Lab2实验报告

程序的加载

由bootloader加载kernel

内核代码加载到0x100000的位置

在boot.h中定义了一个ELFHeader结构体，读取并拆解elf (*physical memory addr to load*) 头的信息，对应找到其中的entry和phoff项即可

> ../bootloader/boot.c

```
// TODO: 阅读boot.h查看elf相关信息，填写kMainEntry、phoff、offset
    ELFHeader *elfHeader = (void *)elf;
    kMainEntry = (void(*)(void))elfHeader->entry;  // entry address of the program
    phoff = elfHeader->phoff; // program header offset
    offset = 0x1000; // .text section offset
```

在../bootloader中make时，遇到block too large的问题，可以对Makefile进行一点调整

```
ERROR: boot block too large: 1000 bytes (max 510)
../bootloader/Makefile:
objcopy -O binary bootloader.elf bootloader.bin
=>  objcopy  -S -j .text  -O binary bootloader.elf bootloader.bin
OK: boot block is 340 bytes (max 510)
```

从磁盘加载用户程序到内存相应地址

（实际顺序在kernel初始化之后）

参照bootloader加载内核的方式，由kernel加载用户程序到0x200000开始的位置

> ../kernel/kernel/kvm.c

```
void loadUMain(void) {
    // TODO: 参照bootloader加载内核的方式，由kernel加载用户程序
    putStr("Into loadUMain\n");
    uint32_t elf = 0x200000, offset = 0x1000, uMainEntry = 0x200000;
    for(int i=0; i<200; i++){
        readSect((void*)(elf+i*512), 201+i);
    }
    struct ELFHeader *elfHeader = (void *)elf;
    uMainEntry = elfHeader->entry;
    for(int i=0; i<200*512; i++){
        *((uint8_t*)(elf+i)) = *((uint8_t*)(elf+offset+i));
    }
    enterUserSpace(uMainEntry);
}
```

完善kernel相关的初始化设置

kernel的main中进行一系列初始化：

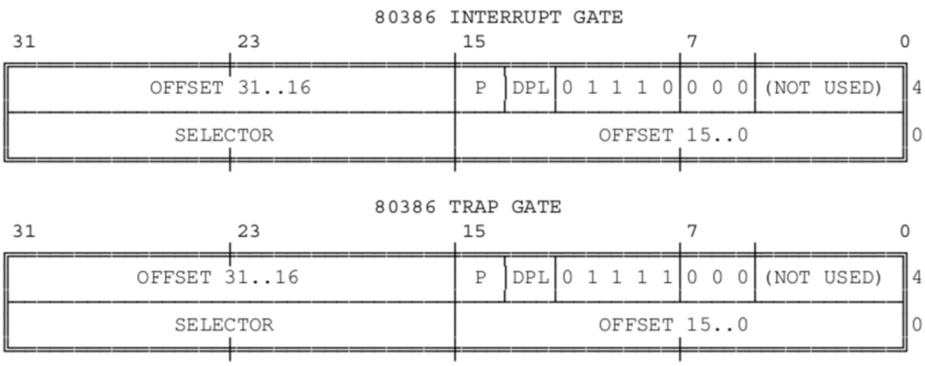
初始化串口输出（initSerial），初始化中断向量表（initIdt），初始化8259a中断控制器（initIntr），初始化GDT表配置TSS段（initSeg），初始化VGA设备（initVga），配置好键盘映射表（initKeyTable）

> ../kernel/main.c

```
void kEntry(void) {
    // Interruption is disabled in bootloader
    initSerial();// initialize serial port
    // TODO: 做一系列初始化
    // initialize idt
    initIdt();
    // initialize 8259a
    initIntr();
    // initialize gdt, tss
    initSeg();
    // initialize vga device
    initVga();
    // initialize keyboard device
    initKeyTable();
    loadUMain(); // load user program, enter user space
    while(1);
    assert(0);
}
```

初始化中断向量表IDT（Interrupt Descriptor Table）

首先初始化中断门和陷阱门。根据下表与GateDescriptor的对应位置填写。该函数的意义是使用后面的参数初始化ptr表项（类型为GateDescriptor），KSEL(selector)表示处于内核级别的段选择子。



> ../kernel/kernel/idt.c

```
/* 初始化一个中断门(interrupt gate) */
static void setIntr(struct GateDescriptor *ptr, uint32_t selector, uint32_t
offset, uint32_t dpl) {
    // TODO: 初始化interrupt gate
    ptr->offset_15_0 = offset & 0xffff;
    ptr->segment = KSEL(selector);
    ptr->pad0 = 0;
    ptr->type = INTERRUPT_GATE_32;
    ptr->system = 0;
```

```
    ptr->privilege_level = dpl;
    ptr->present = 1;
    ptr->offset_31_16 = (offset >> 16) & 0xffff;
}

/* 初始化一个陷阱门(trap gate) */
static void setTrap(struct GateDescriptor *ptr, uint32_t selector, uint32_t
offset, uint32_t dpl) {
    // TODO: 初始化trap gate
    ptr->offset_15_0 = offset & 0xffff;
    ptr->segment = KSEL(selector);
    ptr->pad0 = 0;
    ptr->type = TRAP_GATE_32;
    ptr->system = 0;
    ptr->privilege_level = dpl;
    ptr->present = 1;
    ptr->offset_31_16 = (offset >> 16) & 0xffff;
}
```

使用上面初始化门的函数，根据保护模式下80386执行指令过程中产生的异常对照表，初始化中断向量表IDT，为中断设置中断处理函数

8	#DF	双重错误	Abort	有(或零)	所有能产生异常或 NMI 或 INTR 的指令
9		协处理器段越界	Fault	无	浮点指令(386之后的 IA32 处理器不再产生此种异常)
10	#TS	无效TSS	Fault	有	任务切换或访问 TSS 时
11	#NP	段不存在	Fault	有	加载段寄存器或访问系统段时
12	#SS	堆栈段错误	Fault	有	堆栈操作或加载 SS 时
13	#GP	常规保护错误	Fault	有	内存或其他保护检验
14	#PF	页错误	Fault	有	内存访问
15	--	Intel 保留, 未使用			
16	#MF	x87FPU浮点错(数字错)	Fault	无	x87FPU 浮点指令或 WAIT/FWAIT 指令
17	#AC	对齐检验	Fault	有(ZERO)	内存中的数据访问(486开始)
18	#MC	Machine Check	Abort	无	错误码(如果有的话)和源依赖于具体模式(奔腾 CPU 开始支持)
19	#XF	SIMD浮点异常	Fault	无	SSE 和 SSE2浮点指令(奔腾 III 开始)
20-31	--	Intel 保留, 未使用			
32-255	--	用户定义中断	Interrupt		外部中断或 int n 指令

> ../kernel/kernel/idt.c

```
void initIdt() {
    int i;
    /* 为了防止系统异常终止，所有irq都有处理函数(irqEmpty)。 */
    for (i = 0; i < NR_IRQ; i ++) {
        setTrap(idt + i, SEG_KCODE, (uint32_t)irqEmpty, DPL_KERN);
```

```
    }
    /*init your idt here 初始化 IDT 表, 为中断设置中断处理函数*/
    // TODO: 参考上面第48行代码填好剩下的表项
    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 + 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);
    /* 写入IDT */
    saveIdt(idt, sizeof(idt));//use lidt
}
```

中断处理

中断请求号入栈

> ../kernel/kernel/doIrq.S

```
.global irqKeyboard
irqKeyboard:
    pushl $0
    # TODO: 将irqKeyboard的中断向量号压入栈
    pushl $0x21
    jmp asmDoIrq
```

irqHandle处理中断请求程序

> ./kernel/kernel/irqHandle.c

根据irq（Interrupt Request）中断请求的编号选择中断处理程序

```
switch(tf->irq) {
    // TODO: 填好中断处理程序的调用
    case 0xd:
        GProtectFaultHandle(tf);
        break;
    case 0x21:
        KeyboardHandle(tf);
        break;
    case 0x80:
        syscallHandle(tf);
        break;
    default:break;
}
```

在KeyboardHandle函数中处理键盘输入

输入一个字符后，通过getKeyCode获取键盘码（如下图），然后分情况处理退格符、回车符和正常字符

Scan code	Key	Scan code	Key	Scan code	Key	Scan code	Key
		0x01	escape pressed	0x02	1 pressed	0x03	2 pressed
0x04	3 pressed	0x05	4 pressed	0x06	5 pressed	0x07	6 pressed
0x08	7 pressed	0x09	8 pressed	0x0A	9 pressed	0x0B	0 (zero) pressed
0x0C	- pressed	0x0D	= pressed	0x0E	backspace pressed	0x0F	tab pressed
0x10	Q pressed	0x11	W pressed	0x12	E pressed	0x13	R pressed
0x14	T pressed	0x15	Y pressed	0x16	U pressed	0x17	I pressed
0x18	O pressed	0x19	P pressed	0x1A	[pressed	0x1B	] pressed
0x1C	enter pressed	0x1D	left control pressed	0x1E	A pressed	0x1F	S pressed
0x20	D pressed	0x21	F pressed	0x22	G pressed	0x23	H pressed
0x24	J pressed	0x25	K pressed	0x26	L pressed	0x27	; pressed
0x28	' (single quote) pressed	0x29	` (back tick) pressed	0x2A	left shift pressed	0x2B	\ pressed
0x2C	Z pressed	0x2D	X pressed	0x2E	C pressed	0x2F	V pressed
0x30	B pressed	0x31	N pressed	0x32	M pressed	0x33	, pressed
0x34	. pressed	0x35	/ pressed	0x36	right shift	0x37	(keypad) *

处理正常字符

先把字符存入缓冲区keyBuffer中；再将用户数据段的段选择器复制到`%es`寄存器（这意味着接下来的内存访问操作将在用户数据段中进行）；然后将读取到的字符显示到屏幕上；最后进行行列是否溢出的判断。

```
// TODO: 处理正常的字符
char ch = getChar(code);
if (ch >= 0x20) {
    putChar(ch);
    keyBuffer[bufferTail++] = ch;
    int sel = USEL(SEG_UDATA);
    asm volatile("movw %0, %%es"::"m"(sel));

    uint16_t data = ch | (0x0c << 8);
    int pos = (80 * displayRow + displayCol) * 2;
    asm volatile("movw %0, (%1)"::"r"(data), "r"(pos + 0xb8000));

    displayCol ++;
    if (displayCol >= 80) {
        displayCol = 0;
        displayRow ++;
    }
    while (displayRow >= 25) {
```

```
        scrollScreen();
        displayRow --;
        displayCol = 0;
    }
}
```

在syscallPrint函数中，完成光标的维护和打印到显存

```
if (character == '\n') {
    displayRow += 1;
    displayCol = 0;
}
else {
    data = character | (0x0c << 8);
    pos = (80 * displayRow + displayCol) * 2;
    asm volatile("movw %0, (%1)"::"r"(data), "r"(pos + 0xb8000));
    displayCol ++;
}

if (displayCol >= 80) {
    displayCol = 0;
    displayRow ++;
}
while (displayRow >= 25) {
    scrollScreen();
    displayRow --;
    displayCol = 0;
}
```

实现系统调用 syscallGetChar 和 syscallGetStr：

先记录并清除缓冲区末端的回车，在有回车的情况下返回字符

其中char类型之间通过eax返回字符，str类型需要将str写入edx（返回值eax用于表示有没有读到str）

```
void syscallGetChar(struct TrapFrame *tf){
    // TODO: 自由实现
    int flag = 0;
    if(keyBuffer[bufferTail-1] == '\n') flag = 1;
    while(bufferTail > bufferHead && keyBuffer[bufferTail-1] == '\n')
        keyBuffer[--bufferTail] = '\0';
    if(bufferTail > bufferHead && flag){
        tf->eax = keyBuffer[bufferHead++];
        bufferHead = bufferTail;
    }
    else {
        tf->eax = 0;
    }

}
void syscallGetStr(struct TrapFrame *tf){
    // TODO: 自由实现
    int size = tf->ebx;
    char *str = (char*)tf->edx;
    int i = 0;
```

```
    int sel = USEL(SEG_UDATA);
    asm volatile("movw %0, %%es"::"m"(sel));

    int flag = 0;
    if(keyBuffer[bufferTail-1] == '\n') flag = 1;
    while(bufferTail > bufferHead && keyBuffer[bufferTail-1] == '\n')
        keyBuffer[--bufferTail] = '\0';
    if(!flag && bufferTail-bufferHead < size) {
        tf->eax = 0;
    }
    else{
        for(i=0; i<size && i<bufferTail-bufferHead;i++){
            char ch = keyBuffer[bufferHead+i];
            asm volatile("movb %0, %%es:(%1)"::"r"(ch),"r"(str+i));
        }
        tf->eax = 1;
    }
}
```

系统调用

> ../lib/syscall.c

getChar和getStr：通过syscall调用 syscallGetChar 和 syscallGetStr，用SYS_READ进行调用选择，其余用于传参（然后保存在ecx、edx、ebx等寄存器中）

```
char getChar(){ // 对应SYS_READ STD_IN
    // TODO: 实现getChar函数，方式不限
    char ret = 0;
    while (ret == 0)
        ret = (char)syscall(SYS_READ, STD_IN, 0, 0, 0, 0);
    return ret;
}
void getStr(char *str, int size){ // 对应SYS_READ STD_STR
    // TODO: 实现getStr函数，方式不限
    int ret = 0;
    while (ret == 0)
        ret = syscall(SYS_READ, STD_STR, (uint32_t)str, size, 0, 0);
    return;
}
```

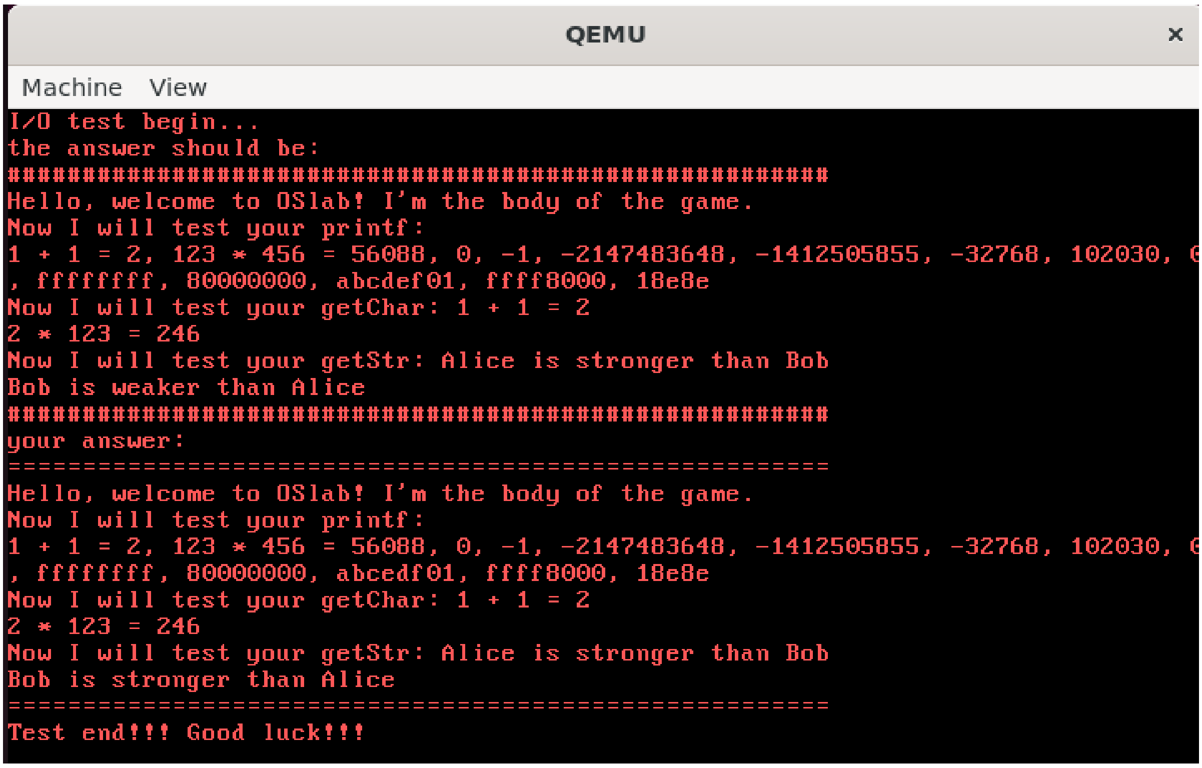
实现%d %x %s %c的处理，可以用自动机的思想，切换state实现。注意前面paraList+=4（移到format后面的位置）

```
// TODO: support format %d %x %s %c
char ch = format[i++];
switch(state) {
    case 0:
        if (ch == '%') state = 1;
        else buffer[count++] = ch;
        break;
    case 1:
```

```
        switch (ch){
            case 'd':
                decimal = *(int *)paraList;
                paraList += 4;
                count = dec2Str(decimal, buffer, MAX_BUFFER_SIZE, count);
                state = 0;
                break;
            case 'x':
                hexadecimal = *(uint32_t *)paraList;
                paraList += 4;
                count = hex2Str(hexadecimal, buffer, MAX_BUFFER_SIZE, count);
                state = 0;
                break;
            case 's':
                string = *(char **)paraList;
                paraList += 4;
                count = str2Str(string, buffer, MAX_BUFFER_SIZE, count);
                state = 0;
                break;
            case '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;
                }
                state = 0;
                break;
            default:
                state = 2;
                break;
        }
        break;

    case 2: return;
    default: break;
}
```

成功通过测试


QEMU
Machine View
I/O test begin...
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,
, 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, 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,
, ffffffff, 80000000, abcedf01, 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!!!