



南开大学  
Nankai University

南 开 大 学

计 算 机 学 院

系统综合课程设计实验报告

---

**PA4 实验报告 + PA5 浮点数部分实验报告**

---

周辰霏 1712991

年级：2017 级

专业：计算机科学与技术

2020 年 6 月 11 日

## 目录

<b>一、 概述</b>	<b>1</b>
(一) 实验目的 . . . . .	1
(二) 实验内容 . . . . .	1
<b>二、 阶段一</b>	<b>1</b>
(一) 加入 PTE . . . . .	1
(二) 用户程序运行在分页机制上 . . . . .	3
(三) 分页机制上运行仙剑 . . . . .	4
<b>三、 阶段二</b>	<b>5</b>
(一) 上下文切换 . . . . .	5
(二) 分时多任务 . . . . .	7
(三) 时间中断 . . . . .	7
<b>四、 阶段三</b>	<b>8</b>
(一) 展示系统切换 . . . . .	8
<b>五、 PA5: 浮点数的支持</b>	<b>9</b>
<b>六、 遇到的 bug 及解决</b>	<b>11</b>
<b>七、 思考题 &amp;&amp; 必答题</b>	<b>12</b>
(一) 思考题 . . . . .	12
(二) 必答题 . . . . .	12
<b>八、 总结</b>	<b>13</b>

## 一、 概述

### (一) 实验目的

- 实现内存管理单元以及虚拟内存的分页机制
- 实现上下文切换并可以对多任务进程进行切换

分时多任务实际上就是最早的 UNIX 批处理系统，在程序员发出指令或者 OS 自身可以进行任务（进程）的切换。而 PA4 就是在内存系统上最终实现这一系统。

### (二) 实验内容

PA3 实验涉及现代指令系统的实现、抽象机器 AM 的原理与应用、输入输出设备三部分

1. 实现分页机制
2. 实现上下文切换
3. 实现真正的分时多任务，并可以实施任务切换

## 二、 阶段一

### (一) 加入 PTE

我们在 main.c 里面取消对 HAS\_PTE 的注释，这样就可以让 nanos 对内存的分页机制进行初始化，主要是分页的内存管理单元初始化。首先在 nemu 中添加 CR0 和 CR3 寄存器并对其初始化，以及操作他们相对应的指令，以及对虚拟地址的读写函数。最后还需要实现分页的地址转换函数，这样分页机制就算完成了。

#### 分页机制

```
1 // nanos-lite / src / main.c
2 #define HAS_PTE
3 // nemu / include / cpu / reg.h
4 #include "memory / mmu.h"
5 CR0 cr0;
6 CR3 cr3;
7 // nemu / src / monitor / monitor.c
8 cpu.cr0.val = 0x60000011;
9 // nemu / src / cpu / exec / system.c
10 make_EHelper(mov_r2cr) {
11     // TODO();
12     switch (id_dest->reg) {
13         case 0: cpu.cr0.val = id_src->val; break;
14         case 3: cpu.cr3.val = id_src->val; break;
15         default: Assert(0, "Shoule reach here for NO cr%d", id_dest->reg); break;
16     }
17     print_asm("movl %s,%scr%d", reg_name(id_src->reg, 4), id_dest->reg);
18 }
19
```

```

20 make_EHelper(mov_cr2r) {
21     //TODO();
22     switch (id_src->reg) {
23         case 0: operand_write(id_dest, &cpu.cr0.val); break;
24         case 3: operand_write(id_dest, &cpu.cr3.val); break;
25         default: Assert(0, "Shoule reach here for NO cr%d", id_dest->reg); break;
26     }
27
28     print_asm("movl %%cr%d,%%%s", id_src->reg, reg_name(id_dest->reg, 4));
29
30 #ifdef DIFF_TEST
31     diff_test_skip_qemu();
32 #endif
33 }
34 //nemu/src/memory/memory.c
35 // 以下内容源自于x86.h
36 #define PDX(va)      (((uint32_t)(va) >> 22) & 0x3ff)
37 #define PTX(va)      (((uint32_t)(va) >> 12) & 0x3ff)
38 #define OFF(va)      ((uint32_t)(va) & 0xfff)
39 #define PTE_ADDR(pte) ((uint32_t)(pte) & ~0xfff)
40 uint32_t vaddr_read(vaddr_t addr, int len) {
41     //PAGE_MASK = 0xfff
42     if (((addr) + (len) - 1) & ~PAGE_MASK) != ((addr) & ~PAGE_MASK) { // data
43         cross the page boundary
44         uint32_t data = 0;
45         for(int i=0; i<len; i++){
46             paddr_t paddr = page_translate(addr + i, false);
47             data += (paddr_read(paddr, 1)) << 8*i;
48         }
49         return data;
50         // assert(0);
51     }
52     else {
53         paddr_t paddr = page_translate(addr, false);
54         return paddr_read(paddr, len);
55     }
56 }
57 void vaddr_write(vaddr_t addr, int len, uint32_t data) {
58     if (((addr) + (len) - 1) & ~PAGE_MASK) != ((addr) & ~PAGE_MASK) { // data
59         cross the page boundary
60         for(int i=0; i<len; i++){ // len 最大为4
61             paddr_t paddr = page_translate(addr + i, true);
62             paddr_write(paddr, 1, data >> 8*i);
63         }
64         // assert(0);
65     }
66     else {

```

```

66     paddr_t paddr = page_translate(addr, true);
67     paddr_write(paddr, len, data);
68 }
69 }
70 paddr_t page_translate(vaddr_t addr, bool wlr0) {
71     // aka page_walk
72     PDE pde, *pgdir;
73     PTE pte, *pgtab;
74     if (cpu.cr0.protect_enable && cpu.cr0.paging) {
75         pgdir = (PDE *) (PTE_ADDR(cpu.cr3.val)); // cr3 存放20位的基址作为页目录入口
76         pde.val = paddr_read((paddr_t)&pgdir[PDX(addr)], 4);
77         assert(pde.present); // 存在
78         pde.accessed = 1;
79         pgtab = (PTE *) (PTE_ADDR(pde.val)); // 页目录存放20位的基址作为页表入口
80         pte.val = paddr_read((paddr_t)&pgtab[PTX(addr)], 4);
81         assert(pte.present);
82         pte.accessed = 1;
83         pte.dirty = wlr0 ? 1 : pte.dirty; // 写则置脏位
84         // pte高20位和线性地址低12位拼接成真实地址
85         return PTE_ADDR(pte.val) | OFF(addr);
86     }
87     return addr;
88 }

```

这之后，仙剑就可以看似成功的运行在 NEMU 上了。

## (二) 用户程序运行在分页机制上

为了让用户程序正确的在分页机制上运行，即在其分配的虚拟空间上运行，首先为了避免用户程序的虚拟地址空间与内核相互重叠要将链接地址-Ttext 参数改为 0x8048000，同时 DEFAULT\_ENTRY 的值也要做出相应修改。最后在 main 里面调用 load\_prog 进行加载。而为了成功调用加载程序的函数，首先要实现 \_map() 函数，即页映射功能，并在 loader 中加入新的有关 mm 的内容。

### 加载

```

1 // navy-apps/Makefile.compile
2 LDFLAGS += -Ttext 0x08048000
3 // nanos-lite/src/loader.c
4 #define DEFAULT_ENTRY ((void *)0x08048000)
5 // nanos-lite/src/main.c
6 load_prog("/bin/dummy");
7 // nexus-am/am/arch/x86-nemu/src/pte.c
8 void _map(_Protect *p, void *va, void *pa) {
9     PDE *pgdir = p->ptr;
10    PDE *pde = &pgdir[PDX(va)];
11    PTE *pgtab;
12    if (*pde & PTE_P) { // present
13        pgtab = (PTE *)PTE_ADDR(*pde);

```

```

14 }
15 else { // 映射过程中发现需要申请新的页表
16     pgtab = (PTE *)palloc_f();
17     *pde = PTE_ADDR(pgtab) | PTE_P;
18 }
19 pgtab[PTX(va)] = PTE_ADDR(pa) | PTE_P;
20 }
21 // nanos-lite/src/loader.c
22 uintptr_t loader(_Protect *as, const char *filename) {
23     //TODO();
24     /*size_t size = get_ramdisk_size();
25     ramdisk_read(DEFAULT_ENTRY, 0, size);*/
26     int fd = fs_open(filename, 0, 0);
27     int bytes = fs_filesz(fd);
28     Log("Load [%d] %s with size: %d", fd, filename, bytes);
29     //fs_read(fd, DEFAULT_ENTRY, fs_filesz(fd));
30     void *pa, *va = DEFAULT_ENTRY;
31     while (bytes > 0) {
32         pa = new_page(); // 申请空闲物理页
33         _map(as, va, pa); // 该物理页映射到用户程序虚拟地址空间
34         fs_read(fd, pa, PGSIZE); // 读一页文件到该物理页
35         va += PGSIZE;
36         bytes -= PGSIZE;
37     }
38     fs_close(fd);
39     return (uintptr_t)DEFAULT_ENTRY;
40 }

```

即可看到如下 Good Trap 运行结果

```

Welcome to NEMU!
[src/monitor/monitor.c,30,welcome] Build time: 16:51:57, May 10 2020
For help, type "help"
(nemu) c
[src/mm.c,40,init_mm] free physical pages starting from 0x1d91000
[src/main.c,19,main] 'Hello World!' from Nanos-lite
[src/main.c,20,main] Build time: 08:26:58, Jun 6 2020
[src/ramdisk.c,26,init_ramdisk] ramdisk info: start = 0x102b40, end = 0x1d4cac1,
size = 29663105 bytes
[src/main.c,27,main] Initializing interrupt/exception handler...
[src/fs.c,39,fs_open] Pathname: /bin/dummy
[src/loader.c,19,loader] Load [16] /bin/dummy with size: 17952
[src/fs.c,39,fs_open] Pathname: /bin/bmptest
[src/loader.c,19,loader] Load [14] /bin/bmptest with size: 29544
[src/irq.c,7,do_event] trap event!
nemu: HIT GOOD TRAP at eip = 0x00100032

```

图 1: dummy 加载成功

### (三) 分页机制上运行仙剑

由于这时我们需要真正分配堆并将其映射到虚拟空间，所以前面 PA3 实现的一半 brk 需要重新进行修改使其进行真正的分配而非次次默认分配成功。首先是 mm\_brk 的修改使其通过 \_map 分配堆为 [current->max\_brk, new\_brk) 这一部分内存空间并将其映射到地址空间 current->as 中

(其他部分框架代码已给出)。由于这一改动我们的堆区并不是一定可以分配成功，所以系统调用部分也要进行修改。

brk

```

1 // nanos-lite/src/mm.c
2 #define K4(va) (((uint32_t)(va)+0xfff) & ~0xfff)
3 int mm_brk(uint32_t new_brk) {
4     if (current->cur_brk == 0)
5         current->cur_brk = current->max_brk = new_brk;
6     else {
7         if (new_brk > current->max_brk) {
8             // TODO: map memory region [current->max_brk, new_brk)
9             // into address space current->as
10            uintptr_t va = K4(current->max_brk);
11            while(va < new_brk) {
12                _map(&current->as, (void *)va, (void *)new_page());
13                va += PGSIZE;
14            }
15            current->max_brk = new_brk;
16        }
17        current->cur_brk = new_brk;
18    }
19    return 0;
20 }
21 // nanos-lite/src/syscall.c
22 case SYS_brk: SYSCALL_ARG1(r) = mm_brk(a[1]); break; //SYSCALL_ARG1(r) = 0;
    break;

```

## 三、 阶段二

### (一) 上下文切换

上下文切换实际上就是将当前运行程序状态(寄存器内存映射空间等)保存下来再加载另一个部分的程序状态进来，而这部分就和 OS 中提及的 PCB 相关，PCB 就保存了各个进程的信息，上下文切换中的回复现场就需要用到 PCB 根据其内容进行恢复。

所以我们需要先有一个现场才能进行上述的上下文恢复，\_umake 即是如此。且由于 nemu 没有特权级的区别，所以在 ustack 的底部初始化一个以 entry 为返回地址的陷阱帧即可，就是构造一个有着通用寄存器以及 eflags 等标志位设置的陷阱帧。再之后需要同系统调用的实现一样在 irq 中包装事件并对其进行处理，同时还需要在 trap.S 中实现 trap 的跳转和调用。同时注释掉 proc 里面原来用于切换的代码改用 umake 初始化。

在现场完成之后我们在进入 trap 陷阱中之后，需要借助调度器，即 schedule 函数进行上下文的切换，实际上就是指向不同的 PCB 交由 CPU 执行。

上下文切换

```

1 // nexus-am/am/arch/x86-nemu/src/pte.c
2 _RegSet *_umake(_Protect *p, _Area ustack, _Area kstack, void *entry, char *
    const argv[], char *const envp[]) {

```

```

3 // _umake 将ustack底部初始化一个entry为返回地址的陷阱帧
4 uint32_t *ptr = ustack.end;
5 // navyapps程序入口函数_start的 栈帧，即8个通用寄存器
6 for (int i = 0; i < 8; i++) {
7     *ptr = 0x0;
8     ptr--;
9 }
10 // 陷阱帧，包括栈帧的8个通用寄存器
11 *ptr = 0x202;
12 ptr--; // eflags, 即IF置1即可
13 *ptr = 0x8;
14 ptr--; // cs 为了diff test
15 *ptr = (uint32_t)entry;
16 ptr--; // eip
17 *ptr = 0x0;
18 ptr--; // error code
19 *ptr = 0x81;
20 ptr--; // irq id
21 for (int i = 0; i < 8; i++) {
22     *ptr = 0x0;
23     ptr--;
24 }
25 ptr++;
26 return (_RegSet *)ptr; // 将会记录到tf
27 }
28 // nanos-lite/src/main.c
29 _trap();
30 // nexus-am/am/arch/x86-nemu/src/asye.c
31 void vectrap();
32 case 0x81: ev.event = _EVENT_TRAP; break;
33 idt[0x81] = GATE(STS_TG32, KSEL(SEG_KCODE), vectrap, DPL_USER);
34 // nexus-am/am/arch/x86-nemu/src/trap.S
35 .globl vectrap; vectrap: pushl $0; pushl $0x81; jmp asm_trap
36 // nanos-lite/src/irq.c
37 case _EVENT_TRAP: return schedule(r);
38 // nanos-lite/src/proc.c
39 int count = 0;
40 extern int current_game;
41 _RegSet* schedule(_RegSet *prev) {
42     // save the context pointer
43     current->tf = prev;
44
45     current = &pcb[0];
46     count++;
47
48     _switch(&current->as);
49     return current->tf;
50 }

```



然后就是真的可以在虚拟内存空间——分页机制上运行仙剑了。并且同时我们的分时多任务也由于上下文切换的实现而完成了。

## (二) 分时多任务

这时候基本内容都已经完成了，在 proc 中对 schedule 进行多任务切换的设定然后在 main 中加载多个内容即可。

### 分时多任务

```
1 //nanos-lite/src/proc.c
2 current = (current == &pcb[0] ? &pcb[1] : &pcb[0]);
3 //nanos-lite/src/main.c
4 load_prog("/bin/pal");
5 load_prog("/bin/hello");
```

可以看到如下结果在缓慢运行仙剑的时候 hello 也在输出，但是何必为难年幼的 NEMU 呢，所以后面除非必要，对仙剑的测试都只运行这一个。

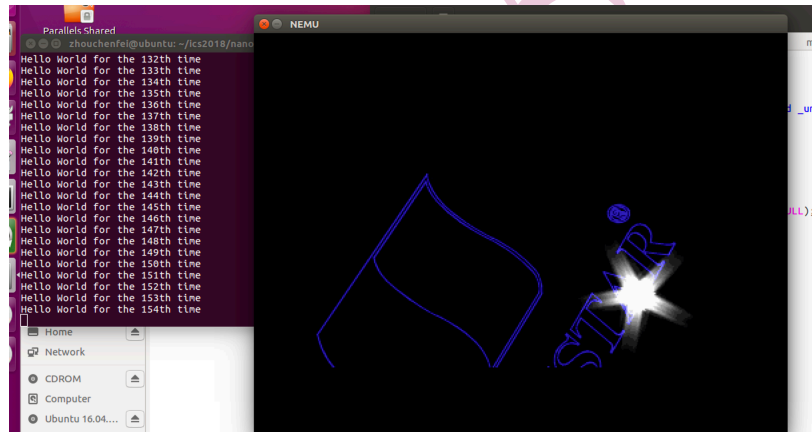


图 2: 分时多任务

## (三) 时间中断

时间中断本质上还是中断，所以实现思路仍然与 PA3 的中断一致。

### 时间中断

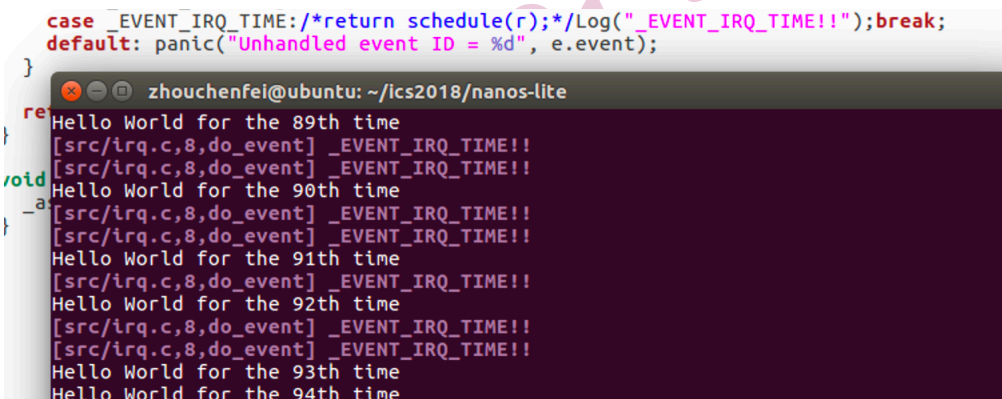
```
1 //nemu/include/cpu/reg.h
2 // 加入INTR引脚
3 bool INTR;
4 //nemu/src/cpu/intr.c
5 // 设置为高电平
6 void dev_raise_intr() {
7     cpu.INTR = true;
8 }
9 //nemu/src/cpu/exec/exec.c
10 // 然后在真正的执行单元中轮询检查是否有时钟中断
11 #define TIMER_IRQ 0x32
12 if (cpu.INTR & cpu.eflags.IF) {
```

```

13     cpu.INTR = false;
14     raise_intr(TIMER_IRQ, cpu.eip);
15     update_eip();
16 }
17 // 同时在异常处理中对IF进行设置,使当前为关中断状态
18 //nemu/src/cpu/intr.c
19 cpu.eflags.IF = 0;
20 //nexus-am/am/arch/x86-nemu/src/pte.c
21 //设置eflags
22 *ptr = 0x202;    ptr--; //eflags,即IF置1即可
23 //之后就是中断异常一样的处理
24 //nexus-am/am/arch/x86-nemu/src/asye.c
25 //异常号
26 idt[0x32] = GATE(STS_TG32, KSEL(SEG_KCODE), vectime, DPL_USER);
27 case 0x32: ev.event = _EVENT_IRQ_TIME; break;
28 //nanos-lite/src/irq.c
29 case _EVENT_IRQ_TIME: /*Log("_EVENT_IRQ_TIME!!");*/return schedule(r);

```

如果取消 Log 的注释就可以看到如下结果



```

case _EVENT_IRQ_TIME:/*return schedule(r);*/Log("_EVENT_IRQ_TIME!!");break;
default: panic("Unhandled event ID = %d", e.event);
}
re
Hello World for the 89th time
[src/irq.c,8,do_event] _EVENT_IRQ_TIME!!
void [src/irq.c,8,do_event] _EVENT_IRQ_TIME!!
-al Hello World for the 90th time
[src/irq.c,8,do_event] _EVENT_IRQ_TIME!!
[src/irq.c,8,do_event] _EVENT_IRQ_TIME!!
Hello World for the 91th time
[src/irq.c,8,do_event] _EVENT_IRQ_TIME!!
Hello World for the 92th time
[src/irq.c,8,do_event] _EVENT_IRQ_TIME!!
[src/irq.c,8,do_event] _EVENT_IRQ_TIME!!
Hello World for the 93th time
Hello World for the 94th time

```

图 3: 时间中断

## 四、 阶段三

### (一) 展示系统切换

首先用 main 加载 videotest, 并且在 eventsread(键盘事件) 中对 F12 按键进行识别, 在 F12 按下的时候切换仙剑奇侠传和 videotest。

展示

```

1 //nanos-lite/src/main.c
2 load_prog("/bin/videotest");
3 //nanos-lite/src/device.c
4 int current_game = 0;
5 size_t events_read(void *buf, size_t len) {
6     //return 0;
7     int key = _read_key();

```

```

8     bool down = false;
9     if (key & 0x8000) {
10         key ^= 0x8000;
11         down = true;
12     }
13     if (key == _KEY_NONE) {
14         unsigned long t = _uptime();
15         sprintf(buf, "t %d\n", t);
16     }
17     else {
18         sprintf(buf, "%s %s\n", down ? "kd" : "ku", keyname[key]);
19         if(key == 13 && down) { // F12 DOWN
20             current_game = (current_game == 0 ? 1 : 0);
21         }
22         Log("Get key: %d %s %s\n", key, keyname[key], down ? "down" :
            "up");
23     }
24     return strlen(buf);
25 }
26 // nanos-lite/src/proc.c
27 extern int current_game;
28 current = (current_game == 0 ? &pcb[0] : &pcb[2]);

```

之后就可以成功看到切换的结果啦。

## 五、 PA5: 浮点数的支持

如果不管 JIT 内容的话 PA5 简直不怎么需要时间，看懂指导书每个函数都是一句话的事情，就是仙剑运行的很慢然后存档还不好使，所以就很绝望，后来换到 native 上快了点，还没来得及截图，虚拟机坏了。

### 浮点数

```

1 // navy-apps/apps/pal/include/FLOAT.h
2 static inline int F2int(FLOAT a) {
3     if(a&0x80000000 == 0)
4         return a/0x10000;
5     else // 负数
6         return -((-a)/0x10000);
7 }
8
9 static inline FLOAT int2F(int a) {
10     return a>0 ? a*0x10000 : -((-a)*0x10000);
11 }
12
13 static inline FLOAT F_mul_int(FLOAT a, int b) {
14     return a*b;
15 }
16

```

```
17 static inline FLOAT F_div_int(FLOAT a, int b) {
18     return a/b;
19 }
20 //navy-apps/apps/pal/src/FLOAT/FLOAT.c
21 //解释了一大堆的部分，实际上把小数点后进行处理即可
22 FLOAT F_mul_F(FLOAT a, FLOAT b) {
23     return a*b/0x10000;
24 }
25
26 FLOAT F_div_F(FLOAT a, FLOAT b) {
27     return a*b*0x10000;
28 }
29 FLOAT f2F(float a) {
30     /* You should figure out how to convert a' into FLOAT without* introducing x87 floating
        point instructions. Else you can* not run this code in NEMU before implementing x87 floating* point
        instructions, which is contrary to our expectation.** Hint: The bit representation of
        a' is already on the
31     * stack. How do you retrieve it to another variable without
32     * performing arithmetic operations on it directly?
33     */
34     return a>0 ? a*0x10000: -((-a)*0x10000);
35 }
36
37 FLOAT Fabs(FLOAT a) {
38     if(a&0x80000000 == 0)
39         return a;
40     else
41         return -a;
42 }
```

然后就能战斗了。



图 4: 战斗

perf 的评测我本来弄了，但是虚拟机崩了没时间再弄了，就这样吧，反正也没来得及分析出什么有用的东西。

## 六、 遇到的 bug 及解决

没什么特别致命的 bug，致命的是我的电脑。

- ☒ 和 PA3 那个 serial 一样要先实现 loader 才行，我不知道是实验书真的顺序有问题还是我的理解哪里有问题
- ☒ PA3 那个 logo 可以正确加载了，在实现了分页加载之后，大概就是超限了吧，不懂非要整这么个坑的意义在哪
- ☒ 直接 make run 切换出了一堆链接还有编译 werror 的错，还是 clean 一遍再说
- ☒ 时间中断作为中断也是要写在 IDT 表里的，80、81 写好就不记得这个了
- ☒ 跑的再慢也不能切 native，因为没法 debug。。
- ☐ 以上问题其实还好，尚在我可以接受的范围内，但是上一次因为我不当操作导致的虚拟机崩溃我还可以忍受，这次吃个饭回来就崩的开不开机是怎么回事 orz，如下图，眼看着就差个截图就可以交报告，然后什么都没了，PA3-PA5 浮点数的代码还没来得及备份，完全重新来。

```
[ 102.1605011] xhci_hcd 0000:00:1d.6: Setup ERROR: setup address command for slot 1.
[ 102.3683231] xhci_hcd 0000:00:1d.6: Setup ERROR: setup address command for slot 1.
[ 102.5765361] usb 4-1: device not accepting address 2, error -22
[ 102.7324731] xhci_hcd 0000:00:1d.6: Setup ERROR: setup address command for slot 1.
[ 102.9406101] xhci_hcd 0000:00:1d.6: Setup ERROR: setup address command for slot 1.
[ 103.1480821] usb 4-1: device not accepting address 2, error -22
[ 103.3058531] xhci_hcd 0000:00:1d.6: Setup ERROR: setup address command for slot 1.
[ 103.5126931] xhci_hcd 0000:00:1d.6: Setup ERROR: setup address command for slot 1.
[ 103.7204101] usb 4-1: device not accepting address 2, error -22
[ 103.8766191] xhci_hcd 0000:00:1d.6: Setup ERROR: setup address command for slot 1.
[ 104.0852131] xhci_hcd 0000:00:1d.6: Setup ERROR: setup address command for slot 1.
[ 104.2928561] usb 4-1: device not accepting address 2, error -22
```

图 5: 死机

## 七、 思考题 && 必答题

### (一) 思考题

1. 如果不同的进程共享同一个栈空间, 会发生什么呢?

上下文会错乱吧。只有这部分独立才可以保证函数调用等正常执行。

2. 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?

因为在查页目录和页表项的作用是生成 20 位的物理页号, 最终与虚拟地址的后 12 位 (页内地址) 构成物理地址。所以这里没必要只需要使用 20 位基址, 最后再和页内地址拼接即可, 可以节省内存的开销。

3. 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?

必须, 不能使用。因为页目录和页表项均储存在内存中, 到内存中寻址需要物理地址。如果表项 (包括 CR3) 中的基地址使用虚拟地址, 那么查表得过程就要先通过 MMU 进行虚拟地址和物理地址得转化, 这个过程中却要使用到表项, 这显然是自相矛盾的。

4. 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?

页表大小过大, 对内存来说是一种浪费, 不容易 hit, 或者说 hit 所需的时间长, 开销大。

5. 空指针真的是 '空' 的吗?

空指针其实还是指向一个地址的, c 语言中 NULL 的本质是 (void \*) 0, 所以空指针指向的是 0 这个地址, 而这个地址不与其他实际地址冲突

### (二) 必答题

以下以外的必答题均为代码实现。

1. 分时多任务的具体过程请结合代码, 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。

时间中断每到一定时间触发一次, 然后引起硬件中断 yield, 中断前会保存上下文, 然后 yield 会使用 schedule 进行调度, 调度器选择下一个将要被执行的程序, 如仙剑切到 hello, 选择之后根据 pcb 的内容对该 pcb 的上下文进行载入恢复, 然后 cpu 继续执行, 如此往复。

## 八、 总结

没有什么好总结的，没写报告前写完 PA5 浮点数的时候是非常开心的，因为很早就完成了任务，然后开始写报告并且又一次运行看看有没有 bug 的时候虚拟机崩掉的时候就只想赶紧弄好赶紧扔掉完事，虽然确实成长了许多，也确实找到了许多丢三落四自己作死导致的天坑，但是远没有吃个饭电脑充着电虚拟机就突然黑屏再也打不开来的坑大，因为这次机制特殊用的虚拟机所以没有很好的把代码同 github 同步，然后因为想着可能还有问题就不去备份代码调好了再备份，然后就多花了不知道多少时间恢复代码，是不是和以前一样有没有更多的 bug 也没有精力去管了。所幸前面的截图还能找到记录，报告不用被耽误，且由于 pa4 开始变为 2018 但代码保留的是 2019 的就很头疼，所以想借 jyb 同学的 docker 镜像一用（省去环境配置时间），结果 Mac 的 ssh 又开始抽风。本想着等代码彻底复原了再提交报告，现在看来是没有什么可能了，毕竟照着写好的报告粘都能粘出问题。祝贺 PA 完结，大三下死亡生活结束。

终于隔了两天算是把代码复原了吧，最终版本用的最简单古早的 2018 代码，实在不想折腾各种乱七八糟的东西了，Mac 上 Xquartz 连不上，用 vmware 在 2019 的显示上又有一点点小问题。所以最后用的 Mac 上的虚拟机 ParallelDesktop 安装了 ubuntu 系统并运行 nemu 代码。发现了即使流程大致记得代码完全粘贴复现也还是会暴露不少问题，还有一些前面忽视的细节的小坑。一言蔽之，对指导书的理解还需加深。