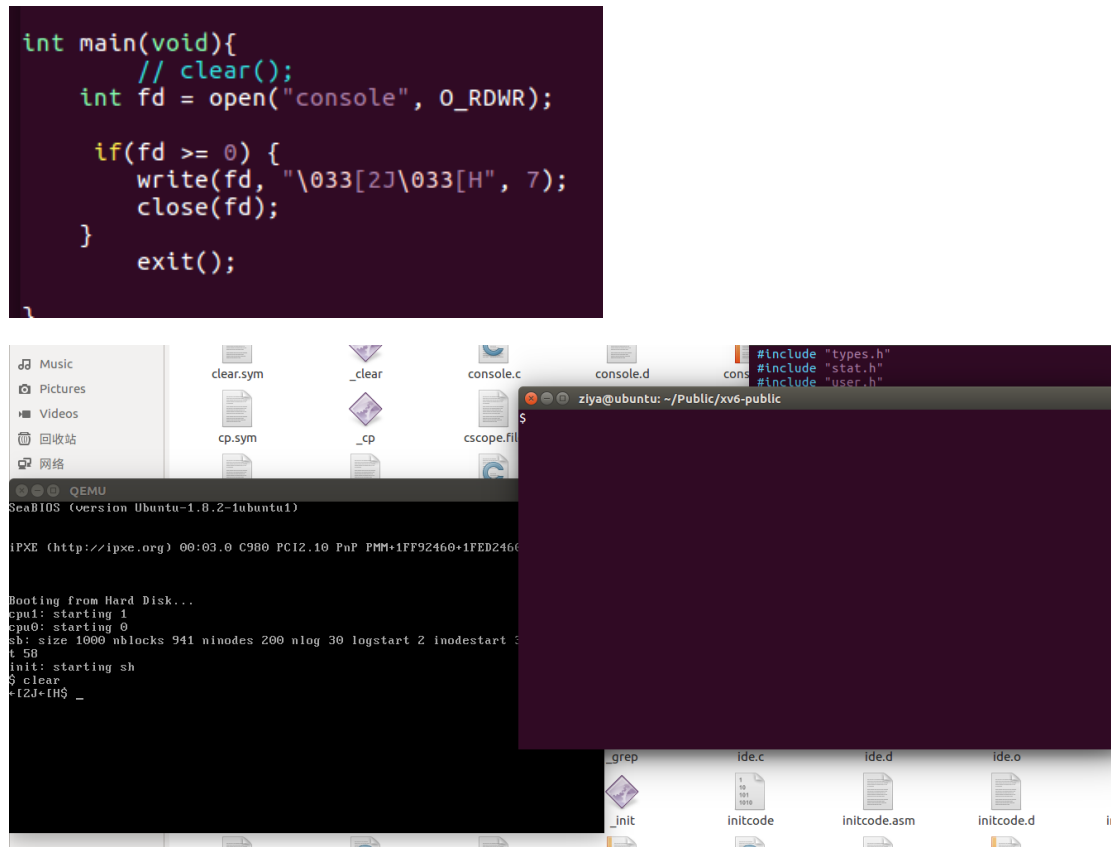


在 xv6 中实现 linux 的 clear 操作

在 terminal 实现 clear 操作

效果演示



写代码过程

查看 linux 清除是如何实现的，`strace clear` 查看输入 `clear` 后屏幕出现的字符，

```
write(1, "\33[3;J\33[H\33[2J", 12) = 12
exit_group(0)
```

由此可知输入这段字符可以清除 terminal

打开当前终端文件，获取文件描述符

```
// clear();
int fd = open("console", O_RDWR);
```

写入这些字符

```
write(fd, "\033[2J\033[H", 7);
```

就达到了清屏的目的，可以格式化输出的函数都可以使用这段字符，比如 `printf` `cprintf`

在 console 中实现 clear 操作

效果演示

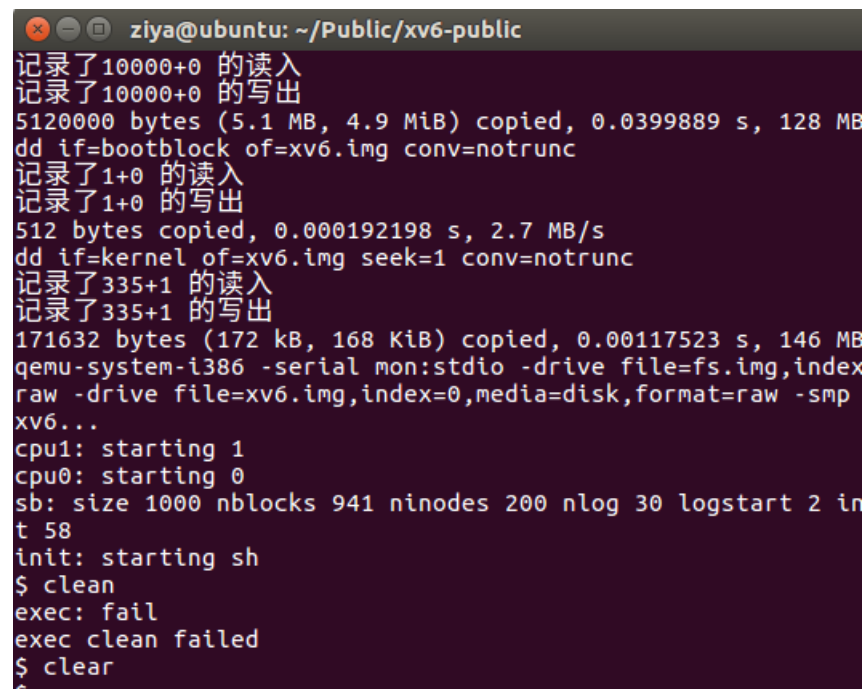


```
ziya@ubuntu: ~/Public/xv6-public
#include "types.h"
#include "stat.h"
#include "user.h"
#include "fcntl.h"

int main(void){
    clear();
    /* int fd = open("console", O_RDWR);

    if(fd >= 0) {
        write(fd, "\033[2J\033[H", 7);
        close(fd);
    }
    exit();
}
```

terminal 没有清屏，console 清屏成功。



```
ziya@ubuntu: ~/Public/xv6-public
记录了10000+0 的读入
记录了10000+0 的写出
5120000 bytes (5.1 MB, 4.9 MiB) copied, 0.0399889 s, 128 MB/s
dd if=bootblock of=xv6.img conv=notrunc
记录了1+0 的读入
记录了1+0 的写出
512 bytes copied, 0.000192198 s, 2.7 MB/s
dd if=kernel of=xv6.img seek=1 conv=notrunc
记录了335+1 的读入
记录了335+1 的写出
171632 bytes (172 kB, 168 KiB) copied, 0.00117523 s, 146 MB/s
qemu-system-i386 -serial mon:stdio -drive file=fs.img,index=0,media=disk,format=raw -smp 1 -m 128M -cdrom xv6.img -nographic
xv6...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 in
t 58
init: starting sh
$ clean
exec: fail
exec clean failed
$ clear
c
```



写代码过程

- 要清除 console 屏幕，需要操纵显示器硬件，更改显存中的内容，显存物理地址是从 0xb8000 开始，只需在这段地址中输入规定好的数值，就可以更改屏幕信息，但进入保护模式之后不能直接读写物理地址，用 P2V 把物理地址变成虚拟地址，在 xv6 中是 $v + 0x80000000 = p$ ，这种映射内存的方式是直接映射，但不能判断 xv6 没有用到 MMU，可能是因为有的地址用到了直接映射，有的地址用到了 MMU
- 得到了想要的地址，用一个 *crt 指针指向它方便操作，

```
for (i = 0; i < 25 * 80; i++) {
    crt[i] = (' ' | 0x0700);
}
```

crt 地址后面 25*80 个是对应 console 屏幕显

示信息，整个屏幕它有 80 列 25 行。

```
// 将硬件光标移动到屏幕左上角
outb(CRTPORT, 14);
outb(CRTPORT+1, 0);
outb(CRTPORT, 15);
outb(CRTPORT+1, 0);
```

- outb 是封装好的汇编操作，直接通过端口与显示器硬件传输信息，由此可知和设备操作有两个方式：1.内存映射 2.port 端口。
CRTPORT 是端口号，向端口号输入正确的值可以达到控制设备的目的
Console 清屏就写完了
- 如果要试试效果可以注册一个快捷键 CTRL+L 。

```

void
consoleintr(int (*getc)(void))
{
    int c, doprocdump = 0;

    acquire(&cons.lock);
    while((c = getc()) >= 0){
        switch(c){
            case C('P'): // Process listing.
                // procdump() locks cons.lock indirectly; invoke later
                doprocdump = 1;
                break;
            case C('U'): // Kill line.
                while(input.e != input.w &&
                    input.buf[(input.e-1) % INPUT_BUF] != '\n'){
                    input.e--;
                    consputc(BACKSPACE);
                }
                break;
            case C('H'): case '\x7f': // Backspace
                if(input.e != input.w){

```

这个是中断程序快捷键的注册，直接在上面加一个 case 分支就可以了

- 也可以把上面清除 terminal 清屏的操作写在 consoleclear 里面，因为 cprintf() 函数在这也有效
- 但是 console 代码不能写在上面的 clear.c 中，因为修改内存和使用端口必须在内核态，写在内核态的代码用户态不能直接调用，会有段错误(权限错误)，使用方法是注册一个系统调用，系统调用会把当前线程切入内核态，再使用该代码，这段过程我们只需注册好，切内核态操作操作系统来实现，
- 下面是注册系统调用过程。
想要找到注册系统调用有关的代码，可以使用 grep 查找，具体是这几个文件
user.h

```
int clear(void);
```

usys.s

```
SYSCALL(sysclear,
SYSCALL(clear)
```

syscall.h

```
#define SYS_close 21
#define SYS_clear 22
```

Syscall.c

```
extern int sys_cls(void);
```

把我们写的函数导入进来，让它知道有这个函数指针

```
[SYS_clear] sys_cls,
};
```

函数指针注册

console.c

```
int sys_cls(void) {
    consoleclear();
    return 0;
}
```

把它封装成系统调用，可以加锁，不加也可以运行。



- 我们看到根据注册在 user.h 的系统调用都能连起来除了问号那里，问号那里是如何对应起来的？

- 打开 usys.S，开头部分解释了

```
#define SYSCALL(name) \
    .globl name; \
    name: \
        movl $SYS_ ## name, %eax; \
        int $T_SYSCALL; \
        ret
```

这是一个宏定义，.globl 表示其他文件可以使用下面函数，把 name 替换成 clear，clear

```
movl eax, SYS_clear;
```

```
int 22::
```

```
ret
```

这里把 clear 和 22 号中断 SYS_clear 联系起来了，eax 应该是返回值，告诉调用者中断名字是什么

- VGA 编程指南([VGA/SVGA 视频编程 - 标准 VGA 芯片组参考 \(osdever.net\)](http://osdever.net))

- 我们要看的是

- 文本模式

- [VGA 文本模式操作](#) -- 有关的详细信息 文本模式操作，包括属性和字体。
 - [操作文本模式光标](#) -- 详细信息 控制光标的外观和位置。

- 背景怎么清空的？

属性 属性
字节被划分 分为两个四位字段。7-4 中的字段用作索引到 调色板注册字体使用的背景颜色 0-3-0 中的字段用作调色板寄存器的索引 字体位为 1 时使用的前景。该属性也可以 控制其他几个方面，这些方面可能会改变角色的方式 显示。
如果 [属性](#) 字段设置为 1，启用字符闪烁。闪烁时 已启用，属性的背景色位 3 强制为 0 生成目的，如果字符的属性字节的第 7 位 设置为 1，前景色在前景和 背景，导致角色闪烁。确定闪烁率 除以垂直同步速率除以 32。
如果属性的位 2-0 字节等于 001b，属性字节的 6-4 位等于 000b，然后下划线指定的字符行 [位置](#) 字段将替换为前景色。注意如果行 由 “[下划线位置](#)” 字段指定 通常不显示，因为它大于最大扫描线 显示的字符，则下划线功能有效 禁用。
属性字节的第 3 位，以及为其相应字符选择前景色，也用于在两个可能的字符集之间进行选择（请参阅下面的 [字体](#)。如果两个字符集相同，则位有效运行 仅选择前景色。

```
' ' | 0x0700)
```

- ◆ 前面的空格是表示的字符，后面是属性，变成二进制是
0000 0111 0000 0000
后面一个字节应该没什么用，只看前面一个字节
7-4 是的背景颜色，000 表示黑色即默认背景色
第 7 位表示没有闪烁或其他特殊属性
111 表示字符的前景色，111 表示白色，即透明色

■ 光标如何移到左上角

```
static inline void
outb(ushort port, uchar data)
{
    asm volatile("out %0,%1" : : "a" (data), "d" (port));
}
```

这段代码封装了一个汇编函数 相当于 out port data
out 指令是从 cpu 到设备传输数据的指令，不同端口号代表不同设备
所以它是把 14h->0x3d4 0->0x3d5 15h->0x3d4 0->0x3d5

port 端口号为 0x3d4 是寄存器索引，赋予不同的值获得不同的寄存器
0x3d5 端口是用来给上面获得的寄存器赋值

```
outb(CRTPORT, 14);
outb(CRTPORT+1, 0);
outb(CRTPORT, 15);
outb(CRTPORT+1, 0);
```

1,3 两句是获得两个寄存器
14h 和 15h 代表寄存器位置

下划线位置寄存器 (索引 14h)

7	6	5	4	3	2	1	0
	DW	四分部	下划线位置				

DW - 双字寻址
“当此位设置为 1 时，内存地址是双字地址。请参阅 CRT 模式控件中字/字节模式位 (位 6) 的说明 寄存器 ”

DIV4 - 将内存地址时钟除以4
“ 当此位设置为1时，内存地址计数器将时钟化 字符时钟除以 4，用于双字地址 被使用。

下划线位置
“这些位指定字符行的水平扫描线 下划线出现。编程值是所需的扫描线 减去 1。”

启动垂直消隐寄存器 (索引 15h)

7	6	5	4	3	2	1	0
开始垂直消隐							

• 开始垂直消隐

这包含“开始垂直消隐”字段的位 7-0。此字段的第 8 位位于溢出寄存器中，位 9 位于最大扫描线寄存器中。此字段确定垂直消隐期的开始时间，并包含 第一个开始的垂直扫描线计数器的值 消隐的垂直扫描线。

第二句和第四句是给这两个寄存器赋值为 0 即把光标移到左上方并刷新屏幕
14h 赋值为 0 的意思：DW 和 DIV4 为默认值，下划线位置为 0
15h 赋值为 0 的意思：开始垂直消隐，即刷新屏幕