PA4 实验报告

14307130078 张博洋

一、实验进度

完成了 PA4 所有内容（包括选做内容“性能调优”和“移植仙剑奇侠传”）。

二、部分选答题

1. 异常与函数调用

函数调用需要保存的信息是由调用约定所决定的。一般来说，调用约定会决定哪些寄存器的值在调用前后不能变化（即由被调用函数保存），哪些寄存器的值可以变化（即由调用者保存）。通过这样的调用约定，编译器在生成函数调用的指令时，可以明确地知道哪些寄存器的值需要保存，哪些不需要。然而异常处理却需要保存更多信息，这是因为，异常的发生不像函数调用那样是事先安排好的，而是几乎不可预测的。因此，在进行异常处理时，操作系统并不知道哪些寄存器需要保存，哪些不需要，只能全部保存下来（包括 CS:EIP，EFLAGS，所有通用寄存器），在处理完毕之后再恢复现场，这样才能确保用户程序运行正常。

2. 特殊的原因

由于程序执行流程需要跳转到异常处理程序，因此在发生异常后，CS:EIP 的值会指向异常处理程序，原来的 CS:EIP 值就被覆盖了。为了能在异常处理结束后，能将执行流程跳转回原始位置，必须由硬件在跳转到异常处理程序前保存 CS:EIP 的值。

而 EFLAGS 需要由硬件保存，是因为 EFLAGS 中存有与程序状态、特权级紧密相关的一些标志位（例如 IF，IOPL 等）。由于跳入异常处理程序的过程中需要修改程序状态、特权级等信息，若硬件不在此过程中保存 EFLAGS 的值，这些信息会丢失，异常处理完毕后就不能完整恢复现场，导致错误。

3. 神奇的 longjmp

如果不使用 longjmp，可以采用检查函数返回值或者检查全局变量的方式来完成，但是这样会多出很多 if 判断语句，不仅降低了程序的可维护性，也降低了程序

的性能。如果要我实现 setjmp 和 longjmp，我会在 setjmp 时把通用寄存器（包括 ESP、EBP 等）和程序计数器 EIP 保存到 jmp_buf 中，当调用 longjmp 时再恢复回去，这样依赖栈的状态和 EIP 都恢复到调用时的状态了。

4. 重新设置 GDT

因为 GDTR 中存放指向 GDT 的指针是线性地址，因此在开启分页并切换到用户程序后，在内核最开始阶段使用的 GDT（在 start.S 中定义，使用的是低地址）将消失不见。所以必须在内核中重新设置 GDT（此时应设置它为 0xC0000000 以上的内核中的地址）。否则在异常产生时，会发生需要读取 GDT 却读取不到的错误。

5. 诡异的代码

在执行这条指令之前，陷阱帧（trap frame）已经在栈上构造好了，由于栈的生长方向是从高地址到低地址，因此此时 ESP 正好就指向陷阱帧的首地址。根据 i386 手册的说明，PUSH ESP 是把这条指令执行前的 ESP 值压栈。因此在这条指令执行后，陷阱帧的首地址就被压到了栈上，而其后紧接着一条 CALL irq_handle 指令，就把这个陷阱帧的首地址(指针)作为函数调用的参数，传递给了 irq_handle 函数做进一步处理。

6. 不可缓存(uncachable)的内存区域

如果从缓存中读而不是直接读，则可能会导致读到的数据与实际数据不一致的情况，因为可能在设备工作期间，对应内存区域的数据已经变化，而缓存却毫不知情，因此实际数据已经更新了而缓存中的数据还是旧数据。写缓存的情况大致相同，由于写入的数据被缓存，导致新的数据没有到达设备。

7. 理解 volatile 关键字

反汇编结果如下：

带 volatile 关键字

```
00000000 <fun>:
   0:   c6 05 00 90 04 08 00    movb   $0x0,0x8049000
   7:   90                      nop
   8:   0f b6 05 00 90 04 08    movzbl 0x8049000,%eax
   f:   3c ff                   cmp    $0xff,%al
  11:   75 f5                   jne    8 <fun+0x8>
  13:   c6 05 00 90 04 08 33    movb   $0x33,0x8049000
  1a:   c6 05 00 90 04 08 34    movb   $0x34,0x8049000
  21:   c6 05 00 90 04 08 86    movb   $0x86,0x8049000
  28:   c3                      ret
```

不带 volatile 关键字

```
00000000 <fun>:
   0:   c6 05 00 90 04 08 00    movb   $0x0,0x8049000
   7:   eb fe                   jmp    7 <fun+0x7>
```

可以看出，在没用 volatile 关键字的情况下，编译器直接认为 0x8049000 处的字节值不会变化，因此推测出 while 循环根本不会退出，直接使用了一条 jmp 指令实现了死循环功能，并且省略了之后的赋值语句（因为根本不会被执行到）。但是若是使用了 volatile 关键字，编译器会认为 0x8049000 处的字节值可能会在程序执行过程中发生变化，因此就不能做出很强的推测，该读写内存是，就只能按照程序原本的意思去读写内存，也不能直接把 while 循环判定为死循环了。

如果 0x8049000 最终被映射到寄存器，去掉 volatile 会导致一些本该执行的内存写（例如看似冗余却有设备意义的赋值语句）被编译器优化没了，导致错误。

8. 内存映射 I/O 的保护机制

根据 i386 手册 8.1.2 节（第 146 页）的说明，内存映射 I/O 对应的内存区域，像其他的内存区域一样，遵从保护模式的管理。因此，操作系统只要设置使用页表设置好对应内存区域的权限，就可以起到存储保护的作用了。

9. 灾难性的后果

根据假设，由于中断信息固定存放于 0x1000 的位置，因此当嵌套的第二个中断到来时，第一个中断存放的陷阱帧就被新的陷阱帧覆盖了。这样一来，虽然第二个中断的处理不会发生问题，但是一旦第二个中断处理完毕，程序返回到第一个中断处理程序后，由于陷阱帧已被破坏，第一个中断处理程序访问到的内容将完全不对。即使中断程序不由于数据破坏而发生崩溃，陷阱帧中所记录的返回地址也被覆盖为自身中某一条指令的地址，因此根本就无法返回到用户程序了。

10. 如何检测多个键同时被按下

实际上，多个键同时被按下，总是有“某个键先按下”，“另一个键再被按下”的先后顺序。因此判断某个键是否被按下，只需要判断这个键收到了通码，却没收到断码即可。而由于多个键之间互相并不冲突，判断多个键同时被按下也按相同的办法即可。

11. 提高磁盘读写的效率

可以直接在 IN 指令中使用内存寻址，避免数据在寄存器与内存直接拷贝；或者是使用 REP INSD 指令代替循环，可以快速读取字符串到内存；或是使用内存映射 IO，直接做内存拷贝，不需要 IN 类型的指令就可完成读取；或是采用 DMA 技术，直接让设备写数据到内存，无需 CPU 中间参与。

12. 神奇的调色板

由于显存中每个像素存储的是颜色索引，所以在实现渐入或渐出效果时，不需要修改显存中的数据，而只需要把调色板中每种颜色都变亮或变暗即可。在 NEMU-PAL 中，游戏的帧率会使用同一种颜色（调色板索引）绘制在屏幕左上角，因此可以通过观察帧率数字的颜色的变化来观察调色板中颜色亮暗的变化。

13. 让随机数更“随机”

由于计算机使用的是伪随机数，因此为了让游戏中随机数看起来更随机，需要使生成伪随机数的随机种子在每次游戏开始前都不相同。

通常的做法是，使用当前的系统时间作为随机数种子。但是 NEMU 并没有提供“获取时间”的功能。一种解决办法是，添加一种 NEMU 的内置功能（即 NEMU TRAP 指令）来使得游戏可以获得当前系统时间。

实际上还有很多可行的做法。其中一种就是利用中断的不可预测性。游戏可以记录中断到来的“时间”（此处“时间”可以是游戏主逻辑循环的循环次数等具有“时间”性质的计数器，并不代表真实时间），把它作为随机种子，产生随机数。

14. 奇怪的关中断

由于 tick 变量可能被时钟中断处理程序改变。假设正在对 tick 值进行处理的逻辑中发生了时钟中断，tick 值被改变，这很有可能导致程序对于 tick 的处理发生不一致情况。例如在刚进入"if (now == tick) { ... }"这个判断语句之后发生了时钟中断，导致 tick 值发生了变化，自然这个判断条件就不成立了，然而此时继续执行很有可能导致错误。为了避免这种不一致的情况出现，在对 tick 进行处理的时候，最好关闭中断。

15. 性能瓶颈的来源

NEMU 为了使指令执行过程简单清晰，所有指令都是逐条解释执行，没有类似 QEMU 的动态翻译功能，也不会使用硬件提供的虚拟化功能，因此模拟执行一条指令需要耗费数十倍到数千倍的代价。

为了测试这几种指令执行方式之间的区别，我选择了一个计算密集型程序作为测试样例，分别在真实机器、NEMU、开启硬件虚拟化的 QEMU、开启动态翻译的 QEMU、关闭动态翻译的 QEMU 内运行，测试程序的执行时间，测试结果如下：

平台	运行时间	相对速度
真实机器	0.186s	1.0000x
QEMU（开启硬件虚拟化）	0.190s	0.9789x
QEMU（开启动态翻译）	1.614s	0.1152x
NEMU	22.905s	0.0081x
QEMU（关闭动态翻译）	25.089s	0.0074x

测试结果表明，虚拟机中速度最快的是开启硬件虚拟化的 QEMU，速度几乎与真实机器无异。其次是开启动态翻译的 QEMU，速度大致在真实机器的十分之一左右。最后是 NEMU 和关闭动态翻译的 QEMU，速度大致在真实机器的百分之一以下。

显然，NEMU 选择的解释执行模式是速度最慢的一种，如果不改变这种模式，运行速度很难有更多的提高了（NEMU 已经运行得比关闭动态翻译的 QEMU 快了）。

16. 读写文件的具体过程

首先，用户程序调用 fprintf 函数，它会对字符串和数据进行格式化并写入到用户程序自身 libc 内部的缓冲区内，libc 内部会使用 fp 对应的文件描述符进行 write 系统调用。随后，用户程序陷入内核，内核会根据文件描述符找到对应的文件，把内容写入到文件中。最后，系统调用完毕，控制流回到用户程序中，在一层一层地返回到调用 fread 处。这就完成了把字符串写入磁盘的功能。

17. 不再神秘的秘技

根据前两个秘技中"先把钱用少，再用一用，钱就会到上限"的现象，可以推测这与无符号整数运算溢出有关。这可能是由于游戏作者在这两处游戏逻辑中没有判断钱是否够用，而直接对钱数做了减法。当一个较小的无符号数，减掉一个比它大的数时，会发生下溢，现象就是负数结果的补码直接被当做无符号数显示和处理了。

第三个秘籍可能是由于游戏作者在游戏逻辑处理过程中对法术、人物、等级的编码压缩到了一个数中（例如十位和个位表示人物等级，而百位表示人物编号）。而作者在"使用金蚕王"这个逻辑中没有对边界条件进行判断，使得人物等级突破了 99 级上限，进位到人物编号那一位了，从而出现了某一人物学习到其他人物的法术的情况。

三、必答题

游戏是如何工作的

以键盘中断为例，整个键盘按键中断的产生与处理过程由 nemu、kernel、game 三个部分配合完成。

1. 中断的产生

在 NEMU 进行初始化的时候，NEMU 通过 setitimer()函数让操作系统每隔 10 毫秒发送一个 SIGVTALRM 信号给 NEMU 自身，并且向操作系统注册了信号处理函数 timer_sig_handler()。也就是说，这个函数每隔 10 毫秒就会被调用一次。（此部分代码位于 nemu/src/device/sdl.c 的 init_sdl()函数中）

timer_sig_handler()最终会通过调用 device_update()函数完成对键盘操作的处理。当按下键盘按键后，SDL 库会收到一个事件，在下一次调用 device_update()函数时，该函数会拉取并处理 SDL 库中的事件，这部分代码如下：

nemu/src/device/sdl.c: device_update()

```
SDL_Event event;
while(SDL_PollEvent(&event)) { // 向 SDL 库拉取一个事件
    // If a key was pressed
    uint32_t sym = event.key.keysym.sym;
    if( event.type == SDL_KEYDOWN ) {
        // 若该事件是“按下键”事件
        keyboard_intr(sym2scancode[sym >> 8][sym & 0xff]);
    }
    else if( event.type == SDL_KEYUP ) {
        // 若该事件是“抬起键”事件
        keyboard_intr(sym2scancode[sym >> 8][sym & 0xff] | 0x80);
    }
    // 以下无关代码略 ...
}
```

可以看到，一旦检测到一个按键，device_update()函数就会调用 keyboard_intr()函数对按键进行处理。

nemu/src/device/keyboard.c: keyboard_intr()

```
void keyboard_intr(uint8_t scancode) {
    // 若 NEMU 正在执行并且按键状态为“没有新按键”
    if(nemu_state == RUNNING && newkey == false) {
        // 将扫描码记录到 IO 端口对应的内存区域中
        i8042_data_port_base[0] = scancode;
        // 引发一次中断
        i8259_raise_intr(KEYBOARD_IRQ);
        // 将按键状态设为“有新按键按下”
        newkey = true;
    }
}
```

可以看到，keyboard_intr()函数在进行了一些必要的处理后，最终使用 KEYBOARD_IRQ 这个中断编号作为参数调用了 i8259_raise_intr()函数，在 NEMU 中引发一次中断。

i8259_raise_intr()函数在经过了一些判优逻辑的处理后，最终调用了do_i8259()函数，其内容如下：

nemu/src/device/i8269.c: do_i8259()

```
static void do_i8259() {
    // 取得当前最高优先级的中断的中断号
    int8_t master_irq = master.highest_irq;
    if(master_irq == NO_INTR) {
        cpu.INTR = false; // 若是没有中断，将 CPU 的 INTR 引脚置 0
        return;
    }
    else if(master_irq == 2) {
        // 判断中断是否来自级联的芯片
        assert(slave.highest_irq != NO_INTR);
        master_irq = 8 + slave.highest_irq;
    }
    // 设置好中断号
    intr_NO = master_irq + IRQ_BASE;
    // 将 CPU 的 INTR 引脚置 1，表示有中断到来
    cpu.INTR = true;
}
```

至此，中断的产生过程（即 cpu.INTR 引脚的修改过程）已经完全清楚了。

2．中断的处理（NEMU 部分）

指令模拟循环位于 cpu_exec()函数中，当每解释执行完毕一条指令后，就会通过一个判断语句检查 INTR 引脚。代码如下：

nemu/src/monitor/cpu-exec.c: cpu_exec()

```
// 若 cpu.INTR 引脚被置 1（即有中断请求等待处理）
// 并且 EFLAGS 中 IF 也被置 1（即允许中断）
if (cpu.INTR && cpu.EFLAGS.IF) {
    // 取得中断号
    uint32_t intr_no = i8259_query_intr();
    // 通知 i8259 中断已被处理
    i8259_ack_intr();
    // 调用 raise_intr 将虚拟机内的控制流迁往中断处理程序
    raise_intr(intr_no);
}
```

这部分代码最终调用 raise_intr()函数，将虚拟机内的控制流迁往中断处理程序，它的代码如下。

nemu/src/cpu/intr/intr.c: raise_intr()

该部分代码为自己编写，并非 NEMU 自带

```
void raise_intr(uint8_t NO) {
    /* Trigger an interrupt/exception with ``NO''.
    * That is, use ``NO'' to index the IDT.
    */
    // 将 EFLAGS、CS、EIP 压栈
    PUSH_DWORD(cpu.EFLAGS.EFLAGS);
    PUSH_DWORD((uint32_t) cpu.CS);
    PUSH_DWORD(cpu.EIP);
```

```
    // 根据中断号读取 IDT 表，计算中断处理程序的地址
    unsigned long long desc;
    uint32_t desc_data[2];
    desc_data[0] = lnaddr_read(cpu.IDTR + NO * 8, 4); // little e
ndian
    desc_data[1] = lnaddr_read(cpu.IDTR + NO * 8 + 4, 4);
    memcpy(&desc, desc_data, 8);
    int next_cs = GET_GATE_SELECTOR(desc);
    int next_eip = GET_GATE_OFFSET(desc);
    // 将虚拟机内的控制流迁往中断处理程序
    load_segment(R_CS, next_cs);
    cpu.EIP = next_eip;
    // 判断门的类型
    int type = desc >> (16 + 16 + 3 + 5) & 7;
    if (type == 6) {
        // 若是穿过中断门，需要把 IF 置零
        cpu.EFLAGS.IF = 0;
    }
    // 最后通过 longjmp，把 NEMU 自己的控制器迁往 cpu_exec()函数中
    // 此后，cpu_exec()函数便将开始执行中断处理程序的代码了
    /* Jump back to cpu_exec() */
    longjmp(jbuf, 1);
    panic("shouldn't reach here");
}
```

至此，NEMU 内处理引发一次中断的代码已经完全解释清楚了。

总的来说，硬件会在键盘按下（或抬起）一个按键的时候，产生一个中断。这时，正在 NEMU 里运行的程序会被打断，控制流由中断处理程序接管。

3. 中断的处理（kernel 部分）

在内核初始化时，init_idt()函数会填写好 IDT 表，并它的首地址装入 IDTR。这样一来，硬件在发生中断时，就可以根据 IDTR 中的内容来把控制流交给内核中的中断处理程序，以进行中断处理。（此部分代码位于 kernel/src/irq/idt.c 文件中）

当键盘中断到来时，硬件根据 IDT 中内容，把控制流交给 irq1 处继续执行。irq1 处代码如下：

kernel/src/irq/do_irq.S

```
.globl irq1;
irq1: // irq1 的入口点，此即键盘中断处理程序
    pushl $0; // 将 errorcode（值为 0）压栈
    pushl $1001; // 将键盘中断的内部代码（值为 1001）压栈
    jmp asm_do_irq // 跳转到中断处理公共代码 asm_do_irq 处

// 中间无关代码略 ...

asm_do_irq:
    pushal // 保存所有通用寄存器
    pushl %esp // 将在栈上构造出的陷阱帧（trap frame）首地址压栈
    call irq_handle // 调用 irq_handle 函数（陷阱帧的首地址为调用参数）
```

```
    addl $4, %esp // 将参数从栈中弹出
    popal // 恢复通用寄存器的内容
    addl $8, %esp // 将 errorcode 和中断内部代码从栈中弹出
    iret // 从中断处理程序中返回，从被中断打断的地方继续执行
```

可以看出，这部分中断处理代码现在栈上构造出了陷阱帧，然后以它为参数调用了 irq_handle()函数，进行进一步的处理。在调用该函数后，又进行了恢复中断现场的工作。

irq_handle()是内核中负责中断处理的函数，它的代码如下：

kernel/src/irq/irq_handle.c: irq_handle()

```
void irq_handle(TrapFrame *tf) {
    // 从陷阱帧中取得中断的内部代码
    int irq = tf->irq;
    // 进行中断类别的判断
    if (irq < 0) {
        panic("Unhandled exception!");
    } else if (irq == 0x80) {
        // 若是 0x80 则表示是系统调用，调用 do_syscall()完成系统调用功能
        do_syscall(tf);
    } else if (irq < 1000) {
        panic("Unexpected exception #%d at eip = %x", irq, tf->ei
p);
    } else if (irq >= 1000) {
        // 若是硬件产生的中断（例如键盘中断）
        // 首先取得 IRQ 编号
        int irq_id = irq - 1000;
        assert(irq_id < NR_HARD_INTR);
        // 找到对应的处理程序链表
        struct IRQ_t *f = handles[irq_id];
        // 循环调用链表里的每一个处理程序
        while (f != NULL) { /* call handlers one by one */
            f->routine();
            f = f->next;
        }
    }
}
```

可以看到，键盘中断到来是，这个函数会找到对应的链表（这个链表是事先初始化好的），调用其中的每一个函数，这样就完成了中断的处理过程。至此，内核中对于中断的处理已经完全清楚了。

4. 中断的处理（game 部分）

在游戏程序的初始化阶段，会调用其游戏代码内的 add_irq_handle()函数，这个函数会通过 0 号系统调用，向内核注册键盘中断的处理程序 keyboard_event()，即把它的地址加入到对应的链表当中。这样一来，当内核每次遇到键盘中断时，就会调用一次 keyboard_event()函数。（此部分代码位于 game/src/common/main.c 文件中）

keyboard_event()负责从 IO 端口读入数据，并且调用游戏逻辑中关于键盘处理的函数。它的代码如下：

game/src/typing/keyboard.c: keyboard_event()

```
void
keyboard_event() {
    // 从 0x60 端口（即键盘的 IO 端口）读入一个字节（即扫描码）
    key_code = in_byte(0x60);
    // 以扫描码为参数，调用游戏逻辑中处理按键的函数
    press_key(key_code);
}
```

至于 press_key() 内则是一些游戏相关的逻辑：

game/src/typing/keyboard.c: press_key()

```
/* a-z 对应的键盘扫描码 */
static int letter_code[] = {
    30, 48, 46, 32, 18, 33, 34, 35, 23, 36,
    37, 38, 50, 49, 24, 25, 16, 19, 31, 20,
    22, 47, 17, 45, 21, 44
};
/* 对应键按下的标志位 */
static bool letter_pressed[26];

void
press_key(int scan_code) {
    int i;
    for (i = 0; i < 26; i ++) {
        // 循环判断每个字母的扫描码
        if (letter_code[i] == scan_code) {
            // 若当前字母的扫描码等于中断处理函数传入的扫描码
            // 则将对应字符设置为“按下”
            letter_pressed[i] = true;
        }
    }
}
```

至此，游戏内关于键盘中断的处理已经完全清楚了。

整个游戏是以时钟中断为驱动而进行的，若是没有时钟中断到来，则游戏会在“hlt”指令上一直停机，游戏逻辑无法继续。而键盘中断主要负责人与计算机的交互，若是没有键盘中断，游戏就无法检测用户的按键情况，没有办法与人产生交互。

从刚才的代码分析中可以看到，中断的产生与处理过程与硬件（nemu）、软件（kernel 和 game）密切相关。在这些环环相扣的一系列步骤中，缺少任何一个都会使中断功能失灵。