南开大学
Nankai University

南　　开　　大　　学

计　算　机　学　院

计算机系统设计实验报告

PA4-虚实交错的魔法: 分时多任务

邵琦

年级：2020 级

专业：计算机科学与技术

指导教师：卢冶

2023 年 5 月 30 日

目录

一、 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。

二、 实验内容

1. 学习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。
2. 实现内核自陷、上下文切换与分时多任务。
3. 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。
4. 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。

三、 阶段一

(一) i386 分页机制

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。

增加 CR0 和 CR3 寄存器

```
  //cr0
  uint32_t CR0;

  //cr3
  uint32_t CR3;
```

在 monitor.c 中的 restart 函数初始化 CR0 值。

初始化 CR0 值

```
static inline void restart() {
  /* Set the initial instruction pointer. */
  cpu.eip = ENTRY_START;

  cpu.cs = 8;

  cpu.CR0=0x60000011;

  unsigned int origin = 2;
  memcpy(&cpu.eflags, &origin, sizeof(cpu.eflags));

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

运行后后遇见 invalid opcode，在 nanos-lite-x86-nemu.txt 中可知，是 CR3 的 mov 指令未实现。我们需要实现新的指令函数，来完成对两个控制寄存器的操作。

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)

图 1: mov 指令

mov 指令

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

在 decode.h 和 decode.c 中注册并实现译码函数。

注册译码函数

```
make_DHelper(mov_load_cr);
make_DHelper(mov_store_cr);
```

实现译码函数

```
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,true,id_dest,false);
#ifdef DEBUG
  snprintf(id_dest->str,5,"%%cr%d",id_dest->reg);
#endif
}
```

在 all_instr.h 中注册指令:

注册指令

```
make_EHelper(mov_store_cr);
```

在 data-move.c 中实现执行函数:

实现执行函数

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

注册 opcode_table:

注册 opcode_table

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

(二) 虚拟地址的转换

在 memory.c 定义辅助宏:

定义辅助宏

```
#define PTXSHFT 12
#define PDXSHFT 22

#define PTE_ADDR(pte) ((uint32_t)(pte) & ~0xfff)

```

```
#define PDX(va) (((uint32_t)(va) >> PDXSHFT) & 0x3ff)

#define PTX(va) (((uint32_t)(va) >> PTXSHFT) & 0x3ff)

#define OFF(va) ((uint32_t)(va) & 0xfff)
```

编写 page_translate 函数，进行页面地址转换。

页面地址转换

```
paddr_t page_translate(vaddr_t addr, bool iswrite) {
  CR0 cr0 = (CR0)cpu.CR0;
  if(cr0.paging && cr0.protect_enable) {
    CR3 crs = (CR3)cpu.CR3;

    PDE *pgdirs = (PDE*)PTE_ADDR(crs.val);
    PDE pde = (PDE)paddr_read((uint32_t)(pgdirs + PDX(addr)), 4);

    PTE *ptable = (PTE*)PTE_ADDR(pde.val);
    PTE pte = (PTE)paddr_read((uint32_t)(ptable + PTX(addr)), 4);
    //printf("hhahah%x, jhhh%x\n", pte.present, addr);
    Assert(pte.present, "addr=0x%x", addr);

    pde.accessed=1;
    pte.accessed=1;
    if(iswrite) {
      pte.dirty=1;
    }
    paddr_t paddr = PTE_ADDR(pte.val) | OFF(addr);
    // printf("vaddr=0x%x, paddr=0x%x\n", addr, paddr);
    return paddr;
  }
        return addr;
}
```

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

实现读写地址时的虚拟地址转换

```
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", addr, len);
    assert(0);
  }
  else {
    paddr_t paddr = page_translate(addr, false);
    return paddr_read(paddr, len);
  }
  // return paddr_read(addr, len);
}

```

```
13 void vaddr_write(vaddr_t addr, int len, uint32_t data) {
14   if(PTE_ADDR(addr) != PTE_ADDR(addr+len-1)) {
15     printf("error: the data pass two pages:addr=0x%x, len=%d!\n", addr, len);
16     assert(0);
17   }
18   else {
19     paddr_t paddr = page_translate(addr, true);
20     paddr_write(paddr, len, data);
21   }
22   // paddr_write(addr, len, data);
23 }
```

在 page_translate 函数当中，参数 addr 代表待处理页面的地址，iswrite 是读或写的标志位，若需要进行的是读操作，该参数就被设置为 false，若需要进行写操作，就被设置为 true。

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。

此时，dummy 可以正常运行，但是在运行仙剑奇侠传时，出现跨页虚拟内存情况，会导致运行失败。

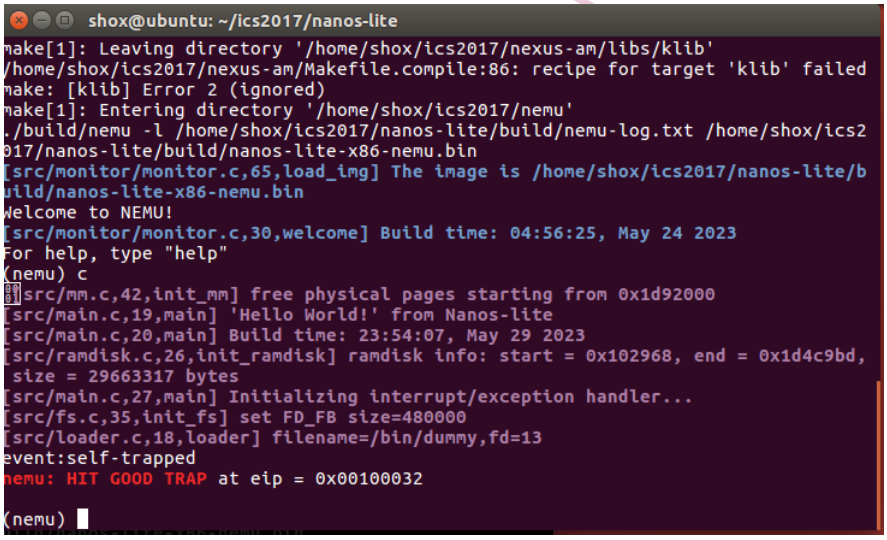

图 2: 运行结果

此时，我们对 vaddr_read 和 vaddr_write 函数进行修改。

如果出现了跨页的情况，就分高低页进行读取，用两个局部变量记录高低页的字节数，paddr 记录高低页对应物理地址，再进行整合读取。跨页写入的情况和跨页读取类似。

修改实现读写地址时的虚拟地址转换

```
1 uint32_t vaddr_read(vaddr_t addr, int len) {
2   if(PTE_ADDR(addr) != PTE_ADDR(addr + len -1)) {
3     // printf("error: the data pass two pages:addr=0x%x, len=%d!\n", addr,
        len);
4     // assert(0);
5     int num1 = 0x1000 - OFF(addr);
6     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);
  }
  // return paddr_read(addr, 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", addr,
        len);
    // assert(0);
    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);
  }
  // paddr_write(addr, len, data);
}
```

此时，仙剑奇侠传也可以运行。

图 3: 运行结果

(三) 让用户程序运行在分页机制上

首先我们修改 Makefile.compile:

修改 Makefile.compile

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

修改 loader.c:

修改 loader.c

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

修改 main.c:

修改 main.c

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

```

```
  extern void load_prog(const char *filename);
  load_prog("/bin/dummy");
```

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。

实现 _map 函数

```
void _map(_Protect *p, void *va, void *pa) {
  if(OFF(va) || OFF(pa)) {
    // printf("page not aligned\n");
    return;
  }

  PDE *dir = (PDE*) p -> ptr;
        PTE *table = NULL;
        PDE *pde = dir + PDX(va);
        if(!(*pde & PTE_P)) {
                table = (PTE*) (palloc_f());
                *pde = (uintptr_t) table | PTE_P;
        }
        table = (PTE*) PTE_ADDR(*pde);
        PTE *pte = table + PTX(va);
        *pte = (uintptr_t) pa | PTE_P;
}
```

修改 loader.c 让 loader 按页加载。

loader 按页加载

```
uintptr_t loader(_Protect *as, const char *filename) {
  // TODO();
  // ramdisk_read(DEFAULT_ENTRY, 0, RAMDISK_SIZE);
  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;
}
```

运行结果如下:

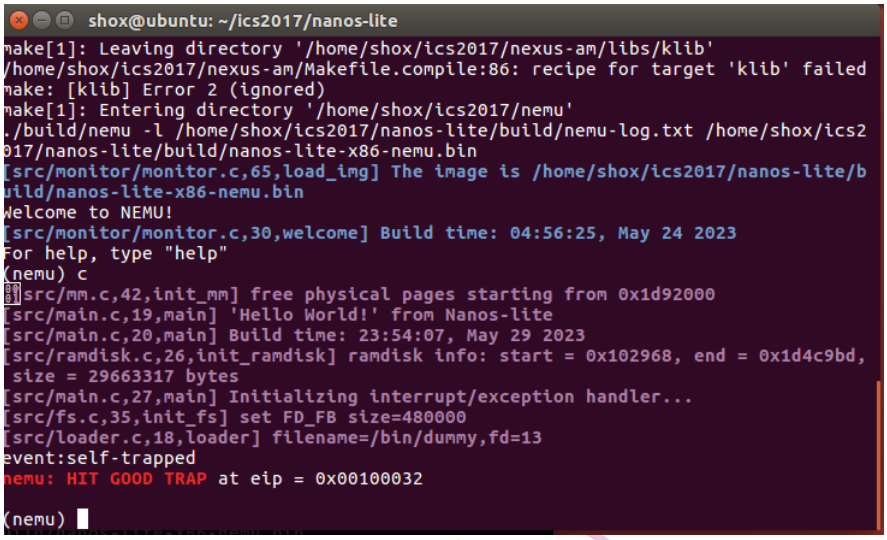

图 4: 运行结果

(四) 在分页机制上运行仙剑奇侠传

完成 mm_brk 函数，实现堆内存的映射:

堆内存的映射

```
/* 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) {
      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;
}
```

在 syscall.c 中修改与其对应的 sys_brk 函数:

syscall.c 修改与其对应的 sys_brk 函数

```
int sys_brk(int addr)
{
  //return 0;
  extern int mm_brk(uint32_t new_brk);
  return mm_brk(addr);
}
```

最后成功运行仙剑奇侠传:

图 5: 运行结果

四、 阶段二

(一) 实现内核自陷

首先我们修改 main.c 和 proc.c:

修改 main.c

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

extern void load_prog(const char *filename);
//load_prog("/bin/dummy");
```

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

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

修改 proc.c

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

修改 asye.c:

修改 asye.c

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

定义内核自陷入口函数 vecself, 该函数压入错误码和异常号 irq (0x81), 并跳转到 asm_trap 中:

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
```

在 ASYE 中定义 vecself 函数:

定义 vecself 函数

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

在 _asye_init 函数中填写 0x81 的门描述符:

填写 0x81 的门描述符

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

在 irq_handle 中将 0x81 异常封装成 _EVENT_TRAP 事件:

将 0x81 异常封装成 _EVENT_TRAP 事件

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

在 do_event 中根据事件再次分发:

根据事件再次分发

```
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;
}
```

此时输出“HIT BAD TRAP”相关信息。

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

图 6: 运行结果

(二) 实现上下文切换

完成 _umake 函数，该函数在加载程序时完成上下文的创建，初始化并返回陷阱帧的指针。

完成 _umake 函数

```
_RegSet *_umake(_Protect *p, _Area ustack, _Area kstack, void *entry, char *
    const argv[], char *const envp[]) {
  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 | FL_IF;
  //tf.cs = 0;
  //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;
}
```

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。

完成 schedule 函数

```
_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;
}
```

完善对 _EVENT_TRAP 事件的处理，调用 schedule 并返回其现场。

完善对 _EVENT_TRAP 事件的处理

```
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);
    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。

修改 trap.S 中的 asm_trap

```
asm_trap:
  pushal

  pushl %esp
  call irq_handle

```

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

popal
addl $8, %esp

iret
```

最后成功运行仙剑奇侠传。

图 7: 运行结果

(三) 分时运行程序

修改 main.c，载入待运行程序：

修改 main.c

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

extern void load_prog(const char *filename);
//load_prog("/bin/dummy");
load_prog("/bin/pal");
load_prog("/bin/hello");

_trap();
```

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

修改 proc.c 中的 schedule 函数

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

修改 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。

修改 irq.c 中的 do_event 函数

```
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;
}
```

修改 proc.c 文件的 schedule 函数，设置 frequency 为频率比例，num 为当前程序运行次数，仙剑奇侠传运行 frequency 次，才切换到 hello 程序运行一次，并重新计数。

修改 proc.c 文件的 schedule 函数

```
_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;
  }
  // current = (current == &pcb[0]? &pcb[1] : &pcb[0]);
  // Log("ptr = 0x%x\n", (uint32_t)current -> as.ptr);
  _switch(&current -> as);
  return current -> tf;
}
```

此时仙剑奇侠传与 hello 分时运行，速度有显著提升。

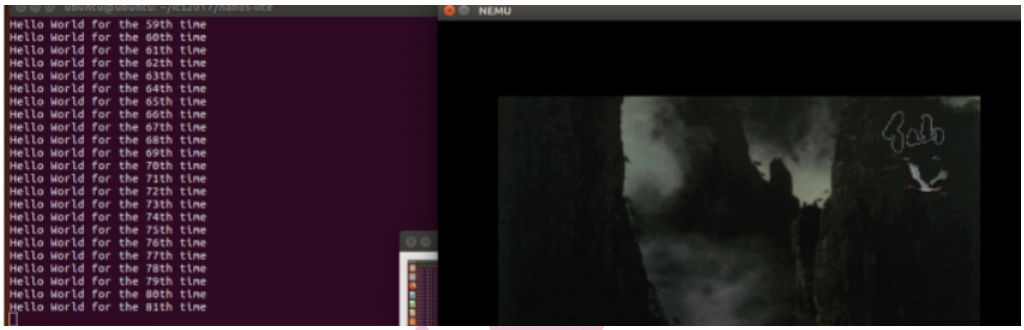

图 8: 运行结果

五、 阶段三

(一) 添加时钟中断

首先我们在寄存器结构中增加 INTR。

在寄存器结构中增加 INTR

```

  bool INTR;
} CPU_state;
```

接着我们在 dev_raise_intr 中设置 INTR:

dev_raise_intr 中设置 INTR

```
void dev_raise_intr() {
  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。

添加轮询 INTR 引脚的代码

```
#define TIME_IRQ 32

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

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

修改 intr.c 中的 raise_intr 函数

```
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.
   */

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

之后，我们添加时钟中断，首先 ASYE 注册 vectime 入口函数：

ASYE 注册 vectime 入口函数

```
void vectime();
```

添加门描述符：

添加门描述符

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

在 irq_handle 和 do_event 函数中添加 _EVENT_IRQ_TIME 事件：

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;
}
```

添加 _EVENT_IRQ_TIME 事件

```
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;
}
```

在 trap.S 中对 vectime 函数进行注册:

对 vectime 函数进行注册

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

编写 _umake 函数，设置 eflags 寄存器，以使用户开中断:

编写 _umake 函数

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

```
  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 | FL_IF;
  //tf.cs = 0;
  //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;
}
```

最终运行结果如下:

图 9: 运行结果

六、 编写不朽的传奇

修改 main.c，使其加载 pal、hello、videotest。

修改 main.c

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

  extern void load_prog(const char *filename);
```

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

  _trap();
```

设置 current_game 维护当前运行程序的进程号，编写 switch_current_game 进行进程切换:

编写 switch_current_game 进行进程切换

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

修改 schedule 函数，使其在 current_game 和 hello 间切换:

修改 schedule 函数

```
_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;
  }
  // current = (current == &pcb[0]? &pcb[1] : &pcb[0]);
  // Log("ptr = 0x%x\n", (uint32_t)current -> as.ptr);
  _switch(&current -> as);
  return current -> tf;
}
```

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

在 device.c 中检测键盘按下 F12 的操作

```
  if(down && key == _KEY_F12) {
```

```
    extern void switch_current_game();
    switch_current_game();
    Log("key down:_KEY_F12, switch current game0!");
  }
```

最终运行结果如下:

图 10: 运行结果

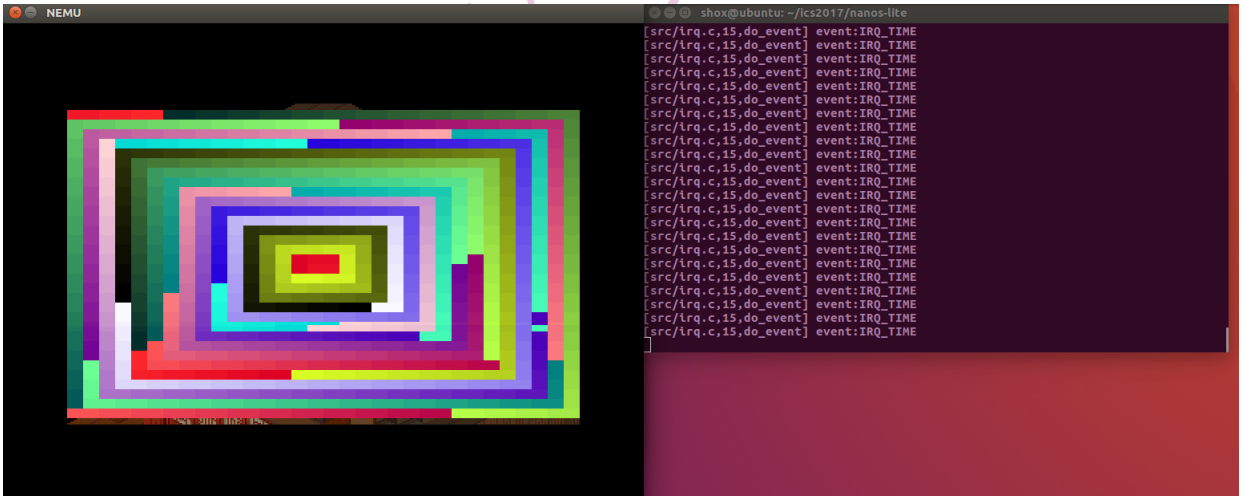

图 11: 运行结果

七、 遇到的问题以及解决方案

切换 current_game 时，出现如下问题:

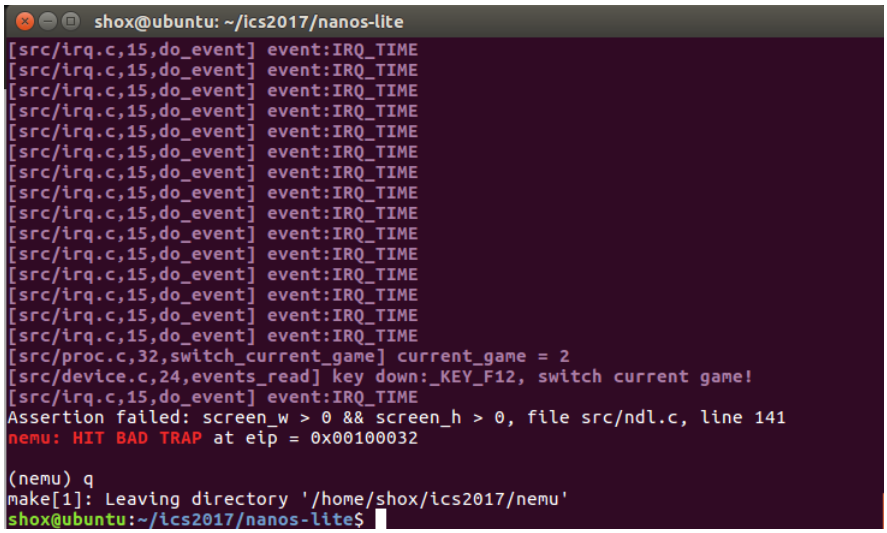

图 12: 切换 current_game 时遇到的问题

解决方案：经过调试，发现测试只运行 current_game 与 hello 程序时：videotest 与 hello 分时运行，结果正常；仙剑奇侠传与 hello 分时运行，结果也正常。测试只运行 current_game 与 hello 程序时：videotest 与 hello 分时运行，结果正常；仙剑奇侠传与 hello 分时运行，结果也正常。排除进程切换时，返回的上下文出错，导致读取的数据有误。

因此，问题应该为 disinfo 读取有误。最终发现在 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。

八、 必答题

图 13: 必答题

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。NEMU 的 vaddr_read 与 vaddr_write 进行分页地址的转换。

必答题

```
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", addr,
        len);
    // assert(0);
    int num1 = 0x1000 - OFF(addr);
    int num2 = len - num1;
    paddr_t paddr1 = page_translate(addr, false);
    paddr_t paddr2 = page_translate(addr + num1, false);
```

```
9
10      uint32_t low = paddr_read(paddr1, num1);
11      uint32_t high = paddr_read(paddr2, num2);
12
13      uint32_t result = high << (num1 * 8) | low;
14      return result;
15    }
16    else {
17      paddr_t paddr = page_translate(addr, false);
18      return paddr_read(paddr, len);
19    }
20    // return paddr_read(addr, len);
21  }
22
23  void vaddr_write(vaddr_t addr, int len, uint32_t data) {
24    if(PTE_ADDR(addr) != PTE_ADDR(addr+len-1)) {
25      // printf("error: the data pass two pages:addr=0x%x, len=%d!\n", addr,
          len);
26      // assert(0);
27      if(PTE_ADDR(addr) != PTE_ADDR(addr + len -1)) {
28        int num1 = 0x1000-OFF(addr);
29        int num2 = len -num1;
30        paddr_t paddr1 = page_translate(addr, true);
31        paddr_t paddr2 = page_translate(addr + num1, true);
32
33        uint32_t low = data & (~0u >> ((4 - num1) << 3));
34        uint32_t high = data >> ((4 - num2) << 3);
35
36        paddr_write(paddr1, num1, low);
37        paddr_write(paddr2, num2, high);
38        return;
39      }
40    }
41    else {
42      paddr_t paddr = page_translate(addr, true);
43      paddr_write(paddr, len, data);
44    }
45    // paddr_write(addr, len, data);
46  }
```

为启动分页机制，操作系统还需要准备内核页表。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。

必答题

```
1  void* new_page(void) {
2    assert(pf < (void *)_heap.end);
```

```
  void *p = pf;
  pf += PGSIZE;
  return p;
}

void free_page(void *p) {
  panic("not implement yet");
}
```

必答题

```
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);
}
```

接下来是磁盘和设备的初始化，中断异常初始化，以及文件系统初始化，如下：

必答题

```
#ifdef HAS_PTE
  init_mm();
```

```
#endif

  Log("'Hello World!' from Nanos-lite");
  Log("Build time: %s, %s", __TIME__, __DATE__);

  init_ramdisk();

  init_device();

#ifdef HAS_ASYE
  Log("Initializing interrupt/exception handler...");
  init_irq();
#endif

  init_fs();
}
```

Nanos-lite 通过 load_prog 加载用户程序。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。

必答题

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

  extern void load_prog(const char *filename);
  //load_prog("/bin/dummy");
  load_prog("/bin/pal");
  load_prog("/bin/hello");
  load_prog("/bin/videotest");

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

必答题

```
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，操作系统将陷入自陷，即一个 0x81 号的中断入口。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。asm_trap 函数将栈顶指针切换到回的堆栈上，然后恢复切换进程现场。

必答题

```
asm_trap:
  pushal

  pushl %esp
  call irq_handle

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

  popal
  addl $8, %esp

  iret
```

加入时钟中断，每隔 10ms，触发 timer_intr，将 cpu 中的 INTR 引脚置成 1。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，并保存当前程序的上下文，Nanos-lite 收到该事件后调用 schedule() 进行进程调度。

我们利用 schedule 进行调度，schedule 函数会跳转到 _switch 函数 _switch 函数的 set_cr3 记录页表页目录项的基地址，进行虚拟地址空间的切换，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。

必答题

```
_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;
  }
  // current = (current == &pcb[0]? &pcb[1] : &pcb[0]);
  // Log("ptr = 0x%x\n", (uint32_t)current -> as.ptr);
  _switch(&current -> as);
  return current -> tf;
}
```

九、 其他问题

(一) 问题一

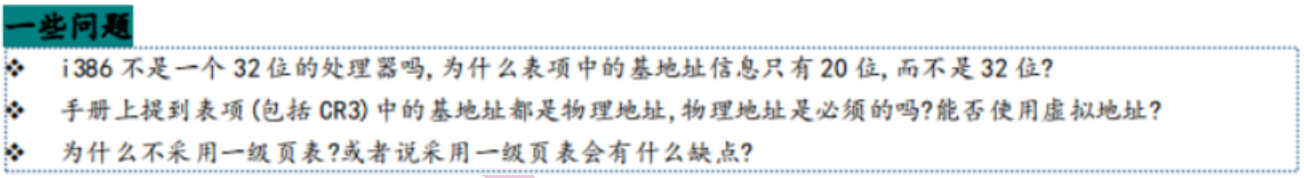

图 14: 问题一

1. 页表项包含基地址信息（20bit）和标志位信息（12bit），标志位实现了 i386 的页级保护机制，页面首地址低 12 位全为零。

2. 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，则无法获取它们的物理地址。

3. 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，而且多级页表可以使页表在内存中离散存储。

(二) 问题二

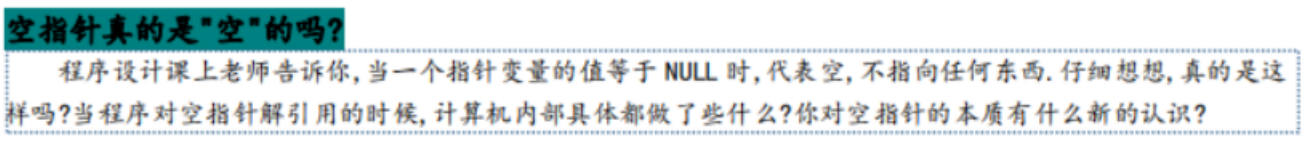

图 15: 问题二

当一个指针变量的值等于 NULL，它表示该指针没有指向任何有效的内存地址。这通常被认为是指针的空值或无效值。当程序对空指针进行解引用时，即尝试访问空指针指向的内存地址

时，会触发一个未定义行为。在计算机内部，当空指针被解引用时，通常会导致一个异常或错误的信号。这可能是由于操作系统对内存访问的保护机制，或者由于硬件对非法内存访问的检测。

对于空指针的本质，它实际上是一个特殊的值，用于表示指针的无效状态。在大多数编程语言中，NULL 或 nullptr 是一个预定义的常量，用于表示空指针。它并不指向任何有效的内存地址，而是被用作一个标记，以便程序能够检测和处理指针是否有效。

(三) 问题三

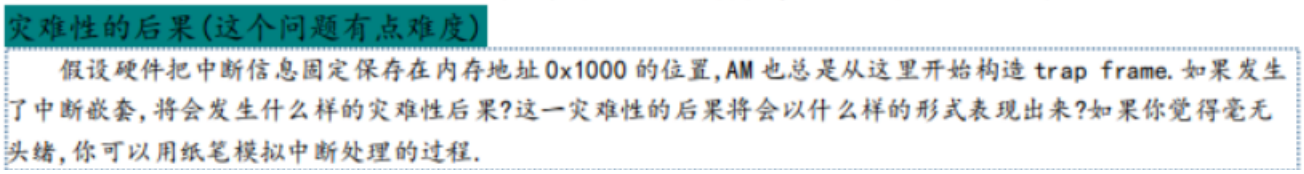

图 16: 问题三

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。