PA4 - 虚实交错的魔法: 分时多任务

蒋璋 1711333

PA4.0 实验目的和概述

0.1 实验目的

1. 理解多道程序系统的基本思想
2. 实现上下文切换
3. 理解外部中断及外部中断处理
4. 理解虚拟内存,实现虚实地址转换

0.2 实验内容

- task PA4.1: 实现基本的多道程序系统
- task PA4.2: 实现支持虚存管理的多道程序系统
- task PA4.3: 实现抢占式分时多任务系统, 并提交完整的实验报告

PA4.1: 实现基本的多道程序系统

1.1 多道程序的概念

PA3中实现的批处理系统其实已经可以执行我们想要其执行的程序的了，比如仙剑奇侠传或者是exit了以后自动去执行菜单程序，虽然菜单程序给我们提供了很多中选择，其实实际执行的还只是一个程序。我们必须执行完一个程序以后，才能去执行另一个程序，而不是可以同时执行两个程序。当一个程序处于读取磁盘或者等待输入输出的时候，整个系统停滞了下来，浪费了cpu的时间。为了不浪费cpu的时间，可以在等待的时候做一些其他的工作，一个简单的想法就是当一个程序处于等待状态的时候，我们将其切换到另一个程序执行，这就是多道程序的思想。

那么我们列出多道程序需要的条件为以下两点

- 在内存中可以同时存在多个进程
- 在满足某些条件的情况下, 可以让执行流在这些进程之间切换

1.2 实现上下文切换

我们在pa3中实现了系统调用，在其中有简单的上下文切换。上下文的本质就是进程的状态，具体一些就是当前cpu中的所有寄存器。在PA4中，由于我们内存中有多个进程，上下文还应该包括地址空间，使得进程不会触碰到其它人的地址。

具体地, 假设进程A运行的过程中触发了系统调用, 陷入到内核. 根据`__am_asm_trap()`的代码, A的上下文结构(`_Context`)将会被保存到A的栈上. 在PA3中, 系统调用处理完毕之后, `__am_asm_trap()`会根据栈上保存的上下文结构来恢复A的上下文. 神奇的地方来了, 如果我们先不着急恢复A的上下文, 而是先将栈顶指针切换到另一个进程B的栈上, 那会发生什么呢? 由于B的栈上存放了之前B保存的上下文结构, 接下来的操作就会根据这一结构来恢复B的上下文. 从`__am_asm_trap()`返回之后, 我们已经在运行进程B了! 通过中断的返回，我们就可以很容易的实现进程切换了。

1.3 进程控制块

```
typedef union {
  uint8_t stack[STACK_SIZE] PG_ALIGN;
  struct {
    _Context *cp;
    _AddressSpace as;
    uintptr_t max_brk;
  };
} PCB;
```

进程控制块包括栈空间和上下文地址空间以及堆。这里值得注意的地方是每个进程应当要有自己独立的栈。因为线程是独立执行的, 函数参数, 局部变量都保存在栈中, 函数名只是一段代码的起始地址而已, 它执行时要取参数(保存在栈中), 要取局部变量, 这些都在栈中, 所以为了线程不互相干扰,堆栈是独立的。进行上下文切换的时候，只要把下一个切换的进程的cp返回给CTE的`__am_irq_handle()`就能恢复下一个进程的上下文了。

1.4 创建上下文

那么, 对于刚刚加载完的进程, 我们要怎么切换到它来让它运行起来呢? 答案很简单, 我们只需要在进程的栈上人工创建一个上下文结构, 使得将来切换的时候可以根据这个结构来正确地恢复上下文即可。

具体的实现为`\_ucontext()`。在这个函数中，我们为上下文做初始化，并且对于x86来说, 参数信息是存放在栈上的, 因此`_ucontext()`还需要为`main()`函数额外创建一个栈帧, 这个栈帧将来会被`navy-apps/libs/libc/src/platform/crt0.c`中的`_start()`函数使用, 它会把参数信息传给`main()`函数，所以我们还要为`_start()`函数准备其栈帧。我们把`ustack.end`以下的三个字节赋值为`args`，即将来main函数的`argc, argv, envp`三个参数。然后我们将初始化context，将其`eip`设置为程序开始的位置，`cs`，`eflags`按照x86体系结构的说明即可。这里或上`0x200`是为了给`eflags`的`IF`置位。

```
_Context *_ucontext(_AddressSpace *as, _Area ustack, _Area kstack, void *entry,
void *args)
{
  _Context *c = (_Context *)(ustack.end - 4 * sizeof(uintptr_t) -
sizeof(_Context));
  *(uintptr_t *)(ustack.end - 3 * sizeof(uintptr_t)) = args;
  memset(c, 0, sizeof(_Context));
  // printf("u addr : %x\n", entry);
  c->as = as;
  c->eip = entry;
  c->eflags = 2;
  c->cs = 8;
  c->eflags |= 0x200;
  return c;
}
```

1.5 进程调度

如果当前只有两个进程的话，那么如果当前进程为0就切换到1，当前进程为1就切换到0。

```
_Context *schedule(_Context *prev)
{
  current->cp = prev;
  current = (current == &pcb[0] ? &pcb[1] : &pcb[0]);
  return current->cp;
}
```

1.6 实现上下文切换过程

前面已经实现上下文的创建，现在要处理的是上下文的切换过程。分发yield事件，并调用schedule返回新的上下文。

```
static _Context *do_event(_Event e, _Context *c)
{
  switch (e.event)
  {
  case _EVENT_YIELD:
    return schedule(c);
    break;
   ...........
}
```

修改am中irq_handler的实现使其能够返回schedule切换到的另一个进程。

```
.................
    next = user_handler(ev, c);
    if (next == NULL)
    {
      next = c;
    }
  }
  __am_switch(next);
  return next;
```

修改trap.S中的`__am_asm_trap`的实现使得它能够从irq_handler的返回值中恢复上下文。根据函数的翻译规则，返回值保存在eax寄存器中，相当于直接指向了0所在的位置，因此esp可以少加一个4。

```
__am_asm_trap:
  pushal

  pushl $0

  pushl %esp
  call __am_irq_handle

  movl %eax, %esp

  # addl $4, %esp

  addl $4, %esp
  popal
  addl $4, %esp

  iret
```

到这里，实现了上下文的创建和切换工作。但是，如果我们加载两个程序以后，程序依然不能正常运行。原因在于，程序被加载到了物理内存的同一个位置，造成了干扰。我们需要为每个进程维护自己的地址空间，使得不同程序的同一个地址能够指向物理地址的不同位置，这就是虚拟内存的管理。

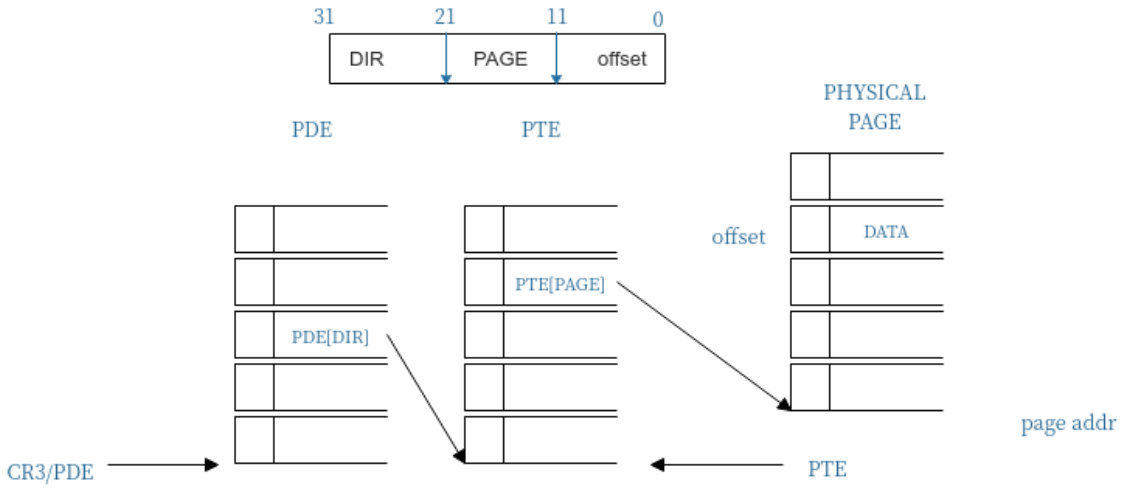
PA4.2: 实现支持虚存管理的多道程序系统

2.1 分页

在分页机制中, 这些小片段称为页面, 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, 来把不必连续的物理页面重新组织成连续的虚拟地址空间. 因此, 为了让分页机制支撑多任务操作系统的运行, 操作系统首先需要以物理页为单位对内存进行管理. 每当加载程序的时候, 就给程序分配相应的物理页(注意这些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这些物理页的映射关系. 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。

2.2 虚拟地址到物理地址的转换

在分页机制下虚拟地址到物理地址的转换可以用下图来演示。CR3寄存器指示了页目录表PDE的位置，通过虚拟地址的高10位作为页目录表的偏移，我们找到了页表的基地址。然后，我们通过页表以及中间10位作为页表偏移，找到了物理页的位置。再通过最后的页偏移，就得到了物理地址。这就是分页机制的核心:虚拟地址到物理地址的转化。



虚拟地址分为三部分, 高10位为页目录项, 用于查页目录表; 中间10位为页表项, 用来查页表; 低12位为页偏移, 用于页表内部的偏移.

转化的过程可以用以下的伪代码表示

```
pde = cr3
pte = pde[dir]
page_addr = pte[page]
paddr = page_addr + offset
```

2.2.1 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?
- 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. 基址只有20位是因为一个页的大小是4k，占据了12位，而每个页是4k对齐的，意味着页面的基地址的低12位都是0，那么我们只需保存它的高20位即可。
2. 必须是物理地址。因为所有虚拟地址都要经过cr3翻译，如果cr3自己页需要翻译的话就会出现递归翻译而没有递归出口的情况。
3. 32位的地址空间一共有2^{20}个物理页，每个物理页需要4B空间存储基地址，那么一共需要4M的空间用于存放页表。非常浪费空间。如果采用二级页表的话最少的情况只需要一个页面用于存放页目录表，一个页面用于存放页表，就是8k的存储空间；最多的时候需要1一个页目录表加上4M的存储空间，仅多出一个页而以。可以看出，在绝绝大多数情况，二级页表比一级页表花费的存储空间要少的多的多。

2.2.2 问题2 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?

空指针通常为(void*)0, 这是因为在大多数计算机中0不是一个有效的地址。空指针用于指示一个不再有效的对象。如果0是一个有效的地址，那么其实0并不适合作空指针。换而言之，空指针需要指向的是一个无效的地址，而不一定非要是0，如果0是一个有效的地址，那么它就不“空”。对空指针进行解引用的时候，如果空指针按照前面所说是一个无效的地址，那么在地址翻译的时候由于present位为0，就会触发segment fault，导致程序崩溃。

2.3 将虚存管理抽象成VME

2.3.1 硬件支持

在cpu的结构中加入cr0和cr3寄存器

```
union {
    struct
    {
      uint32_t PE : 1;
      uint32_t paddings : 30;
      uint32_t PG : 1;
    } CR0;
    rtlreg_t cr0;
  };
  union {
    struct
    {
      uint32_t paddings0 : 3;
      uint32_t page_write_through : 1;
      uint32_t page_cached_disable : 1;
      uint32_t paddings1 : 7;
      uint32_t page_directory_base : 20;
    } CR3;
    rtlreg_t cr3;
  };
```

mov_r2cr和mov_cr2r能够为cr0和cr3寄存器的读写提供支持。

```
make_EHelper(mov_r2cr)
{
  switch (id_dest->reg)
  {
  case 0:
    cpu.cr0 = id_src->val;
    break;
  case 3:
    cpu.cr3 = id_src->val;
    break;
  default:
    assert(0);
  }

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

make_EHelper(mov_cr2r)
{
  switch (id_src->reg)
  {
  case 0:
    t0 = cpu.cr0;
    break;
  case 3:
    t0 = cpu.cr3;
    break;
  default:
    assert(0);
  }
  operand_write(id_dest, &t0);

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

#if defined(DIFF_TEST)
  difftest_skip_ref();
#endif
}
```

page_translate实现mmu的功能，来进行地址的自动转换。需要注意的是只有cr0的PG位是1的时候才开始地址的转换，否则不进行地址转化。在地址转换过程中，要检查表项的最低位，即present位是否为1，如果不是应该报错以尽早发现bug。

```
paddr_t page_translate(vaddr_t vaddr, bool iswrite)
{
  paddr_t dir = (vaddr >> 22) & 0x3ff;
  paddr_t page = (vaddr >> 12) & 0x3ff;
  paddr_t offset = vaddr & 0xfff;
  paddr_t paddr = vaddr;
  if (cpu.CR0.PG)
  {
    uint32_t page_directory_base = cpu.CR3.page_directory_base;
    uint32_t page_table = paddr_read((page_directory_base << 12) + (dir << 2),
4);
```

```
    if ((page_table & 1) == 0)
    {
      printf("[translating] %x\n", vaddr);
      assert(0);
    }
    uint32_t page_frame = paddr_read((page_table & 0xfffff000) + (page << 2),
4);
    if ((page_frame & 1) == 0)
    {
      printf("[translating] %x\n", vaddr);
      assert(0);
    }
    paddr = (page_frame & 0xfffff000) + offset;
  }
  return paddr;
}
```

vaddr_read和vaddr_write和老师写的一样就不贴上来献丑了。需要注意的是由于x86地址的开始位置没有对齐，因此可能对一个物理地址的读写可能需要翻译多次。

2.3.2 操作系统的支持

在创建用户进程使用的`context_uload`中，我们需要调用`_protect`来创建进程的地址空间。在`_protect`中会给进程分配一个页用作页目录表，并且会把内核的地址空间复制到进程的地址空间中。

```
void context_uload(PCB *pcb, const char *filename)
{
  _protect(&pcb->as);
  uintptr_t entry = loader(pcb, filename);
  _Area stack;
  stack.start = pcb->stack;
  stack.end = stack.start + sizeof(pcb->stack);
  pcb->cp = _ucontext(&pcb->as, stack, stack, (void *)entry, NULL);
}
```

loader loader在pa3中也是难题，咬牙坚持到了pa4，发现支持一个feature容易，带着维护更新就麻烦了。由于支持了虚拟内存，loader不应该直接往物理内存写入程序了，而是应该分配物理页，把内容写入到物理页之中，再把物理页映射到程序相应的虚拟地址的位置。

```
static uintptr_t loader(PCB *pcb, const char *filename)
{
  int fd = fs_open(filename, 0, 0);
  Log("fd : %d, filename : %s", fd, filename);
  //读取文件头
  Elf_Ehdr elfHeader;
  size_t len = fs_read(fd, &elfHeader, sizeof(Elf_Ehdr));
  assert(len == sizeof(Elf_Ehdr));
  //读取段头
  fs_lseek(fd, elfHeader.e_phoff, SEEK_SET);
  fs_read(fd, pHeaders, elfHeader.e_phentsize * elfHeader.e_phnum);
  uint32_t paddr, vaddr=0x40000000;
  int pages = 0;
  for (int i = 0; i < elfHeader.e_phnum; i++)
  {
    if (pHeaders[i].p_type != PT_LOAD)
      continue;
```

```
    //将该段读取到制定的内存位置
    fs_lseek(fd, pHeaders[i].p_offset, SEEK_SET);
    pages = (pHeaders[i].p_memsz + PAGE_SIZE - 1) / PAGE_SIZE;
    paddr = (uint32_t)new_page(pages);
    fs_read(fd, (void *)paddr, pHeaders[i].p_filesz);
    vaddr = pHeaders[i].p_vaddr;
    for (int j = 0; j < pages; j++)
    {
      //需要一页一页的映射
      // printf("[loader]pa:%x mapped on va:%x\n", paddr, vaddr);
      _map(&pcb->as, (void *)vaddr, (void *)paddr, _PROT_READ | _PROT_WRITE |
_PROT_EXEC);
      vaddr += PAGE_SIZE;
      paddr += PAGE_SIZE;
    }
    // fs_read(fd, (void *)pHeaders[i].p_vaddr, pHeaders[i].p_filesz);
    //如果出现没对齐的情况把相应的内存区域清0
    if (pHeaders[i].p_filesz < pHeaders[i].p_memsz)
      memset((void *)(paddr + pHeaders[i].p_filesz), 0, pHeaders[i].p_memsz -
pHeaders[i].p_filesz);
  }
  pcb->max_brk = (uintptr_t)vaddr;
  return elfHeader.e_entry;
}
```

最后, 为了让这一地址空间生效, 我们还需要将它落实到MMU中. 具体地, 我们希望在CTE恢复进程上下文的时候来切换地址空间. 为此, 我们需要将进程的地址空间描述符指针加入到上下文中. 框架代码已经实现了这一功能(见`nexus-am/am/include/arch/$ISA-nemu.h`), 但你还需要

- 修改`_ucontext()`的实现, 在创建的用户进程上下文中设置地址空间相关的指针`as`
- 在`__am_irq_handle()`的开头调用`__am_get_cur_as()`(在`nexus-am/am/src/$ISA/nemu/vme.c`中定义), 来将当前的地址空间描述符指针保存到上下文中
- 在`__am_irq_handle()`返回前调用`__am_switch()`(`nexus-am/am/src/$ISA/nemu/vme.c`中定义) 来切换地址空间, 将调度目标进程的地址空间落实到MMU中

```
_Context *__am_irq_handle(_Context *c)
{
    __am_get_cur_as(c);
```

```
__am_switch(next);
```

```
}
```

2.3.3 问题3 对于x86和riscv32, 在`_protect()`中创建地址空间的时候, 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。

这段代码的作用是将内核的地址空间拷贝到用户进程的地址空间中。在_switch函数中会出错。_switch函数切换内核虚拟空间为用户程序创建的虚拟地址空间,此时,虚拟地址是根据用户进程的页目录表来转换的。注释掉_Protect 中拷贝内核函数的代码后,进程的页目录表未初始化,造成内核部分拥有的虚拟地址-物理地址的映射缺失。内核页表中的内容为所有进程共享,每个进程都有自己的“进程页表”,“进程页表”中映射的线性地址包括两部分:用户态和内核态。其中,内核态地址对应的相关页表项,对于所有进程来说都是相同的(因为内核空间对所有进程来说都是共享的),而这部分页表内容其实就来源于“内核页表”,即每个进程的“进程页表”中内核态地址相关的页表项都是“内核页表”的一个拷贝。

mm_brk()的实现

mm_brk用于堆区的管理。PCB中记录了进程的max_brk，如果new_brk大于max_brk的话，就需要分配新的物理页，将大于的部分全部映射到当前进程的地址空间中。

```
int mm_brk(uintptr_t new_brk)
{
  // Log("new_brk = %x max_brk = %x\n", new_brk, current->max_brk);
  if (new_brk > current->max_brk)
  {
    uint32_t va = (current->max_brk & 0xfffff000);
    while (va < (new_brk & 0xfffff000))
    {
      void *pa = new_page(1);
      // printf("[brk] map %x to %x\n", va, pa);
      _map(&current->as, (void *)va, pa, 0);
      va += PGSIZE;
    }
    current->max_brk = new_brk;
  }
  return 0;
}
```

制造切换时机

在device中相关的系统调用前加上yield()来模拟设备访问缓慢的情况。

至此，虚存管理系统已经实现，我们可以同时运行hello和pal了。可以看到一边打印了hello world一边能够看到pal的信息。只是pal慢的令人发指。

```
    current = &pcb[1];
    counter = 0;
  }
  else
  {
    current = &pcb[0];
  }
  return current->cp;
}
```

问题 输出 终端 调试控制台

```
Hello World from Navy-apps for the 48th time!
Hello World from Navy-apps for the 49th time!
Hello World from Navy-apps for the 50th time!
Hello World from Navy-apps for the 51th time!
Hello World from Navy-apps for the 52th time!
Hello World from Navy-apps for the 53th time!
Hello World from Navy-apps for the 54th time!
Hello World from Navy-apps for the 55th time!
Hello World from Navy-apps for the 56th time!
Hello World from Navy-apps for the 57th time!
Hello World from Navy-apps for the 58th time!
Hello World from Navy-apps for the 59th time!
Hello World from Navy-apps for the 60th time!
Hello World from Navy-apps for the 61th time!
Hello World from Navy-apps for the 62th time!
Hello World from Navy-apps for the 63th time!
Hello World from Navy-apps for the 64th time!
Hello World from Navy-apps for the 65th time!
Hello World from Navy-apps for the 66th time!
Hello World from Navy-apps for the 67th time!
Hello World from Navy-apps for the 68th time!
Hello World from Navy-apps for the 69th time!
Hello World from Navy-apps for the 70th time!
```

PA4.3: 实现抢占式分时多任务系统

cpu中添加INTR管脚

```
typedef struct
{
.............
  bool INTR;
} CPU_state;
```

产生时钟中断

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

cpu执行过程中检查是否有中断

```
bool isa_query_intr(void)
{
  // printf("query intr\n");
  if (cpu.eflags.IF && cpu.INTR)
  {
    cpu.INTR = false;
    raise_intr(IRQ_TIMER, cpu.pc);
    return true;
  }
  return false;
}

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

在raise_intr中将eflags的IF位置为0使得cpu进入关中断状态禁用嵌套中断。

打包时间中断并进行分发处理。

irq_handle中识别中断号32，将事件类型设为timer。

```
    case 0x20:
      ev.event = _EVENT_IRQ_TIMER;
      break;
```

在do_event中timer事件的处理时调用`_yield()`触发调度。

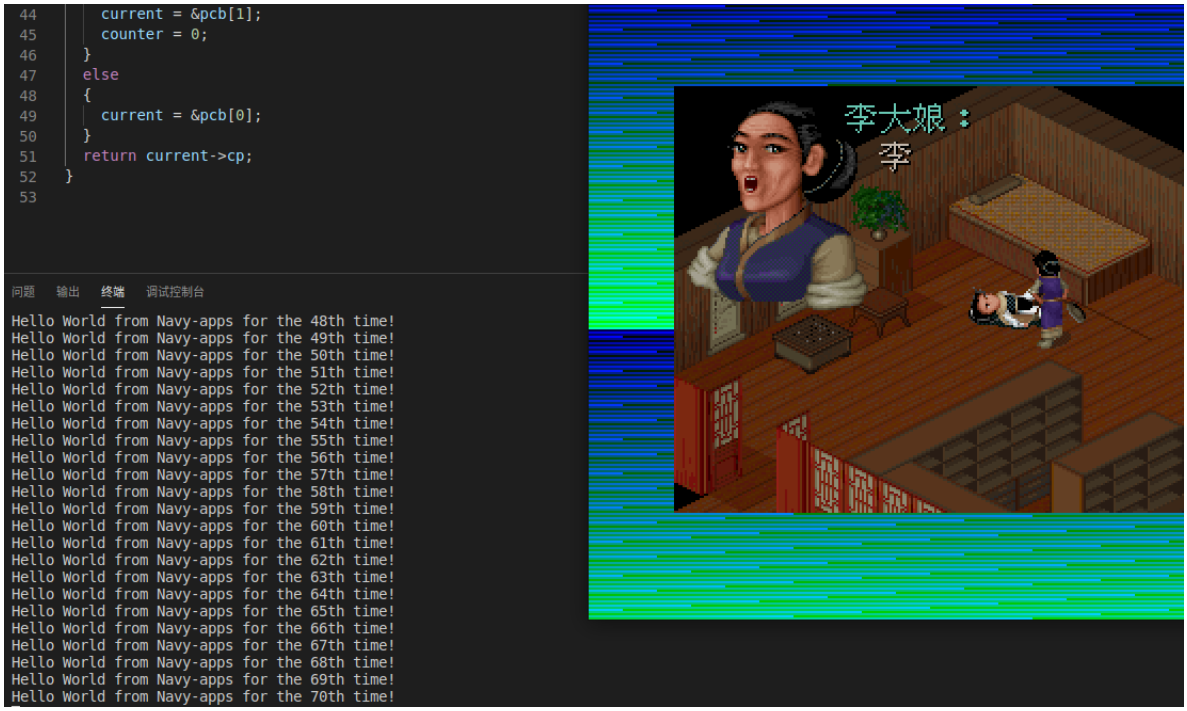
```
  case _EVENT_IRQ_TIMER:
    // Log("time event");
    _yield();
```

基于时间片的进程调度

修改schedule使得pcb[0]每执行100次pcb[1]执行一次，这样我们可以给更多的时间片给pal运行。

```
_Context *schedule(_Context *prev)
{
  current->cp = prev;
  if (counter++ > 100)
  {
    current = &pcb[1];
    counter = 0;
  }
  else
  {
    current = &pcb[0];
  }
  return current->cp;
}
```

可以看到hello程序只是偶尔打印信息，大部分时间片在pal中。



至此，PA4的所有部分完成。

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.

分页机制使得每个进程建立了自己独立的地址空间，使得不同的进程同时被加载到了内存之中。硬件中断定时的发出时钟信号，提供了可以用来进行进程切换的时机。

分页机制的支持

在`vme_init`之前，页面转换还没有建立，此时虚拟地址就是物理地址。在`vme_init`中，建立内核空间的地址映射，并把CR3设置为内核空间的页目录表。然后置位CR0寄存器的PG位，分页机制就开启了。分页机制开启后，nemu通过mmu将虚拟地址翻译成物理地址，进行实际的地址访问。应用程序在需要内存时，调用am提供的new_page获取物理页，然后再通过`mmap`将物理页映射到用户进程的地址空间中。

在加载用户程序时`protect`给用户进程分配页目录表并建立地址空间，并把内核地址空间映射到用户地址空间中。loader将elf的文件读取到用户的地址空间中，建立用户的虚拟地址到物理页的映射。`_ucontext`给进程创建上下文，并把上下文的eip设置为用户程序的开始位置。至此，进程的上下文和地址空间都已经准备好，为真正执行做好了准备。切换到当前进程，当前进程就可以从开始位置执行。

```
void context_uload(PCB *pcb, const char *filename)
{
  _protect(&pcb->as);
  uintptr_t entry = loader(pcb, filename);
  _Area stack;
  stack.start = pcb->stack;
  stack.end = stack.start + sizeof(pcb->stack);
  pcb->cp = _ucontext(&pcb->as, stack, stack, (void *)entry, NULL);
}
```

分时机制的支持

时钟定期发出时钟中断。cpu每执行一条指令便查看一次INTR管脚，如果INTR管脚为高电平，就发出时钟中断，中断号为32。在发出中断的过程中，nemu和am还是会保存好上下文。AM还负责保存地址空间以及恢复下一个进程的上下文。在AM中cte会将事件打包为时间事件，并交给nanos-lite注册的事件处理函数执行。nanos-lite的irq识别事件为时钟信号，使用schedule进行进程调度，schedule返回下一个进程的上下文，最后交给AM进行恢复。AM恢复下一个进程的上下文和地址空间，当cpu执行下一条指令时，就开始执行下一个进程了。