PA4 实验报告

一、PA4 完成情况简述

1、已完成讲义要求必做全部内容。

二、部分蓝框思考题回答

1、进行函数调用的时候也需要保存调用者的状态：返回地址，而进行异常处理之前却要保存更多的信息. 尝试对比它们，并思考两者保存信息不同是什么原因造成的.

答： 因为函数调用不需要进行特权级转换，而在异常处理过程中需要从用户态转换到内核态，它们代码段特权级不同导致需要转换 CS 寄存器，因而需要在转换前由硬件保存 CS 寄存器,且在陷入内核态时很有可能要使用到通用寄存器、标志寄存器以及 PC 这些都需要在陷入内核态之前保存，并在返回用户态时恢复。

2、诡异的代码

答： 因为在后面需要用到%esp 寄存器，故需将它保存在栈中。

三、必答题回答

1、游戏是如何工作的？

答：游戏工作的过程大部分与外部中断相关。就键盘中断而言，总的来说，当用户按下一个按键时，键盘接口会得到一个代表该按键的键盘扫描码（通码），该扫描码说明了按键在键盘上的位置，该扫描码被送到主板上键盘接口所在芯片的寄存器中，该寄存器的端口地址为 0x60；同理，在松开按键时，也会产生一个扫描码（断码），断码说明了松开键在键盘上的位置，同样的，断码也将被送入 0x60 端口中。紧接着，当键盘的输入到达 0x60 端口时，i8259 芯片（中断控制器）通过中断判优电路查询当前优先级最高的中断，若当前键盘中断优先级最高，则芯片就向 CPU 发出键盘中断，并给出键盘中断的中断类型码，CPU 检测到该中断信息后，如果 IF == 1，则响应中断，引发中断过程，转去执行键盘中断服务程序，最后通知中断控制器本次中断结束并实现中断返回。

结合代码来说，上述当用户按下按键或松开按键时，.就会调用中断控制器（i8259.c）中的 i8259_i8259_intr()函数，该函数根据当前中断请求状态选择优先级最高的中断，并将中断号记录在 intr_NO 中，并把并把 cpu.INTR 设为 true 以通知 CPU 响应该中断，若此时 CPU 检测到 IF == 1 且 cpu.INTR == true 则执行如下代码

```
if(cpu.INTR & cpu.eflags.IF) {
    uint32_t intr_no = i8259_query_intr();
    i8259_ack_intr();
    raise_intr(intr_no);
```

}

其中 i8259_query_intr()函数给出中断类号,i8259_ack_intr()函数向中断控制器发出确认信息，以消除刚刚发出的中断请求信息，接着 raise_intr()函数会保存 eflags，eip，cs 等断点信息，并根据中断类型号在中断描述符表中读出相应的中断描述符，装入中断描述符中的段描述符和段内偏移量，再将段描述符中的基地址与偏移量相加得到出口地址（即中断服务程序的地址）去执行，首先跳转到 kernel 中的 asm_do_irq.s 中，保存通用寄存器等现场信息（pusha 指令），接着调用 irq_handle 函数，由 irq 函数判断是软中断（系统调用）还是外部中断，若为外部中断则转到相应的中断服务例程去执行，完成后通过 iret 指令恢复断点信息。最终通过 longjump 返回到 cpu_exec()中继续执行。

具体到打字小游戏，在 game/common 的 main 函数代码中添加 add_irq_handle(0,keyboard_event);为游戏注册键盘中断处理函数，若我们已经注册了键盘中断处理程序，则在上述的 irq_handle 函数中键盘中断处理程序会被被内核调用，这样游戏代码就可以通过中断来控制游戏的逻辑了。

四、遇到的问题及思考

1、问题：PA4 遇到的问题较多，所以就谈谈几个困扰我比较久的问题。

（1）lgdt 及 lidt 指令的错误，因为在 PA3 中仅有直接寻址

方式的 lgdt 指令，所以在 PA3 中我就直接用 instr_fetch()函数读取 lgdt 操作码后若干字节当做 gdt 首地址，而在 PA4 中出现了形如 lgdt (%eax)的指令，导致引发缺页及地址越界的情况。刚开始出现这种情况的时候我首先怀疑的是我的分段分页机制实现是否发生了错误，反复排查发现没有错，当时我犯了一个重大的错误因为 log.txt 只显示到 lgdt （%eax）指令的前一条，所以我总是纠结于那条指令，直到后来我反汇编了 kernel 比对后发现了罪魁祸首 lgdt。

（2）call 指令，这是 PA2 的遗留问题当时写 call_rm 指令的时候想当然直接默认指令的长度为 2，于是在运行 hello 程序的时候出现了长度为 3 的 call_rm(如 ff 57 24 call *0x24(%edi))指令，导致在执行 call 指令后跳过了 push %ebp 这条指令，使得栈帧破坏，引发错误。解决这个问题我用了比较简单粗暴的方法就是反复 log.txt 与反汇编代码，最终发现了这个 bug，不过这也花了我好几个小时的时间。

（3）repz ret 指令导致 eip 多跳了一步引发错误，这条指令的错误与我的 ret 指令实现有误相关，我在实现 ret 指令的时候在 make_helper 函数中直接返回的指令长度为 0，这正好与我的 call 指令配对，因为我写的 call 指令在 push %eip 的时候就是直接把 call 指令的下一条指令压入栈中，但在 repz ret 中这样的实现就出现了问题，因为执行完 repz ret 后会返回 1 使得 eip 多加了 1 从而产生错误。

（4）最最重大的问题就非 sdl 莫属了，首先是对这个库比较陌生，一时无从下手，其次是就算你写好了代码，但是一旦出现 bug 你将无从下手，所以对于 sdl 库的部分实现都是一步一步摸爬滚打过来的。

2、感想与收获：总体来说，PA4 的难度较大，尤其是第三阶段的重写 sdl 库部分，刚开始确实让我无从下手，在这种情况下我首先想到能不能在 sdl 库源码中找到破解困境的办法，于是我在官网上下载了一份 sdl 源码，翻开源码的那一刻其实我的内心是崩溃的，因为源码与我们要实现的 sdl 有一定的区别，但这也给了我一定的启发，再加上维基百科上的一些解释以及大神们的指导，终于我有了一点眉目，但写出来的代码仍然 bug 不断，这期间的调试过程难度十分巨大，因为想要定位到出错的位置变得十分困难，但最终通过努力还是解决了问题。虽然 PA4 的过程是充满挑战和痛苦的，但完成之后有着一种无以言表的成就感。