

# 北京邮电大学

## 实验报告



题目： 键盘驱动程序的分析与修改

班 级： \_\_\_\_\_

学 号： \_\_\_\_\_

姓 名： \_\_\_\_\_

学 院： \_\_\_\_\_

2023 年 12 月 8 日

## 一、实验目的

- 1、理解 I/O 系统调用函数和 C 标准 I/O 函数的概念和区别；
- 2、建立内核空间 I/O 软件层次结构概念，即与设备无关的操作系统软件、设备驱动程序和中断服务程序；
- 3、了解 Linux-0.11 字符设备驱动程序及功能，初步理解控制台终端程序的工作原理；
- 4、通过阅读源代码，进一步提高 C 语言和汇编程序的编程技巧以及源代码分析能力；
- 5、锻炼和提高对复杂工程问题进行分析的能力，并根据需求进行设计和实现的能力。

## 二、实验环境

- 1、硬件：学生个人电脑 (x86-64)
- 2、软件：Windows 10, VMware Workstation 15 Player, 32 位 Linux-Ubuntu 16.04.1
- 3、gcc-3.4 编译环境
- 4、GDB 调试工具

## 三、实验内容

解压 lab4.tar.gz 文件，解压后进入 lab4 目录得到如下文件和目录：

```
lsy@ubuntu:~/Desktop/lab4$ ls
a.out          dbg-asm      gdb          hdc-0.11.img  mount-hdc
bochs          dbg-c       gdb-cmd.txt  linux-0.11    run
bochsout.txt  files       hdc         linux-0.11.tar.gz  run gdb
```

图 1：lab4 目录下文件

安装 gcc 编译器：

```
lsy@ubuntu:~/Desktop/lab4$ gcc -v
Using built-in specs.
COLLECT_GCC=gcc
COLLECT_LTO_WRAPPER=/usr/lib/gcc/i686-linux-gnu/5/lto-wrapper
Target: i686-linux-gnu
Configured with: ../src/configure -v --with-pkgversion='Ubuntu 5.4.0-6ubuntu1~16.04.12' --with-bugurl=file:///usr/share/doc/gcc-5/README.Bugs --enable-languages=c,ada,c++,java,go,d,fortran,objc,obj-c++ --prefix=/usr --program-suffix=-5 --enable-shared --enable-linker-build-id --libexecdir=/usr/lib --without-included-gettext --enable-threads=posix --libdir=/usr/lib --enable-nls --with-sysroot=/ --enable-clocale=gnu --enable-libstdcxx-debug --enable-libstdcxx-time=yes --with-default-libstdcxx-abi=new --enable-gnu-unique-object --disable-vtable-verify --enable-libmpx --enable-plugin --with-system-zlib --disable-browser-plugin --enable-java-awt=gtk --enable-gtk-cairo --with-java-home=/usr/lib/jvm/java-1.5.0-gcj-5-i386/jre --enable-java-home --with-jvm-root-dir=/usr/lib/jvm/java-1.5.0-gcj-5-i386 --with-jvm-jar-dir=/usr/lib/jvm-exports/java-1.5.0-gcj-5-i386 --with-arch-directory=i386 --with-ecj-jar=/usr/share/java/eclipse-ecj.jar --enable-objc-gc --enable-targets=all --enable-multilib --disable-werror --with-arch-32=i686 --with-multilib-list=m32,m64,mx32 --enable-multilib --with-tune=generic --enable-checking=release --build=i686-linux-gnu --host=i686-linux-gnu --target=i686-linux-gnu
Thread model: posix
gcc version 5.4.0 20160609 (Ubuntu 5.4.0-6ubuntu1~16.04.12)
```

图 2：gcc 版本

实验常用执行命令如下：

执行 `./run`，可启动 bochs 模拟器，进而加载执行 `Linux-0.11` 目录下的 `Image` 文件启动 `linux-0.11` 操作系统；

进入 `lab4/linux-0.11` 目录，执行 `make` 编译生成 `Image` 文件，每次重新编译（`make`）前需先执行 `make clean`；

如果对 `linux-0.11` 目录下的某些源文件进行了修改，执行 `./run init` 可把修改文件回复初始状态。

本实验包含 2 关，要求如下：

#### (1) Phase 1

键入 F12：激活\*功能。键入学生本人的姓名拼音，首尾字母等显示\*。比如：zhangsan，显示为：\*ha\*gsa\*；

再次键入 F12：取消该功能，正常显示。

#### (2) Phase 2

键入“学生本人的学号”：激活\*功能。键入学生本人的姓名拼音，首尾字母等显示\*。比如：zhangsan，显示为：\*ha\*gsa\*；

再次键入“学生本人的学号-”：取消显示\*功能。

提示：完成本实验需要对 `lab4/linux-0.11/kernel/chr_drv/` 目录下的 `keyboard.s`、`console.c` 和 `tty_io.c` 源文件进行分析，理解按下按键到回显到显示频上程序的执行过程，然后对涉及到的数据结构进行分析，完成对前两个源程序的修改。修改方案有两种：

在 C 语言源程序层面进行修改；

在汇编语言源程序层面进行修改。

其他说明见“实验四.ppt”。 “linux 内核完全注释(高清版).pdf”一书中对源代码有详细的说明和注释。

## 四、实验步骤及实验分析

### （一）准备工作

#### 1、安装 VMware Workstation 15 Player

网址如下：[VMware Workstation Player – VMware Customer Connect](#)，参考“实验四.pptx”的操作步骤进行勾选。

## 2、安装与添加 ubuntu

安装 32 位 Linux-Ubuntu 16.04.1 (版本: Ubuntu 16.04.1-desktop-i386.iso, 下载地址: [Index of /releases/xenial \(ubuntu.com\)](http://releases.xenial.ubuntu.com/));

在 VMware Workstation 15 Player 中按提示添加虚拟机。

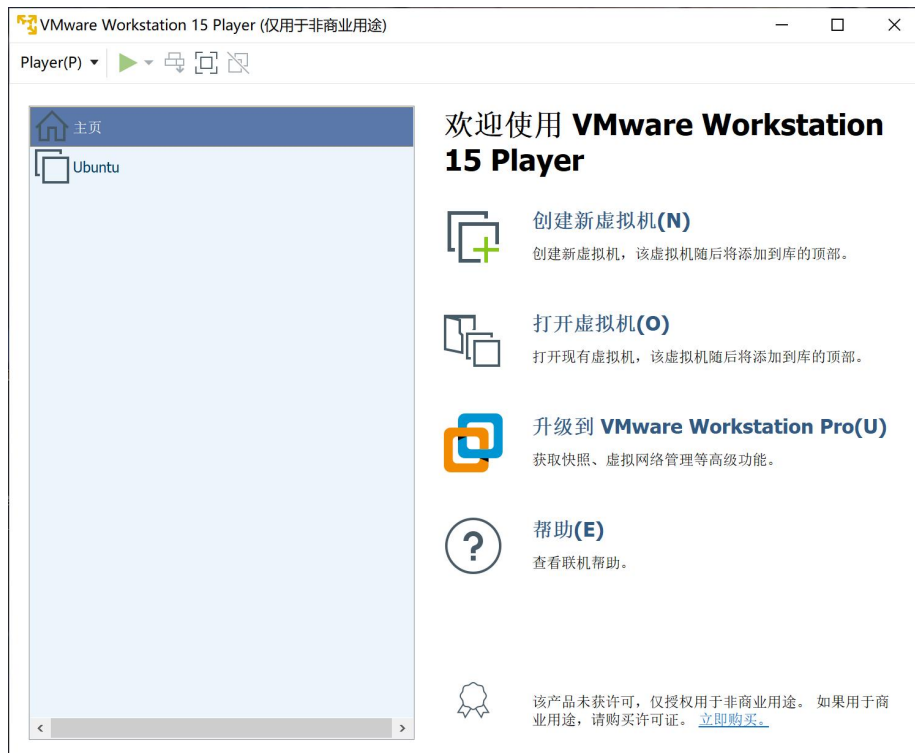


图 3: VMware Workstation 15 Player

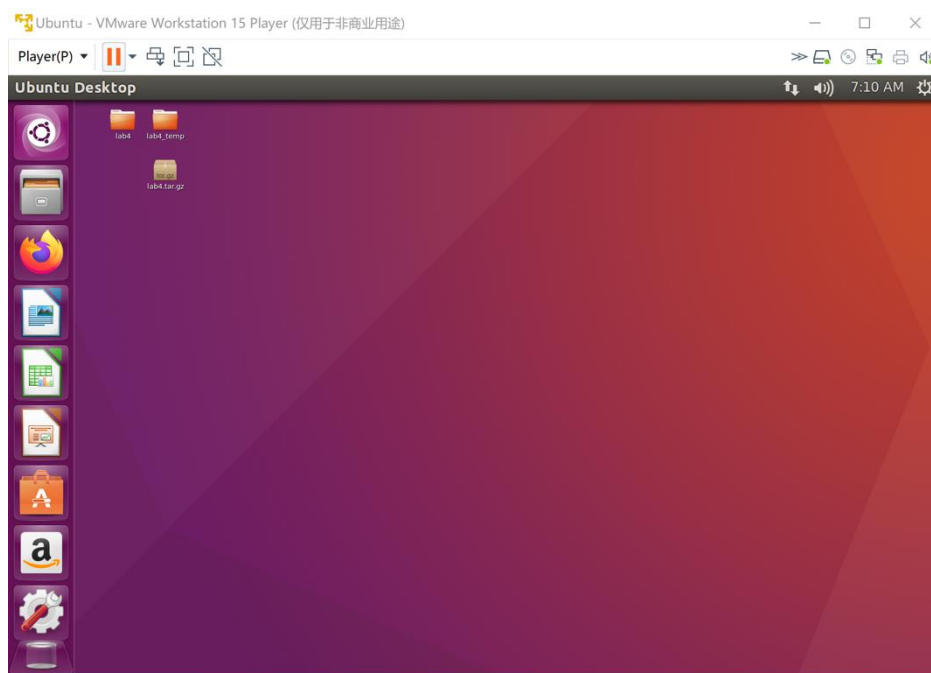


图 4: ubuntu 桌面

### 3、安装 gcc-3.4 和 as86

打开 terminal，依次输入命令安装 gcc-3.4:

```
wget http://old-releases.ubuntu.com/ubuntu/pool/universe/g/gcc-3.4/gcc-3.4-base_3.4.6-6ubuntu3_i386.deb
```

```
sudo dpkg -i gcc-3.4-base_3.4.6-6ubuntu3_i386.deb
```

```
wget http://old-releases.ubuntu.com/ubuntu/pool/universe/g/gcc-3.4/cpp-3.4_3.4.6-6ubuntu3_i386.deb
```

```
sudo dpkg -i cpp-3.4_3.4.6-6ubuntu3_i386.deb
```

```
wget http://old-releases.ubuntu.com/ubuntu/pool/universe/g/gcc-3.4/c-3.4_3.4.6-6ubuntu3_i386.deb
```

```
sudo dpkg -i gcc-3.4_3.4.6-6ubuntu3_i386.deb
```

```
wget http://old-releases.ubuntu.com/ubuntu/pool/universe/g/gcc-3.4/libstdc++6-dev_3.4.6-6ubuntu3_i386.deb
```

```
sudo dpkg --force-depends -i libstdc++6-dev_3.4.6-6ubuntu3_i386.deb
```

```
wget http://old-releases.ubuntu.com/ubuntu/pool/universe/g/gcc-3.4/g++-3.4_3.4.6-6ubuntu3_i386.deb
```

```
sudo dpkg -i g++-3.4_3.4.6-6ubuntu3_i386.deb
```

输入命令 `gcc -v` 查看版本，安装成功。

A terminal window showing the output of the 'gcc -v' command. The output displays the GCC configuration for Ubuntu 5.4.0-6ubuntu1~16.04.12, including the target architecture (i686-linux-gnu), various enabled and disabled features, and the final gcc version (5.4.0 20160609).

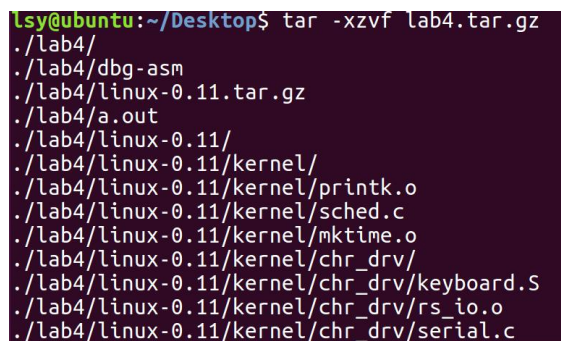
```
lsy@ubuntu:~/Desktop/lab4$ gcc -v
Using built-in specs.
COLLECT_GCC=gcc
COLLECT_LTO_WRAPPER=/usr/lib/gcc/i686-linux-gnu/5/lto-wrapper
Target: i686-linux-gnu
Configured with: ../src/configure -v --with-pkgversion='Ubuntu 5.4.0-6ubuntu1~16.04.12' --with-bugurl=file:///usr/share/doc/gcc-5/README.Bugs --enable-languages=c,ada,c++,java,go,d,fortran,objc,obj-c++ --prefix=/usr --program-suffix=-5 --enable-shared --enable-linker-build-id --libexecdir=/usr/lib --without-included-gettext --enable-threads=posix --libdir=/usr/lib --enable-nls --with-sysroot=/ --enable-clocale=gnu --enable-libstdcxx-debug --enable-libstdcxx-time=yes --with-default-libstdcxx-abi=new --enable-gnu-unique-object --disable-vtable-verify --enable-libmpx --enable-plugin --with-system-zlib --disable-browser-plugin --enable-java-awt=gtk --enable-gtk-cairo --with-java-home=/usr/lib/jvm/java-1.5.0-gcj-5-i386/jre --enable-java-home --with-jvm-root-dir=/usr/lib/jvm/java-1.5.0-gcj-5-i386 --with-jvm-jar-dir=/usr/lib/jvm-exports/java-1.5.0-gcj-5-i386 --with-arch-directory=i386 --with-ecj-jar=/usr/share/java/eclipse-ecj.jar --enable-objc-gc --enable-targets=all --enable-multilib --disable-werror --with-arch-32=i686 --with-multilib-list=m32,m64,mx32 --enable-multilib --with-tune=generic --enable-checking=release --build=i686-linux-gnu --host=i686-linux-gnu --target=i686-linux-gnu
Thread model: posix
gcc version 5.4.0 20160609 (Ubuntu 5.4.0-6ubuntu1~16.04.12)
```

图 5: gcc 版本

输入命令 `sudo apt install bin86`，安装 as86。

### 4、解压

将 lab4.tar.gz 拖到 ubuntu 的 Desktop 中，进入该目录，输入命令 `tar -xzf lab4.tar.gz` 解压。

A terminal window showing the output of the 'tar -xzf lab4.tar.gz' command. The output lists the files and directories extracted from the archive, including lab4/, lab4/dbg-asm, lab4/linux-0.11.tar.gz, lab4/a.out, lab4/linux-0.11/, lab4/linux-0.11/kernel/, lab4/linux-0.11/kernel/printk.o, lab4/linux-0.11/kernel/sched.c, lab4/linux-0.11/kernel/mktime.o, lab4/linux-0.11/kernel/chr\_drv/, lab4/linux-0.11/kernel/chr\_drv/keyboard.S, lab4/linux-0.11/kernel/chr\_drv/rs\_io.o, and lab4/linux-0.11/kernel/chr\_drv/serial.c.

```
lsy@ubuntu:~/Desktop$ tar -xzf lab4.tar.gz
./lab4/
./lab4/dbg-asm
./lab4/linux-0.11.tar.gz
./lab4/a.out
./lab4/linux-0.11/
./lab4/linux-0.11/kernel/
./lab4/linux-0.11/kernel/printk.o
./lab4/linux-0.11/kernel/sched.c
./lab4/linux-0.11/kernel/mktime.o
./lab4/linux-0.11/kernel/chr_drv/
./lab4/linux-0.11/kernel/chr_drv/keyboard.S
./lab4/linux-0.11/kernel/chr_drv/rs_io.o
./lab4/linux-0.11/kernel/chr_drv/serial.c
```

图 4: 解压 lab4.tar.gz



```
lsy@ubuntu:~/Desktop/lab4$ ls
a.out          dbg-asm  gdb      hdc-0.11.img  mount-hdc
bochs          dbg-c   gdb-cmd.txt linux-0.11    run
bochsout.txt  files   hdc      linux-0.11.tar.gz  rungdb
```

图 5: 解压后 lab4 目录下的文件

注意到该目录下有可执行文件 run。输入命令 ./run，可启动 bochs 模拟器，启动 linux-0.11 操作系统；

注意到 lab4 中已存在 linux-0.11 文件夹，但“实验四.pptx”提示我们解压 linux-0.11.tar.gz 至 linux-0.11 文件夹内。

输入命令 mv linux-0.11 linux-0.11.1，先改原文件夹的名称为 linux-0.11.1 防止重名；

输入命令 mkdir linux-0.11 新建一个 linux-0.11 文件夹；

输入命令 tar -xzf linux-0.11.tar.gz -C linux-0.11，解压 linux-0.11.tar.gz 至 linux-0.11 文件夹内。

```
lsy@ubuntu:~/Desktop/lab4$ mv linux-0.11 linux-0.11.1
lsy@ubuntu:~/Desktop/lab4$ mkdir linux-0.11
lsy@ubuntu:~/Desktop/lab4$ tar -xzf linux-0.11.tar.gz -C linux-0.11
boot/
boot/head.s
boot/setup.s
boot/bootsect.s
fs/
fs/truncate.c
fs/bitmap.c
fs/inode.c
fs/stat.c
fs/file_dev.c
fs/open.c
fs/Makefile
fs/ioctl.c
fs/read_write.c
fs/fcntl.c
fs/file_table.c
```

图 6: 解压 linux-0.11.tar.gz

```
lsy@ubuntu:~/Desktop/lab4$ cd linux-0.11
lsy@ubuntu:~/Desktop/lab4/linux-0.11$ ls
boot fs include init kernel lib Makefile mm tags tools
```

图 7: 解压后 linux-0.11 目录下的文件

## (二) 阶段 1

### 1、阅读实验说明，理解原理

本阶段需要实现的功能是：键入 F12，激活\*功能；再次键入 F12，取消\*功能。其中\*功能表现为：键入学生姓名，首尾字母改为\*。比如我的名字 liushiyuan，显示为\*iushiyua\*，且

linux-0.11 操作系统下所有的字符 ‘l’ ‘n’ 都被改为 ‘\*’ 。

实验提示中提到，完成本实验需要对 lab4/linux-0.11/kernel/chr\_drv/目录下的 keyboard.s、console.c 和 tty\_io.c 源文件进行分析，理解按下按键到回显到显示频上程序的执行过程，然后对涉及到的数据结构进行分析，完成对前两个源程序的修改，可以在 C 语言源程序层面、汇编语言源程序层面进行修改。

查看 lab4/linux-0.11/kernel/chr\_drv 目录下的文件：

```
lsy@ubuntu:~/Desktop/lab4/linux-0.11$ cd ./kernel/chr_drv
lsy@ubuntu:~/Desktop/lab4/linux-0.11/kernel/chr_drv$ ls
chr_drv.a  keyboard.o  Makefile  serial.c  tty_ioctl.c
console.c  keyboard.s  rs_io.o   serial.o  tty_ioctl.o
console.o  keyboard.S  rs_io.s   tty_io.c  tty_io.o
```

图 8: lab4/linux-0.11/kernel/chr\_drv 目录下的文件

阅读“linux 内核完全注释(高清版).pdf”第十章中 keyboard.s、console.c 和 tty\_io.c 的相关内容。可知该目录下的程序可分成三部分：

- (1) RS-232 串行线路驱动程序：rs\_io.s、serial.c；
- (2) 控制台驱动程序：键盘中断驱动程序 keyboard.S、控制台显示驱动程序 console.c；
- (3) 终端驱动程序与上层接口：终端输入输出程序 tty\_io.c、终端控制程序 tty\_ioctl.c。

先看控制台终端：它接收 tty\_io.c 程序传递下来的显示字符或控制信息，经由 console.c 程序控制屏幕上的字符显示；同时，控制台将键盘按键产生的代码经由 keyboard.s 传送到 tty\_io.c 程序去处理。

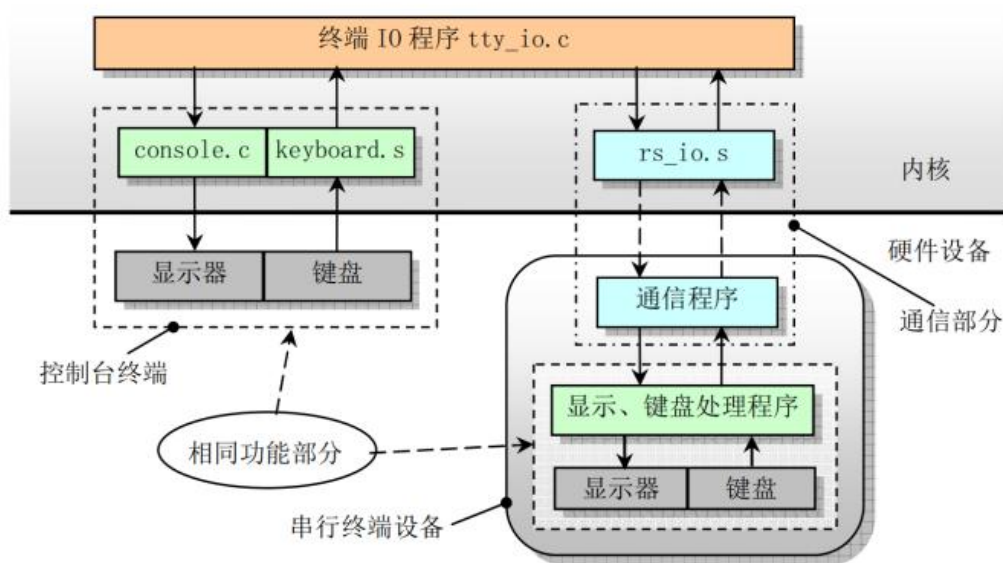


图 9: 控制台终端与串行终端设备示意图

再看控制台驱动程序：涉及 keyboard.S 和 console.c 程序。对于用户键入字符-->屏幕显示字符，以下是完整的控制台键盘中断处理进程：

相应程序	完成操作
keyboard.S	读入该字符对应的键盘扫描码，根据映射表译成相应字符
	调用 put_queue 函数，将字符放入 tty 读缓冲队列 read_q 中
tty_io.c	调用 do_tty_interrupt 函数对字符进行过滤处理，放入 tty 辅助队列 secondary 中；函数内部还调用了 copy_to_cooked 函数，将字符放入 tty 写队列 write_q 中
console.c	调用函数 con_write，将字符显示到屏幕上

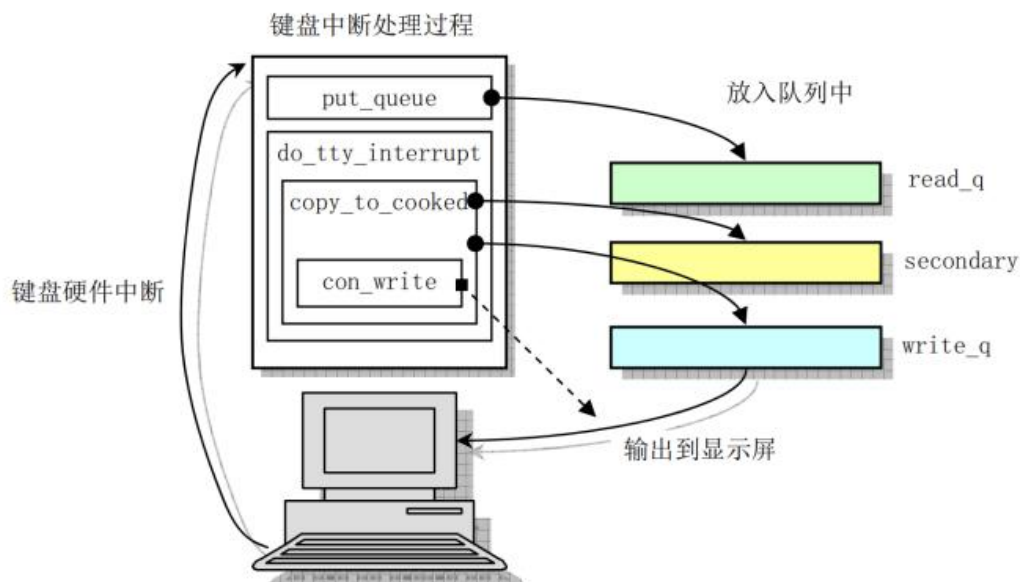


图 10：控制台键盘中断处理进程

因此，我们需要完成的是：

- (1) 使屏幕上的显示字符变为 '\*'：修改 console.c 程序的 con\_write 函数。读到 'l' 'n' 字符时，将其改为 '\*'；
- (2) 按下 F12 关闭或打开\*功能：可以设置一个 f12Flag，当按下 F12 时，执行在 console.c 上修改的程序。

## 2、修改程序 keyboard.S

输入命令 vi keyboard.S，搜索 f12，发现跳转表 key\_table。



```

key_table:
.long none,do_self,do_self,do_self /* 00-03 s0 esc 1 2 */
.long do_self,do_self,do_self,do_self /* 04-07 3 4 5 6 */
.long do_self,do_self,do_self,do_self /* 08-0B 7 8 9 0 */
.long do_self,do_self,do_self,do_self /* 0C-0F + ' bs tab */
.long do_self,do_self,do_self,do_self /* 10-13 q w e r */
.long do_self,do_self,do_self,do_self /* 14-17 t y u i */
.long do_self,do_self,do_self,do_self /* 18-1B o p } ^ */
.long do_self,ctrl,do_self,do_self /* 1C-1F enter ctrl a s */
.long do_self,do_self,do_self,do_self /* 20-23 d f g h */
.long do_self,do_self,do_self,do_self /* 24-27 j k l | */
.long do_self,do_self,lshift,do_self /* 28-2B { para lshift , */
.long do_self,do_self,do_self,do_self /* 2C-2F z x c v */
.long do_self,do_self,do_self,do_self /* 30-33 b n m , */
.long do_self,minus,rshift,do_self /* 34-37 . - rshift * */
.long alt,do_self,caps,func /* 38-3B alt sp caps f1 */
.long func,func,func,func /* 3C-3F f2 f3 f4 f5 */
.long func,func,func,func /* 40-43 f6 f7 f8 f9 */
.long func,num,scroll,cursor /* 44-47 f10 num scr home */
.long cursor,cursor,do_self,cursor /* 48-4B up pgup - left */
.long cursor,cursor,do_self,cursor /* 4C-4F n5 right + end */
.long cursor,cursor,cursor,cursor /* 50-53 dn pgdn ins del */
.long none,none,do_self,func /* 54-57 sysreq ? < f11 */
.long func,none,none,none /* 58-5B f12 ? ? ? */
.long none,none,none,none /* 5C-5F ? ? ? ? */
.long none,none,none,none /* 60-63 ? ? ? ? */
.long none,none,none,none /* 64-67 ? ? ? ? */
.long none,none,none,none /* 68-6B ? ? ? ? */
.long none,none,none,none /* 6C-6F ? ? ? ? */
.long none,none,none,none /* 70-73 ? ? ? ? */
.long none,none,none,none /* 74-77 ? ? ? ? */
.long none,none,none,none /* 78-7B ? ? ? ? */
.long none,none,none,none /* 7C-7F ? ? ? ? */
.long none,none,none,none /* 80-83 ? br br br */

.long none,none,none,none /* 84-87 br br br br */
.long none,none,none,none /* 88-8B br br br br */
.long none,none,none,none /* 8C-8F br br br br */
.long none,none,none,none /* 90-93 br br br br */
.long none,none,none,none /* 94-97 br br br br */
.long none,none,none,none /* 98-9B br br br br */
.long none,unctrl,none,none /* 9C-9F br unctrl br br */
.long none,none,none,none /* A0-A3 br br br br */
.long none,none,none,none /* A4-A7 br br br br */
.long none,none,unlshift,none /* A8-AB br br unlshift br */
.long none,none,none,none /* AC-AF br br br br */
.long none,none,none,none /* B0-B3 br br br br */
.long none,none,unrshift,none /* B4-B7 br br unrshift br */
.long unalt,none,uncaps,none /* B8-BB unalt br uncaps br */
.long none,none,none,none /* BC-BF br br br br */
.long none,none,none,none /* C0-C3 br br br br */
.long none,none,none,none /* C4-C7 br br br br */
.long none,none,none,none /* C8-CB br br br br */
.long none,none,none,none /* CC-CF br br br br */
.long none,none,none,none /* D0-D3 br br br br */
.long none,none,none,none /* D4-D7 br br br br */
.long none,none,none,none /* D8-DB br ? ? ? */
.long none,none,none,none /* DC-DF ? ? ? ? */
.long none,none,none,none /* E0-E3 e0 e1 ? ? */
.long none,none,none,none /* E4-E7 ? ? ? ? */
.long none,none,none,none /* E8-EB ? ? ? ? */
.long none,none,none,none /* EC-EF ? ? ? ? */
.long none,none,none,none /* F0-F3 ? ? ? ? */
.long none,none,none,none /* F4-F7 ? ? ? ? */
.long none,none,none,none /* F8-FB ? ? ? ? */
.long none,none,none,none /* FC-FF ? ? ? ? */

```

图 11~12: 扫描码跳转表 key\_table

查询“linux 内核完全注释(高清版).pdf”，解释如下：

```

*/
/* 下面是一张子程序地址跳转表。当取得扫描码后就根据此表调用相应的扫描码
* 处理子程序。大多数调用的子程序是 do_self, 或者是 none, 这取决于按键
* (make) 还是释放键(break)。
*/

```

图 13: 扫描码跳转表 key\_table 的解释

从图 11 找到 f12 对应的处理函数为 func。在 keyboard.S 中查找函数 func:

```

/*
 * this routine handles function keys
 */
func:
    pushl %eax
    pushl %ecx
    pushl %edx
    call show_stat
    popl %edx
    popl %ecx
    popl %eax
    subb $0x3B,%al
    jb end_func
    cmpb $9,%al
    jbe ok_func
    subb $18,%al
    cmpb $10,%al
    jb end_func
    cmpb $11,%al
    ja end_func

```

图 14: 函数 func

```

* 下面子程序处理功能键。
*/
// 把功能键扫描码转换成转义字符序列并存放读到读队列中。
210 func:
211     pushl %eax
212     pushl %ecx
213     pushl %edx
214     call _show_stat           // 调用显示各任务状态函数 (kernel/sched.c, 37 行)。
215     popl %edx
216     popl %ecx
217     popl %eax

218     subb $0x3B,%al           // 键'F1'的扫描码是 0x3B, 因此 al 中是功能键索引号。
219     jb end_func              // 如果扫描码小于 0x3b, 则不处理, 返回。
220     cmpb $9,%al              // 功能键是 F1-F10?
221     jbe ok_func              // 是, 则跳转。
222     subb $18,%al              // 是功能键 F11, F12 吗? F11、F12 扫描码是 0x57、0x58。
223     cmpb $10,%al             // 是功能键 F11?
224     jb end_func              // 不是, 则不处理, 返回。
225     cmpb $11,%al             // 是功能键 F12?
226     ja end_func              // 不是, 则不处理, 返回。
227 ok_func:
228     cmpl $4,%ecx              // /* check that there is enough room */ /*检查空间*/
229     jl end_func               // [??]需要放入 4 个字符序列, 如果放不下, 则返回。
230     movl func_table(,%eax,4),%eax // 取功能键对应字符序列。
231     xorl %ebx,%ebx            // 因要放入队列字符数=4, 因此执行 put_queue 之前
232     jmp put_queue             // 需把 ebx 清零。
233 end_func:
234     ret

```

图 15~16: 函数 func 的解释

当键入 f12, 就跳转到函数 func, 且在下面的 ok\_func (227 行) 中就跳转到 put\_queue 将字符放入 tty 读缓冲队列 read\_q 中了, 因此可以在这之前改变 f12Flag, 表示按下 f12 可以激活\* 功能。可以在 console.c 中写一个函数 changeF12Flag, 编辑函数 func 调用之。

```
func:
    pushl %eax
    pushl %ecx
    pushl %edx
    call show_stat
    popl %edx
    popl %ecx
    popl %eax
    subb $0x3B,%al
    jnb end_func
    cmpb $9,%al
    jbe ok_func
    subb $18,%al
    cmpb $10,%al
    jnb end_func
    cmpb $11,%al
    ja end_func
    call changeF12Flag
```

图 17: 在 func 中调用 changeF12Flag

### 3、修改程序 console.c

查看“linux 内核完全注释(高清版).pdf”中 console.c 的说明, 得知其主要函数 con\_write 会从终端 tty 的写缓冲队列 write\_q 中取出字符或字符序列, 根据性质 (普通字符、控制字符、转义序列、控制序列) 将字符显示在终端屏幕上或进行屏幕控制操作。

输入命令 vi console.c, 查看其中的函数 con\_write。

```
void con_write(struct tty_struct * tty)
{
    int nr;
    char c;

    nr = CHARS(tty->write_q);
    while (nr-- > 0) {
        GETCH(tty->write_q,c);
        switch(state) {
            case 0:
                if (c>31 && c<127) {
                    if (x>=video_num_columns) {
                        x -= video_num_columns;
                        pos -= video_size_row;
                        lf();
                    }
                    __asm__ ("movb attr,%%ah\n\t"
                             "movw %%ax,%1\n\t"
                             :: "a" (c), "m" (*(short *)pos)
                             );
                    pos += 2;
                    x++;
                } else if (c==27)
                    state=1;
                else if (c==10 || c==11 || c==12)
                    lf();
                else if (c==13)
                    cr();
                else if (c==ERASE_CHAR(tty))
                    del();
                else if (c==8) {
                    if (x) {
                        x--;
                    }
                }
            }
    }
}
```

图 18: 函数 con\_write 的前半段



查看 console.c 的注释，由于我们要修改的是 ‘l’ ‘n’，因此 case0 之后的这一段（对应普通显示字符）是合适的插入 f12Flag 代码的地方。

```
// 如果从写队列中取出的字符是普通显示字符代码，就直接从当前映射字符集中取出对应的显示
// 字符，并放到当前光标所处的显示内存位置处，即直接显示该字符。然后把光标位置右移一个
// 字符位置。具体地，如果字符不是控制字符也不是扩展字符，即(31<c<127)，那么，若当前光
// 标处在行末端或末端以外，则将光标移到下行头列。并调整光标位置对应的内存指针 pos。然
// 后将字符 c 写到显示内存中 pos 处，并将光标右移 1 列，同时也将 pos 对应地移动 2 个字节。
    if (c>31 && c<127) { // 是普通显示字符。
        if (x>=video_num_columns) { // 要换行？
            x -= video_num_columns;
            pos -= video_size_row;
            lf();
        }
        __asm__( "movb _attr, %%ah\n|t" // 写字符。
                "movw %%ax, %l\n|t"
                :: "a" (c), "m" (*(short *)pos)
                : "ax");
        pos += 2;
        x++;
    }
// 如果字符 c 是转义字符 ESC，则转换状态 state 到 1。
```

图 19：函数 con\_write 的其中一段解释

在 console.c 中编写函数 changeF12Flag:

```
void changeF12Flag(void)
{
    f12Flag=!f12Flag;
    return;
}
```

图 20：函数 changeF12Flag

在文件前部进行全局变量 f12Flag 定义和函数声明:

```
static f12Flag = 0;
void changeF12Flag(void);
```

图 21：定义全局变量 f12Flag 和声明函数 changeF12Flag

以上工作完成了在按下 f12 按键时，f12Flag 变量的状态更新。接下来在 con\_write 函数中添加条件，使得 f12Flag 条件满足且输入字符 ‘l’ ‘n’ 时改为 ‘\*’。代码编写如下：

```
if (f12Flag==1 && (c == 'l' || c == 'n'))
    c = '*' ;
```

编写完成后，在函数 con\_write 中添加代码。

```

void con_write(struct tty_struct * tty)
{
    int nr;
    char c;

    nr = CHARS(tty->write_q);
    while (nr--) {
        GETCH(tty->write_q,c);
        switch(state) {
            case 0:
                if (c>31 && c<127) {
                    if (f12Flag==1 && (c=='l' || c=='n'))
                        c='*';
                    if (x>=video_num_columns) {
                        x -= video_num_columns;
                        pos -= video_size_row;
                        lf();
                    }
                }
            }
        }
    }
}

```

图 22: 修改后的函数 con\_write

#### 4、运行 linux-0.11 操作系统，测试功能

输入命令 `cd ../../`，回到 linux-0.11 目录下；

输入命令 `make clean`，清除上次编译结果；

输入命令 `make all`，重新编译；

输入命令 `./run`，执行该目录下的可执行文件 `run`，进入 linux-0.11 操作系统。

```

lsy@ubuntu:~/Desktop/lab4/linux-0.11/kernel/chr_drv$ cd ..
lsy@ubuntu:~/Desktop/lab4/linux-0.11/kernel$ cd ..
lsy@ubuntu:~/Desktop/lab4/linux-0.11$ make clean
rm -f Image System.map tmp_make core boot/bootsect boot/setup
rm -f init/*.o tools/system tools/build boot/*.o
(cd mm;make clean)
make[1]: Entering directory '/home/lsy/Desktop/lab4/linux-0.11/mm'
rm -f core *.o *.a tmp_make
for i in *.c;do rm -f `basename $i .c`.s;done
make[1]: Leaving directory '/home/lsy/Desktop/lab4/linux-0.11/mm'
(cd fs;make clean)
make[1]: Entering directory '/home/lsy/Desktop/lab4/linux-0.11/fs'
rm -f core *.o *.a tmp_make
for i in *.c;do rm -f `basename $i .c`.s;done
make[1]: Leaving directory '/home/lsy/Desktop/lab4/linux-0.11/fs'

lsy@ubuntu:~/Desktop/lab4/linux-0.11$ make all
as86 -O -a -o boot/bootsect.o boot/bootsect.s
ld86 -O -s -o boot/bootsect boot/bootsect.o
as86 -O -a -o boot/setup.o boot/setup.s
ld86 -O -s -o boot/setup boot/setup.o
gcc-3.4 -m32 -g -I./include -traditional -c boot/head.s
mv head.o boot/
gcc-3.4 -march=i386 -m32 -g -Wall -O2 -fomit-frame-pointer \
-nostdinc -Iinclude -c -o init/main.o init/main.c
init/main.c:23: warning: static declaration of 'fork' follows non-static declarati
include/unistd.h:210: warning: previous declaration of 'fork' was here
init/main.c:24: warning: static declaration of 'pause' follows non-static declarati
include/unistd.h:224: warning: previous declaration of 'pause' was here
init/main.c:26: warning: static declaration of 'sync' follows non-static declarati
include/unistd.h:235: warning: previous declaration of 'sync' was here
init/main.c:105: warning: return type of 'main' is not 'int'
(cd kernel; make)
make[1]: Entering directory '/home/lsy/Desktop/lab4/linux-0.11/kernel'
gcc-3.4 -march=i386 -m32 -g -Wall -O -fstrength-reduce -fomit-frame-pointer -finli
ostdinc -I./include \
-c -o sched.o sched.c
as --32 -o system_call.o system_call.s
system_call.s: Assembler messages:

```



```
lsy@ubuntu:~/Desktop/lab4$ ./run
=====
Bochs x86 Emulator 2.3.7
Build from CVS snapshot, on June 3, 2008
=====
00000000000i[      ] reading configuration from ./bochs/bochsrc.bxrc
00000000000i[      ] installing x module as the Bochs GUI
00000000000i[      ] using log file ./bochsout.txt
```

图 23~25: make clean、make all、run

可以看到，按下 f12，所有的 ‘l’ ‘n’ 都成功变成 ‘\*’，\*功能开启；重新按下 f12，\*功能关闭。该阶段通过。

```
Options: apmbios pcibios eltorito rombios32
ata0 master: Generic 1234 ATA-6 Hard-Disk ( 60 MBytes)
Booting from Floppy...
Loading system ...
Partition table ok.
39044/62000 free blocks
19520/20666 free inodes
3454 buffers = 3536896 bytes buffer space
Free mem: 12582912 bytes
Ok.
[/usr/root]# 0: pid=0, state=1, 2740 (of 3140) chars free in kernel stack
1: pid=1, state=1, 2568 (of 3140) chars free in kernel stack
2: pid=4, state=1, 1448 (of 3140) chars free in kernel stack
3: pid=3, state=1, 1448 (of 3140) chars free in kernel stack
*iu shiyua*
*iu shiyua*: comma*d *ot fou*d
[/usr/root]# 0: pid=0, state=1, 2588 (of 3140) chars free i* ker*e* stack
1: pid=1, state=1, 2568 (of 3140) chars free i* ker*e* stack
2: pid=4, state=1, 1448 (of 3140) chars free i* ker*e* stack
3: pid=3, state=1, 1448 (of 3140) chars free i* ker*e* stack
liushiyuan_
```

图 26: 阶段 1 成功

### (三) 阶段 2

#### 1、阅读实验说明，理解原理

本阶段需要实现的功能是：键入“学生本人的学号”（“2022211073”）：激活\*功能。  
再次键入“学生本人的学号-”：取消显示\*功能。

\*功能同样可以由阶段 1 console.c 中插入的代码实现，即：

```
if (f12Flag==1 && (c == 'l' || c == 'n'))
    c = '*' ;
```

但是 f12Flag==1 的条件不再是键入 f12，需要修改。

阶段 1 编写了函数 changeF12Flag，在键入 f12 后执行的函数 func 中调用该函数，使 f12Flag=1。  
不同的是，本阶段键入的字符变为 ‘2’ ‘0’ ‘1’ ‘7’ ‘3’ ‘-’，且需要满足连续输入特

定的字符串才会修改 f12Flag。可以编写一个函数 checkChLine 来判断，思路与阶段 1 类似。

## 2、修改程序 keyboard.S

查找跳转表 key\_table 中 ‘2’ ‘0’ ‘1’ ‘7’ ‘3’ ‘-’ 这几个字符对应的处理函数，由图 11 可知，都为函数 do\_self。我们需要在函数 do\_self 中插入函数 checkChLine。查看函数 do\_self 的汇编代码：

```
do_self:
    lea alt_map,%ebx
    testb $0x20,mode                /* alt-gr */
    jne 1f
    lea shift_map,%ebx
    testb $0x03,mode
    jne 1f
    lea key_map,%ebx
1:   movb (%ebx,%eax),%al
    orb %al,%al
    je none
    testb $0x4c,mode                /* ctrl or caps */
    je 2f
    cmpb $'a',%al
    jb 2f
    cmpb $'}',%al
    ja 2f
    subb $32,%al
2:   testb $0x0c,mode                /* ctrl */
    je 3f
    cmpb $64,%al
    jb 3f
    cmpb $64+32,%al
    jae 3f
    subb $64,%al
3:   testb $0x10,mode                /* left alt */
    je 4f
    orb $0x80,%al
4:   andl $0xff,%eax
    xorl %ebx,%ebx
    call put_queue
none: ret
```

图 27：函数 do\_self 的汇编代码

\* do\_self 用于处理“普通”键，也即含义没有变化并且只有一个字符返回的键。  
\*/

```
453 do_self:
    // 首先根据 mode 标志选择 alt_map、shift_map 或 key_map 映射表之一。
454     lea alt_map,%ebx                // 取 alt 键同时按下时的映射表基址 alt_map。
455     testb $0x20,mode                /* alt-gr */ /* 右 alt 键同时按下了? */
456     jne 1f                          // 是，则向前跳转到标号 1 处。
457     lea shift_map,%ebx              // 取 shift 键同时按下时的映射表基址 shift_map。
458     testb $0x03,mode                // 有 shift 键同时按下了吗？
459     jne 1f                          // 有，则向前跳转到标号 1 处。
460     lea key_map,%ebx                // 否则使用普通映射表 key_map。
    // 然后根据扫描码取映射表中对应的 ASCII 字符。若没有对应字符，则返回（转 none）。
461 1:   movb (%ebx,%eax),%al            // 将扫描码作为索引值，取对应的 ASCII 码→al。
462     orb %al,%al                    // 检测看是否有对应的 ASCII 码。
463     je none                        // 若没有（对应的 ASCII 码=0），则返回。
    // 若 ctrl 键已按下或 caps 键锁定，并且字符在 'a'--'}' (0x61--0x7D) 范围内，则将其转成
    // 大写字母 (0x41--0x5D)。
464     testb $0x4c,mode                /* ctrl or caps */ /* 控制键已按下或 caps 亮? */
465     je 2f                          // 没有，则向前跳转标号 2 处。
466     cmpb $'a',%al                  // 将 al 中的字符与 'a' 比较。
467     jb 2f                          // 若 al 值 < 'a'，则转标号 2 处。
468     cmpb $'}',%al                  // 将 al 中的字符与 '}' 比较。
```

```

469         ja 2f                                // 若 al 值>'`', 则转标号 2 处。
470         subb $32,%al                          // 将 al 转换为大写字符(减 0x20)。
// 若 ctrl 键已按下, 并且字符在 '`_-' (0x40—0x5F) 之间, 即是大写字符, 则将其转换为
// 控制字符 (0x00—0x1F)。
471 2:      testb $0x0c,mode                      /* ctrl */ /* ctrl 键同时按下吗? */
472         je 3f                                // 若没有则转标号 3。
473         cmpb $64,%al                          // 将 al 与 '@' (64) 字符比较, 即判断字符所属范围。
474         jb 3f                                // 若值<'@', 则转标号 3。
475         cmpb $64+32,%al                      // 将 al 与 '`' (96) 字符比较, 即判断字符所属范围。
476         jae 3f                               // 若值>'`', 则转标号 3。
477         subb $64,%al                          // 否则 al 减 0x40, 转换为 0x00—0x1f 的控制字符。
// 若左 alt 键同时按下, 则将字符的位 7 置位。即此时生成值大于 0x7f 的扩展字符集中的字符。
478 3:      testb $0x10,mode                      /* left alt */ /* 左 alt 键同时按下? */
479         je 4f                                // 没有, 则转标号 4。
480         orb $0x80,%al                        // 字符的位 7 置位。
// 将 al 中的字符放入读缓冲队列中。
481 4:      andl $0xff,%eax                       // 清 eax 的高字和 ah。
482         xorl %ebx,%ebx                       // 由于放入队列字符数<=4, 因此需把 ebx 清零。
483         call put_queue                       // 将字符放入缓冲队列中。
484 none:   ret

```

图 28~29: 函数 do\_self 的解释

参考书提到, 函数 do\_self 用于处理“普通”键, 也即含义没有变化并且只有一个字符返回的键。与阶段 1 类似的, 需要在函数 put\_queue 前调用函数 checkChLine。

```

do_self:
    lea alt_map,%ebx
    testb $0x20,mode                          /* alt-gr */
    jne 1f
    lea shift_map,%ebx
    testb $0x03,mode
    jne 1f
    lea key_map,%ebx
1:      movb (%ebx,%eax),%al
    orb %al,%al
    je none
    testb $0x4c,mode                          /* ctrl or caps */
    je 2f
    cmpb $'a,%al
    jb 2f
    cmpb $'}',%al
    ja 2f
    subb $32,%al
2:      testb $0x0c,mode                      /* ctrl */
    je 3f
    cmpb $64,%al
    jb 3f
    cmpb $64+32,%al
    jae 3f
    subb $64,%al
3:      testb $0x10,mode                      /* left alt */
    je 4f
    orb $0x80,%al
4:      andl $0xff,%eax
    xorl %ebx,%ebx
    call checkChLine
    call put_queue
none:   ret

```

图 30: 在 do\_self 中调用 checkChLine (第一次修改)

上图是第一次修改，但是在完成后面的实验步骤之后，运行 linux-0.11 系统时发现无法正常输入字符。查询资料之后发现，这是由于函数 checkChLine 没有返回值，所以需要额外保护%eax（不是%rax，该系统为 32 位）寄存器，否则下面的函数 put\_queue 无法正常工作。

```
4:      andl $0xff,%eax
      xorl %ebx,%ebx
/*      push %rax
*/
      push %eax
      call checkChLine
/*      pop %rax
*/
      pop %eax
      call put_queue
none:   ret
```

图 31：在 do\_self 中调用 checkChLine（第二次修改）

### 3、修改程序 console.c

编写函数 checkChLine，该函数用于检查输入字符序列是否是连续的“2022211073”或“2022211073-”。如果是，改变 flagFlag 的状态，以激活\*功能。

```
void checkChLine(char c)
{
    const char id[11] = "2022211073-";
    int cnt = 0;
    if(cnt < 11 && id[cnt] == c)
        cnt++;
    else
    {
        if(cnt == 11)
            f12Flag = 0;
        else if(cnt == 10)
            f12Flag = 1;
        cnt = 0;
    }
    return;
}
```

图 30：函数 checkChLine（第一次编写）

上图是第一次编写，但是在运行 linux-0.11 系统时发现，输入“2022211073”时\*功能并没有被激活。

重新检查，发现该函数是在每键入一个值时重新调用一遍，因此其中的计数变量会被反复地初始化为 0，无法达到条件中 cnt==11 或 cnt==10。可以通过设置 cnt 为静态变量解决这个问题。



```

void checkChLine(char c)
{
    const char id[11] = "2022211073-";
    /*
    int cnt = 0;
    */
    static int cnt = 0;
    if(cnt < 11 && id[cnt] == c)
        cnt++;
    else
    {
        if(cnt == 11)
            f12Flag = 0;
        else if(cnt == 10)
            f12Flag = 1;
        cnt = 0;
    }
    return;
}

```

图 31：函数 checkChLine（第二次编写）

这是正确的函数 checkChLine。

在 console.c 文件前部进行函数 checkChLine 的声明：

```
void checkChLine(char c);
```

图 32：声明函数 checkChLine

5、运行 linux-0.11 操作系统，测试功能

6、回到目录 linux-0.11 下，依次执行阶段 1 的指令，运行 linux-0.11 操作系统。

7、可以看到，输入“2022211073”，所有的‘l’ ‘n’ 都成功变成‘\*’，\*功能开启；重新输入“2022211073-”，\*功能关闭。其他输入（如不连续的“2022221111073” “222022211073”）都不会激活\*功能。该阶段通过。

```

Loading system ...
Partition table ok.
39044/62000 free blocks
19520/20666 free inodes
3454 buffers = 3536896 bytes buffer space
Free mem: 12582912 bytes
Ok.
[/usr/root]# 2022211073
2022211073: comma*d *ot fou*d
[/usr/root]# *iushiyua*
*iushiyua*: comma*d *ot fou*d
[/usr/root]# 2022211073-
2022211073-: command not found
[/usr/root]# liushiyuan
liushiyuan: command not found
[/usr/root]# 20222111073
20222111073: command not found
[/usr/root]# liushiyuan
liushiyuan: command not found
[/usr/root]# 222022211073
222022211073: command not found
[/usr/root]# liushiyuan
liushiyuan: command not found
[/usr/root]#

```

图 33：阶段 2 成功



## 五、总结体会

## 六、诚信声明

在完成本次实验过程中，我曾分别与以下各位同学就以下方面做过交流：

此外，我还参考了以下资料：

在我提交的程序中，还在对应的位置以注释形式记录了具体的参考内容。

我独立完成了本次实验除以上方面之外的所有工作，包括分析、设计、编码、调试与测试。

我清楚地知道，从以上方面获得的信息在一定程度上降低了实验的难度，可能影响起评分。

我从未使用他人代码，不管是原封不动地复制，还是经过某些等价转换。

我未曾也不会向同一课程（包括此后各届）的同学复制或公开我这份程序的代码，我有义务妥善保管好它们。

我编写这个程序无意于破坏或妨碍任何计算机系统的正常运行。

我清楚地知道，以上情况均为本课程纪律所禁止，若违反，对应的实验成绩将按照 0 分计。

(签名)