Reference

Orange 一个操作系统的实现

Intro

本次实验需要完成的内容已经在之前实现完成，接下来简单介绍一下保护模式下系统调用的实现

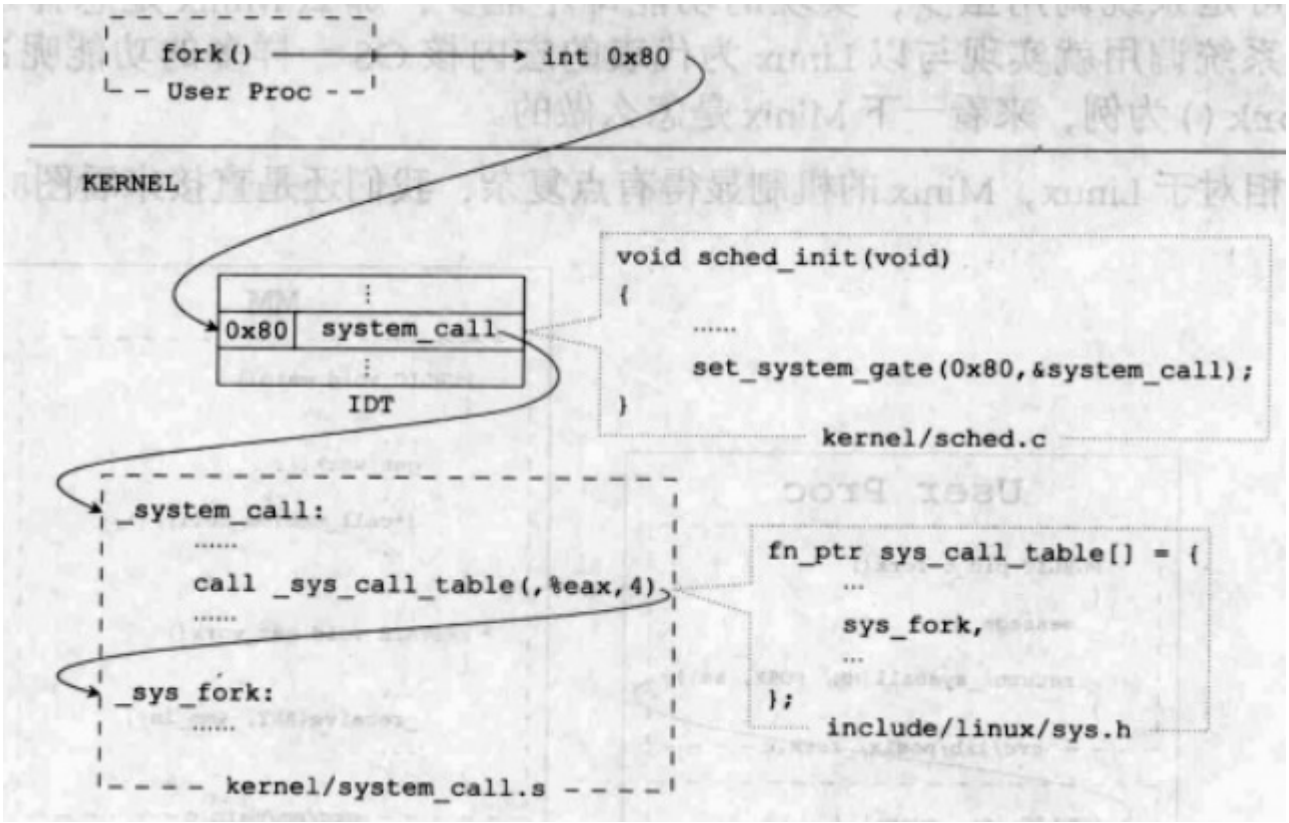
需求实现

实现了系统调用，并实现了几个有实用功能的系统调用，并根据验证，可以在用户程序分开编译运行成功.

1. printf
2. getchar
3. sendrec(消息传递)

Linux系统调用

下面简单介绍一下linux中`fork`的流程，来研究系统调用的实现.



注意到系统调用实际上是中断的封装，根据参数选择合适的系统调用函数.

系统调用的实现

系统调用函数

```
sendrec:
    mov eax, _NR_sendrec
    mov ebx, [esp + 4]  ; function
    mov ecx, [esp + 8]  ; src_dest
    mov edx, [esp + 12] ; p_msg
    int INT_VECTOR_SYS_CALL
    ret
```

1. 用`eax`传递选择参数
2. 用其余寄存器传递函数参数
3. 调用系统中断

`systemcall`中断

```
; ====================================================================================
;                                 sys_call
; ====================================================================================
sys_call:
        call    save

        sti
    push    esi

    push    dword [p_proc_ready]
    push    edx
    push    ecx
    push    ebx
        call    [sys_call_table + eax * 4]
    add esp, 4 * 4

    pop esi
        mov     [esi + EAXREG - P_STACKBASE], eax
        cli

        ret
```

注意到几个细节

1. `call save`，这是为了在进程栈保存数据寄存器，段寄存器，并将栈切换到内核栈
2. 开中断，因为CPU处理中断会自动关中断
3. 系统调用将当前进程指针作为额外的参数
4. 将函数参数入栈
5. 根据系统调用表调用正确的函数
6. 关中断，因为CPU会自动开中断

getchar实现的细节

实现三级缓冲和`RingBuffer`，处理键盘输入`backspace`

因为如果键盘输入的是回退键，buffer就会变得很复杂，所以实现三个缓存，第一个缓存记录键盘的make和break，第二个缓存记录当前输入缓冲区记录的字符，如果不为`\b`，则加入ring buffer，如果为`\b`，则pop ring buffer，如果为`\n`，就清空缓冲区，放入第三个缓冲，等待中断读取字符.

一级缓冲，位于键盘驱动

```
static KB_INPUT kb_in;

void keyboard_handler(int irq)
{
    u8 scan_code = in_byte(KB_DATA);

    if (kb_in.count < KB_IN_BYTES) {
        *(kb_in.p_head) = scan_code;
        kb_in.p_head++;
        if (kb_in.p_head == kb_in.buf + KB_IN_BYTES) {
            kb_in.p_head = kb_in.buf;
        }
        kb_in.count++;
    }
}
```

二级缓冲，位于tty，为RingBuffer，每次键盘驱动发送信息给tty，tty会执行两种工作

1. 处理显示
2. 将字符放入ringbuffer，或从ringbuffer中删除字符

```
struct RingBuffer
{
    u8 buffer[RingBufferSize];

    u32 head;
    u32 tail;
};

void in_process(Tty* p_tty, u32 key)
{
        char output[2] = {'\0', '\0'};

        if (!(key & FLAG_EXT)) {
            put_key(p_tty, key);
            push_key(&p_tty->rb, key);
        }
        else {
            ...
            case BACKSPACE:
                put_key(p_tty, '\b');
                back_key(&p_tty->rb);
                break;
            ...
        }
}
```

三级缓冲，也是一个ringbuffer，当二级缓冲接收到一个`\n`，二级缓冲将全部字符发送到三级缓冲，等待`sys_getch()`

```
switch(raw_code) {
    case ENTER:
    put_key(p_tty, '\n');
    push_key(&p_tty->rb, '\n');
    while (ring_length(&p_tty->rb)) push_key(&p_tty->gb, pop_key(&p_tty->rb));
    break;
```

printf实现

printf实现

- printf调用vsprintf处理出字符串
- vsprintf调用write系统调用
- write通过中断实现，调用sys_write
- sys_write接受一个额外参数，当前进程指针

可变参函数的实现原理是c/cpp调用约定：调用者维护堆栈，即调用者push参数，被调用函数返回后，调用函数维护`esp`

```
int printf(const char *fmt, ...)
{
    int i;
    char buf[256];

    va_list arg = (va_list)((char*)(&fmt) + 4); /*4是参数fmt所占堆栈中的大小*/
    i = vsprintf(buf, fmt, arg);
    buf[i] = '\0';
    write(buf);

    return i;
}
```

printf调用了vsprintf，由vsprintf根据字符串判断多的参数的数目和类型

```
int vsprintf(char *buf, const char *fmt, va_list args)
{
    char*   p;

    va_list p_next_arg = args;
    int m;

    char    inner_buf[STR_DEFAULT_LEN];
    char    cs;
    int align_nr;

    for (p=buf;*fmt;fmt++) {
        if (*fmt != '%') {
            *p++ = *fmt;
            continue;
        }
        else {      /* a format string begins */
```

```
        align_nr = 0;
    }

    fmt++;

    if (*fmt == '%') {
        *p++ = *fmt;
        continue;
    }
    else if (*fmt == '0') {
        cs = '0';
        fmt++;
    }
    else {
        cs = ' ';
    }
    while (((unsigned char)(*fmt) >= '0') && ((unsigned char)(*fmt) <= '9')) {
        align_nr *= 10;
        align_nr += *fmt - '0';
        fmt++;
    }

    char * q = inner_buf;
    memset(q, 0, sizeof(inner_buf));

    switch (*fmt) {
    case 'c':
        *q++ = *((char*)p_next_arg);
        p_next_arg += 4;
        break;
    case 'x':
        m = *((int*)p_next_arg);
        i2a(m, 16, &q);
        p_next_arg += 4;
        break;
    case 'd':
        m = *((int*)p_next_arg);
        if (m < 0) {
            m = m * (-1);
            *q++ = '-';
        }
        i2a(m, 10, &q);
        p_next_arg += 4;
        break;
    case 's':
        strcpy(q, (*((char**)p_next_arg)));
        q += strlen(*((char**)p_next_arg));
        p_next_arg += 4;
        break;
    default:
        break;
    }
```

```
        int k;
        for (k = 0; k < ((align_nr > strlen(inner_buf)) ? (align_nr - strlen(inner_buf)) : 0);
k++) {
            *p++ = cs;
        }
        q = inner_buf;
        while (*q) {
            *p++ = *q++;
        }
    }

    *p = '\0';

    return (p - buf);
}
```

```
write:
        mov     eax, _NR_write
        mov     edx, [esp + 4]
        int     INT_VECTOR_SYS_CALL
        ret
```

sys_write通过进程指针写入正确console

```
int sys_write(u32 u1, u32 u2, char* s, Process* p_proc)
{
        char * p = s;
        char ch;

        while ((ch = *p++) != 0) {
                if (ch == MAG_CH_PANIC || ch == MAG_CH_ASSERT)
                        continue; /* skip the magic char */

                out_char(tty_table[p_proc->nr_tty].p_console, ch);
        }

        return 0;
}
```

这里因为系统调用的参数固定为四个，所以填充了无用参数

心得体会

1. 系统调用通过中断实现十分方便，但是有个潜在问题，可传递的参数过少(3个)，而且有时候需要填充无用参数
2. 保护模式下完成任务十分简单，只要基本的框架(分页，GDT表，进程控制)搭好，剩下一路顺风.