计算机系统设计 PA4

- 学号：2013750
- 姓名：管昀玫
- 专业：计算机科学与技术

1 概述

1.1 实验目的

- 学习虚拟内存映射，实现分页机制
- 学习上下文切换基本原理，并实现上下文切换、进程调度、分时多任务
- 学习硬件中断，实现时钟中断

1.2 实验内容

1. 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. 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
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。

2 阶段一

2.1 实现分页机制

2.1.1 虚拟内存

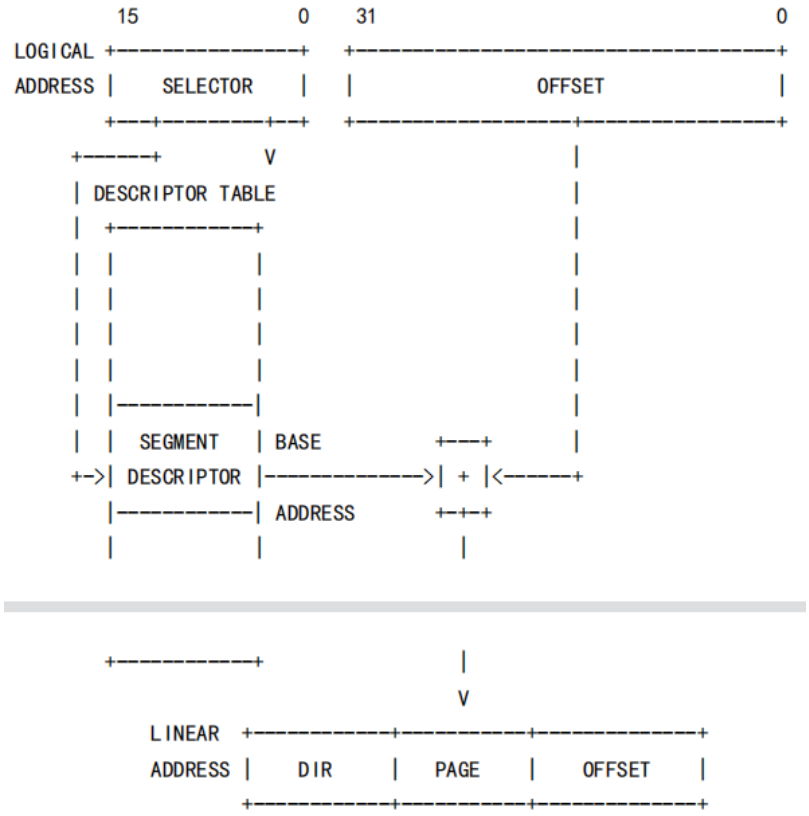
我们希望在多任务操作系统中实现这样的功能：操作系统需要记录内存的分配情况，需要运行一个新程序的时候，就给它分配一片空闲的内存位置，把它加载到这一内存位置上即可。为了管理内存，提出了虚拟内存的概念。

所谓虚拟内存，就是在真正的内存(也叫物理内存)之上的一层专门给程序使用的抽象。有了虚拟内存之后，程序只需要认为自己运行在虚拟地址上就可以了，真正运行的时候，才把虚拟地址映射到物理地址。这样，我们只要把程序链接到一个固定的虚拟地址,加载程序的时候把它们加载到不同的物理地址，并维护好虚拟地址到物理地址的映射关系，就可以实现让程序认为自己在某个固定的内存位置的同时，把程序加载到不同的内存位置去执行。

在引入虚拟内存之后，EIP就是一个虚拟地址了。操作系统难以干涉指令具体执行过程，因此需要在处理器和存储器之间添加一个新的硬件模块 MMU(Memory Management Unit，内存管理单元)，它是虚拟内存机制的核心，肩负起这一机制最重要的地址映射功能。程序运行的时候，MMU 就会进行地址转换，把程序的虚拟地址映射到操作系统希望的物理地址。

2.1.2 分段

关于MMU如何进行地址映射，最简单的方法为$物理地址 = 虚拟地址 + 偏移量$，这种最朴素的方式就是段式虚拟内存管理机制，简称分段机制。i386引入了短描述符、段选符、全局描述符表（GDT），全局描述符表寄存器（GDTR）等概念，以完善分段机制。如下所示：

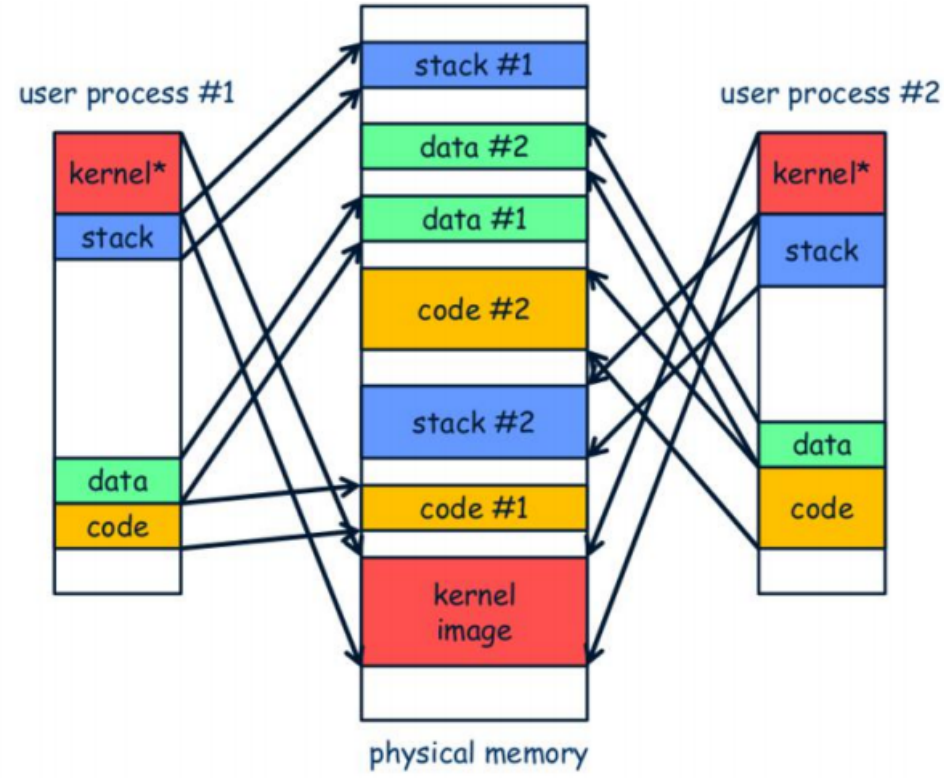


现在的绝大部分操作系统都不再使用分段机制，i386将段基地址设成 0，长度设成 4GB，这样看来就像没有段的概念一样，这就是 i386 手册中提到的"扁平模式"。

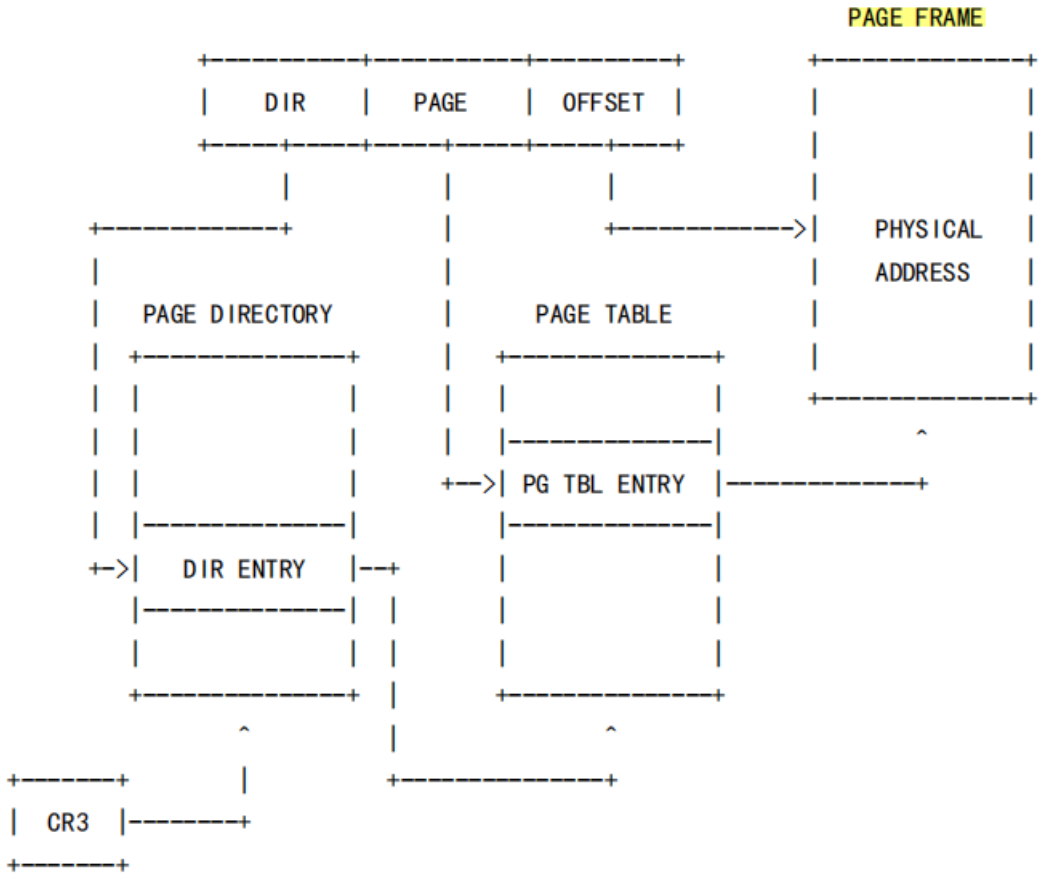
2.1.3 分页

段的粒度太大，不容易实现按需分配，因此提出了分页机制。

在分页机制中，小片段称为页面，在虚拟地址空间和物理地址空间中也分别称为虚拟页和物理页。分页机制做的事情，就是把一个个的虚拟页分别映射到相应的物理页上。分页机制引入了一个叫"页表"的结构，页表中的每一个表项记录了一个虚拟页到物理页的映射关系，来把不必连续的页面重新组织成连续的虚拟地址空间。因此，每当加载程序的时候，操作系统就给程序分配相应的物理页(物理页之间不必连续)，并为程序准备一个新的页表，在页表中填写程序用到的虚拟页到分配到的物理页的映射关系。等到程序运行的时候，操作系统就把之前为这个程序填写好的页表设置到MMU 中，MMU 就会根据页表的内容进行地址转换，把程序的虚拟地址空间映射到操作系统所希望的物理地址空间上。



我们在操作系统这么课中已经学到，i386吧物理内存划分为4KB为单位的页面，同时采用二级页表的结构，每一张页目录和页表都有 1024 个表项，每个表项的大小都是 4 字节，除了包含页表(或者物理页)的基地址，还包含一些标志位信息。页表会存放在内存中。CR3寄存器专门用于存放目录的基地址，其转换过程如下所示：



> 问题：

> 1. 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?
> 2. 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?
> 3. 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?

回答:

1. 在32位的x86架构中，使用的是分页机制来进行虚拟内存管理。分页机制将内存划分为固定大小的页面，通常是4KB。每个页面都有一个对应的表项，用于记录该页面的物理地址或其他相关信息。

   在分页表项中，常见的结构是由32位组成的，包括页面基地址和其他控制信息。页表的大小为4KB = 2^12且页与页直接没有重叠的区域，物理地址总线只有20位，这意味着最多可以寻址2^20个物理内存页面，即1MB。

   因此，在i386架构中，表项中的基地址字段只使用了低20位，用于存储物理页面的起始地址。高12位则用于存储其他控制信息，例如页面权限、缓存策略等。通过使用分页机制和表项中的基地址加上偏移量，可以将虚拟内存映射到物理内存。

2. 在手册上有这样一句话：

   > The first paging structure used for any translation is located at the **physical address in CR3**.

   因此物理地址是必须的。因为这些表项（包括CR3）的作用是将虚拟地址翻译到物理地址，如果写一个虚拟地址的话无法翻译。每个**页表项（PML4E, PDPTE, PDE，PTE）里的基址，都是物理地址。**

   但是，整个**页转换表结构**是存放内存里，属于虚拟地址。也就是：**页转换表结构需要进行内存映射。**

3. 一级页表存在以下一些缺点：
   - 内存开销：一级页表需要为整个地址空间创建一个非常大的表，而i386架构下的地址空间大小为4GB。如果使用一级页表，将需要4GB大小的连续内存来存储页表项，这对于当时的硬件资源来说是非常昂贵和不实际的。
   - 内存访问：一级页表会导致内存访问的复杂性增加。由于一级页表需要直接映射整个地址空间，每次内存访问都需要查找一级页表以获得物理地址。这会增加内存访问的延迟，并增加了硬件设计的复杂性。

   如果采用二级页表可以解决上述问题：
   - 内存开销：二级页表使用了两级的结构，将整个地址空间分割为更小的单元，每个单元只需要一个较小的页表来映射。这样可以大大减小页表的大小，节省了内存开销。
   - 内存访问：二级页表通过两级索引的方式进行内存访问，首先使用Page Directory（一级页表）查找对应的Page Table（二级页表），然后再在Page Table中查找页表项并获取物理地址。这样可以减少每次内存访问时需要查找的页表项数量，降低了访问延迟。

> 问题：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?

答：当一个指针变量的值等于 NULL 时，即指针变量被显式地设置为 NULL 或 nullptr，即指向的地址是0x0，其物理存储内容没有访问权限

当程序尝试对空指针解引用时，计算机内部会发生以下情况：

解引用的时候：获得变量的值->访问0地址->mmu进行地址转换->在页表中找不到/没有对应权限->引发异常，进入异常处理程序->进程被杀死。

这里需要辨别一下空指针和野指针的区别：

野指针就是不知道指向哪里，或者说不知道指向的内存是否可以使用，一般都是刚刚声明但没有初始化的指针。空指针不是指向常数0，只指向地址0，即NULL，其实换句话说，指针的本质就是地址，空指针就是指针本身的值（地址）为0。空指针的作用是防止野指针的出现。

2.1.4 准备内核页表

操作系统为了启动分页机制，首先需要准备一些内核页表，框架代码已在`nexus-am/am/arch/x86nemu/src/pte.c`的`_pte_init()`函数中实现该功能了，现在需要在`nanos-lite/src/main.c`中定义宏`HAS_PTE`。

```
/* Uncomment these macros to enable corresponding functionality. */
#define HAS_ASYE
#define HAS_PTE
```

此时，尝试在nanos-lite中make run，遇到了以下错误：

```
[src/monitor/monitor.c,65,load_img] The image is /mnt/d/LessonProjects/SystemDesign/ComputerSystemDesign/nanos-lite/build/nanos-lite-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 14:26:00, May 22 2023
For help, type "help"
(nemu) c
[src/mm.c,24,init_mm] free physical pages starting from 0x46c3000
invalid opcode(eip = 0x0010186b): 0f 22 d8 0f 20 c0 89 45 ...

There are two cases which will trigger this unexpected exception:
1. The instruction at eip = 0x0010186b is not implemented.
2. Something is implemented incorrectly.
Find this eip(0x0010186b) in the disassembling result to distinguish which case it is.

If it is the first case, see
```

```
for more details.

If it is the second case, remember:
* The machine is always right!
* Every line of untested code is always wrong!

(nemu)
```

这说明了有指令未实现。查看`nanos-lite-x86-nemu.txt`，得到以下信息：

```
101866:   b8 00 20 6c 04          mov    $0x46c2000,%eax
10186b:   0f 22 d8                mov    %eax,%cr3
10186e:   0f 20 c0                mov    %cr0,%eax
101871:   89 45 f4                mov    %eax,-0xc(%ebp)
```

这里可以看出，是一个有关cr3的mov指令未实现。

我们需要在`nemu/include/cpu/reg.h`当中，添加 CR0 和 CR3 两个控制寄存器：

```
typedef struct {
  ...
  struct IDTR
  {
    /* data */
    uint32_t base;
    uint16_t limit;
  } idtr;

  rtlreg_t cs;
```

```
  rtlreg_t es; // 配x64
  rtlreg_t ds;
  uint32_t CR0;
  uint32_t CR3;
  bool INTR;
} CPU_state;
```

目前初始化 MM 的工作有两项，第一项工作是将 TRM 提供的堆区起始地址作为空闲物理页的首地址，将来会通过 `new_page()` 函数来分配空闲的物理页。第二项工作是调用 AM 的 `\_pte_init()` 函数，填写内核的页目录和页表，然后设置 CR3 寄存器，最后通过设置 CR0 寄存器来开启分页机制。这样以后，Nanos-lite 就运行在分页机制之上了。调用 `_pte_init()` 函数的时候还需要提供物理页的分配和回收两个回调函数，用于在 AM 中获取/释放物理页。 为了简化实现，MM 中采用顺序的方式对物理页进行分配，而且分配后无需回收。

我们需要修改 `nemu/include/cpu/rtl.h`，实现新的指令，实现对两个寄存器的控制。首先是 `rtl_load_cr` 函数，这个函数用于对控制信息的读取，根据不同的情况把 CR0 或 CR3 寄存器当中存储的信息读取到 dest 参数中。

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

之后是 `rtl_store_cr` 函数，这个函数实现对控制信息的存储，根据情况把参数保存到CR0寄存器或CR3寄存器中：

```
static inline 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;
```

```
}
```

我们需要进一步完善CR0和CR3寄存器。对于 CR0寄存器，我们只需要实现 PG 位即可。如果发现 CR0 的 PG 位为 1，则开启分页机制，从此所有虚拟地址的访问(包括`vaddr_read()`，`vaddr_write()`)都需要经过分页地址转换。为了让 differential testing 机制正确工作，在`nemu/src/monitor/monitor.c`的`resart`函数中我们需要对 CR0 寄存器初始化为 0x60000011：

```
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
}
```

由于系统只能通过MOV指令的变体访问CR0和CR3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。接下来我们要实现这个mov指令。

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

查阅i386手册：

MOV —— Move to/from Special Registers

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

Operation

DEST ← SRC;

Description

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

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

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

在 `nemu/src/cpu/decode/decode.c` 中实现译码函数，其中 `make_DHelper(mov_load_cr)` 完成的是把控制寄存器中所存储的值进行加载读取的操作，`make_DHelper(mov_load_cr)` 完成的是把目标寄存器的值保存到控制寄存器的操作。具体代码如下所示：

```
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_src -> str, 5, "%%cr%d", id_dest -> reg);
#endif
}
```

在`decode.h`中加上声明：

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

注意需要修改`exec.c`的2 byte op_table：

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

还需要在`nemu/src/cpu/exec/data-mov.c`进行修改，以完成控制寄存器的store操作

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

至此，分页机制已基本完成。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。首先在nanos-lite目录下`make run`，然后通过设置断点，我们可以查看：

首先在`0x10156c`设置断点（之前中断的位置）

```
(nemu) w $eip==0x10156c
[src/monitor/debug/expr.c,91,make_token] match rules[3] = "\$(eax|ecx|edx|ebx|esp|ebp|esi|edi|eip|ax|cx|dx|bx|sp|bp|si|di|al|cl|dl|bl|ah|ch|dh|bh)" at positi
on 0 with len 4: $eip
Success record : nr_token = 0, dtype = 259, str = eip
[src/monitor/debug/expr.c,91,make_token] match rules[4] = "==" at position 4 with len 2: ==
Success record : nr_token = 1, dtype = 260, str =
[src/monitor/debug/expr.c,91,make_token] match rules[1] = "0x[0-9A-Fa-f][0-9A-Fa-f]*" at position 6 with len 8: 0x10156c
Success record : nr_token = 2, dtype = 258, str = 10156c
Success: set watchpoint 2, oldvalue = 0
```

```
(nemu) info r
eax  0x423232be
ecx  0x4b05bf62
edx  0x197b6110
ebx  0x598850a9
esp  0x3d6ed119
ebp  0x771a0f22
esi  0x3edb3be6
edi  0x31233cb3
eip  0x100000
ax  0x32be
cx  0xbf62
dx  0x6110
bx  0x50a9
sp  0xd119
bp  0xf22
si  0x3be6
di  0x3cb3
al  0xbe
cl  0x62
dl  0x10
bl  0xa9
ah  0x32
ch  0xbf
dh  0x61
bh  0x50
eflags:CF=0,ZF=0,SF=0,IF=0,OF=0
CR0=0x60000011, CR3=0x0
(nemu)
```

然后单步运行直0x10157d，可以看到：

```
CR0=0xe0000011,CR3=0x1d6a000
```

此时CR0和CR3已设置完毕。

2.1.5 虚拟地址的转换

首先我们查看`nemu/include/memory/mmu.h`，了解其中的重要数据结构：

- CR0：即控制寄存器CR0，为Union结构
- CR3：同CR0，即控制寄存器CR3，为Union结构
- PDE：页目录表表项（Page Directory Entry），为Union 结构
- PTE：二级页表表项（Page Table Entry），本质为 Union 结构

查看`nexus-am/am/arch/x86-nemu/include/x86.h`，了解其重要的数据结构：

- PDE与PTE：同上
- 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
- PTE_P：即 0x0001，标志 Present 位

这里再解释一下标志位：

- present 位表示物理页是否可用,不可用的时候又分两种情况:

    - 物理页面由于交换技术被交换到磁盘中，即 Page fault 的情况之一，这时候可以通知操作系统内核将目标页面换回来，这样就能继续执行了
    - 进程试图访问一个未映射的线性地址，并没有实际的物理页与之相对应，因此这就是一个非法操作
- R/W 位表示物理页是否可写，如果对一个只读页面进行写操作，就会被判定为非法操作
- U/S 位表示访问物理页所需要的权限，如果一个 ring 3 的进程尝试访问一个 ring 0 的页面，当然也会被判定为非法操作

接下来根据实验指导书的要求对 `nemu/src/memory/memory.c` 中的 `vaddr_read()` 和 `vaddr_write()` 函数作少量修改，并实现 `page_translate()` 函数，使得所有虚拟地址的访问都需要经过分页地址的转换。

仿照`x86.h`，在`nemu/src/memory/memory.c`中添加必要的宏定义，这将在`page_translate`中用到：

```
#define PTXSHFT 12 //线性地址偏移量
#define PDXSHFT 22 //线性地址偏移量

#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)
```

修改`vaddr_read()`和`vaddr_write()`两个函数，实现读写地址时的虚拟地址转换。

```
uint32_t vaddr_read(vaddr_t addr, int len) {
  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);
    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);
    }
  }
  else {
    paddr_t paddr = page_translate(addr, true);
    paddr_write(paddr, len, data);
  }
  // paddr_write(addr, len, data);
}
```

然后我们实现`page_translate()`函数。这个函数是用来实现页面地址转换，`addr`为待处理的页面地址，首先判断页目录项和页表是否存在，然后使用`iswrite`来判断读或写的操作：

- 读操作：`iswrite=false`
- 写操作：`iswrite=true`

并依此修改对应的`accessed`位与`dirty`位。如果页面不存在，则报告错误并结束程序。

```
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);
    Assert(pte.present, "addr=0x%x", addr);
    PTE *ptable = (PTE*)PTE_ADDR(pde.val);
    PTE pte = (PTE)paddr_read((uint32_t)(ptable + 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(pte.val) | OFF(addr);
    return paddr;
  }
    return addr;
}
```

运行dummy，可以看到HIT GOOD TRAP：

```
[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
```

但是运行仙剑奇侠传时，会出现数据跨页的情况，这时就会直接结束程序。

```
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
```

2.1.5 数据跨过虚拟页边界

对于数据跨页的情况，这就需要进行两次页级地址转换，分别读出这两个物理页中需要的字节，然后拼接起来组成一个完成的数据返回。需要重新修改`vaddr_read`和`vaddr_write`：

- `vaddr_read`：用`num1`和`num2`两个变量记录高低页的字节数，用`paddr1`和`paddr2`来记录对应的地址，最后将两次读取到的字节进行整合
- `vaddr_write`：将需要写入的字节拆分，写入两个页面

```
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);
  }
}

void vaddr_write(vaddr_t addr, int len, uint32_t data) {
  if(PTE_ADDR(addr) != PTE_ADDR(addr+len-1)) {
    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);
}
```

之后即可成功运行仙剑奇侠传：

2.2 让程序运行在分页机制之上

首先需要将 `navy-apps/Makefile.compile` 中的链接地址 `-Ttext` 参数改为 `0x8048000`，这是为了避免用户程序的虚拟地址空间与内核相互重叠，从而产生非预期的错误。同样的，`nanos-lite/src/loader.c` 中的 `DEFAULT_ENTRY` 也需要作相应的修改。这样，链接器和程序都不需要关心程序运行时刻具体使用哪一段物理地址,它们只要使用虚拟地址就可以了。虚拟地址和物理地址之间的映射则全部交给操作系统的 MM 来管理。

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

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

修改`main.c`，使之运行dummy。

```
load_prog("/bin/dummy");
```

此时`loader()`不能直接把用户程序加载到内存位置 0x8048000 附近了，因为这个地址并不在内核的虚拟地址空间中，内核不能直接访问它。loader()要做的事情是，获取用户程序的大小之后，以页为单位进行加载:

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

因此我们需要修改`loader`，使其按页加载：

```
uintptr_t loader(_Protect *as, const char *filename) {
  int fd = fs_open(filename, 0, 0);
  Log("filename=%s,fd=%d",filename,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;
}
```

然后开始实现页的分配和映射。修改`nexus-am/am/arch/x86-nemu/src/pte.c`，实现`_map()`函数，其具体功能位将虚拟地址空间 p 中的虚拟地址 va 映射到物理地址 pa。

首先通过`p->ptr`可以获取页目录的基地址，然后通过`PDE *pde = dir + PDX(va)`找到页目录项。如果页目录项不存在，即需要申请新页，则调用`palloc_f()`向 Nanos-lite 获取一页空闲的物理页。

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

  PDE *dir = (PDE*) p -> ptr; // page directory
    PTE *table = NULL;
    PDE *pde = dir + PDX(va); // page directory entry
    if(!(*pde & PTE_P)) { // page directory entry not exist
        table = (PTE*) (palloc_f());
        *pde = (uintptr_t) table | PTE_P;
    }
    table = (PTE*) PTE_ADDR(*pde); // page table
    PTE *pte = table + PTX(va); // page table entry
    *pte = (uintptr_t) pa | PTE_P;
}
```

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

此时运行dummy，能显示HIT GOOD TRAP：

```
[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
```

> 问题：内核映射的作用
>
> 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

答：每个进程都包含了内核部分的地址映射（高地址的部分留给操作系统内核），在用户程序请求系统调用服务的时候，需要陷入内核态执行内核的代码。假如没有这一段内核映射的复制，那么就需要在trap的时候进行地址空间的切换，这样的开销是很大的。

2.3 在分页机制上运行仙剑奇侠传

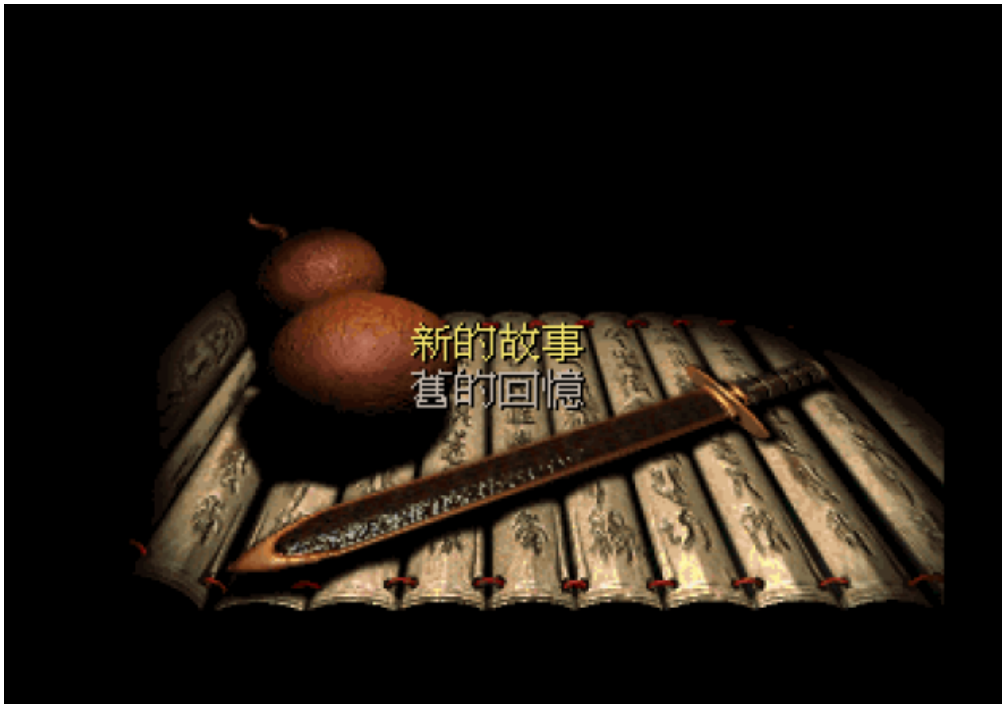
之前在PA3中，我们让 `mm_brk()` 函数直接返回 0，表示用户程序的堆区大小修改总是成功，这是因为在实现分页机制之前，0x4000000 之上的内存都可以让用户程序自由使用。现在用户程序运行在虚拟地址空间之上，我们还需要在 `mm_brk()` 中把新申请的堆区映射到虚拟地址空间中。为了实现在分页机制上运行仙剑奇侠传，我们需要作以下修改：

```
/* 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;
}
```

同时也需要修改 `nanos-lite/src/syscall.c` 中的 `sys_brk()` 函数：

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

实现后，测试仙剑奇侠传：



可以正常运行。

3 阶段二 上下文切换

之前的阶段一已经实现让用户程序运行在相互独立的虚拟地址空间上了，我们只需要再加入上下文切换的机制，就可以实现一个真正的分时多任务操作系统了。

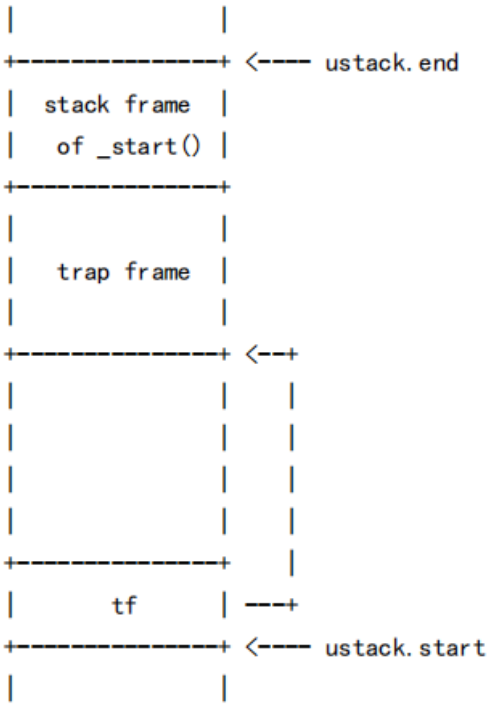
具体来说，上下文切换的过程为：假设程序 A 运行的过程中触发了系统调用，陷入到内核。根据`asm_trap()`的代码，A 的陷阱帧将会被保存到 A 的堆栈上。本来系统调用处理完毕之后，`asm_trap()`会根据 A 的陷阱帧恢复 A 的现场。如果先将栈顶指针切换到另一个程序 B 的堆栈上，接下来的恢复现场操作将会恢复成 B 的现场：恢复 B 的通用寄存器，弹出`#irq`和错误码，恢复 B 的 EIP,CS,EFLAGS。从`asm_trap()`返回之后，则在运行程序B。程序A暂时被挂起，在被挂起之前，它已经把现场的信息保存在自己的堆栈上了，如果将来的某一时刻栈顶指针被切换到 A 的堆栈上，代码将会根据 A 的"陷阱帧"恢复 A 的现场，A将得以唤醒并执行。上下文的切换实际是不同程序之间的堆栈切换。

为了找到别的程序的陷阱帧，我们需要一个指针tf来记录陷阱帧的位置。

我们之前在操作系统课程中学过，为了方便对进程进行管理，操作系统使用一种叫进程控制块(PCB,process control block)的数据结构，为每一个进程维护一个 PCB。代码为每一个进程分配了一个 32KB 的堆栈，在进行上下文切换的时候，只需要把 PCB中的 tf 指针返回给 ASYE 的`irq_handle()`函数即可，剩余部分的代码会根据上下文信息恢复现场。

我们需要在用户进程的堆栈上初始化一个陷阱帧，并把陷阱帧中的cs设置为8，这一部分在`nexus-am/am/arch/x86-nemu/src/pte.c`中的`_umake()`函数中实现。除此之外，还需要在陷阱帧之前设置好`_start()`函数的栈帧，这是为了`_start()`开始执行的时候,可以访问到正确的栈帧。我们只需要把这一栈帧中的参数设置为 0 或 NULL 即可

`_umake`函数在栈上初始化的内容如下，然后返回陷阱帧的指针，由 Nanos-lite 把这一指针记录到用户进程 PCB 的 tf 中:



3.1 实现内核自陷

内核自陷的功能与 ISA 相关是由 ASYE 的`_trap()`函数提供的。在 x86-nemu 的 AM 中，我们约定内核自陷通过指令`int $0x81`触发。

首先我们修改`nexus-am/am/arch/x86-nemu/src/asye.c`中的`_trap()`函数:

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

之后，需要注释掉`nanos-lite/src/proc.c`中的三行代码:

```
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];
  // Log("run proc go to %x", entry);
```

```
  // ((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);
}
```

ASYE 的`irq_handle()`函数发现触发了内核自陷之后,会包装成一个`_EVENT_TRAP`事件。Nanos-lite 收到这个事件之后，就可以返回第一个用户程序的现场了。以此修改`irq_handle()`函数：

```
_RegSet* irq_handle(_RegSet *tf) {
  _RegSet *next = tf;
  if (H) {
    _Event ev;
    switch (tf->irq) {
      case 0x80:
        ev.event = _EVENT_SYSCALL;
        break;
      case 0x81:
        ev.event = _EVENT_TRAP;
        break;
      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;
}
```

为了测试`_umake()`的正确性，我们也先通过自陷的方式触发第一次上下文切换。由于`irq_handle()`函数发现触发了内核自陷之后,会包装成一个`_EVENT_TRAP`事件，因此我们需要在`do_event`的`_EVETN_TRAP`事件中增加一条输出信息，以验证正确性。

```
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;
}
```

在`nexus-am/am/arch/x86-nemu/src/asye.c`中声明中断描述符的入口函数：

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

而且还要再`asye_init()`函数中增加0x81描述符，入口函数设置为`vecself`，其他内容仿照0x80即可：

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

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

  set_idt(idt, sizeof(idt));

  H = h;
}
```

在`nexus-am/am/arch/x86-nemu/src/trap.S`中，补充`vecself()`函数的具体定义：

```
#----|-------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
```

运行程序，显示event: self-trapped，且提示system panic，说明内核自陷已成功完成：

```
[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
```

3.2 实现上下文切换

首先我们先实现 `_umake()` 函数，将 `_start()` 的三个参数和eip入栈，然后完成对tf内容的初始化设置。

```
_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.eip = (uintptr_t) entry;
  void *ptf = (void*) (ustack.end - 16 - sizeof(_RegSet));
  memcpy(ptf, (void*)&tf, sizeof(_RegSet));
  return (_RegSet*) ptf;
}
```

进程调度决定具体切换到哪个进程的上下文。进程调度是由 `schedule()` 函数(在 `nanos-lite/src/proc.c` 中定义)来完成的，它用于返回将要调度的进程的上下文。通过 `current` 指针(在 `nanos-lite/src/proc.c` 中定义)，我们记录了正在运行的进程，该指针指向当前运行进程的PCB，因此需要把当前进程的上下文信息保存在PCB中。

目前 `schedule()` 只需要总是切换到第一个用户进程即可，即 `pcb[0]`。它的上下文是在加载程序的时候通过 `_umake()` 创建的，在 `schedule()` 中才决定要切换到它，然后在 ASYE 的 `asm_trap()` 中才真正地恢复这一上下文。在 `schedule()` 返回之前，还需要切换到新进程的虚拟地址空间。

```
_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 的`schedule()`函数，Nanos-lite 收到`_EVENT_TRAP`事件后，调用`schedule()`并返回其现场。

```
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;
}
```

最后修改 ASYE 中`asm_trap()`的实现，使得从`irq_handle()`返回后，先将栈顶指针切换到新进程的陷阱帧， 然后才根据陷阱帧的内容恢复现场，从而完成上下文切换的本质操作。这一部分代码在`nexus-am/am/arch/x86-nemu/src/trap.S`中：

```
asm_trap:
  pushal

  pushl %esp
  call irq_handle

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

  popal
  addl $8, %esp

  iret
```

现在，我们可以通过自陷的方式触发仙剑奇侠传程序了：

3.3 分时多任务

首先在`main.c`中加载第二个用户程序，以实现让`schedule()`轮流返回仙剑奇侠传和hello的现场：

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

然后根据指导书上所写的修改调度代码，修改`current`值，让`schedule()`轮流返回仙剑奇侠传和hello的现场：

```
_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;
}
```

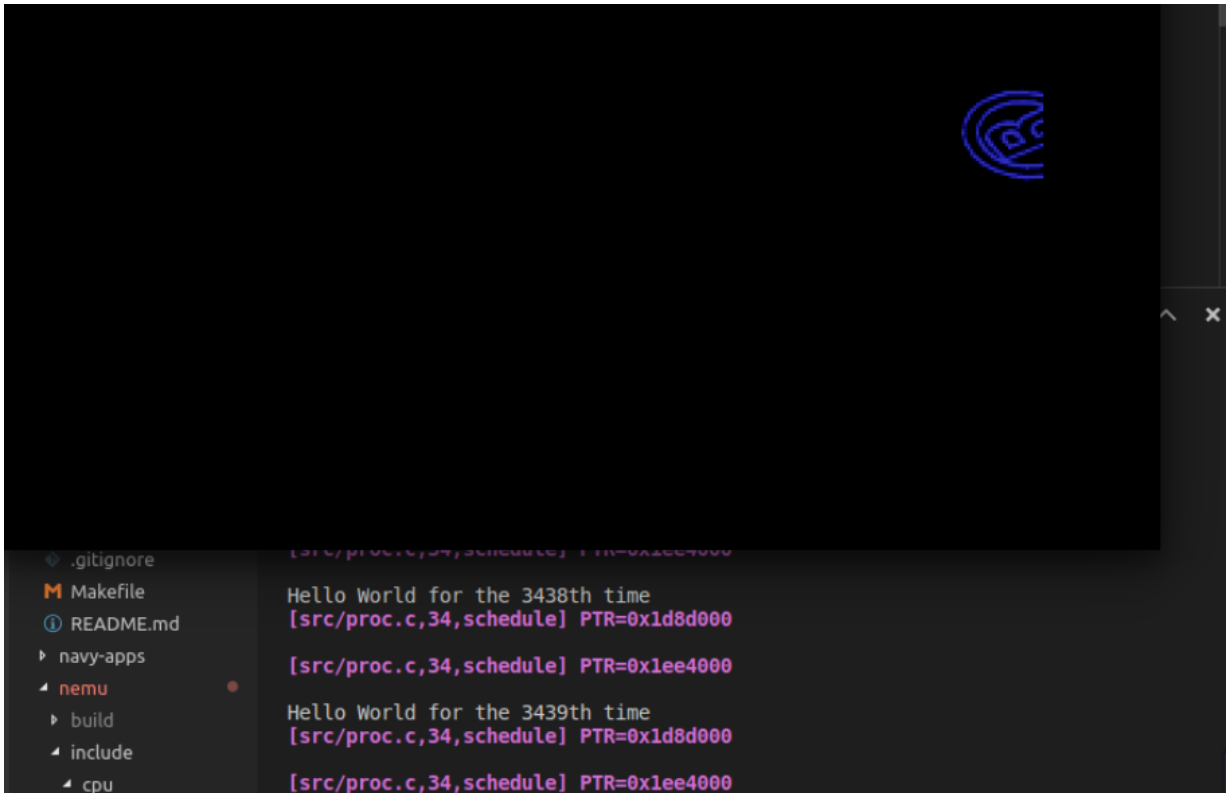
最后需要修改`do_event()`的代码，在处理完系统调用之后，调用`schedule()`函数并返回其现场：

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

此时make run，可以发现仙剑奇侠传一边运行的同时，hello程序也会输出，但仙剑奇侠传的运行速度有明显的下降。即，分时的本质为：程序之间只是轮流使用处理器，它们并不是真正意义上的"同时"运行。



3.4 优先级调度

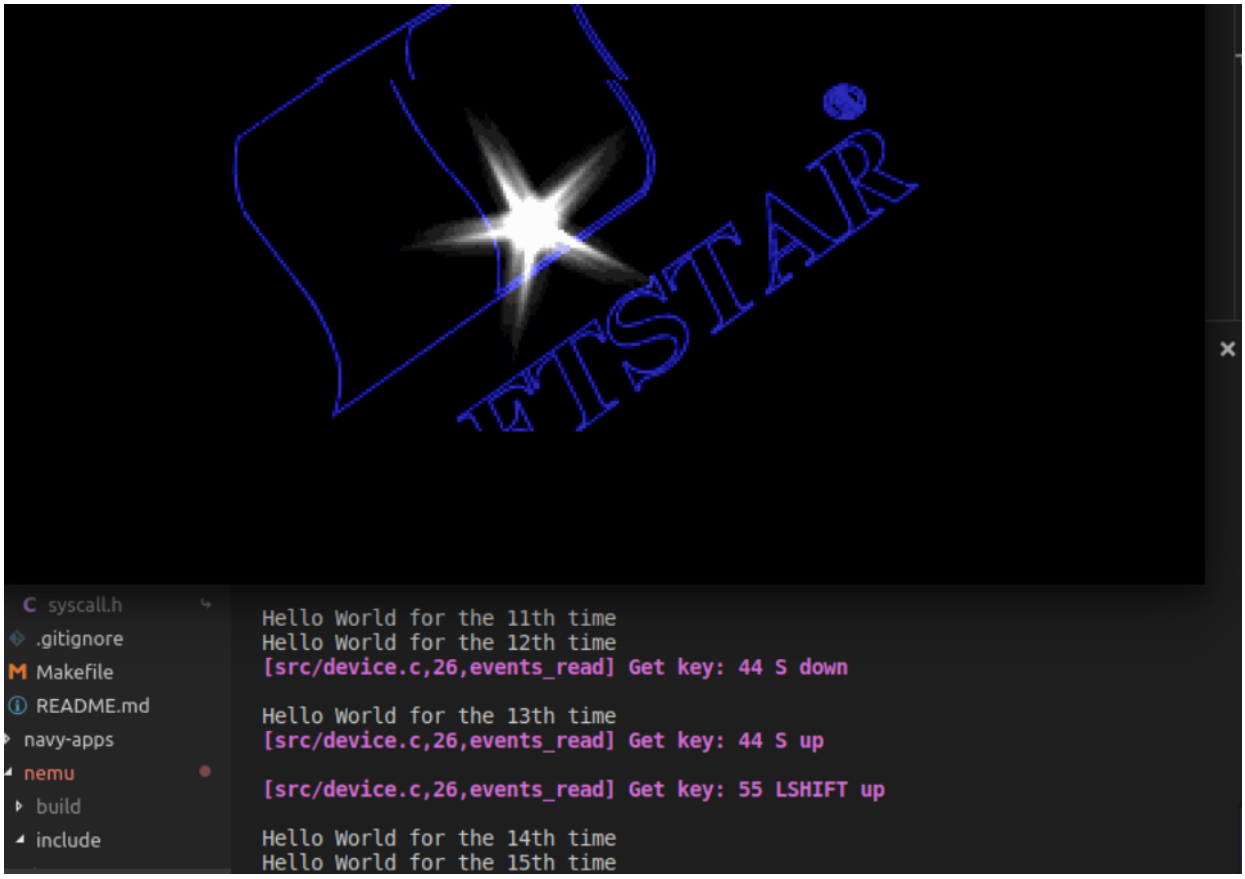
为了让仙剑奇侠传尽量保持原来的性能，我们可以在调度的时候进行一些修改。我们可以修改 `schedule()` 的代码，来调整仙剑奇侠传和 hello 程序调度的频率比例，使得仙剑奇侠传调度若干次，才让 hello 程序调度 1 次。这是因为 hello 程序做的事情只是不断地输出字符串，我们只需要让 hello 程序偶尔进行输出，以确认它还在运行就可以了。

再次修改`schedule()`函数，设置频率比例`frequency`，`num`为当前程序运行的次数。

```
_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;
}
```

重新make run，发现仙剑奇侠传的速度有所提升：



4 时钟中断

在这一部分，我们将主要实现中断。

CPU 每次执行完一条指令的时候，都会看看 INTR 引脚，看是否有设备的中断请求到来。 CPU 每次执行完一条指令的时候，都会看看 INTR 引脚，看是否有设备的中断请求到来。 对 CPU 来说，设备的中断请求何时到来是不可预测的，在处理一个中断请求的时候到来了另一个中断请求也 是有可能的。如果希望支持中断嵌套--即在进行优先级低的中断处理的过程中，响应另一个优先级高的中断 —— 那么堆栈将是保存中断现场信息的唯一选择。

> 灾难性的后果

> 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?

答：如果现场信息被保存在0x1000这个地址处，trap frame的信息就会被覆盖，进入中断嵌套，等到结束中断嵌套时由于trap frame的信息会被覆盖掉所以会一直卡在嵌套中断的位置处理中断而不继续运行，造成死循环，影响后续程序运行，严重情况下也可能会使电脑死机。

4.1 添加时钟中断

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

首先在 cpu 结构体中添加一个 bool 成员 `INTR`：

```
  rtlreg_t cs;
  rtlreg_t es; // 配x64
  rtlreg_t ds;
  uint32_t CR0;
  uint32_t CR3;
  bool INTR;
```

然后在 `dev_raise_intr()` 中将 `INTR` 引脚设置为高电平：

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

在 `exec_wrapper()` 的末尾添加轮询 `INTR` 引脚的代码，每次执行完一条指令就查看是否有硬件中断到来：

```
#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 `nemu/src/cpu/intr.c` 的 `raise_intr()` 的代码。在这里，保存 `eflags` 后关中断（将IF位设置为0），这是必须的，保证中断处理时不会被时钟中断打断进而导致中断嵌套。

```
  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);
```

之后还需要在 `nexus-am/am/arch/x86-nemu/src/asye.c` 中添加时钟中断相关的代码，以支持软件中断的实现。首先声明 `vectime`，然后在 `asye_init()` 中添加int32的门描述符：

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

修改 `irq_handle()` 函数，加入 `_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;
}
```

和 `vecsys` 一样，需要在 `nexus-am/am/arch/x86-nemu/src/trap.S` 中声明 `vectime`

```
#----|-------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
```

在 `nanos-lite/src/irq.c` 的 `do_event` 函数中完成中断事件分发：

```
static _RegSet* do_event(_Event e, _RegSet* r) {
```

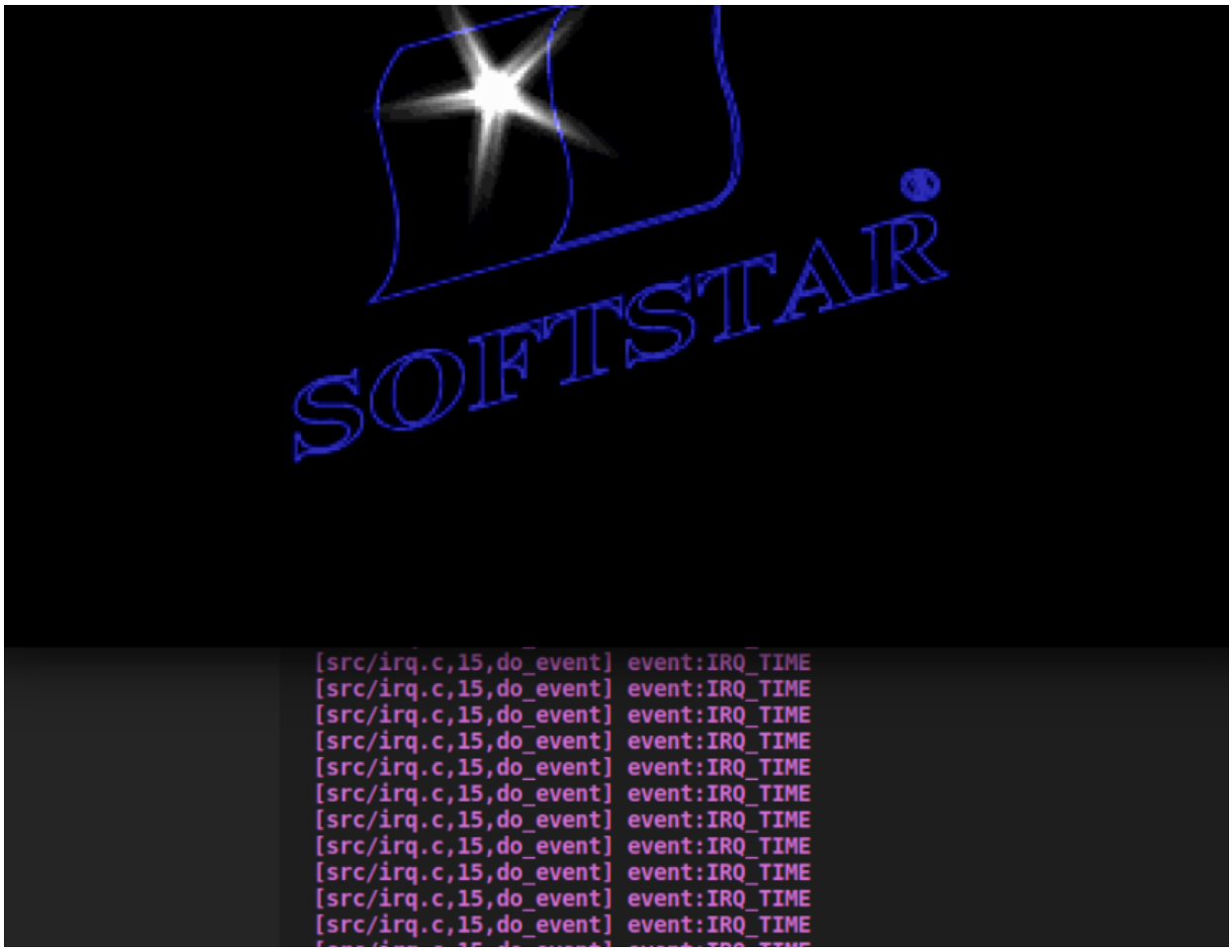
```
  switch (e.event) {
    case _EVENT_SYSCALL:
      // return do_syscall(r);
      do_syscall(r);
      return schedule(r);
    case _EVENT_TRAP:
      printf("event:self-trapped\n");
      return schedule(r);
    case _EVENT_IRQ_TIME:
      Log("event:IRQ_TIME");
      return schedule(r);
    default: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

修改`nexus-am/am/arch/x86-nemu/src/pte.c`中的`_umake`函数，对eflages寄存器进行设置，以保证程序能对时钟中断做出正确的响应。其中`FL_IF`在`nexus-am/am/arch/x86-nemu/include/x86.c`中定义。

```
_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.eip = (uintptr_t) entry;
  void *ptf = (void*) (ustack.end - 16 - sizeof(_RegSet));
  memcpy(ptf, (void*)&tf, sizeof(_RegSet));
  return (_RegSet*) ptf;
}
```

到现在为止，时钟中断已完成。这是分时运行仙剑奇侠传和hello程序，进程切换不会在系统调用时发生，而是在时钟中断时尝试进程切换。



4.2 展示计算机系统

让 Nanos-lite 加载第 3 个用户程序 `/bin/videotest`，并在 Nanos-lite的 `events_read()` 函数中添加以下功能：当发现按下 F12 的时候，让游戏在仙剑奇侠传和 videotest 之间切换。

为了实现这一功能，我们需要修改 `schedule()` 的代码，通过一个变量 `current_game` 来维护当前的游戏，在 `current_game` 和 hello 程序之间进行调度。这里需要修改 `nanos-lite/src/proc.c` 文件：

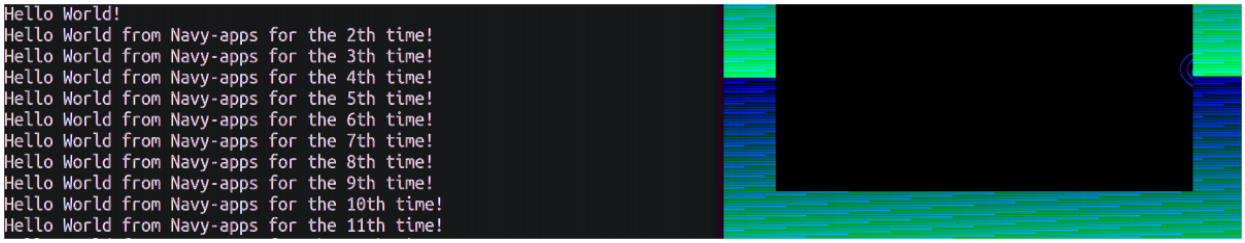
```
int current_game = 0;
void switch_current_game() {
  current_game = 2 - current_game;
  Log("current_game = %d", current_game);
}
```

在这里我们定义了一个 `switch_current_game` 作为辅助函数。然后需要修改 `device.c` 中的 `events_read()` 函数，在该函数中检测F12按键，并切换程序作为相应。

```
size_t events_read(void *buf, size_t len) {
  char buffer[40];
  int key = _read_key();
  int down = 0;
  if(key & 0x8000) {
      key ^= 0x8000;
      down = 1;
  }
```

```
  if(down && key == _KEY_F12) {
    extern void switch_current_game();
    switch_current_game();
    Log("key down:_KEY_F12, switch current game0!");
  }
  if(key != _KEY_NONE) {
     sprintf(buffer, "%s %s\n", down ? "kd": "ku", keyname[key]);
  }
  else {
      sprintf(buffer,"t %d\n", _uptime());
  }
  if(strlen(buffer) <= len) {
    strncpy((char*)buf, buffer,strlen(buffer));
      return strlen(buffer);
  }
  Log("strlen(event)>len, return 0");
  return 0;
}
```

现在，一开始是仙剑奇侠传和 hello 程序分时运行，按下 F12 之后,就变成 videotest 和 hello 程序分时运行。



5 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.

答:

为了支持分页机制，实验中我们主要做了三部分的修改

1. AM部分
   该部分主要包括`_pte_init`（初始化内核分页机制）、`_map`(虚拟地址到物理地址的映射)、`_umake`(在堆栈上初始化陷阱帧)、`_switch`(切换cr3)等，在`nexus-am/am/arch/x86-nemu/src/pte.c`中涉及的代码均是与硬件相关，是硬件层面支持页表机制的状态维护。
2. Nanos-lite部分
   该部分与分页机会相关的内容主要包括`loader`，将文件以页的形式载入内存，并通过`_map`建立虚拟地址也物理地址的映射。
3. Nemu部分
   该部分对分页自己的支持主要是通过`page_translate`，`vaddr_write`，`vaddr_read`在对内存进行读写时进行虚拟地址到物理地址的变换，这是整个分页机制得以成功运行的关键。

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开启分页后，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()`。具体来说，`page_translate` 这个函数通过页表翻译的过程，将虚拟地址转换为物理地址。它首先获得页目录项和页表项，并对 `present` 位进行判断，随后设置access位和dirty位，最后根据页表项中的物理地址和虚拟地址的偏移量，计算出最终的物理地址。代码如下：

```
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);
    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);
    return paddr;
  }
    return addr;
}
```

而`vaddr_read()`与`vaddr_write()`则调用`page_translate`，对地址进行操作。

在启动分页机制之前，操作系统需要准备内核页表。这个过程涉及到 Nanos-lite和 AM之间的协作。

首先，Nanos-lite负责初始化存储管理器。它将从 TRM（Tiny Real Mode）提供的堆区起始地址作为空闲物理页的首地址，并注册用于分配和回收物理页的函数，即`new_page()`和`free_page()`。接着，Nanos-lite调用 AM 提供的`pte_init()`函数来准备内核页表。

`pte_init()`函数是 AM 实现的一个基本框架，用于填写内核的二级页表并设置 CR0 和 CR3 寄存器。在这个过程中，内核页表的基本结构被建立起来，使得分页机制能够正确地进行地址转换和访问控制。

通过这样的协作，Nanos-lite 和 AM 实现了操作系统启动分页机制所需的准备工作，包括初始化存储管理器、设置空闲物理页、注册物理页分配与回收函数，以及构建内核页表的基本框架。这样，操作系统就能够开始使用分页机制来管理虚拟内存并提供更高级的内存管理功能。

硬件中断机制

硬件中断是分时多任务实现的又一关键，而对于我们的实验来说，这关键中的关键便是时钟中断（`_EVENT_IRQ_TIME`），和内核自陷（`_EVENT_TRAP`）。通过这样两种中断保证操作系统中分时多任务机制的正常运行。

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

  vaddr_t gate_addr = cpu.idtr.base + NO * sizeof(GateDesc);
  // Log("%d %d %d\n", gate_addr, cpu.idtr.base, cpu.idtr.limit);
  assert(gate_addr <= cpu.idtr.base + cpu.idtr.limit);

  uint32_t off_15_0 = vaddr_read(gate_addr,2);
  uint32_t off_32_16 = vaddr_read(gate_addr+sizeof(GateDesc)-2,2);
  uint32_t target_addr = (off_32_16 << 16) + off_15_0;
#ifdef DEBUG
  Log("target_addr=0x%x",target_addr);
#endif
  decoding.is_jmp = 1;
  decoding.jmp_eip = target_addr;
}

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

每隔10ms，nemu中的设备timer就会调用一次`timer_intr()`，进而触发`dev_raise_intr`，请求一次硬件中断。在这里，INTR引脚被设置为1。nemu中的CPU在执行完一条指令后便会去查询当前是否需要响应中断。响应中断的条件是：CPU处于开中断状态(IF不为1)，且有硬件中断到来(INTR为1)。

当CPU)响应中断的方式是自陷，通过`raise_intr`保存现场，随后跳转到约定的中断处理代码，中断处理代码由`raise_intr`的参数中断号决定。

在nexus-am中，内核自陷的地址是`0x81`的中断入口，它是在`_asye_init`的中断向量表中定义。AM会根据`vectrap`进行相应处理，完成一次进程调度，进入`asm_trap`函数中。在这里调用`irq_handle`，将栈顶的指针切换回堆栈上，恢复现场。

```
#----|-------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

asm_trap:
  pushal
```

```
  pushl %esp
  call irq_handle

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

  popal
  addl $8, %esp

  iret
```

在nanos-lite的调度函数`schedule`中，对各个用户进程做时间片分配，即完成了分时多任务的进行。schedule 函数会跳转到 `_switch` 函数，在`_switch`中只有一个`set_cr3`记录页表页目录项的基地址，进行地址空间的切换。

最终，各个函数进行返回， 执行`trap.S`中的最后一条指令，完成一次中断调用。这样的话，虽然多任务的虚拟地址是重叠的，但是通过这种中断，可以切换二者的地址空间并保护上下文，顺利实现分时多任务。

总得来说，在整个分时多任务机制中，页表机制保证了操作系统的不同的进程按照相同的虚拟地址来对内存进程访问，这使得操作系统能够运行位置无关代码，同时也实现了按需分配内存和突破物理地址的上限。但是上述折现过程也仅仅是实现了多任务机制对内存资源的有效利用，而保证分时多任务的切换则是通过硬件中断机制得以实现。通过时钟中断来保证不同的进程能够根据一定规则相对公平的利用cpu实行，从而在宏观上实现多任务同时进行。

6 遇到的问题

6.1 HIT BAD TRAP at eip = 0x00100032

在实现分页机制的时候，我发现我的程序一直在HIT BAD TRAP，但是检查了代码觉得又没错。忽然想到自己在上一份报告里也报告了相似的问题，再返回看了一次错误记录，发现又是make update的问题，感觉自己还是太粗心了。

6.2 page_translate: Assertion 'pte.present' failed.

这个错误查了好久，最终发现是`sys_brk`函数的问题。这个函数在PA3中是直接返回0，但是这里应该要改为调用mm_brk`，所以最后return语句应修改为：

```
return mm_brk(addr);
```

就可以解决问题了。

6.3 文件指针

在最后一个任务F12切换任务遇到的问题，同样也是HIT BAD TRAP at eip = 0x00100032，但是这次是`fs_close`函数出的问题。之前的文件指针没有置为0，导致下一次打开别的文件的时候偏移量发生错误，遂报错。

正确的实现应该是：

```
int fs_close(int fd) {
  assert(fd >= 0 && fd < NR_FILES);
  file_table[fd].open_offset = 0;
  return 0;
}
```