

操作系统Lab2实验报告

211300082 谭荣熙 rongxitan@smail.nju.edu.cn

1. 实验要求

本次实验通过磁盘加载引入内核, 区分内核态和用户态, 完善中断机制, 并通过实现用户态I/O函数基于中断实现系统调用的全过程.

1.1 实现中断机制

1.2 实现系统调用库函数printf和对应的处理例程

1.3 键盘按键的串口回显

1.4 实现系统调用库函数getChar, getStr和对应的处理例程

2. 实验过程

在Lab2中, 我完成了实验的全部内容, 接下来我将从三个阶段来介绍完成实验的过程.

2.1 从实模式进入保护模式与加载内核

- 填写bootMain函数: 源代码已从磁盘中读出引导程序所处的扇区, 需要读取kMainEntry, phoff. 注意到elf已经指明内核态elf文件的首地址, 故只需对其进行读取即可.

```
struct ELFHeader *ehdr = (void *)elf;
kMainEntry = (void (*)(void)) (ehdr->entry);
phoff = ehdr->phoff;
```

- 填写start.S文件中的esp: 为了确定bootMain函数的地址, 使用了readelf指令进行反汇编, 读得其入口地址位于0x7c00处, 故设置esp为0x7c00, 使得程序跳转到该地址执行以执行引导程序.

```
movl $0x7c00, %eax
movl %eax, %esp
```

2.2 完善kernel相关的初始化设置

- 初始化中断门和陷阱门: 通过阅读手册与查阅资料, 我了解清楚中断门, 陷阱门对应idt中断向量表中每一位的含义, 故对其进行初始化设置. 下方只给出中断门的设置, 陷阱门类似.

```
static void setIntr(struct GateDescriptor *ptr, uint32_t selector,
uint32_t offset, uint32_t dpl) {
    ptr->offset_15_0 = offset & 0xffff;
    ptr->segment = KSEL(selector);
    ptr->pad0 = 0;
    ptr->type = 0xe;
    ptr->system = 0;
    ptr->privilege_level = dpl;
    ptr->present = 1;
    ptr->offset_31_16 = (offset >> 16) & 0xffff;
}
```

- 初始化IDT表, 为中断设置中断处理函数: 首先在doIrq.S汇编代码中补全keyboard事件的中断事件码0x21, 再根据给出的函数, 在idt.c中为各中断补全中断处理函数.

```
    setTrap(idt + 0x8, SEG_KCODE, (uint32_t)irqDoubleFault,
DPL_KERN);
    setTrap(idt + 0xa, SEG_KCODE, (uint32_t)irqInvalidTSS,
DPL_KERN);
    setTrap(idt + 0xb, SEG_KCODE, (uint32_t)irqSegNotPresent,
DPL_KERN);
    setTrap(idt + 0xc, SEG_KCODE, (uint32_t)irqStackSegFault,
DPL_KERN);
    setTrap(idt + 0xd, SEG_KCODE, (uint32_t)irqGProtectFault,
DPL_KERN);
    setTrap(idt + 0xe, SEG_KCODE, (uint32_t)irqPageFault, DPL_KERN);
    setTrap(idt + 0x8, SEG_KCODE, (uint32_t)irqDoubleFault,
DPL_KERN);
    setTrap(idt + 0x11, SEG_KCODE, (uint32_t)irqAlignCheck,
DPL_KERN);
    setTrap(idt + 0x1e, SEG_KCODE, (uint32_t)irqSecException,
DPL_KERN);

    setIntr(idt + 0x21, SEG_KCODE, (uint32_t)irqKeyboard, DPL_KERN);
    setIntr(idt + 0x80, SEG_KCODE, (uint32_t)irqSyscall, DPL_USER);
```

- 补充内核main函数: 由于各init函数已实现好, 故只需要加入各init函数即可, 此处不过多赘述.
- 填充中断处理程序的调用: 根据各中断给出的中断号, 在irqHandle.c文件中, 根据不同的中断填充其对应调用的处理函数.

```

switch(tf->irq) {
    case -1: break;
    case 0xd: GProtectFaultHandle(tf); break;
    case 0x80: syscallHandle(tf); break;
    case 0x21: KeyboardHandle(tf); break;
    default: assert(0);
}

```

- 填写系统调用的处理函数: 根据lib中的定义的STD_IN和STD_STR, 填写syscallRead函数的调用分配.

```

void syscallRead(struct TrapFrame *tf) {
    switch(tf->ecx) {
        case 0:
            syscallGetChar(tf);
            break;
        case 1:
            syscallGetStr(tf);
            break;
        default: break;
    }
}

```

2.3 由kernel加载用户程序

- 填写loadUMain函数: 根据指引, 参照bootloader中加载内核的方式进行填写. 值得注意的是, 在bootMain中, 从磁盘的前200个512字节中读取程序; 故在loadUMain中, 应从第201个区域开始读.

```

void loadUMain(void) {
    int i = 0;
    int offset = 0x1000;
    unsigned int elf = 0x200000;
    uint32_t uMainEntry = 0x0;
    for (i = 0; i < 200; i++) {
        readSect((void *) (elf + i*512), 201+i);
    }
    struct ELFHeader *ehdr = (void *)elf;
    uMainEntry = (uint32_t) (ehdr->entry);
    for (i = 0; i < 200 * 512; i++) {
        *(uint8_t *) (elf + i) = *(uint8_t *) (elf + i + offset);
    }
    enterUserSpace(uMainEntry);
}

```

- 填写中断处理函数: 填写未完全实现的keyboardHandle, syscallPrint, syscallGetChar, syscallGetStr等函数. 具体可使用keyBuffer数组来辅助相应功能的实现.
 - keyboardHandle的实现: 输入回车符时, 只更新displayCol和displayRow, 并在keyBuffer中加入\n. 输入退格符时, 先判断退格是否位于当行行首, 若是则不进行退格, 否则调用print处理函数, 将退格后的当行字符串重新打印到显存中, 同时维护keyBuffer和光标位置. 输入正常字符时, 若其为可打印字符, 则将其加入keyBuffer, 并直接将其打印到显存中(而不通过syscallPrint), 并注意维护光标位置.
 - syscallPrint的实现: 首先维护段选择子int sel = USEL(SEG_UDATA);并通过实验手册给出的指引, 依次将每个字符打印到显存中, 并同步维护displayCol和displayRow的值. 同时要考虑换行和翻页问题, 当displayRow >= 25时, 使用keyboard.c中的scrollScreen函数进行翻页处理.
 - syscallGetChar的实现: 首先判断用户是否输入了回车符来确定输入是否结束, 即通过keyBuffer[bufferTail-1]是否为\n来确定, 若未结束则返回0. 若已结束且并不位于行首, 返回当前buffer区中的第一个字符, 即keyBuffer[bufferHead++].
 - syscallGetStr的实现: 首先和GetChar一样, 需要判断输入是否结束. 其次需要将keyBuffer中相应长度的字符串写到tf->edx对应的内存中. 而这部分内存是用户占用的, 不能直接通过高级语言访问, 需通过初始化段选择子并以内联汇编的形式访问用户相应的段. 核心代码部分如下.

```
for (i = 0; i < min(tf->ebx, bufferTail-bufferHead); i++) {
    asm volatile("movb %1, %%es:(%0)":"r"(tf->edx+i), "r"
    (keyBuffer[bufferHead+i]));
}
```

2.4 实现用户需要的库函数

- printf函数的实现: 使用文件中已封装好的转换函数, 根据不同的case进行相应处理即可.

```
void printf(const char *format, ...) {
    ...
    void *paraList = (void *)&format + sizeof(uint32_t);
    ...
    while(format[i] != 0) {
        char ch = format[i++];
        switch(state) {
            case 0:
                if (ch == '%') state = 1;
                else buffer[count++] = ch;
                break;
```

```

        case 1:
            if (ch == 'd') {
                decimal = *(int *)paraList;
                paraList += 4;
                count = dec2Str(decimal, buffer,
MAX_BUFFER_SIZE, count);
            }
            else if (ch == 'x') {
                ... //similar to above
            }
            else if (ch == 's') {
                ... //similar to above
            }
            else if (ch == 'c') {
                character = *(char *)paraList;
                paraList += 4;
                buffer[count++] = character;
                if (count == MAX_BUFFER_SIZE) {
                    syscall(SYS_WRITE, STD_OUT,
(uint32_t)buffer, (uint32_t)count, 0, 0);
                    count = 0;
                }
            }
            else {
                state = 2;
                return;
            }
            state = 0;
            break;
        case 2: return;
        default: break;
    }
}
}

```

- getChar函数的实现: 判断返回值是否为0来进行返回.

```

char getChar() {
    char ret = 0;
    while (ret == 0) ret = (char)syscall(SYS_READ, STD_IN, 0, 0, 0,
0);
    return ret;
}

```

- getStr函数的实现: 同样地, 判断返回值是否为0进行返回.

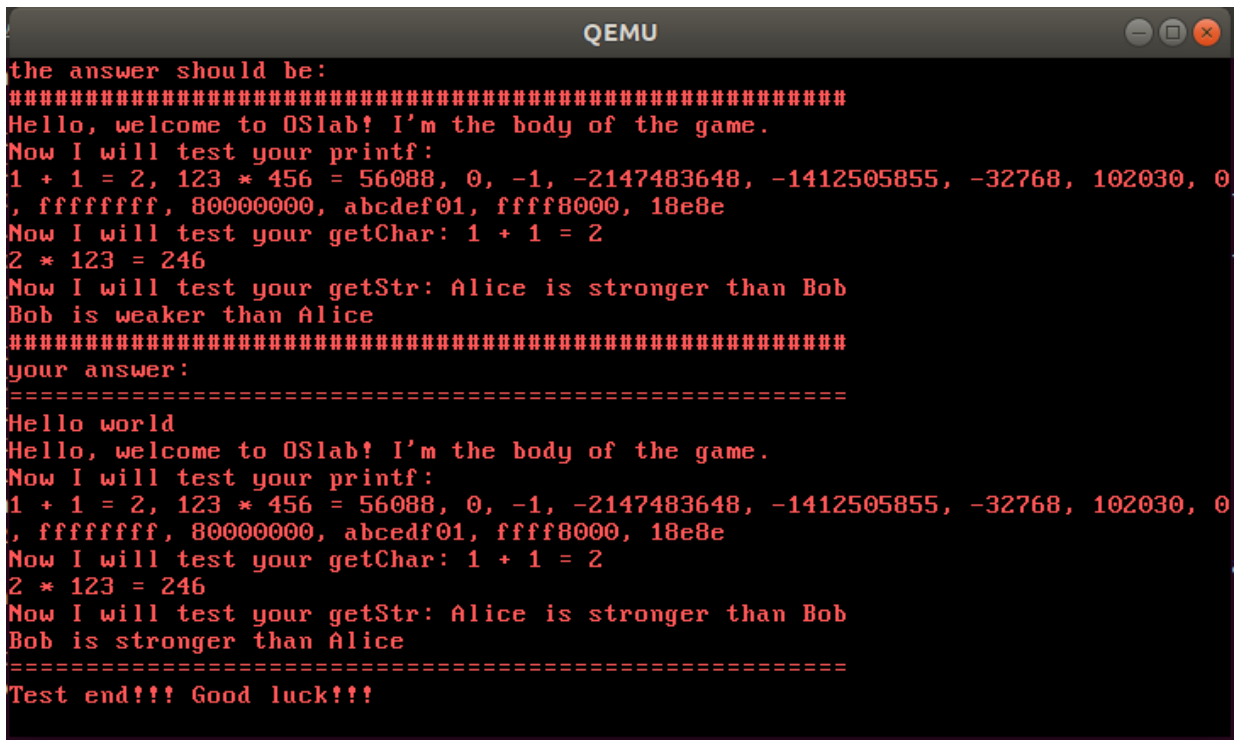
```

void getStr(char *str, int size) {
    int ret = 0;
    while (ret == 0) ret = syscall(SYS_READ, STD_STR, (uint32_t)str,
size, 0, 0);
    return;
}

```

3. 实验结果

- 上述实现顺利通过用户程序的测试.

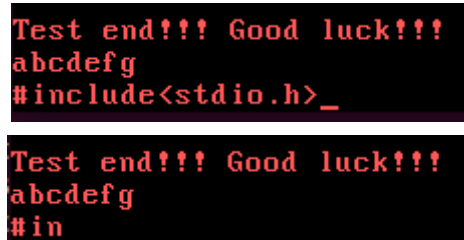


```

QEMU
the answer should be:
#####
Hello, welcome to OSlab! I'm the body of the game.
Now I will test your printf:
1 + 1 = 2, 123 * 456 = 56088, 0, -1, -2147483648, -1412505855, -32768, 102030, 0
, ffffffff, 80000000, abcdef01, ffff8000, 18e8e
Now I will test your getchar: 1 + 1 = 2
2 * 123 = 246
Now I will test your getStr: Alice is stronger than Bob
Bob is weaker than Alice
#####
your answer:
=====
Hello world
Hello, welcome to OSlab! I'm the body of the game.
Now I will test your printf:
1 + 1 = 2, 123 * 456 = 56088, 0, -1, -2147483648, -1412505855, -32768, 102030, 0
, ffffffff, 80000000, abcdef01, ffff8000, 18e8e
Now I will test your getchar: 1 + 1 = 2
2 * 123 = 246
Now I will test your getStr: Alice is stronger than Bob
Bob is stronger than Alice
=====
Test end!!! Good luck!!!

```

- 实现也能自动输入字符, 并实现退格.



```

Test end!!! Good luck!!!
abcdefg
#include<stdio.h>_

Test end!!! Good luck!!!
abcdefg
#in

```

综上结果分析, 实验大致正确.

4. 问题回答

4.1 计算机系统的中断机制在内核处理硬件外设的 I/O 这一过程中发挥了什么作用?

计算机系统的中断机制主要发挥了异步事件处理, 节省CPU资源, 处理多任务操作, 错误处理和恢复等作用.

- 异步事件处理: 指中断机制允许外设通过发送中断请求来异步地通知处理器, 从而及时相应外设的事件.
- 节省CPU资源: 若没有中断机制, 处理器需不断轮询外设状态来检查是否有新的数据或操作, 这使得处理器浪费大量的事件和资源. 而中断机制可让处理器不需要主动轮询外设.

- 处理多任务操作: 中断机制允许内核在处理硬件外设的I/O操作时与其他应用程序进行上下文操作, 从而实现多任务操作.
- 错误处理和恢复: 中断处理机制可根据终端类型和错误码等信息, 对硬件外设出错时进行相应的错误处理和恢复操作.

4.2 IA-32提供了4个特权级, 但TSS中只有3个堆栈位置信息, 分别用于ring0, ring1, ring2的堆栈切换. 为什么TSS中没有ring3的堆栈信息?

因为在IA-32体系架构中, ring3是用户态, 具有最低的特权级, 只能访问受限的用户空间资源, 不能直接访问内核资源. 而其他特权级到ring3的切换是通过使用不同的堆栈, 即加载任务的用户态堆栈指针来实现的, 而不需要通过TSS中保存堆栈位置信息来实现切换.

4.3 我们在使用eax, ecx, edx, ebx, esi, edi前将寄存器的值保存到了栈中, 如果去掉保存和恢复的步骤, 从内核返回之后会不会产生不可恢复的错误?

会发生不可恢复的错误, 因为这些寄存器被当作通用寄存器, 用于存放临时数据和函数调用时的参数传递. 若没有正确保存和恢复, 这些寄存器中的值在内核执行过程中可能会被内核代码修改, 从而影响到用户态的执行.

4. 实验心得

本次实验让我了解了计算机系统从按下键盘到输出到屏幕的一系列过程, 特别是理解清楚了中断机制和系统调用的概念和意义. 同时通过手动实现库函数, 我也对这些常用的函数在抽象层有了更进一步的认识. 这次实验的难点在于kernel部分的正确, 且在基本完全正确实现之前, 无法将整个程序跑起来, 这给debug带来了极大的困难. 但最终能顺利完成该实验, 我也收获满满.