计算机系统设计PA4

> 1911533
>
> 朱昱函

实验目的

- 学习虚拟内存映射，并实现分页机制
- 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
- 学习硬件中断并实现时钟中断

实验内容

- PA4 实验中主要涉及三大部分。
  - 第一阶段，虚拟地址空间的作用，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，使不同的游戏与hello程序分时运行。

PA4.1

代码：在NEMU中实现分页机制

打开 HAS_PTE 后遇见 invalid opcode

```
./build/nemu -l /home/anjin/ics2017/nanos-lite/build/nemu-log.txt /home/anjin/
ics2017/nanos-lite/build/nanos-lite-x86-nemu.bin
[src/monitor/monitor.c,65,load_img] The image is /home/anjin/ics2017/nanos-lit
e/build/nanos-lite-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 18:47:21, May 22 2022
For help, type "help"
(nemu) c
[src/mm.c,24,init_mm] free physical pages starting from 0x1d6c000
invalid opcode(eip = 0x0010156c): 0f 22 d8 0f 20 c0 89 45 ...

There are two cases which will trigger this unexpected exception:
```

```
10156c: 0f 22 d8              mov    %eax,%cr3
10156f: 0f 20 c0              mov    %cr0,%eax
```

在 nanos-lite-x86-nemu.txt 中可知，是 CR3 的 mov 指令未实现。

CR3与CR0的实现

- 控制寄存器：CR0 与 CR3
- 对于 CR0，PA 中只实现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，保存页目录表的物理地址， 页目录表总是放在以4K字节为单位的存储器边界上，因此，它的地址的低12 位总为 0，不起作用，即使写上内容，也不会被理会，我们需要关前20位。系统程序员只能通过MOV指令的变体访问这些寄存器，指令允许在控制寄存器-通用寄存器执行load与store命令。

- MOV指令用于访问与修改 CR0、CR3 寄存器，Mod R/M 字节中的 reg 字段指定 了是哪个特殊寄存器，mod 字段始终为11B. r / m 字段指定所涉及的通用寄存器

```
  uint32_t CR0;
  uint32_t CR3;
} CPU_state;
```

restart函数初始化

```
cpu.CR0=0x60000011;
```

新增了CR0 与CR3 寄存器后，首先新增rtl指令实现对二者的访问与修改

```
static inline void rtl_load_cr(rtlreg_t* dest, int r){
  switch(r){
    case 0:*dest=cpu.CR0;return;
    case 3:*dest=cpu.CR3;return;
    default:assert(0);
  }
  return;
}

static inline void rtl_store_cr(const rtlreg_t* src, int r){
  switch(r){
    case 0:cpu.CR0=*src;return;
    case 3:cpu.CR3=*src;return;
    default:assert(0);
  }
  return;
}
```

```
make_DHelper(mov_load_cr){
  decode_op_rm(eip,id_dest,false,id_src,false);
  rtl_load_cr(&id_src->val,id_src->reg);
#ifdef DEBUG
  snprintf(id_src->str, 5,"%%cr%d",id_dest->reg);
#endif
}

make_DHelper(mov_store_cr){
  decode_op_rm(eip,id_src,false,id_dest,false);
#ifdef DEBUG
  snprintf(id_dest->str, 5,"%%cr%d",id_src->reg);
#endif
}
```

```
make_EHelper(mov_store_cr);
```

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

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

根据nanos-lite-x86-nemu.txt 的反汇编代码可知，在$eip == 0x10156c$ 处完成 CR3 的设置，在$eip == 0x10157d$处完成 CR0 的设置

nanos-lite文件夹下make run，因为上次发生invalid opcode 的eip 位于 0x10156c，故在该位置设置监视点。

info r查看寄存器内容，发现此时CR0=0x60000011，CR3=0x0

输入c运行至断点

```
Success record : nr token=1,type=260,str=
[src/monitor/debug/expr.c,94,make_token] match rules[1] = "0x[1-9A-Fa-f][0-9A-
Fa-f]*" at position 6 with len 8: 0x10156c
Success record : nr token=2,type=258,str=10156c
Hardware watchpoint 0:$eip==0x10156c
Old value:0
New value:1
```

然后单步调试直到0x10157d，此时CR0与CR3均已经设置完毕

```
CR0=0xe0000011.CR3=0x1d6a000
```

虚拟地址的转换

相关变量与函数

虚拟内存机制是一个软硬件协同的机制，操作系统负责物理内存的管理， MMU 将其落实到硬件中。为实现分页系统，我们先查看两个十分重要的头文件：

nemu/include/memory/mmu.h（以下简称 mmu.h）

nexus-am/am/arch/x86-nemu/include/x86.h（以下简称 x86.h）

二者分别对应NEMU与AM 这两个维度上的分页机制的维护，对比查看两文件， 可以发现有许多相似性。

虽然在代码实现中并不需要用到x86.h，但为了与 mmu.h 进行对比，便于 理解和实现之后的代码，这里先说明x86.h 中的各个宏。

1）x86.h 说明

① 数据结构

PDE：页目录表表项（Page Directory Entry），本质为 uint32_t 结构

PTE：二级页表表项（Page Table Entry），本质为 uint32_t 结构

② 相关宏：函数

PDX(va)：根据虚拟地址 va 获取其页目录表索引（Page Directory Index）

PTX(va)：根据 va 获取页表索引（Page Table Index）

OFF(va)：根据 va 获取页内偏移量

PGADDR(d,t,o)：根据 PDX，PTX，OFF 计算其虚拟地址

PTE_ADDR(pte)：根据页表表项获取前 20 位的基地址

③ 相关宏：标志位

PTE_P：即 0x0001，标志 Present 位

2）mmu.h 数据结构说明

CR0：对应控制寄存器CR0，本质为Union结构

CR3：对应控制寄存器CR3，本质为 Union 结构

PDE：页目录表表项（Page Directory Entry），本质为Union 结构

PTE：二级页表表项（Page Table Entry），本质为 Union 结构

备注：Union 与Struct的区别已经在 PA1 中说明了。以PDE为例，其所占字 节数仍保持32bit，但可通过Union对这32位进行分别的读取或写入，如 PDE.present直接访问PDE表项最末的present标志位，PDE.val访问完整的32bit。

代码实现

仿照 x86.h 定义辅助宏，在 page_translate 中将会使用到

```
#define PTE_ADDR(pte)   ((uint32_t)(pte) & ~0xfff)
#define PDX(va)     (((uint32_t)(va) >> 22) & 0x3ff)
#define PTX(va)     (((uint32_t)(va) >> 12) & 0x3ff)
#define OFF(va)     ((uint32_t)(va) & 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 iswrite){
  CR0 cr0=(CR0)cpu.CR0;
  if(cr0.paging && cr0.protect_enable){
    CR3 cr3=(CR3)cpu.cr3;

    PDE* pgdirs=(PDE*)PTE_ADDR(cr3.val);
    PDE pde=(PDE)paddr_read((uint32_t)(pgdirs+PDX(addr)),4);
    Assert(pde.present,"addr=0x%x",addr);

    PTE* ptab=(PTE*)PTE_ADDR(pde.val);
    PTE pte=(PTE)paddr_read((uint32_t)(ptab+PTX(addr)),4);
    Assert(pte.present,"addr=0x%x",addr);

    pde.accessed=1;
    pte.accessed=1;
    if(iswrite)
    {
      pte.dirty=1;
    }

    paddr_t paddr=PTE_ADDR(pde.val) | OFF(addr);
    return paddr;
  }
  return addr;
}
```

vaddr_read 与 vaddr_write

```c
uint32_t vaddr_read(vaddr_t addr, int len) {
  if(PTE_ADDR(addr)!=PTE_ADDR(addr+len-1)){
    printf("error:the data pass two pages:addr=0x%x,len=%d!\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) {
  if(PTE_ADDR(addr)!=PTE_ADDR(addr+len-1)){
    printf("error:the data pass two pages:addr=0x%x,len=%d!\n"
    assert(0);
  }
  else{
    paddr_t paddr=page_translate(addr,true);
    paddr_write(paddr, len, data);
  }
}
```

运行dummy HIT GOOD TRAP

```
[src/mm.c,24,init_mm] free physical pages starting from 0x1d6c000
[src/main.c,19,main] 'Hello World!' from Nanos-lite
[src/main.c,20,main] Build time: 16:42:22, May 25 2022
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x1029e0, end = 0x1d47d0
1, size = 29643553 bytes
[src/main.c,27,main] Initializing interrupt/exception handler...
[src/fs.c,39,init_fs] set FD_FB size=480000
fd = 13
loader end!
nemu: HIT GOOD TRAP at eip = 0x00100032
```

运行仙剑奇侠传时，就出现了数据跨越虚拟页内存的情况。

```
error:the data pass two pages:addr=0x1c43ffd,len=4!
nemu: src/memory/memory.c:67: vaddr read: Assertion `0' failed.
Makefile:46: recipe for target 'run' failed
make[1]: *** [run] Aborted (core dumped)
make[1]: Leaving directory '/home/anjin/ics2017/nemu'
/home/anjin/ics2017/nexus-am/Makefile.app:35: recipe for target 'run' failed
make: *** [run] Error 2
```

数据跨过虚拟页边界

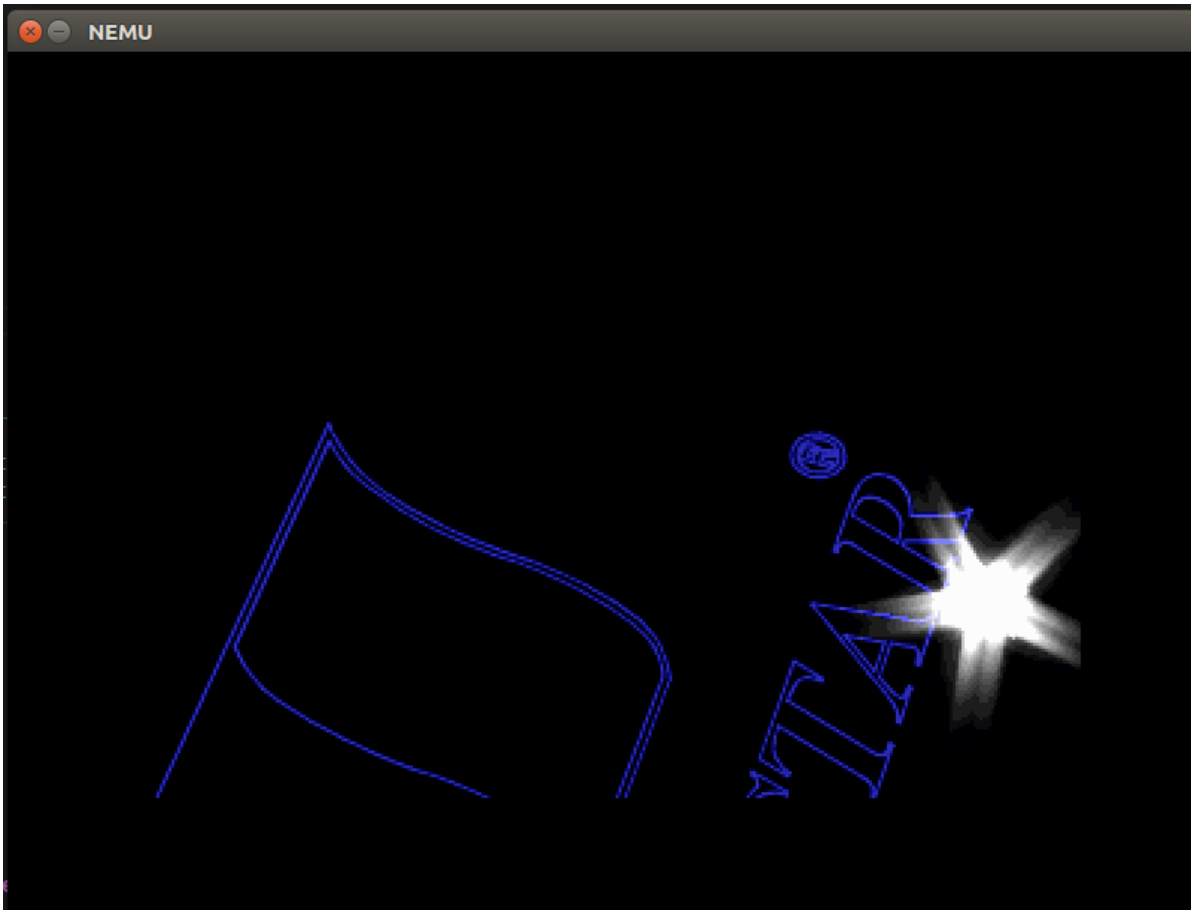
对于数据跨越虚拟内存边界的情况，需要进行两次页级地址转换。

在 vaddr_read 中将两次读取的字节进行整合，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

```c
uint32_t vaddr_read(vaddr_t addr, int len) {
  if(PTE_ADDR(addr)!=PTE_ADDR(addr+len-1)){
    int num1=0x1000-OFF(addr);
    int num2=len-num1;
    paddr_t paddr1=page_translate(addr,false);
    paddr_t paddr2=page_translate(addr+num1,false);

    uint32_t low=paddr_read(paddr1,num1);
    uint32_t high=paddr_read(paddr2,num2);

    uint32_t result=high<<(num1*8) | low;
    return result;
  }
  else{
    paddr_t paddr=page_translate(addr,false);
    return paddr_read(paddr, len);
  }
}
```

```c
void vaddr_write(vaddr_t addr, int len, uint32_t data) {
  if(PTE_ADDR(addr)!=PTE_ADDR(addr+len-1)){
    int num1=0x1000-OFF(addr);
    int num2=len-num1;
    paddr_t paddr1=page_translate(addr,true);
    paddr_t paddr2=page_translate(addr+num1,true);

    uint32_t low=data & (~0u >> ((4-num1) << 3));
    uint32_t high=data>>((4-num2) << 3);

    paddr_write(paddr1,num1,low);
    paddr_write(paddr2,num2,high);
    return ;
  }
  else{
    paddr_t paddr=page_translate(addr,true);
    paddr_write(paddr, len, data);
  }
}
```

成功运行仙剑奇侠传



代码：让用户程序运行在分页机制上

修改navy-apps/Makefile.compile

```
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

```
#endif

  init_fs();

  load_prog("/bin/dummy");
  //uint32_t entry = loader(NULL, "/bin/pal");
  //uint32_t entry = loader(NULL, NULL);
  printf("loader end!\n");
  //((void (*)(void))entry)();

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

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

```
void _map(_Protect *p, void *va, void *pa) {
  if(OFF(va)||OFF(pa))
  {
    printf("not aligned!!");
    return;
  }
  PDE *pgdir=(PDE*)p->ptr;
  PTE *pgtab=NULL;

  PDE *pde=pgdir+PDX(va);
  if(!(*pde&PTE_P)){
    pgtab=(PTE*)(palloc_f());
    *pde=(uintptr_t)pgtab | PTE_P;
  }
  pgtab=(PTE*)PTE_ADDR(*pde);

  PTE *PTE=pgtab+PTX(va);
  *pte=(uintptr_pa) | PTE_P;
}
```

nanos-lite/src/loader.c

```
  return (uintptr_t)DEFAULT_ENTRY;*/
  int fd= fs_open(filename, 0, 0);
  Log("filename=%s,fd = %d",filename, fd);
  //fs_read(fd,DEFAULT_ENTRY, fs_filesz(fd));
  int size=fs_filesz(fd);
  int ppnum=size/PGSIZE;
  if(size%PGSIZE!=0)
    ppnum++;
  void *pa=NULL;
  void *va=DEFAULT_ENTRY;
  for(int i=0;i<ppnum;i++)
  {
    pa=new_page();
    _map(as,va,pa);
    fs_read(fd,pa,PGSIZE);
    va+=PGSIZE;
  }
  fs_close(fd);
  return (uintptr_t)DEFAULT_ENTRY;
```

运行dummy hit good trap

```
For help, type "help"
(nemu) c
[src/mm.c,24,init_mm] free physical pages starting from 0x1d8d000
[src/main.c,20,main] 'Hello World!' from Nanos-lite
[src/main.c,21,main] Build time: 17:32:02, May 25 2022
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x102ba0, end = 0x1d47ec
1, size = 29643553 bytes
[src/main.c,28,main] Initializing interrupt/exception handler...
[src/fs.c,39,init_fs] set FD_FB size=480000
[src/loader.c,19,loader] filename=/bin/dummy,fd = 13
nemu: HIT GOOD TRAP at eip = 0x00100032
```

代码：在分页机制上运行仙剑奇侠传

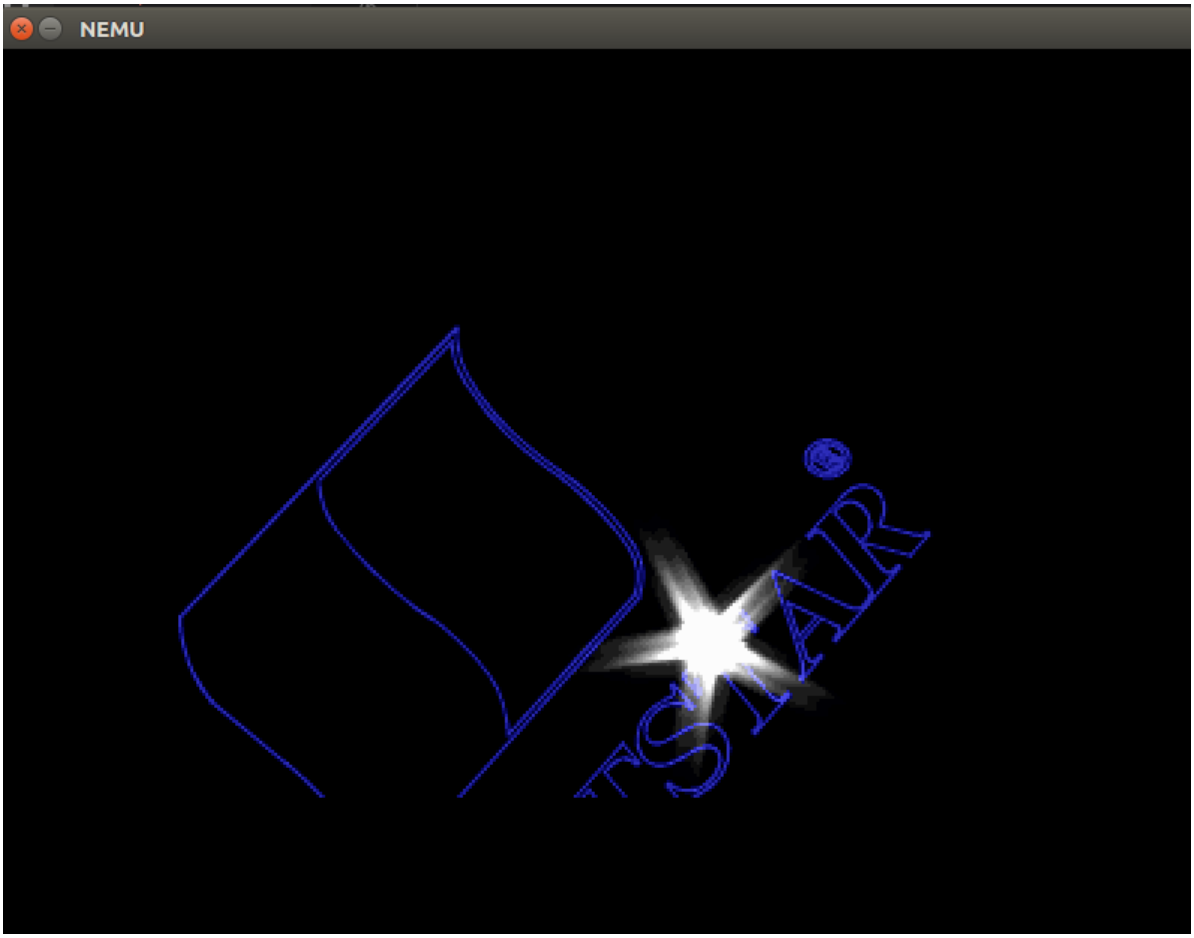
nanos-lite/src/mm.c

```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){
      uint32_t first=PGROUNDUP(current->max_brk);
      uint32_t end=PGROUNDDOWN(new_brk);
      if((new_brk & 0xfff)==0){
        end-=PGSIZE;
      }
      for(uint32_t va=first;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;
}
```

sys_brk

```c
int sys_brk(int addr){
  extern int mm_brk(uint32_t new_brk);
  return mm_brk(addr);
}
```

成功运行仙剑奇侠传



PA4.2

代码：实现内核自陷

nanos-lite/src/main.c

```
  init_fs();

  //load_prog("/bin/pal");
  //uint32_t entry = loader(NULL, "/bin/pal");
  //uint32_t entry = loader(NULL, NULL);
  //printf("loader end!\n");
  //((void (*)(void))entry)();
  _trap()

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

nanos-lite/src/proc.c

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

_trap 触发 int 0x81 指令

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

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

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

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

```
void vecsys();
void vecnull();
void vecself();
```

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

```
idt[0x81] = GATE(STS_TG32, KSEL(SEG_KCODE), vecself, DPL_USER);
```

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

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

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

```
switch (tf->irq) {
  case 0x80: ev.event = _EVENT_SYSCALL; break;
  case 0x81: ev.event = _EVENT_TRAP; break;
  default: ev.event = _EVENT_ERROR; break;
}
```

nanos-lite/src/irq.c

```
extern _RegSet* do_syscall(_RegSet *r);
static _RegSet* do_event(_Event e, _RegSet* r) {
  switch (e.event) {
    case _EVENT_SYSCALL:
      return do_syscall(r);
    case _EVENT_TRAP:
      printf("event:self-trapped\n");
      return NULL;
    default: panic("Unhandled event ID = %d", e.event);
  }
  return NULL;
}
```

```
[src/mm.c,43,init_mm] free physical pages starting from 0x1d8d000
[src/main.c,20,main] 'Hello World!' from Nanos-lite
[src/main.c,21,main] Build time: 18:13:07, May 25 2022
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x102ae0, end = 0x1d47e0
1, size = 29643553 bytes
[src/main.c,28,main] Initializing interrupt/exception handler...
[src/fs.c,39,init_fs] set FD_FB size=480000
event:self-trapped
[src/main.c,41,main] system panic: Should not reach here
nemu: HIT BAD TRAP at eip = 0x00100032
```

代码：实现上下文切换

- umake 函数 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，我 们也不会从_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。

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

```
_RegSet *_umake(_Protect *p, _Area ustack, _Area kstack, void
  extern void* memcpy(void *,const void *, int);
  int arg1=0;
  char *arg2=NULL;
  memcpy((void*)ustack.end-4,(void*)arg2,4);
  memcpy((void*)ustack.end-8,(void*)arg2,4);
  memcpy((void*)ustack.end-12,(void*)arg1,4);
  memcpy((void*)ustack.end-16,(void*)arg1,4);

  _RegSet tf;
  tf.eflags=0x02;
  tf.cs=8;
  tf.eip=(uintptr_t)entry;
  void* ptf=(void*)(ustack.end-16-sizeof(_RegSet));
  memcpy(ptf,(void*)&tf,sizeof(_RegSet));

  return (_RegSet*)ptf;
}
```

- 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。 注意，这一进程的上下文是在加载程序时由 umake()人工创建的，在 schedule() 中决定切换到它，在 ASYE 的 asm_trap()才真正恢复这一现场。

nanos-lite/src/proc.c

```
_RegSet* schedule(_RegSet *prev) {
  if(current!=NULL)
    current->tf=prev;
  current=&pcb[0];
  Log("PTR=0x%x\n",(uint32_t)current->as.ptr);
  _switch(&current->as);
  return current->tf;
}
```

nanos-lite/src/irq.c

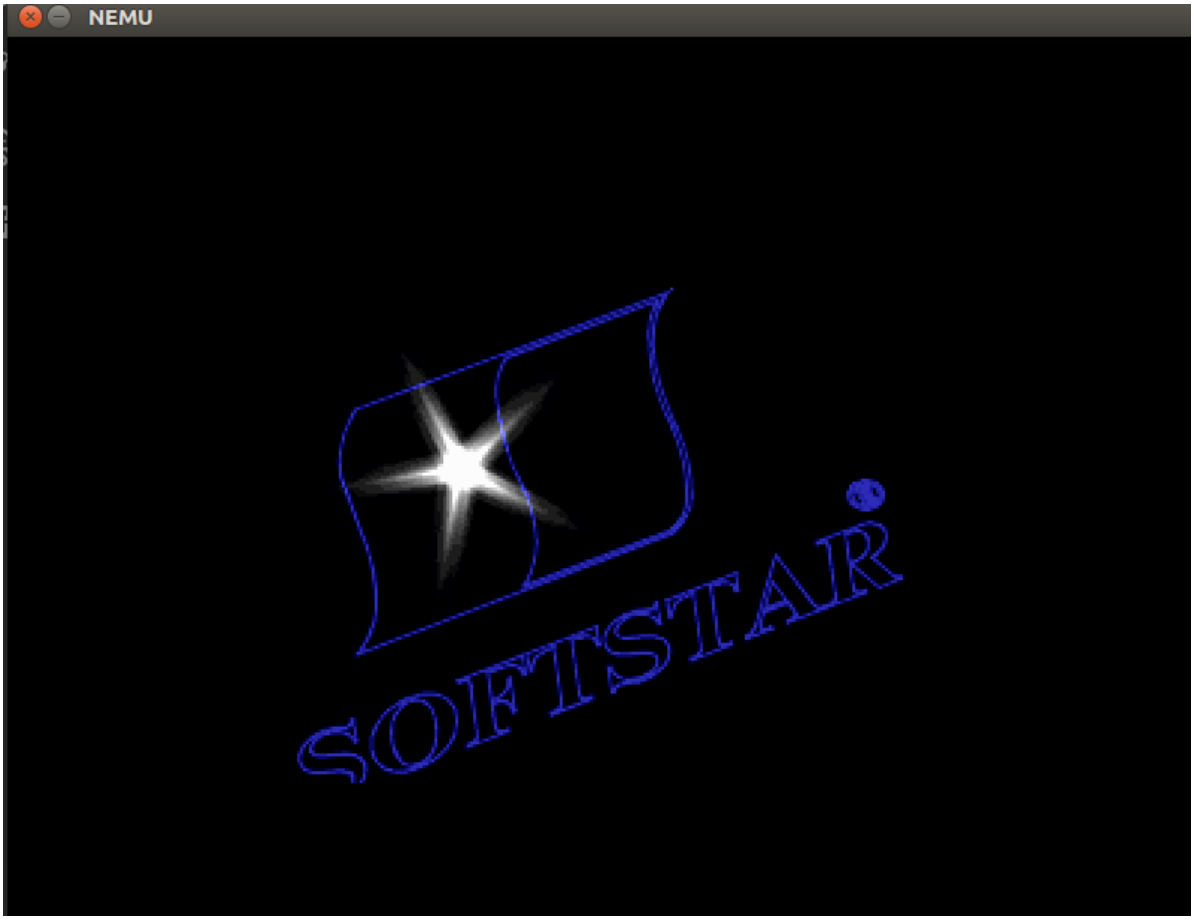
对于_EVENT_TRAP 事件，调用 schedule() 并返回其现场

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

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

```
pushal

pushl %esp
call irq_handle

#addl $4, %esp
movl %eax,%esp

popal
addl $8, %esp

iret
```

可以运行仙剑奇侠传



代码：分时运行仙剑奇侠传和hello程序

nanos-lite/src/main.c

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

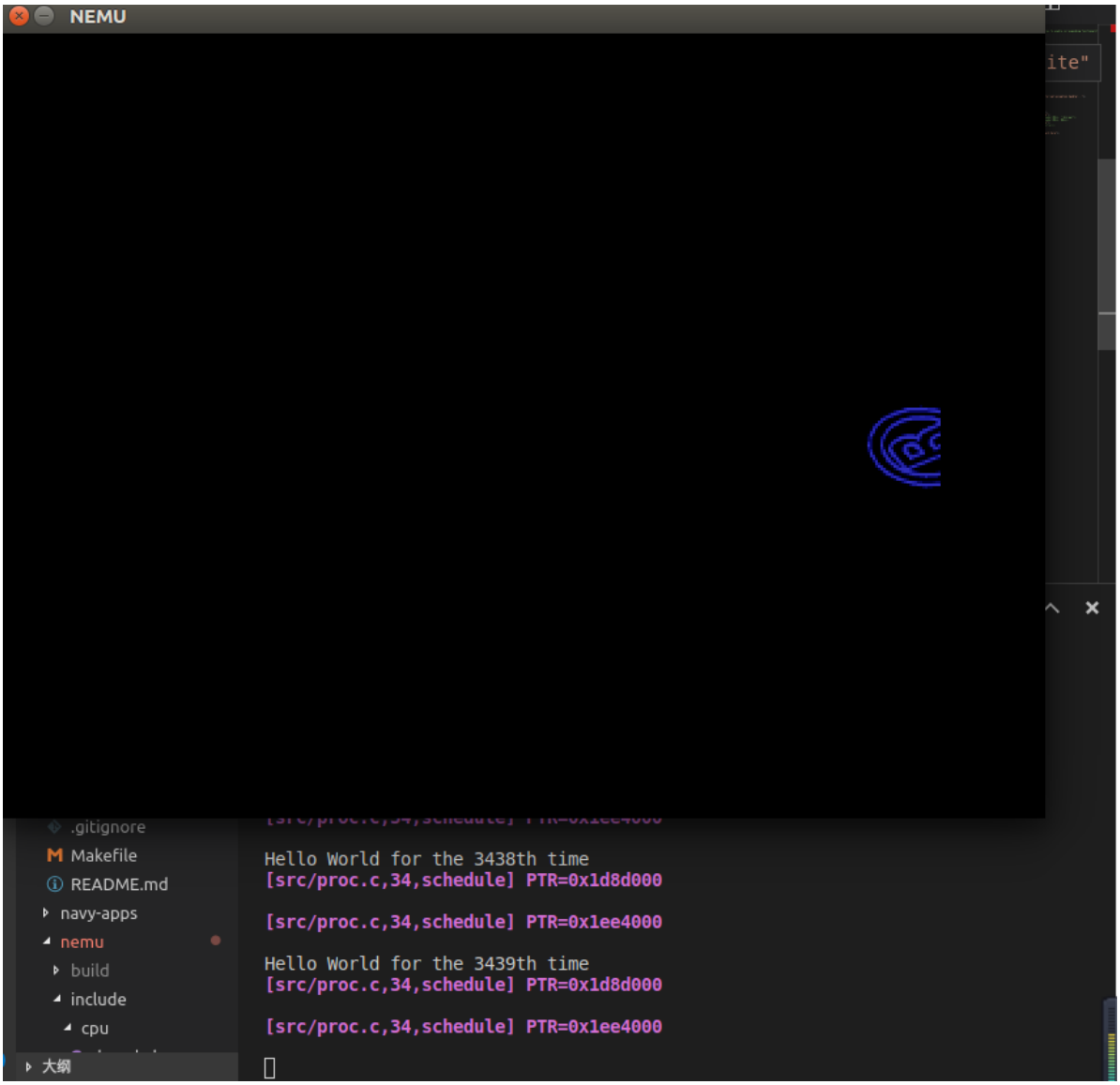
nanos-lite/src/proc.c

```
_RegSet* schedule(_RegSet *prev) {
  if(current!=NULL)
    current->tf=prev;
  //current=&pcb[0];
  current= (current==&pcb[0]? &pcb[1] : &pcb[0]);
  Log("PTR=0x%x\n",(uint32_t)current->as.ptr);
  _switch(&current->as);
  return current->tf;
}
```

nanos-lite/src/irq.c

```c
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("event:self-trapped\n");
      do_syscall(r);
      return schedule(r);
    default: panic("Unhandled event ID = %d", e.event);
  }
  return NULL;
}

void init_irq(void) {
  _asye_init(do_event);
}
```

可以看到hello和仙剑奇侠传同时运行，但仙剑运行极为缓慢这是因为每次系统调用都导致了进程切换。



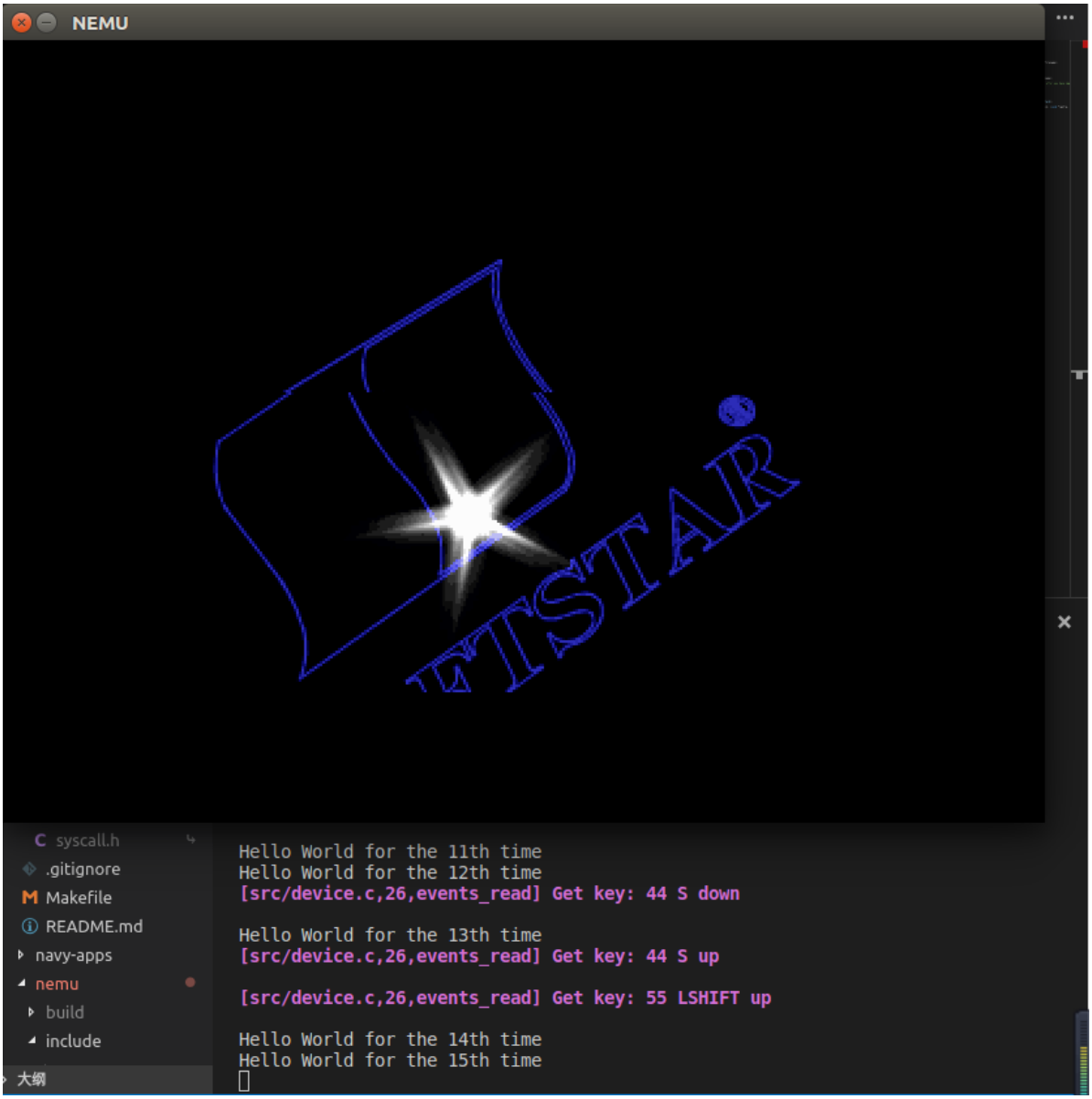
代码：优先级调度

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。

nanos-lite/src/proc.c

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

  static int num=0;
  static const int frequency=1000;
  if(current==&pcb[0])
    num++;
  else
    current=&pcb[0];

  if(num==frequency){
    current=&pcb=[1];
    num=0;
  }
  _switch(&current->as);
  return current->tf;
}
```

仙剑与hello分时运行，仙剑速度和之前相比显著提升。



PA4.3

代码：添加时钟中断

cpu 结构中增加 INTR 引脚

nemu/include/cpu/reg.h

```
  uint32_t CR0;
  uint32_t CR3;
  bool INTR;
} CPU_state;
```

dev_raise_intr()设置 INTR 为高电平

nemu/src/cpu/intr.c

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

exec_wrapper()末尾添加轮询 INTR 引脚的代码 nemu/src/cpu/exec/exec.c

```
#define IDEXW(id, ex, w)   {concat(decode_, id), concat(exec_,
#define IDEX(id, ex)       IDEXW(id, ex, 0)
#define EXW(ex, w)         {NULL, concat(exec_, ex), w}
#define EX(ex)             EXW(ex, 0)
#define EMPTY              EX(inv)
#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();
}
```

raise_intr()保存 eflags 后关中断

这是为了保证中断处理时不会被时钟中断打断。

nemu/src/cpu/intr.c

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

ASYE 中添加时钟中断

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

```
void vectime();
```

```
// ------------------- system call ------------------------
idt[0x80] = GATE(STS_TG32, KSEL(SEG_KCODE), vecsys, DPL_USER
idt[0x81] = GATE(STS_TG32, KSEL(SEG_KCODE), vecself, DPL_USE
idt[32] = GATE(STS_TG32, KSEL(SEG_KCODE), vectime, DPL_USER)
set_idt(idt, sizeof(idt));
```

```
  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;
  }
```

```
.globl vecself;  vecself:  pushl $0;  pushl $0x81; jmp asm_tra
.globl vectime;  vectime:  pushl $0;  pushl $  32; jmp asm_tra
```

do_event()事件分发

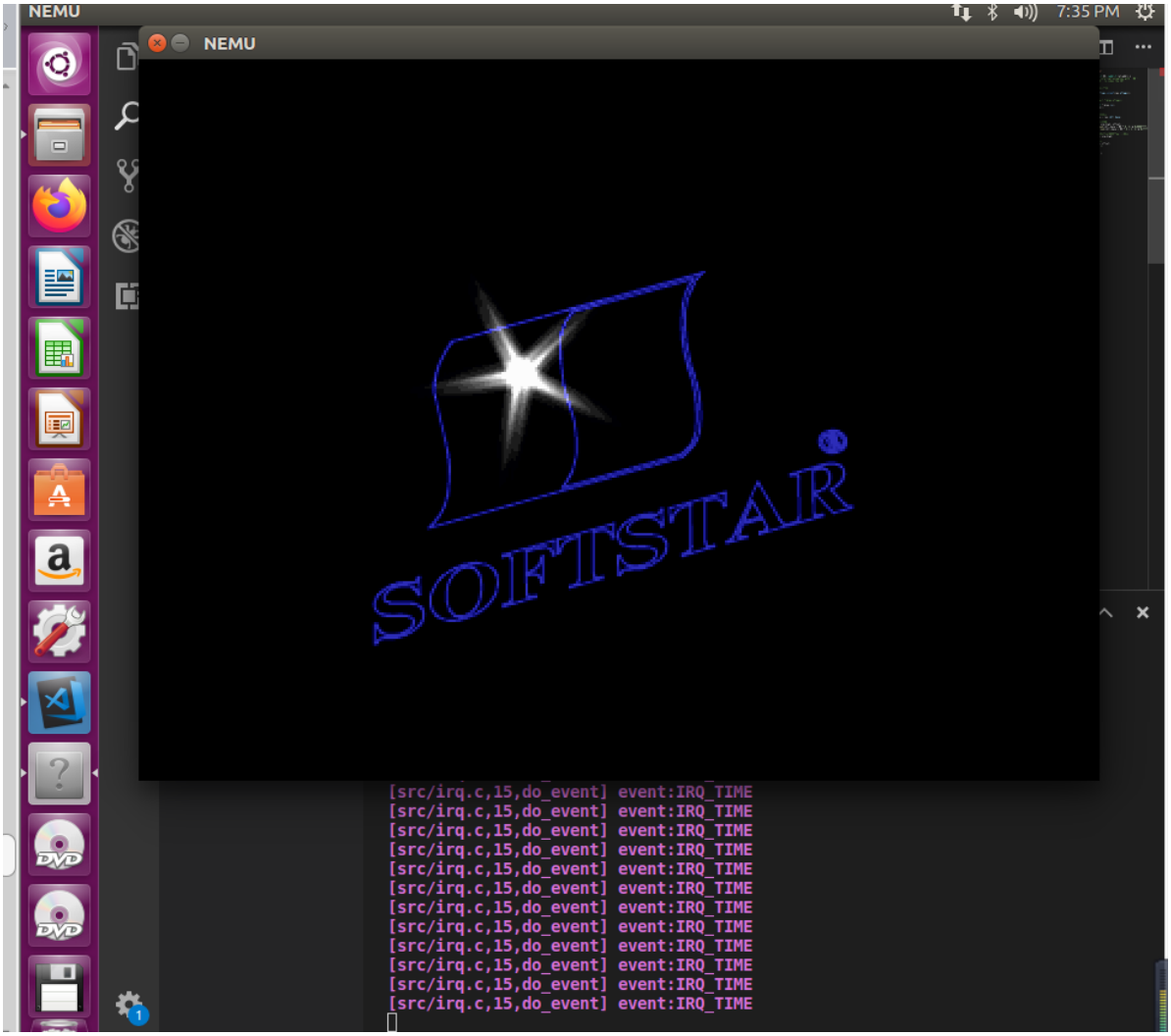
```
extern _RegSet* do_syscall(_RegSet *r);
extern _RegSet* schedule(_RegSet *prev);
static _RegSet* do_event(_Event e, _RegSet* r) {
  switch (e.event) {
    case _EVENT_SYSCALL:
      //return do_syscall(r);
      do_syscall(r);
      return schedule(r);
    case _EVENT_TRAP:
      printf("event:self-trapped\n");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

umake()设置用户进程开中断

FL_IF 在 x86.h 中定义。

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

return (_RegSet*)ptf;
```

分时运行仙剑奇侠传和hello程序。这时进程切换不在系统调用时发生，而在时钟中断时尝试进程切换



必答题

必答题

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.

- 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。 首先，NEMU 提供 CR0 与 CR3 寄存器来辅助实现分页机制，CR0用于开启 分页，CR3 记录页表基地址。随后，MMU 进行分页地址的转换，在代码中表现 为 NEMU 的 vaddr_read()与 vaddr_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*pte_init()*来准备内核页表。pte_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。
- 分页机制下用户程序的加载
    - 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，加载时根据页面分配函数 new_page()分配新的物理页，利用分页机制将 用户程序链接到固定虚拟地址 0x8048000，但加载到 new_page()分配的不同的物理 位置去执行；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。
- 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，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， 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

nanos-lite/src/main.c

```
load_prog("/bin/pal");
load_prog("/bin/hello");
load_prog("/bin/videotest");
//uint32_t entry = loader(NULL, "/bin/pal");
//uint32_t entry = loader(NULL, NULL);
printf("loader end!\n");
//((void (*)(void))entry)();
_trap();

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

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

```
int current_game=0;
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) {
  if(current!=NULL)
    current->tf=prev;
  else
    current=&pcb[current_game];

  static int num=0;
  static const int frequency=1000;
  if(current==&pcb[current_game])
    num++;
  else
    current=&pcb[current_game];

  if(num==frequency){
    current=&pcb[1];
    num=0;
  }
  _switch(&current->as);
  return current->tf;
}
```

成功从仙剑和hello切换到了videotest