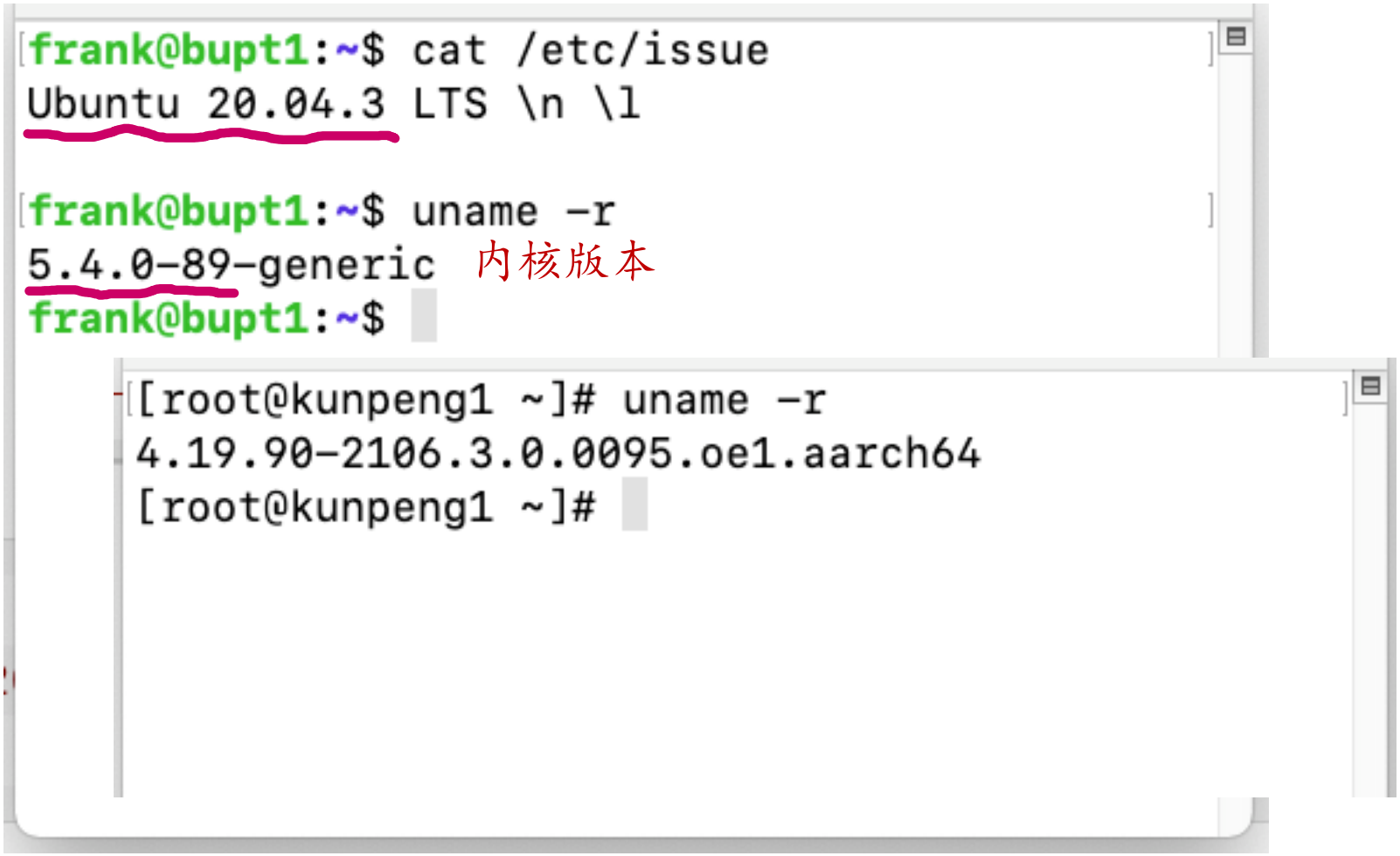
实验四

键盘驱动程序的分析与修改

Linux操作系统

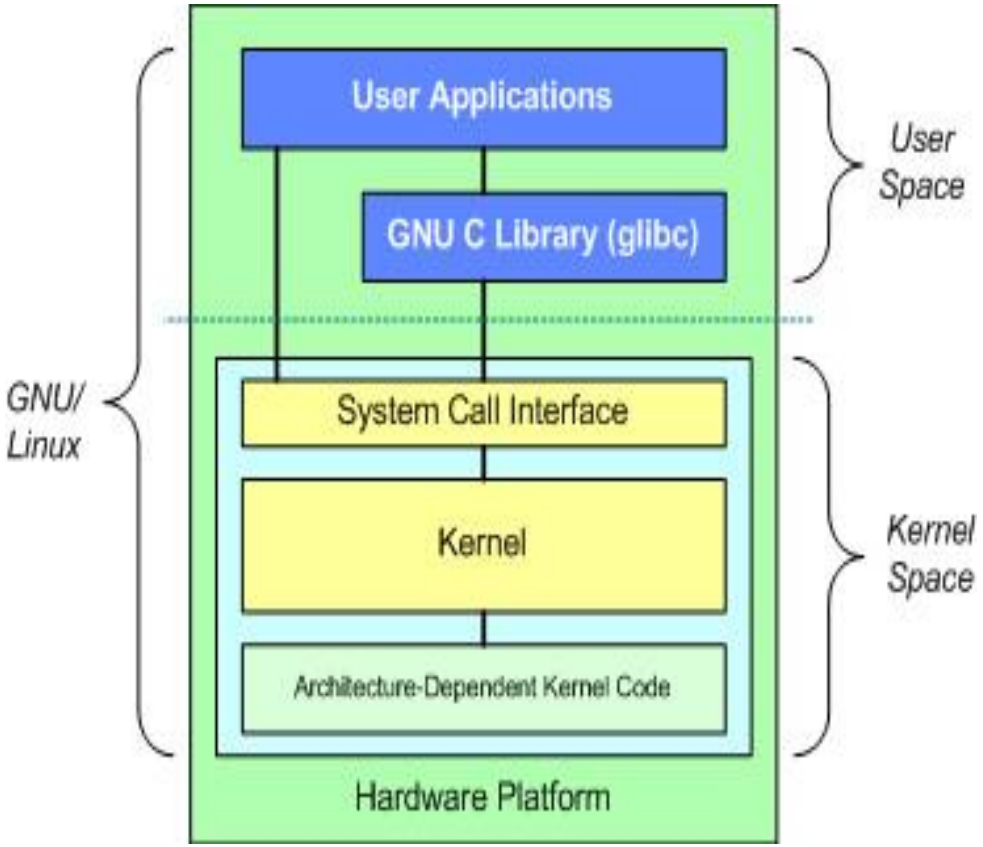

frank@bupt1:~$ cat /etc/issue
Ubuntu 20.04.3 LTS \n \l
frank@bupt1:~$ uname -r
5.4.0-89-generic
内核版本
frank@bupt1:~$
[root@kunpeng1 ~]# uname -r
4.19.90-2106.3.0.0095.oe1.aarch64
[root@kunpeng1 ~]#

Linux内核（1）

- Linux内核是Linux社区从零开始开发的原始软件。而Linux系统包括许多组件，有些是从头编写的，有些是从其他开发项目中借用的，还有一些是与其他团队合作创建的。
- 许多Linux操作系统发行商如RedHat、 Debian等都采用Linux内核，然后加入用户需要的工具软件和程序库，最终构成一个完整的操作系统
- 嵌入式Linux系统是运行在嵌入式硬件系统上的，也都包括了必要的工具软件和程序库

Linux内核（2）

- 内核是操作系统的核心部分，能为应用程序提供安全访问硬件资源的功能。内核通过硬件抽象的方法屏蔽了硬件的复杂性和多样性，向应用程序提供了统一、简洁的接口，降低了应用程序设计复杂程度
- 内核可以被看做是一个系统资源管理器，内核管理计算机系统中所有的软件和硬件资源



Linux内核（3）

- 实际上，应用程序可直接运行在计算机硬件上，从这个角度看，内核不是必要的。早期的计算机系统由于系统资源受限，可以直接在硬件上运行应用程序。但还需要一些辅助程序，如程序加载器、调试器等
- 随着计算机性能的不断提高，硬件和软件源都变得复杂，需要一个统一管理的程序，操作系统的概念也逐渐建立起来

本实验基于Linux-0.11内核版本

- 1991年5月发布的Linux首个版本0.01不支持网络，仅在Intel80386处理器的PC机上运行，对设备驱动器的支持很有限，仅支持Minix文件系统
- Linux-0.11内核源代码只有一万四千行左右，所涵盖的内容基本上都是Linux系统的精髓，很适合linux系统的初学者

tar命令解压linux-0.11.tar.gz后结构如下

```
linux
├── boot            系统引导汇编程序
├── fs              文件系统
├── include         头文件(*.h)
│   ├── asm         与 CPU 体系结构相关的部分
│   ├── linux       Linux 内核专用部分
│   └── sys         系统数据结构部分
├── init            内核初始化程序
├── kernel          内核进程调度、信号处理、系统调用等程序
│   ├── blk_drv     块设备驱动程序
│   ├── chr_drv     字符设备驱动程序
│   └── math        数学协处理器仿真处理程序
├── lib             内核库函数
├── mm              内存管理程序
└── tools           生成内核 Image 文件的工具程序
```

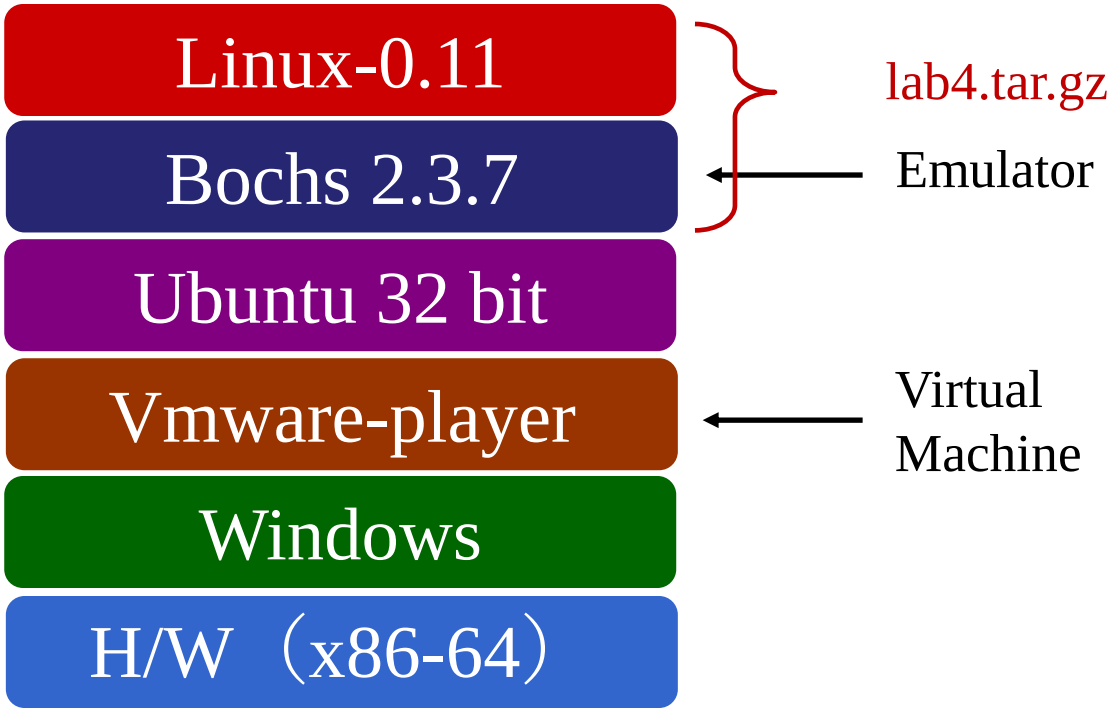
Linux内核源代码目录结构

搭建运行Linux-0.11环境

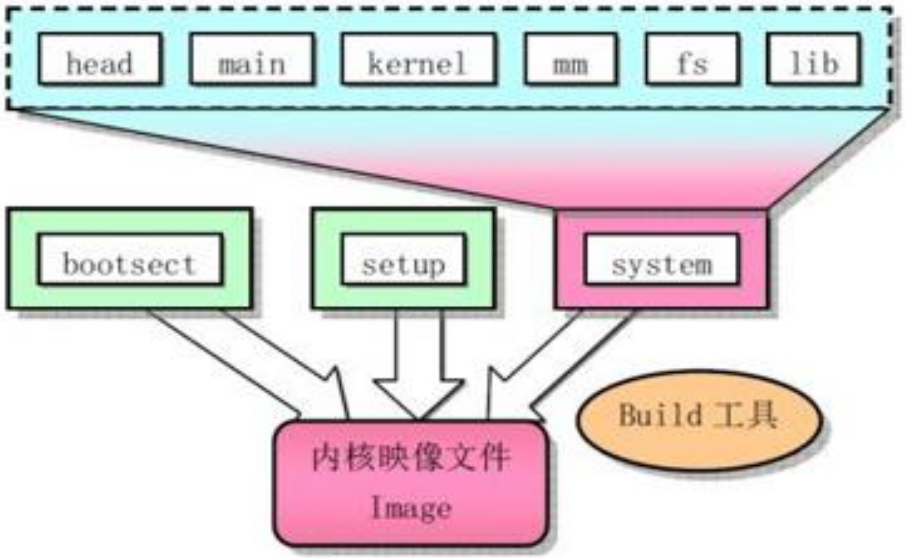
- 版本古老，1991年12月发布
- 32位操作系统，运行在Intel80386处理器PC机上
- 现实：普遍使用64位Windows系统

Linus Torvalds
赫尔辛基大学
大二学生

Bochs is a portable IA-32 and x86-64 IBM PC compatible emulator and debugger mostly written in C++. It supports emulation of the processor(s) (including protected mode), memory, disks, display, Ethernet, BIOS and common hardware peripherals of PCs.



内核映像文件的生成



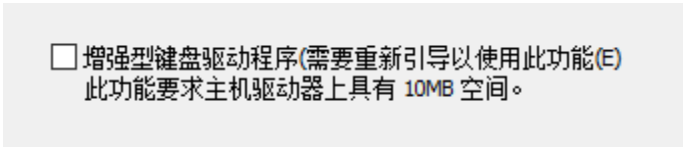
- 在**Bochs**中运行**OS**，需要：
  - Bochs执行文件
  - bios映象文件
  - VGA bios映象文件
  - 一个可引导运行的内核映象文件image
- 内核映像文件的生成方法
  - 对boot/中的bootsect.s、setup.s使用8086汇编器进行编译，分别生成各自的执行模块
  - 对源代码中的其他所有程序使用GNU的编译器gcc/gas进行编译，并链接生成可执行模块system
  - 使用build工具将上述3个模块组合成一个可运行的内核映像文件image

虚拟机环境安装

详细步骤见教学云平台“实验内容及说明”中“虚拟机环境搭建”

一、安装VMware Workstation 15 Player

1、不勾选



2、提示软件更新：安装 VMware Tools for Linux

二、安装32位Linux-Ubuntu 16.04.1

三、启动32位Linux-Ubuntu 16.04.1

1、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
… …
```

2、安装as86

```
sudo apt install bin86
```

注：另一种安装方法，可直接用gcc5-8版本生成Linux-0.11的内核映像

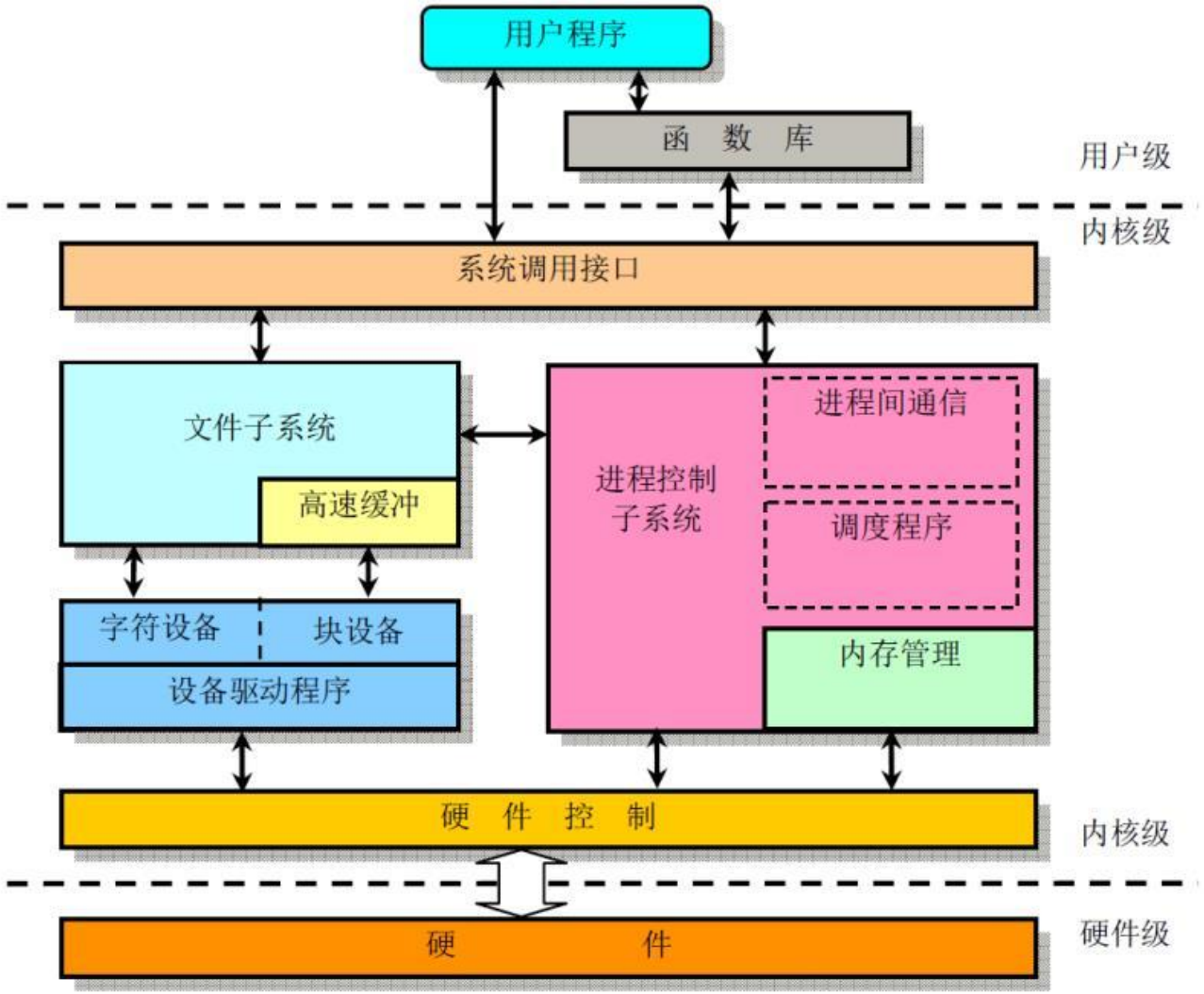
https://github.com/name1e5s/linux-0.11

编译并运行Linux-0.11

- 解压lab4.tar
- 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
- 进入lab4 目录，执行./run ，启动Linux，按键观察效果

- 注：./run init 可把修改文件回复初始状态。需重新编译生成Image文件

内核系统与用户程序关系



内核空间与用户空间

- **Linux**的两种运行模式：
  - 内核模式
  - 用户模式
- 内核模式对应内核空间，而用户模式对应用户空间。
- 驱动程序是内核的一部分，它对应内核空间，应用程序不能直接调用
- 区分用户态和内核态目的在于安全考虑
  - 禁止用户程序和底层硬件直接打交道。例如，如果用户程序往硬件控制寄存器写入不恰当的值，可能导致硬件无法正常工作
  - 禁止用户程序访问任意的物理内存。否则可能会破坏其他程序的正常执行，如果对内核所在的地址空间写入数据的话，可能会导致系统崩溃

用户程序如何同设备打交道？

例如，用户需通过网卡发送数据

- 硬件被linux 内核隔离，只能通过内核控制
- 用户不能直接调用操作系统的函数：不可行，也不安全

- Linux提供的解决方法：系统调用

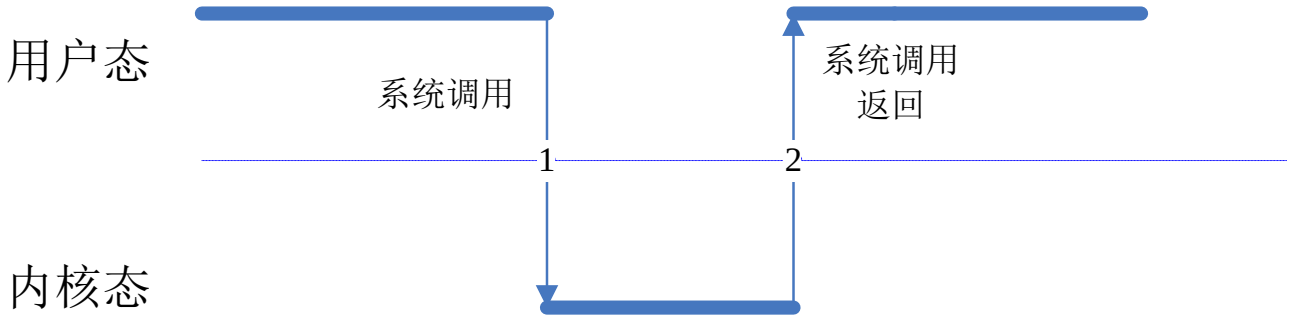
系统调用的意义

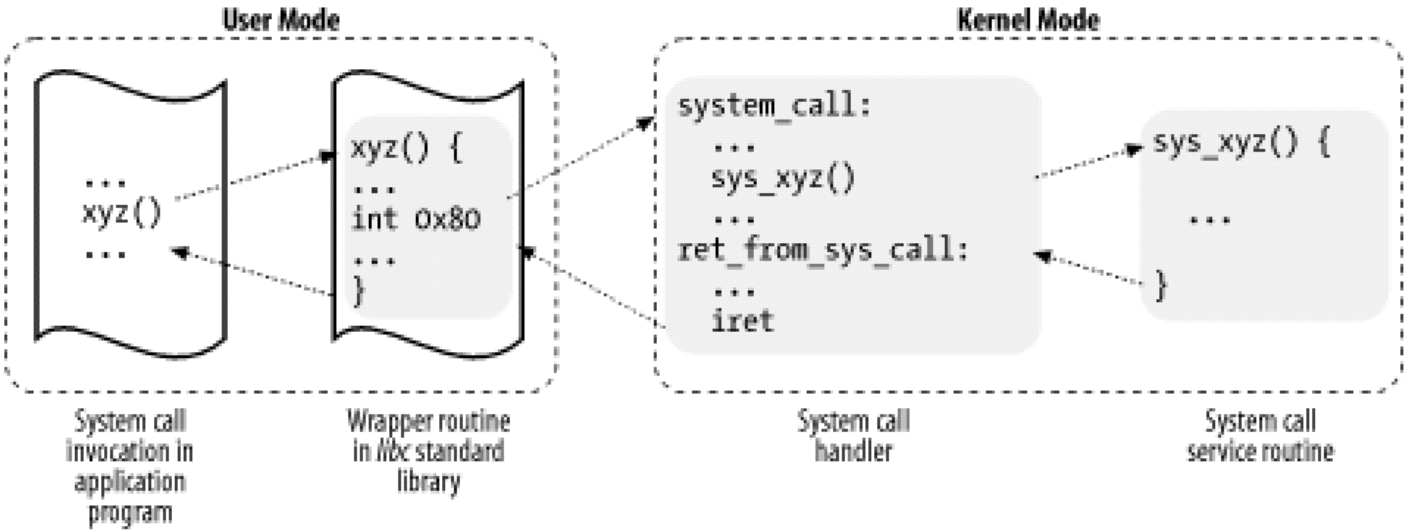
- 操作系统为用户态进程与硬件设备进行交互提供了一组接口—系统调用
  - 把用户从底层的硬件编程中解放出来
  - 极大的提高了系统的安全性
  - 使用户程序具有可移植性

- **Linux系统中，系统调用接口**

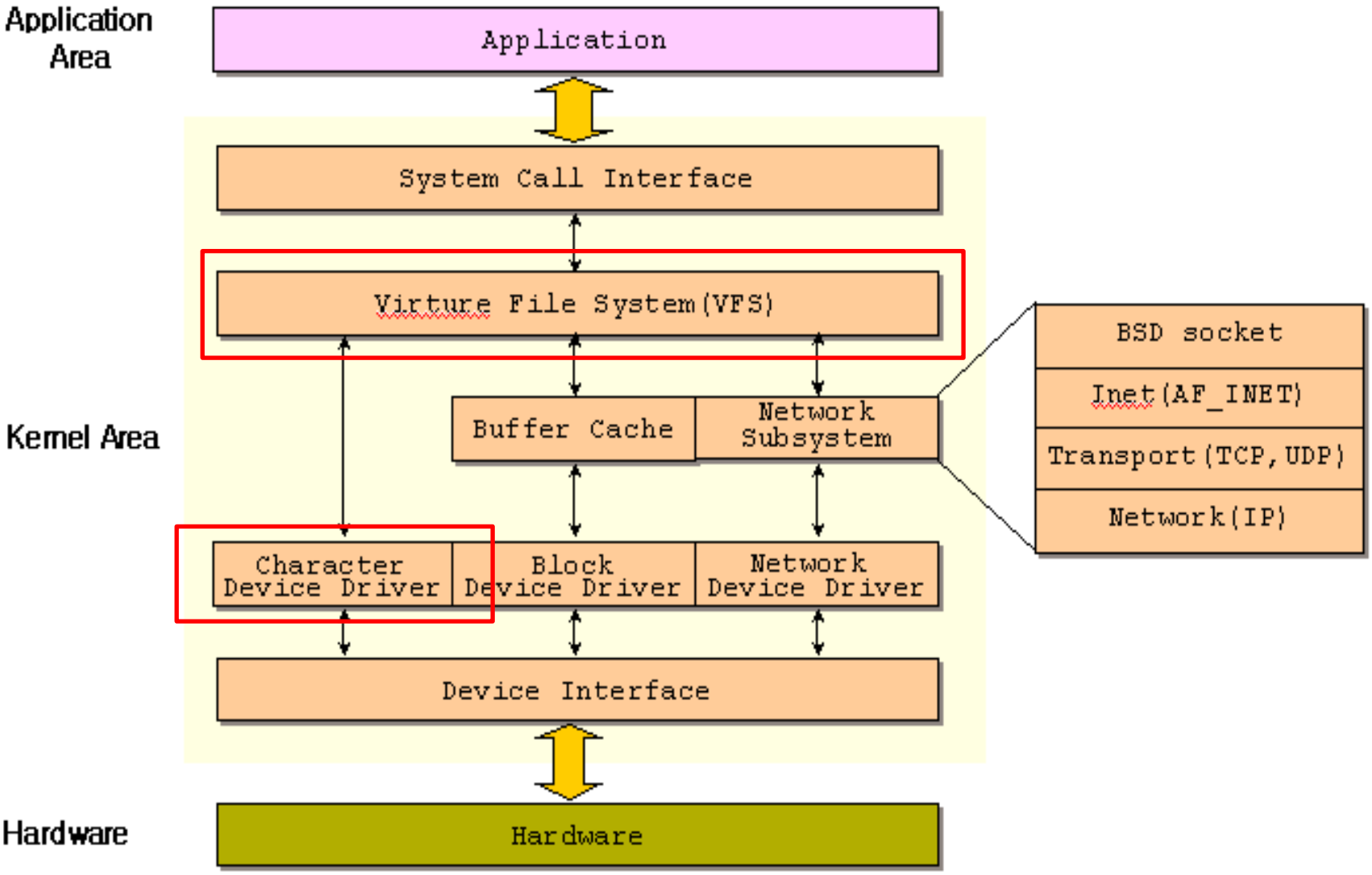
  int 0x80 或 syscall

系统调用图示


用户态
系统调用
系统调用
返回
1
2
内核态


User Mode
Kernel Mode
...
xyz()
...
xyz() {
...
int 0x80
...
}
system_call:
...
sys_xyz()
...
ret_from_sys_call:
...
iret
sys_xyz() {
...
}
System call invocation in application program
Wrapper routine in libc standard library
System call handler
System call service routine

内核中设备驱动的层次

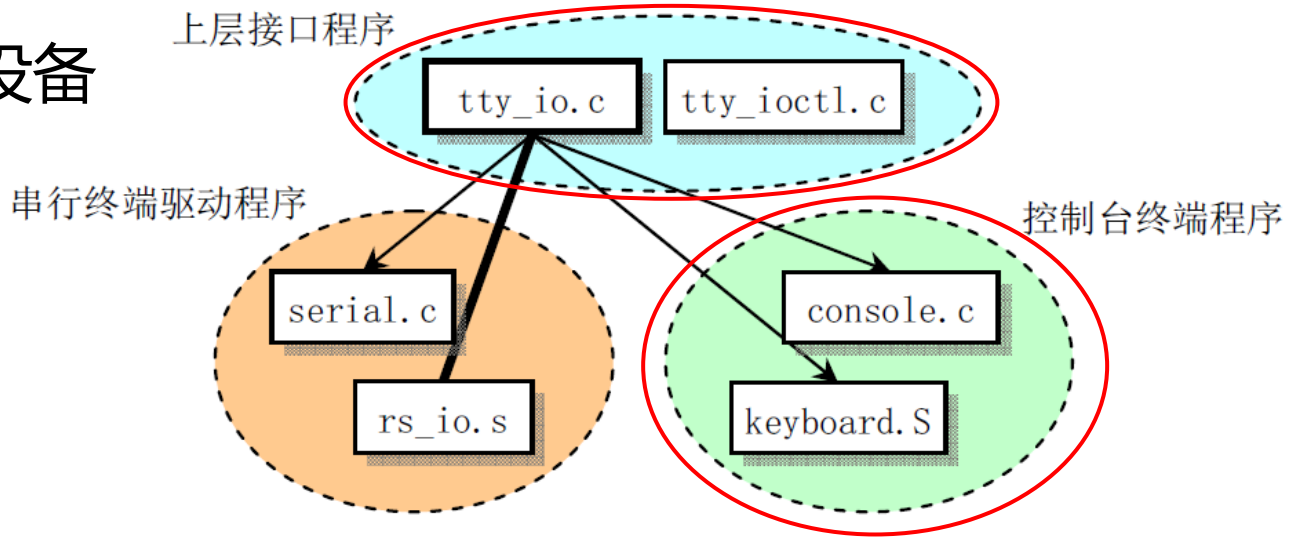

Application Area
Application
System Call Interface
Virture File System(VFS)
Kernel Area
Buffer Cache
Network Subsystem
BSD socket
Inet(AF_INET)
Transport(TCP,UDP)
Network(IP)
Character Device Driver
Block Device Driver
Network Device Driver
Device Interface
Hardware
Hardware

虚拟文件系统 Virtual File System

- VFS是一个软件层，是用户应用程序与具体文件系统实现之间的抽象层：
  - 对用户界面：一组标准的、抽象的文件操作，以系统调用提供，如read()、write()、open()等。
  - 对具体文件系统界面：主体是file_operations结构，全是函数指针，提供函数跳转表。

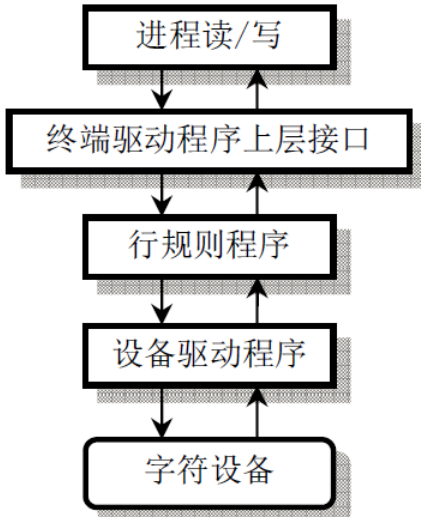
Linux-0.11 字符设备驱动程序

- 仅支持3个终端设备
  - 控制台终端
  - 串行终端1
  - 串行终端2



列表 10-1 linux/kernel/chr_drv 目录

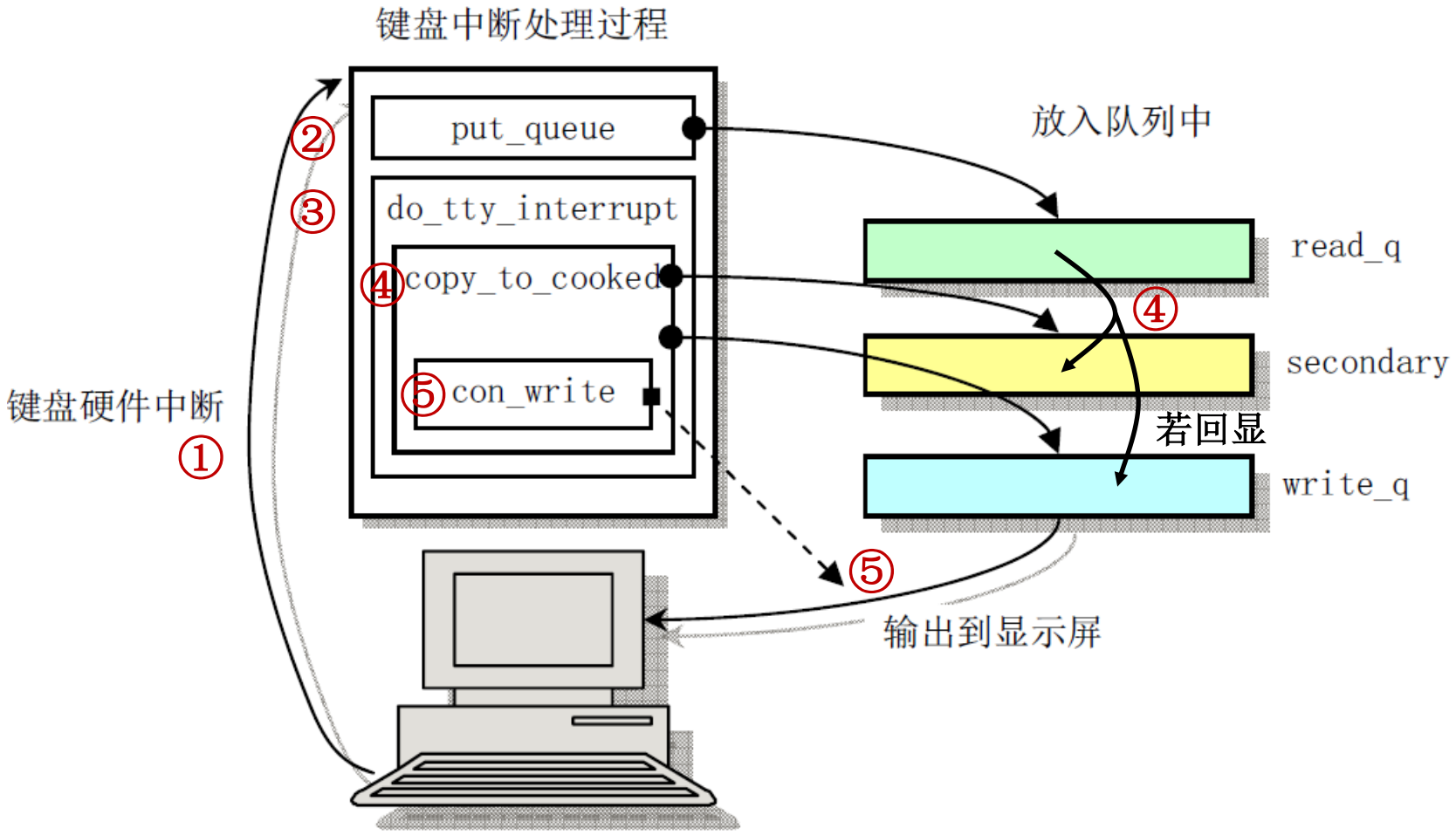
文件名	大小	最后修改时间(GMT)	说明
Makefile	2443 bytes	1991-12-02 03:21:41	Make 配置文件
console.c	14568 bytes	1991-11-23 18:41:21	控制台处理
keyboard.S	12780 bytes	1991-12-04 15:07:58	键盘中断处理
rs_io.s	2718 bytes	1991-10-02 14:16:30	串行中断处理
serial.c	1406 bytes	1991-11-17 21:49:05	串行初始化
tty_io.c	7634 bytes	1991-12-08 18:09:15	终端 IO 处理
tty_ioctl.c	4979 bytes	1991-11-25 19:59:38	终端 IO 控制

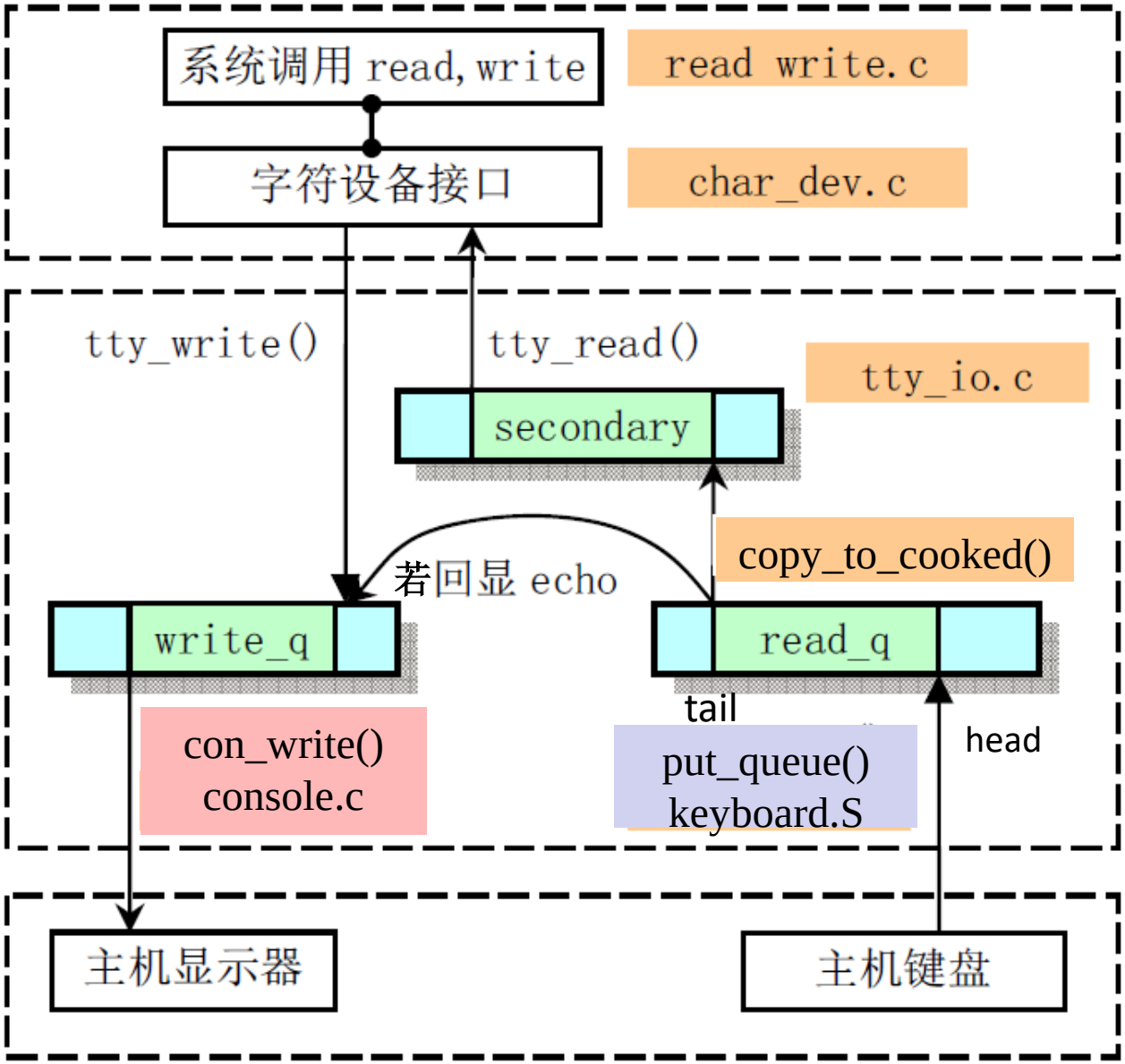


程序功能

- tty_io.c 程序中包含tty字符设备读函数tty_read()函数和写函数tty_write(),为文件系统提供了上层访问接口
- console.c文件主要包含控制台初始化程序和控制台写函数con_write()
- rs_io.s 汇编程序用于实现两个串行接口的中断处理程序
- serial.c 用于对异步串行通信芯片UART进行初始化操作
- keyboard.S程序主要实现了键盘中断处理过程（异步异常）keyboard_interrupt
- tty_ioctl.c程序实现了tty的IO控制接口函数tty_ioctl()

控制台键盘中断处理程序





文件系统中

字符设备驱动程序

注：函数 tty_write()、tty_read()和 copy_to_cooked() 均在 tty_io.c 中

控制台终端设备

终端基本数据结构

```
struct tty_struct {
        struct termios termios;                 // 终端 io 属性和控制字符数据结构。
        int pgrp;                               // 所属进程组。
        int stopped;                            // 停止标志。
        void (*write)(struct tty_struct * tty); // tty 写函数指针。
        struct tty_queue read_q;                // tty 读队列。
        struct tty_queue write_q;               // tty 写队列。
        struct tty_queue secondary;             // tty 辅助队列(存放规范模式字符序列),
};                                              // 可称为规范(熟)模式队列。
extern struct tty_struct tty_table[];           // tty 结构数组。
```

```
struct tty_queue {
        unsigned long data;                     // 等待队列缓冲区中当前数据统计值。
                                                // 对于串口终端，则存放串口端口地址。
        unsigned long head;                     // 缓冲区中数据头指针。
        unsigned long tail;                     // 缓冲区中数据尾指针。
        struct task_struct * proc_list;         // 等待本缓冲队列的进程列表。
        char buf[1024];                         // 队列的缓冲区。
};
```

XT键盘扫描码表

- 键盘上每一个键都有一个位置编号，称为键盘扫描码，从左到右，从上到下
- 例如，键‘2’的扫描码为03，键‘A’的扫描码为0x1E

F1	F2	`	1	2	3	4	5	6	7	8	9	0	-	=	\	BS	ESC	NUML	SCRL	SYSR
3B	3C	29	02	03	04	05	06	07	08	09	0A	0B	0C	0D	2B	0E	01	45	46	**
F3	F4	TAB	Q	W	E	R	T	Y	U	I	O	P	[	]			Home	↑	PgUp	PrtSc
3D	3E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B			47	48	49	37
F5	F6	CNTL	A	S	D	F	G	H	J	K	L	;	’		ENTER		←	5	→	-
3F	40	1D	1E	1F	20	21	22	23	24	25	26	27	28		1C		4B	4C	4D	4A
F7	F8	LSHFT		Z	X	C	V	B	N	M	,	.	/		RSHFT		End	↓	PgDn	+
41	42	2A		2C	2D	2E	2F	30	31	32	33	34	35		36		4F	50	51	4E
F9	F10	ALT		Space											CAPLOCK		Ins		Del	
43	44	38		39											3A		52		53	

修改键盘驱动程序DEMO

- 按F12健

  激活键盘功能：将键入的英文字母在屏幕上均显示为*
- 再按F12

  键盘功能恢复正常

代码修改（1）

1、增加全局变量f12Flag，每按一次F12，将该变量值翻转(0/1)

- 阅读/kernel/chr_drv/keyboard.S源代码可知key_table是扫描码到对应按键处理程序的转跳表，分析得知F1~F12的扫描码用函数func()处理

```
.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 ? ? ? */
```

p443

代码修改（2）

- 进一步分析/kernel/chr_drv/keyboard.S/函数func()

```
        cmpb $9,%al             // 功能键是 F1-F10?
        jbe ok_func             // 是，则跳转。
        subb $18,%al            // 是功能键 F11，F12 吗？F11、F12 扫描码是 0x57、0x58。
        cmpb $10,%al            // 是功能键 F11?
        jb end_func             // 不是，则不处理，返回。
        cmpb $11,%al            // 是功能键 F12?
        ja end_func             // 不是，则不处理，返回。
ok_func:
        cmpl $4,%ecx            /* check that there is enough room */ /*检查空间*/
        jl end_func             // [??]需要放入 4 个字符序列，如果放不下，则返回。
```

插入

```
jnz ok_func
call change_f12Flag
```

p437

代码修改（3）

- 在/kernel/chr_drv/console.c中增加函数change_f12Flag

```
int f12Flag=0;

void change_f12Flag(void){
    switch(f12Flag){
        case 1:
            f12Flag=0;
            break;
        case 0:
            f12Flag=1;
            break;
    }
}
```

- 每按F12键，该函数被调用

代码修改（4）

2、f12Flag置位时，将英文字母显示为*

- 分析/kernel/chr_drv/console.c中函数con_write（）

```
// 控制台写函数。
从终端对应的 tty 写缓冲队列中取字符，针对每个字符进行分析。若是控制字符或转义或控制
序列，则进行光标定位、字符删除等的控制处理；对于普通字符就直接在光标处显示。
参数 tty 是当前控制台使用的 tty 结构指针。
```

增加

```
if(f12Flag && ((c>64&&c<91)||(c>96&&c<123)))
    c='*';
```

```
// 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。
__asm__("movb _attr,%%ah\n\t"
        "movw %%ax,%1\n\t"
        ::"a" (c),"m" (*(short *)pos)
        :"ax");
pos += 2;
```

- 字符c在写到内存pos处后就显示在屏幕上，更改为*即可实现

p463

修改后运行Linux-0.11

- 修改源文件keyboard.S，console.c，再次编译
- 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
- 然后进入lab4目录执行./run，启动Linux，观察修改驱动程序后按键效果

注：./run init 可把修改文件回复初始状态。必需重新编译生成Image文件

实验内容

- **Phase 1**

  键入**F12**，激活*功能，键入学生姓名拼音，首尾字母等显示*

  比如：**zhangsan**，显示为：***ha*gsa***

- **Phase 2**

  键入“学号”：激活*功能，键入学生姓名，首尾字母等显示*

  比如：**zhangsan**，显示为：***ha*gsa***，

  键入“学号-”：取消该功能

参考阅读

- 《Linux内核完全注释》 修正版 V3.0 赵炯 编著

  第十章 10.1（串口内容除外）、10.3.1、10.3.2、10.3.3.4、10.4.1、10.4.2