设备驱动 实验报告

1 实验原理及步骤

Xinu实现了面向串口的驱动程序，但是Xinu的终端支持是完全基于串口硬件完成的，没有针对普通的CGA显示屏的输入输出功能，也就是所在QEMU刚启动的界面没有办法通过键盘实现输入和输出功能。因此本次实验要完成的是一个支持“键盘+显示屏”的简易驱动程序。

1.1 设备声明

首先要完成的是设备名绑定。Xinu使用小的整数来识别设备，并允许命令解释器在程序启动时将每个整数链接到具体的设备，对于每一个设备，操作系统以党指导每一个抽象的I/O操作对应于哪个驱动器函数，也知道每一个抽象设备对应于哪个底层硬件设备。这些对应都是在编译时通过将Configuration文本文件中的内容转化为conf.c以及conf.h两个文件。

在config/Configuration中增加有关键盘线视频的设备声明如下：

```
kbd:
    on uart
        -i kbdvgainit   -o ionull   -c ionull
        -r ionull       -g kbdgetc  -p vgaputc
        -w ionull       -s ionull   -n ionull
        -intr kbddisp
        -irq 33
```

其中kbd是我自己定义的type_name，而uart是现有的hardware_name，之后通过缩写来得到每一个I/O操作所对应的函数，由于我们只实现了输入和输出单个的字符，而没有实现read或者write 等其他的功能，所以只是指定了九个抽象I/O操作之中的三个：

- -i init初始化操作，对应了kbdvgainit(void)函数，会在后面介绍
- -g getc读取单个字符操作，对应kbdgetc(struct dentry *devptr)函数，这个由老师提供了实现代码
- -p putc输入单个字符操作，对应vgaputc(struct dentry *devptr, char ch)函数，会在后面介绍

其余的内容全部都是指定了ionull，代表对应的驱动函数并没有实现，此外，还制定了这个设备对应的中断处理函数和中断号33

1.2 屏幕字符串处理

接下来的工作是编写vgaputc()函数，用于完成回显、换行、退格等操作.首先，我先编写了vgaputc可能会使用到的几个辅助函数：

```
uint16 get_cursor(){
    uint16 pos;
    outb(0x3d4,14);
    pos=inb(0x3d5)<<8;
    outb(0x3d4,15);
    pos|=inb(0x3d5);
}
```

- get_cursor()，用于获得鼠标的位置，由于显示器I/O端口为0x3d4 和0x3d5,所以可以通过读取这些端口来获得光标的位置

```
void set_cursor(uint16 pos){
    outb(0x3d4,14);
    outb(0x3d5,pos>>8);
    outb(0x3d4,15);
    outb(0x3d5,pos);
}
```

- set_cursor，用于设置光标的位置，同理，通过向I/O端口0x3d4和0x3d5写入pos的高八位和第八位来实现

```
uint32 line_roundup(uint16 pos){
    uint16 newpos;
    newpos=((pos+80)/80)*80;
    return newpos;
}
```

- line_roundup(),在换行的时候需要，用于计算光标应处的位置。由于屏幕的容量是24×80，所以通过这种变换，可以让光标从当前位置改变到下一行的起始位置，从而实现换行的功能

最终vgaputc()函数实现如下所示：

```
devcall vgaputc(struct dentry *devptr, char ch){
    static uint32 pos=0;
    uint32 base=0xb8000;
    static uint16 *ptr=0xb800;
    char blank=' ';
    int i,j,k;
    if(ch=='\b'){
        if(pos%80==0){
            if(pos==0)
                return OK;
            pos=(pos/80)*80-1;
            set_cursor(pos);
            return OK;
        }
        pos--;
        ptr=(uint16*)(base+pos*2);
        *ptr=(blank & 0xff)|0x0700;
        set_cursor(pos);
    }
    else if(ch=='\r' || ch=='\n'){
        pos=get_cursor();
        pos=line_roundup(pos);
        if(pos==24*80){
            int i;
            for(i=0;i<23*160;i+=2){
                uint16* oldline;
                uint16* newline;
                oldline=(uint16*)(base+i);
                newline=(uint16*)(base+i+160);
                *oldline=*newline;
            }
            for(i=23*160;i<24*160;i+=2){
                ptr=(uint16*)(base+i);
                *ptr=(blank & 0xff)|0x0700;
            }
```

```
            set_cursor(23*80);
            pos=23*80;
        }
        else{
            set_cursor(pos);
        }
    }
    else {
        ptr=(uint16*)(base+pos*2);
        *ptr=(ch& 0xff)|0x0700;
        pos+=1;
        set_cursor(pos);
    }
}
```

将所有接收到的字符分为3类，一类是普通字符，一类是\b,另外一类是\r或者\n，分别采用不同的处理方法。

- 对于普通字符，处理方法最为简单：

  pos是一个生命在函数之中的static静态变量，记录了当前光标所处的位置（其实这个地方也可以通过get_cursor()函数来得到，但是为了避免每次敲击一个字符都需要调用这个函数，使用一个静态变量来存储似乎可以更加简单一点）

  base是0xb8000，这个位置是黑白显示适配器在内存之中的映射位置，从这个位置开始的32kb，写到这块空间中的字符将会被显示在屏幕上。显示器界面中一个字符需要占据两个字节的空间，因为前一个字节表示这个字符的颜色属性如字体颜色、字体亮度、背景色、字体是否闪烁等等，为了方便起见，统一使用0x07(背景色为黑色，前景色为白色)作为第一个字节，也就是使用一个uint16类型的指针，通过让(ch & 0xff)|0x0700，作为一个16位的无符号数来设置这两个连续的字节，使之可以显示我们想要的字符。

  由于一个字符占两个字节，所以我们写入的目标位置相对于base的偏移量并不是pos，而是二倍的关系。之后让pos加一并重新设置光标的位置即可。

- 对于'\b'字符，需要实现回退，这里我的处理方法是让pos减一，从而让光标回退到前一个位置，这个位置就是我们需要删除的位置。在这个位置我们输出一个blank空白字符覆盖掉我们原来输出的字符，从而让实际效果看起来就好像我们是清除了字符一样。

  需要特殊处理的情况是光标位于一行开始的情况，这里如果键入'\b'的位置是第一行，那么应当没有任何反应，否则的话就应当回退到上一行的最后一个位置处。

- 对于'\n' 或者'\r'字符，需要实现换行，这里就需要使用到前面所提到的line_roundup函数，计算光标位置使之到达下一行的开头并设置光标位置即可。

  特殊地，如果显示屏已经满了，也就是当前光标已经处于24行的位置了，那么还需要额外处理，也就是"清屏"。这里我的实现方法是，首先从第一行开始，每一个字符都等于对应地下一行中对应位置的字符，也就是24行整体上上移，第一行自然被第二行所覆盖掉。这个时候，还要注意最后一行其实是重复了两次，第二十四行的位置本身以及上移所覆盖的第二十三行。此时，应当将第二十四行直接清空，也就是使用blank字符全部覆盖一遍。

1.3 中断处理与初始化

```
devcall kbdvgainit(void){
    uint16* ptr;
    char blank=' ';
    uint32 base=0xb8000;
    int i=0;
    for(i=0;i<24*160;i+=2){
        ptr=(uint16*)(base+i);
        *ptr=(blank&0xff)|0x0700;
    }
    set_cursor(0);
    set_evec(33, kbddisp);
}
```

对于kdbvgainit()初始化函数，所需要做的工作主要有两个，首先将从0xb8000位置开始的24行字符全部清空一遍，方法是使用指针遍历并用blank覆盖原来可能有的字符。然后设置光标位置位初始位置，也就是0，屏幕第一行的第一个字符。最后，需要设置中断。由于键盘的中断向量号是33，所以只需要设置33号中断的位置为kbddisp中断分派器即可。

```
void kbdhandler(void){
    int ch;
    struct dentry* devptr;
    /* complete (or modify) the code s.t. it works */
    while((ch =kbdgetc(devptr)) >= 0)
    {
        vgaputc(devptr,ch);
    }

}
```

kbdhandler()是kdbdisp引导到的中断处理函数，这个处理也比较简单，因为kbdgetc之中已经写好了，并且返回值也有明确的含义：如果返回-1，表示没有任何输出；返回0，表示当前操作时控制字符或者是按键弹起，如果是其他返回值，就表示是具体的、有效的按键。因此，我们只需要处理kbdgetc的返回值大于等于0的情况就可以了，其余的情况不需要考虑。

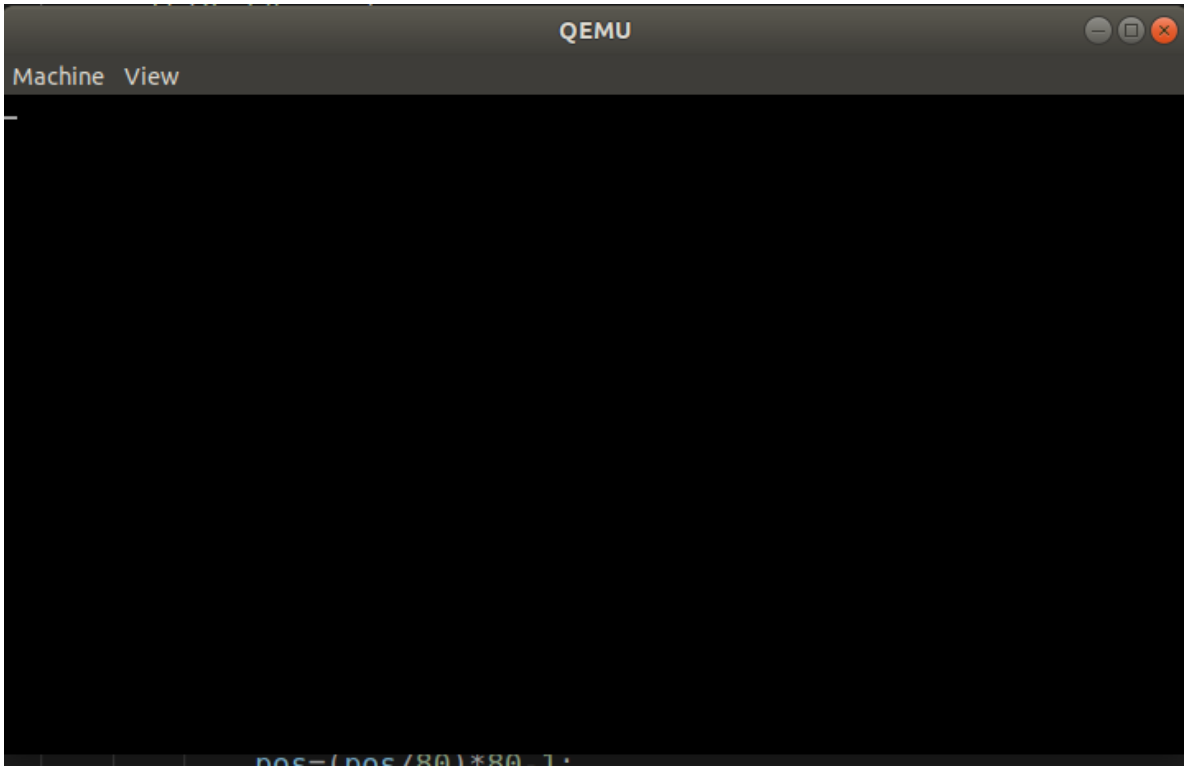
将以上六个文件：kbd.h,kbddisp.S,kdbhandler.c kbdvgainit.c,vgaputc.c,kbdgetc.c,放入一个新的文件夹，命名为kbd，将这个文件夹放入到device之下，并修改makefile文件，修改REBUILDFLAGS如下所示：

```
REBUILDFLAGS    =   -s $(TOPDIR)/system  debug.c        \
            -s $(TOPDIR)/lib                \
            -s $(TOPDIR)/device/tty         \
            -s $(TOPDIR)/device/nam         \
            -s $(TOPDIR)/device/ram         \
            -s $(TOPDIR)/device/lfs         \
            -s $(TOPDIR)/device/kbd         \
            -s $(TOPDIR)/shell
```

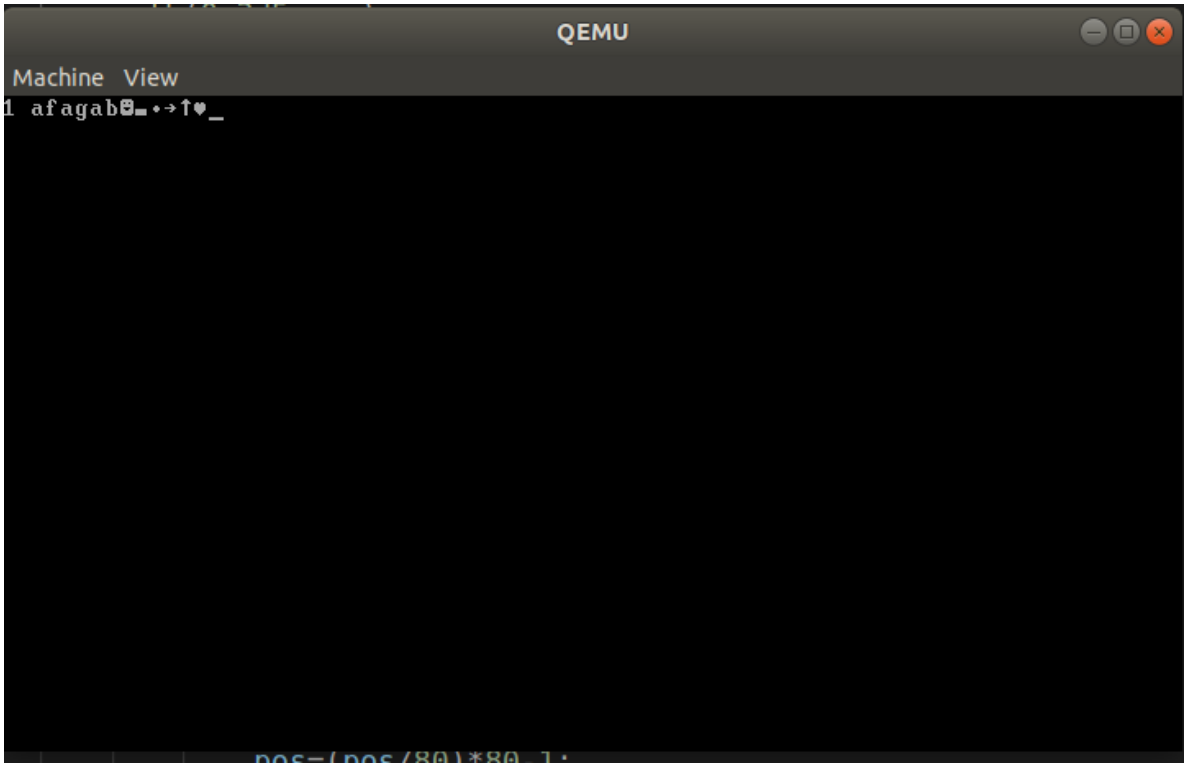
2 实验效果

2.1 初始化与输入

首先完成初始化之后页面如下所示，

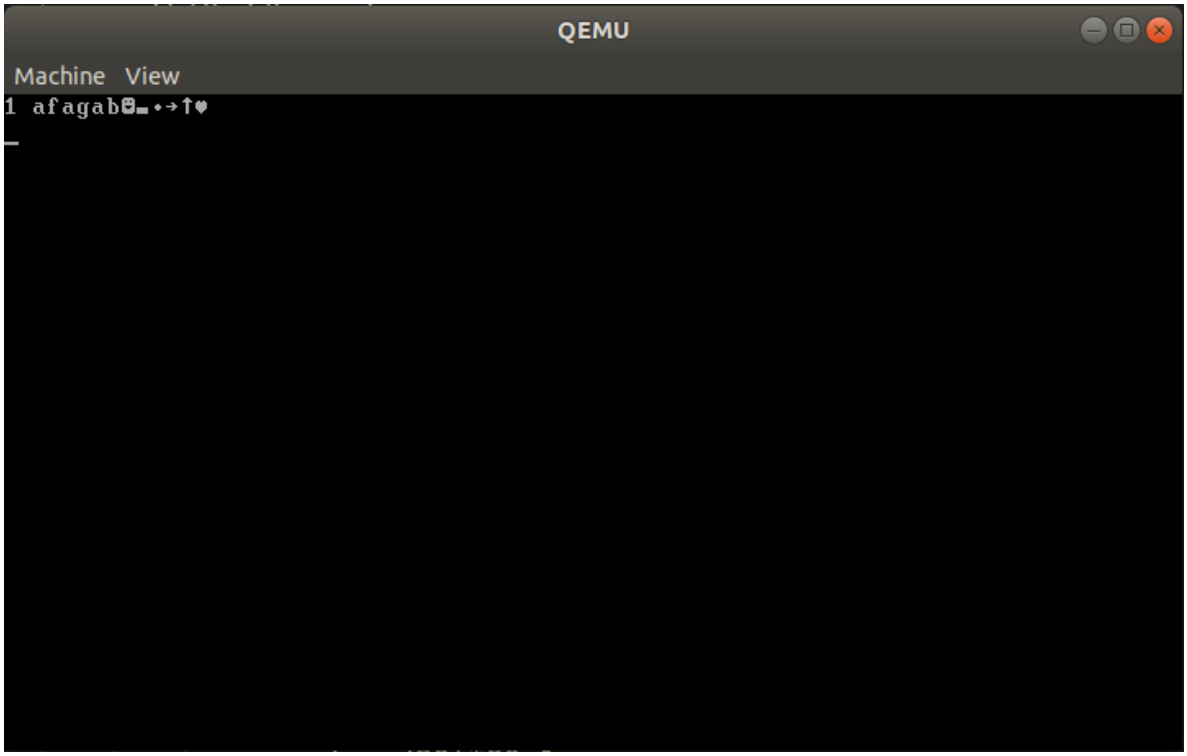


可以看到光标已经放置到了正确的位置并且屏幕已经清空。尝试输入一些字符，有些是正常字符，有些是nonprintable的字符，无论哪一种字符，可以看到都已经正常显示。

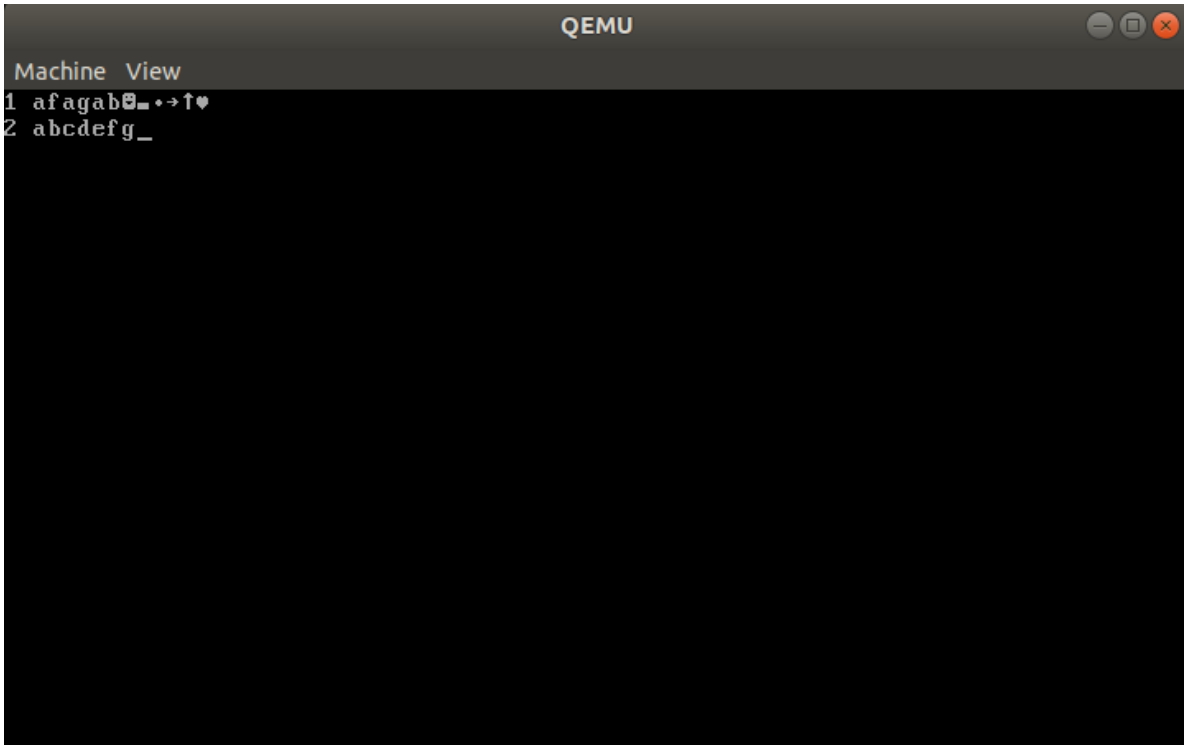


2.2 换行与退格

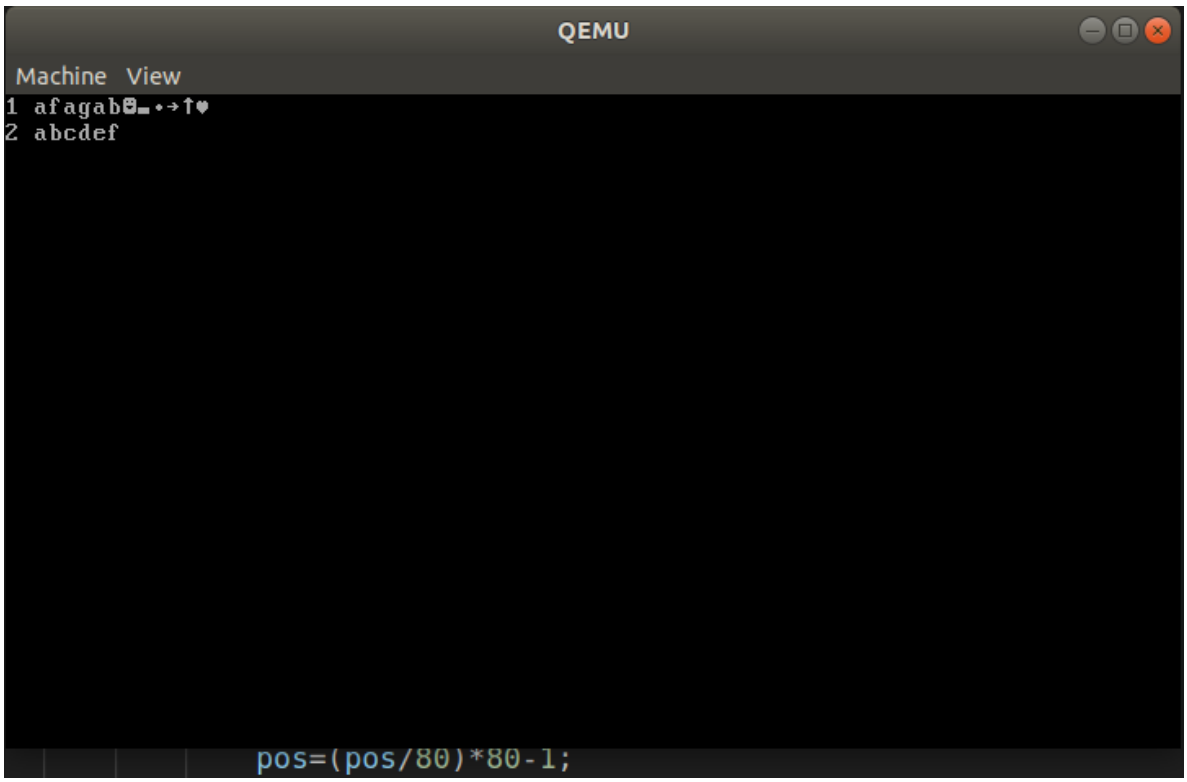
使用换行进行测试，可以看到光标到达了第二行的位置。



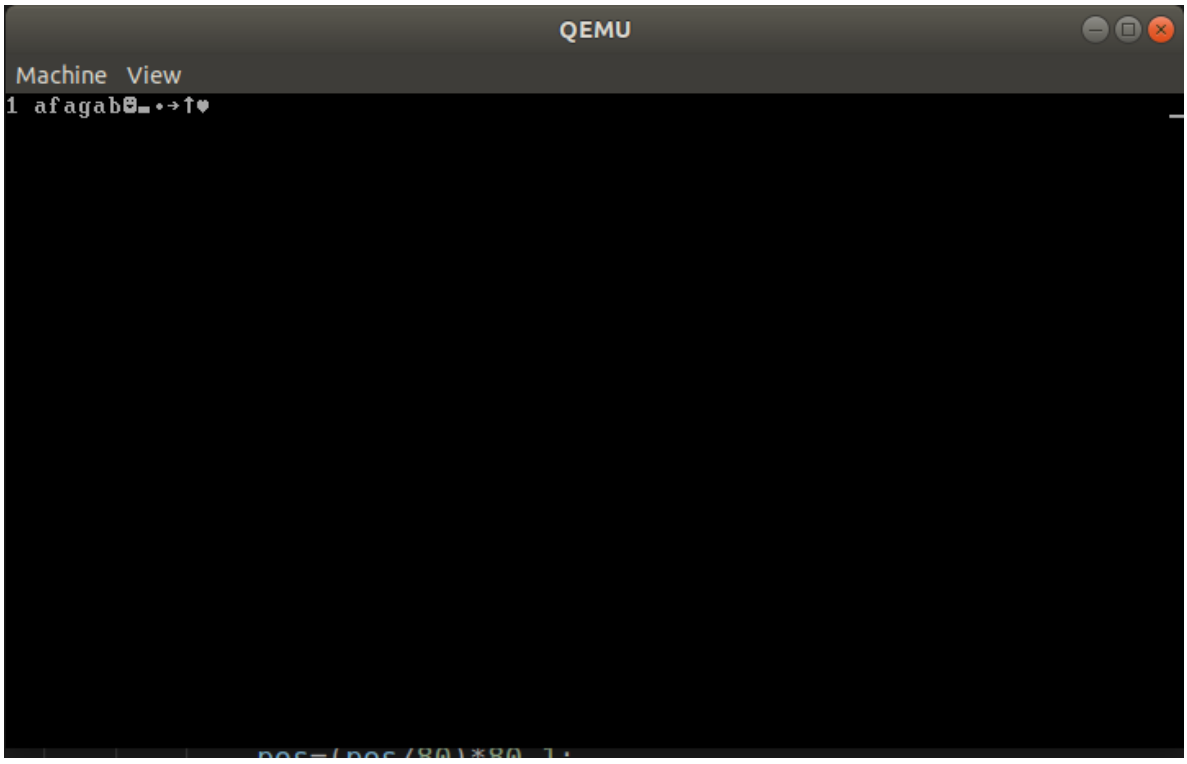
下面测试退格操作。在退格之前，显示屏的显示如下所示：



在执行了一个退格操作之后：



可以看到字符'g'被成功地去掉了，退格操作可以正常执行。

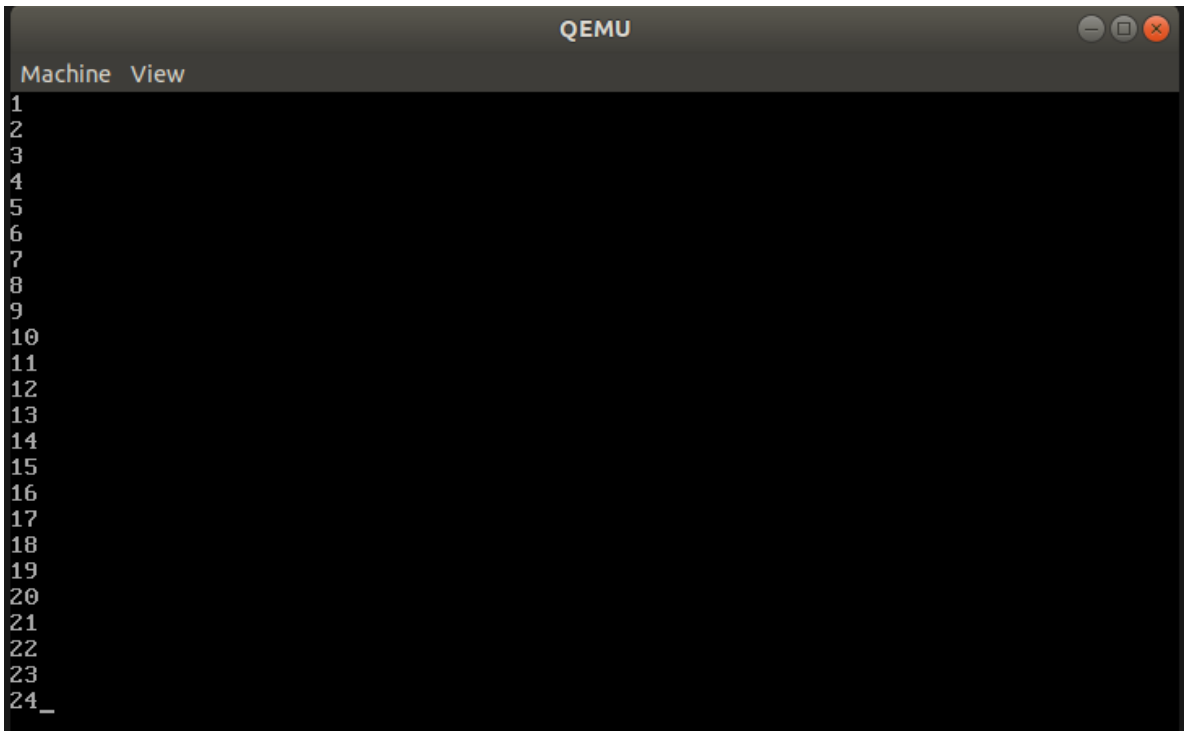


如上图所示，如果持续退格，可以从第二行的位置返回到第一行的位置，光标的位置在第一行的末尾。此时如果持续向前退格，则会继续清空前面所输入的那些字符

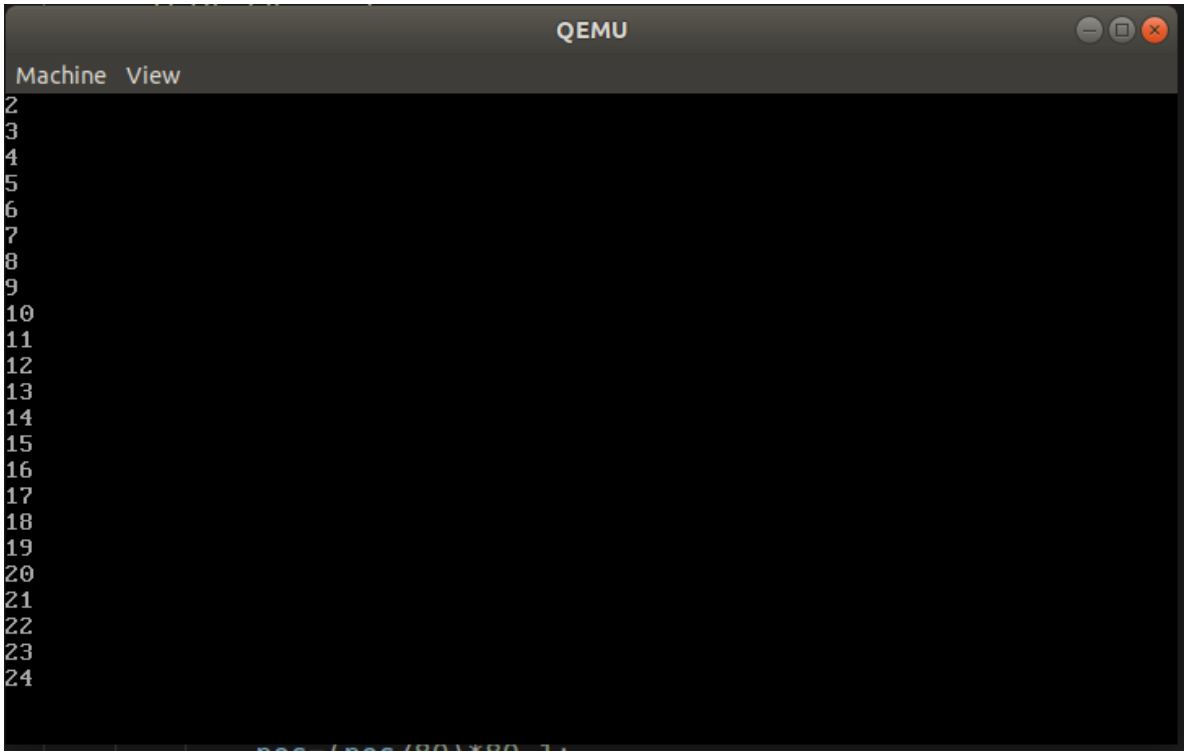
2.3 清空屏幕

接下来测试清屏功能。当已经在屏幕上输入了24行之后，如果此时按下'\r'或者'\n'来进行换行，那么就会导致第一行消失，其余各行顺次向上平移，实验效果如下图所示：

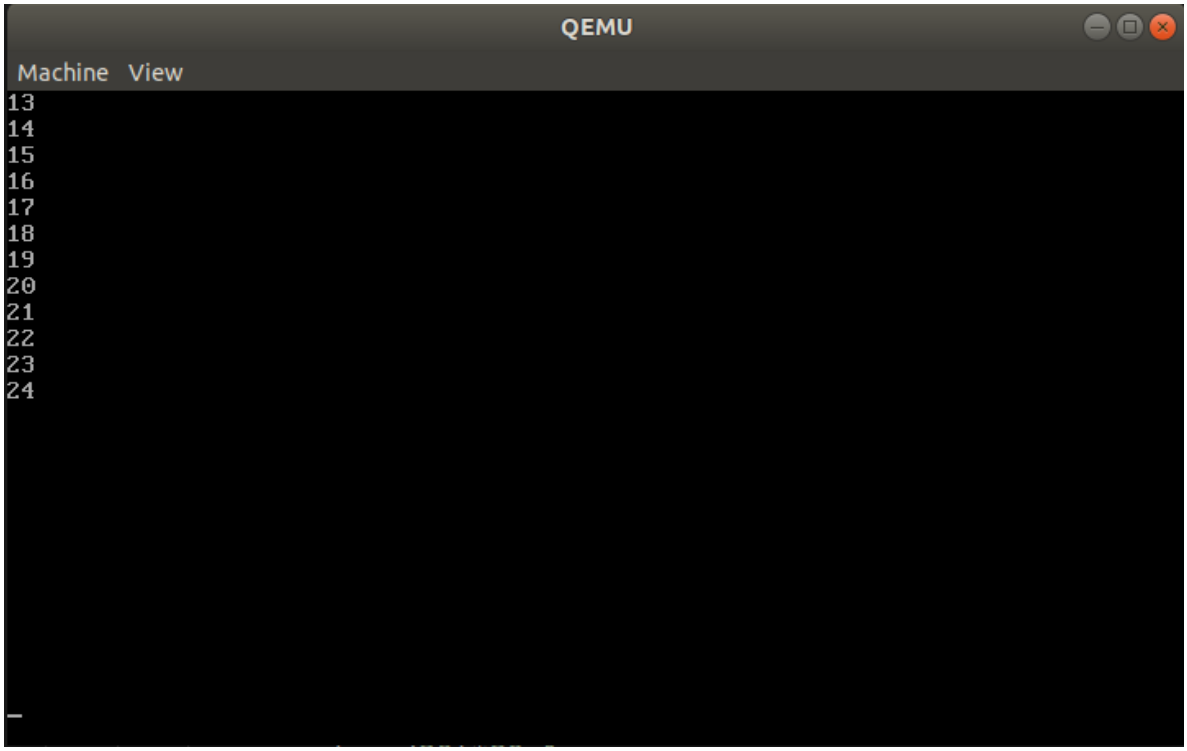
在换行之前，屏幕上已经有了24行：



接下来在这个位置按下换行键之后，效果如下图所示：



可以看到数字1已经消失了，代表第一行已经被清除；如果持续地再次键入回车换行，那么效果如下图所示：


QEMU
Machine View
13
14
15
16
17
18
19
20
21
22
23
24