4.日志及printf格式化输出

实现基本的信息输出

RS232是一种早期PC机提供的串行通信接口，其通过两根信号线：发送信号、接收信号，便能实现与外部的数据输入输出通信。现代的计算机已经不提供这种通信接口了，但是qemu有实现对这种接口的模拟，所以我们仍然可以使用。

采用RS232接口后，使用log_printf打印的字符串信息，EMU都能完整的显示在窗口中，并且能自动地对换行等字符进行转换处理显示。在Qemu的Serial窗口中查看串口的输出

换行与回车

课程中提到了换行（\n）和回车（\r），这两个字符的区别总结如下：

- 换行（\n）：将光标移到下一行，列号不变
- 回车（\n）：将光标移到本行的开头

因此，如果需要将光标移到下一行开始，需要使用\r\n。
不过，不同的显示终端对于\n的解释不同。在windows中编程时，使用printf输出，其会将\n自动转换成\r\n。

串口初始化

```
#define COM1_PORT           0x3F8       // RS232端口0初始化
/**
 * @brief 初始化日志输出
 */
void log_init (void) {
    outb(COM1_PORT + 1, 0x00);    // Disable all interrupts
    outb(COM1_PORT + 3, 0x80);    // Enable DLAB (set baud rate divisor)
    outb(COM1_PORT + 0, 0x03);    // Set divisor to 3 (lo byte) 38400 baud
    outb(COM1_PORT + 1, 0x00);    //                  (hi byte)
    outb(COM1_PORT + 3, 0x03);    // 8 bits, no parity, one stop bit
    outb(COM1_PORT + 2, 0xC7);    // Enable FIFO, clear them, with 14-byte
threshold

    // If serial is not faulty set it in normal operation mode
    // (not-loopback with IRQs enabled and OUT#1 and OUT#2 bits enabled)
    outb(COM1_PORT + 4, 0x0F);
}
```

```
/**
 * @brief 日志打印
 */
void log_printf(const char * fmt, ...) {
    char str_buf[128];
    va_list args;

    kernel_memset(str_buf, '\0', sizeof(str_buf));

    va_start(args, fmt);
    kernel_vsprintf(str_buf, fmt, args);
    va_end(args);


    const char * p = str_buf;
    while (*p != '\0') {
        while ((inb(COM1_PORT + 5) & (1 << 6)) == 0);
        outb(COM1_PORT, *p++);
    }
    outb(COM1_PORT, '\r');
    outb(COM1_PORT, '\n');
}
```

若干字符串和内存操作函数实现

```
void kernel_strcpy (char * dest, const char * src);
void kernel_strncpy(char * dest, const char * src, int size);
int kernel_strncmp (const char * s1, const char * s2, int size);
int kernel_strlen(const char * str);
void kernel_memcpy (void * dest, void * src, int size);
void kernel_memset(void * dest, uint8_t v, int size);
int kernel_memcmp (void * d1, void * d2, int size);
void kernel_itoa(char * buf, int num, int base);
void kernel_sprintf(char * buffer, const char * fmt, ...);
void kernel_vsprintf(char * buffer, const char * fmt, va_list args);
```

可变参数的使用

函数名（参数1, 参数2, ...)的形式，即在最后有名称的参数后面增加一个...的参数。

```
#include <stdarg.h>
```

```
定义va_list类型的变量，如ap，用于存放取可变参数的一些信息
使用va_start(ap, 最后一个有名字的参数-参数2)：初始化ap
依次使用va_arg(ap, type)获取传入的可变部分的参数
使用va_end(ap)结束可变参数的取参，释放相应的资源
```

可变参数的处理分析

如func_arg(a, b, c, d, e)，则栈中压入的参数从上（高地址）到下（低地址）依次为e, d, c, b , a。在函数内部，只有func_arg(int a, ...)中的a参数有名字可以引用，而要获取其它参数，则需要借助a找到对应的栈位置，然后再在栈中依次往上（高地址）逐个去取出相应的参数。因此，在获得参数时：

- 首先，使用va_list定义一个变量，如args，这个变量会被va_start()初始化。我猜测，在va_start()内部，可能维护了一个指针，指向了栈中参数所在的位置。例如，在调用时va_start(args, a)，会将args指向了参数a的后一个参数（更高地址）。
- 然后，要获取这个参数，通过va_arg(args, type)实现。其会将当前args指向的栈单元中的值取出来。取出来后转换成什么类型，则由type指定（内部可能做了强制类型转换）。
- 取完之后，args内部的指针再继续往一个参数移动。

字符串的格式化输出

kernel_vsprintf分析

kernel_vsprintf内部定义一个简单的状态机对字符串进行格式化，具体为：

- NORMAL：表示正在对普通的需要直接显示的字符进行处理，处理方式为直接写入字符串缓存中
- READ_FMT：表示正在进行参数的转换处理，例如遇到了%s，正在从可变参数列表中取出字符串写入字符串缓存中

该函数功能类似于C库中的vsprintf

```
/**
 * 格式化字符串
 */
void kernel_vsprintf(char * buffer, const char * fmt, va_list args) {
    enum {NORMAL, READ_FMT} state = NORMAL;
    char ch;
    char * curr = buffer;
    while ((ch = *fmt++)) {
        switch (state) {
            // 普通字符
            case NORMAL:
```

```
                if (ch == '%') {
                    state = READ_FMT;
                } else {
                    *curr++ = ch;
                }
                break;
            // 格式化控制字符，只支持部分
            case READ_FMT:
                if (ch == 'd') {
                    int num = va_arg(args, int);
                    kernel_itoa(curr, num, 10);
                    curr += kernel_strlen(curr);
                } else if (ch == 'x') {
                    int num = va_arg(args, int);
                    kernel_itoa(curr, num, 16);
                    curr += kernel_strlen(curr);
                } else if (ch == 'c') {
                    char c = va_arg(args, int);
                    *curr++ = c;
                } else if (ch == 's') {
                    const char * str = va_arg(args, char *);
                    int len = kernel_strlen(str);
                    while (len--) {
                        *curr++ = *str++;
                    }
                }
                state = NORMAL;
                break;
        }
    }
}
```

利用assert辅助调试

在GCC编译参数中添加预定义的宏的方法，具体为-D宏名。

```
#ifndef RELEASE
#define ASSERT(condition)     \
    if (!(condition)) panic(__FILE__, __LINE__, __func__, #condition)
void panic (const char * file, int line, const char * func, const char *
cond);
#else
```

```
#define ASSERT(condition)    ((void)0)
#endif
```