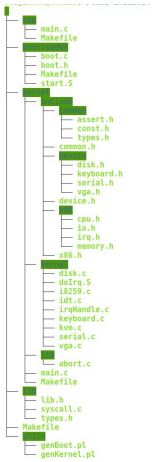
Lab2 系统调用实验报告

vigorweijia@foxmail.com

框架代码解读

各模块功能

框架代码结构如下：



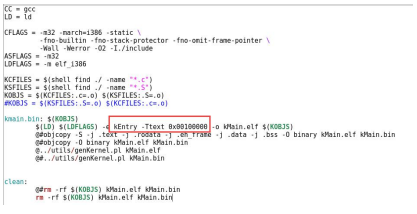
相较于上一次实验的代码，本次实验的代码结构要复杂得多。

app 文件夹是用户程序；bootloader 是操作系统内核运行前的引导程序；kernel 是操作系统内核；lib 是库函数；utils 是编译等的工具。

用户例程前的执行流程

bootloader/start.S：会开启保护模式然后跳转到 boot.c

bootloader/boot.c：会从磁盘的 1 号扇区开始读取内容，把操作系统的内核代码加载到主存中，然后跳转到 kMainEntry 执行，即执行操作系统内核的起始内容。



```
void bootMain(void) {
        int i = 0;
        int phoff = 0x34;
        int offset = 0x1000;
        unsigned int elf = 0x100000;
        void (*kMainEntry)(void);
        kMainEntry = (void(*)(void))0x100000; //Entry of kernel

        for (i = 0; i < 200; i++) {
                readSect((void*)(elf + i*512), 1+i);
        }

        kMainEntry = (void(*)(void))((struct ELFHeader *)elf)->entry;//elf file head
        phoff = ((struct ELFHeader *)elf)->phoff;
        offset = ((struct ProgramHeader *)(elf + phoff))->off;

        for (i = 0; i < 200 * 512; i++) {
                *(unsigned char *)(elf + i) = *(unsigned char *)(elf + i + offset);
        }

        kMainEntry();
}
```

kernel/main.c：查看 kernel 的 Makefile，0x100000 对应的是 kEntry，即 kernel 的 main 文件的 kEntry()函数，这个函数会调用外设的驱动进行初始化，同时初始化段表 GDT，中断描述符表 IDT，任务状态段 TSS，文件对应关系如下：

```
void kEntry(void) {

        // Interruption is disabled in bootloader

        initSerial();// initialize serial port at kernel/serial.c
        initIdt(); // initialize idt at kernel/idt.c
        initIntr(); // iniialize 8259 interrupt controller at kernel/8s59.c
        initSeg(); // initialize gdt, tss at kernel/kvm.c
        initVga(); // initialize vga device at kernel/vga.c
    initKeyTable(); // initialize keyboard device at kernel/keyboard.c
        loadUMain(); // load user program, enter user space at kernel/kvm.c

        while(1);
        assert(0);
}
```

由于操作系统内核代码大小不超过 100kb，因此最多加载 200 个扇区的内容。

在 kernel/main.c 的末尾，会执行 kvm.c 中的 loadUMain 函数。

kernel/kvm.c：loadUMain 函数会设置进入用户代码的入口，即 uMainEntry，从 app 的 Makefile 文件中可以看到，用户的入口位置被设置为了 0x200000。

```
void loadUMain(void) {
        int i = 0;
        int phoff = 0x34; // program header offset
        int offset = 0x1000; // .text section offset
        uint32_t elf = 0x200000; // physical memory addr to load
        uint32_t uMainEntry = 0x200000;

        for (i = 0; i < 200; i++) {
                readSect((void*)(elf + i*512), 201+i);
        }

        uMainEntry = ((struct ELFHeader *)elf)->entry; // entry address of the program
        phoff = ((struct ELFHeader *)elf)->phoff;
        offset = ((struct ProgramHeader *)(elf + phoff))->off;

        for (i = 0; i < 200 * 512; i++) {
                *(uint8_t *)(elf + i) = *(uint8_t *)(elf + i + offset);
        }

        enterUserSpace(uMainEntry);
}
```

```
uMain.bin: $(UOBJS)
        @#$(LD) $(LDFLAGS) -e uEntry -Ttext 0x00200000 -o uMain.elf $(UOBJS)
        $(LD) $(LDFLAGS) -e uEntry -Ttext 0x00000000 -o uMain.elf $(UOBJS)
        @#objcopy -S -j .text -j .rodata -j .eh_frame -j .data -j .bss -O binary uMain.elf uMain.bin
        @#objcopy -O binary uMain.elf uMain.bin
```

接着加载磁盘中从第 201 号扇区开始的内容至主存，然后进一步执行 enterUserSpace 函数。由于用户程序不大于 100kb，因此最多加载 200 个扇区的内容。

```
void enterUserSpace(uint32_t entry) {
        /*
         * Before enter user space
         * you should set the right segment registers here
         * and use 'iret' to jump to ring3
         */
        uint32_t EFLAGS = 0;
        asm volatile("pushl %0":: "r" (USEL(SEG_UDATA))); // push ss
        asm volatile("pushl %0":: "r" (0x2fffff)); //TODO push esp, further modification
        asm volatile("pushfl"); //push eflags, sti
        asm volatile("popl %0":"=r" (EFLAGS));
        asm volatile("pushl %0"::"r"(EFLAGS|0x200));
        asm volatile("pushl %0":: "r" (USEL(SEG_UCODE))); // push cs
        asm volatile("pushl %0":: "r" (entry)); //TODO push eip, further modification
        //asm volatile("pushl %0":: "r" (0x200000));
        //asm volatile("pushl %0":: "r" (0x000000));
        //asm volatile("sti"); //TODO may be not necessary
        asm volatile("iret"); //jump to UserProgram
/*
mov $0x23,%eax
push %eax
mov $0x2fffff,%eax
push %eax
pushf
pop %eax
or %0x2,%ah
push %eax
mov $0x1b,%eax
push %eax
mov 0x8(%ebp),%eax
push %eax
iret
```

此时对内核堆栈进行设定，然后使用 iret 指令跳转到用户程序。

app/main.c：uEntry()函数中，会调用库函数 printf 完成格式化输出和 scanf 完成格式化输入。

printf/scanf 的执行流程

lib/syscall.c：printf 在 lib/syscall.c 被定义。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。

printf()函数会完成格式串的解析，把需要输出的内容与格式串的内容相拼接，然后执行系统调用函数 syscall()进行输出。

syscall()会将相应的参数传入通用寄存器中，然后执行 int $0x80 指令陷入内核态。该中断向量的门描述符对应的处理程序是 irqSyscall，这个描述符在 kernel/kernel/idt.c 中被加载到 IDT 中。irqSyscall 在文件 kernel/kernel/doIrq.S 中被定义。因此，执行完该指令后会跳转到 irqSyscall 执行。

kernel/kernel/doIrq.S：会最终执行 irqHandle，在 kernel/kernel/irqHandle.c 中被定义

kernel/kernel/irqHandle.c：irqHandle 会判断中断号，发现是 0x80，会执行对应的处理函数 syscallHandle，即系统调用的处理程序。syscallHandle 会根据 eax 的值来判断对应的功能，此时 eax 为 0，表示输出，然后跳转到 syscallWirte 执行。syscallWirte 会判断文件描述符 ecx，然后跳转到 syscallPrint 执行，表示 STD_OUT。

syscallPrint 就是把相应字符串（首地址存在 edx，长度存在 ebx）的内容输出到显存。

在 irqHandle 执行完后，回到 doIrq.S 的 asmDoIrq 中，此时相应的内容出战，然后执行 iret 函数，回到用户态。

对与 scanf 的执行，内容大致与 printf 相同。scanf 会解析格式串，然后执行系统调用的函数 syscall 把缓冲区的内容保存到一个字符串中。

在执行 syscall 的时候，同样会在指令 int $0x80 陷入内核态，只不过在 syscallHandle 中，eax 的值为 1，然后跳转到 syscallRead，进一步执行 syscallScan，扫描输入，存储到缓冲区。最后回到用户态的时候，同样地把缓冲区的内容保存到一个字符串，然后把字符转换为相应类型的变量，修改相应的参数。

syscallScan 的执行流程

syscallScan 在内核态执行，其功能是扫描输入的内容，然后把相应的内容保存到输入的缓冲区。

在操作系统内核初始化的时候，会初始化扫描码的表格。

在 syscallScan 中调用 getKeyCode（在 kernel/kernel/keyboard.c 中定义）就可以获取键盘的扫描码，把这些扫描码存在一个缓冲区，然后转化为相应的 ASCII 码的字符，就成功地把输入的内容保存在了输入缓冲区。

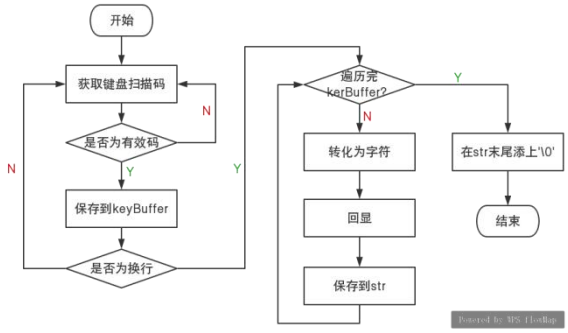
其中，getChar 函数能够把扫描码转化为 ASCII 码字符，putChar 把字符通过串口（serial）输出，而串口与 linux 的 terminal 相连，宏观表现为在 terminal 上显示回显信息。

代码填写

在框架代码的解读中，已经清晰地理顺了执行的流程，因此在写缺失的代码的时候，显得更加流畅。

syscallScan 的填写

前面讲到，syscallScan 中，调用 getKeyCode 把键盘的扫描码保存在缓冲数组中，具体流程如下：

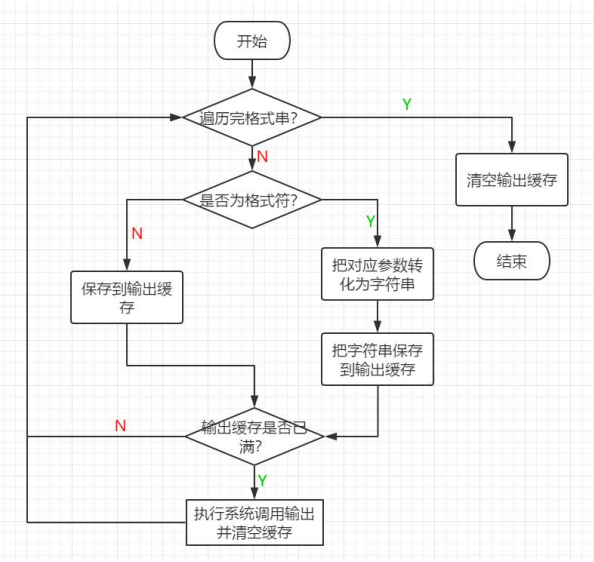


那么，现在只需要写一个循环，不断读入非 0 的扫描码，保存到 keyBuffer 中，直到遇到换行符，表示输入结束。

需要注意的是，虽然程序遇见换行符会结束，但是还是需要把换行符保存到 keyBuffer 中，不然会在后面输出的时候判断失败。

printf 的编写

printf 的执行流程如下：



把相应参数转化为字符串以及输出到缓存有相应的接口，在这个函数中，需要添加格式符的判断。

另一个问题是不定参数的获取，需要使用到 va_list，几个宏定义如下：

```
typedef char * va_list;//字符串指针
#define _INTSIZEOF(n) ( (sizeof(n) + sizeof(int) - 1) & ~(sizeof(int) - 1) )
#define va_start(ap,v) ( ap = (va_list)&v + _INTSIZEOF(v) )
#define va_arg(ap,t) ( *(t *)((ap += _INTSIZEOF(t)) - _INTSIZEOF(t)) )
#define va_end(ap) ( ap = (va_list)0 )
```

即根据变量的长度以及第一个参数的地址，去获取后面的参数。

关于格式符的判断，需要额外考虑%后跟的是否还是%。

在保存到输出缓存的时候需要判断缓存是否已满，而这些内容已经包含在了相应的转换接口内（调用 syscall）。

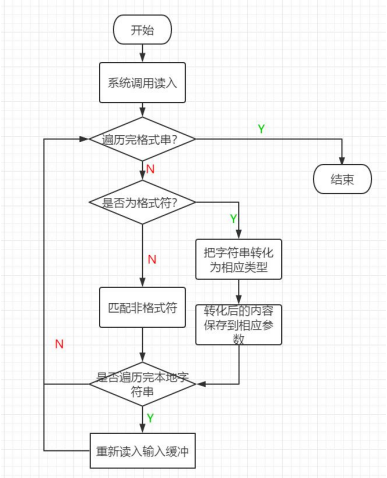
scanf 的编写

相较于 printf，scanf 的编写就显得比较困难了。

首先需要读取缓冲区的内容，把它保存到本地的一个字符串中。然后与格式串匹配。如果是在匹配非格式符的内容，则需要忽略掉字符串前中后的空格，若匹配失败，则返回已经匹配成功的格式符的个数。若匹配的是格式符，可以调用相应的接口，把输入的内容转化为相应的类型，然后保存到相应的参数中。

最终返回的是匹配成功的变量的个数。

在匹配的过程中还需要注意，如果匹配到某一个位置，读入的内容为'\0'，此时需要重新从缓冲区中读取相应的内容。流程如下：



流程类似 printf，唯一缺少的接口是对%c 格式符的处理。此外，还需要注意到空格，\n 以及\t 字符，在匹配非格式符的时候应当忽视它们。

同样地，scanf 需要判断是否已经遍历完毕缓存的内容，如果是，就需要继续输入并加入到新的缓存（再一次调用 syscall），而这一部分也包含在了相应的接口中。

心得体会

刚刚开始写的时候却是是比较懵逼的，因为框架代码比上一次实验复杂了很多倍。在理清楚了各个模块功能后，写起来的思路就比较清晰了，不过还是遇见了不少困难以及 bug。

首先是没有注意到在 printf 中的转义符，即%后面跟的是%。

然后是在保存扫描码的时候，没有注意到在输入换行后，需要把换行符的扫描码也保存在 keyBuffer 中。

其次是在写 scanf 的时候没有考虑到需要对非格式符的内容进行匹配，通过添加遇见非格式符的代码解决了这个问题。同时而来的还有无法实现多行输入（即字符串中间用换行分

隔而不用空格），通过随时判断是否到达缓冲末尾解决了这个问题。

由于错误提示、返回信息编写不完整，scanf 只能保证在正确输入的时候能够返回正确的结果。