

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： 2019211309

学 号： 2019211397

姓 名： 毛子恒

学 院： 计算机学院

2020 年 12 月 10 日

一、实验目的

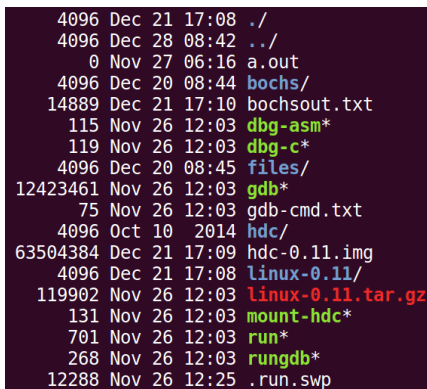
1. 理解 I/O 系统调用函数和 C 标准 I/O 函数的概念和区别；
2. 建立内核空间 I/O 软件层次结构概念，即与设备无关的操作系统软件、设备驱动程序和中断服务程序；
3. 了解 Linux-0.11 字符设备驱动程序及功能，初步理解控制台终端程序的工作原理；
4. 通过阅读源代码，进一步提高 C 语言和汇编程序的编程技巧以及源代码分析能力；
5. 锻炼和提高对复杂工程问题进行分析的能力，并根据需求进行设计和实现的能力。

二、实验环境

1. 硬件：学生个人电脑（x86-64）
2. 软件：macOS Big Sur 11.0.1, VMware Fusion Player 12.1.0, 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

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



```
4096 Dec 21 17:08 ./
4096 Dec 28 08:42 ../
0 Nov 27 06:16 a.out
4096 Dec 20 08:44 bochs/
14889 Dec 21 17:10 bochsout.txt
115 Nov 26 12:03 dbg-asm*
119 Nov 26 12:03 dbg-c*
4096 Dec 20 08:45 files/
12423461 Nov 26 12:03 gdb*
75 Nov 26 12:03 gdb-cmd.txt
4096 Oct 10 2014 hdc/
63504384 Dec 21 17:09 hdc-0.11.img
4096 Dec 21 17:08 linux-0.11/
119902 Nov 26 12:03 linux-0.11.tar.gz
131 Nov 26 12:03 mount-hdc*
701 Nov 26 12:03 run*
268 Nov 26 12:03 run gdb*
12288 Nov 26 12:25 .run.swp
```

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：

- ✧ Phase 1
键入 F12，激活*功能，键入学生本人姓名拼音，首尾字母等显示*
比如：zhangsan，显示为：*ha*gsa*
- ✧ Phase 2
键入“学生本人学号”：激活*功能，键入学生本人姓名拼音，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

实验 4 的其他说明见 lab4.pdf 课件和爱课堂中虚拟机环境搭建相关内容。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、源代码的分析及修改

(一) 有关背景知识

在 Linux 0.11 系统中可以使用两类终端。一类是主机上的控制台终端，另一类是串行硬件终端设备。控制台终端由内核中的键盘终端处理程序 keyboard.S 和显示控制程序 console.c 进行管理。控制台终端对应有一个 tty_struct 数据结构，主要用来保存终端设备当前的参数、字符 IO 缓冲队列等信息，其中三个缓冲队列，分别是 read_q, write_q 和 secondary，分别保存从键盘输入的原始字符序列、写到控制台显示屏的数据、从 read_q 取出的经过行规则程序处理的数据。

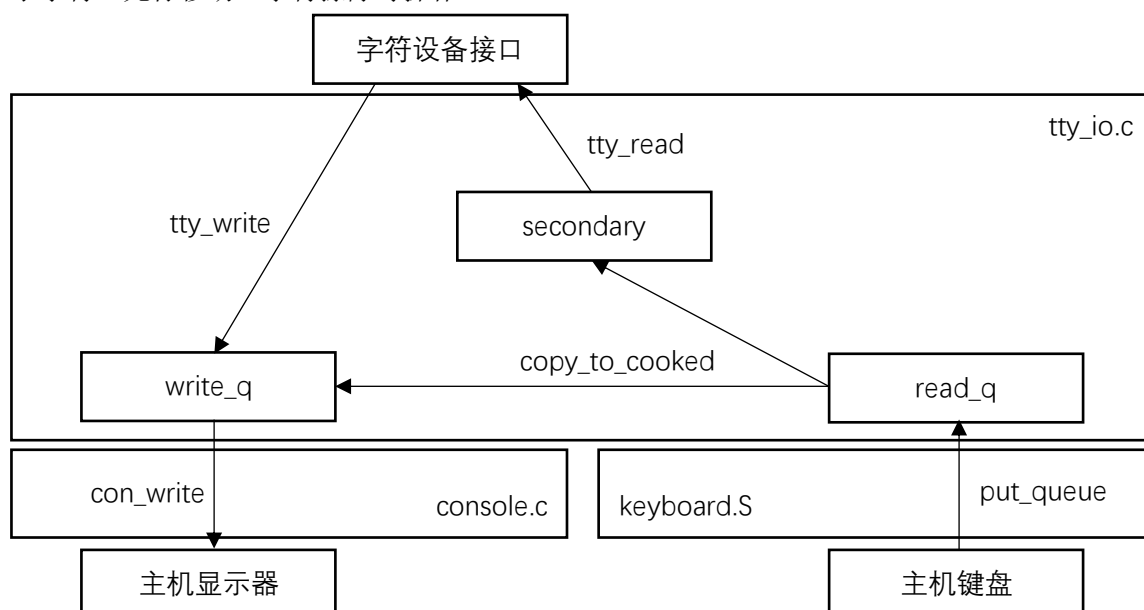
tty_io.c 中包含字符设备的上层接口函数，包括终端读函数 tty_read 和终端写函数 tty_write，当文件系统中操作字符设备文件时会调用这两个函数。另外行规则函数 copy_to_cooked 也在这个函数中实现，下文中有解释。

keyboard.S 和 console.c 接收上层 tty_io.c 程序传递下来的显示字符或控制信息，并控制在主机屏幕上字符的显示，同时控制台把键盘按键产生的代码经由 keyboard.S 传送到 tty_io.c 程序去处理。这两个程序实际上是 Linux 系统主机中使用显示器和键盘模拟一个硬件终端设备的仿真程序。

keyboard.S 中主要包括键盘中断处理程序，它首先根据键盘特殊键的状态设置状态标记变量 mode 的值，然后根据引起键盘终端的按键扫描码，调用已经编排成跳转表的相应扫描码处理子程序，把扫描码对应的字符放入 read_q 中，之后调用 tty_io.c 中的 do_tty_interrupt 函数。

do_tty_interrupt 函数包含对 copy_to_cooked 函数的调用，这个函数对 read_q 中的字符进行适当处理之后放到 secondary 队列中，在此期间，如果设置了回显标志，则字符还会被放入到 write_q 队列中，以在终端屏幕上显示刚键入的字符。

console.c 中的 con_write 函数从 write_q 中取出字符序列，根据字符的性质进行在屏幕终端上显示字符、光标移动、字符擦除等操作。



(二) Phase 1

大体思路是通过在 console.c 的 con_write 函数中改变字符 c 的值来改变回显结果，首先需要在 console.c 中增加一个标记 F12Flag，用来判断是否激活*功能，当 F12Flag 为真，并且 c 字符是 m

或者 g 时，将 c 置为*。同时，实现一个函数 changeF12Flag 用来将 F12Flag 取反。在 keyboard.S 中增加指令，使得检测到按下 F12 时调用 changeF12Flag 函数。

首先在 console.c 的 con_write 函数之前加入如下几行：

```
445 int F12Flag=0;
446
447 void changeF12Flag(void)
448 {
449     F12Flag = !F12Flag;
450 }
```

再在 con_write 函数中加入如下语句，在 F12 被按下时将回显置为*

```
462         if (c>31 && c<127) {
463             if (x>=video_num_columns) {
464                 x -= video_num_columns;
465                 pos -= video_size_row;
466                 lf();
467             }
468             if (F12Flag && (c=='m' || c=='q')) c='*';
469             __asm__ ("movb attr,%%ah\n\t"
470                    "movw %%ax,%1\n\t"
471                    ::"a" (c),"m" (*(short *)pos)
472                    );
```

观察 keyboard.S 函数，找到跳转表中 F12 的信息：

```
524     .long cursor,cursor,cursor,cursor /* 50-53 dn pgdn ins del */
525     .long none,none,do_self,func      /* 54-57 sysreq ? < f11 */
526     .long func,none,none,none         /* 58-5B f12 ? ? ? */
527     .long none,none,none,none         /* 5C-5F ? ? ? ? */
```

发现 F12 使用 func 进行处理，查看 func 的说明：

```
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。
```

于是我们在 226 行之后，即判断是功能键 F12，加上以下指令：

```
226     ja end_func
227     call changeF12Flag
228 ok_func:
```

输入 make clean 和 make 指令，再启动 run 程序：

输入 maoziheng，键入回车，按下 F12，再输入 maoziheng，键入回车，按下 F12，再输入 maoziheng。得到回显如下图：

```
[usr/root]# maoziheng
maoziheng: command not found
[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
B*aozihen*
[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
[usr/root]# maoziheng
maoziheng: command not found
[usr/root]#
```

发现功能可以正常激活/反激活。

(三) Phase 2

思路仍然是通过在 console.c 的 con_write 函数中改变字符 c 的值来改变回显结果，但是标记相比上个阶段有一些区别，标记为 IDFlag，值为 0~20 的整数，分别表示 20 个不同的状态。

状态 0 表示没有输入字符串，状态 1 表示输入数字 2，状态 2 表示输入数字 20，以此类推，状态 10 表示输入 2019211397；状态 11 表示再次输入 2，状态 12 表示再次输入 20，以此类推，状态 20 表示输入 2019211397-。状态 ≥ 10 时激活*功能。

各状态之间转移：如果输入了正确的字符则跳转至下一个状态，即状态 0 输入 2 转到状态 1；状态 1 输入 0 转到状态 2；状态 2 输入 1 转到状态 3；依次类推。当状态 < 10 时，如果没有输入匹配的数字，则回到状态 0，当状态 ≥ 10 时，如果没有输入匹配的数字，则回到状态 10。特别的是如果在状态 5（即 20192）输入字符 0 则回到状态 2（即 20），在状态 15 输入字符 0 则回到状态 12。如果在 < 10 的非 0、4 状态输入 2 则回到 1 状态，在 ≥ 10 的非 10、14 状态输入 2 则回到 11 状态。

大体思路是根据输入不同的字符调用不同的函数，如果输入 0、1、2、3、7、9、- 字符中的一个则调用相应的跳转函数，否则调用状态重置函数（即重置到状态 0 或者 10）。

首先在 console.c 的 con_write 函数之前加入如下定义和函数，分别表示状态变量和输入不同数字时的状态转移：

```
445 int IDFlag=0;
446
447 void changeFlag0(void)
448 {
449     if (IDFlag == 1) IDFlag = 2;
450     else if (IDFlag == 5) IDFlag = 2;
451     else if (IDFlag == 11) IDFlag = 12;
452     else if (IDFlag == 15) IDFlag = 12;
453     else if (IDFlag < 10) IDFlag = 0;
454     else IDFlag = 10;
455 }
456
457 void changeFlag1(void)
458 {
459     if (IDFlag == 2) IDFlag = 3;
460     else if (IDFlag == 5) IDFlag = 6;
461     else if (IDFlag == 6) IDFlag = 7;
462     else if (IDFlag == 12) IDFlag = 13;
463     else if (IDFlag == 15) IDFlag = 16;
464     else if (IDFlag == 16) IDFlag = 17;
465     else if (IDFlag < 10) IDFlag = 0;
466     else IDFlag = 10;
467 }
468
469 void changeFlag2(void)
470 {
471     if (IDFlag == 0) IDFlag = 1;
472     else if (IDFlag == 4) IDFlag = 5;
473     else if (IDFlag == 10) IDFlag = 11;
474     else if (IDFlag == 14) IDFlag = 15;
475     else if (IDFlag < 10) IDFlag = 1;
476     else IDFlag = 11;
477 }
478
479 void changeFlag3(void)
480 {
481     if (IDFlag == 7) IDFlag = 8;
482     else if (IDFlag == 17) IDFlag = 18;
483     else if (IDFlag < 10) IDFlag = 0;
484     else IDFlag = 10;
485 }
486
487 void changeFlag7(void)
488 {
489     if (IDFlag == 9) IDFlag = 10;
490     else if (IDFlag == 19) IDFlag = 20;
491     else if (IDFlag < 10) IDFlag = 0;
492     else IDFlag = 10;
493 }
494
495 void changeFlag9(void)
496 {
497     if (IDFlag == 8) IDFlag = 9;
498     else if (IDFlag == 3) IDFlag = 4;
499     else if (IDFlag == 18) IDFlag = 19;
500     else if (IDFlag == 13) IDFlag = 14;
501     else if (IDFlag < 10) IDFlag = 0;
502     else IDFlag = 10;
503 }
504
505 void changeFlag_(void)
506 {
507     if (IDFlag == 20) IDFlag = 0;
508     else if (IDFlag < 10) IDFlag = 0;
509     else IDFlag = 10;
510 }
511
512 void resetFlag(void)
513 {
514     if (IDFlag < 10) IDFlag = 0;
515     else IDFlag = 10;
516 }
```

changeFlag0, changeFlag1, changeFlag2, changeFlag3, changeFlag7, changeFlag9, changeFlag_ 分别表示输入数字 0, 1, 2, 3, 7, 9 和符号-时的状态转移，resetFlag 表示输入其他字符时的状态转移。

在 con_write 函数中加入以下语句，表示当状态 ≥ 10 时激活*功能：

```

523         if (x>=video_num_columns) {
524             x -= video_num_columns;
525             pos -= video_size_row;
526             lf();
527         }
528         if (IDFlag>=10 && (c=='\n' || c=='\t')) c='*';
529         __asm__ ("movb attr,%%ah\n\t"
530                "movw %%ax,%%1\n\t"
531                :: "a" (c), "m" (*(short *)pos)
532                );
533         pos += 2;
534         x++;

```

观察 keyboard.S 函数，找到跳转表中数字 0~9 和字符-的有关信息：

```

531 key_table:
532 .long none,do_self,do_self,do_self /* 00-03 s0 esc 1 2 */
533 .long do_self,do_self,do_self,do_self /* 04-07 3 4 5 6 */
534 .long do_self,do_self,do_self,do_self /* 08-0B 7 8 9 0 */
535 .long do_self,do_self,do_self,do_self /* 0C-0F + ' bs tab */
536 .long do_self,do_self,do_self,do_self /* 10-13 q w e r */
537 .long do_self,do_self,do_self,do_self /* 14-17 t y u i */
538 .long do_self,do_self,do_self,do_self /* 18-1B o p } ^ */
539 .long do_self,ctrl,do_self,do_self /* 1C-1F enter ctrl a s */
540 .long do_self,do_self,do_self,do_self /* 20-23 d f g h */
541 .long do_self,do_self,do_self,do_self /* 24-27 j k l | */
542 .long do_self,do_self,lshift,do_self /* 28-2B { para lshift , */
543 .long do_self,do_self,do_self,do_self /* 2C-2F z x c v */
544 .long do_self,do_self,do_self,do_self /* 30-33 b n m , */
545 .long do_self,minus,rshift,do_self /* 34-37 . - rshift * */
546 .long alt,do_self,caps,func /* 38-3B alt sp caps f1 */
547 .long func,func,func,func /* 3C-3F f2 f3 f4 f5 */
548 .long func,func,func,func /* 40-43 f6 f7 f8 f9 */
549 .long func,num,scroll,cursor /* 44-47 f10 num scr home */
550 .long cursor,cursor,do_self,cursor /* 48-4B up pgup - left */
551 .long cursor,cursor,do_self,cursor /* 4C-4F n5 right + end */
552 .long cursor,cursor,cursor,cursor /* 50-53 dn pgdn ins del */
553 .long none,none,do_self,func /* 54-57 sysreq ? < f11 */

```

观察到这些字符都使用 do_self 进行处理，查看 do_self 的说明：

```

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 键同时按下了吗? */
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

```

发现在 481 行之前处理按下功能键的情况，于是修改标记 4 之后的部分：

```

481 4:  cmpb $0x30,%al
482      jne 5f
483      call changeFlag0
484      jmp 12f
485 5:  cmpb $0x31,%al
486      jne 6f
487      call changeFlag1
488      jmp 12f
489 6:  cmpb $0x32,%al
490      jne 7f
491      call changeFlag2
492      jmp 12f
493 7:  cmpb $0x33,%al
494      jne 8f
495      call changeFlag3
496      jmp 12f
497 8:  cmpb $0x37,%al
498      jne 9f
499      call changeFlag7
500      jmp 12f
501 9:  cmpb $0x39,%al
502      jne 10f
503      call changeFlag9
504      jmp 12f
505 10: cmpb $0x2d,%al
506      jne 11f
507      call changeFlag_
508      jmp 12f
509 11: call resetFlag
510 12: andl $0xff,%eax
511      xorl %ebx,%ebx
512      call put_queue
513 none: ret

```

这一段指令根据输入字符的不同调用之前编写的各函数，实现状态之间的转移。

输入 make clean 和 make 指令，再启动 run 程序，输入

maoziheng2019211397maoziheng2019211397-

maoziheng20192019211397maoziheng201922019212019211397-maoziheng

得到回显如下图：

```

[/usr/root]# maoziheng2019211397*aozihen*2019211397-maoziheng20192019211397*aozihen*201922019212019211397-maoziheng_

```

可以看到激活/反激活功能无误。

五、总结体会

在本次实验中我阅读了大量材料，了解了 Linux 内核中控制台终端的键盘驱动程序实现，并且研究了一部分代码细节，深感其中的奥妙，并且为之震撼。

在初步研究回显部分的实现之后我发现需要增添的代码量不小，逻辑也有些复杂，并且这段程序没有有效的手段可以调试，于是花费了一些时间画出状态图，但是在实现时还是因为输入数字 2 的特殊转移方式没有注意到而花费了很多时间找 bug。最终我的实现能够考虑所有可视字符，并且在大部分极端条件下都能够正常检测学号序列并且改变状态。在本次的实验中我第一次上手编写了汇编代码，对其逻辑有了更深刻的了解。

最后我的实现仍然有诸多不足之处，比如在第二部分实现中没有考虑其他特殊按键的情况，在输入学号中途不会考虑按下 F 功能区按键的情况。另外在*功能关闭的情况下输入 2019211397-仍然会激活功能，这是一个小 bug。而限于我的能力没有解决这些问题。