北京郵電大學

实验报告

题目：键盘驱动程序的分析与修改

班级： 2022211320

学号： 2022212408

姓名： 胡宇杭

学院： 计算机学院 (国家示范性软件学院)

2023 年 12 月 10 日

目录

1 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。

2 实验环境

- 硬件：学生个人电脑 (**x86-64**)
- 软件：**Windows 11，VMware Workstation 16 Player，32 位 Linux-Ubuntu 16.04.1**
- **gcc-3.4** 编译环境
- **GDB** 调试工具

3 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：

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。**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

4 源代码的分析及修改

4.1 源代码分析

注：此处只给出针对一次按键

通过查阅 **lab4** 课件，我们知道当敲击了一下键盘，键盘硬中断，进而调用 **keyboard.S** 中的 **keyboard_interrupt** 函数，这就是我们分析的起点。

4.1.1 keyboard.S

1. 首先查看 **keyboard_interrupt** 函数的源代码，在一系列初始化和读取过程后，函数首先检查读取到的扫描码是不是 **0xe0** 或 **0xe1**。如果是，则转跳到设置标志代码处，由于本次实验要求的内容不涉及此处，故跳过此处，接着向下分析。

```
inb $0x60,%al
cmpb $0xe0,%al
je set_e0
cmpb $0xe1,%al
```

其中，**inb $0x60, %al** 对应操作为读取扫描码。

2. 紧接着。函数通过跳转表调用对应的字符处理函数，我们向下查阅跳转表，根据注释找到对应的字符处理函数（以 **F12** 键为例）

```
call key_table(,%eax,4)

.long none,none,do_self,func            /* 54-57 sysreq ? < f11 */
.long func,none,none,none               /* 58-5B f12 ? ? ? */
```

发现 **F12** 对应的字符处理函数为 **func**。

3. **func** 函数：

   - 查阅 **func** 函数。函数首先通过 **subb** 命令获得了功能键 **F1 - F10** 的索引。如果索引小于零，说明不合法，直接跳转至 **end_func**；接下来判断是否属于 **F1 - F12**，此处由于 **F11**、**F12** 对应的扫描码和前几个功能键不连续，因此要单独判断。如果是，就调用 **ok_func**。

```
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
```

   - 在函数 **ok_func** 中，函数首先判断空间是否足够（因为对应转义字符是由 4 个字符组成的），然后根据跳转表取得相应的转义字符。

```
ok_func:
        cmpl $4,%ecx            /* check that there is enough room */
        jl end_func
        movl func_table(,%eax,4),%eax
        xorl %ebx,%ebx
        jmp put_queue
```

   - 查阅跳转表可知，**F12** 对应的转义字符序列为 **esc**、’[’、’[’、’**L**’(1b, 5b, 5b, 4c)

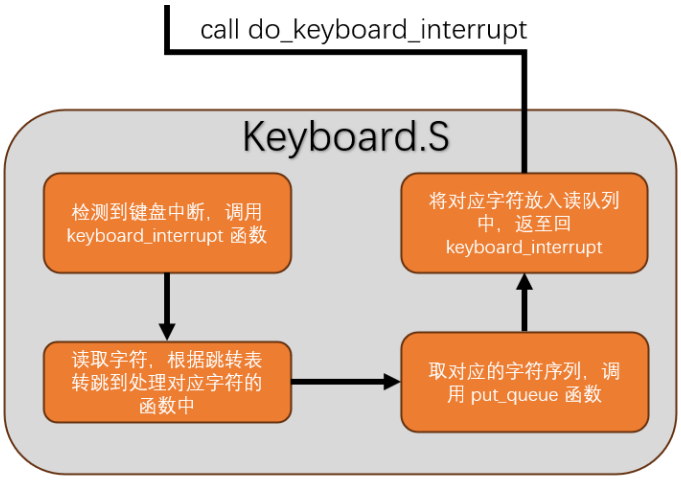
```
/*
 * function keys send F1:'esc [ [ A' F2:'esc [ [ B' etc.
 */
func_table:
        .long 0x415b5b1b,0x425b5b1b,0x435b5b1b,0x445b5b1b
        .long 0x455b5b1b,0x465b5b1b,0x475b5b1b,0x485b5b1b
        .long 0x495b5b1b,0x4a5b5b1b,0x4b5b5b1b,0x4c5b5b1b
```

4. 字符处理函数结束后，程序跳转至 **put_queue** 函数，将字符放入读队列中，接着返回继续执行 **keyboard_interrupt** 函数。

5. 紧接着是一段硬件处理语句，与本实验关系不大，不做赘述。接下来 **pushl $0** 命令指定 **tty** 参数为 **0**，然后调用了 **tty_io.c** 中的函数 **do_keyboard_interrupt**。

```
pushl $0
call do_tty_interrupt
```

通过上述分析，我们得知，在按下键盘后，系统在 **keyboard.S** 中的执行顺序为 **检测到键盘中断 → keyboard.S → 调用 keyboard_interrupt() → 根据跳转表跳转至判断对应字符处理函数 → 字符处理完毕后转跳至 put_queue 函数 → 字符入队 → 返回 keyboard_interrupt() 函数 → 调用 tty_io.c 的 do_tty_interrupt()**，可得如下流程图：



4.1.2 ttc_io.c

1. 在函数 **do_tty_interrupt** 中，执行的唯一语句是调用 **copy_to_cooked** 函数，参数为 **tty_table + tty**，其中，**tty** 是之前传入的参数，用作偏移量来指定 **tty** 设备。

```
void do_tty_interrupt(int tty)
{
        copy_to_cooked(tty_table+tty);
}
```

2. **copy_to_cooked** 函数由一个 **while** 循环和依据唤醒语句组成，其中，**while** 循环主要涉及以下操作：

- 判断当前读缓冲队列是否为空以及辅助缓冲队列是否已满。如果都为否，则从读队列中提取一个字符。

```
while (!EMPTY(tty->read_q) && !FULL(tty->secondary)) {
        GETCH(tty->read_q,c);
```

- 首先判断根据各标志置位状态对读入字符进行一系列的判断和处理，由于不涉及实验要求的核心部分，这里不做详细说明。
- 当前字符如果是换行符或文件结束符 **EOF**，说明一行已经读取完，让当前字符行数 **secondary.data** 增加 1，方便后续 **tty_read** 函数判断是否进行读取。

```
if (c==10 || c==EOF_CHAR(tty))
        tty->secondary.data++;
```

- 当 **tty** 设备回显标志在置位状态时，会对换行符以及控制字符做特殊处理，然后将字符放入写缓冲队列中。然后调用写函数 **tty->write()**。由于此处 **tty** 为控制台终端，因此会直接调用 **console.c** 中的 **con_write** 函数。

```
if (L_ECHO(tty)) {
        if (c==10) {
                PUTCH(10,tty->write_q);
                PUTCH(13,tty->write_q);
        } else if (c<32) {
                if (L_ECHOCTL(tty)) {
                        PUTCH('^',tty->write_q);
                        PUTCH(c+64,tty->write_q);
                }
        } else
                PUTCH(c,tty->write_q);
        tty->write(tty);
}
```

- 执行完以上步骤后将处理后的字符放入辅助缓冲队列中供便上级程序读入。最后，在函数结束前唤醒等待该辅助缓冲队列的进程。

```
        PUTCH(c,tty->secondary);
}
wake_up(&tty->secondary.proc_list);
```

3. 理论上来说我们现在应该顺着流程跳转到 **console.c** 中继续分析，但此处为了方便阅读（省事），决定对 **tty_io.c** 中的剩余部分分析完毕后再进入下一模块。
4. 上文提到，在 **copy_to_cooked** 函数中，程序会唤醒等待该辅助缓冲队列的进程。通过查阅相关资料可知，对于控制台终端而言，在系统启动时就会立刻创建一个 **read** 进程，然后调用 **tty_read** 函数。但由于辅助队列为空，一直处于睡眠状态。唤醒后，进程继续执行函数 **tty_read**。

5. 在 **tty_read** 函数中，首先进行了一系列预操作，然后进行判断：如果辅助队列为空或当前规范模式标志为置位、当前字符行数为 0、空闲空间大于 20 时会让进程睡眠，这也是上文 **read** 调用 **tty_read** 后处于睡眠状态的原因。而当字符行数不为 0 或行数为 0 但是剩余空间小于等于 20 时，会继续执行取字符操作。

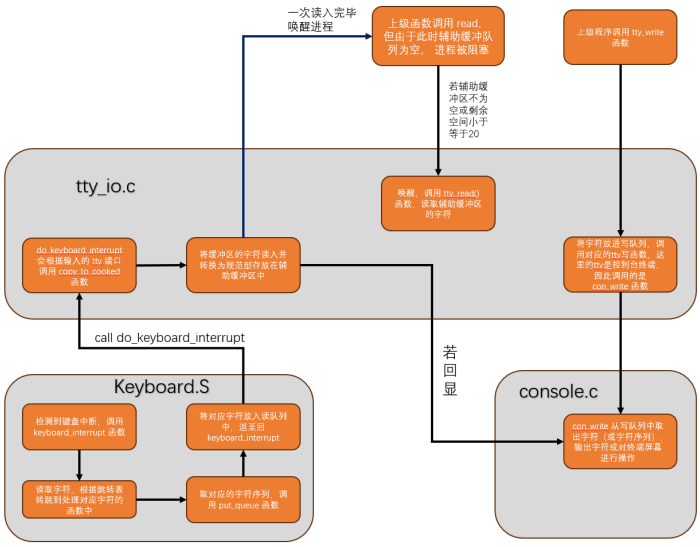
```
if (EMPTY(tty->secondary) || (L_CANON(tty) &&
!tty->secondary.data && LEFT(tty->secondary)>20)) {
        sleep_if_empty(&tty->secondary);
        continue;
}
```

6. 字符被上级程序读取后会根据输入做出反馈，调用 **write**，进而调用 **tty_write** 函数，将字符处理后放入写缓冲队列，并调用写函数 **tty->write()**，即 **con_write** 函数。

```
            PUTCH(c,tty->write_q);
    }
    tty->write(tty);
```

至此，我们可以向之前一样总结出在 **tty_io.c** 程序的执行顺序：**调用 copy_to_cooked 函数，将缓冲区的字符替换为规范模式存到辅助队列中；**

- **若辅助缓冲区不为空或剩余空间小于等于 20 → 唤醒进程，调用 tty_read 函数 → 反馈 → 调用 tty_write 函数，将字符放入写队列 → 调用写函数 con_write 函数**
- **如果 echo 置位，直接放入写队列中 → 调用写函数 con_write 函数**



4.2 源代码修改

4.2.1 Phase 1

要想实现本功能，我们需要完成两件事：

- 检测键盘键入了 **F12**
- 当检测到键入后将对应字符输出更改为 *

首先，我们可以设置全局变量 **f12Flag** 标志，当其置位时打开 * 功能，反之则关闭。接下来，我们分别实现检测和修改功能：

由于不同方法对应的修改功能完全一致，所以先实现修改功能。

- 修改功能：根据流程图可知，控制台终端输出发生在函数 **con_write** 函数中，因此我们的修改功能实现应该在其中。
  1. 首先，我们发现该函数是由一个 **while** 循环以及一个 **set_cursor** 函数组成，在循环中，依次取出写缓冲队列中的字符并进行对特殊字符进行处理，然后输出到控制台终端上。
  2. 因此，我们可以在其每次取出字符时判断我们的 **f12Flag** 是否处于置位。如果是，并且该字符是姓名的首尾字母，就将其赋值为 *。由于 * 不是需要处理的特殊字符，所以我们不需要做其他更改。

     ```
     if (f12Flag && ((c == 'H') || (c == 'G') || (c == 'h') || (c == 'g'))) c = '*';
     ```

  3. 同时，我们需要为检测功能预留修改接口，定义函数 **change_f12Flag**，其功能为每次调用会改变一次 **f12Flag** 的状态。

     ```
     void change_f12Flag(void)
     {
             f12Flag ^= 1;
     }
     ```

- 检测功能：根据之前得到的流程图，我们知道检测需要在读入时执行。如此，我们的选择可以是在 **keyboard.S** 中读取字符的过程中检测或在 **tty_io.c** 中 **copy_to_cooked** 读入辅助缓冲队列，抑或是 **con_write** 从写辅助队列取出时进行检测。接下来会分别给出两种检测方式的实现。（最后一种在 **phase 2** 中给出）
  1. 由之前的分析，我们知道在键入字符后 **keyboard.S** 会根据输入的字符调用对应的处理函数，我们根据专挑表找到 **F12** 对应的处理函数 **func**。

在 **func** 函数中，函数会判断是否功能键 **F1 - F12**，通过最后两个比较，我们发现其会将小于 10 和大于 11 的情况跳过，因此，这两个比较语句是用来判断键入的是否为 **F11 或 F12**，所以我们只需要再判断一次其是否为 **F12**，就能达到检测的目的，然后执行修改操作。

```
subb $18,%al
cmpb $10,%al
jb end_func
cmpb $11,%al
ja end_func
```

因此，我们可以先判断其是不是 **F11**，如果是，就跳转到 **ok_func**，否则调用之前定义的 **change_f12Flag** 函数修改 **f12Flag** 标志状态。

```
jnz ok_func
call change_f12Flag
```

2. 根据之前的分析，**F12** 对应的转义字符为 **esc**、‘[’、‘[’、‘**L**’(1b, 5b, 5b, 4c)，因此我们可以在 **copy_to_cooked** 函数的循环前定义相应的字符串和指针，每次读到相应的字符就将指针指向下一个位置，否则指令。如果全部读完，说明此时输入为 **F12**，调用 **change_f12Flag** 函数修改标识状态。接下来，我们对其进行实现。

   首先，由于要调用外部函数，因此需要先声明 **extern void change_f12Flag(void);**。

```
extern void change_f12Flag(void);
```

接着，在循环外定义字符串和指针变量。注意，此处指针变量不能设置为全局变量，因为此时如果连续键入 **esc**、’[’、’[’、’**L**’，程序会误以为成 **F12** 导致错误。

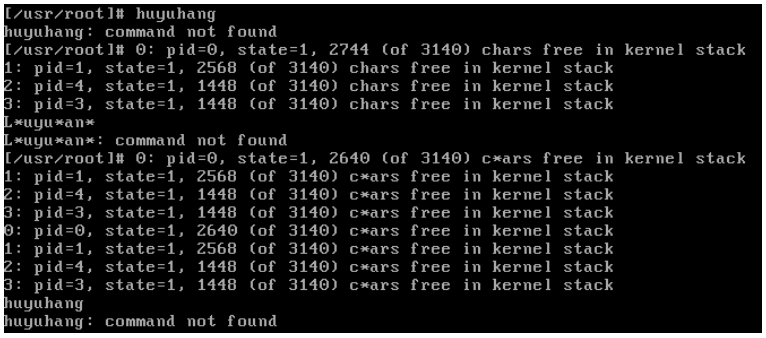
```
int ptr = 0;
char s[4];
s[0] = 0x1b, s[1] = s[2] = 0x5b, s[3] = 0x4c;
```

然后依据上述分析，判断当前字符是否与 **s[ptr]** 相等，如果相等，就将指针向后移动一位，否则置零，如果全检测完了说明此时键入的是 **F12**，调用 **change_f12Flag** 函数。

```
if (ptr < 4 && c == s[ptr]) ptr++;

if (ptr == 4) {
        change_f12Flag();
        ptr = 0;
}
```

经三次测试，第一次不开启 * 功能，正常输出；第二次键入 **F12**，开启 * 功能，输出被替换；第三次再次键入 **F12**，关闭 * 功能，正常输出。



4.2.2 Phase 2

Phase 2 与 **Phase 1** 的唯一区别是从检测字符 **F12** 变成了检测序列 **2022212408** 和 **2022212408-**。而对于序列的检测在 **Phase 1** 中已经实现，此处不在赘述。此处检测会使用第三种方案，在 **con_write** 中进行。

首先，定义指针变量 **ptr**，这里必须定义为全局变量，因为无法保证指定字符序列在同一次 **con_write** 函数调用中被全部读取。

```
int f12Flag = 0;
int ptr = 0;
```

接着对读入字符进行检测，首先检测其是否为 **-**，若是，则继续判断此时 **ptr** 是否等于 11，如果等于，说明前面已经读取到了字符序列 **2022212408**，构成关闭序列，将 **f12Flag** 置为 0。内层判断结束后别忘了将 **ptr** 也置零。

```
if (ptr == 11) {
        if (c == '-') f12Flag = 0;
        ptr = 0;
}
```

然后与 **Phase 1** 类似，判断当前字符是否与 **s[ptr]** 相等，如果相等，就将指针向后移动一位，否则置零，如果全检测完了说明此时读取到了字符序列 **2022212408**，调用 **change_f12Flag** 函数并让 **ptr** 加一。（由于需要在下次迭代判断是否为关闭指令，此处 **ptr** 不置零）

```
if (ptr < 10 && c == s[ptr]) ptr++;
else ptr = 0;

if (ptr == 10) {
        f12Flag = 1;
        ptr++;
}
```

接下来，我们键入如下字符序列来判断可行性 **huyuhang2022212408huyuhang2022212408-huyuhang**。

```
huyuhang2022212408*uyu*an*2022212408-huyuhang
```

输出与预期一致，**Phase 2** 完成。

5 总结体会

通过本次实验，我理解 **I/O** 系统调用函数和 **C** 标准 **I/O** 函数的概念和区别；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；了解 **Linux-0.11** 字符设备驱动程序及功能，初步理解了控制台终端程序的工作原理；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。