
系统综合课程设计-实验报告

PA4-虚实交错的魔法：分时多任务

姓名：孙铭

学号：1711377

学院：计算机学院

专业：计算机科学与技术

时间：2020年6月8日

系统综合课程设计-实验报告

PA4-虚实交错的魔法：分时多任务

PA4-阶段一

- 1.1 代码：在NEMU中实现分页机制
- 1.2 代码：让用户程序运行在分页机制上
- 1.3 问题：内核映射的作用
- 1.4 代码：在分页机制上运行仙剑奇侠传

PA4-阶段二

- 2.1 代码：实现内核自陷
- 2.2 代码：实现上下文切换
- 2.3 代码：分时运行仙剑奇侠传和hello程序
- 2.4 代码：优先级调度

PA4-阶段三

- 3.1 代码：添加时钟中断
- 3.2 必答题

编写不朽的传奇

PA4-阶段一

第一阶段的主要内容是了解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。接下来将展开叙述。

1.1 代码：在NEMU中实现分页机制

在 NEMU 中实现分页机制

根据上述的讲义内容，在 NEMU 中实现 i386 分页机制，如有疑问，请查阅 i386 手册。

首先，在 `nanos-lite/src/main.c` 中定义宏 `HAS_PTE` 如下。

```
1 #include "common.h"
2
3 /* Uncomment these macros to enable corresponding functionality. */
4 #define HAS_ASYE
5 #define HAS_PTE
```

在 `nanos-lite` 下执行 `make run`，报错如下。

```
Welcome to NEMU!
[src/monitor/monitor.c,31,welcome] Build time: 16:56:42, Jun 10 2020
For help, type "help"
(nemu) c
[src/mm.c,24,init_mm] free physical pages starting from 0x1d70000
invalid opcode(eip = 0x001015eb): 0f 22 d8 0f 20 c0 89 45 ...
There are two cases which will trigger this unexpected exception:
1. The instruction at eip = 0x001015eb is not implemented.
2. Something is implemented incorrectly.
Find this eip(0x001015eb) in the disassembling result to distinguish which case
it is.
```

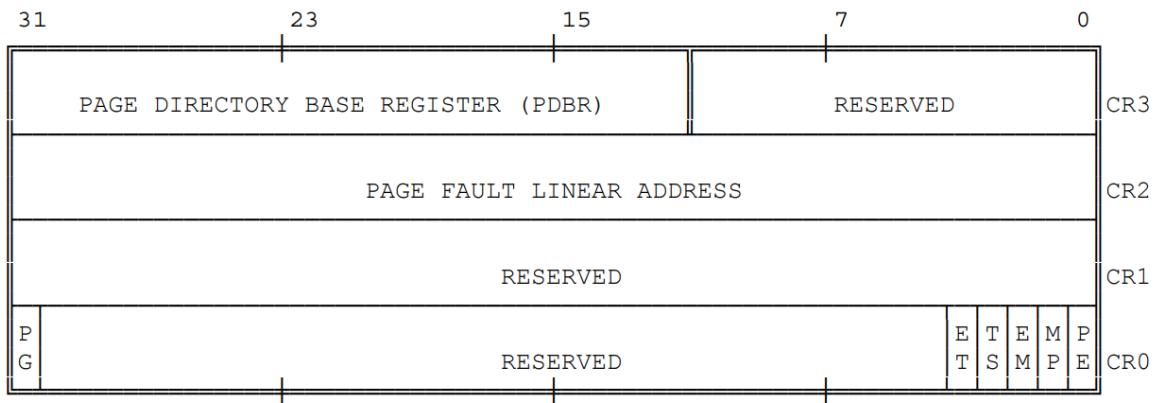
在 `nanos-lite/build` 目录下输入命令 `objdump -S nanos-lite-x86-nemu` 查看反汇编代码，找到 `eip=0x001015eb` 位置的指令，为 `mov` 指令。

```
static inline void set_cr3(void *pdир) {
    asm volatile("movl %0, %%cr3" : : "r"(pdир));
1015e6:        b8 00 e0 d6 01          mov    $0x1d6e000,%eax
1015eb:        0f 22 d8          mov    %eax,%cr3
```

控制寄存器CR0、CR3的实现

首先需要实现控制寄存器CR0、CR3。参考i386手册，控制寄存器结构如下。

Figure 4-2. Control Registers



因此在 `nemu/include/cpu/reg.h` 中添加控制寄存器CR0与CR3实现代码如下。

```
58 // 控制寄存器CR0、CR3
59     uint32_t CR0;
60     uint32_t CR3;
```

在 `nemu/src/monitor/monitor.c` 中对restart函数初始化如下。

```
83 static inline void restart() {
84     /* Set the initial instruction pointer. */
85     cpu.eip = ENTRY_START;
86     // 进行eflags的初始化
87     unsigned int origin = 2;
88     memcpy(&cpu.eflags, &origin, sizeof(cpu.eflags));
89     // 对cs寄存器初始化
90     cpu.cs = 8;
91     // 控制寄存器初始化
92     cpu.CR0 = 0x60000011;
93
94 #ifdef DIFF_TEST
95     init_qemu_reg();
96 #endif
97 }
```

mov指令实现

查阅i386手册，手册中关于mov指令的内容如下。

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)

首先利用rtl指令实现对CR3与CR0的访问与修改的操作，在 `nemu/include/cpu/rtl.h` 中添加如下代码。

```

226 static inline void rtl_load_cr(rtlreg_t* dest, int r){
227     switch(r){
228         case 0: *dest = cpu.CR0; return;
229         case 3: *dest = cpu.CR3; return;
230         default: assert(0);
231     }
232     return;
233 }
234
235 static inline void rtl_store_cr(int r, const rtlreg_t* src){
236     switch(r){
237         case 0: cpu.CR0 = *src; return;
238         case 3: cpu.CR3 = *src; return;
239         default: assert(0);
240     }
241     return;
242 }
```

在 nemu/include/cpu/decode.h 中声明编码函数。

```

117 make_DHelper(mov_load_cr);
118 make_DHelper(mov_store_cr);
```

在 nemu/src/cpu/decode/decode.c 中实现编码函数如下。

```

316 make_DHelper(mov_load_cr){
317     decode_op_rm(eip, id_dest, false, id_src, false);
318     rtl_load_cr(&id_src->val, id_src->reg);
319 #ifdef DEBUG
320     sprintf(id_src->str, 5, "%%cr%d", id_dest->reg);
321 }
322
323 make_DHelper(mov_store_cr){
324     decode_op_rm(eip, id_src, true, id_dest, false);
325 #ifdef DEBUG
326     sprintf(id_dest->str, 5, "%%cr%d", id_dest->reg);
327 }
```

在 nemu/src/cpu/exec/all-instr.h 中声明执行函数如下。

```

62 make_EHelper(mov_store_cr);
```

在 nemu/src/cpu/exec/data-mov.c 中实现执行函数如下。

```

112 make_EHelper(mov_store_cr){
113     rtl_store_cr(id_dest->reg, &id_src->val);
114     print_asm_template2(mov);
115 }
```

在 nemu/src/cpu/exec/exec.c 中补充 opcode 如下 (2 bytes)。

```

150 /* 0x20 */    IDEX(mov_load_cr, mov), EMPTY, IDEX(mov_store_cr, mov_store_cr), EMPTY,
```

参考老师给的实验指导，为了便于查看控制寄存器 CR 中的内容，在简易调试器的 info r 中增加查看 CR0 与 CR3 寄存器的代码如下，代码位于 nemu/src/monitor/debug/ui.c 目录下。

```

125     if(s == 'r'){
126         int i;
127         //32位寄存器
128         for(i = 0; i < 8; i++)
129             printf("%s    0x%08x\n", regsl[i], reg_l(i)); //regsl[i]和reg_l(i)定义见
130             reg.h和reg.c
131             printf("eip    0x%08x\n", cpu.eip);
132             //16位寄存器
133             for(i = 0; i < 8; i++)
134                 printf("%s    0x%04x\n", regsw[i], reg_w(i));
135             //8位寄存器
136             for(i = 0; i < 8; i++)
137                 printf("%s    0x%02x\n", regsb[i], reg_b(i));
138             printf("CR0=0x%08x, CR3=0x%08x\n", cpu.CR0, cpu.CR3);
139             return 0;
140     }

```

按照之前反汇编代码中CR3完成设置的位置设置相应监视点信息如下。

```

static inline void set_cr3(void *pdcr) {
asm volatile("movl %0, %%cr3" : : "r"(pdcr));
1015e6:    b8 00 e0 d6 01          mov    $0x1d6e000,%eax
1015eb:    0f 22 d8              mov    %eax,%cr3
{ (uint32_t)(entry) & 0xffff, (cs), 0, 0, (type), 0, (dpl), \
1, (uint32_t)(entry) >> 16 }

```

在 nanos-lite 下执行 `make run`，并设置监视点，结果如下。

```
(nemu) w $eip==0x1015eb
Success: set watchpoint 1, oldValue=0
```

输入 `info r` 查看此时寄存器中的内容如下。

```
CR0=0x60000011, CR3=0x0
```

输入 `c` 运行至断点处如下。

```
(nemu) w $eip==0x1015eb
Success: set watchpoint 0, oldValue=0
(nemu) c
[0]src/mm.c,24,init_mm] free physical pages starting from 0x1d70000
Hardware watchpoint 0:$eip==0x1015eb
Old Value:0
New value:1
```

输入 `si` 单步执行代码如下。可以看到在 `$eip==0x1015ee` 处完成了CR0与CR3的设置。

```
(nemu) si
1015ee: 0f 20 c0          movl %cr0,%eax
(nemu) si
1015f1: 89 45 f4          movl %eax,-0xc(%ebp)
```

再次利用 `info r` 查看寄存器中的内容如下。

```
CR0=0xe0000011, CR3=0x1d6e000
```

可以看到CR0寄存器最高位被置为1，CR3寄存器存储页目录表地址。

虚拟地址转换

根据实验手册，想要让用户程序运行在分页机制上，首先需要实现虚拟地址的转换。查阅i386手册，手册中关于虚拟地址到物理地址的转换方式如下。

Figure 5-8. Format of a Linear Address

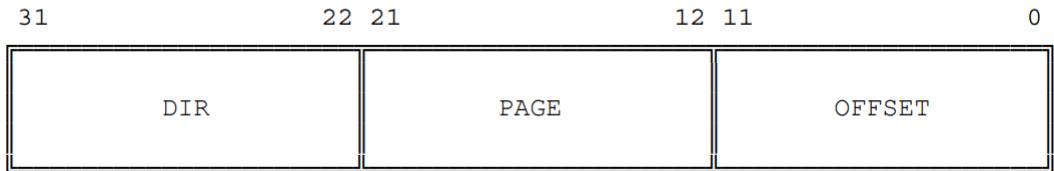
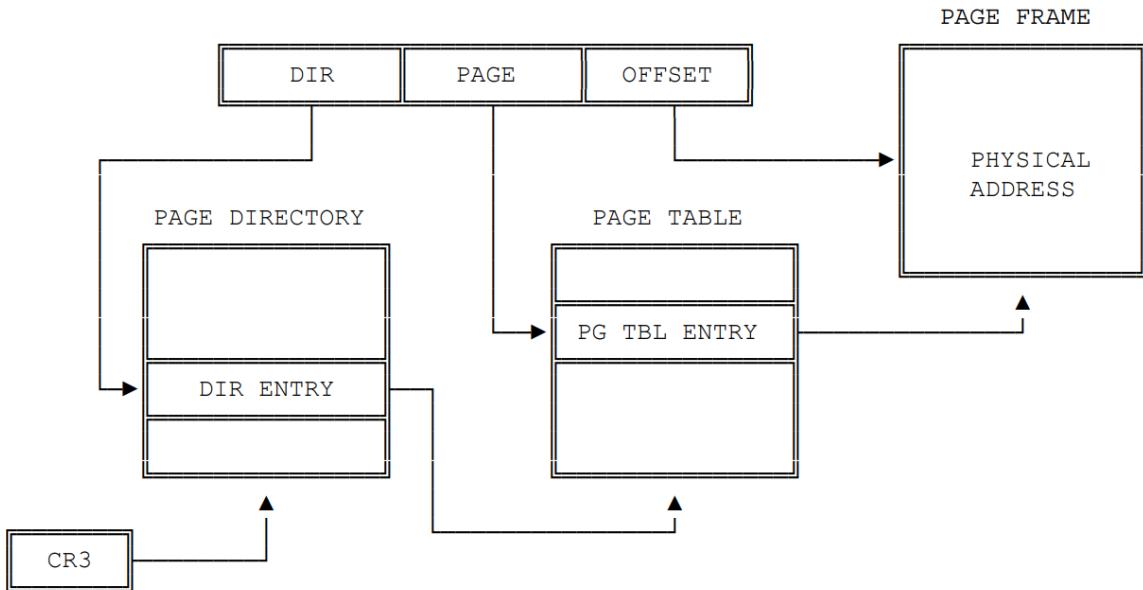


Figure 5-9. Page Translation



首先，在 `nemu/src/memory/memory.c` 中添加头文件的引用如下。

```

1 #include "nemu.h"
2 #include "device/mmio.h"
3 #include "memory/mmu.h"

```

仿照x86.h定义的辅助宏，在该文件中添加代码如下。

```

12 #define PTE_ADDR(pte) (((uint32_t)(pte)) & ~0xffff)
13 #define PDX(va) (((uint32_t)(va) >> 22) & 0x3ff)
14 #define PTX(va) (((uint32_t)(va) >> 12) & 0x3ff)
15 #define OFF(va) ((uint32_t)(va) >> 0xffff)

```

接下来需要在 `page_translate` 中增加参数 `iswrite` 来判断读或写操作，并据此修改对应的 `Accessed` 与 `Dirty` 位。

```

39 paddr_t page_translate(vaddr_t addr, bool iswrite){
40     CR0 cr0 = (CR0)cpu.CR0;
41     if(cr0.paging && cr0.protect_enable){
42         CR3 cr3 = (CR3)cpu.CR3;
43
44         PDE* pgdirs = (PDE*)PTE_ADDR(cr3.val);
45         PDE pde = (PDE)paddr_read((uint32_t)(pgdirs + PDX(addr)), 4);
46         Assert(pde.present, "addr=0x%x", addr);
47
48         PTE* ptab = (PTE*)PTE_ADDR(pde.val);
49         PTE pte = (PTE)paddr_read((uint32_t)(ptab + PTX(addr)), 4);
50         Assert(pte.present, "addr=0x%x", addr);
51
52         pde.accessed = 1;
53         pte.accessed = 1;
54         if(iswrite)
55             pte.dirty = 1;
56         paddr_t paddr = PTE_ADDR(pte.val) | OFF(addr);
57         return paddr;
58     }
59     return addr;
60 }
```

参照实验手册，修改 `vaddr_read` 与 `vaddr_write` 如下。

```

62 uint32_t vaddr_read(vaddr_t addr, int len) {
63     if(PTE_ADDR(addr) != PTE_ADDR(addr + len - 1)){
64         printf("error:the data pass two pages:addr=0x%x,len=%d!\n", addr, len);
65         assert(0);
66     }
67     else{
68         paddr_t paddr = page_translate(addr, false);
69         return paddr_read(paddr, len);
70     }
71 }
72 void vaddr_write(vaddr_t addr, int len, uint32_t data) {
73     if(PTE_ADDR(addr) != PTE_ADDR(addr + len - 1)){
74         printf("error:the data pass two pages:addr=0x%x,len=%d!\n", addr, len);
75         assert(0);
76     }
77     else{
78         paddr_t paddr = page_translate(addr, true);
79         paddr_write(paddr, len, data);
80     }
81 }
```

运行dummy，出现数据跨越虚拟页内存的情况。

```

(nemu) c
[0] [src/mm.c,24,init_mm] free physical pages starting from 0x1d70000
[src/main.c,19,main] 'Hello World!' from Nanos-lite
[src/main.c,20,main] Build time: 10:26:56, Jun 12 2020
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x102568, end = 0x1d4c4c9,
size = 29663073 bytes
[src/main.c,27,main] Initializing interrupt/exception handler...
[src/cpu/exec/system.c,18,exec_lidt] idtr.limit=0x7ff
[src/cpu/exec/system.c,19,exec_lidt] idtr.base=0x1d6f020
[src/fs.c,31,init_fs] set FD_FB size=480000
[src/loader.c,17,loader] filename=/bin/dummy, fd=50
error:the data pass two pages:addr=0x1ca4ffd,len=4!
nemu: src/memory/memory.c:65: 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/sun/Desktop/ics2018/nemu'
/home/sun/Desktop/ics2018/nexus-am/Makefile.app:35: recipe for target 'run' failed
make: *** [run] Error 2
```

为解决上述问题，修改 `vaddr_read` 和 `vaddr_write` 两个函数如下。

```

62 uint32_t vaddr_read(vaddr_t addr, int len) {
63     if(PTE_ADDR(addr) != PTE_ADDR(addr + len - 1)){
64         //printf("error:the data pass two pages:addr=0x%x,len=%d!\n",
65         len);
66         //assert(0);
67         int num1 = 0x1000 - OFF(addr);
68         int num2 = len - num1;
69         paddr_t paddr1 = page_translate(addr, false);
70         paddr_t paddr2 = page_translate(addr + num1, false);
71         uint32_t low = paddr_read(paddr1, num1);
72         uint32_t high = paddr_read(paddr2, num2);
73         uint32_t result = high << (num1 * 8) | low;
74         return result;
75     }
76     else{
77         paddr_t paddr = page_translate(addr, false);
78         return paddr_read(paddr, len);
79     }

```

```

81 void vaddr_write(vaddr_t addr, int len, uint32_t data) {
82     if(PTE_ADDR(addr) != PTE_ADDR(addr + len - 1)){
83         //printf("eror:the data pass two pages:addr=0x%x,len=%d!\n", addr, l
84         en);
85         //assert(0);
86         int num1 = 0x1000 - OFF(addr);
87         int num2 = len - num1;
88         paddr_t paddr1 = page_translate(addr, true);
89         paddr_t paddr2 = page_translate(addr + num1, true);
90         uint32_t low = data & (~0u >> ((4 - num1) << 3));
91         uint32_t high = data >> ((4 - num2) << 3);
92         paddr_write(paddr1, num1, low);
93         paddr_write(paddr2, num2, high);
94         return;
95     }
96     else{
97         paddr_t paddr = page_translate(addr, true);
98         paddr_write(paddr, len, data);
99     }
100 }

```

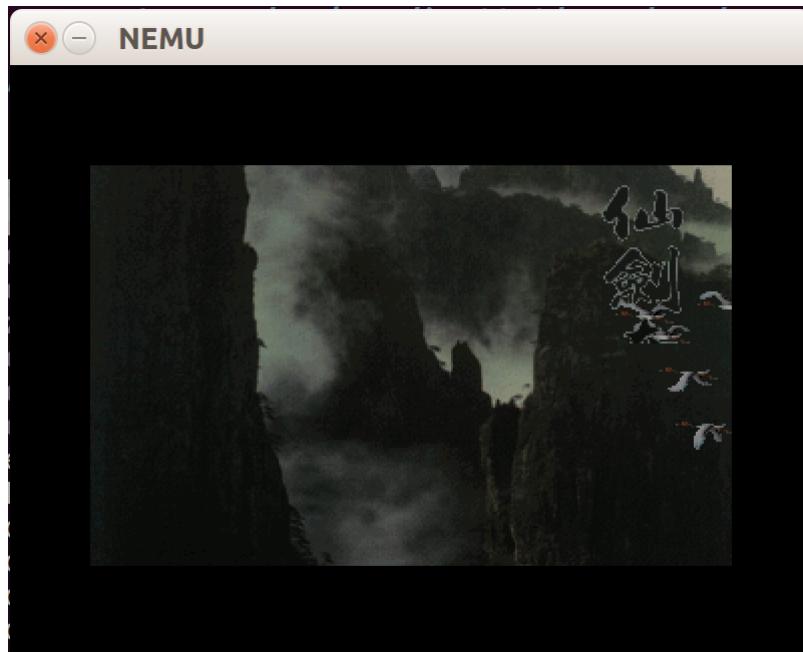
再次运行dummy，运行成功，显示 HIT GOOD TRAP。

```

(nemu) c
[0]src/mm.c,24,init_mm] free physical pages starting from 0x1d70000
[src/main.c,19,main] 'Hello World!' from Nanos-lite
[src/main.c,20,main] Build time: 10:26:56, Jun 12 2020
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x102568, end = 0x1d4c4c9,
size = 29663073 bytes
[src/main.c,27,main] Initializing interrupt/exception handler...
[src/cpu/exec/system.c,18,exec_lidt] idtr.limit=0x7ff
[src/cpu/exec/system.c,19,exec_lidt] idtr.base=0x1d6f020
[src/fs.c,31,init_fs] set FD_FB size=480000
[src/loader.c,17,loader] filename=/bin/dummy, fd=50
nemu: HIT GOOD TRAP at eip = 0x00100032

```

修改main.c函数，运行仙剑奇侠传，运行成功，结果如下。



1.2 代码：让用户程序运行在分页机制上

让用户程序运行在分页机制上

根据上述的讲义内容，在PTE中实现`_map()`，然后修改`loader()`的内容，通过`_map()`在用户程序的虚拟地址空间中创建虚拟页，并把用户程序加载到虚拟地址空间上。

实现正确后，你会看到`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。

为了让用户程序运行在操作系统为其分配的虚拟地址空间之上，需要对工程做一些变动，首先，将`navy-apps/Makefile.compile`中的链接地址`-Ttext`参数改为`0x8048000`，这是为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。

```
21 ifeq ($LINK, dynamic)
22   CFLAGS += -fPIE
23   CXXFLAGS += -fPIE
24   LDFLAGS += -fpie -shared
25 else
26   LDFLAGS += -Ttext 0x8048000
27 endif
```

接下来对`nanos-lite/src/loader.c`中的`DEFAULT_ENTRY`作相应修改如下。

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

之后，让Nanos-lite通过在`nanos-lite/src/proc.c`中定义的`load_prog()`函数，进行用户程序的加载，修改`main.c`中的代码如下。

```
33 //uint32_t entry = loader(NULL, "/bin/pal");
34 //((void (*)(void))entry)();
35
36 load_prog("/bin/dummy");
37 panic("Should not reach here");
```

在使得用户程序的代码在`0x8048000`附近开始之后，运行`dummy`如下。

```
(nemu) c
[0]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: 11:17:22, Jun 12 2020
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x102648, end = 0x1d4c5a9,
size = 29663073 bytes
[src/main.c,28,main] Initializing interrupt/exception handler...
[src/fs.c,31,init_fs] set FD_FB size=480000
[src/loader.c,17,loader] filename=/bin/dummy, fd=50
addr=0x8048000
nemu: src/memory/memory.c:46: 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/sun/Desktop/ics2018/nemu'
/home/sun/Desktop/ics2018/nexus-am/Makefile.app:35: recipe for target 'run' failed
make: *** [run] Error 2
```

在 `nexus-am/arch/x86-nemu/src/pte.c` 中添加如下代码。

```
68 void _map(_Protect *p, void *va, void *pa) {
69     if(OFF(va) || OFF(pa)){
70         //return;
71     }
72     PDE *pgdir = (PDE*)p->ptr;
73     PTE *pgtab = NULL;
74
75     PDE *pde = pgdir + PDX(va);
76     if(!(*pde & PTE_P)){
77         pgtab = (PTE*)(palloc_f());
78         *pde = (uintptr_t)pgtab | PTE_P;
79     }
80     pgtab = (PTE*)PTE_ADDR(*pde);
81
82     PTE *pte = pgtab + PTX(va);
83     *pte = (uintptr_t)pa | PTE_P;
84 }
```

在 `nanos-lite/src/loader.c` 中添加代码如下。

```
13 intptr_t loader(_Protect *as, const char *filename) {
14     // TODO();
15     //ramdisk_read(DEFAULT_ENTRY, 0, RAMDISK_SIZE);
16
17     int fd = fs_open(filename, 0, 0);
18     Log("filename=%s, fd=%d", filename, fd);
19
20     int size = fs_filesz(fd);
21     int ppnum = size / PGSIZE;
22     if(size % PGSIZE != 0)
23         ppnum++;
24     void *pa = NULL;
25     void *va = DEFAULT_ENTRY;
26     for(int i = 0; i < ppnum; i++){
27         pa = new_page();
28         _map(as, va, pa);
29         fs_read(fd, pa, PGSIZE);
30         va += PGSIZE;
31     }
32
33     fs_close(fd);
34     return (uintptr_t)DEFAULT_ENTRY;
35 }
```

在 `nanos-lite` 下输入 `make run` 运行代码，结果如下。

```
(nemu) c
[0] [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: 11:17:22, Jun 12 2020
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x1026e8, end = 0x1d4
size = 29663073 bytes
[src/main.c,28,main] Initializing interrupt/exception handler...
[src/fs.c,31,init_fs] set FD_FB size=480000
[src/loader.c,18,loader] filename=/bin/dummy, fd=50
nemu: HIT GOOD TRAP at eip = 0x00100032
```

1.3 问题：内核映射的作用

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。

参照实验手册，在 `nexus-am/am/arch/x86-nemu/src/pte.c` 中注释掉题目中要求的代码如下。

```
void _protect(_Protect *p) {
    PDE *updir = (PDE*)(palloc_f());
    p->ptr = updir;
    // map kernel space
    //for (int i = 0; i < NR_PDE; i++) {
    //    updir[i] = kpdirs[i];
    //}

    p->area.start = (void*)0x80000000;
    p->area.end = (void*)0xc0000000;
}
```

运行代码，结果如下。

```
(nemu) c
[0] [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: 11:17:22, Jun 12 2020
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x1026e8, end = 0x1d4c649,
size = 29663073 bytes
[src/main.c,28,main] Initializing interrupt/exception handler...
[src/fs.c,31,init_fs] set FD_FB size=480000
[src/loader.c,18,loader] filename=/bin/dummy, fd=50
addr=0x101720
nemu: src/memory/memory.c:46: 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/sun/Desktop/ics2018/nemu'
/home/sun/Desktop/ics2018/nexus-am/Makefile.app:35: recipe for target 'run' failed
make: *** [run] Error 2
```

出现这种错误的原因是，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。

`_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，虚拟地址根据用户进程的页目录表来转换。注释掉 `_Protect()` 函数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。

1.4 代码：在分页机制上运行仙剑奇侠传

在分页机制上运行仙剑奇侠传

之前我们让 `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：

```
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) {
            // TODO: map memory region [current->max_brk, new_brk)
            // into address space current->as
        }

        current->max_brk = new_brk;
    }

    return 0;
}
```

你需要填充上述 TODO 处的代码，其中 `current` 是一个特殊的指针，我们在后面介绍它。你需要注意 `_map()` 参数是否需要按页对齐的问题（这取决于你的 `_map()` 实现）。为了简化，我们也不实现堆区的回收功能了。实现正确后，仙剑奇侠传就可以正确在分页机制上运行了。

在 `nanos-lite/src/mm.c` 中对 `mm_brk` 函数进行修改如下。

```
17 /* The brk() system call handler. */
18 int mm_brk(uint32_t new_brk) {
19     if(current->cur_brk == 0){
20         current->cur_brk = current->max_brk = new_brk;
21     }
22     else{
23         if(new_brk > current->max_brk){
24             uint32_t first = PGROUNDUP(current->max_brk);
25             uint32_t end = PGROUNDDOWN(new_brk);
26             if((new_brk & 0xffff) == 0){
27                 end -= PGSIZE;
28             }
29             for(uint32_t va = first; va <= end; va+=PGSIZE){
30                 void* pa = new_page();
31                 _map(&(current->as), (void*)va, pa);
32             }
33             current->max_brk = new_brk;
34         }
35         current->cur_brk = new_brk;
36     }
37     return 0;
38 }
```

在 `nanos-lite/src/syscall.c` 中添加代码如下。

```
30 int sys_brk(int addr){
31     extern int mm_brk(uint32_t new_brk);
32     return mm_brk(addr);
33 }
```

最后修改 `nanos-lite/src/main.c` 中的代码，在分页机制上运行仙剑奇侠传，成功运行，结果如下。

```
(nemu) c
[0]src/mm.c,42,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: 14:53:12, Jun 12 2020
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x102788, end = 0x1d4c
size = 29663073 bytes
[src/main.c,28,main] Initializing interrupt/exception handler...
[src/fs.c,31,init_fs] set FD_FB size=480000
[src/loader.c,18,loader] filename=/bin/pal, fd=46
game start!
VIDEO_Init success
loading fbp.mkf
loading mgo.mkf
loading ball.mkf
loading data.mkf
loading f.mkf
loading fire.mkf
loading rgm.mkf
loading sss.mkf
loading desc.dat
PAL_InitGolbals success
PAL_InitFont success
PAL_InitUI success
PAL_InitText success
PAL_InitInput success
PAL_InitResources success
```



以上是本次实验第一阶段内容。

PA4-阶段二

第二阶段主要内容是实现内核自陷、上下文切换与分时多任务。

2.1 代码：实现内核自陷

实现内核自陷

修改 Nanos-lite 的如下代码：

```
--- nanos-lite/src/main.c
+++ nanos-lite/src/main.c
@@ -33,3 +33,5 @@
    load_prog("/bin/pal");

+ _trap();
+
    panic("Should not reach here");
--- nanos-lite/src/proc.c
+++ nanos-lite/src/proc.c
@@ -17,4 +17,4 @@
    // TODO: remove the following three lines after you have implemented _umake()
- _switch(&pcb[i].as);
- current = &pcb[i];
- ((void (*)(void))entry)();
+ // _switch(&pcb[i].as);
+ // current = &pcb[i];
+ // ((void (*)(void))entry)();
```

并在 ASYE 添加相应的代码，使得 `irq_handle()` 可以识别内核自陷并包装成 `_EVENT_TRAP` 事件，Nanos-lite 接收到 `_EVENT_TRAP` 之后可以输出一句话，然后直接返回即可，因为真正的上下文切换还需要正确实现 `_umake()` 之后才能实现。实现正确之后，你会看到 Nanos-lite 触发了 `main()` 函数中最后的 `panic`。如果你不知道应该怎么做，请参考你对 PA3 必答题中关于系统调用部分的回答。

在 `nanos-lite/src/main.c` 中启用内核自陷如下。

```

34 //uint32_t entry = loader(NULL, "/bin/pal");
35 //((void (*)(void))entry)();
36
37 //load_prog("/bin/pal");
38 _trap();
39
40 panic("Should not reach here");
41 }

```

在 `nanos-lite/src/proc.c` 中注释掉代码如下。

```

17 // TODO: remove the following three lines after you have implemented _umak
18 e()
19 //switch(&pcb[i].as);
20 //current = &pcb[i];
21 //((void (*)(void))entry)();

```

准备工作完成之后，进入代码实现部分，首先，在 `nexus-am/am/arch/x86-nemu/src/asye.c` 中使`_trap`触发int 0x81指令如下。

```

48 void _trap() {
49     asm volatile("int $0x81");
50 }

```

在 `nexus-am/am/arch/x86-nemu/src/trap.s` 中将该函数压入错误码和异常号irq(0x81)中，并跳转至`asm_trap`中如下。

```

1 #----|-----entry-----|-errorcode-|---irq id---|---handler---|
2 .globl vecsys;    vecsys:  pushl $0;   pushl $0x80;  jmp asm_trap
3 .globl vecnull;   vecnull: pushl $0;   pushl $-1;   jmp asm_trap
4 .globl vecself;   vecself: pushl $0;   pushl $0x81;  jmp asm_trap

```

在 `nexus-am/am/arch/x86-nemu/src/asye.c` 中定义`vecself`函数如下。

```

6 void vecsys();
7 void vecnull();
8 void vecself();

```

填写0x81的门描述符如下。

```

36 // -----
37 idt[0x80] = GATE(STS_TG32, KSEL(SEG_KCODE), vecsys, DPL_USER);
38 idt[0x81] = GATE(STS_TG32, KSEL(SEG_KCODE), vecself, DPL_USER);

```

将0x81异常封装成`_EVENT_TRAP`事件如下。

```

14         switch (tf->irq) {
15             case 0x80: ev.event = _EVENT_SYSCALL; break;
16             case 0x81: ev.event = _EVENT_TRAP; break;
17             default: ev.event = _EVENT_ERROR; break;
18         }

```

在 `nanos-lite/src/irq.c` 中根据事件再次分发代码如下。

```

4 static _RegSet* do_event(_Event e, _RegSet* r) {
5     switch (e.event) {
6         case _EVENT_SYSCALL:
7             return do_syscall(r);
8         case _EVENT_TRAP:
9             printf("event:self-trapped\n");
10            return NULL;
11        default: panic("Unhandled event ID = %d", e.event);
12    }
13
14    return NULL;
15 }

```

运行代码，出现 HIT BAD TRAP，结果如下。

```

(nemu) c
[0]src/mm.c,42,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: 16:05:26, Jun 12 2020
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x102628, end = 0x1d4c589,
size = 29663073 bytes
[src/main.c,28,main] Initializing interrupt/exception handler...
[src/fs.c,31,init_fs] set FD_FB size=480000
event:self-trapped
[src/main.c,40,main] system panic: Should not reach here
nemu: HIT BAD TRAP at eip = 0x00100032

```

2.2 代码：实现上下文切换

实现上下文切换

根据讲义的上述内容，实现以下功能：

- ❖ PTE 的 _umake() 函数
- ❖ Nanos-lite 的 schedule() 函数，Nanos-lite 收到 _EVENT_TRAP 事件后，调用 schedule() 并返回其现场
- ❖ 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

实现成功后，Nanos-lite 就可以通过内核自陷触发上下文切换的方式运行仙剑奇侠传了。

首先，在 nexus-am/am/arch/x86-nemu/src/pte.c 中添加如下代码。

```

89 _RegSet *_umake(_Protect *p, _Area ustack, _Area kstack, void *entry, char *
const argv[], char *const envp[]) {
90     extern void* memcpy(void*, const void*, int);
91     int arg1 = 0;
92     char *arg2 = NULL;
93     memcpy((void*)ustack.end - 4, (void*)arg2, 4);
94     memcpy((void*)ustack.end - 8, (void*)arg2, 4);
95     memcpy((void*)ustack.end - 12, (void*)arg1, 4);
96     memcpy((void*)ustack.end - 16, (void*)arg1, 4);
97
98     _RegSet tf;
99     tf.eflags = 0x02;
100    tf.cs = 8;
101    tf.eip = (uintptr_t)entry;
102    void* ptf = (void*)(ustack.end - 16 - sizeof(_RegSet));
103    memcpy(ptf, (void*)&tf, sizeof(_RegSet));
104    return (_RegSet*)ptf;
105 }

```

在 nanos-lite/src/proc.c 中添加如下代码。

```

29 _RegSet* schedule(_RegSet *prev) {
30     //return NULL;
31     if(current != NULL)
32         current->tf = prev;
33     current = &pcb[0];
34     Log("ptr=0x%x\n", (uint32_t)current->as.ptr);
35     _switch(&current->as);
36     return current->tf;
37 }

```

在 `nanos-lite/src/irq.c` 中添加调用如下。

```

3 extern _RegSet* do_syscall(_RegSet *r);
4 extern _RegSet* schedule(_RegSet *prev);
5 static _RegSet* do_event(_Event e, _RegSet* r) {
6     switch (e.event) {
7         case _EVENT_SYSCALL:
8             return do_syscall(r);
9         case _EVENT_TRAP:
10            printf("event:self-trapped\n");
11            return schedule(r);
12        default: panic("Unhandled event ID = %d", e.event);
13    }
14
15    return NULL;
16 }

```

下一步，在 `nexus-am/am/arch/x86-nemu/src/trap.s` 中修改代码如下。

```

6 asm_trap:
7     pushal
8
9     pushl %esp
10    call irq_handle
11
12    #addl $4, %esp
13    movl %eax, %esp
14
15    popal
16    addl $8, %esp
17
18    iret

```

最后在 `nanos-lite` 下运行仙剑奇侠传代码，运行成功，由于仙剑运行太慢，这里就只放一个初始界面的动画截图。

```

[sun@ests: ~/Desktop/ics2018/nanos-lite]
[src/main.c,28,main] Initializing interrupt/exception handler...
[src/fs.c,31,init_fs] set FD_FB size=480000
[src/loader.c,18,loader] filename=/bin/pal, fd=46
event:self-trapped
[src/proc.c,34,schedule] ptr=0x1d92000

game start!
VIDEO_Init success
loading fbp.mkf
loading mgo.mkf
loading ball.mkf
loading data.mkf
loading f.mkf
loading fire.mkf
loading rgm.mkf
loading sss.mkf
loading desc.dat
PAL_InitGobals success
PAL_InitFont success
PAL_InitUI success
PAL_InitText success
PAL_InitInput success
PAL_InitResources success

```

2.3 代码：分时运行仙剑奇侠传和hello程序

分时运行仙剑奇侠传和 hello 程序

根据讲义的上述内容，添加相应的代码来实现仙剑奇侠传和 hello 程序之间的分时运行。实现正确后，你会看到仙剑奇侠传一边运行的同时，hello 程序也会一边输出。

首先，在 `nanos-lite/src/main.c` 中添加代码如下。

```

37     load_prog("/bin/pal");
38     load_prog("/bin/hello");

```

在 `nanos-lite/src/proc.c` 中修改 `schedule()` 函数如下。

```

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

```

在 `nanos-lite/src/irq.c` 中修改 `do_event()` 函数如下。

```

7         case _EVENT_SYSCALL:
8             do_syscall(r);
9             return schedule(r);

```

最后运行仙剑奇侠传和hello代码，发现hello输出很快，但仙剑奇侠传运行很慢。

2.4 代码：优先级调度

优先级调度

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。

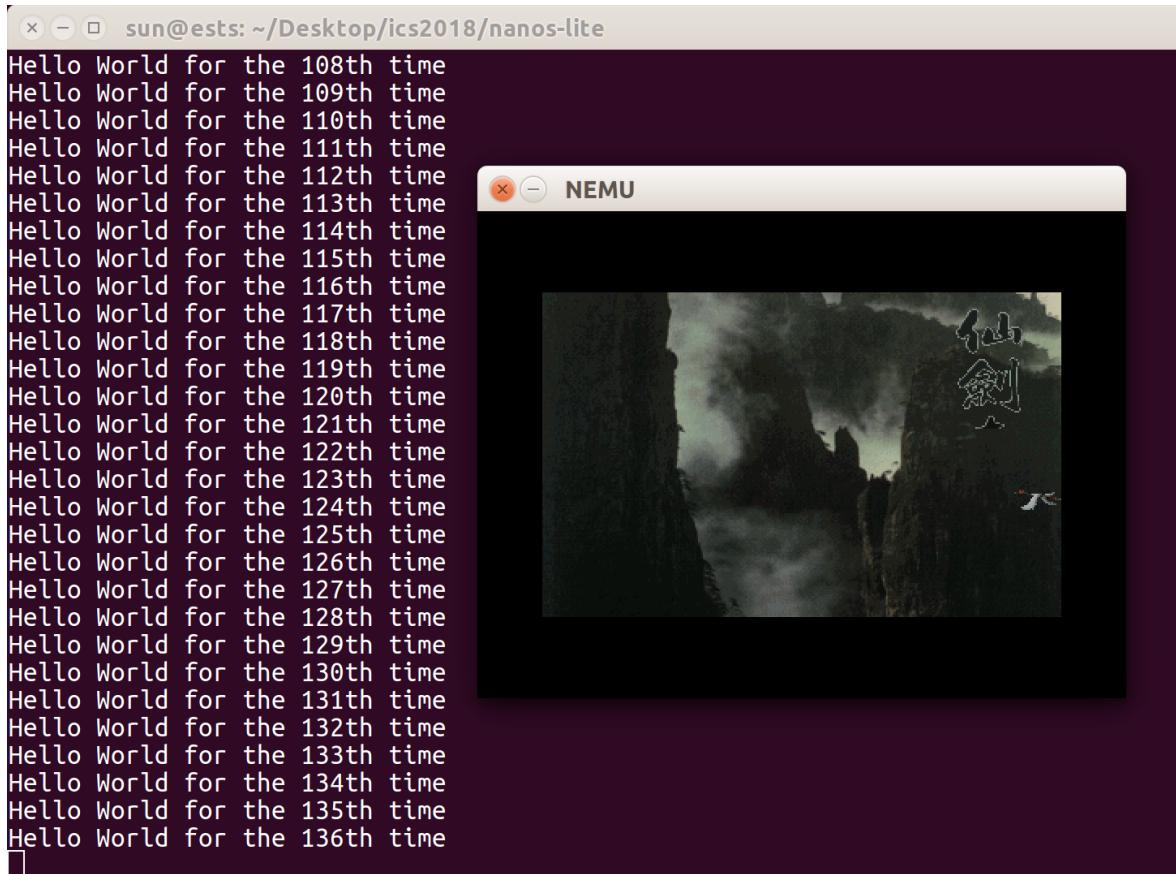
首先，在 `nanos-lite/src/proc.c` 中添加如下代码，用于设置频率比例，即当仙剑奇侠传代码运行指定次数时运行一次hello代码。

```

29 _RegSet* schedule(_RegSet *prev) {
30     //return NULL;
31     if(current != NULL)
32         current->tf = prev;
33     else
34         current = &pcb[0];
35     //current = (current == &pcb[0] ? &pcb[1] : &pcb[0]);
36     //Log("ptr=0x%x\n", (uint32_t)current->as.ptr);
37
38     static int num = 0;
39     static const int frequency = 1000;
40     if(current == &pcb[0])
41         num++;
42     else
43         current = &pcb[0];
44     if(num == frequency){
45         current = &pcb[1];
46         num = 0;
47     }
48
49     _switch(&current->as);
50     return current->tf;
51 }

```

再次运行仙剑奇侠传代码，效率明显提升。



以上是本次实验阶段二部分。

PA4-阶段三

第三阶段的主要任务是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。

3.1 代码：添加时钟中断

添加时钟中断

根据讲义的上述内容,添加相应的代码来实现真正的分时多任务.为了证明时钟中断确实在工作,你可以在 `Nanos-lite` 收到 `_EVENT_IRQ_TIME` 事件后用 `Log()` 输出一句话.

需要注意的是,添加时钟中断之后,differential testing 机制就无法正确工作了.这是因为,我们无法给 QEMU 注入时钟中断,无法保证 QEMU 与 NEMU 处于相同的状态.不过,differential testing 作为一个强大的工具用到这时候,指令实现的正确性也基本上得到相当大的保证了.

首先,在 `nemu/include/cpu/reg.h` 中添加bool变量代表硬件中断。

```
61     bool INTR;
62 } CPU_state;
```

在 `nemu/src/cpu/intr.c` 中初始化变量。

```
33 void dev_raise_intr() {
34     cpu.INTR = true;
35 }
```

在 `nemu/src/cpu/exec/exec.c` 中添加如下代码。

```
15 #define TIME_IRQ 32
```

```
256     if(cpu.INTR & cpu.eflags.IF){
257         cpu.INTR = false;
258         extern void raise_intr(uint8_t NO, vaddr_t ret_addr);
259         raise_intr(TIME_IRQ, cpu.eip);
260         update_eip();
261     }
```

在 `nemu/src/cpu/intr.c` 中修改代码如下。

```
10 //依次将eflags,cs,eip入栈
11 memcpy(&t1, &cpu.eflags, sizeof(cpu.eflags));
12 rtl_li(&t0, t1);
13 rtl_push(&t0); //eflags
14 cpu.eflags.IF = 0;
15 rtl_push(&cpu.cs); //cs
16 rtl_li(&t0, ret_addr);
17 rtl_push(&t0); //eip
```

在 `nexus-am/am/arch/x86-nemu/src/asye.c` 中添加如下代码。

```
6 void vecsys();
7 void vecnull();
8 void vecself();
9 void vectime();

39     idt[0x80] = GATE(STS_TG32, KSEL(SEG_KCODE), vecsys, DPL_USER);
40     idt[0x81] = GATE(STS_TG32, KSEL(SEG_KCODE), vecself, DPL_USER);
41     idt[32] = GATE(STS_TG32, KSEL(SEG_KCODE), vectime, DPL_USER);

15     switch (tf->irq) {
16         case 0x80: ev.event = _EVENT_SYSCALL; break;
17         case 0x81: ev.event = _EVENT_TRAP; break;
18         case 32: ev.event = _EVENT_IRQ_TIME; break;
19         default: ev.event = _EVENT_ERROR; break;
20     }
```

在 `nexus-am/am/arch/x86-nemu/src/trap.s` 中添加如下代码。

```
2 .globl vecsys;    vecsys:  pushl $0;  pushl $0x80; jmp asm_trap
3 .globl vecnull;   vecnull:  pushl $0;  pushl $-1; jmp asm_trap
4 .globl vecself;   vecself:  pushl $0;  pushl $0x81; jmp asm_trap
5 .globl vectime;   vectime:  pushl $0;  pushl $32; jmp asm_trap
```

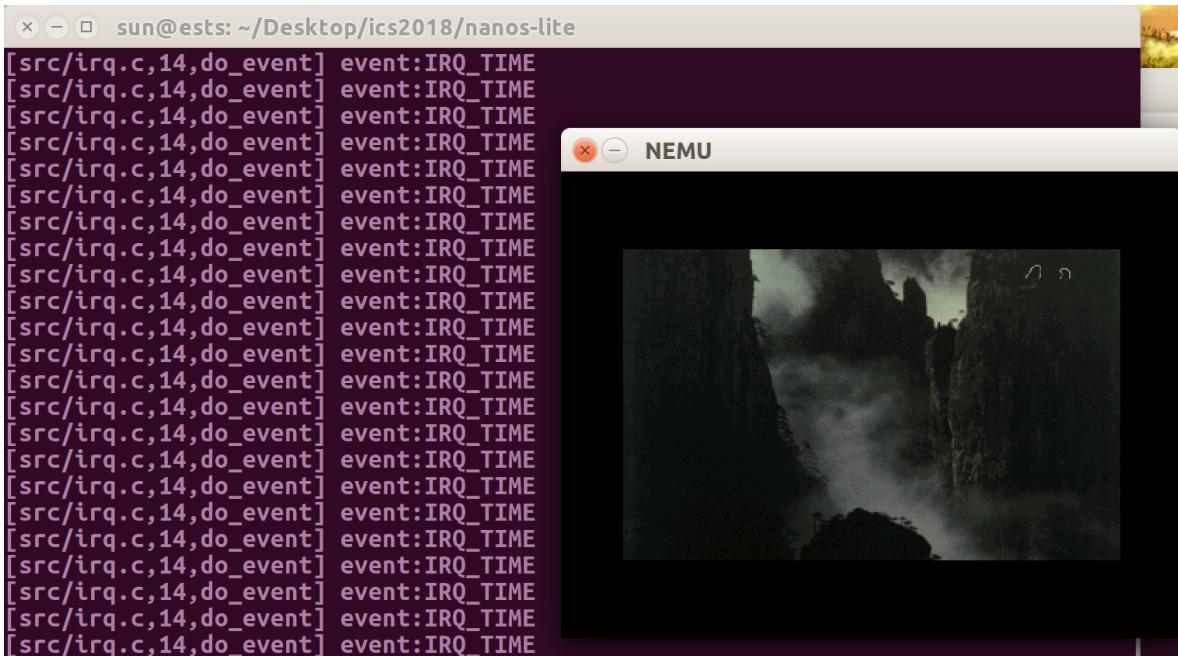
接下来，在 `nanos-lite/src/irq.c` 中添加事件分发代码如下。

```
13         case _EVENT_IRQ_TIME:
14             Log("event:IRQ_TIME");
15             return schedule(r);
```

在 `nexus-am/am/arch/x86-nemu/src/pte.c` 中添加代码如下。

```
99     tf.eflags = 0x02 | FL_IF;
```

分时运行仙剑奇侠传和hello程序，结果如下。



此时进程切换不在系统调用时发生，而在时钟中断时尝试进程切换。

3.2 必答题

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。

首先，为实现多进程分时运行，引入了虚拟内存的概念，让程序链接到固定的虚拟地址的同时，加载到不同的物理位置去执行，分页机制保证了不同进程拥有独立的存储空间。具体而言，NEMU提供控制寄存器CR0、CR3来辅助实现分页机制。MMU进行分页地址的转换，详见代码中的 `vaddr_read()` 和 `vaddr_write()`。之后，Nanos-lite与AM协作，一起准备内核页表以启动分页机制。nanos-lite通过 `load_prog()` 函数实现用户程序的加载。最后，硬件中断与上下文切换保证了程序的分时运行。`exec_wrapper` 每完成一条指令，便检查是否有硬件中断到来，触发中断时，中断被打包成 `IRQ_TIME` 事件，之后调用 `schedule()` 进行进程调度。

以上是本次实验阶段三部分。

编写不朽的传奇

展示你的计算机系统

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，你还需要修改 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。例如，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。

首先，在 nanos-lite/src/main.c 添加如下代码。

```
37     load_prog("/bin/pal");
38     load_prog("/bin/hello");
39     load_prog("/bin/videotest");
```

在 nanos-lite/src/proc.c 中添加代码如下。

```
29 int current_game = 0;
30 void switch_current_game(){
31     current_game = 2 - current_game;
32     Log("current_game=%d", current_game);
33 }
34
35 _RegSet* schedule(_RegSet *prev) {
36     //return NULL;
37     if(current != NULL)
38         current->tf = prev;
39     else
40         current = &pcb[current_game];
41     //current = (current == &pcb[0] ? &pcb[1] : &pcb[0]);
42     //Log("ptr=0x%x\n", (uint32_t)current->as.ptr);
43
44     static int num = 0;
45     static const int frequency = 1000;
46     if(current == &pcb[current_game])
47         num++;
48     else
49         current = &pcb[current_game];
50     if(num == frequency){
51         current = &pcb[1];
52         num = 0;
53     }
54
55     _switch(&current->as);
56     return current->tf;
57 }
```

在 nanos-lite/src/device.c 中添加如下代码。

```
21     if(down && key == KEY_F12){
22         extern void switch_current_game();
23         switch_current_game();
24         Log("key down:_KEY_F12, switch current game");
25     }
```

最后，运行程序。可以观测到仙剑奇侠传与 hello 程序分时运行。按下 F12 后变为 videotest 与 hello 分时运行。

以上是本次实验报告内容，感谢批阅！
