcpy(void*,const void*,int);
      //设置好_start()函数的栈帧
      //参数从右至左
      memcpy((void*)ustack.end-4,(void*)NULL,4);
      memcpy((void*)ustack.end-8,(void*)NULL,4);
      memcpy((void*)ustack.end-12,(void*)0,4);
      memcpy((void*)ustack.end-16,(void*)0,4);//_start->eip,不会从eip返回
      //trap frame
      _RegSet tf;
```

```
        tf.eflags=0x02|FL_IF;
        tf.cs=8;
        tf.eip=(uintptr_t)entry;
        //ustack中存tf的地址
        void* ptf=(void*)(ustack.end-16-sizeof(_RegSet));
        memcpy(ptf,(void*)&tf,sizeof(_RegSet));
        return (_RegSet*)ptf;

}
```

- schedule 函数

  完成 schedule 函数，用于进程调度。若当前存在运行的用户进程，则保存其现场；随后切换进程（默认选择 pcb[0]），将新进程记录在 current 上，切换虚拟地址空间，返回其上下文。

```
_RegSet* schedule(_RegSet *prev) {
  if(current!=NULL)
  {
      current->tf=prev;
  }
  current=&pcb[0];
  _switch(&current->as);
  return current-tf;
}
```

- _EVENT_TRAP 事件的处理，调用 schedule 并返回其现场。即每次内核自陷时，使用schedule切换进程，切换虚拟地址空间，返回上下文现场。**注：（此时只有pal一个进程）**

```
extern _RegSet* schedule(_RegSet *prev);
static _RegSet* do_event(_Event e, _RegSet* r) {
  switch (e.event) {
    case _EVENT_SYSCALL:
        return do_syscall(r);
    case _EVENT_TRAP:
        printf("lite:event_trap");
        //break;切换进程
        return schedule(r);
    case _EVENT_IRQ_TIME:
        //Log("event:IRQ_TIME\n");
        return schedule(r);
    default: panic("Unhandled event ID = %d", e.event);
  }
  return NULL;
}
```

- 修改 trap.S 中的 asm_trap，从中断控制函数返回后，先把栈顶指针切换到新进程的 trap 帧，再据此恢复现场。

```
asm_trap:
  pushal

  pushl %esp
  call irq_handle

  #addl $4, %esp
  #切换进程，irq_handle()将trap frame存在eax
```

```
  mov %eax,%esp

  popal
  addl $8, %esp

  iret
```

do_event()->schedule()返回上下文之后，irq_handle()会将上下文陷阱帧作为函数返回值存在eax中。所以，asm_trap只需要读取eax寄存器中的值。

- 运行结果

  成功运行，这里不再重复截图。

分时运行程序

此时，实验要求分时运行pal和hello进程，所以需要重写schedule函数

- proc.c 中的 schedule 函数，轮流返回仙剑奇侠传和 hello 的现场。

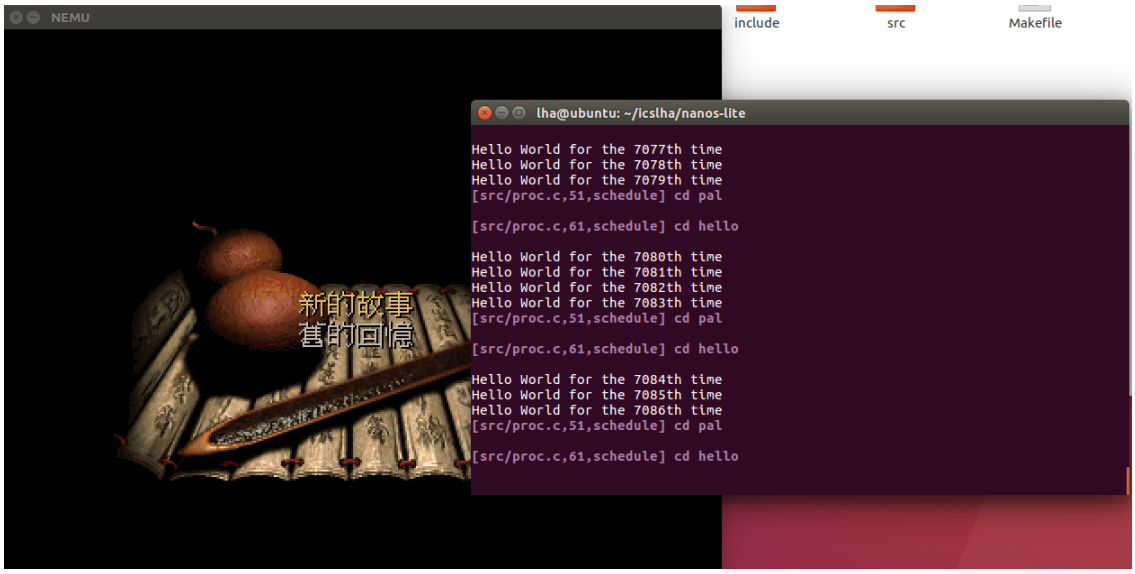
```
_RegSet* schedule(_RegSet *prev) {
  if(current!=NULL){
      current->tf=prev;
  }
  //切换到不同的进程
  current=(current==&pcb[0]?&pcb[1]:&pcb[0]);
  Log("as.ptr=0x%x\n",(uint32_t)current->as.ptr);
  _switch(&current->as);//切换地址空间
  return current->tf;
}
```

- irq.c 中的 do_event 函数，在处理完 syscall 之后，调用 schedule 函数并返回其现场。

  在每次系统调用后，使用schedule函数

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

- 运行程序



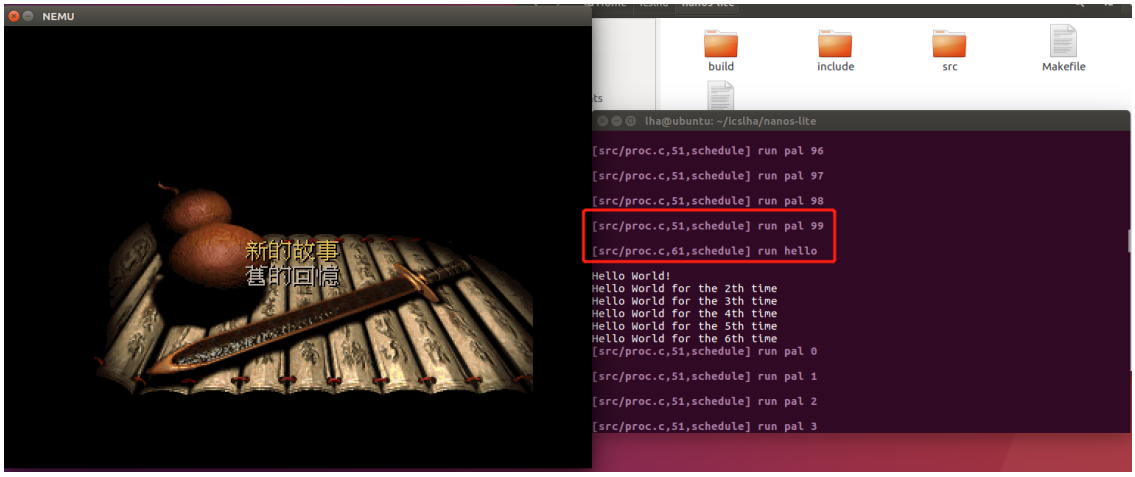
可以从Log中看出，进程一直从pal和hello中进行切换，此时pal的运行速度慢的难以忍受，所以需要加入优先级调度。

优先级调度

- 设置频率比例 frequency，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。修改 nanos-lite/src/proc.c 文件

```
_RegSet* schedule(_RegSet *prev) {
  //return NULL;
  if(current!=NULL)
  {
    current->tf=prev;
  }
  else
  {
    current=&pcb[current_game];
  }
  static int num=0;
  static const int frequent=100;
  if(current==&pcb[current_game])
  {
    Log("run pal %d\n",num);
    num++;
  }
  else
  {
    current=&pcb[current_game];
  }
  if(num==frequent)
  {
    current=&pcb[1];
    Log("run hello 1\n");
    num=0;
  }
  _switch(&current->as);//切换地址空间
  //返回上下文
   return current->tf;
}
```

- 运行结果



可以看到，每运行100次pal才运行一次hello，结果仙剑奇侠传的速度快了很多。

阶段三

添加时钟中断

- CPU中加入 INTR 引脚

  INTR高电平为开启中断，低电平为关闭中断。

```
bool INTR;
```

- dev_raise_intr()设置 INTR 为高电平

```
void dev_raise_intr() {
    //设置为高电平
    cpu.INTR=true;
}
```

- 在 exec.c 中的 exec_wrapper 函数末尾添加轮询 INTR 引脚的代码。其具体功能是，每次执行完一条 指令就查看是否有硬件中断到来。

```
#ifdef DIFF_TEST
  void difftest_step(uint32_t);
  difftest_step(eip);
#endif

//pa4 add
if(cpu.INTR & cpu.eflags.IF) {
   //Log("cd .. in\n");
   cpu.INTR = false;
   extern void raise_intr(uint8_t NO, vaddr_t ret_addr);
   raise_intr(TIME_IRQ, cpu.eip);
   update_eip();
}
```

- 修改 intr.c 中的 raise_intr 函数，保证中断处理不会被时钟中断打断

```
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.
   */
  //TODO();
  memcpy(&t1,&cpu.eflags,sizeof(cpu.eflags));
  rtl_li(&t0,t1);//赋值给t0
  rtl_push(&t0);//eflags
  cpu.eflags.IF=0;//关中断 pa4 part3
  rtl_push(&cpu.cs);//cs
  rtl_li(&t0,ret_addr);//返回地址
  rtl_push(&t0);//eip
  //门描述符地址
  vaddr_t read_begin=cpu.idtr.base+NO*sizeof(GateDesc);
  //Log("%x",cpu.idtr.base);
  //读取
  uint32_t offset_0to15 = vaddr_read(read_begin,2);
  //Log("%x",offset_0to15);
  //16-32 btye 7 8
```

```
  uint32_t offset_16to32 =vaddr_read(read_begin+sizeof(GateDesc)-2,2);
  //Log("%x",offset_16to32);
  //跳转地址
  uint32_t target_addr=(offset_16to32<<16)+offset_0to15;
  decoding.is_jmp=1;
  decoding.jmp_eip=target_addr;
  //Log("target_addr %x",target_addr);
}
```

增加一行`cpu.eflags.IF=0;//关中断 pa4 part3`，将eflags寄存器中的IF设置为0，表示在处理中断的过程中不会被其他中断所打断，从而形成嵌套。

- 添加时钟中断，首先 ASYE 注册 vectime 入口函数

```
void vectime ();
```

- 添加门描述符

```
//pa4 time_irq
idt[32] = GATE(STS_TG32, KSEL(SEG_KCODE),vectime,DPL_USER);
```

- irq_handle 和 do_event 函数中添加 _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;
    }
  }
  return next;
}
```

- 修改 nexus-am/am/arch/x86-nemu/src/trap.S

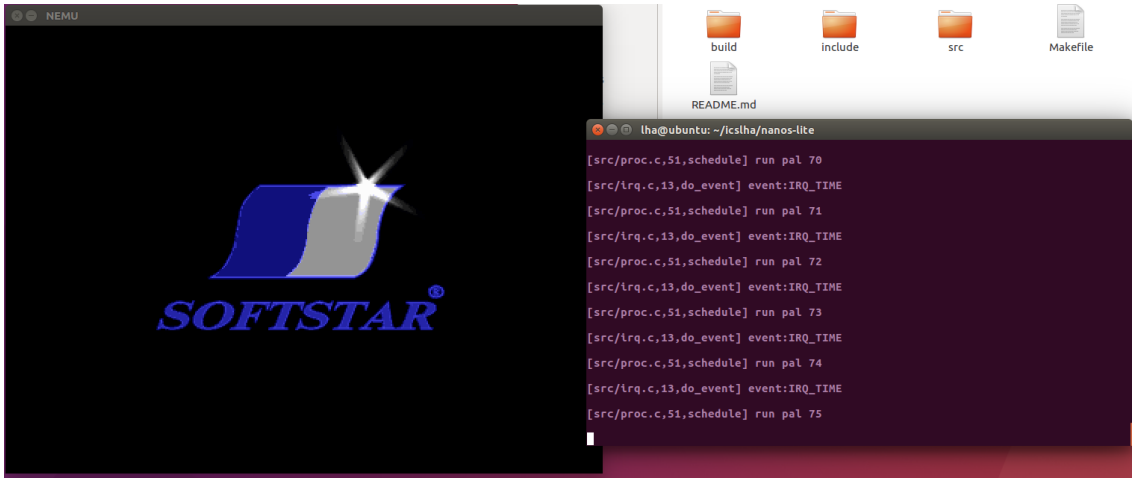
```
#----|-------entry-------|-errorcode-|---irq id---|---handler---|
.globl vecsys;    vecsys:  pushl $0;  pushl $0x80; jmp asm_trap
.globl vecnull;  vecnull:  pushl $0;  pushl   $-1; jmp asm_trap
.globl vecself;  vecself:  pushl $0;  pushl $0x81; jmp asm_trap
.globl vectime; vectime: pushl $0; pushl $32; jmp asm_trap
```

- do_event()事件分发

```
extern _RegSet* do_syscall(_RegSet *r);
extern _RegSet* schedule(_RegSet *prev);
static _RegSet* do_event(_Event e, _RegSet* r) {
  switch (e.event) {
```

```
    case _EVENT_SYSCALL:
        return do_syscall(r);
    case _EVENT_TRAP:
        printf("lite:event_trap");
        //break;切换进程
        return schedule(r);
    case _EVENT_IRQ_TIME:
        //Log("event:IRQ_TIME\n");
        return schedule(r);
    default: panic("Unhandled event ID = %d", e.event);
  }
  return NULL;
}
```

- 运行结果



Log中打印event:IRQ_TIME，表示时间中断实现成功。

展示你的计算机系统

- main.c中再加入videotest进程

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

- 设置current_game维护正在运行的进程号，实现switch_current_game

```
int current_game=0;
void switch_current_game()
{
    current_game=2-current_game;
    //0 仙剑 2 videotest
    Log("current_game=%d",current_game);
}
```

- 改写schedule

```
_RegSet* schedule(_RegSet *prev) {
  //return NULL;
  if(current!=NULL)
  {
    current->tf=prev;
```

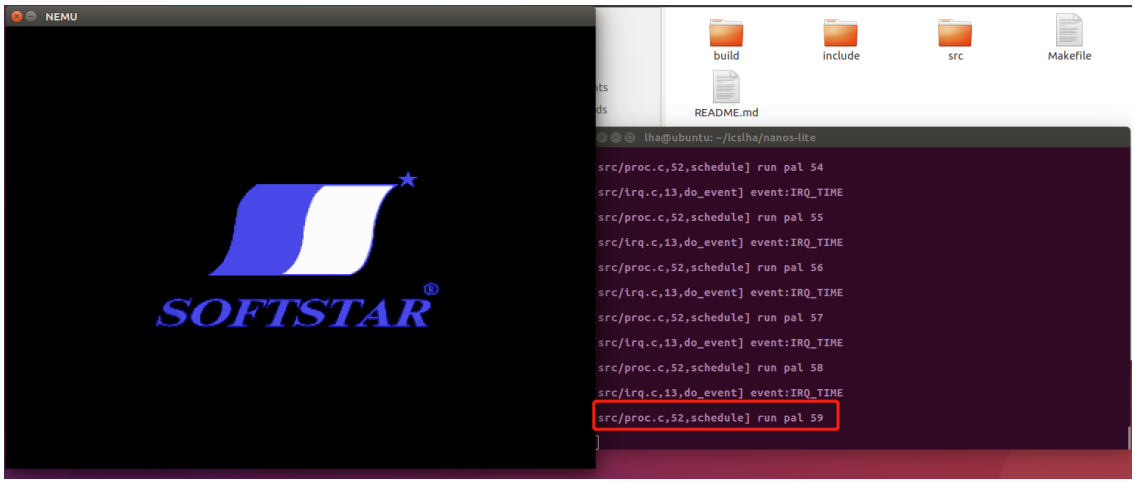
```
  }
  else
  {
    current=&pcb[current_game];
  }
  static int num=0;
  static const int frequent=100;
  if(current==&pcb[current_game])
  {
      if(current_game==0)
        Log("run pal %d\n",num);
      else if(current_game==2)
        Log("run videotest %d\n",num);
       num++;
  }
  else
  {
    current=&pcb[current_game];
  }
  if(num==frequent)
  {
    current=&pcb[1];
    Log("run hello \n");
    num=0;
  }
  _switch(&current->as);//切换地址空间
  //返回上下文
   return current->tf;
}
```

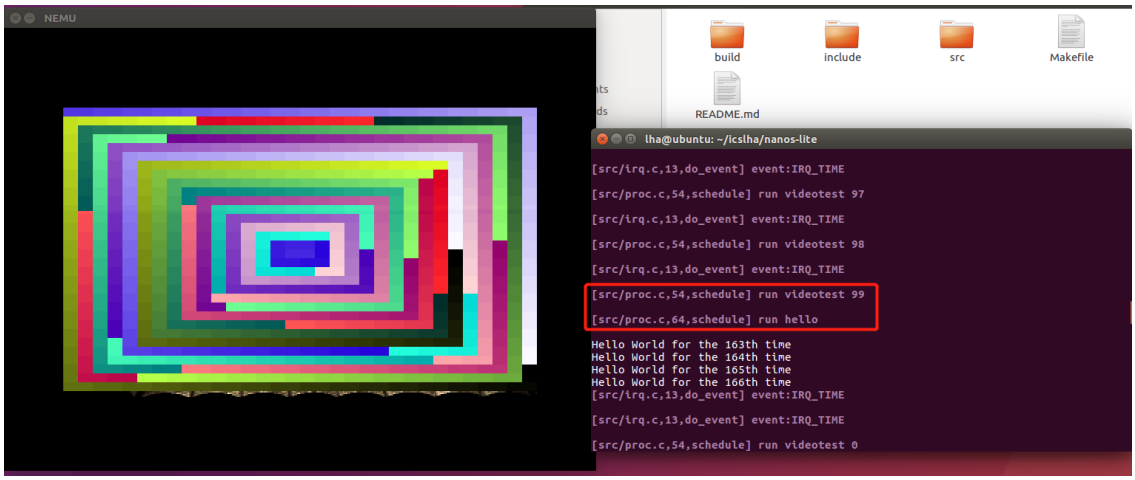
- device.c 中检测键盘按下 F12 的操作，调用 switch_current_game 进行进程切换

```
size_t events_read(void *buf, size_t len) {
  //return 0;
  int key=_read_key();
  bool down = false;
  if (key & 0x8000) {
        key ^= 0x8000;
        down = true;
  }
  char temp[20];
  if(down && key == _KEY_F12) {
      extern void switch_current_game();
      switch_current_game();
      Log("key down:_KEY_F12, switch current game!");
   }
  if (key != _KEY_NONE)
  {
    if(down){
        sprintf(temp,"kd %s\n",keyname[key]);}
    else{
        sprintf(temp,"ku %s\n",keyname[key]);}
  }
  else{
    sprintf(temp,"t %d\n",_uptime());}

  if(strlen(temp)<=len)
```

```
  {
    strncpy((char*)buf,temp,len);
    return strlen(temp);
  }
  Log("strlen(event)=%d>len=%d",strlen(temp),len);
  return 0;
}
```

- 运行结果





可以看出，时钟中断正常运行，同时当按下**F12**时，运行videotest，每100次pal和videotest运行一次hello，计算机系统被综合起来，终于完成了！！！

Bug总结

遇到的Bug太多了，这里选取两个：

- 在时钟中断时，pal能正常运行，但是helloworld却迟迟不能输出，应为前面阶段的结果正确，一开始怀疑是eflags寄存器中的IF位设置错了、asye.c中的门描述符初始化、或者asye.c中异常处理出现了问题，但统统不是。

  开始打log，发现exec_wrapper（处理中断部分）从来没有进去过，只能去检查asm_trap中所有的指令。

  最终发现**iret**指令实现错了。

```
make_EHelper(iret) {
  //TODO();
  //它将栈顶的三个元素来依次解释成 EIP,CS,EFLAGS,并恢复它们.
  rtl_pop(&cpu.eip);
  rtl_pop(&cpu.cs);
  rtl_pop(&t0);
  memcpy(&cpu.eflags,&t0,sizeof(cpu.eflags));
  decoding.jmp_eip=1;//
  decoding.seq_eip=cpu.eip;
  print_asm("iret");
}
```

我一开始将pop的值存在t1中，却用t0给eflags进行赋值。很难相信前面一直没有报错，结果在最后伏笔了。

- 在编写不朽的传奇阶段时，出现切换时 Assertion failed: screen_w>0 && screen_h>0 错误。

  现在 PA3 中编写 fs_open 时，忘记重新设置指针在文件开头。添加 set_open_offset(i,0) 语句后成功 运行。

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.

- 分页机制

  （1）分页机制需要Nanos-lite、AM 和 NEMU 配合实现。NEMU提供CR0，CR3寄存器CR0：用于是否开启分页机制;CR3:存储页表的基地址。NEMU的vaddr_read和vaddr_write用于使用分页机制，如何在虚拟地址和物理地址间进行切换。

```
uint32_t vaddr_read(vaddr_t addr, int len) {
  //return paddr_read(addr, len);
  if(PTE_ADDR(addr)!=PTE_ADDR(addr+len-1))
  {
    //Log("read跨页了，需要拼接\n");
    int len1=0x1000-OFF(addr);
    int len2=len-len1;
    //分别映射
    paddr_t paddr1=page_translate(addr,false);
    paddr_t paddr2=page_translate(addr+len1,false);

    //组合返回值
    uint32_t low=paddr_read(paddr1,len1);
    uint32_t high=paddr_read(paddr2,len2);
    uint32_t result=high<<(len1*8)|low;
    //Log("high:%x\t low:%x\t result:%x\n",high,low,result);
    return result;
  }
  else
  {
    paddr_t paddr=page_translate(addr,false);
    return paddr_read(paddr,len);
  }
}
```

```
void vaddr_write(vaddr_t addr, int len, uint32_t data) {
  //paddr_write(addr, len, data);
    if(PTE_ADDR(addr)!=PTE_ADDR(addr+len-1))
    {
      Log("write跨页了，需要拼接\n");
      int len1=0x1000-OFF(addr);
      int len2=len-len1;
      //分别映射
      paddr_t paddr1=page_translate(addr,true);
      paddr_t paddr2=page_translate(addr+len1,true);
      //组合
      uint32_t low=data&(~0u>>((4-len1)<<3));
      uint32_t high=data>>((4-len2)<<3);

      paddr_write(paddr1,len1,low);
      paddr_write(paddr2,len2,high);
    }
    else
    {
      paddr_t paddr=page_translate(addr,true);
      paddr_write(paddr,len,data);
    }
}
```

（2）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: mm(存储管理器)初始化，将TRM提供的entry作为空闲的物理页首地址，定义new_page()、free_page()，随后再通过init_mm()函数调用AM中的pte_init()

```
void init_mm() {
  pf = (void *)PGROUNDUP((uintptr_t)_heap.start);
  Log("free physical pages starting from %p", pf);

  _pte_init(new_page, free_page);
}
```

AM:_pte_init()函数用来准备一些内核页表

```
//准备一些内核页表
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;
  }

  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);
  set_cr0(get_cr0() | CR0_PG);
}
```

此函数填写了二级页表，并且设置CR3寄存器中页目录表的起始地址，设置CR0开启分页模式。

再接下来时磁盘和设备的初始化、中断和异常的初始化、文件系统的初始化。

```
int main() {
#ifdef HAS_PTE
  init_mm();
#endif
  init_ramdisk();
  init_device();
#ifdef HAS_ASYE
  Log("Initializing interrupt/exception handler...");
  init_irq();
#endif
  init_fs();
  load_prog("/bin/pal");
  load_prog("/bin/hello");
  load_prog("/bin/videotest");
  _trap();//内核自陷
  panic("Should not reach here");
}
```

最后才是加载进程。

（3）加载进程

使用load_prog()加载，在proc.c中定义

```
void load_prog(const char *filename) {
  int i = nr_proc ++;
  _protect(&pcb[i].as);
  uintptr_t entry = loader(&pcb[i].as, filename);
  _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);
}
```

1、先调用AM中的`void _protect(_Protect *p)`,创建虚实地址映射。

2、loader加载程序，loader.c提供

3、通过 umake()函数创建进程的上下文。

- 硬件中断与上下文切换保证程序的分时运行

  （1）自陷

  1、umake()后，操作系统会陷入自陷，即0x81号的中断入口。

```
 void _trap () {
    asm volatile ( "int $0x81" ) ;
}
```

  2、自陷被打包成事件

```
case 32: ev.event = _EVENT_IRQ_TIME; break ;
```

  跳转到门描述符中进行处理

```
idt [32] = GATE(STS_TG32, KSEL(SEG_KCODE) , vectime , DPL_USER)
```

  AM 根据全局标号 vectrap 进行相应的处理，内核自陷实际上完成了一次进程调度，跳到了 asm_trap 函数中。

```
globl vectime ; vectime : pushl $0 ; pushl $32 ; jmp asm_trap
```

  asm_trap 函数将栈顶指针切换到回的堆栈上`mov %eax,%esp`

  然后恢复切换进程现场`iret`

```
asm_trap:
  pushal
  pushl %esp
  call irq_handle
  #addl $4, %esp
  #切换进程，irq_handle()将trap frame存在eax
  mov %eax,%esp
  popal
  addl $8, %esp
  iret
```

  （2）时钟中断

  1、时钟中断的处理过程类似，触发 timer_intr，将 cpu 中的 INTR 引脚置成 1，的exec_wrapper 每执行完一条指令，便查看是否开中断且有硬件中断到来，当触发时钟中 断时，将在 AM 中将时钟中断打包成 IRQ_TIME 事件。

  在`Nanos-lite`中的`irq.c`中，

```
static _RegSet* do_event(_Event e, _RegSet* r) {
  switch (e.event) {
    case _EVENT_SYSCALL:
        return do_syscall(r);
    case _EVENT_TRAP:
        printf("lite:event_trap");
        //break;切换进程
        return schedule(r);
    case _EVENT_IRQ_TIME:
        Log("event:IRQ_TIME\n");
```

```
        return schedule(r);
      default: panic("Unhandled event ID = %d", e.event);
    }
    return NULL;
}
```

2、每次接收到事件_EVENT_IRQ_TIME就会调用schedule，进行进程的切换管理。

3、schedule在选定此时应该运行哪个进程后，使用`_switch(&current->as);`切换地址空间

再`return current->tf;`,返回上下文。

4`_switch()`由AM提供，切换页目录表的地址。

```
void _switch(_Protect *p) {
  set_cr3(p->ptr);
}
```

5、tf上下文也返回给AM，此时运行asm_trap就切换到了下一个进程。

其他问题

一些问题

- 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？
- 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？
- 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？

1. 页表项包含基地址信息（20bit）和标志位信息（12bit），页的大小为4K，即12bits，所以32位的低12位全为0。
2. 必须使用物理地址，不能使用虚拟地址，页表表项和 CR3 寄存器的作用是实现虚拟地址到物理地址的转换，若使用虚拟地址，进入无尽的循环。
3. （1）一级页表消耗内存大、存储难度大，一级页表需要连续的内存空间来存放所有的页表项。多级页表 只为进程实际使用的虚拟地址内存区请求页表，来减少内存使用量。

   （2）多级页表可以使页表在内存中离散存储。

空指针真的是"空"的吗？

程序设计课上老师告诉你，当一个指针变量的值等于 NULL 时，代表空，不指向任何东西。仔细想想，真的是这样吗？当程序对空指针解引用的时候，计算机内部具体都做了些什么？你对空指针的本质有什么新的认识？

NULL 是在计算中具有保留的值，用于指示指针不引用有效对象。解引用是 NULL 的未定义行为。解 引用空指针可能会导致读取或写入未映射的内存，从而触发分段错误或内存访问冲突。这可能表现为 程序崩溃，或者转换为可由程序代码捕获的软件异常。

内核映射的作用

在_protect()函数中创建虚拟地址空间的时候，有一处代码用于拷贝内核映射：

```
for (int i = 0; i < NR_PDE; i ++) {
  updir[i] = kpdirs[i];
}
```

尝试注释这处代码，重新编译并运行，你会看到发生了错误。请解释为什么会发生这个错误。

会发生缺页错误，`set_cr3(kpdirs)`，CR3寄存器中页目录表的首地址为kpdirs。注释掉_protect中拷贝内核映射的代码，会使得进程的页目录表的内核部分未被初始化，内核部分的虚拟内存->物理地址的映射消失。

灾难性的后果(这个问题有点难度)

假设硬件把中断信息固定保存在内存地址 0x1000 的位置,AM 也总是从这里开始构造 trap frame.如果发生了中断嵌套,将会发生什么样的灾难性后果?这一灾难性的后果将会以什么样的形式表现出来?如果你觉得毫无头绪,你可以用纸笔模拟中断处理的过程.

第二个中断信息可能会被覆盖或丢失，出现系统崩溃，执行错误，信息丢失，中断丢失。或者可能 出现系统死锁、系统性能下降的问题。