《操作系统》实验报告

实验二：系统调用

实验内容：基于中断实现系统调用

实验时间：2018.3.29

姓名：敬舒舒

学号：161220056

班级：操作系统 2 班

邮箱：1151545191@qq.com

一、实验目的

本次实验的目的包括：

1）实现一个简单的应用程序加本实验通过实现一个简单的应用程序

2）调用一个自定义实现的系统调用

3）完成基于中断实现系统调用的全过程

二、实验原理（知识背景）

实验流程如下（具体内容在实验讲义上，此处不进行复制描述）

1. Bootloader 从实模式进入保护模式,加载内核至内存，并跳转执行
2. 内核初始化 IDT（Interrupt Descriptor Table，中断描述符表），初始化 GDT，初始化 TSS（Task State Segment，任务状态段）
3. 内核加载用户程序至内存，对内核堆栈进行设置，通过 iret 切换至用户空间，执行用户程序
4. 用户程序调用自定义实现的库函数 printf 打印字符串
5. printf 基于中断陷入内核，由内核完成在视频映射的显存地址中写入内容，完成字符串的打印

三、实验环境等

1）环境：VMware 虚拟机　ubuntu-16.04.4

2）实验基础：助教提供的框架代码

实验 2.0：框架代码理解

一、整体框架

首先阅读 makefile 文件，获知对文件在操作及代码存储/运行地址：

（1） Bootloader 文件：生成引导程序

$(BOBJS)

$(LD) $(LDFLAGS) -e start -Ttext 0x7c00 -o bootloader.elf $(BOBJS)

将编译生成的.o 文件链接起来生成 bootloader.elf，指定 start 为程序入口处，并将 start 的初始地址重定位至 0x7c00。

objcopy -O binary bootloader.elf bootloader.bin

将.elf 的内容复制到.bin 中去

@../utils/genBoot.pl bootloader.bin

对主引导扇区的 bootloader 进行检查

将其补充至 512 字节，末尾两个字节为 0x55 和 0xaa

（2） Kernel 文件夹：生成内核程序

$(KOBJS)

$(LD) $(LDFLAGS) -e kEntry -Ttext 0x00100000 -o kMain.elf $(KOBJS)

将编译生成的.o 文件链接起来生成 kMain.elf，指定 kEntry 为程序入口处，并将 kEntry 的初始地址重定位至 0x00100000。

@../utils/genKernel.pl kMain.elf

对内核代码 kMain.elf 进行检查，将其补充至 200 个扇区大小。

需要注意的是，由于在生成 kMain.elf 时并没有像生成将代码段复制出来，我们在 **bootloader 加载内核代码时需要对读取的 elf 文件进行解析**，找到 kEntry 代码段的位置，并跳转执行。

（3） App 文件夹：生成用户程序

$(UOBJS)

$(LD) $(LDFLAGS) -e uEntry -Ttext 0x00200000 -o uMain.elf $(UOBJS)

将编译生成的.o 文件链接起来生成 uMain.elf，指定 uEntry 为程序入口处，并将 kEntry 的初始地址重定位至 0x00200000。

与 kenel 中的内核代码一样，这里没有将代码段单独拿出来，我们在 **kvm.c 中函数 loadUMain 加载用户程序时需要对读取的 elf 文件进行解析**，找到 uEntry 代码段的位置，并跳转执行

Bootloader 文件夹生成引导程序以及 aap 用户程序文件夹与头文件 lib 文件夹没什么好说的，本次实验的重点在于实现 kernel 中的内核代码。

Kenel 文件夹中又分为 include、kernel、lib 三个文件夹与 main.c 文件。

（1）main.c 文件：

调用了头文件中的库函数对操作系统进行了初始化并跳转到用户程序。

（2）include 文件夹：

为头文件，声明了内核代码需要的函数和宏（由于我们需要实现操作系统，False、True 等基础函数/变量都需要自己声明，这里的框架代码已经帮我们完成了）。

（3）Kernel 文件夹中定义了头文件声明的函数：

serial.c　initSerial()初始化串口输出

idt.c　初始化中断描述表，在这里我们初始化了中断门及陷阱门

声明了汇编代码中的函数

初始化中断表中断表一共有 256 项，由于本次实验只使用 0x80 及 0xd 中断向量，我们将其他中断跳转都设置为 irqEmpty（不执行任何系统调用就返回）

i8259.c　initIntr() 初始化 8259 中断控制器

doIrq.S　中断处理（硬件根据中断参数查询中断表跳转至相应的函数中，根据情况压入错误码和中断向量，并保存现场，切换至内核。

irqHandle.c 中断处理程序，根据中断向量调用对应的中断处理程序

实现了系统调用的中断处理程序

kvm.c　初始化 GDT 表，初始化段寄存器，TSS，

加载用户程序，跳转进入用户态

实验报告将按程序执行顺序进行编写，不止讲述如何填补代码，还将在相应位置对代码框架及细节进行解释，以便更好的理解系统调用的过程与代码的具体实现

其中我们需要完成的部分包括：

bootloader：　start.s　boot.c

kernel/kernel：　irqHandle.c　kvm.c

lib:　syscall.c

实验 2.1：实现系统调用

一、引导程序 Bootloader

按照程序执行流程，我们首先完成引导程序 bootloader 从实模式进入保护模式,加载内核至内存，并跳转执行。Start 代码在实验一中已实现，但此处不需要初始化寄存器，因为在内核中会重新加载 gdt 表并初始化寄存器。而由于我们从实模式切换到保护模式是通过指令 data32 ljmp $0x08, $start32 所以 gdt 表必须初始化代码段

```
.code16

.global start
start:
        cli                             #关闭中断
        inb $0x92, %al                  #启动A20总线
        orb $0x02, %al
        outb %al, $0x92
        data32 addr32 lgdt gdtDesc      #加载GDTR
        movl %cr0, %eax                 #启动保护模式
        orb $0x01, %al
        movl %eax, %cr0                 #置位PE，使能保护模式
        data32 ljmp $0x08, $start32     #长跳转切换至保护模式

.code32
start32:
        jmp bootMain                    #跳转至bootMain函数 定义于boot.c

.p2align 2
gdt: # 8 bytes for each table entry, at least 1 entry
        .word 0,0 # empty entry
        .byte 0,0,0,0

        .word 0xffff,0                  # kernel code descriptor
        .byte 0,0x9a,0xcf,0

gdtDesc: # 6 bytes in total
        .word (gdtDesc - gdt -1)
        .long gdt
```

跳转至 bootMain 函数后，我们读取 kernel 代码，解析其 elf 文件头获取代码入口地址并跳转。Kernel 代码储存在磁盘的第 1 至 200 个扇区中，我们通过函数 readSect()循环将其读至一个足够大的不影响其它代码的地址(此处我们使用 buf=（char*）0x5000000)，对其进行解析。在 boot.h 中定义了 elf 文件头和程序头的数据结构，我们首先从其所处的地址 buf 获取文件头（elf)，再根据文件头内容获取程序头（ph）位置以及程序段数目（elf->phnum)。通过程序头我们可以得知该程序段的类型，虚拟地址，段长度，段所占内存大小。遍历所

有的 ProgramHeader，验证类型是否正确（可加载的内存段），并将其加载到正确的虚拟地址，超过代码大小未到内存大小的部分使用 0 补充。

```
void bootMain(void) {
	/* 加载内核至内存，并跳转执行 */
	char *buf, *dst, *src;
	struct ELFHeader *elf;
	struct ProgramHeader *ph;
	void (*entry)(void);
	int i, j;

	/* load kernel code*/
	buf = (char *)0x5000000;
	for (i = 1; i <= 200; i++)
		readSect((void*)(buf + 512 * (i - 1)), i);

	elf = (struct ELFHeader *)buf;
	ph = (struct ProgramHeader *)(buf + elf->phoff);

	for (i = 0; i < elf->phnum; i++)
	{
		/* laod each program segment to their virtual address
		 * type=1 : LOAD section */
		if(ph->type == 1)
		{
			/* the location of the segment is mapped to the virtual address space
			 * (for the segment of the PT_LOAD type*/
			dst = (char *)ph->vaddr;
			src = buf + ph->off;

			for (j = 0; j < ph->filesz; j++)
				dst[j] = src[j];
			for (; j < ph->memsz; j++)
				dst[j] = 0;
		}
		ph++;
	}

	/* jump to kernel */
	//((void (*)(void))elf->entry)();
	entry = (void(*)(void))(elf->entry);
	entry();
}
```

将所有的程序代码加载完成后，我们从文件头中读取程序入口地址（elf->entry）跳转执行内存代码。

二、内核程序 kernel

框架代码中已经帮我们实现了内核的头文件结构体定义、函数声明和宏定义，以及部分库函数实现。我们需要补充完整的部分仅包括 kernel 子文件夹下的 irqHandle.c kvm.c 两个文件的部分函数实现。阅读给出的头文件代码，我们发现框架中除了给出内核程序必须的数据结构、函数及宏定义，还给出了断言函数 asser()以及终端打印函数 putChar()这些函数能帮助我们调试程序和完善程序结构。

(1) kvm.c

在此文件中我们需要初始化 GDT 表并加载和段寄存器，加载用户程序并跳转执行。

首先需要生成 gdt 表，在 kvm.c 开头已经声明了数组形式的 gdt 表，和 tss 段。

```
SegDesc gdt[NR_SEGMENTS];
TSS tss;
```

在 include/x86/memory.h 中我们找到了 SegDesc 的结构体定义和 NR_SEGMENTS 的宏定义，可以看到 gdt 是一个大小为 7 的 SegDesc 数组。

```
// GDT entries
#define NR_SEGMENTS      7           // GDT size
#define SEG_KCODE   1           // Kernel code
#define SEG_KDATA   2           // Kernel data/stack
#define SEG_UCODE   3           // User code
#define SEG_UDATA   4           // User data/stack
#define SEG_TSS     5           // Global unique task state segement
#define SEG_VIDEO   6
```

```
struct SegDesc {
        uint32_t lim_15_0 : 16;  // Low bits of segment limit
        uint32_t base_15_0 : 16; // Low bits of segment base address
        uint32_t base_23_16 : 8; // Middle bits of segment base address
        uint32_t type : 4;       // Segment type (see STS_ constants)
        uint32_t s : 1;          // 0 = system, 1 = application
        uint32_t dpl : 2;        // Descriptor Privilege Level
        uint32_t p : 1;          // Present
        uint32_t lim_19_16 : 4;  // High bits of segment limit
        uint32_t avl : 1;        // Unused (available for software use)
        uint32_t rsv1 : 1;       // Reserved
        uint32_t db : 1;         // 0 = 16-bit segment, 1 = 32-bit segment
        uint32_t g : 1;          // Granularity: limit scaled by 4K when set
        uint32_t base_31_24 : 8; // High bits of segment base address
};
typedef struct SegDesc SegDesc;
```

而且我们可以看到在定义 GDT entries 时，加上第一个必须为 0 的 GTD 我们只有 6 个 GDT 还少了一个视频段（而在本实验中我们必须使用到视频段在 qemu 中打印），我们在宏定义部分补充上视频段以便在 kvm.c 中初始化。在初始化并加载 gdt 时我们使用了 SEG 等宏定义，

```
#define SEG(type, base, lim, dpl) (SegDesc)                    \
{       ((lim) >> 12) & 0xffff, (uint32_t)(base) & 0xffff,      \
        ((uint32_t)(base) >> 16) & 0xff, type, 1, dpl, 1,       \
        (uint32_t)(lim) >> 28, 0, 0, 1, 1, (uint32_t)(base) >> 24 }

#define SEG16(type, base, lim, dpl) (SegDesc)                  \
{       (lim) & 0xffff, (uint32_t)(base) & 0xffff,              \
        ((uint32_t)(base) >> 16) & 0xff, type, 0, dpl, 1,       \
        (uint32_t)(lim) >> 16, 0, 0, 1, 0, (uint32_t)(base) >> 24 }
```

```
#define DPL_KERN                0
#define DPL_USER                3

// Application segment type bits
#define STA_X       0x8         // Executable segment
#define STA_W       0x2         // Writeable (non-executable segments)
#define STA_R       0x2         // Readable (executable segments)

// System segment type bits
#define STS_T32A    0x9         // Available 32-bit TSS
#define STS_IG32    0xE         // 32-bit Interrupt Gate
#define STS_TG32    0xF         // 32-bit Trap Gate
```

include/x86/memory.h 中能够找到他们的具体定义。

需要注意的是，TSS 是系统段，它的段描述符与其他段描述符的结构不同，需要使用 SEG16 初始化，它的段地址为 tss 地址。接下来初始化 TSS，将 tss 的 esp0（栈起始地址）指向 0x200000（用户程序在 0x200000 开始，而栈是从高地址向低地址生长，不会影响到代码运行），并设置段类型为核心数据段）。

```
void initSeg() {
        gdt[SEG_KCODE] = SEG(STA_X | STA_R, 0,       0xffffffff, DPL_KERN);
        gdt[SEG_KDATA] = SEG(STA_W,         0,       0xffffffff, DPL_KERN);
        gdt[SEG_UCODE] = SEG(STA_X | STA_R, 0,       0xffffffff, DPL_USER);
        gdt[SEG_UDATA] = SEG(STA_W,         0,       0xffffffff, DPL_USER);
        gdt[SEG_TSS] = SEG16(STS_T32A,      &tss, sizeof(TSS)-1, DPL_KERN);
        gdt[SEG_TSS].s = 0;
        gdt[SEG_VIDEO] = SEG(STA_W,  0x0b8000,       0xffffffff, DPL_KERN);
        setGdt(gdt, sizeof(gdt));

        /*
         * 初始化TSS
         */
        tss.esp0 = 0x200000;    // set kernel esp to 0x200,000
        tss.ss0  = KSEL(SEG_KDATA);
        asm volatile("ltr %%ax":: "a" (KSEL(SEG_TSS)));

        /* set segment register */
        asm volatile("movl %0, %%eax":: "r"(KSEL(SEG_KDATA)));
        asm volatile("movw %ax, %ds");
        asm volatile("movw %ax, %es");
        asm volatile("movw %ax, %ss");
        asm volatile("movw %ax, %fs");
        asm volatile("movl %0, %%eax":: "r"(KSEL(SEG_VIDEO)));
        asm volatile("movw %ax, %gs");

        /*设置正确的段寄存器*/

        lLdt(0);
}
```

将 tss 初始化好后，使用 ltr 指令加载 TR。接下来使用内联汇编初始化其他通用寄存器并初始化 ldt。

在完成 initSeg()后，我们将使用 kvm.c 中的 loadUMain()函数来加载用户程序，与在 boot.c 中加载内核代码的方式相同，只需修改 readSect 读取磁盘的序号，我们先将用户代码放置一个足够远的地址，再根据它的文件

头和程序头将代码加载到代码应处于的对应虚拟地址处。但与加载内核代码不同的是，这里不是直接跳转到用户程序，而是使用 enterUserSpace()函数跳转至用户态。

```
void enterUserSpace(uint32_t entry) {
        /*
         * Before enter user space
         * you should set the right segment registers here
         * and use 'iret' to jump to ring3
         */
        asm volatile("pushl %0":: "r"(USEL(SEG_UDATA)));
        asm volatile("pushl %0":: "r"(128 << 20));
        asm volatile("pushfl");
        asm volatile("pushl %0":: "r"(USEL(SEG_UCODE)));
        asm volatile("pushl %0":: "r"(entry));
        asm volatile("iret"); // return to user space
}
```

首先压入用户态的 ss 和 esp，压入标志寄存器，压入代码段寄存器，压入 esp 寄存器（程序入口地址 entry），然后使用内联汇编 iret 从 ring0 返回 ring3 的用户态程序，在本实验中也就是 app 中的打印函数的开始地址（uEntry），代码如下：

```
void loadUMain(void) {

        /*加载用户程序至内存*/
        /* load main code*/
        char *buf, *dst, *src;
        struct ELFHeader *elf;
        struct ProgramHeader *ph;
        int i, j;

        /* load kernel code*/
        buf = (char *)0x6000000;
        for (i = 1; i <= 100; i++)
                readSect((void*)(buf + 512 * (i - 1)), 200 + i);

        elf = (struct ELFHeader *)buf;
        ph = (struct ProgramHeader *)(buf + elf->phoff);

        for (i = 0; i < elf->phnum; i++)
        {
                /* laod each program segment to their virtual address
                 * type=1 : LOAD section */
                if(ph->type == 1)
                {
                        /* the location of the segment is mapped to the virtual address space
                         * (for the segment of the PT_LOAD type*/
                        dst = (char *)ph->vaddr;
                        src = buf + ph->off;

                        for (j = 0; j < ph->filesz; j++)
                                dst[j] = src[j];
                        for (; j < ph->memsz; j++)
                                dst[j] = 0;
                }
                ph++;
        }
        /* enter user space */
        enterUserSpace(elf->entry);a
}
```

目前，我们已完成了引导程序加载内核程序，内核程序初始化，并加载用户程序的过程，接下来运行用户程序，而用户程序需要调用库函数 printf 在 qemu 打印。而 printf 则需要调用 syscall 函数来基于中断陷入内核，关

于 syscal.c 的说明将在下一模块中说明，这里我们先完成在 kernel 里系统调用的部分。

(2) irqHandle.c

在开始填补 irqHandle 前，我们先再次理解中断过程与代码实现。

陷入中断后，硬件依据中断向量查找中断表（根据 idt.c 中断表初始化的设置，0x80 对应的中断表的系统调用地址为定义在 dolrq.S 中的 irqSyscall）

```
        setTrap(idt + 0xd, SEG_KCODE, (uint32_t)irqGProtectFault, DPL_KERN);

        setIntr(idt + 0x80, SEG_KCODE, (uint32_t)irqSyscall, DPL_USER); // for int 0x80,
interrupt vector is 0x80, Interruption is disabled
```

```
.global irqSyscall
irqSyscall:
        pushl $0 // push dummy error code
        pushl $0x80 // push interrupt vector into kernel stack
        jmp asmDoIrq

.global asmDoIrq
asmDoIrq:
        pushal // push process state into kernel stack
        pushl %esp
        call irqHandle
        addl $4, %esp
        popal
        addl $4, %esp //interrupt vector is on top of kernel stack
        addl $4, %esp //error code is on top of kernel stack
        iret
```

在 irqSyscall 中首先压入错误码与中断向量，然后跳转至 asmDoirq 函数保存寄存器状态和栈顶位置，再调用 irqHandle 进行中断处理，当中断处理完成后，弹出所有寄存器回复现场，由于我们在之前还压入了错误码和中断向量，esp 寄存器的值需要两次 add 4 才能回复原样，最后使用 iret 回到用户态，继续执行用户程序。

在理清中断过程后我们再看 irqHandle 是如何实现系统调用的。

irqHandle 参数解释：在调用 irqHandle 时，发现这个函数有参数 struct TrapFrame *tf，结构体在 memory.h 中定义：

```
struct TrapFrame {
        uint32_t edi, esi, ebp, xxx, ebx, edx, ecx, eax;
        int32_t irq;
};
```

可以看出在 asmDoirq 保存寄存器时，我们多 push 了一次 esp，这是将我们 push 进去的错误码和寄存器看成一个结构体，在这个结构体的内容被保存后，再将他的地址 push 进去作为 call irqHandle 的参数，这样就完

```
PUSHAD  把EAX,ECX,EDX,EBX,ESP,EBP,ESI,EDI依次压入堆栈.
```

成了一次传参。再观察 push error code 和 pushal 的寄存器压入顺序，恰好与 tf 结构体的顺序相反，恰好符合结构体压栈的顺序。

在 irqHandle 中，首先依据中断向量（tf->irq）判断中断的类型（0x80 为系统调用），调用系统调用函数 syscallHandle 函数，依据储存在 eax 中的系统调用号确定调用哪个系统函数。这里我们进行了几项约定：

系统调用号储存在 eax 寄存器中，且屏幕打印的函数的调用号为 1，edx 为字符串长度，ebx 为字符串类型（输出“1”或错误信息“2”）。

```
#define SYS_write       1
```

依据系统调用号，我们调用 sys_write（tf）实现屏幕打印，由于我们只实现系统调用功能，不存在其他情况，则断言 default 不会发生。

```
void syscallHandle(struct TrapFrame *tf) {
        /* 实现系统调用*/
        switch(tf->eax) {
                case SYS_write: sys_write(tf); break;
                /* we can add more syscall */
                default: assert(0);
        }
}
```

```c
void sys_write(struct TrapFrame *tf)
{
        asm volatile("movl %0, %%eax":: "r"(KSEL(SEG_VIDEO)));
        asm volatile("movw %ax, %gs");
        static int row = 0, col = 0;
        char c = '\0';
        /* ebx:str type  ecx:str  edx:length */
        if (tf->ebx == 1 || tf->ebx == 2) {
                int i;
                for(i = 0; i < tf->edx; i++) {
                        c = *(char *)(tf->ecx + i);
                        putChar(c);
                        if (c == '\n') {
                                row++;
                                col = 0;
                                continue;
                        }
                        if (col == 80) {
                                row++;
                                col = 0;
                        }
                        video_print(row, col++, c);
                }
                tf->eax = tf->edx;
        }
}
```

每次屏幕输出前，都需要重新设置视频段寄存器。

接下来读取 tf 的屏幕打印的参数，循环打印字符串，并维护打印的行列位置 row 和 col。写函数返回值为打印字符个数，我们将 edx 赋值给 eax。

这里我们使用了 putChar 函数和 video_print 函数。putChar 为在终端打印，而 video_print 是自己实现的利用现存打印的函数。video_print 函数声明在 device.h 中，在 serial.c 里实现。

```c
/* print to video segment */
void video_print(int row, int col, char c) {
        asm ("movl %0, %%edi;"::"r"(((80 * row + col) * 2)):"%edi");
        asm ("movw %0, %%eax;"::"r"(0x0c00 | c):"%eax");     // 0x0黑底,0xc红字, 字母ASCII码
        asm ("movw %%ax, %%gs:(%%edi);":::"%edi");           // 写入显存
}
```

至此，我们已经完成了系统初始化和中断过程。接下来我们需要完成库函数 printf。

三、库函数 printf

printf 的具体实现在 syscall.c 中，这个函数除了接收字符串外还接受其他参数，我们需要将这些参数放到字符串中，再陷入中断打印。由于有不同的参数类型，我们需要不同的函数对字符串进行修改。在 uEntry 中的参数类型有字符串 prints(%s)、单个字符 printc(%c)、十进制整数 printd(%d)、十六进制整数 printx（%x)。printf 需要 stdarg.h 库对参数列表进行操作。

```
#include<stdarg.h>
#define SYS_write       1
```

printf 函数首先通过 va_list 获取参数列表，va_start 获得当前参数并自增 (va_start 执行获得当前参数后会自增指向下一个参数)，接下来对整个字符串循环遍历每个字符，直到访问到字符串结束符号 ‘\0’ 。

若访问到字符%，则检查其后面一个字符判断打印参数类型，并调用对应的打印函数将参数处理为字符串或字符形式，再调用 prints 输出参数结束后返回 printf 继续循环。我们使用 va_arg 传递参数

若当前字符不是%，则调用 printc 输出该字符。

```
void printf(const char *str, ...)
{
        char type;
        va_list ap;
        va_start(ap, str);
        if (str == 0)
                return;
        while(*str != '\0') {
                if(*str == '%') {
                type = *++str;
                switch (type) {
                        case 'd': printd(va_arg(ap, int));    break;
                        case 's': prints(va_arg(ap, char*)); break;
                        case 'c': printc(va_arg(ap, int));    break;
                        case 'x': printx(va_arg(ap, int));    break;
                }
        }
                else {
                        printc(*str);
                }
                str++;
        }
        va_end(ap);
}
```

printd和printx在将参数转化字符或字符串后都需要调用prints打印，循环中的字符或者参数的单个字符直接调用 printc 打印，而 prints 和 printc 都调用 syscall 来陷入中断打印，区别只是传递给 edx 的字符串长度。Syscall 函数使用内联汇编陷入中断，并将参数输入给对应寄存器。

eax：系统调用号，sys_write 约定为 1。返回值存储在 ret 中，正确的返回值应为字符串的长度

ebx：字符串类型（out / error）

ecx：字符/字符串地址

edx：字符串长度（单个字符则为 1）

```
int32_t syscall(uint32_t eax, uint32_t ebx, uint32_t ecx, uint32_t edx)
{
        int32_t ret = 0;
        asm volatile("int $0x80": "=a"(ret) : "a"(eax), "b"(ebx), "c"(ecx), "d"(edx));
        return ret;
}
```

接下来我们再看这四个 print？的具体实现

A. printc：参数为单个字符

单个字符则直接调用 syscall 打印，

```
void printc(char c) {
        syscall(SYS_write, 1, (uint32_t)&c, 1);
        return;
}
```

B. prints：参数为字符串

首先得出字符串的长度再调用 syscall 打印

```
void prints(const char* str) {
        int str_size = 0;
        while (str[str_size] != '\0')
                str_size++;
        syscall(SYS_write, 1, (uint32_t)str, str_size);
}
```

C.　printd：输出十进制整数

在这种情况下我们需要考虑几种特殊值：溢出（0x8000000），为 0，为负。在处理完特殊情况后再生成字符串，调用 prints 打印然后返回 printf 继续打印。

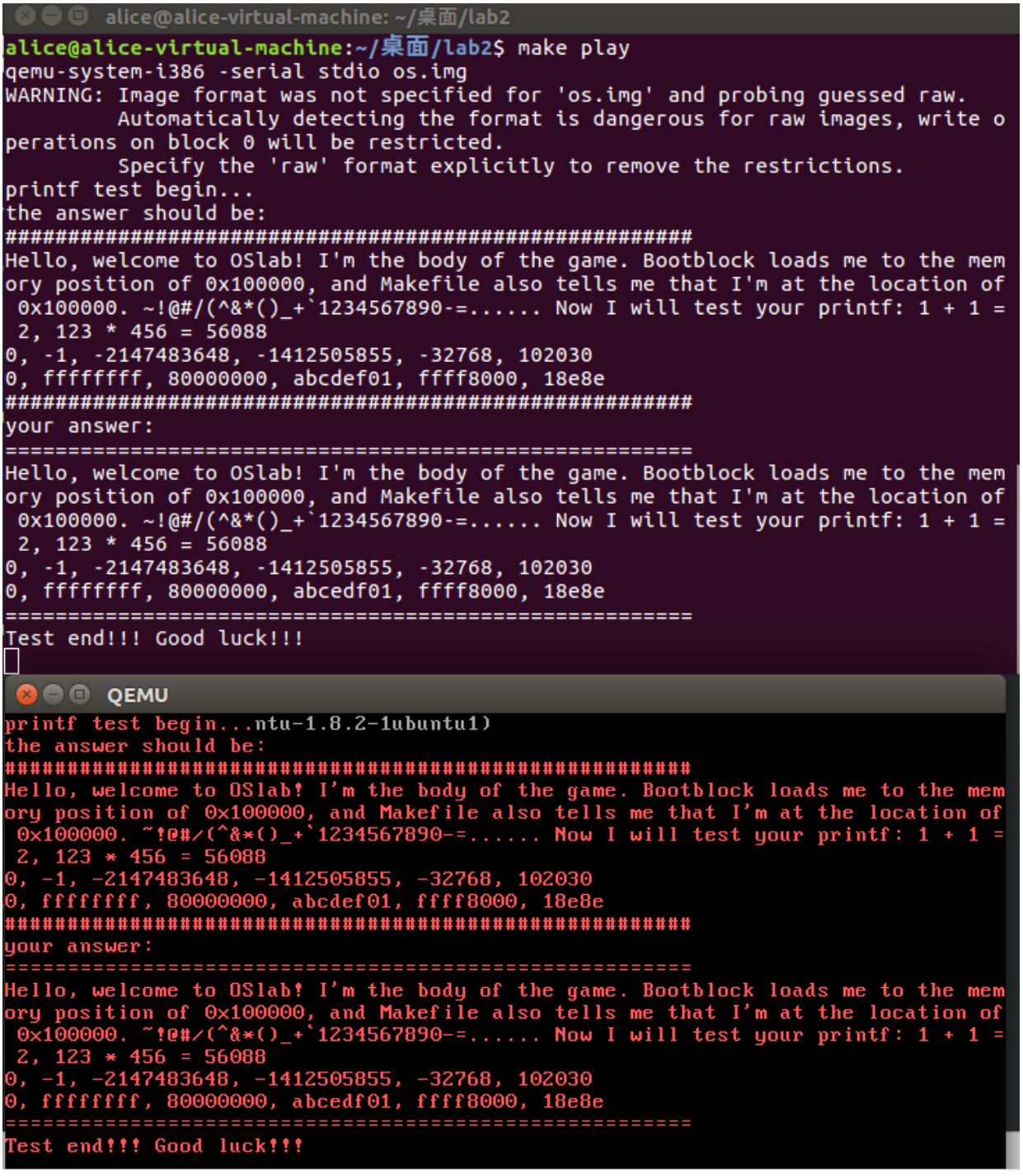
```
void printd(int d) {
        char buf[100];
        int str_size = 0;
        if (d == 0) {
            prints("0");
            return;
        }
        if (d == 0x80000000) {
            prints("-2147483648");
            return;
        }
        if (d < 0) {
            prints("-");
            d = -d;
        }
        while (d) {
            buf[str_size++] = d % 10 + '0';
            d /= 10;
        }
        for (int i = 0, j = str_size - 1; i < j; i++, j--) {
            char tmp = buf[i];
            buf[i] = buf[j];
            buf[j] = tmp;
        }
        buf[str_size] = '\0';
        prints(buf);
}
```

D.　pritnx：输出十六进制整数

同样是要考虑 0 和负大的特殊情况，再将十进制数转换为十六进制字符串，调用 prints 打印然后返回 printf 继续打印。

printf 库函数完成。

四、运行结果


alice@alice-virtual-machine: ~/桌面/lab2
alice@alice-virtual-machine:~/桌面/lab2$ make play
qemu-system-i386 -serial stdio os.img
WARNING: Image format was not specified for 'os.img' and probing guessed raw.
Automatically detecting the format is dangerous for raw images, write operations on block 0 will be restricted.
Specify the 'raw' format explicitly to remove the restrictions.
printf test begin...
the answer should be:
##
Hello, welcome to OSlab! I'm the body of the game. Bootblock loads me to the memory position of 0x100000, and Makefile also tells me that I'm at the location of 0x100000. ~!@#/(^&*()_+`1234567890-=..... Now I will test your printf: 1 + 1 = 2, 123 * 456 = 56088
0, -1, -2147483648, -1412505855, -32768, 102030
0, ffffffff, 80000000, abcdef01, ffff8000, 18e8e
##
your answer:
==
Hello, welcome to OSlab! I'm the body of the game. Bootblock loads me to the memory position of 0x100000, and Makefile also tells me that I'm at the location of 0x100000. ~!@#/(^&*()_+`1234567890-=..... Now I will test your printf: 1 + 1 = 2, 123 * 456 = 56088
0, -1, -2147483648, -1412505855, -32768, 102030
0, ffffffff, 80000000, abcedf01, ffff8000, 18e8e
==
Test end!!! Good luck!!!
QEMU
printf test begin...ntu-1.8.2-1ubuntu1)
the answer should be:
##
Hello, welcome to OSlab! I'm the body of the game. Bootblock loads me to the memory position of 0x100000, and Makefile also tells me that I'm at the location of 0x100000. ~!@#/(^&*()_+`1234567890-=..... Now I will test your printf: 1 + 1 = 2, 123 * 456 = 56088
0, -1, -2147483648, -1412505855, -32768, 102030
0, ffffffff, 80000000, abcdef01, ffff8000, 18e8e
##
your answer:
==
Hello, welcome to OSlab! I'm the body of the game. Bootblock loads me to the memory position of 0x100000, and Makefile also tells me that I'm at the location of 0x100000. ~!@#/(^&*()_+`1234567890-=..... Now I will test your printf: 1 + 1 = 2, 123 * 456 = 56088
0, -1, -2147483648, -1412505855, -32768, 102030
0, ffffffff, 80000000, abcedf01, ffff8000, 18e8e
==
Test end!!! Good luck!!!

实验完成

五、一些问题及解决

1.bootloader 的大小问题

在编译时 bootloader 的大小不满足条件（小于等于 510bytes）在尝试修改代码无果后，我们考虑编译选项的问题。汇编中的编译选项为：

```
CFLAGS = -m32 -march=i386 -static \
         -fno-builtin -fno-stack-protector -fno-omit-frame-pointer \
         -Wall -Werror -O2
```

其中-O2 为编译优化选项，其他优化选项如下：

```
-O0：关闭所有优化选项
-O1：第一级别优化，使用此选项可使可执行文件更小、运行更快，并不会增加太多编译时间，可以简写为-O
-O2：第二级别优化，采用了几乎所有的优化技术，使用此选项会延长编译时间
-O3：第三级别优化，在-O2的基础上增加了产生inline函数、使用寄存器等优化技术
-Os：此选项类似于-O2，作用是优化所占用的空间，但不会进行性能优化，常用于生成最终版本
```

我们尝试将优化选项改为-Os，成功将 wbootloader 大小压缩到 500bytes 以下。

2.loadUMain 函数编译错误

在进入用户态时总是跳转错误，使用指令 **objdump –d kMain.elf** 读取汇编代码发现，在读取 elf->entry 时，编译器地址编译错误，将 elf->entry 的地址（而非其值传递给了 Enteruserspace 函数作为跳转地址。

```
100345:       50                      push   %eax
100346:       a1 18 00 00 06          mov    0x6000018,%eax
```

在检查代码无误后，怀疑是编译的问题，于是将编译优化选项-O2 改为 -O0 关闭所有优化，再次编译发现编译正确。

3.代码细节及过程问题

这些问题在之前的实验代码实现报告中作为原理和注意事项进行了解释，这里不再单独列出来说明。