南開大學
Nankai University

南　　开　　大　　学

计　算　机　学　院

计算机系统实验报告

PA4

年级：2022 级

专业：计算机科学与技术

姓名：姚知言

学号：2211290

指导教师：卢冶

2025 年 5 月 14 日

目录

一、 实验目的

1. 理解页表的含义，完成分页机制下虚拟地址到物理地址的映射。
2. 了解并实现多用户程序上下文切换。
3. 实现时钟中断，最终完成分时多任务及优先级调度的设计。
4. 最终实现完整的进程调度逻辑，在仙剑奇侠传，videotest 和 hello 中按照要求进行调度。

二、 实验重要内容

PA4 的实验共分为 3 个阶段，包括以下要点：

- 第 1 阶段：了解 i386 的分段机制。

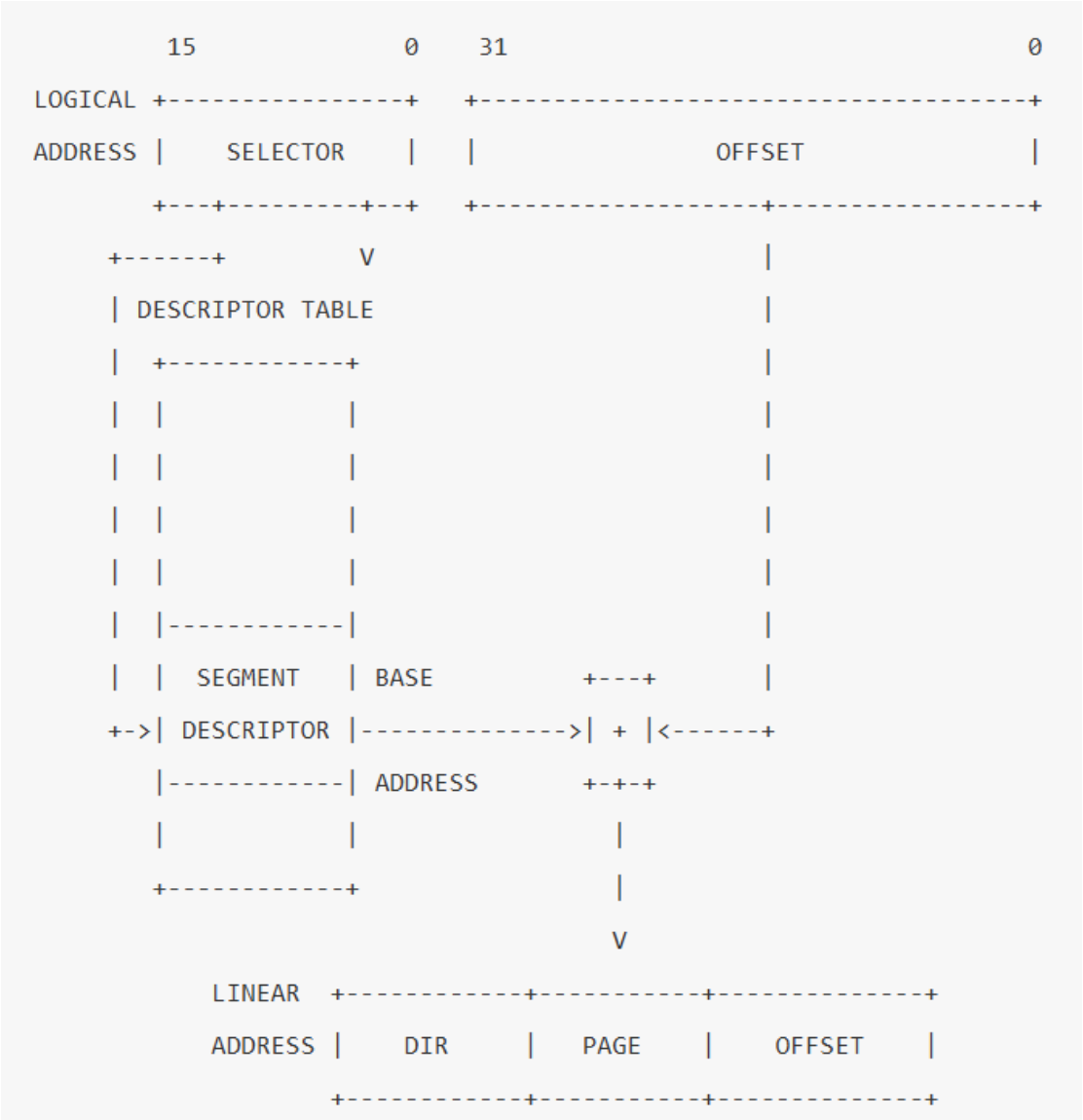

图 1: i386 分段机制

- 第 1 阶段：了解页式存储和页表，并完成虚拟地址经二级页表到物理地址映射的设计。

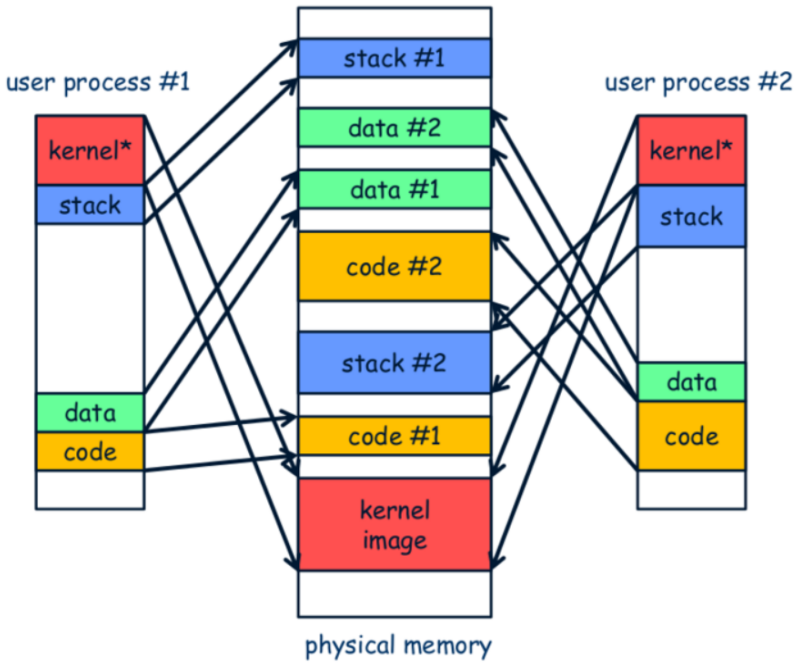

图 2: 页式内存存储

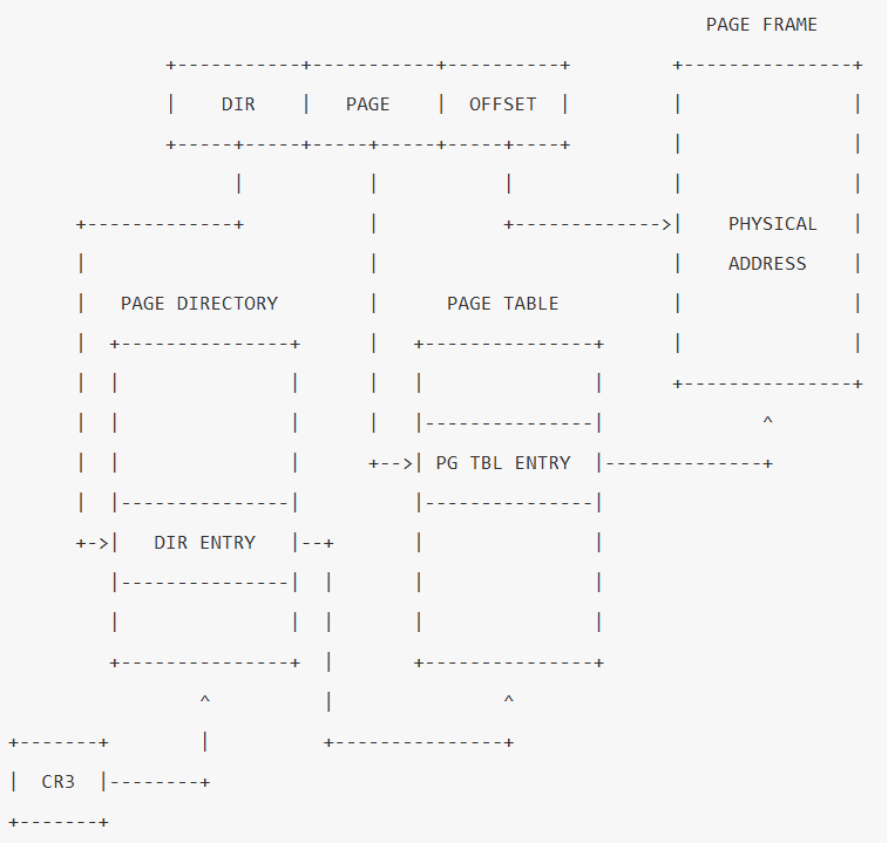

图 3: 二级页表的结构

- 第 1 阶段：借助物理页面的动态申请和页表的映射，实现用户程序的页式读取和动态内存分配。
- 第 2 阶段：实现陷阱帧及用户程序的内核自陷。

图 4: Trapframe

- 第 2 阶段：实现分时多任务及程序的优先级调度。
- 第 3 阶段：完成时钟中断的设计，借助时钟中断完成定时程序调度。
- 最终阶段：借助键盘，完成用户控制下的程序调度。

三、 实验方法与结果

(一) 阶段 1：在 NEMU 中实现分页机制

为准备内核页表，首先我需要在 nanos-lite/src/main.c 中打开 HAS_PTE 宏定义。

```
#define HAS_PTE
```

重新执行 pal，会发现运行到没有实现的指令。经查看是对 cr0 和 cr3 的 mov 指令。

为解决该问题，首先需要实现 cr0 和 cr3 寄存器，以启动页表并记录页表基址。在 CPU_state 中增加定义，并在 restart 函数中将 cr0 初始化为 0x60000011，以满足 difftest 要求。

```
uint32_t cr0, cr3;
//restart
cpu.cr0 = 0x60000011;
```

然后补全读写 cr0，cr3 的相关指令。首先实现 mov_c2r 和 mov_r2c，并在译码表中填写对应执行函数。该辅助函数实现根据寄存器选择从 cr0 或者 cr3 中读取，并增加 assert 保证没有意外的结果出现。

```
//2bytes
/* 0x20 */      IDEX(G2E, mov_c2r), EMPTY, IDEX(E2G, mov_r2c), EMPTY,

make_EHelper(mov_c2r);
make_EHelper(mov_r2c);

make_EHelper(mov_c2r){
  assert(id_src->reg == 0 || id_src->reg == 3);
  id_src->val = id_src->reg ? cpu.cr3 : cpu.cr0;
  operand_write(id_dest, &id_src->val);
  print_asm_template2(mov_c2r);
}
make_EHelper(mov_r2c){
  assert(id_dest->reg == 0 || id_dest->reg == 3);
  if(id_dest -> reg)
    cpu.cr3 = id_src->val;
  else
    cpu.cr0 = id_src->val;
  print_asm_template2(mov_r2c);
}
```

重新运行 pal，可以看到 pal 已经能够正常执行。通过 Log 可以看到在 init_mm 阶段，首先把 CR3 更新为 0x01d6e000，然后把 CR0 更新为 0xe0000011。

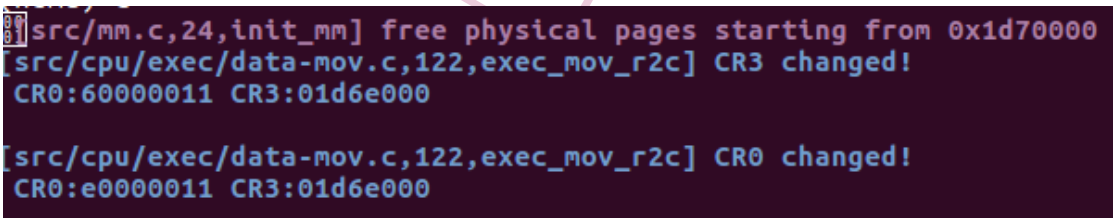

图 5: CR0/CR3 更新 Log

然后更新 vaddr_read 和 vaddr_write。为完成虚拟页面到物理页面的翻译，还需要实现 page_translate 函数。

首先需要定义一些宏和辅助函数，说明如下。

- BEGIN(pte)：根据页表项获取基地址（高 20 位）
- PDX(addr)：根据虚拟地址计算一级页表（页目录表）索引（最高 10 位）
- PTX(addr)：根据虚拟地址计算二级页表索引（中间 10 位）
- OFFSET(addr)：根据虚拟地址计算页内偏移量（低 12 位）
- use_paging()：检查分页使能和保护模式使能，以判断是否需要进行虚拟地址转换。

因为每个页表项占 4 字节，所以直接在宏定义处完成偏移量的左移。

具体实现如下:

```
#define BEGIN(pte) (((uint32_t)pte) & (~((1 << 12) - 1)))
#define PDX(addr) (((((uint32_t)addr) >> 22) & ((1 << 10) - 1))<< 2)
#define PTX(addr) (((((uint32_t)addr) >> 12) & ((1 << 10) - 1))<< 2)
#define OFFSET(addr) (((uint32_t)addr) & ((1 << 12) - 1))
static bool use_paging(){
  return (cpu.cr0 & 0x1) && (cpu.cr0 & 0x80000000);
}
```

然后实现 page_translate 函数，该函数逐级完成虚拟地址经页表到物理地址的翻译，并检查 present 位，更新 accessed 和 write 位。

```
paddr_t page_translate(vaddr_t addr, bool write){
  PDE pde = (PDE)(paddr_read(BEGIN(cpu.cr3) + PDX(addr),4));
   assert(pde.present);
  PTE pte = (PTE)(paddr_read(BEGIN(pde.val) + PTX(addr),4));
   assert(pte.present);
   pde.accessed = 1;
   pte.accessed = 1;
   pte.dirty |= write;
   return BEGIN(pte.val) | OFFSET(addr);
}
```

据此，完成 vaddr_read 和 vaddr_write 的更新。首先检查是否开启了端页机制需要翻译。如果需要的话则进行翻译，read 和 write 的主要区别在于是否需要进行翻译时是否需要置 dirty 位，以及最终调用的物理地址执行函数。

```
uint32_t vaddr_read(vaddr_t addr, int len) {
  if(!use_paging())
    return paddr_read(addr, len);
  paddr_t paddr = addr;
  if(BEGIN(addr) != BEGIN(addr+len-1))
    assert(0);
  else
    paddr = page_translate(addr, false);
  return paddr_read(paddr, len);
}

void vaddr_write(vaddr_t addr, int len, uint32_t data) {
  if(!use_paging())
    return paddr_write(addr, len, data);
  paddr_t paddr = addr;
  if(BEGIN(addr) != BEGIN(addr+len-1))
    assert(0);
  else
    paddr = page_translate(addr, true);
  return paddr_write(paddr, len, data);
}
```

首先对 dummy 进行测试，可以看到 HIT GOOD TRAP，且分页机制能够正确完成翻译。

```
[src/memory/memory.c,40,page_translate] CR3:1d6e000 BEGIN:1d6e000 ADDR:100032 PD
X:0
[src/memory/memory.c,42,page_translate] PDE:1d4e001
[src/memory/memory.c,45,page_translate] PDE:1d4e001 BEGIN:1d4e000 ADDR:100032 PT
X:400
[src/memory/memory.c,47,page_translate] PTE:100001
[src/memory/memory.c,53,page_translate] VADDR:100032 PADDR:100032
nemu: HIT GOOD TRAP at eip = 0x00100032
```

图 6: DUMMY 测试

然后对 pal 进行测试，会发现由于有跨页读写的情况，会发生 assert 0。

```
nemu: src/memory/memory.c:64: vaddr_read: Assertion `0' failed.
Makefile:46: recipe for target 'run' failed
make[1]: *** [run] 已放弃 (core dumped)
make[1]: Leaving directory '/home/latesoon/ics2017/nemu'
/home/latesoon/ics2017/nexus-am/Makefile.app:35: recipe for target 'run' failed
make: *** [run] Error 2
```

图 7: PAL 测试

在我们的设计下，最多一次读取 4 字节的数据，所以最多会发生跨两个页面读取数据的情况。对于多于两个页面的情况直接 assert0。然后对于两个页面的情况，本质上就是拆成两个页面分别去读写，然后进行拼接。获取尾地址的 begin（记为 vaddr2），从 addr 到 vaddr2（不含）为第一个界面的访问范围，vaddr2 之后为第二个页面的访问范围。

```
uint32_t vaddr_read(vaddr_t addr, int len) {
  if(!use_paging())
    return paddr_read(addr, len);
  paddr_t paddr = addr;
  if(BEGIN(addr) != BEGIN(addr+len-1)){
    if(BEGIN(addr) != BEGIN(addr+len-1)-(1<<12))
      assert(0);//do not reach
    vaddr_t vaddr2 = BEGIN(addr+len-1);
    int len1 = vaddr2 - addr;
    int len2 = len - len1;
    return (paddr_read(page_translate(addr,false),len1) ) | ((paddr_read(
        page_translate(vaddr2,false),len2))<< (len1 << 3));
  }
  paddr = page_translate(addr, false);
  return paddr_read(paddr, len);
}

void vaddr_write(vaddr_t addr, int len, uint32_t data) {
  if(!use_paging())
    return paddr_write(addr, len, data);
  paddr_t paddr = addr;
  if(BEGIN(addr) != BEGIN(addr+len-1)){
    if(BEGIN(addr) != BEGIN(addr+len-1)-(1<<12))
      assert(0);//do not reach
```

```
    vaddr_t vaddr2 = BEGIN(addr+len-1);
    int len1 = vaddr2 - addr;
    int len2 = len - len1;
    paddr_write(page_translate(addr,true),len1,data & ((1<<(len1<<3))-1));
    paddr_write(page_translate(vaddr2,true),len2,data>>(len1 << 3));
    return;
  }
  paddr = page_translate(addr, true);
  return paddr_write(paddr, len, data);
}
```

重新进行 pal 测试，可以运行。

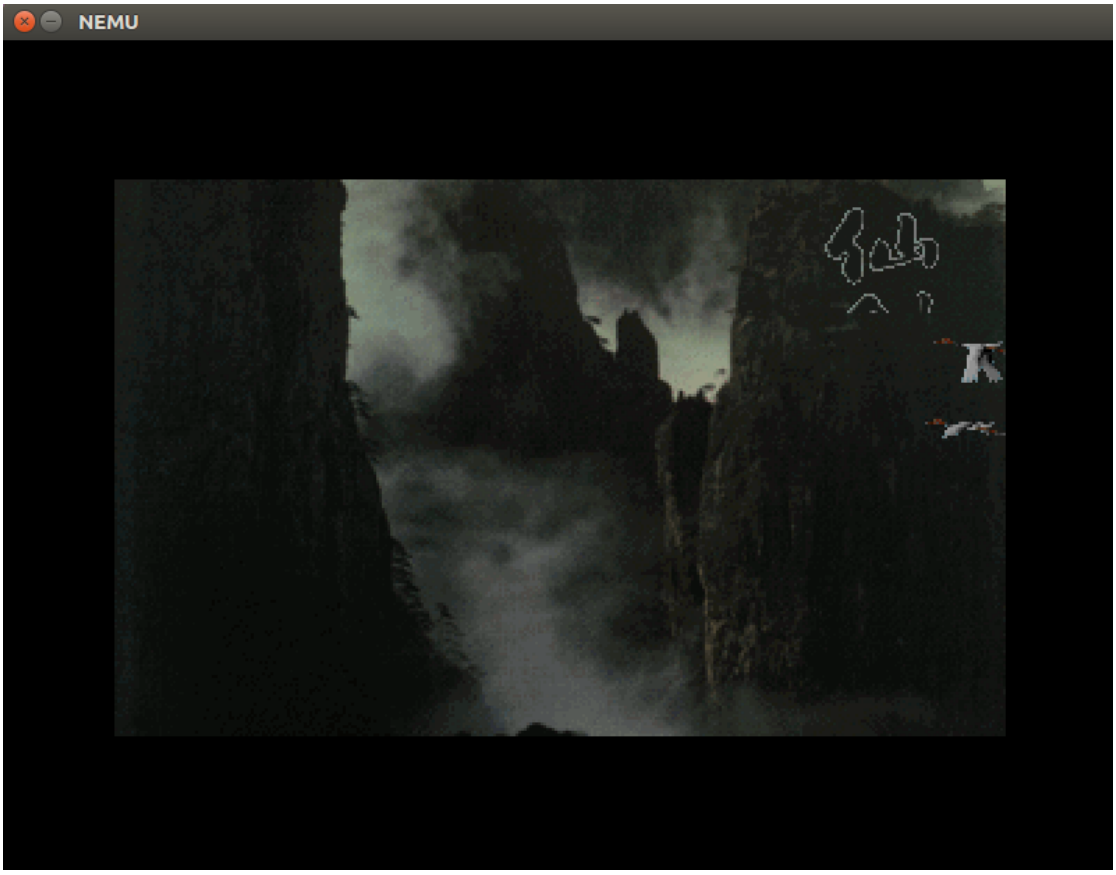

图 8: 重新进行 PAL 测试

(二) 阶段 1：让用户程序运行在分页机制上

首先需要修改开始地址，以保证用户程序运行在操作系统为其分配的虚拟地址空间之上，并且修改 main.c，以保证页面分配机制执行。

```
//navy-apps/Makefile.compile
LDFLAGS += -Ttext 0x8048000
//nanos-lite/src/loader.c
#define DEFAULT_ENTRY ((void *)0x8048000)
//nanos-lite/src/main.c
//uint32_t entry = loader(NULL, "/bin/pal");
```

```
//((void (*)(void))entry)();
load_prog("/bin/dummy");
```

然后重新实现 loader，不再像上一阶段一样一次性完成文件的读取，而是对每一个虚拟页分配新的物理页，然后按页读取数据。主要流程如下：

- 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。

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

  //PA3.2
  //fs_read(fd, DEFAULT_ENTRY, fs_filesz(fd));

  int size = fs_filesz(fd);
  for (int sz = 0; sz < size; sz+= PGSIZE){
    void* pa = new_page();
    _map(as, DEFAULT_ENTRY + sz, pa);
    fs_read(fd, pa, ((size - sz > PGSIZE) ? PGSIZE : (size - sz)));
  }
  fs_close(fd);
  return (uintptr_t)DEFAULT_ENTRY;
}
```

然后实现 _map 函数，完成虚拟地址空间到物理地址空间的映射。在这个过程中如果遇到一些页目录表还没有进行分配，则需要对其进行分配。

```
void _map(_Protect *p, void *va, void *pa) {
  PDE* dir = (PDE*)p->ptr;
  if(!(dir[PDX(va)] & PTE_P))
    dir[PDX(va)] = ((uint32_t)palloc_f() & ~0xFFF) | PTE_P;
  PTE* pte = (PTE*)(dir[PDX(va)] & ~0xFFF);
  pte[PTX(va)] = (uint32_t)pa | PTE_P;
}
```

dummy 测试通过，如图9所示。成功 HIT GOOD TRAP，且能够看到 map 函数正常运行。

```
(nemu) c
[src/mm.c,24,init_mm] free physical pages starting from 0x1d92000
[src/main.c,20,main] 'Hello World!' from Nanos-lite
[src/main.c,21,main] Build time: 14:53:57, May 12 2025
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x1029a0, end = 0x1d4c8fd,
 size = 29663069 bytes
[src/main.c,28,main] Initializing interrupt/exception handler...
[src/loader.c,26,loader] map!va:8048000 pa:1d93000
[src/loader.c,26,loader] map!va:8049000 pa:1d95000
[src/loader.c,26,loader] map!va:804a000 pa:1d96000
[src/loader.c,26,loader] map!va:804b000 pa:1d97000
[src/loader.c,26,loader] map!va:804c000 pa:1d98000
nemu: HIT GOOD TRAP at eip = 0x00100032
```

图 9: dummy 测试结果

(三) 阶段 1：在分页机制上运行仙剑奇侠传

首先需要实现 mm_brk 对堆区大小的调整。在这里我希望我的页面分配始终是以 4K 为单位的 max_brk 也始终保持 4K 单位。

```c
#define ALIGN_UP(x) (((x) + 0xFFF) & ~0xFFF)

int mm_brk(uint32_t new_brk) {
  if (current->cur_brk == 0) {
    current->cur_brk = new_brk;
    current->max_brk = ALIGN_UP(new_brk);
  }
  else {
    if (new_brk > current->max_brk) {
      current->max_brk = ALIGN_UP(current->max_brk);
      while(new_brk > current->max_brk){
        _map(&current->as, (void*)current->max_brk, new_page());
        current->max_brk += PGSIZE;
      }
    }
    current->cur_brk = new_brk;
  }
  return 0;
}
```

然后据此更新 syscall 的调用。在 SYS_brk 中调用 mm_brk，而不是直接返回 0。

```c
extern int mm_brk(uint32_t new_brk);

case SYS_brk:
  //SYSCALL_ARG1(r) = 0;
  SYSCALL_ARG1(r) = mm_brk((uint32_t)(a[1]));
  break;
```

进行 make update 后进行 make run，成功运行仙剑奇侠传。

图 10: 运行仙剑奇侠传

查看 log 可以发现 brk 系统调用堆区大小实现了正确更新。

```
[src/mm.c,37,mm_brk] new_brk:837d000 max_brk:837d000
[src/mm.c,37,mm_brk] new_brk:8381000 max_brk:8381000
[src/mm.c,37,mm_brk] new_brk:8381000 max_brk:8381000
[src/mm.c,37,mm_brk] new_brk:8383000 max_brk:8383000
[src/mm.c,37,mm_brk] new_brk:8383000 max_brk:8383000
[src/mm.c,37,mm_brk] new_brk:8387000 max_brk:8387000
[src/mm.c,37,mm_brk] new_brk:8387000 max_brk:8387000
[src/mm.c,37,mm_brk] new_brk:838b000 max_brk:838b000
[src/mm.c,37,mm_brk] new_brk:838b000 max_brk:838b000
[src/mm.c,37,mm_brk] new_brk:838b800 max_brk:838c000
[src/mm.c,37,mm_brk] new_brk:838b800 max_brk:838c000
[src/mm.c,37,mm_brk] new_brk:838c000 max_brk:838c000
[src/mm.c,37,mm_brk] new_brk:838c000 max_brk:838c000
```

图 11: brk log

(四) 阶段 2：实现内核自陷

首先更新 irq_handle 函数。函数的作用是处理硬件中断或软件异常，根据中断号（tf->irq）分发到不同的事件处理逻辑，调用上层事件处理程序。在此处补充 0x81 调用 _EVENT_TRAP。

```
1 _RegSet* irq_handle(_RegSet *tf) {
2     _RegSet *next = tf;
```

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

然后更新 _asye_init 函数，设置新的中断门（中断处理入口）。这里的 vectrap 函数需要在 trap.S 中以汇编语言的形式实现。

```
void vectrap();
void _asye_init(_RegSet*(*h)(_Event, _RegSet*)) {
  // ...
  // ------------------- system call ------------------------
  idt[0x80] = GATE(STS_TG32, KSEL(SEG_KCODE), vecsys, DPL_USER);
  idt[0x81] = GATE(STS_TG32, KSEL(SEG_KCODE), vectrap, DPL_USER);//新增
  //...
}
```

在 trap.S 中实现 vectrap 的原型。

```
.globl vectrap;  vectrap:  pushl $0;  pushl $0x81; jmp asm_trap
```

然后，完成对 _trap 的实现，本质上就是 int 0x81。

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

然后，对 nanos-lite 进行一些调整。

```
//在main.c新增_trap
load_prog("/bin/pal");
_trap();//新增
panic("Should not reach here");

//在proc.c中注释以下代码
// TODO: remove the following three lines after you have implemented _umake()
//_switch(&pcb[i].as);
//current = &pcb[i];
//((void (*)(void))entry)();
```

最后，在 do_event 中补充 _EVENT_TRAP 的分支，现在只需要简单打印输出测试，在后续补全逻辑。

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

```
    case _EVENT_SYSCALL: return do_syscall(r);
    case _EVENT_TRAP:
      printf("Do Trap!\n");
      break;
    default: panic("Unhandled event ID = %d", e.event);
  }
  return NULL;
}
```

成功打印了 Do trap!，说明成功完成了 trap 的调用。

```
(nemu) c
[src/mm.c,43,init_mm] free physical pages starting from 0x1d92000
[src/main.c,20,main] 'Hello World!' from Nanos-lite
[src/main.c,21,main] Build time: 09:30:45, May 13 2025
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x102a40, end = 0x1d4c99d,
 size = 29663069 bytes
[src/main.c,28,main] Initializing interrupt/exception handler...
Do Trap!
[src/main.c,40,main] system panic: Should not reach here
nemu: HIT BAD TRAP at eip = 0x00100032
```

图 12: Do trap

（五） 阶段 2：实现上下文切换

首先实现 _umake 函数。

- 首先要保存 start 的相关栈帧，因为不会真正使用，所以直接初始化为 0。
- 然后完成 trapframe 的初始化和压栈。
- 最后返回栈顶指针，记录在 PCB 中。

```
extern void* memset(void* v, int c, size_t n);
extern void* memcpy(void* dst, const void* src, size_t n);

_RegSet *_umake(_Protect *p, _Area ustack, _Area kstack, void *entry, char *
    const argv[], char *const envp[]) {
  memset((void*)ustack.end - 16, 0, 16);
  _RegSet tf;
  tf.eflags = 0x2;
  tf.cs = 0x8;
  tf.eip = (uintptr_t)entry;
  memcpy(ustack.end - 16 - sizeof(_RegSet), (void*)&tf, sizeof(_RegSet));
  return (_RegSet*)(ustack.end - 16 - sizeof(_RegSet));
}
```

然后完成 schedule 函数的实现，在这个函数中，需要保存当前的上下文（如有，通过 assert 观察 current 为 NULL 的时候会调用此函数），然后切换进程（目前全部切换到 pcb[0]）并返回新的上下文。

```
_RegSet* schedule(_RegSet *prev) {
  if(current!=NULL)
```

```
    current->tf = prev;
  current = &pcb[0];
  _switch(&current->as);
  return current->tf;
}
```

然后在 do_event 中更新 trap 的处理方式为调用 schedule，并删除之前临时实现用于验证功能的 print。

```
case _EVENT_TRAP: return schedule(r);
  //printf("Do Trap!\n");
  //break;
```

最后修改 trap.S 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。具体来说就是把先前的返回值（保存在 eax 中）读取到 esp 中来。

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

验证 PAL，成功运行。

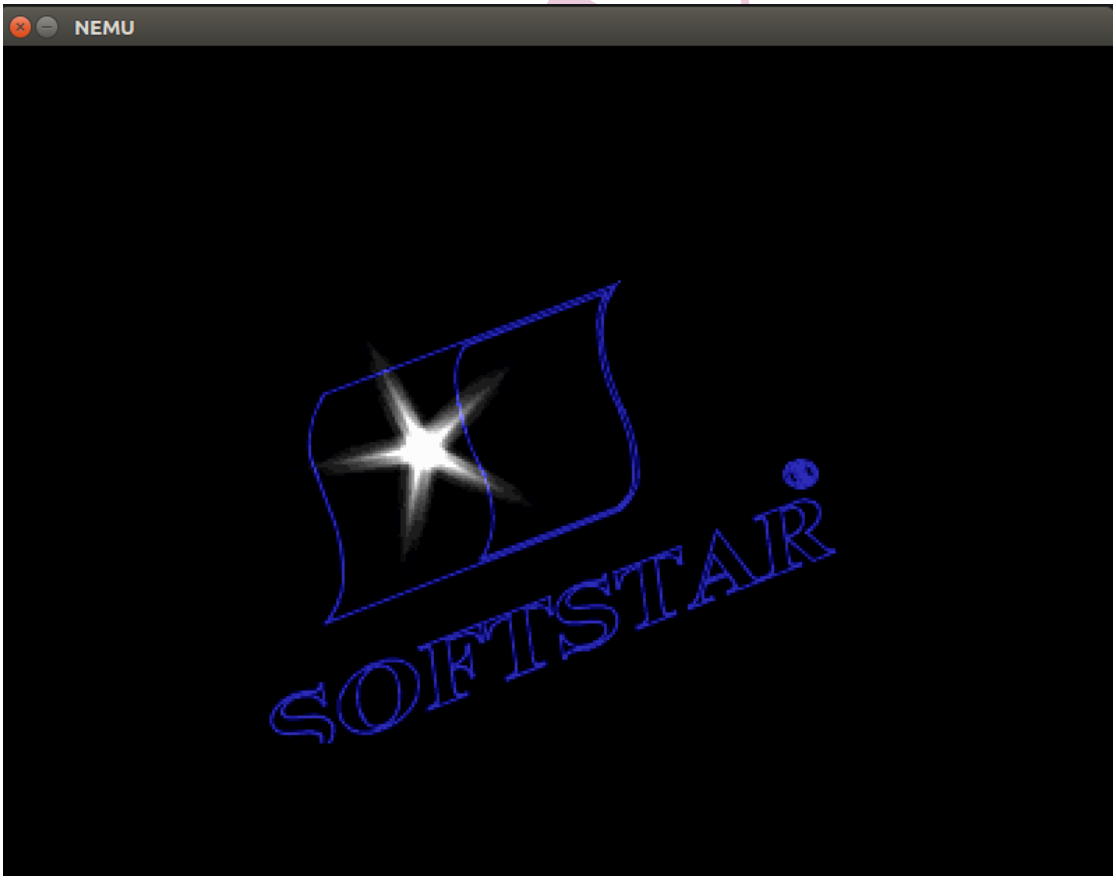

图 13: 验证 PAL

(六) 阶段 2：分时运行仙剑奇侠传和 hello 程序 & 优先级调度

为进行分时运行，首先需要在 main 函数中将仙剑奇侠传和 hello 程序全部加载进来。

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

然后更改 _EVENT_SYSCALL (在 irq.c 中), 使得每次系统调用之后都进行一次进程调度。

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

最后在 schedule 中修改进程调度规则，不再是每次都切换到第一个进程，而是在前两个进程之间相互切换。

```
//current = &pcb[0];
current = (current == &pcb[0] ? &pcb[1] : &pcb[0]);
```

重新执行 make run，可以看到 hello 和 pal 同时交替运行。但是可以看到 pal 的运行变得非常缓慢，原因是进程调度和 hello 的占用较大，但 hello 程序每次知识打印一行文字。

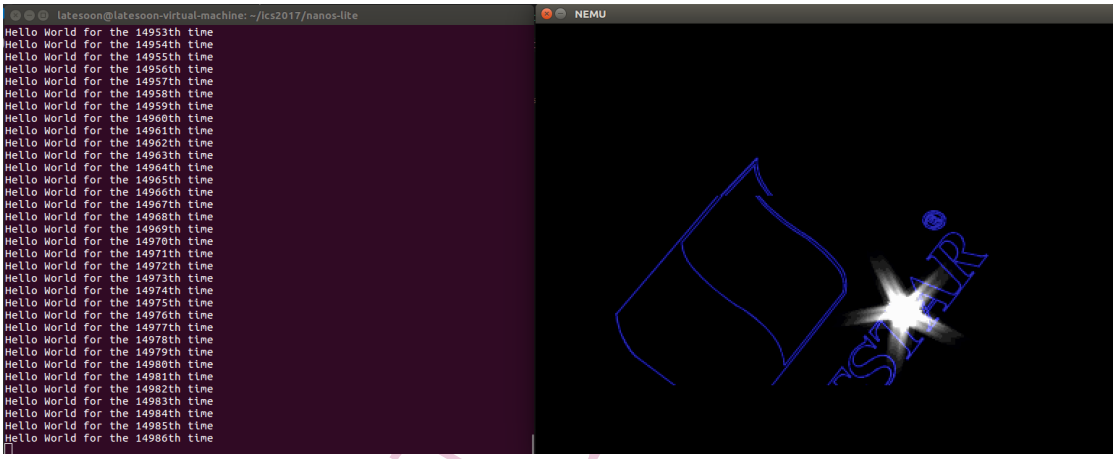

图 14: 分时运行仙剑奇侠传和 hello

为加速 pal 运行，我们进行优先级调度。我希望 pal 每进行 10000 次系统调用，才切换到 hello，而 hello 每次调用都切换到 pal。

进一步修改 schedule 函数如下。我们增加一个计数器 cnt，以计数 pal 执行系统调用的次数。注意第一个分支不仅仅包括正在执行 hello 进程，也包括 current 进程为 null 的情况，为简化实现都直接切换到 pal。而 pal 计数器到达 10000 时，则切换到 hello。

```
//current = (current == &pcb[0] ? &pcb[1] : &pcb[0]);
static int cnt = 0;
if(current != &pcb[0])//pcb[1] or NULL
    current = &pcb[0];
else if(++cnt >= 10000){
    cnt = 0;
    current = &pcb[1];
}
```

能够正常执行，如图15所示，pal 运行明显加快。

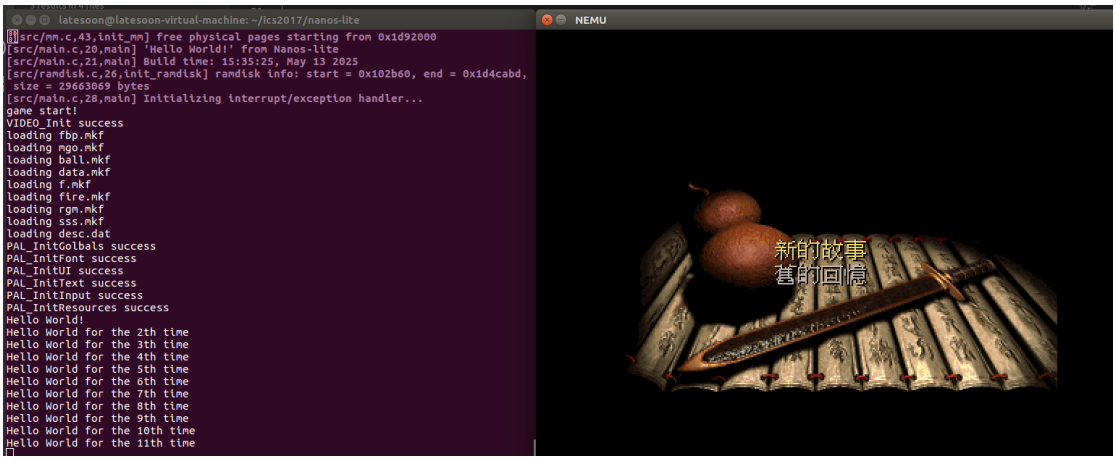

图 15: 优先级调度

(七) 阶段 3：添加时钟中断

为完成时钟中断，首先需要在 CPU_state 中添加 intr 寄存器，它是一个 bool 型变量，标记是否开启时钟中断。

```
bool intr;
```

在 dev_raise_intr 函数完成 intr 的初始化，将其初始化为 1，即高电平。

```
void dev_raise_intr() {
  cpu.intr = 1;
}
```

然后修改 exec.c 中 exec_wrapper 函数，在函数末尾添加轮询 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 TIMER_IRQ 32
void raise_intr(uint8_t NO, vaddr_t ret_addr);
void exec_wrapper(bool print_flag) {
  //...
  if (cpu.intr & cpu.IF) {
    cpu.intr = false;
    raise_intr(TIMER_IRQ, cpu.eip);
    update_eip();
  }
}
```

然后更新 raise_intr，将 eflags 保存后更新 IF 为 0，让处理器进入关中断状态，以确保在触发中断的时候不会被其他中断打断。

```
void raise_intr(uint8_t NO, vaddr_t ret_addr) {
  //...
  rtl_push(&cpu.eflags);
  cpu.IF = 0;//新增
  //...
}
```

接下来需要把时钟中断更新到事件中。首先是 do_event 函数，增加 _EVENT_IRQ_TIME 的分支，在里面调用 schedule 函数。（在此处的 log 是为了验证功能是否正确执行，在下一部分就注释掉了）

同时，syscall 分支里面的 schedule 已经不需要了。

```
static _RegSet* do_event(_Event e, _RegSet* r) {
  switch (e.event) {
    case _EVENT_SYSCALL: //do_syscall(r);return schedule(r);
      return do_syscall(r);
    case _EVENT_TRAP: return schedule(r);
    case _EVENT_IRQ_TIME:
      Log("Time interrupt!\n");
      return schedule(r);
    default: panic("Unhandled event ID = %d", e.event);
  }
}
```

在 asye.c 中，在 irq_handle 中添加 0x20(32) 分支，调用 _EVENT_IRQ_TIME。

```
_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 0x20: ev.event = _EVENT_IRQ_TIME;break;//新增
      default: ev.event = _EVENT_ERROR; break;
    }
    ..
  }

  return next;
}
```

然后在下面的 _asye_init 函数中，实现对应的陷门，并最终调用后续 trap.S 中的 vectime 函数。

```
void vectime();

idt[0x20] = GATE(STS_TG32, KSEL(SEG_KCODE), vectime, DPL_USER);
```

在 trap.S 中，增加 vectime 函数原型的实现。将事件编号 32（0x20）压栈。

```
.globl vectime;  vectime:  pushl $0;  pushl $0x20; jmp asm_trap
```

在 _umake 函数中，在 eflags 中开启 IF 位，以确保能够正常触发时钟中断。

```
_RegSet *_umake(_Protect *p, _Area ustack, _Area kstack, void *entry, char *
    const argv[], char *const envp[]) {
  //...
  tf.eflags = 0x2 | FL_IF;
```

```
    // ...
}
```

如图16所示，程序能够正常运行，左边时钟中断能够正常调用，schedule 中计数器正常运行，测试通过。

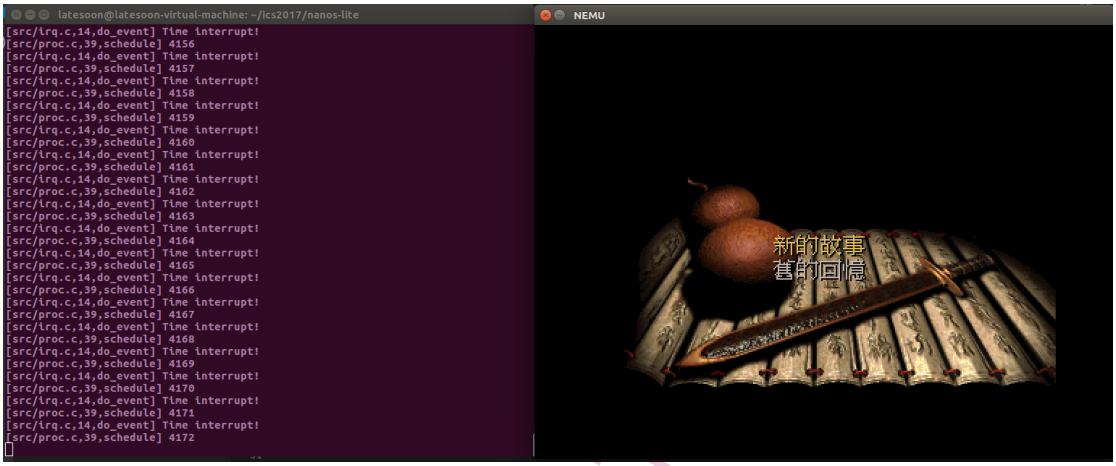

图 16: 时钟中断

如图17所示，进一步分析可以发现，在计数器达到 10000 时，发生一次进程调度到 hello 程序，然后在下一次时钟中断再调度回来。且可以看出 syscall 中的进程调度已经删掉了，否则打印一行就会调度，这里的调度就只能是时间中断造成的。

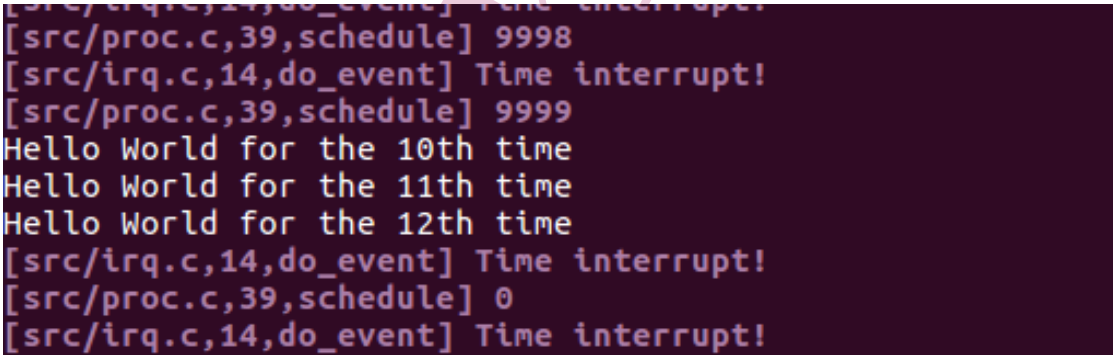

图 17: 进一步分析

(八) 编写不朽的传奇

首先在 main.c 中把 videotest 作为下标为 2 的进程加载进来。

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

然后修改 schedule，在：

- 当前进程是 hello 或者没有进程。
- 当前进程是 pal 或者 videotest，且计数器没有达到 10000。

情况下，根据 ispal 变量决定指定进程是 pal 或 videotest，否则指定为 hello。

ispal 是一个布尔型变量，通过检查键盘是否按下 F12 键进行更新。

```
bool ispal = 1;
_RegSet* schedule(_RegSet *prev) {
  if(current!=NULL)
    current->tf = prev;
  static int cnt = 0;
  if(current != &pcb[0] && current != &pcb[2])//pcb[1] or NULL
    current = (ispal ? &pcb[0] : &pcb[2]);
  else if(++cnt >= 10000){
    cnt = 0;
    current = &pcb[1];
  }
  else current = (ispal ? &pcb[0] : &pcb[2]);
  _switch(&current->as);
  return current->tf;
}
```

同时更新 event_read 函数，在检查到案件输入的时候，捕捉按下 F12 的信号，对 ispal 变量进行取反更新。

```
extern bool ispal;
size_t events_read(void *buf, size_t len) {
  int key = _read_key();
  if(key == _KEY_NONE)
    sprintf(buf,"t %d\n",_uptime());
  else{
    sprintf(buf,"%s %s\n",(key & 0x8000) ? "kd" : "ku", keyname[key & 255]);
    if((key & 0x8000) && ((key & 255) == _KEY_F12))//新增
      ispal = !ispal;//新增
  }
  return strlen(buf);
}
```

为了不同进程都能够访问相同文件 (dispinfo) 而不会报错，需要更新 fs_close 函数，在关闭文件的时候将其偏移量更新为 0。

```
int fs_close(int fd){
  update_offset3(fd,0,SEEK_SET);
  return 0;
}
```

重新运行，能够正常在 pal 和 hello 之间切换。

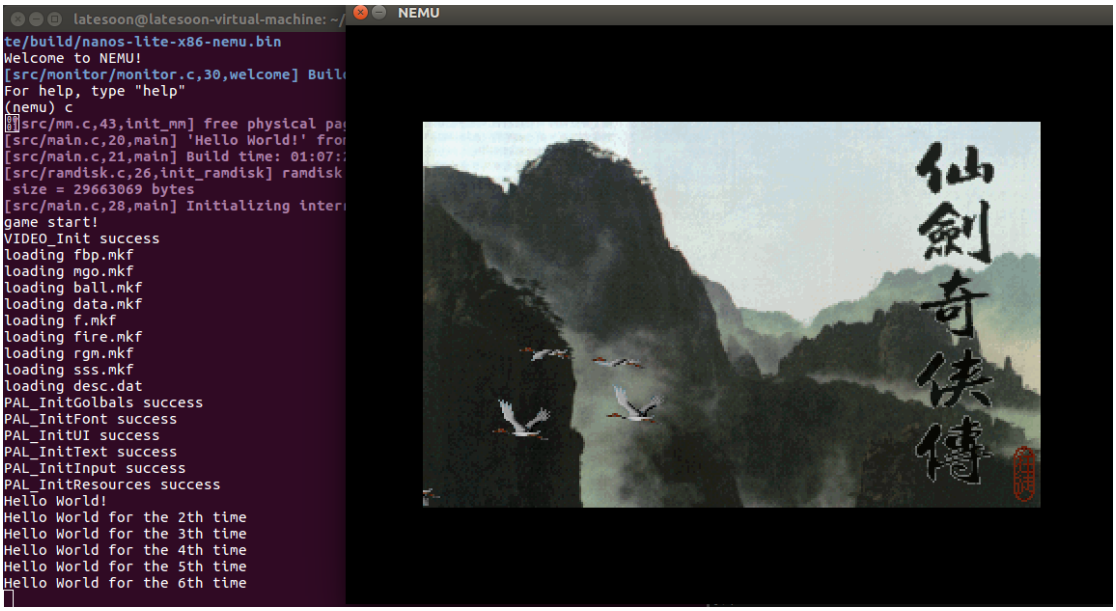

图 18: PAL/HELLO

按下 F12，videotest 正确执行，在 videotest 和 hello 之间切换。再点击 F12，返回 pal 和 hello 之间切换。

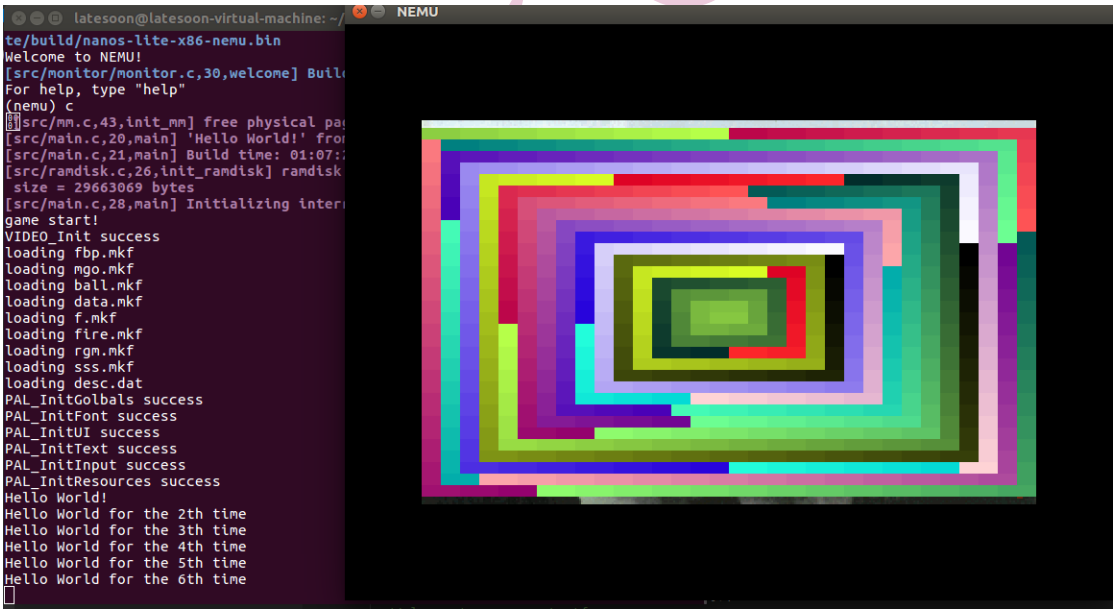

图 19: VIDEOTEST/HELLO

四、 问答题

(一) 阶段 1：页表设计原理

- Q1:

  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?

- A1:

i386 的每一页有 4KB，存储也是 4K 对齐的，地址的最后 12 位是页内偏移量，页表项中的 20 位基地址与 12 位偏移量组合形成完整的 32 位物理地址。

这样的结构不仅可以降低开销，还可以在低 12 位中保存标志位（如 Present, R/W, U/S 等），维护页表项的状态。

- Q2:

  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?

- A2:

  物理地址是必须的，不能使用虚拟地址。若页表本身通过虚拟地址访问，转换虚拟地址需要先访问页表，而访问页表又需要转换虚拟地址，这样显然就形成了无限的递归调用。

  因此，页表中存储的必须是物理地址，才能保障地址转换的正确运行。

- Q3:

  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?

- A3:

  一级页表容易造成以下问题:

  – 空间浪费：对于 32 位地址空间，一级页表需要 2^{20} 个表项（1MB 内存），而大多数进程实际使用的地址空间很小。

  – 灵活性差：无法实现按需分配，即使只使用少量内存也必须维护完整页表。

  – 共享困难：多级页表更容易实现部分地址空间共享。

  虽然在极限情况下，二级页表的开销并不比一级页表小。但在实际应用中，大量没有分配的虚拟内存在二级页表的结构下并不需要创建第二级页表，从而大幅减少开销。

(二) 阶段 1：空指针真的是” 空” 的吗?

- Q:

  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?

- A:

  在软件实现上来说有一定的道理，C/C++ 等语言标准规定 NULL（或 nullptr）表示” 无效指针”，解引用它属于未定义行为。

  但实际上系统做不到真的不在指针对应的内存里面存储数据。NULL 通常是全零值(0x00000000), 而 CPU/MMU 会将其视为普通地址。在大多数操作系统中，包括 nemu 中，地址 0 所在的页面会被故意设置为不可访问（即检查 present 时无法通过）。

  程序解引用过程中，CPU 将指针值 0x00000000 发送，并逐级进行地址转换。最终发现对应的物理地址不能被访问。操作系统捕获该异常，并触发错误终止程序。

  因此，空指针的” 无效性” 不是硬件特性，而是软件约定。这一情况如果不合理的进行保护的话，则有成为系统漏洞的风险。

(三) 阶段 1：内核映射的作用

- Q:

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

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

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

- A:

  注释后发生了以下错误信息。

```
CR3:1d92000 BEGIN:1d92000 ADDR:1013b0 PDX:0 PDE:0
nemu: src/memory/memory.c:43: page_translate: Assertion `pde.present' failed.
Makefile:46: recipe for target 'run' failed
make[1]: *** [run] 已放弃 (core dumped)
make[1]: Leaving directory '/home/latesoon/ics2017/nemu'
/home/latesoon/ics2017/nexus-am/Makefile.app:35: recipe for target 'run' failed
make: *** [run] Error 2
```

图 20: 错误信息

其原因是，在创建虚拟地址空间的时候，对页目录表进行了初始化。初始化后页目录表中的合法页表项的 present 位为 1，指向的物理地址也被正确赋值，因此可以进一步进行二级页表的访问。若不对页目录表进行初始化，在访问虚拟地址对应的页目录表项的时候，会发生 present 为 0 的报错。

(四) 阶段 3：灾难性的后果

- Q:

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

- A:

  假设在中断 1 处理的过程中，新的中断 2 开始执行。这样的话中断 2 会覆盖掉中断 1 的 trapframe。在中断 2 返回后，中断 1 只能在已经损坏的 trapframe 中继续执行，这样有可能会导致以下问题。

  - 寄存器/状态损坏：中断 1 的原始上下文（如 EIP、ESP）被中断 2 覆盖，导致中断 1 返回时跳转到无效地址或发生栈指针错误（栈溢出或访问非法内存）。
  - 双重故障：若中断 1 返回时因上下文损坏导致新的异常（如页错误），可能触发双重故障。
  - 数据一致性问题：若中断处理涉及关键数据结构（如调度器），嵌套中断可能导致数据竞争或死锁。

（五） 万变之宗 - 重新审视计算机

- Q:

  什么是计算机？为什么看似平淡无奇的机械，竟然能够搭建出如此缤纷多彩的计算机世界？那些酷炫的游戏画面，究竟和冷冰冰的电路有什么关系？看着仙剑奇侠传运行的画面，不妨思考一下，NEMU 和 AM 分别如何支撑仙剑奇侠传的运行？

图 21: 计算机结构

- A:

  计算机的本质是一种通过电路实现数学逻辑的通用机器，其核心能力来自其图灵完备性，冯·诺依曼架构，多层级抽象等。普通的与非门电路，通过数亿个晶体管的协同运作，在布尔代数的数学规律下形成了信息处理能力。

  NEMU 和 AM 分别作为计算机模拟器和操作系统的部分,支撑着仙剑奇侠传的运行。NEMU 通过” 欺骗” 让游戏认为自己运行在真实硬件，AM 通过标准化接口消除硬件差异。

  用确定性的物理规律，通过层层抽象，最终支撑起人类情感丰富的数字世界。

五、 必答题：分时多任务的具体过程

- Q:

  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。

- A:

  分页机制保证了不同进程拥有独立的存储空间。NEMU 提供了 CR0,CR3 两个寄存器。其中 CR0 表示了是否开启虚拟地址到物理地址的映射，CR3 表示了当前进程下页表的基地址。

  MMU 实现了虚拟地址到物理地址的转换，具体逻辑为在 vaddr_read 和 vaddr_write 中调用 page_translate 函数，完成一次 page walk，并且对页表项记录标记位。

```
paddr_t page_translate(vaddr_t addr, bool write){
  PDE pde = (PDE)(paddr_read(BEGIN(cpu.cr3) + PDX(addr),4));
  assert(pde.present);
  PTE pte = (PTE)(paddr_read(BEGIN(pde.val) + PTX(addr),4));
  assert(pte.present);
  pde.accessed = 1;
  pte.accessed = 1;
  pte.dirty |= write;
  return BEGIN(pte.val) | OFFSET(addr);
}
```

Nanos-lite 通过 new_page 函数，使得用户程序读取或者程序执行的时候，能够动态的申请新的物理页，并通过 _map 函数完成页表的映射关系。

```
void* pa = new_page();
_map(as, DEFAULT_ENTRY + sz, pa);
```

此外，_pte_init 函数实现了内核页表，以便于使用内核的虚拟空间。

对于用户程序 pal/hello/videotest/...，Nanos-lite 通过 load_prog 实现用户程序的加载。该函数调用 _protect 函数将内核页表的内容拷贝到用户进程的页表，完成虚实地址映射。然后通过 loader 加载文件，最后开辟栈帧空间，并通过 umake 初始化 trapframe，以进行进程切换。

```
void load_prog(const char *filename) {
  int i = nr_proc ++;
  _protect(&pcb[i].as);
  uintptr_t entry = loader(&pcb[i].as, filename);
  _Area stack;
  stack.start = pcb[i].stack;
  stack.end = stack.start + sizeof(pcb[i].stack);
  pcb[i].tf = _umake(&pcb[i].as, stack, stack, (void *)entry, NULL, NULL);
}
```

NEMU 的 exec_wrapper 每执行完一条指令，就会检查时钟中断。若触发时钟中断则会将其包装成 event，以最终调用 schedule 函数，切换进程。

在 schedule 函数中，会保存当前进程的 trapframe，并进行进程调度，读取新进程的上下文。最终将新进程的 tf 作为返回值，在中断返回的时候 asm_trap 通过 mov 指令让栈顶指针指向它，从而实现了程序的分时运行。

六、 所遇 BUG 及解决思路

(一) 跨页读写拼接问题

在完成分页机制的 pal 跨页读写的时候，遇到了 present 为 0 的报错。
增加详细 assert 打印，发现这个位置读到的 PDE 为 0，也就是根本没有进行初始化。

```
PAL_InitGolbals success
PAL_InitFont success
PAL_InitUI success
PAL_InitText success
PAL_InitInput success
PAL_InitResources success
CR3:1d6e000 BEGIN:1d6e000 ADDR:80000a0 PDX:80 PDE:0
nemu: src/memory/memory.c:43: page_translate: Assertion `pde.present' failed.
```

图 22: 读写拼接问题

我一开始以为是我递归调用占用了太多空间导致的问题，但去掉之后也没能解决。在反复阅读多次代码后，最终发现问题来源是跨页读数据的时候，我错误的把较前获取的数据向左移位后进行拼接，这与 i386 数据扩展方式是不一致的。

修改其为将新页读取的数据向左移位后拼接，解决了该问题。

(二) 指针计算问题

一开始，我在实现上下文切换中遇到了未定义指令访问：ff ff... 的行为。经过多次检查后没有收获，最终发现是在设计 _umake 中出现的问题。

我先前的设计如下：

```
return (_RegSet*)ustack.end - 16 - sizeof(_RegSet);
```

这样的写法看起来没什么问题，但实际上先转换了 ustack.end 的指针从 void* 到 _RegSet* 后，后续的加减运算都是以 sizeof(_RegSet) 为基本单位进行偏移了。

修改后如下，这样先以 void* 进行计算，最后完成类型转换，则不再出现该问题。

```
return (_RegSet*)(ustack.end - 16 - sizeof(_RegSet));
```

(三) 不同文件访问 DISPINFO 造成的冲突问题

在 PA3 的实现中，fs_close 可以直接返回 0，因为当时一次只会运行一个程序。但在编写不朽的传奇中，由于 pal 和 videotest 都需要读取该文件。于是无论谁先开始执行，执行第二个文件的时候都会发生 assert 断言错误。

一开始我认为是进程调度的问题，但打 log 后发现没有问题，且之前在 hello 和 pal 切换的时候也没有发生过问题。

仔细分析，这两个参数应该是从 dispinfo 中读出来的，这个报错应该是在读取 dispinfo 的时候没能正确读取。

```
Assertion failed: screen_w > 0 && screen_h > 0, file src/ndl.c, line 141
nemu: HIT BAD TRAP at eip = 0x00100032
```

图 23: DISPINFO 读取失败

进一步分析，ndl.c 中 get_display_info 进行了一次完整的 open，read，close。那么问题就比较确定了，dispinfo 是 offset 相关的，需要在 close 的时候更新它的 offset。

```
static void get_display_info() {
  FILE *dispinfo = fopen("/proc/dispinfo", "r");
  assert(dispinfo);
  screen_w = screen_h = 0;
```

```
  char buf[128], key[128], value[128], *delim;
  while (fgets(buf, 128, dispinfo)) {
    *(delim = strchr(buf, ':')) = '\0';
    sscanf(buf, "%s", key);
    sscanf(delim + 1, "%s", value);
    if (strcmp(key, "WIDTH") == 0) sscanf(value, "%d", &screen_w);
    if (strcmp(key, "HEIGHT") == 0) sscanf(value, "%d", &screen_h);
  }
  fclose(dispinfo);
  assert(screen_w > 0 && screen_h > 0);
}
```

修改 fs_close 后，程序能够成功运行。

```
int fs_close(int fd){
  update_offset3(fd,0,SEEK_SET);
  return 0;
}
```