

Lab2 系统调用

俞星凯 171830635

2651904866@qq.com

实验目的

1. 实现一个简单的应用程序，并在其中调用两个自定义实现的系统调用。
2. 了解基于中断实现系统调用的全过程。

实验内容

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 `scanf` 完成格式化输入和 `printf` 完成格式化输出。
5. `scanf` 基于中断陷入内核，内核扫描按键状态获取输入完成格式化输入（现阶段不需要考虑键盘中断）。
6. `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。

背景知识

1. IA-32 中断机制
 - (1) 确定与中断或异常关联的向量 $i(0-255)$ 。
 - (2) 读取 IDTR 寄存器指向的 IDT 中的第 i 项门描述符。
 - (3) 从 GDTR 寄存器获得 GDT 的基地址，并在 GDT 中查找，以读取上述门描述符中的段选择子所标识的段描述符。
 - (4) 若为软中断，需比较 CPL 与门描述符中的 DPL，若 $CPL > DPL$ ，则产生 #GP 异常。
 - (5) 比较 CPL 与段描述符中的 DPL，若 $CPL > DPL$ ，则发生特权级变化：读取 TR 寄存器，访问 TSS；选取 TSS 中记录的与 DPL 一致的 SS 与 ESP 切换堆栈；在切换后的堆栈中保存之前堆栈的 SS 与 ESP。
 - (6) 在堆栈中保存 EFLAGS，若中断为 Fault，则在堆栈中保存引起中断的 CS 与 EIP；否则，在堆栈中保存下条指令的 CS 与 EIP。
 - (7) 若中断产生一个 Error Code，则将其保存在堆栈中。
 - (8) 依据门描述符装载 CS 与 EIP，即执行中断处理程序。
 - (9) 使用 `iret` 指令从高特权级返回低特权级：对于 `iret` 指令，硬件会依次从当前

栈顶 pop 出 EIP, CS, EFLAGS, 即返回执行产生中断时的程序。若 pop 出的 CS 的 CPL 小于当前程序的 CPL, iret 还会继续 pop 出 ESP 以及 SS, 即切换堆栈。

2. 系统调用

可以将所有系统调用使用 `int $0x80` 软中断实现, 也可以为不同的系统调用分配不同的中断向量。每个系统调用至少需要一个参数, 即系统调用号, 用以确定通过中断陷入内核后, 该用哪个函数进行处理。可以使用 EAX, EBX 等等这些通用寄存器从用户态向内核态传递参数。

实验过程

1. bootloader

与 Lab1 类似, 从实模式进入保护模式, 加载内核至内存, 并跳转执行。

2. kernel

内核首先完成一些初始化工作。

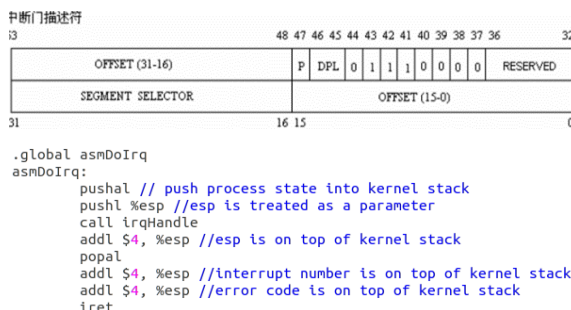
(1) 串口

在 `kernel/kernel/serial.c` 中通过 `outByte` 将 data 传入串口对应的几个端口完成初始化, 并提供了 `putChar` 函数用于终端显示。

```
void initSerial(void) {
    outByte(SERIAL_PORT + 1, 0x00);
    outByte(SERIAL_PORT + 3, 0x80);
    outByte(SERIAL_PORT + 0, 0x01);
    outByte(SERIAL_PORT + 1, 0x00);
    outByte(SERIAL_PORT + 3, 0x03);
    outByte(SERIAL_PORT + 2, 0xC7);
    outByte(SERIAL_PORT + 4, 0x0B);
}
```

(2) IDT

中断门描述符的结构如右图所示, 在 `kernel/kernel/idt.c` 中定义了两个分别用于初始化中断门和陷阱门描述符的函数, 利用它们初始化整个中断描述符表, 并且写入 IDTR 寄存器中。而中断描述符中 `offset` 是中断处理程序的入口, 它们在 `kernel/kernel/doirq.S` 中通过汇编代码定义, 可以看出最后都会执行 `asmDoIrq`, 并且调用 `kernel/kernel/irqHandle.c` 中的 `irqHandle` 函数, 它根据中断向量调用相应处理函数。



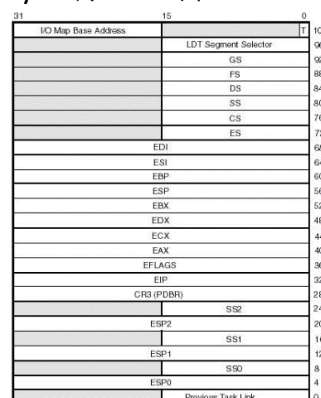
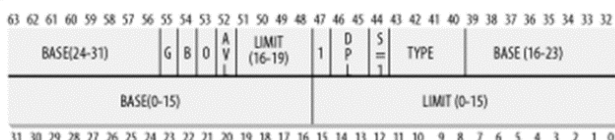
(3) 8259A

与串口的初始化类似, 在 `kernel/kernel/i8259.c` 中通过 `outByte` 将 data 传入 8259A 对应的几个端口完成初始化。

(4) GDT 和 TSS

在 `kernel/kernel/kvm.c` 中使用宏定义装载内核和用户的代码和数据以及 TSS 等七个段描述符, 并且写入 GDTR 中。由于不使用 LDTR, 故初始化为 0。

Data Segment Descriptor



(5) VGA

在 kernel/kernel/vga.c 初始化显示屏，将其清空并把光标置为最左上角。此外提供了 scrollScreen 函数以便在显示屏满时下移一行。

(6) 键盘

先宏定义键盘扫描码，再定义大小写键盘下扫描码对应的字符码，用 buffer 保存扫描码，并将键盘初始化为小写键盘。除此之外提供了 getKeyCode 函数来获取键盘扫描码和 getChar 函数来将扫描码转化为对应的字符码。

随后内核执行 loadUMain 函数，它从磁盘上加载用户程序至内存，将用户 SS, ESP, STI, EFLAGS, CS, EIP 等 push 入内核栈，通过 iret 指令跳转到用户态执行。

3. app

在 app/main.c 中调用了 printf 和 scanf，下面分别阐明调用过程和实现机制。

(1) printf

仿照 C 标准函数库中<stdarg.h>头文件，运用一组宏定义处理可变参数。

```
typedef char * va_list;
#define _INTSIZEOF(n) ((sizeof(n) + sizeof(int) - 1) & ~(sizeof(int) - 1))
#define va_start(ap, format) ( ap = (va_list)&format+ _INTSIZEOF(format) )
#define va_arg(ap, type) ( *(type*)((ap += _INTSIZEOF(type)) - _INTSIZEOF(type)) )
#define va_end(ap) ( ap = (va_list)0 )
```

其中，va_list 是用宏定义的标识符，是指向字符类型的指针；va_start(ap,v) 取出 va_list 定义的变量的地址，并加上可变元素的数目；va_arg(ap,t)每次取指针指向的内容，并在宏的内部将指针后移；va_end(ap)；将原指针指向空，以防止野指针的出现。

```
int printf(const char *format,...) {
    va_list ap;
    va_start(ap, format);
    int i=0; // format index
    char buffer[MAX_BUFFER_SIZE];
    int count=0; // buffer index
    int decimal=0;
    uint32_t hexadecimal=0;
    char *string=0;
    char character=0;
    while(format[i]!=0) {
        // TODO: support more format %s %d %x and so on
        buffer[count++]=format[i];
        if(format[i]=='%') {
            count--; i++;
            switch(format[i]) {
                case 'c':
                    character=va_arg(ap, char);
                    buffer[count++]=character;
                    break;
                case 's':
                    string=va_arg(ap, char*);
                    count=str2Str(string, buffer, (uint32_t)MAX_BUFFER_SIZE, count);
                    break;
                case 'x':
                    hexadecimal=va_arg(ap, uint32_t);
                    count=hex2Str(hexadecimal, buffer, (uint32_t)MAX_BUFFER_SIZE, count);
                    break;
                case 'd':
                    decimal=va_arg(ap, int);
                    count=dec2Str(decimal, buffer, (uint32_t)MAX_BUFFER_SIZE, count);
                    break;
                case 'X':
                    count++;
                    break;
            }
        }
        if(count==MAX_BUFFER_SIZE) {
            syscall(SYS_WRITE, STD_OUT, (uint32_t)buffer, (uint32_t)MAX_BUFFER_SIZE, 0, 0);
            count=0;
        }
        i++;
    }
    if(count!=0)
        syscall(SYS_WRITE, STD_OUT, (uint32_t)buffer, (uint32_t)count, 0, 0);
    va_end(ap);
    return 0;
}
```

printf 的实现如上，首先利用 ap 指向 printf 的可变参数，使用 buffer 来缓存要输出的字符。随后将 format 的内容依次装入 buffer 中，如果当前的 format 字符是 '%'，需要特殊处理：若下一个字符是 'c'，'s'，'x'，'d'，需要把 ap 指向的参数利用框架代码提供的 API 转化成字符串存入 buffer 中，并将 ap 指向下一个参数；若下一个字符是 '%'，代表是 '%' 的转义字符，只需在 buffer 装入一次 '%'。在装入 buffer 的过程中，如果 buffer 已满或者 format 读完，则需要调用 syscall。

由于本次实验使用 EAX，EBX 等等这些通用寄存器从用户态向内核态传递参数，对于 printf，将 EAX 设置为 0 表示输出操作，将 ECX 设置为 0 表示使用标准设备输出，将 EDX 设置为 buffer 的地址表示要输出的字符串，将 EBX 设置为 buffer 当前大小表示要输出的字符个数，然后使用 int 0x\$80 来完成系统调用。

先读取 IDTR 寄存器指向的 IDT 中的第 0x80 项门描述符，得知其段选择子是内核代码段选择子且偏移量是 kernel/kernel/dolrq.S 中 irqSyscall 函数首地址。然后从 GDTR 寄存器获得 GDT 的基地址，并在 GDT 中查找，获得内核代码段描述符。因为 CPL 和门描述符中的 DPL 都为 3，所以不会产生 GP 异常。而内核代码段描述符的 DPL 是 0，所以发生特权级变化：读取 TR 寄存器，访问 TSS；选取 TSS 中记录的与 DPL 一致的 SS 与 ESP 切换堆栈；在切换后的堆栈中保存之前堆栈的 SS 与 ESP。接着在堆栈中保存 EFLAGS 和下条指令的 CS 与 EIP。再根据门描述符装载新的 CS 和 EIP，即执行中断处理程序 irqSyscall 函数。

irqSyscall 函数在内核栈中保存 Error Code 和中断向量 0x80，跳转至 asmDolrq 执行。asmDolrq 在内核栈中先保存各通用寄存器，再调用 kernel/kernel/irqHandle.c 中的 irqHandle 函数。irqHandle 根据中断向量 0x80 调用 syscallHandle，syscallHandle 根据 EAX 为 0 调用 syscallWrite 进行输出操作，syscallWrite 根据 ECX 为 0 调用 syscallPrint 进行标准输出操作，syscallPrint 将要打印的内容输出到显示屏上，在显示屏满时下移一行，更新光标的位置。最后调用堆栈中的各函数依次返回直至 asmDolrq，它将各通用寄存器恢复，使用 iret 指令从高特权级返回低特权级：硬件依次从当前栈顶弹出 EIP，CS，EFLAGS，返回执行产生中断时的程序，并且继续弹出 ESP 以及 SS，即切换堆栈。

(2) scanf

```
int scanf(const char *format,...) {
    // TODO: implement scanf function, return the number of input parameters
    va_list ap;
    va_start(ap, format);
    int i=0; // format index
    char buffer[MAX_BUFFER_SIZE];
    int ret=0, avail=0;
    int count=0; // buffer index
    while(format[i]!='\0') {
        matchWhiteSpace((char*)format, MAX_BUFFER_SIZE, &i);
        if(buffer[count]!='\0') {
            syscall(SYS_READ, STD_IN, (uint32_t)buffer, (uint32_t)MAX_BUFFER_SIZE, 0, 0);
            count=0;
        }
        matchWhiteSpace(buffer, MAX_BUFFER_SIZE, &count);
        if(buffer[count]!='\0') {
            syscall(SYS_READ, STD_IN, (uint32_t)buffer, (uint32_t)MAX_BUFFER_SIZE, 0, 0);
            count=0;
        }
        if(format[i]=='%') {
            ret++; i++;
            avail=0;
            while('0'<=format[i]&&'9')==format[i])
                avail=10*avail+format[i++]-'0';
            switch(format[i]) {
                case 'c':
                    *((char*)va_arg(ap, char*))=buffer[count++];
                    break;
                case 's':
                    str2Str2((char*)va_arg(ap, char*), avail?avail:MAX_BUFFER_SIZE, buffer, MAX_BUFFER_SIZE, &count);
                    break;
                case 'x':
                    str2Hex((int*)va_arg(ap, int*), buffer, MAX_BUFFER_SIZE, &count);
                    break;
                case 'd':
                    str2Dec((int*)va_arg(ap, int*), buffer, MAX_BUFFER_SIZE, &count);
                    break;
            }
            count--;
        }
        i++; count++;
    }
    va_end(ap);
    return ret;
}
```

scanf 的实现如上，采用双指针法，用 `i` 和 `count` 分别指向 `format` 和 `buffer`，利用框架代码提供的 API 跳过空白符，如果当前的 `format` 字符是 `'%'`，需要特殊处理：若下一个字符是 `'c'`，`'s'`，`'x'`，`'d'`，需要把 `buffer` 中从 `count` 开始的字符串利用框架代码提供的 API 转化成对应内容存入 `ap` 指向的参数中，并将 `ap` 指向下一个参数。在此过程中，一旦 `buffer[count]` 为 0，说明 `buffer` 中的字符串已经用完，需要继续输入，调用 `syscall`。

与 `printf` 调用 `syscall` 的过程类似，依次执行直至 `syscallHandle`，与 `printf` 不同的是，`syscallHandle` 调用 `syscallRead` 进行输入操作，`syscallRead` 调用 `syscallScan` 进行标准输入操作，在 `syscallScanf` 中，先使用 `kernel/kernel/keyboard.c` 中提供的 `getKeyCode` 函数不断获取键盘扫描码并将其保存，直至获取的是回车键的扫描码为止，接着利用 `getChar` 函数将扫描码转化成字符串，之后过程与 `printf` 相同，调用堆栈各函数依次返回。

实验收获

1. 加深了基于中断实现系统调用全过程的理解。
2. 了解了一个简易操作系统的部分实现机制。