OSLab2 系统调用

实验报告

吴澄杰　　151220122　　151220122@smail.nju.edu.cn　　2017年4月1日

一、实验进度

此次实验，我将内核和游戏的代码分开并且分别编译，在kernel中从磁盘上加载游戏，并且设置好环境后，通过iret指令从ring0进入ring3的游戏中。我实现了一系列库函数，进而通过操作系统中实现的系统调用为游戏提供了运行的接口。

二、实验心得

1、理解陷阱帧与特权级切换过程

个人认为本次实验最难的地方就是从ring3进入ring0的过程。从ring3进入ring0需要用到iret指令。在处理中断的过程中，CPU会先保存当前的陷阱帧，进入内核态执行，完毕后通过iret指令，利用保存好的陷阱帧恢复原来的进程。从kernel进入游戏就利用了此过程的“一半”。在kernel中人为地构造游戏的陷阱帧，并且通过iret指令，让CPU从构造出的这一个陷阱帧中恢复，就可以顺利地进入ring3的游戏了。

由于有了ring0和ring3的差别，因此系统中断的处理程序也要稍加改变。需要人为地额外设置段寄存器。

从kernel进入游戏部分的代码和注释如下：

```
/* set tss.esp0 to current kernel stack position, where trap frame will be built*/
asm volatile("movl %%esp, %0" : "=r"(tss.esp0));

asm volatile("movl %0, %%eax" : : "r"(elf->e_entry));

asm volatile("pushl %0" : : "i"(SELECTOR_USER(4))); //change to user's segments
asm volatile("popl %ds");
asm volatile("pushl %0" : : "i"(SELECTOR_USER(4)));
asm volatile("popl %es");
asm volatile("pushl %0" : : "i"(SELECTOR_USER(4)));
asm volatile("popl %fs");
asm volatile("pushl %0" : : "i"(SELECTOR_USER(4)));
asm volatile("popl %gs");

asm volatile("pushl %0" : : "i"(SELECTOR_USER(4))); //push user's ss
asm volatile("pushl %0" : : "i"(USER_STACK));       //push user's esp
asm volatile("pushfl");                             //push eflags
asm volatile("pushl %0" : : "i"(SELECTOR_USER(3))); //push user's cs
asm volatile("pushl %eax");                         //push user's eip

/* Here we go! */
asm volatile("iret");
```

首先，在tss中保存了当前内核栈的%esp的值。然后将%cs，%ss以外的段寄存器从内核态切换到用户态。接着是构造陷阱帧的过程，将用户程序的%ss，%esp，%eflags，%cs，%eip压栈。其中，%eip中的压入的值，就是约定好的用户程序（游戏）的入口位置。因此，在iret指令执行后，就切换到了游戏运行。

2、关于tss

每次在处理中断时，从用户态陷入内核态，都会访问tss寄存器，将内核的%esp和%ss寄存器加载上来。因此，kernel必需在加载用户程序之前设置好tss寄存器，使得中断到来时能够正确地陷入内核态处理。

3、关于关中断

在中断处理函数asm_do_irq中，第一步必需首先用cli关中断。若不关中断，则非常容易在处理中断的过程中发生新的中断。由于tss.esp的值不是在中断处理的一开始就进行的，因此新的中断到来之时可能tss.esp的值仍旧是没有更新的。所以新的中断的陷阱帧构筑在旧的中断的陷阱帧相同的位置，造成一种嵌套。这样的话，就会永远陷在恢复栈帧的过程中出不来，陷入了死循环。我曾经在这个Bug中困惑了很久。因此，必需一进入内核就关闭中断。

由于用iret指令恢复到原进程，CPU是会恢复原eflags寄存器的，因此也会恢复到原进程中开中断的状态。因此，再使用sti打开中断是不必要并且会额外增加风险的。

asm_do_irq的代码如下：

```
asm_do_irq:
    cli     #avoid twice interruptions, eflags has been stored and iret will restore it,
            #at the same time, allow interruptions again

    pushl %ds
    pushl %es
    pushl %fs
    pushl %gs
    pushal

    movw $SELECTOR_KERNEL(SEG_KERNEL_DATA), %ax
    movw %ax, %ds
    movw %ax, %es
    movw %ax, %gs
    movw %ax, %fs

    pushl %esp
    call irq_handle

    movl current, %eax
    movl (%eax),%esp

    popal
    popl %gs
    popl %fs
    popl %es
    popl %ds
    addl $8, %esp

    iret
```

4、分段式存储管理

在进入内核后，必需重新填制GDT。在此次实验中，我选择用GDT表项来代替LDT，因此，同时还要初始化一些GDT表项来给以后的用户进程使用。此外，也必需准备好中断向量表，来应对可能到来的中断。

5、从ring0进入ring3的具体过程实现，在以往的实验中是没有接触过的，比较不熟悉，甚至在PA里也没有涉及到。通过此次实验中的实现，进一步理解了操作系统对进程切换的处理，陷阱帧的使用的知识点。对这些概念的理解对之后实现进一步的进程管理也是很有用的。

6、在操作系统中，可以直接访问硬件，使用特权指令来实现I / O。但是在用户程序中，必需要通过操作系统的帮助才可以完成。在用户层面，库函数将各种系统调用封装成易于使用的形式，使得在用户程序中可以比较方便地使用。在内核层面，则要提供各种系统调用的处理函数，通过处理陷阱帧中的系统调用号和传入参数，正确地响应用户程序的要求。

此次实现的一些使用系统调用的库函数如下：

```
static inline int //__attribute__((__noinline__))
syscall(int id, ...) {
    int ret;
    int *args = &id;
    asm volatile("int $0x80": "=a"(ret) : "a"(args[0]), "b"(args[1]), "c"(args[2]),
"d"(args[3]));
    return ret;
}


int serial_output(int fd, char *buf, int len) {
    return syscall(SYS_write, fd, buf, len);
}

int loadVideo(const uint8_t *buffer, int position, int size){
    return syscall(SYS_loadVideo, buffer, position, size);
}

int fullVideo(const unsigned char* src) {
    return syscall(SYS_fullVideo, src);
}

uint32_t getTime() {
    return syscall(SYS_time);
}

int readKey() {
    return syscall(SYS_keyboard);
}
```