

计算机系统设计实验

PA4 - 虚实交错的魔法：分时多任务

安祺 1913630

南开大学计算机学院

日期：2022 年 5 月 23 日

目录

1 概述	2	3 阶段二	7
1.1 实验目的	2	3.1 实现内核自陷	7
1.2 实验内容	2	3.2 实现上下文切换	8
		3.3 分时多任务	10
2 阶段一	2		
2.1 回答一些问题	2	4 阶段三	10
2.2 加入 PTE	2	4.1 灾难性的后果	10
2.2.1 CR0、CR3 及相应指		4.2 添加时钟中断	10
令的实现	3	4.3 展示你的计算机系统	11
2.2.2 虚拟地址读写函数			
的修改	3	5 bug 汇总	13
2.3 让用户程序运行在分页机		5.1 移位	13
制上	4	5.2 mm_brk	13
2.3.1 dummy	4		
2.3.2 内核映射的作用	5	6 必答题	13
2.3.3 仙剑奇侠传	6	7 体会与感悟	13

1 概述

1.1 实验目的

1. 理解体会分时多任务系统的原理

1.2 实验内容

阶段一 实现分页机制。

阶段二 实现上下文切换。

阶段三 实现真正的分时多任务。

2 阶段一

2.1 回答一些问题

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? 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? 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?

表项中 20 位基地址就够了, 因为只有两级页表, 页表中有 1024 个表项, 同时 32 位表项中还需要保留 12 位偏移, 在 4KB 的页帧中进行寻址。

CR3 物理地址是必须的, 否则无法自己寻址自己。

一级页表的话, 要么页表太大不好管理, 要么页表太小无法管理足够大的内存空间。

NULL 其实是 0, 作为保留地址, 访问时会触发页错误, 然后进一步处理触发空指针异常。

2.2 加入 PTE

首先记得在 *nanos-lite/src/main.c* 中定义宏 *HAS_PTE* 以前实验经常范类似错误。

2.2.1 CR0、CR3 及相应指令的实现

在 mmu.h 中定义了 CR0 和 CR3 两个结构体，我直接在 CPU_state 中定义相应的 cr0 和 cr3 寄存器。在 i386 中操作 cr0 与 cr3 的指令是 MOV(操作码为 0x0F20 和 0x0F22)，在 NEMU 中用 mov_r2cr 和 mov_cr2r 表示相应的执行函数，两者分布借助 mov_G2E 和 mov_E2G 进行译码。mov_r2cr 和 mov_cr2r 的执行函数有 TODO 没有完成，需要进行实现。mov_cr2r 的实现如下，mov_r2cr 类似不再重复。

Listing 1: exec_mov_cr2r 函数的实现

```
switch (id_src->reg) {
    default: Assert(false, "register is unsupported");
    case 0: t0 = cpu.cr0.val; break;
    case 3: t0 = cpu.cr3.val; break;
}
operand_write(id_dest, &t0);
```

此外，为了 diff_test 正确运行，需要在 restart() 函数中将 CR0 寄存器初始化为 0x60000011

2.2.2 虚拟地址读写函数的修改

除了指导书[1]中给出的代码,还需要实现数据是否跨页边界的判断和 page_translate 函数,二者实现如下。其中, page_translate 函数需要实现分页机制中 accessed 位和 dirty 位的功能,但暂时不实现数据跨页边界的处理。

Listing 2: 是否跨页边界的判断

```
static inline bool cross_page(vaddr_t addr, int len){
    vaddr_t startpg=addr & ~PAGE_MASK;
    vaddr_t endpg=(addr + len - 1) & ~PAGE_MASK;
    return startpg != endpg;
}
```

Listing 3: page_translate 函数的实现

```
paddr_t page_translate(vaddr_t addr, bool write) {
    /* if 没有开启分页机制 */
    if(!(cpu.cr0.protect_enable && cpu.cr0.paging)){
        return addr;
    }

    /* 访问页目录表 */
    PDE *pdir=(PDE*)(cpu.cr3.page_directory_base << 12);
    PDE pde;
```

```

intptr_t ppde=(intptr_t)&pdir[addr >> 22];
pde.val=paddr_read(ppde, sizeof(PDE));
assert(pde.present);
pde.accessed=true;

/* 访问页表 */
PTE *ptab=(PTE*)(pde.page_frame << 12);
PTE pte;
intptr_t ppte=(intptr_t)&ptab[(addr<<10) >> 12];
pte.val=paddr_read(ppte, sizeof(PTE));
assert(pte.present);
pte.accessed = true;
if(write) pte.dirty=true;

/* 写回页目录、页表 */
paddr_write(ppde, sizeof(PDE), pde.val);
paddr_write(ppte, sizeof(PTE), pte.val);

intptr_t paddr= (pte.page_frame << 12) | (addr & PAGE_MASK);
return paddr;
}

```

2.3 让用户程序运行在分页机制上

2.3.1 dummy

按照要求，实现 `_map` 和 `loader` 函数如下。

Listing 4: `_map` 函数的实现

```

void _map(_Protect *p, void *va, void *pa) {
    PDE *pdir=p->ptr;
    PDE *pde=&pdir[PDX(va)];

    if(!(*pde & PTE_P)){
        PTE *pg=(PTE *)palloc_f();
        for (int i = 0; i < NR_PTE; i++) pg[i] = 0;
        *pde=PTE_ADDR(pg) | PTE_P;
    }

    PTE *ptab= (PTE*)PTE_ADDR(*pde);
    PTE *pte=&ptab[PTX(va)];
    *pte=PTE_ADDR(pa) | PTE_P;
}

```

```
}
```

Listing 5: loader 函数的实现

```
uintptr_t loader(_Protect *as, const char *filename) {
    int fd = fs_open(filename,0,0);
    size_t len=fs_filesz(fd);
    void *va = DEFAULT_ENTRY;

    while(len>0){
        void *pa =new_page();
        _map(as,va,pa);
        int read_len= len > PGSIZE ? PGSIZE:len;
        read_len=fs_read(fd,pa,read_len);
        va += read_len;
        len -= read_len;
    }

    fs_close(fd);
    return (uintptr_t)DEFAULT_ENTRY;
}
```

最后可成功运行 dummy 程序，如图1所示。

```
(nemu) c
[src/mm.c,24,init_mm] free physical pages starting from 0x1d91000
[src/main.c,20,main] 'Hello World!' from Nanos-lite
[src/main.c,21,main] Build time: 15:59:00, May 18 2022
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x1028e0,
end = 0x1d4c865, size = 29663109 bytes
[src/main.c,28,main] Initializing interrupt/exception handler..
.
[src/main.c,34,main] start to load img.
nemu: HIT GOOD TRAP at eip = 0x00100032
```

图 1: dummy 在在分页机制中运行

2.3.2 内核映射的作用

Listing 6: 需要解释的代码

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

这处代码若注释掉会触发页不存在的断言，从命名来看 `updir` 表示用户态页目录地址，`kpdirs` 表示内核态页目录地址，但是 `nemu` 中无特权级转换，故二者相同，直接复制，在 `load` 之后 `cr3` 的值会设为 `updir`。

2.3.3 仙剑奇侠传

运行仙剑奇侠传之前，需要先实现 `mm_brk` 函数，注意按页对齐。

Listing 7: `mm_brk` 的实现

```
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) {
            // DONE: map memory region [current->max_brk, new_brk)
            // into address space current->as
            // 按页对齐
            uintptr_t startva = (current->max_brk + 0xfff) & ~0xfff;
            for (uintptr_t va = startva; va < new_brk; va += PGSIZE) {
                _map(&current->as, (void *)va, new_page());
            }

            current->max_brk = new_brk;
        }

        current->cur_brk = new_brk;
    }

    return 0;
}
```

在运行仙剑奇侠传的时候，之前跨页读取的断言被触发了，因此需要实现，如下。

Listing 8: 跨页读取的实现

```
if (cross_page(addr, len)) {
    /* this is a special case, you can handle it later. */
    union {
        uint32_t val;
        uint8_t bytes[4];
    } temp = {0};
    for (int i = 0; i < len; i++) {
        paddr_t paddr = page_translate(addr + i, false);
```

```
temp.bytes[i] = (uint8_t)paddr_read(paddr, 1);  
}  
return temp.val;  
}
```

最终成功在分页机制上运行仙剑奇侠传，如图2所示。

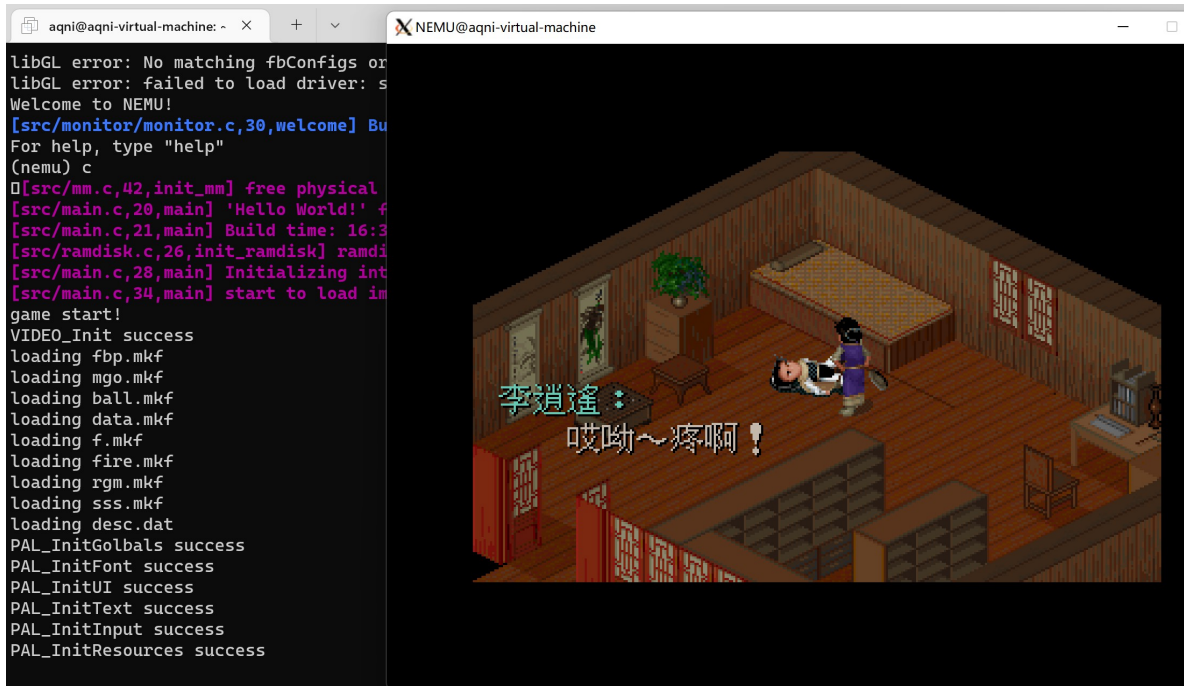


图 2: 仙剑奇侠传在在分页机制中运行

3 阶段二

3.1 实现内核自陷

这一步，需要实现 `_trap()` 函数定义，并且修改 `irq_handle()` 函数将内核自陷并包装成 `_EVENT_TRAP` 事件，此外还需要添加 `idt` 表项及相应处理函数相应代码如下，并且在在 `do_event` 中输出一句话验证正确性。

Listing 9: 实现内核自陷

```
void _trap() {  
    asm volatile("int $0x81");  
}  
  
// 在 irq_handle 的 switch 语句中添加一段  
case 0x81: ev.event = _EVENT_TRAP; break;
```

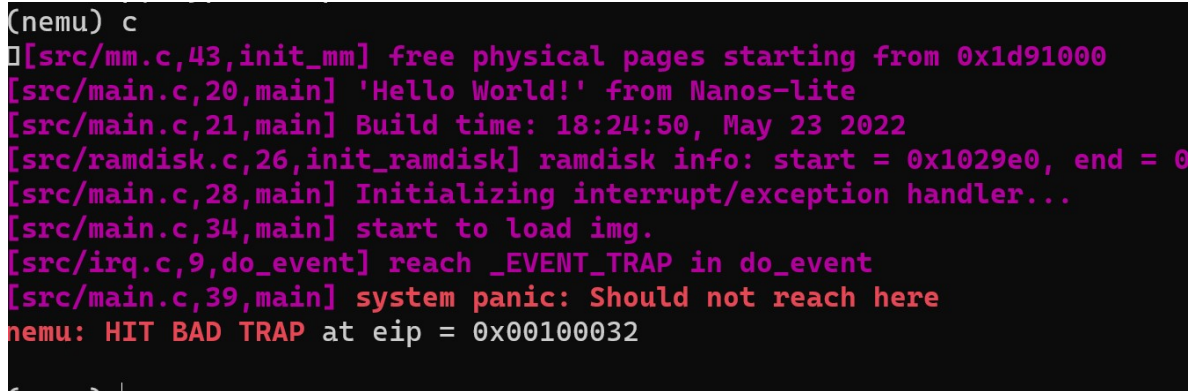
```
// 在_asye_init总添加idt表项
idt[0x81] = GATE(STS_IG32, KSEL(SEG_KCODE), vectrap, DPL_USER);

// 在trap.s中定义处理过程
.globl vectrap; vectrap: pushl $0; pushl $0x81; jmp asm_trap

// 相应的声明

void vectrap();
// 在do_event中输出一句话
case _EVENT_TRAP:
    Log("reach _EVENT_TRAP in do_event");
    return NULL;
```

最终成功触发自陷，如图3所示。



```
(nemu) c
[src/mm.c,43,init_mm] free physical pages starting from 0x1d91000
[src/main.c,20,main] 'Hello World!' from Nanos-lite
[src/main.c,21,main] Build time: 18:24:50, May 23 2022
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x1029e0, end = 0
[src/main.c,28,main] Initializing interrupt/exception handler...
[src/main.c,34,main] start to load img.
[src/irq.c,9,do_event] reach _EVENT_TRAP in do_event
[src/main.c,39,main] system panic: Should not reach here
nemu: HIT BAD TRAP at eip = 0x00100032
```

图 3: 成功触发自陷

3.2 实现上下文切换

首先按要求实现 _umake 函数，相应代码如下。

Listing 10: 实现 _umake 函数

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

    // stack frame of _start() 设置三个空参数，忽略返回值
    uint32_t *stack = (uint32_t *) (ustack.end - 4 - 12);
    for (int i = 0; i < 3; i++) stack[i] = 0;

    // 设置tf
    tf = (void *) (stack - sizeof(_RegSet));
```



```
tf->eflags = 0x2;
tf->cs = 8;
tf->eip = (uintptr_t)entry; // 为了保证 differential testing 的正确运行,

return tf;
}
```

然后按要求实现调度器。

Listing 11: 实现 schedule()

```
_RegSet* schedule(_RegSet *prev) {
    // save the context pointer
    current->tf = prev;

    // always select pcb[0] as the new process
    current = &pcb[0];

    // TODO: switch to the new address space,
    // then return the new context
    _switch(&current->as);
    return current->tf;
}
```

然后在 Nanos-lite 收到 `_EVENT_TRAP` 事件后, 调用 `schedule()` 并返回其现场。并修改 `asm_trap()` 切换 `trapframe` 指针, 代码如下。

Listing 12: 改 `asm_trap()` 切换 `trapframe` 指针

```
asm_trap:
    pushal

    pushl %esp
    call irq_handle

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

    popal
    addl $8, %esp

    iret
```

3.3 分时多任务

修改 `schedule()`, 并在处理完系统调用之后, 调用 `schedule()` 函数并返回其现场, 最后可轮流运行仙剑奇侠传和 `hello`, 如图4。

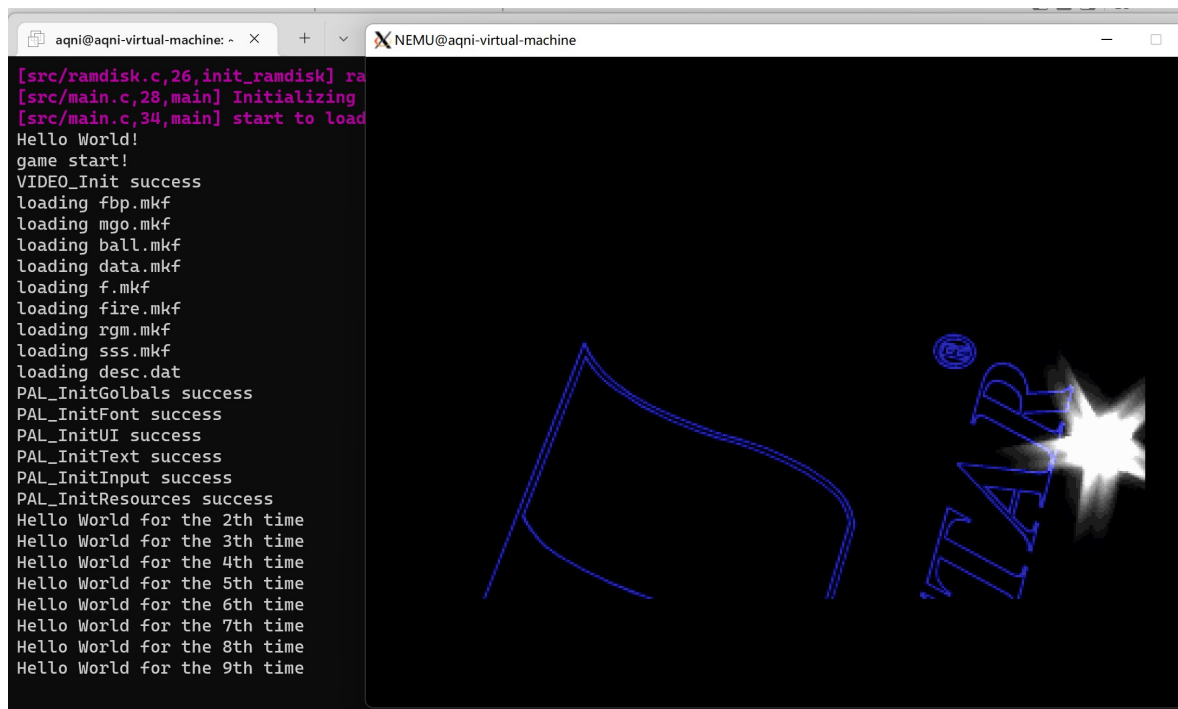


图 4: 成功调度

4 阶段三

4.1 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? 如果你觉得毫无头绪, 你可以用纸笔模拟中断处理的过程。

会陷入到无休止地中断返回过程中, 因为总是恢复相同的上下文。

4.2 添加时钟中断

安装指导书一步一步做, 同时, 在 `idt` 中添加对应的表项与处理函数。最终可成功实现时间中断, 如图5。

```

game start!
VIDEO_Init success
loading fbp.mkf
loading mgo.mkf
loading[src/irq.c,12,do_event] reach time irq
ball.mkf
loading data.mkf
loading f.mkf
loading fire.mkf
loading rgm.mkf
loading sss.mkf
loading desc.dat
[src/irq.c,12,do_event] reach time irq
[src/irq.c,12,do_event] reach time irq
[src/irq.c,12,do_event] reach time irq
[src/irq.c,12,do_event] reach time irq

```

图 5: 时间中断

4.3 展示你的计算机系统

在加载仙剑奇侠传和 hello 后，再加载一个/bin/videotest 程序。通过 current_game 维护当前游戏，并通过按键 F12 进行修改，代码如下。

Listing 13: 切换游戏

```

PCB *current_game=&pcb[0];
void change_game(){
    current_game=(current_game == &pcb[2] ? &pcb[0] : &pcb[2]);
}

// 在 schedule 中
current = (count++%100 == 0 ? &pcb[1] : current_game);

// 在 _read_key() 中
extern void change_game();
if (keyinput == _KEY_F12)
    change_game();

```

最后可成功通过 F12 进行游戏切换，如图??所示。



图 6: 切换游戏

5 bug 汇总

5.1 移位

在截取页表项偏移时，希望截断高 10 位和低 12 位，保留中间 10 位，一开始是这样写的，少偏移了 10 位。

```
intptr_t ppte=(intptr_t)&ptab[(addr<<10) >> 12];  
//修改后  
intptr_t ppte=(intptr_t)&ptab[(addr<<10) >> 22];
```

同时，做实验的时候网络将 navy-apps/Makefile.compile 中的链接地址-Ttext 参数改为 0x8048000，排查许多时间才发现问题。

5.2 mm_brk

在实现了 mm_brk 后，发现程序没有正确执行，原来时在上次实验的时候系统调用没有调用 mm_brk，如下所示。

```
case SYS_brk:SYSCALL_RET(r)=0; break;
```

此外，mm_brk 中进行映射时注意按页对齐。

6 必答题

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.

分页机制提供内存空间的隔离，具体是通过 nemu 中的 vaddr_read/vaddr_write 作为接口实现虚拟内存地址寻址。分页机制在 am 中对多级页表进行管理，通过 _map() 函数建立映射。分时运行依赖上下文的保存与切换，具体是通过中断的上下文恢复实现，通过 asm_trap 实现上下文保存与恢复，通过 schedule 实现上下文 trapframe 的切换。

7 体会与感悟

参考文献

- [1] YU Z. Ics2017 programming assignment[EB/OL]. [March 22, 2022]. <https://nju-ics.gitbooks.io/ics2017-programming-assignment/>.