南开大学
Nankai University

南　　开　　大　　学

计算机系统设计实验报告

PA4 - 虚实交错的魔法：分时多任务

蒋薇

年级：2021 级

专业：计算机科学与技术

2024 年 6 月 4 日

目录

一、 实验内容

虚拟内存映射与分页机制

上下文切换基本原理、进程调度、分时多任务

硬件中断并实现时钟中断

stage1：理解虚拟地址空间、分页机制, 并在 NEMU 中实现分页机制, 先让用户程序 dummy 运行在分页机制，后在分页机制上运行仙剑奇侠传。

stage2: 实现内核自陷、上下文切换与分时多任务。

stage3: 分时多任务改为使用时钟中断来进行进程调度, 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。

二、 Stage 1

虚拟内存空间

虚拟内存, 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。

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。

MMU(Memory Management Unit, 内存管理单元), 它是虚拟内存机制的核心，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.

分页机制，把连续的存储空间分割成小片段, 以这些小片段为单位进行组织, 分配和管理。

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, 一张页目录或页表的大小是 4KB, 放在内存中, 个 CR3(control register 3) 寄存器, 专门用于存放页目录的基地址.

(一) 在 NEMU 中实现分页机制

准备内核页表

在 nano-lite/src/main.c 中定义宏 HAS_PTE

$defineHAS_PTE$

make run 后发现错误 $invalidopcode(eip = 0x0010186b) : 0f22d80f20c08945$

10186b: 0f 22 d8 mov %eax,%cr3

即 cr3 的 mov 指令未实现。

在 nemu/include/cpu/reg.h 当中，添加 CR0 和 CR3 两个控制寄存器

CPU_state 中 unit32_t CR0;unit32_t CR3;

1. 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。

2. 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。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。

$$\mathrm{rtl}_{l}oad_{c}rrtl_{s}tore_{c}r$$

```
//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;
}

//rtl_store_c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;
}
```

实现 CR0 寄存器 PG 位置，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。

CR0 寄存器初始化为 0x60000011

```
//differential testing 机制正确工作
    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
}
```

实现 mov 指令于访问与修改 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

0f 20 /r MOV r32,CR0/CR2/CR3 ,move (control register) to (register)

$$\mathrm{make}_D Helper(mov_load_cr) make_D Helper(mov_store_cr)$$

```
//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) {
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
}

//把目标寄存器的值保存到控制寄存器
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);

//修改 exec.c 的2 byte op_table:

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

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

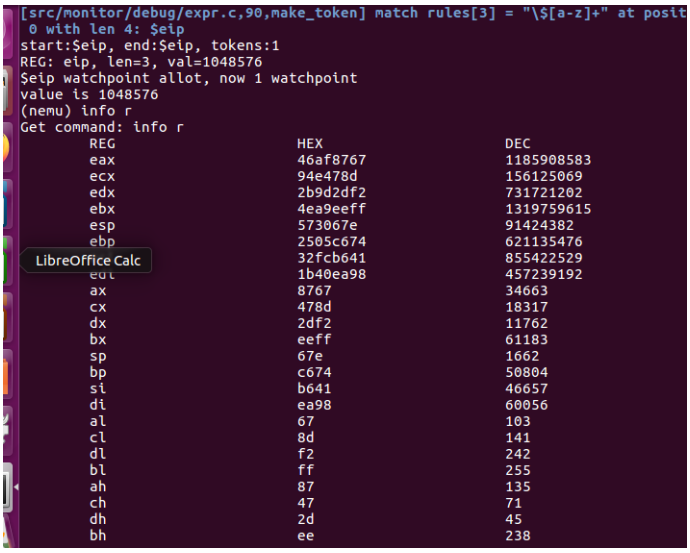

图 1: 相关寄存器信息

(二) dummy 运行在分页机制上准备

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

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

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

修改的 vaddr_read() 和 vaddr_write() 函数，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。

$vaddr_read vaddr_write$

```
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);
}
```

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。

page$_t$*ranslate*()

```
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;
}
```

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

(三) 仙剑奇侠传运行在分页机制上准备

数据跨页, 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.

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

重新修改 vaddr$_read$ vaddr$_write$

```
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);
}
```

(四) dummy 运行在分页机制上

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。

nanos-lite/src/loader.c 中的 DEFAULT_ENTRY 也需要作相应的修改:$define DEFAULT_ENTRY((void*)0x80$

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

修改 loader

```
//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。

修改$_map()$

```
//按页加载
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。

实现正确后，看到 dummy 程序最后输出 GOOD TRAP 的信息，说明它确实在虚拟地址空间上成功运行了.

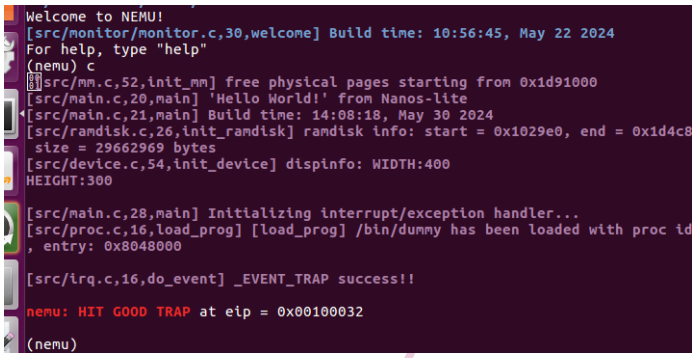

图 2: dummy

(五) 仙剑奇侠传运行在分页机制上

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。

()

```
//按页加载
/* 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;
}
int sys_brk(int addr) {
extern int mm_brk(uint32_t new_brk);
return mm_brk(addr);
}
```

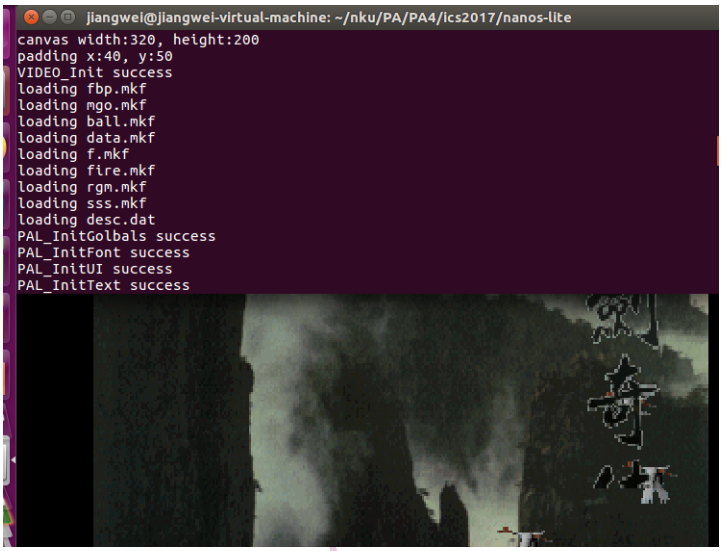

图 3: pal

实现正确后, 仙剑奇侠传就可以正确在分页机制上运行。

三、 Stage2

(一) 实现内核自陷

我们约定内核自陷通过指令 int $0x81 触发。

实现内核自陷

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

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，返回第一个用户程序的现场了。

irq$_h$*andle*()

```
 _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;
}
```

在 do_event 的 _EVETN_TRAP 事件中增加一条输出信息，以验证正确性。

do$_e$*vent*()

```
 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;
}

//声明中断描述符的入口函数
void vecsys();
void vecnull();
void vecself();

//asye_init() 函数中增加0x81描述符, 入口函数设置为 vecself

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;
}

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

(二) 上下文切换准备

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。

指针 tf 来记录陷阱帧的位置

设置好 _start() 函数的栈帧，_start() 开始执行的时候, 可以访问到正确的栈帧, 把栈帧中的参数设置为 0 或 NULL .

操作系统使用进程控制块 (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，需要把 PCB 中的 tf 指针返回给 ASYE 的 irq_handle() 函数.

_umake() 函数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.

_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。

$_{m}ake()$

```
    _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;
}
```

schedule() 函数完成进程调度，定具体切换到哪个进程的上下文，current 指针们记录了正在运行的进程，该指针指向当前运行进程的 PCB，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。

schedual()

```
   _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 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;
}
```

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。

栈顶指针切换到新进程的陷阱帧

```
  asm_trap:
pushal
pushl %esp
call irq_handle
# addl $4, %esp
movl %eax, %esp
popal
addl $8, %esp
iret
```

可通过自陷形式触发 pal。

(三) 分时运行仙剑奇侠传和 hello 程序

main.c 中加载第二个用户程序，让 schedule() 轮流返回仙剑奇侠传和 hello

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

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.

分时运行 hellopal

```
 _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 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;
}
```

pal 加载速度变缓慢

（四） 优先级调度

为了让仙剑奇侠传尽量保持原来的性能，调整仙剑奇侠传和 hello 程序调度的频率比例，仙剑奇侠传调度若干次，才让 hello 程序调度 1 次。

优先级调度

```
_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;
}
```

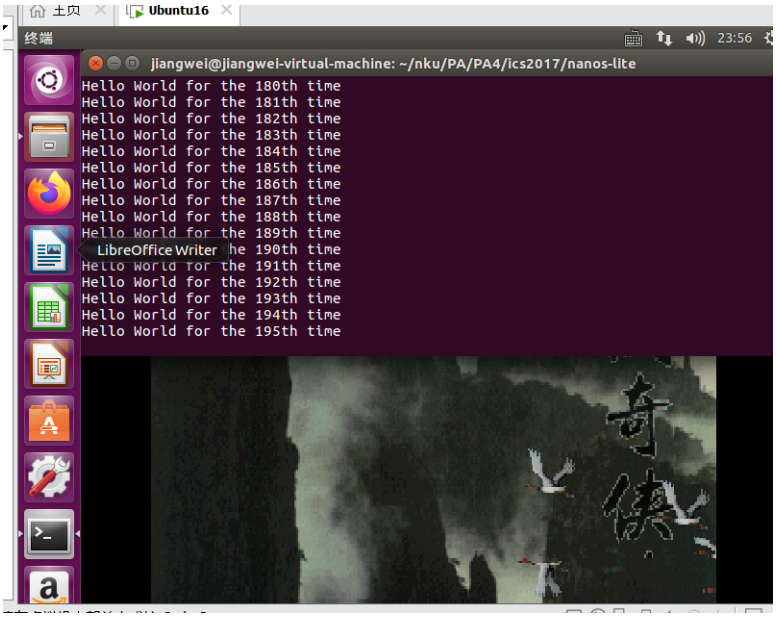

图 4: hello&pal 分时

四、 State3

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。

(一) 添加时钟中断

让时钟中断连接到 CPU 的 INTR 引脚，约定时钟中断的中断号是 32。时钟中断通过 timer_intr() 触发，每 10ms 触发一次, 触发后，调用 dev_raise_intr() 函数.

添加时钟中断

```
    //在 cpu 结构体中添加一个 bool 成员 INTR
    bool INTR;
    // 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 raise$_i ntr()$

```
    //保存 eflags 后关中断（将IF位设置为0),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);
```

添加代码，支持软件中断

修改 raise$_i ntr()$

```
    //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;
}
```

声明 vectime 修改 do$_{e}$*vent*

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

//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;
}
```

$_{u}$*make*()

```
  //设置eflags寄存器
  _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;
}
```

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。

(二) F12 切换仙剑奇侠传 videotest

添加第 3 个用户程序 /bin/videotest，添加功能，按下 F12 的时候，让游戏在仙剑奇侠传和 videotest 之间切换。

切换

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

//events_read()检测F12按键, 并切换程序
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;
}
```

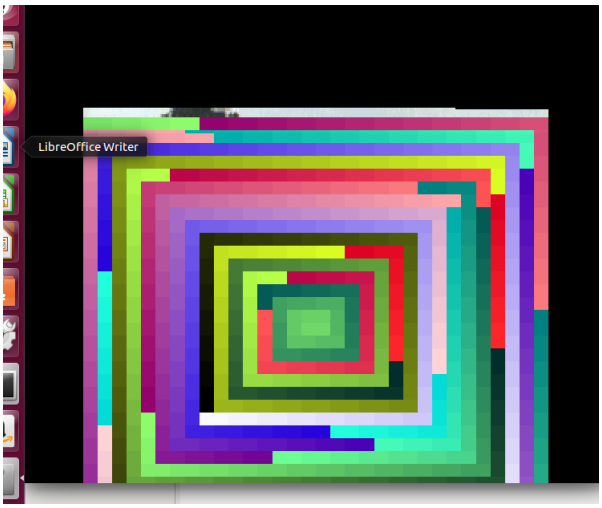

图 5: 切换

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

AM 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。

Nanos-lite 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。

Nemu 部分分页主要是通过 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。

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, 时钟中断（_EVENT_IRQ_TIME ），和内核自陷（_EVENT_TRAP ）

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。

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.

五、 问题

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？

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 = 212 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 220 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。

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？

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。

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？

一级页表指的是将整个虚拟地址空间映射到一个单一级别的页表结构，这意味着每个虚拟地址都直接映射到物理地址，因此查找页表项的时间复杂度是常数级别的。

它需要为整个虚拟地址空间分配一个相同大小的页表，这可能导致大量内存浪费。随着地址空间的增长，页表可能会变得非常大，导致查找时间增加，从而影响性能。如果页表太大而无法完全存储在 CPU 高速缓存中，每次访问页表都需要访问内存，这也会降低性能。一级页表结构不够灵活，无法提供一些高级内存管理功能，例如页面共享或只读权限。

在实践中，多级页表结构更常见，因为它们能够更好地解决这些问题并提供更高效和灵活的内存管理机制。

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?

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。

内核映射的作用

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.

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。