



南開大學
Nankai University

计算机学院
计算机系统设计实验报告

PA4 - 虚实交错的魔法：分时多任务

姓名：李颖

学号：2110939

专业：计算机科学与技术

2024 年 6 月 3 日

目录

1 实验介绍	2
1.1 实验目的	2
1.2 实验流程	2
2 阶段一：虚拟地址空间	2
2.1 一些问题的解答	2
2.2 在 NEMU 中实现分页机制	3
2.2.1 准备 CR0 和 CR3 寄存器及相关操作	3
2.2.2 准备内核页表	7
2.3 让用户程序运行在分页机制上	10
2.3.1 实现运行在用户空间	10
2.3.2 实现 <code>_map()</code>	11
2.4 问题：内核映射的作用	12
2.5 在分页机制上运行仙剑奇侠传	13
3 阶段二：上下文切换	14
3.1 实现内核自陷	14
3.2 实现上下文切换	16
3.3 分时多任务：同时运行两个程序	18
3.4 优先级调度	19
4 阶段三：来自外部的声音	20
4.1 问题：灾难性的后果	21
4.2 添加时钟中断	21
4.2.1 在 NEMU 中添加 INTR	21
4.2.2 在软件上添加对时钟中断的处理	22
5 必答题	24
6 展示计算机系统	26
7 BUG 汇总	28
7.1 dummy 无法正常运行	28
7.2 实现时钟中断后不断显示 <code>event: trap</code>	28

1 实验介绍

在 PA3 中,《仙剑奇侠传》已经能够被成功启动,并新建存档开始游戏。但是游戏实际上运行在内核空间上,十分不安全也容易被覆盖。因此,在本次实验中,我们将实现把用户程序运行在虚拟地址空间上,并实现上下文切换,使操作系统能同时运行多个程序;最后还需要完善时钟中断,防止恶意程序一直占有资源。

1.1 实验目的

- 学习并了解虚拟地址空间及虚拟地址映射;了解分段机制,学会如何处理超越容量的界限。
- 学习并了解上下文切换的原理,并实现内核自陷、进程调度与分时多任务。
- 学习中断机制,实现时钟中断。

1.2 实验流程

- 在 NEMU 中实现分页机制:准备 CR0 和 CR3 寄存器并完善相关操作。实现经过分页地址转换后再访问虚拟地址。
- 将程序链接地址改为虚拟地址,实现虚拟地址空间的映射,让程序可以运行在用户空间。
- 实现上下文切换:实现内核自陷,完善上下文切换,实现分时多任务并同时运行两个程序,完善优先级调度策略。
- 实现时钟中断:添加 INTR 并完善操作,添加对时钟中断的处理。

2 阶段一：虚拟地址空间

2.1 一些问题的解答

1. i386 不是一个 32 位的处理器吗,为什么表项中的基地址信息只有 20 位,而不是 32 位?

解答:原因有以下几点:

- 物理内存寻址限制:在 32 位系统中,寻址的物理内存地址范围是 2^{32} ,即 4GB。但实际系统中并不会将整个 4GB 的物理地址空间都映射到页表中,这是出于性能和空间的考虑。因此,为了节省空间并提高效率,只需要部分位数来表示基地址信息即可。
- 虚拟内存管理:页表不仅仅用于直接映射物理内存地址,还要支持虚拟内存管理和分页机制。在分页机制中,虚拟地址被分割为固定大小的页,页表项中的基地址信息只需表示页内的偏移量,而不是整个地址空间的范围。
- 其他控制信息:除了基地址信息外,页表项还包含其他重要的控制信息,如页表标志位、权限位等,这些信息也需要占用部分位数。其他较小的位数来表示页内偏移量。这种设计可以节省内存空间和提高访问效率。

2. 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?

解答：虚拟地址和物理地址之间的映射关系是通过页表来实现的，页表中存储的地址必须是物理地址，因为操作系统和硬件需要根据物理地址来定位内存中的页框或页表项。虚拟地址和物理地址之间的转换需要依赖硬件支持，并且需要保证页表中存储的地址是物理地址，以正确进行地址转换和内存访问操作。

3. 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？

解答：虽然一级页表结构简单直接，但由于空间浪费、内存访问效率低、缺乏灵活性和不支持复杂的地址转换需求等缺点，一般操作系统在设计时会选择采用更复杂的多级页表结构，以便更好地管理内存、提高效率 and 灵活性。多级页表结构可以降低内存占用和提高内存访问效率，同时支持更复杂的内存管理需求。

4. 空指针的本质

解答：关于空指针本质，有以下几点解释：

- 空指针并不一定是值为 0：虽然 NULL 指针通常定义为 0，但在实际情况中，空指针的二进制表示可以是非 0 值，只要它不指向任何有效的内存地址即可。因此，判断指针是否为空应该使用指针的布尔判断，而不是直接和 0 进行比较。
- 空指针并不是一定指向地址 0：空指针不一定指向物理地址 0，而是指向一个操作系统或编译器规定的无效地址范围，用于表示指针不指向有效的内存区域。因此，空指针并不一定实际指向地址 0，而是表示不指向任何有效地址。
- 空指针的概念是编程语言层面上的抽象：空指针的概念是编程语言层面上的抽象表示，用于表示指针不指向有效的内存地址。在实际硬件层面，计算机对空指针的处理方式可能有所不同，但通常会触发异常以防止对无效内存的访问。

2.2 在 NEMU 中实现分页机制

2.2.1 准备 CR0 和 CR3 寄存器及相关操作

为了实现分页机制，首先需要准备内核页表。内核页表在初始化存储管理器 MM 时，会需要设置 CR3 和 CR0 寄存器。

因此，我们首先需要在 reg.h 中添加这两个控制寄存器。

```
1  rtlreg_t  cs;  
2  rtlreg_t  ds;  
3  rtlreg_t  es;  
4  uint32_t  CR0;  
5  uint32_t  CR3;
```

然后，需要在 monitor.c 的 restart() 函数中初始化 CR0 为 0x60000011。

```
1  static inline void restart() {  
2      /* Set the initial instruction pointer. */  
3      cpu.eip = ENTRY_START;  
4      cpu.cs = 8;  
5      cpu.CR0 = 0x60000011;  
6      unsigned int origin = 2;  
7      memcpy(&cpu.eflags, &origin, sizeof(cpu.eflags));  
8  }
```

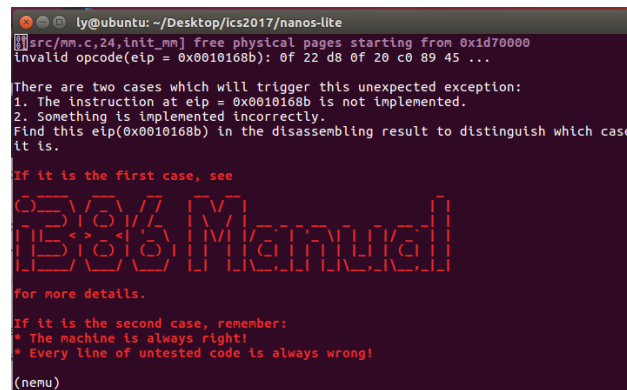
```

9 #ifdef DIFF_TEST
10     init_qemu_reg();
11 #endif
12 }

```

在 nanos-lite/src/main.c 中定义宏 HAS_PTE，即可在初始化时调用 init_mm() 函数对 MM 进行初始化。

运行后得到如下画面。



为了实现对 CR0 和 CR3 的操作，还需要在 rtl.h 中补充 rtl_store_cr() 函数，使其能够存储控制信息，根据情况将信息保存到 CR0 或 CR3 中。

```

1 static inline rtl_store_cr(int r, const rtlreg_t* src) {
2     switch (r)
3     {
4     case 0:
5         cpu.CR0 = *src;
6         return;
7     case 3:
8         cpu.CR3 = *src;
9         return;
10    default:
11        assert(0);
12    }
13    return;
14 }

```

相对应的，还需要实现 rtl_load_cr() 函数，将 CR0 或 CR3 的内容读取出来。

```

1 static inline rtl_load_cr(rtlreg_t* dest, int r) {
2     switch (r)
3     {
4     case 0:
5         *dest = cpu.CR0;
6         return;
7         break;
8     case 3:
9         *dest = cpu.CR3;

```

```

10     return;
11     default:
12         assert(0);
13     }
14     return;
15 }

```

之后需要实现 `init_mm` 中遇到的指令，即关于 `CR0` 和 `CR3` 的 `mov` 操作。需要在 `decode.c` 中实现对应的译码函数。`mov_load_cr` 为把控制寄存器中存储的值读取到目的寄存器中，`mov_save_cr` 为把源寄存器的值存储到控制寄存器中。

`decode.h`:

```

1 make_DHelper(mov_load_cr);
2 make_DHelper(mov_store_cr);

```

`decode.c`:

```

1 make_DHelper(mov_load_cr) {
2     decode_op_rm(eip, id_dest, false, id_src, false);
3     rtl_load_cr(&id_src -> val, id_src -> reg);
4     #ifdef DEBUG
5         snprintf(id_src -> str, 5, "%cr%d", id_dest -> reg);
6     #endif
7 }
8
9 make_DHelper(mov_store_cr) {
10     decode_op_rm(eip, id_src, true, id_dest, false);
11     #ifdef DEBUG
12         snprintf(id_src -> str, 5, "%cr%d", id_dest -> reg);
13     #endif
14 }

```

`exec.c`:

```

1 /* 0x20 */ IDEX(mov_load_cr, mov), EMPTY, IDEX(mov_store_cr, mov_store_cr), EMPTY,

```

在 `data-mov.c` 中添加实际执行函数，如下所示。

```

1 make_EHelper(mov_store_cr) {
2     rtl_store_cr(id_dest -> reg, &id_src -> val);
3     print_asm_template2(mov);
4 }

```

之后，在 `monitor/debug/ui.c` 中修改 `cmd_info()` 函数，使其能够打印 `CR0` 和 `CR3` 等更多寄存器信息。

```

1
2 static int cmd_info(char *args) {
3     char s;
4     if(args == NULL) {

```

```
5     printf("args error in cmd_info (miss args)\n");
6     return 0;
7 }
8 int temp = sscanf(args, "%c", &s);
9 if(temp <= 0) {
10     //解析失败
11     printf("args error in cmd_info\n");
12     return 0;
13 }
14 if(s == 'w') {
15     //打印监视点信息
16     print_wp();
17     return 0;
18 }
19 if(s == 'r') {
20     //打印寄存器
21     //32bit
22     for(int i = 0; i < 8; i++) {
23         printf("%s 0x%x\n", regsl[i], reg_l(i));
24     }
25     printf("eip 0x%x\n", cpu.eip);
26     //16bit
27     for(int i = 0; i < 8; i++) {
28         printf("%s 0x%x\n", regsw[i], reg_w(i));
29     }
30     //8bit
31     for(int i = 0; i < 8; i++)
32     {
33         printf("%s 0x%x\n", regsb[i], reg_b(i));
34     }
35     printf("eflags:CF=%d,ZF=%d,SF=%d,IF=%d,OF=%d\n", cpu.eflags.CF, cpu.eflags.ZF,
36           cpu.eflags.SF, cpu.eflags.IF, cpu.eflags.OF);
37     printf("CR0=0x%x, CR3=0x%x\n", cpu.CR0, cpu.CR3);
38     return 0;
39 }
40 //如果产生错误
41 printf("args error in cmd_info\n");
42 return 0;
43 }
```

查看寄存器信息，可以看到 CR0 的值已被初始化为 0x60000011。

```

ly@ubuntu: ~/Desktop/ics2017/nanos-lite
(nemu) Info: r
eax 0x5ad3a4e
ecx 0x3d793665
edx 0x3bbd287
ebx 0x29341312
esp 0x10ce2816
ebp 0x5069b133
esi 0x2f8e5ba2
edi 0x2d6624a4
eip 0x100000
ax 0x3a4e
cx 0x3665
dx 0xd287
bx 0x1312
sp 0x2816
bp 0xb133
si 0x5ba2
di 0x24a4
al 0x4e
cl 0x65
dl 0x87
bl 0x12
ah 0x3a
ch 0x36
dh 0xd2
bh 0x13
eflags:CF=0,ZF=0,SF=0,IF=0,OF=0
CR0=0x00000011, CR3=0x0
(nemu)

```

2.2.2 准备内核页表

之后，需要对 memory.c 中的 vaddr_read() 和 vaddr_write() 函数作少量修改。
在修改之前，首先需要再 memory.c 中添加需要的宏定义，如下所示。

```

1 #define PTXSHFT 12 //线性地址偏移量
2 #define PDXSHFT 22 //线性地址偏移量
3
4 #define PTE_ADDR(pte) (((uint32_t)(pte) & ~0xfff))
5 #define PDX(va) (((uint32_t)(va) >> PDXSHFT) & 0x3ff)
6 #define PTX(va) (((uint32_t)(va) >> PTXSHFT) & 0x3ff)
7 #define OFF(va) ((uint32_t)(va) & 0xfff)

```

之后补充 vaddr_read() 函数，该函数作用为实现读地址时虚拟地址转换。

```

1 uint32_t vaddr_read(vaddr_t addr, int len) {
2     if(PTE_ADDR(addr) != PTE_ADDR(addr + len - 1)) {
3         // printf("error: the data pass two pages:addr=0x%x, len=%d!\n", addr, len);
4         assert(0);
5     }
6     else {
7         paddr_t paddr = page_translate(addr, false);
8         return paddr_read(paddr, len);
9     }
10    // return paddr_read(addr, len);
11 }

```

补充补充 vaddr_write() 函数，该函数作用为实现写地址时虚拟地址转换

```

1 void vaddr_write(vaddr_t addr, int len, uint32_t data) {
2     if(PTE_ADDR(addr) != PTE_ADDR(addr+len-1)) {
3         // printf("error: the data pass two pages:addr=0x%x, len=%d!\n", addr, len);
4         assert(0);
5     }
6     else {
7         paddr_t paddr = page_translate(addr, true);
8         paddr_write(paddr, len, data);

```



```

9     }
10    // paddr_write(addr, len, data);
11 }

```

由于 i386 并没有严格要求数据对齐, 因此可能会出现数据跨越虚拟页边界的情况, 例如一条很长的指令的首字节在一个虚拟页的最后, 剩下的字节在另一个虚拟页的开头。然而, 这两个函数目前只能处理在某一页内读写的情况, 若出现多页读写, 则暂时不做任何处理。

之后, 需要按要求编写 `page_translate()` 函数。由于不打算实现保护机制, 在 `page_translate()` 函数的实现中, 必须使用 `assertion` 检查页目录项和页表项的 `present` 位, 如果发现了一个无效的表项, 及时终止 NEMU 的运行。实现的 `page_translate()` 函数如下所示。该函数需要判断页目录、页表是否存在, 并根据情况对页面的脏位进行标记; 若页面不存在, 则需要报错并终止程序。

```

1 paddr_t page_translate(vaddr_t addr, bool iswrite) {
2     CR0 cr0 = (CR0)cpu.CR0;
3     if(cr0.paging && cr0.protect_enable) {
4         CR3 cr3 = (CR3)cpu.CR3;
5
6         PDE *pgdirs = (PDE*)PTE_ADDR(cr3.val);
7         PDE pde = (PDE)paddr_read((uint32_t)(pgdirs + PDX(addr)), 4);
8
9         PTE *ptable = (PTE*)PTE_ADDR(pde.val);
10        PTE pte = (PTE)paddr_read((uint32_t)(ptable + PTX(addr)), 4);
11        //printf("hhahah%x, jhhh%x\n", pte.present, addr);
12        Assert(pte.present, "addr=0x%x", addr);
13
14        pde.accessed=1;
15        pte.accessed=1;
16        if(iswrite) {
17            pte.dirty=1;
18        }
19        paddr_t paddr = PTE_ADDR(pte.val) | OFF(addr);
20        // printf("vaddr=0x%x, paddr=0x%x\n", addr, paddr);
21        return paddr;
22    }
23    return addr;
24 }

```

运行结果如下所示, 可以发现, 虚拟地址与物理地址相同, 说明分页机制基本完成; 当出现跨页的情况时, 程序会触发 `assertion` 并终止。

```

ly@ubuntu: ~/Desktop/ics2017/nanos-lite
vaddr=0x1014e5, paddr=0x1014e5
vaddr=0x4000ca0, paddr=0x4000ca0
vaddr=0x1014e6, paddr=0x1014e6
vaddr=0x1014e7, paddr=0x1014e7
vaddr=0x1014e8, paddr=0x1014e8
vaddr=0x1014e9, paddr=0x1014e9
vaddr=0x4000ca4, paddr=0x4000ca4
vaddr=0x1014ea, paddr=0x1014ea
vaddr=0x1014eb, paddr=0x1014eb
vaddr=0x1014ec, paddr=0x1014ec
vaddr=0x1014ed, paddr=0x1014ed
vaddr=0x1b3cff9, paddr=0x1b3cff9
vaddr=0x1014ee, paddr=0x1014ee
vaddr=0x1014ef, paddr=0x1014ef
vaddr=0x1014f0, paddr=0x1014f0
vaddr=0x1014f1, paddr=0x1014f1
nenu: src/memory/memory.c:71: vaddr_read: Assertion '0' failed.
Makefile:46: recipe for target 'run' failed
make[1]: *** [run] Aborted (core dumped)
make[1]: Leaving directory '/home/ly/Desktop/ics2017/nenu'
/home/ly/Desktop/ics2017/nexus-am/Makefile.app:35: recipe for target 'run' failed
make: *** [run] Error 2
ly@ubuntu: ~/Desktop/ics2017/nanos-lite$

```

对于数据跨越虚拟页边界的情况，解决办法为：如果这两个虚拟页被映射到两个不连续的物理页，就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。因此，我们需要修改 `vaddr_read()` 和 `vaddr_write()` 函数，使其能处理这种情况，修改后的代码如下所示。

```

1  uint32_t vaddr_read(vaddr_t addr, int len) {
2      if(PTE_ADDR(addr) != PTE_ADDR(addr + len - 1)) {
3          // printf("error: the data pass two pages:addr=0x%x, len=%d!\n", addr, len);
4          // assert(0);
5          int num1 = 0x1000 - OFF(addr);
6          int num2 = len - num1;
7          paddr_t paddr1 = page_translate(addr, false);
8          paddr_t paddr2 = page_translate(addr + num1, false);
9
10         uint32_t low = paddr_read(paddr1, num1);
11         uint32_t high = paddr_read(paddr2, num2);
12
13         uint32_t result = high << (num1 * 8) | low;
14         return result;
15     }
16     else {
17         paddr_t paddr = page_translate(addr, false);
18         return paddr_read(paddr, len);
19     }
20     // return paddr_read(addr, len);
21 }
22
23 void vaddr_write(vaddr_t addr, int len, uint32_t data) {
24     if(PTE_ADDR(addr) != PTE_ADDR(addr+len-1)) {
25         // printf("error: the data pass two pages:addr=0x%x, len=%d!\n", addr, len);
26         // assert(0);
27         if(PTE_ADDR(addr) != PTE_ADDR(addr + len - 1)) {
28             int num1 = 0x1000-OFF(addr);
29             int num2 = len -num1;
30             paddr_t paddr1 = page_translate(addr, true);
31             paddr_t paddr2 = page_translate(addr + num1, true);
32
33             uint32_t low = data & (~0u >> ((4 - num1) << 3));

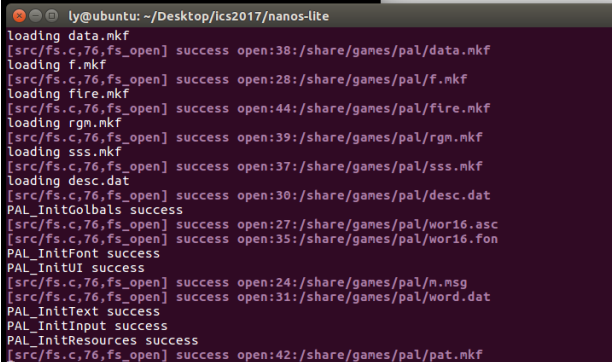
```

```

34     uint32_t high = data >> ((4 - num2) << 3);
35
36     paddr_write(paddr1, num1, low);
37     paddr_write(paddr2, num2, high);
38     return;
39 }
40 }
41 else {
42     paddr_t paddr = page_translate(addr, true);
43     paddr_write(paddr, len, data);
44 }
45 // paddr_write(addr, len, data);
46 }

```

重新运行程序，不再触发 assertion 错误，能够成功启动《仙剑奇侠传》。



```

ly@ubuntu: ~/Desktop/lcs2017/nanos-lite
loading data.mkf
[src/fs.c,76,fs_open] success open:38:/share/games/pal/data.mkf
loading f.mkf
[src/fs.c,76,fs_open] success open:28:/share/games/pal/f.mkf
loading fire.mkf
[src/fs.c,76,fs_open] success open:44:/share/games/pal/fire.mkf
loading rgn.mkf
[src/fs.c,76,fs_open] success open:39:/share/games/pal/rgn.mkf
loading sss.mkf
[src/fs.c,76,fs_open] success open:37:/share/games/pal/sss.mkf
loading desc.dat
[src/fs.c,76,fs_open] success open:30:/share/games/pal/desc.dat
PAL_InitGlobals success
[src/fs.c,76,fs_open] success open:27:/share/games/pal/wor16.asc
[src/fs.c,76,fs_open] success open:35:/share/games/pal/wor16.fon
PAL_InitFont success
PAL_InitUI success
[src/fs.c,76,fs_open] success open:24:/share/games/pal/n.msg
[src/fs.c,76,fs_open] success open:31:/share/games/pal/word.dat
PAL_InitText success
PAL_InitInput success
PAL_InitResources success
[src/fs.c,76,fs_open] success open:42:/share/games/pal/pat.mkf

```

2.3 让用户程序运行在分页机制上

虽然《仙剑奇侠传》能够重新运行，但游戏运行在内核的虚拟空间上，因此，需要该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。

2.3.1 实现运行在用户空间

对此，我们需要对工程进行相应的改动。首先需要将 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。修改后的 makefile 文件如下所示，将链接地址从 0x4000000 改为 0x8048000。用户程序的代码从 0x8048000 附近开始，这个地址已经超过了物理地址的最大值 (NEMU 提供的物理内存是 128MB)，但分页机制保证了程序能够正确运行。这样，链接器和程序都不需要关心程序运行时刻具体使用哪一段物理地址，它们只要使用虚拟地址就可以了，而虚拟地址和物理地址之间的映射则全部交给操作系统的 MM 来管理。

```

1 ifeq ($(LINK), dynamic)
2     CFLAGS    += -fPIE
3     CXXFLAGS += -fPIE
4     LDFLAGS   += -fpie -shared
5 else
6     LDFLAGS   += -Ttext 0x8048000

```

```
7 endif
```

同时，还需修改 nanos-lite/src/loader.c 中的 DEFAULT_ENTRY，如下所示。

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

同时，在 main.c 中将运行的程序修改为 dummy，运行结果如下所示。可以发现由于未实现页的分配与映射，程序报了缺页错误。

```
ly@ubuntu: ~/Desktop/ics2017/nanos-lite
Welcome to NEMU!
[src/monitor/monitor.c:30, welcome] Build time: 01:51:01, May 25 2024
For help, type "help"
(nemu) c
[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: 01:44:02, May 28 2024
[src/randisk.c:26, init_randisk] randisk info: start = 0x1025e8, end = 0x1d4c5a9,
size = 29663169 bytes
[src/main.c:27, main] Initializing interrupt/exception handler...
[src/cpu/exec/system.c:19, exec_ldt] idtr.limit=0x7fff
[src/cpu/exec/system.c:20, exec_ldt] idtr.base=0x1d4d0c0
[src/fs.c:42, init_fs] set FD_FB size = 480000
[src/fs.c:76, fs_open] success open:45:/bin/dummy
[src/loader.c:17, loader] filename=/bin/dummy, fd=45
addr=0x8048000
nemu: src/memory/memory.c:54: page_translate: Assertion 'pte.present' failed.
makefile:46: recipe for target 'run' failed
make[1]: *** [run] Aborted (core dumped)
make[1]: Leaving directory '/home/ly/Desktop/ics2017/nemu'
/home/ly/Desktop/ics2017/nexus-am/Makefile.app:35: recipe for target 'run' failed
make: *** [run] Error 2
ly@ubuntu:~/Desktop/ics2017/nanos-lite$
```

2.3.2 实现 __map()

因此，我们需要在 nexus-am/am/arch/x86-nemu/src/pte.c 中实现 __map() 函数，将虚拟空间 p 中的虚拟地址 va 映射到物理地址 pa，通过 p->ptr 可以获取页目录的基地址；如果映射过程中发现需要申请新的页，就调用 palloc_f() 函数来完成。

```
1 void __map(_Protect *p, void *va, void *pa) {
2     if(OFF(va) || OFF(pa)) {
3         // printf("page not aligned\n");
4         return;
5     }
6
7     PDE *dir = (PDE*) p -> ptr;
8     PTE *table = NULL;
9     PDE *pde = dir + PDX(va);
10    if(!(*pde & PTE_P)) {
11        table = (PTE*) (palloc_f());
12        *pde = (uintptr_t) table | PTE_P;
13    }
14    table = (PTE*) PTE_ADDR(*pde);
15    PTE *pte = table + PTX(va);
16    *pte = (uintptr_t) pa | PTE_P;
17 }
```

在 load_prog() 调用 loader() 函数加载用户程序时，loader() 需要以页的单位进行加载，具体流程为：

1. 申请一页空闲的物理页
2. 把这一物理页映射到用户程序的虚拟地址空间中

3. 从文件中读入一页的内容到这一物理页上

因此，修改后的 loader() 函数如下所示：

```

1  uintptr_t loader(_Protect *as, const char *filename) {
2      // ramdisk_read(DEFAULT_ENTRY, 0, RAMDISK_SIZE);
3      int fd = fs_open(filename, 0, 0);
4      Log("filename=%s,fd=%d", filename, fd);
5      // fs_read(fd, DEFAULT_ENTRY, fs_filesz(fd));
6      int size = fs_filesz(fd);
7      int ppnum = size / PGSIZE;
8      if(size % PGSIZE != 0) {
9          ppnum++;
10     }
11     void *pa = NULL;
12     void *va = DEFAULT_ENTRY;
13     for(int i = 0; i < ppnum; i++) {
14         pa = new_page();
15         _map(as, va, pa);
16         fs_read(fd, pa, PGSIZE);
17         va += PGSIZE;
18     }
19
20     fs_close(fd);
21     return (uintptr_t)DEFAULT_ENTRY;
22 }

```

启动程序，发现 dummy 能够成功 hit good trap。

```

ly@ubuntu: ~/Desktop/ics2017/nanos-lite
make: [klib] Error 2 (ignored)
make[1]: Entering directory '/home/ly/Desktop/ics2017/nemu'
./build/nemu -l /home/ly/Desktop/ics2017/nanos-lite/build/nemu-log.txt /home/ly/
Desktop/ics2017/nanos-lite/build/nanos-lite-x86-nemu.bin
[src/monitor/monitor.c,65,load_img] The image is /home/ly/Desktop/ics2017/nanos-
lite/build/nanos-lite-x86-nemu.bin
Nice Calc to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 01:51:01, May 25 2024
For help, type "help"
(nemu) c
[src/mm.c,24,init_mm] free physical pages starting from 0x1d92000
[src/main.c,19,main] 'Hello World!' from Nanos-lite
[src/main.c,20,main] Build time: 19:34:28, May 28 2024
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x1027e8, end = 0x1d4c7a9,
size = 29663169 bytes
[src/main.c,27,main] Initializing interrupt/exception handler...
[src/cpu/exec/system.c,19,exec_ldtr] ldtr.limit=0x7fff
[src/cpu/exec/system.c,20,exec_ldtr] ldtr.base=0x1d6f0a0
[src/fs.c,42,init_fs] set FD_FB size = 480000
[src/fs.c,76,fs_open] success open:45:/bin/dummy
[src/loader.c,18,loader] filename=/bin/dummy,fd=45
nemu: HIT GOOD TRAP at eip = 0x00100032

```

2.4 问题：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：

```

1  for (int i = 0; i < NR_PDE; i++) {
2      updir[i] = kpdirs[i];
3  }

```

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。

解答：

运行后，报缺页的错误，如下所示。

```
ly@ubuntu: ~/Desktop/ics2017/nanos-lite
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 01:51:01, May 25 2024
For help, type "help"
(nemu) c
[src/mm.c,24,init_mm] free physical pages starting from 0x1d92000
[src/main.c,19,main] 'Hello World!' from Nanos-lite
[src/main.c,20,main] Build time: 19:34:28, May 28 2024
[src/randisk.c,26,init_randisk] randisk info: start = 0x1027c8, end = 0x1d4c789,
size = 29663169 bytes
[src/main.c,27,main] initializing interrupt/exception handler...
[src/cpu/exec/system.c,19,exec_ldtr] idtr.limit=0x7ff
[src/cpu/exec/system.c,20,exec_ldtr] idtr.base=0x1d6f0a0
[src/fs.c,42,init_fs] set FD_FB size = 480000
[src/fs.c,76,fs_open] success open:45:/bin/dummy
[src/loader.c,18,loader] filename=/bin/dummy,fd=45
addr=0x1017c0
nemu: src/memory/memory.c:54: page_translate: Assertion 'pte.present' failed.
Makefile:46: recipe for target 'run' failed
make[1]: *** [run] Aborted (core dumped)
make[1]: Leaving directory '/home/ly/Desktop/ics2017/nemu'
/home/ly/Desktop/ics2017/nexus-am/Makefile.app:35: recipe for target 'run' failed
make: *** [run] Error 2
ly@ubuntu:~/Desktop/ics2017/nanos-lite$
```

这是因为，这段代码是为了将内核空间的页表项复制到用户空间的页表项中，这样用户空间才能访问到内核空间的内存。如果你注释掉这段代码，用户空间的页表项将为空，因此用户空间就无法访问到内核空间的内存，导致程序运行错误。

2.5 在分页机制上运行仙剑奇侠传

之前 `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，还需要在 `mm_brk()` 中把新申请的堆区映射到虚拟地址空间中。

在 `nanos-lite/src/mm` 修改后的代码如下所示：

```
1 int mm_brk(uint32_t new_brk) {
2     if(current -> cur_brk == 0) {
3         current -> cur_brk = current -> max_brk = new_brk;
4     }
5     else {
6         if(new_brk > current -> max_brk) {
7             uint32_t first = PGROUNDUP(current -> max_brk);
8             uint32_t end = PGROUNDDOWN(new_brk);
9             if((new_brk & 0xfff) == 0) {
10                 end -= PGSIZE;
11             }
12             for(uint32_t va = first; va <= end; va += PGSIZE) {
13                 void *pa = new_page();
14                 _map(&(current -> as), (void*)va, pa);
15             }
16             current -> max_brk = new_brk;
17         }
18         current -> cur_brk = new_brk;
19     }
20     return 0;
21 }
```

相应地，修改 `nanos-lite/src/syscall.c` 的 `sys_brk()` 函数。

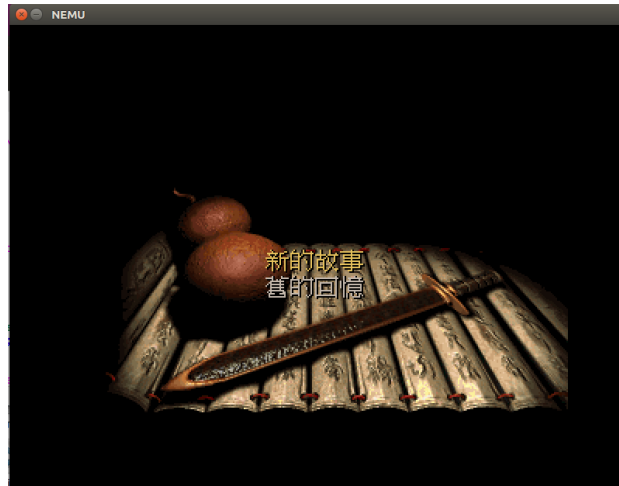
```
1 int sys_brk(int addr) {
```

```

2  extern int mm_brk(uint32_t new_brk);
3  return mm_brk(addr);
4  }

```

重新运行仙剑奇侠传，如下所示。



3 阶段二：上下文切换

当前，用户程序已经可以运行在相互独立的虚拟地址空间上了。为了实现一个真正的分时多任务操作系统，还需要加入上下文切换机制。而实现不同程序之间的陷阱帧之间的切换，就能实现上下文切换。

3.1 实现内核自陷

程序需要通过自陷指令 `int $0x81` 触发上下文切换，因此，首先需要实现内核自陷功能 `__trap()`。

由于内核自陷通过指令 `int $0x81` 触发，因此将该汇编码传递给 `asye.c` 中的 `__trap()` 函数，如下所示。

```

1  void __trap() {
2      asm volatile("int $0x81");
3  }

```

将 `nanos-lite/src/main.c` 中添加 `__trap()` 函数调用：

```

1  extern void load_prog(const char *filename);
2  //load_prog("/bin/dummy");
3  load_prog("/bin/pal");
4
5  __trap();
6
7  panic("Should not reach here");

```

在 `nanos-lite/src/proc.c` 中注释以下代码：

```

1  // TODO: remove the following three lines after you have implemented __umake()

```

```

2 //__switch(&pcb[i].as);
3 //current = &pcb[i];
4 //(((void (*)(void))entry)());

```

之后，在 nexus-am/am/arch/x86-nemu/src/asye.c 文件中为 irq_handle() 函数新增 0x81 的情况，将该异常封装为 _EVENT_TRAP 事件。

```

1 __RegSet* irq_handle(__RegSet *tf) {
2     __RegSet *next = tf;
3     if (H) {
4         __Event ev;
5         switch (tf->irq) {
6             case 0x80: ev.event = _EVENT_SYSCALL; break;
7             case 0x81: ev.event = _EVENT_TRAP; break;
8             default: ev.event = _EVENT_ERROR; break;
9         }
10
11         next = H(ev, tf);
12         if (next == NULL) {
13             next = tf;
14         }
15     }
16     return next;
17 }

```

在 nanos-lite/src/irq.c 的 do_event() 中，补充对内核自陷事件的处理，即输出一句话并直接返回。

```

1 static __RegSet* do_event(__Event e, __RegSet* r) {
2     switch (e.event) {
3         case _EVENT_SYSCALL:
4             return do_syscall(r);
5         case _EVENT_TRAP:
6             printf("Event trap!\n");
7             return NULL;
8         default: panic("Unhandled event ID = %d", e.event);
9     }
10
11     return NULL;
12 }

```

然后需要设置中断描述符。在 asye.c 中声明入口函数 vecself()。

```

1 void vecsys();
2 void vecnull();
3 void vecself();

```

在 trap.S 中补充该函数的具体定义：除了 pushl 处改为"0x81"以外，其余不变。

```

1 #-----entry-----|-----errorcode-----|-----irq id-----|-----handler-----|
2 .globl vecsys;      vecsys:  pushl $0;  pushl $0x81; 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

```

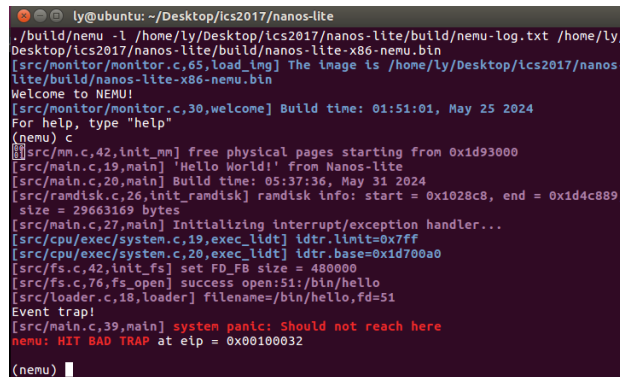
在 asye.c 的 `_ayse_init()` 函数中添加 0x81 的描述符，并且标注入口函数为 `vecself`，其余内容不变。

```

1 void _ayse_init(_RegSet*(*h)(_Event, _RegSet*)) {
2     // initialize IDT
3     for (unsigned int i = 0; i < NR_IRQ; i++) {
4         idt[i] = GATE(STS_TG32, KSEL(SEG_KCODE), vecnull, DPL_KERN);
5     }
6
7     // ----- system call -----
8     idt[0x80] = GATE(STS_TG32, KSEL(SEG_KCODE), vecsys, DPL_USER);
9     idt[0x81] = GATE(STS_TG32, KSEL(SEG_KCODE), vecself, DPL_USER);
10
11     set_idt(idt, sizeof(idt));
12
13     // register event handler
14     H = h;
15 }

```

运行程序后，能够看到屏幕输出了我们设置的打印信息，并且遇到了 BAD TRAP。



```

ly@ubuntu: ~/Desktop/ics2017/nanos-lite
./build/nemu -l /home/ly/Desktop/ics2017/nanos-lite/build/nemu-log.txt /home/ly/
Desktop/ics2017/nanos-lite/build/nanos-lite-x86-nemu.bin
[src/monitor/monitor.c,65,load_img] The image is /home/ly/Desktop/ics2017/nanos-
lite/build/nanos-lite-x86-nemu.bin
Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 01:51:01, May 25 2024
For help, type "help"
(nemu) c
[src/main.c,42,init_mm] free physical pages starting from 0x1d93000
[src/main.c,19,main] 'Hello World!' from Nanos-lite
[src/main.c,20,main] Build time: 05:37:36, May 31 2024
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x1d28c8, end = 0x1d4c889,
size = 29663169 bytes
[src/main.c,27,main] Initializing interrupt/exception handler...
[src/cpu/exec/system.c,19,exec_ldt] idtr.limit=0x7ff
[src/cpu/exec/system.c,20,exec_ldt] idtr.base=0x1d700a0
[src/fs.c,42,init_fs] set FD_FB size = 480000
[src/fs.c,76,fs_open] success open:51:/bin/hello
[src/loader.c,18,loader] filename=/bin/hello,fd=51
Event trap!
[src/main.c,39,main] system panic: Should not reach here
nemu: HIT BAD TRAP at eip = 0x00100032
(nemu)

```

3.2 实现上下文切换

目前 `schedule()` 只需要总是切换到第一个用户进程即可，即 `pcb[0]`。而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。

因此，我们首先需要实现 `nexus-am/am/arch/x86-nemu/src/pte.c` 中的 `_unmake()` 函数。该函数在加载程序时完成上下文的创建，因此，需要把入口的三个参数和指令寄存器入栈，然后初始化 `trap-frame`。

```

1 _RegSet *_umake(_Protect *p, _Area ustack, _Area kstack, void *entry, char *const
   argv[], char *const envp[]) {
2     extern void* memcpy(void *, const void *, int);
3     int arg1 = 0;
4     char *arg2 = NULL;

```

```

5  memcpy((void*)ustack.end - 4, (void*)arg2, 4);
6  memcpy((void*)ustack.end - 8, (void*)arg2, 4);
7  memcpy((void*)ustack.end - 12, (void*)arg1, 4);
8  memcpy((void*)ustack.end - 16, (void*)arg1, 4);
9
10 __RegSet tf;
11 tf.eflags = 0x02 | FL_IF;
12 tf.cs = 0;
13 tf.eip = (uintptr_t) entry;
14 void *ptf = (void*) (ustack.end - 16 - sizeof(__RegSet));
15 memcpy(ptf, (void*)&tf, sizeof(__RegSet));
16 return (__RegSet*) ptf;
17 }

```

完善 nanos-lite/src/proc.c 中的 schedule() 函数，作用为返回现场。

```

1 __RegSet* schedule(__RegSet *prev) {
2     if(current != NULL) {
3         current -> tf = prev;
4     }
5     current = &pcb[0];
6     Log("ptr = 0x%x\n", (uint32_t)current -> as.ptr);
7     __switch(&current -> as);
8     return current -> tf;
9 }

```

在 ./irq.c 中，修改 __EVENT_TRAP 事件，将原本的 return NULL 改为调用 schedule() 函数，如下所示。

```

1 extern __RegSet* schedule(__RegSet *prev);
2
3 static __RegSet* do_event(__Event e, __RegSet* r) {
4     switch (e.event) {
5         case __EVENT_SYSCALL:
6             return do_syscall(r);
7         case __EVENT_TRAP:
8             printf("Event trap!\n");
9             return schedule(r);
10        default: panic("Unhandled event ID = %d", e.event);
11    }
12
13    return NULL;
14 }

```

最后，需要在 trap.S 的 asm_trap() 函数中实现以下操作：从中断控制函数返回后，先把栈顶指针切换到新进程的 trap 帧，再根据 trapframe 恢复现场。

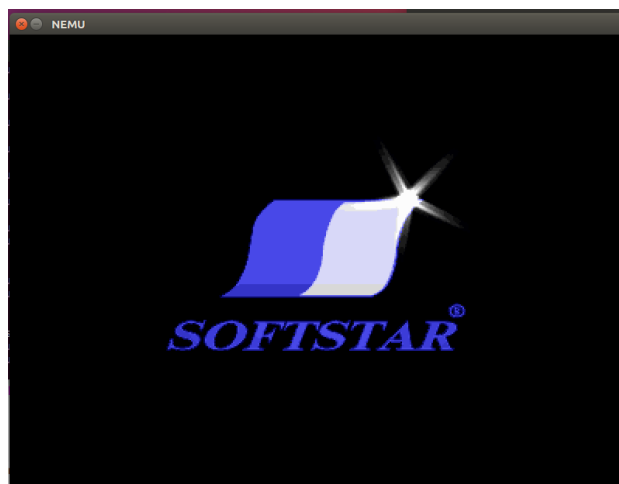
```

1 asm_trap:
2     pushal

```

```
3
4   pushl %esp
5   call irq_handle
6
7   # addl $4, %esp
8   movl %eax, %esp
9
10  popal
11  addl $8, %esp
12
13  iret
```

然后便可以通过内核自陷的方式运行《仙剑奇侠传》。



3.3 分时多任务：同时运行两个程序

在之前的工作中已实现虚拟内存及上下文切换机制，因此，nanos-lite 已经可以支持分时多任务。修改 main.c 中的内容，使其同时加载两个用户程序，如下所示。

```
1   init_fs();
2
3   extern void load_prog(const char *filename);
4   load_prog("/bin/pal");
5   load_prog("/bin/hello");
6
7   __trap();
8
9   panic("Should not reach here");
```

目前只允许最多一个需要更新画面的进程参与调度，这是因为多个这样的进程分时运行会导致画面被相互覆盖，影响画面输出的效果。

因此，需要修改 do_event() 的代码，在处理完系统调用后，调用 schedule() 函数并返回其现场。

首先修改 nanos-lite/src/proc.c 中的 schedule() 函数，使其轮流返回这两个程序的现场。修改后如下所示。

```

1 _RegSet* schedule(_RegSet *prev) {
2     if(current != NULL) {
3         current->tf = prev;
4     }
5     //current = &pcb[0];
6     current = (current == &pcb[0] ? &pcb[1] : &pcb[0]);
7     Log("ptr = 0x%x\n", (uint32_t)current->as.ptr);
8     _switch(&current->as);
9     return current->tf;
10 }

```

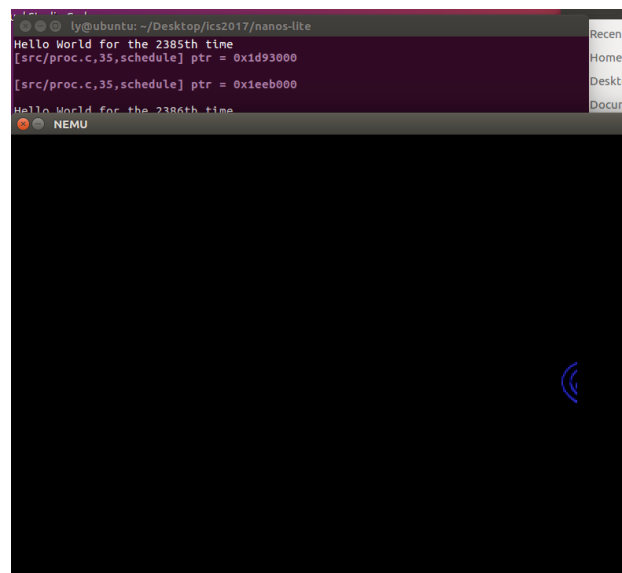
然后需要修改 do_event() 的代码，处理完 syscal 后，调用 schedule() 函数返回现场。

```

1 static _RegSet* do_event(_Event e, _RegSet* r) {
2     switch (e.event) {
3         case _EVENT_SYSCALL:
4             //return do_syscall(r);
5             do_syscall(r);
6             return schedule(r);
7         case _EVENT_TRAP:
8             printf("Event trap!\n");
9             return schedule(r);
10        default: panic("Unhandled event ID = %d", e.event);
11    }
12    return NULL;
13 }

```

启动 nemu，可以发现《仙剑奇侠传》与 hello 程序能够同时运行。



3.4 优先级调度

虽然已经实现了分时多任务运行，但《仙剑奇侠传》的运行速度明显下降了。因此，还可以通过 schedule() 函数调整这两个程序的调度比例，让《仙剑奇侠传》调度若干次，才让 hello 程序调度一次。

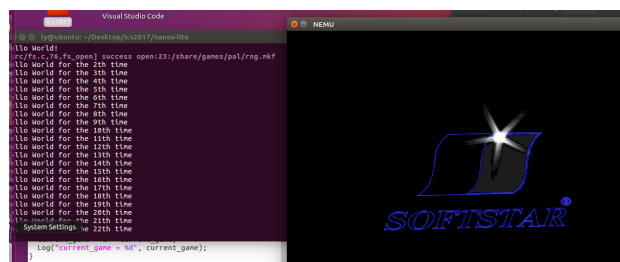
修改后的 `schedule()` 函数如下所示。

```

1 int current_game = 0;
2 void switch_current_game() {
3     current_game = 2 - current_game;
4     Log("current_game = %d", current_game);
5 }
6
7 _RegSet* schedule(_RegSet *prev) {
8     if(current != NULL) {
9         current -> tf = prev;
10    }
11    else {
12        current = &pcb[current_game];
13    }
14    static int num = 0;
15    static const int frequency = 1000;
16    if(current == &pcb[current_game]) {
17        num++;
18    }
19    else {
20        current = &pcb[current_game];
21    }
22    if(num == frequency) {
23        current = &pcb[1];
24        num = 0;
25    }
26    // current = (current == &pcb[0]? &pcb[1] : &pcb[0]);
27    // Log("ptr = 0x%x\n", (uint32_t)current -> as.ptr);
28    _switch(&current -> as);
29    return current -> tf;
30 }

```

再次运行程序，可以发现《仙剑奇侠传》的运行速度明显变快了。



4 阶段三：来自外部的声音

实现了通过 `schedule()` 进行进程的上下文切换以后，这套机制运行得十分顺利，却存在一些风险。假如该程序为死循环或恶意程序，则系统将永远无法触发系统调用，整个计算机的资源都会被该程序调用。因此，需要在系统中实现硬件中断机制。

4.1 问题：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?

【解答】: 如果发生中断嵌套, 可能会导致当前中断处理程序被中断, 从而造成处理过程错乱和数据的 inconsistency. 这可能会导致系统崩溃或数据丢失。

这种灾难性的后果可能以各种形式表现出来, 比如系统卡死、程序运行异常、数据丢失等。由于中断处理不当, 可能导致中断处理程序出现错误或无法正确恢复现场, 从而使系统出现严重问题。在最严重的情况下, 可能导致系统崩溃, 无法正常运行。

4.2 添加时钟中断

在 nemu 中, 只需要添加时钟中断即可模拟硬件中断。

4.2.1 在 NEMU 中添加 INTR

首先, 需要在 reg.h 中新增一个 bool 变量 INTR。

```
1  rtlreg_t  cs;
2  rtlreg_t  ds;
3  rtlreg_t  es;
4  uint32_t  CR0;
5  uint32_t  CR3;
6
7  bool INTR;
8  } CPU_state;
```

之后, 需要在 nemu/src/cpu/intr.c 中的 dev_raise_intr() 中将 INTR 引脚设置为高电平。

```
1  void dev_raise_intr() {
2      cpu.INTR = true;
3  }
```

在 nemu/src/cpu/exec.c 文件中的 exec_wrapper() 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。

```
1  #define TIME_IRQ 32
2
3  if(cpu.INTR & cpu.eflags.IF) {
4      cpu.INTR = false;
5      extern void raise_intr(uint8_t NO, vaddr_t ret_addr);
6      raise_intr(TIME_IRQ, cpu.eip);
7      update_eip();
8  }
```

之后, 需要修改 raise_intr() 中的代码, 在保存 EFLAGS 寄存器后, 将其 IF 位置为 0, 让处理器进入关中断状态。

```

1 memcpy(&t1, &cpu.eflags, sizeof(cpu.eflags));
2 rtl_li(&t0, t1);
3 rtl_push(&t0);
4 cpu.eflags.IF = 0;
5 rtl_push(&cpu.cs);
6 rtl_li(&t0, ret_addr);
7 rtl_push(&t0);

```

4.2.2 在软件上添加对时钟中断的处理

除此之外，还需要在 ASYE 中添加时钟中断的支持，将时钟中断打包成 `_EVENT_IRQ_TIME` 事件。

首先，需要在 `nexus-am/am/arch/x86-nemu/src/asye.c` 中声明时钟中断的入口函数 `vectime()`，并在 `asye_init()` 中添加 `int32` 的门描述符。

```

1 void vectime();
2
3 idt[32] = GATE(STS_TG32, KSEL(SEG_KCODE), vectime, DPL_USER);

```

之后，在 `asye.c` 的 `irq_handle()` 函数中添加 `_EVENT_IRQ_TIME` 事件。

```

1 _RegSet* irq_handle(_RegSet *tf) {
2     _RegSet *next = tf;
3     if (H) {
4         _Event ev;
5         switch (tf->irq) {
6             case 0x80: ev.event = _EVENT_SYSCALL; break;
7             case 0x81: ev.event = _EVENT_TRAP; break;
8             case 32: ev.event = _EVENT_IRQ_TIME; break;
9             default: ev.event = _EVENT_ERROR; break;
10        }
11
12        next = H(ev, tf);
13        if (next == NULL) {
14            next = tf;
15        }
16    }
17    return next;
18 }

```

之后，需要在 `nanos-lite/src/irq.c` 中，为 `do_event()` 函数添加对 `_EVENT_IRQ_TIME` 事件的操作：在收到时钟中断的信号后，直接调用 `schedule()` 进行调度，并输出 `log` 信息便于验证实验结果。

```

1 static _RegSet* do_event(_Event e, _RegSet* r) {
2     switch (e.event) {
3         case _EVENT_SYSCALL:
4             //return do_syscall(r);
5             do_syscall(r);

```

```

6     return schedule(r);
7 case _EVENT_TRAP:
8     printf("Event: trap!\n");
9     return schedule(r);
10 case _EVENT_IRQ_TIME:
11     Log("Event: IRQ_TIME!\n");
12     return schedule(r);
13 default: panic("Unhandled event ID = %d", e.event);
14 }
15
16 return NULL;
17 }

```

此外，还需要在 nexus-am/am/arch/x86-nemu/src/trap.S 中注册 vectime 函数。

```

1 .globl vectime;    vectime:  pushl $0;  pushl $32; jmp asm_trap

```

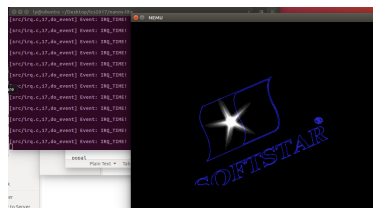
最后，为了可以让处理器在运行用户进程的时候响应时钟中断，还需要修改 nexus-am/am/arch/x86-nemu/src/pte.c_umake() 的代码，在构造现场的时候，设置正确的 EFLAGS。

```

1 _RegSet *_umake(_Protect *p, _Area ustack, _Area kstack, void *entry, char *const
   argv[], char *const envp[]) {
2     extern void* memcpy(void *, const void *, int);
3     int arg1 = 0;
4     char *arg2 = NULL;
5     memcpy((void*)ustack.end - 4, (void*)arg2, 4);
6     memcpy((void*)ustack.end - 8, (void*)arg2, 4);
7     memcpy((void*)ustack.end - 12, (void*)arg1, 4);
8     memcpy((void*)ustack.end - 16, (void*)arg1, 4);
9
10    _RegSet tf;
11    tf.eflags = 0x02 | FL_IF;
12    tf.cs = 0;
13    tf.eip = (uintptr_t) entry;
14    void *ptf = (void*) (ustack.end - 16 - sizeof(_RegSet));
15    memcpy(ptf, (void*)&tf, sizeof(_RegSet));
16    return (_RegSet*) ptf;
17 }

```

至此，时钟中断已成功实现。重新启动程序，发现能够正常运行，说明实验成功。



5 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。

【解答】:

程序开始运行时，nanos-lite 会先运行 main.c: 初始化 mm，将设置的地址作为空闲物理页首地址；初始化磁盘和设备；初始化中断异常以及文件系统。

之后，加载需要运行的程序，即 pal(仙剑奇侠传) 和 hello 程序。

```

1  int main() {
2  #ifdef HAS_PTE
3      init_mm();
4  #endif
5
6      Log("'Hello World!' from Nanos-lite");
7      Log("Build time: %s, %s", __TIME__, __DATE__);
8
9      init_ramdisk();
10
11     init_device();
12
13 #ifdef HAS_ASYE
14     Log("Initializing interrupt/exception handler...");
15     init_irq();
16 #endif
17
18     init_fs();
19
20     extern void load_prog(const char *filename);
21     load_prog("/bin/pal");
22     load_prog("/bin/hello");
23
24     __trap();
25
26     panic("Should not reach here");
27 }

```

加载每一个进程时，会将上下文保存在进程控制块 PCB 中。即，AM 会通过 __protect 函数初始化一个虚拟地址空间并对其进行映射，将上下文信息保存在 PCB 的 as 中。之后，Nanos-lite 的 loader() 函数会逐页加载用户程序，再通过 AM 的 __unmake() 函数创建用户进程的现场。至此，虽然虚拟地址的入口相同，但所有进程都可以加载在不同的物理空间中，从而解决进程的物理地址会被相互覆盖的问题。

AM 中的 __protect() 函数:

```

1  void __protect(__Protect *p) {
2      PDE *updir = (PDE*)(palloc_f());
3      p->ptr = updir;
4      // map kernel space

```

```

5  for (int i = 0; i < NR_PDE; i++) {
6      updir[i] = kpdirs[i];
7  }
8
9  p->area.start = (void*)0x8000000;
10 p->area.end = (void*)0xc0000000;
11 }

```

Nanos-lite 中的 loader() 函数:

```

1  uintptr_t loader(_Protect *as, const char *filename) {
2      int fd = fs_open(filename, 0, 0);
3      Log("filename=%s,fd=%d",filename,fd);
4      fs_read(fd, DEFAULT_ENTRY, fs_filesz(fd));
5      fs_close(fd);
6      return (uintptr_t)DEFAULT_ENTRY;
7  }

```

之后,操作系统内核会自陷,中断入口为 0x81,在 AM 的 asye.c 中定义。AM 根据 trap.S 中定义的 vectrap(0x81) 进行相应的处理,跳转到了 asm_trap 函数中。而 asm_trap 函数的作用为将栈顶指针切换回堆栈上,从而恢复进程现场。

```

1  asm_trap:
2      pushal
3
4      pushl %esp
5      call irq_handle
6
7      addl $4, %esp
8
9      popal
10     addl $8, %esp
11
12     iret

```

运行游戏时,由于添加了时钟中断,每隔一小段时间便触发 timer_intr(),把 CPU 中的 INTR 引脚设置为 1。每次执行完一条指令后,如果 INTR 为 1 且中断允许位 IF 为 0 (中断未打开),则会执行一次时钟中断,将《仙剑奇侠传》的上下文信息保存,并跳转到处理时钟中断的函数中。其中,该处理函数会使用 schedule() 进行调度,即跳转到 _switch() 函数中。

```

1  _RegSet* schedule(_RegSet *prev) {
2      if(current != NULL) {
3          current->tf = prev;
4      }
5      else {
6          current = &pcb[current_game];
7      }
8      static int num = 0;
9      static const int frequency = 1000;

```

```

10  if(current == &pcb[current_game]) {
11      num++;
12  }
13  else {
14      current = &pcb[current_game];
15  }
16  if(num == frequency) {
17      current = &pcb[1];
18      num = 0;
19  }
20  __switch(&current -> as);
21  return current -> tf;
22  }

```

而 __switch() 函数中的操作为设置页表页目录项基地址，切换地址空间。之后，schedule() 函数会返回到 hello 程序的陷阱帧，并切换到 hello 程序的地址空间中，再次切换上下文。

```

1  void __switch(__Protect *p) {
2      set_cr3(p->ptr);
3  }

```

之后，各个函数返回，直至运行到 trap.S 的最后一条指令，最终成功实现一次中断调用。因此，虽然多任务的虚拟地址空间会有重叠，但通过这种中断机制，可以切换不同程序的地址空间并保存好上下文，分时多任务功能也就能顺利进行。

6 展示计算机系统

在这一小节中，需要让 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，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，需要添加 current_game 变量，并添加函数，用于切换记录当前的应用程序。

```

1  int current_game = 0;
2  void switch_current_game() {
3      current_game = 2 - current_game;
4      Log("current_game = %d", current_game);
5  }

```

修改后的 schedule() 函数如下所示。

```

1  __RegSet* schedule(__RegSet *prev) {
2      if(current != NULL) {
3          current -> tf = prev;
4      }
5      else {
6          current = &pcb[current_game];

```

```

7  }
8  static int num = 0;
9  static const int frequency = 1000;
10 if(current == &pcb[current_game]) {
11     num++;
12 }
13 else {
14     current = &pcb[current_game];
15 }
16 if(num == frequency) {
17     current = &pcb[1];
18     num = 0;
19 }
20 // current = (current == &pcb[0]? &pcb[1] : &pcb[0]);
21 // Log("ptr = 0x%x\n", (uint32_t)current -> as.ptr);
22 _switch(&current -> as);
23 return current -> tf;
24 }

```

此外，还需要在 nanos-lite/src/device.c 的 events_read() 函数中添加对 F12 按键的检测和处理。

```

1  if(down && key == _KEY_F12) {
2      extern void switch_current_game();
3      switch_current_game();
4      Log("key down: _KEY_F12, switch current game0!");
5  }

```

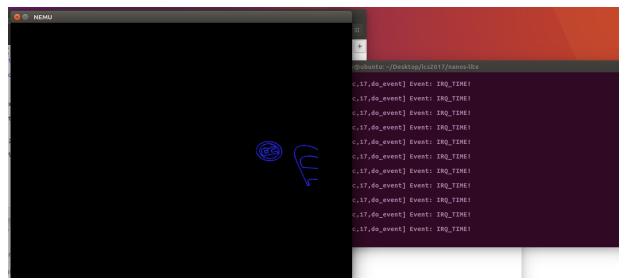
最后，在 main.c 中修改 loader() 函数，使其加载 pal、hello、videotest 三个用户程序。

```

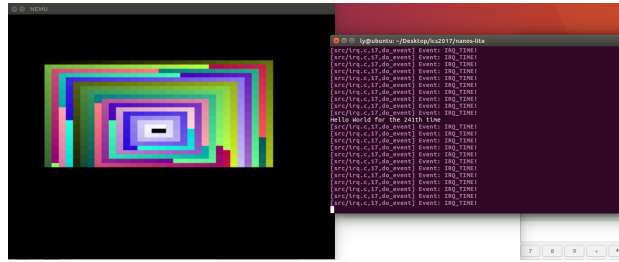
1  extern void load_prog(const char *filename);
2  //load_prog("/bin/dummy");
3  load_prog("/bin/pal");
4  load_prog("/bin/hello");
5  load_prog("/bin/videotest");

```

运行 nemu，未按 F12 前的画面如下所示。



按 F12 后，切换成 hello 和 videotest 一起运行。



7 BUG 汇总

7.1 dummy 无法正常运行

补充 loader() 函数后, dummy 程序仍不能运行, 仍显示缺页错误。对照实验手册后发现, 是未使用 load_prog() 函数进行程序加载。

需要修改的代码为:

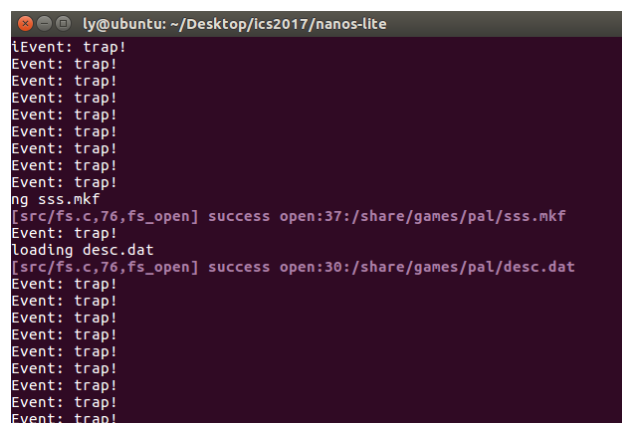
```
1 uint32_t entry = loader(NULL, "/bin/dummy");
2 ((void (*)(void))entry)();
```

修改后的代码为:

```
1 extern void load_prog(const char *filename);
2 load_prog("/bin/dummy");
```

7.2 实现时钟中断后不断显示 event: trap

如下所示。



经过排查后发现, 是 trap.S 对 vectime 进行注册时, 忘记把 pushl 的内容由 \$0x81 改为 \$32。修改后可正常运行。