奔跑吧LINUX内核* 入门篇

第五章 简单的字符设备驱动

笨叔叔

目 录

设备驱动Overview

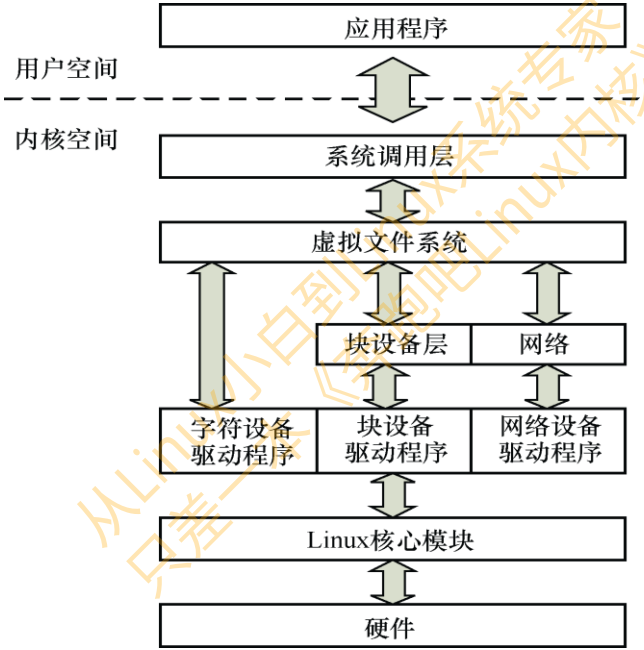
设备驱动概述

- 什么是设备驱动?
  - ✓ 设备驱动是操作系统和输入输出设备间的粘合剂。驱动负责将操作系统的请求传输，转化为特定物理设备控制器能够理解的命令。
  - ✓ 通俗来说，就是操作系统或者应用程序可以控制硬件设备

- Linux设备驱动分类
  - 字符设备以字节为单位的I/O传输，这种字符流的传输率通常比较低，常见的字符设备有鼠标、键盘、触摸屏等
  - 块设备是以块为单位传输的，常见的块设备是磁盘。
  - 网络设备是一类比较特殊的设备，涉及网络协议层。

设备驱动框架

- 什么是设备驱动?



写好设备驱动需要具备的技能

- 1）了解Linux内核字符设备驱动程序的架构。
- 2）了解Linux内核字符设备驱动相关的API。
- 3）了解Linux内核内存管理的API。
- 4）了解Linux内核中断管理的API。
- 5）了解Linux内核中同步和锁等相关的API。
- 6）了解你所要编写驱动的芯片原理。
- 7）了解Linux内核块设备驱动的框架（option）
- 8）了解Linux内核网络设备驱动的框架（option）

从简单的字符设备驱动入手

实验1：从一个简单的字符设备开始

- 实验目的
  - ✓ 1）编写一个简单的字符设备驱动，实现基本的open、read和write方法。
  - ✓ 2）编写相应的用户空间测试程序，要求测试程序调用read()函数，并能看到对应的驱动程序执行了相应的read方法。

- 实验步骤

字符驱动的init初始化函数

simple_char.c

```
51 static int __init simple_char_init(void)
52 {
53         int ret;
54
55         ret = alloc_chrdev_region(&dev, 0, count, DEMO_NAME);
56         if (ret) {
57                 printk("failed to allocate char device region");
58                 return ret;
59         }
60
61         demo_cdev = cdev_alloc();
62         if (!demo_cdev) {
63                 printk("cdev_alloc failed\n");
64                 goto unregister_chrdev;
65         }
66
67         cdev_init(demo_cdev, &demodrv_fops);
68
69         ret = cdev_add(demo_cdev, dev, count);
70         if (ret) {
71                 printk("cdev_add failed\n");
72                 goto cdev_fail;
73         }
74
75         printk("succeeded register char device: %s\n", DEMO_NAME);
76         printk("Major number = %d, minor number = %d\n",
77                         MAJOR(dev), MINOR(dev));
78
79         return 0;
80
```

NORMAL <le_driver simple_char.c

1:simple_char.c

字符驱动的file_operation方法集

```
42 static const struct file_operations demodrv_fops = {
43         .owner = THIS_MODULE,
44         .open = demodrv_open,
45         .release = demodrv_release,
46         .read = demodrv_read,
47         .write = demodrv_write
48 };
```

字符驱动的打开和关闭函数

```
12 static int demodrv_open(struct inode *inode, struct file *file)
13 {
14         int major = MAJOR(inode->i_rdev);
15         int minor = MINOR(inode->i_rdev);
16
17         printk("%s: major=%d, minor=%d\n", __func__, major, minor);
18
19         return 0;
20 }
21
22 static int demodrv_release(struct inode *inode, struct file *file)
23 {
24         return 0;
25 }
```

字符驱动的read和write函数

```
27 static ssize_t
28 demodrv_read(struct file *file, char __user *buf, size_t lbuf, loff_t *ppos)
29 {
30         printk("%s enter\n", __func__);
31         return 0;
32 }
33
34 static ssize_t
35 demodrv_write(struct file *file, const char __user *buf, size_t count, loff_t *f_pos)
36 {
37         printk("%s enter\n", __func__);
38         return 0;
39
40 }
```

test程序

```c
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

#define DEMO_DEV_NAME "/dev/demo_drv"

int main()
{
        char buffer[64];
        int fd;

        fd = open(DEMO_DEV_NAME, O_RDONLY);
        if (fd < 0) {
                printf("open device %s failded\n", DEMO_DEV_NAME);
                return -1;
        }

        read(fd, buffer, 64);
        close(fd);

        return 0;
}
```

字符设备驱动详解

字符设备驱动的抽象

- 字符设备驱动管理的核心对象是以字符为数据流的设备。从Linux内核设计的角度来看，需要有一个数据结构来对其进行抽象和描述，这就是struct cdev数据结构

```
12 struct cdev {
13         struct kobject kobj;
14         struct module *owner;
15         const struct file_operations *ops;
16         struct list_head list;
17         dev_t dev;
18         unsigned int count;
19 };
20
```

include/linux/cdev.h

- ✓ kobj：用于Linux设备驱动模型。
- ✓ owner：字符设备驱动程序所在的内核模块对象指针。
- ✓ ops：字符设备驱动程序中最关键的一个操作函数，在和应用程序交互过程中起到桥梁枢纽的作用。
- ✓ list：用来将字符设备串成一个链表。
- ✓ dev：字符设备的设备号，由主设备号和次设备号组成。
- ✓ count：同属一个主设备号的次设备号的个数。

设备号

- Linux设计哲学，一切皆文件
- 主设备号用来区分不同类型的设备
- 次设备号用来区分同一类型内的多个设备（及其设备分区）
- Linux的设备号用dev_t类型来表示，其实是u32类型

```
 6 #define MINORBITS       20
 7 #define MINORMASK       ((1U << MINORBITS) - 1)
 8
 9 #define MAJOR(dev)      ((unsigned int) ((dev) >> MINORBITS))
10 #define MINOR(dev)      ((unsigned int) ((dev) & MINORMASK))
11 #define MKDEV(ma,mi)    (((ma) << MINORBITS) | (mi))
```

include/linux/kdev_t.h

和设备号相关的有用的几个宏

设备号的分配

- 设备号是系统中珍贵的资源，内核必须避免发生两个设备驱动使用同一个主设备号的情况，因此在编写驱动程序时要格外小心
- 分配设备号的几个API：
  - int **register_chrdev_region**(dev_t from, unsigned count, const char *name)

需要指定主设备号，可以连续分配多个。也就是说，在使用该函数之前，驱动程序编写者必须保证要分配的主设备号在系统中没有被人使用。内核文档documentation/devices.txt文件描述了系统中已经分配出去的主设备号，因此使用该接口函数的程序员都应该事先约定该文档，避免使用已经被系统占用的主设备号。

  - int **alloc_chrdev_region**(dev_t *dev, unsigned baseminor, unsigned count, const char *name)

会自动分配一个主设备号，可以避免和系统占用的主设备号重复。建议驱动开发者使用这个接口函数来分配主设备号。

设备节点

- 万物皆文件。设备节点也算一个特殊的文件，称为设备文件，是连接内核空间驱动程序和用户空间应用程序的桥梁。
- 如果应用程序想使用驱动程序提供的服务或者操作设备，那么需要通过访问该设备文件来完成。
- 系统中所有的设备节点都存放在/dev/目录中。dev目录是一个动态生成的、使用devtmpfs虚拟文件系统挂载的、基于RAM的虚拟文件系统。

```
$ ls -l /dev/
total 0
crw-r--r--  1 root root     10, 235 May 12 05:25 autofs
drwxr-xr-x  2 root root         640 May  12 05:24 block
drwxr-xr-x  2 root root          60 May  12 05:24 bsg
crw-------  1 root root     10, 234 May 12 05:25 btrfs-control
drwxr-xr-x  3 root root          60 May  12 05:24 bus
drwxr-xr-x  2 root root        3960 May  12 05:25 char
crw-------  1 root root      5,    1 May  12 05:25 console
```

第一列中的c表示字符设备，d表示块设备。后面还会显示设备的主设备号和次设备号。

字符设备操作方法集

- 字符设备驱动程序的核心开发工作是实现file_operations方法集中的各类方法

应用程序的open()函数执行时，会通过系统调用进入内核空间，在内核空间的虚拟文件系统层（VFS）经过复杂的转换，最后会调用设备驱动的file_operations方法集中的open方法。

```
struct file_operations {
        struct module *owner;
        loff_t (*llseek) (struct file *, loff_t, int);
        ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
        ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
        ssize_t (*aio_read) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
        ssize_t (*aio_write) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
        ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);
        ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);
        int (*iterate) (struct file *, struct dir_context *);
        unsigned int (*poll) (struct file *, struct poll_table_struct *);
        long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
        long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
        int (*mmap) (struct file *, struct vm_area_struct *);
        int (*mremap)(struct file *, struct vm_area_struct *);
        int (*open) (struct inode *, struct file *);
        int (*flush) (struct file *, fl_owner_t id);
        int (*release) (struct inode *, struct file *);
        int (*fsync) (struct file *, loff_t, loff_t, int datasync);
        int (*aio_fsync) (struct kiocb *, int datasync);
        int (*fasync) (int, struct file *, int);
        int (*lock) (struct file *, int, struct file_lock *);
```

include/linux/fs.h

实验2：使用misc机制来创建设备

- 实验目的
  - ✓ 1）学会使用misc机制创建设备驱动
- 实验步骤

misc设备

- misc device称为杂项设备，Linux内核把一些不符合预先确定的字符设备划分为杂项设备，这类设备的主设备号是10。
- Linux内核使用struct miscdevice数据结构描述这类设备。

```
struct miscdevice  {
        int minor;
        const char *name;
        const struct file_operations *fops;
        struct list_head list;
        struct device *parent;
        struct device *this_device;
        const char *nodename;
        umode_t mode;
};
```

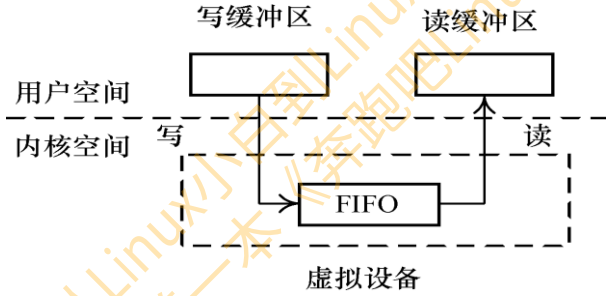
include/linux/miscdevice.h

- 内核提供了注册杂项设备的两个接口函数，驱动程序采用misc_register()函数来注册。它会自动创建设备节点，不需要使用mknod命令手工创建设备节点

```
extern int misc_register(struct miscdevice *misc);
extern int misc_deregister(struct miscdevice *misc);
```

FIFO虚拟设备

- 在实际项目中，一些字符的硬件设备内部有一个缓冲区（buffer），在一些外设芯片资料中称为FIFO。芯片内部提供了寄存器来访问这些FIFO，可以通过读寄存器把FIFO的内容读取出来，或者通过写寄存器把数据写入FIFO。为了提高效率，一般外设芯片支持中断模式，如FIFO有数据到达时，外设芯片通过中断线来告知CPU。



一个虚拟的FIFO设备

实验3：为虚拟设备编写驱动

- 实验目的
  - ✓ 1）通过一个虚拟设备，学习如何实现一个字符设备驱动程序的读写函数。
  - ✓ 2）在用户空间编写测试程序来检验读写函数是否成功。

- 实验步骤

实验4：使用KFIFO改进设备驱动

- 实验目的
  - ✓ 1）学会使用内核的KFIFO的环形缓冲区实现虚拟字符设备的读写函数。
- 实验步骤

KFIFO API的使用

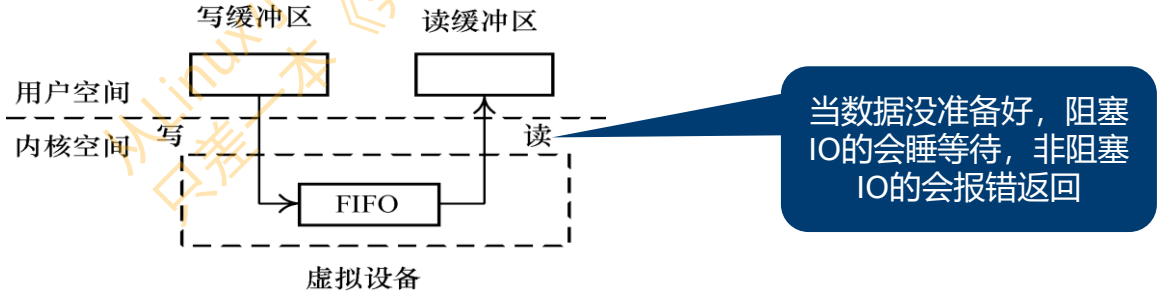
- 环形缓冲区
  - ✓ 一个典型的“生产者和消费者”的问题。环形缓冲区通常有一个读指针和一个写指针，读指针指向环形缓冲区可读的数据，写指针指向环形缓冲区可写的数据。通过移动读指针和写指针来实现缓冲区的数据读取和写入。
- KFIFO的环形缓冲区的机制
  - ✓ 它可以在一个读者线程和一个写者线程并发执行的场景下，无须使用额外的加锁来保证环形缓冲区的数据安全。KFIFO提供的接口函数定义在include/linux/kfifo.h文件中。

```
#define DEFINE_KFIFO(fifo, type, size)
#define kfifo_from_user(fifo, from, len, copied)
#define kfifo_to_user(fifo, to, len, copied)
```

- ✓ DEFINE_KFIFO()宏用来初始化一个环形缓冲区
- ✓ kfifo_from_user()宏用来将用户空间的数据写入环形缓冲区
- ✓ kfifo_to_user()宏用来读出环形缓冲区的数据并且复制到用户空间中

阻塞I/O和非阻塞I/O

- 一次IO的过程
  - ✓ 用户空间进程调用read()函数。
  - ✓ 通过系统调用进入驱动程序的read()函数。
  - ✓ 若缓冲区有数据，则把数据复制到用户空间的缓冲区中。
  - ✓ 若缓冲区没有数据，那么需要从设备中读取数据。硬件设备I/O是慢速设备，不知道什么时候能把数据准备好，因此进程需要睡眠等待。
  - ✓ 当硬件数据准备好时，唤醒正在等待数据的进程来取数据。



阻塞I/O和非阻塞I/O

- 非阻塞
  - ✓ 进程发起I/O系统调用后，如果设备驱动的缓冲区没有数据，那么进程返回一个错误而不会被阻塞。如果驱动缓冲区中有数据，那么设备驱动把数据直接返回给用户进程。
- 阻塞
  - ✓ 进程发起I/O系统调用后，如果设备的缓冲区没有数据，那么需要到硬件I/O中重新获取新数据，进程会被阻塞，也就是睡眠等待。直到数据准备好，进程才会被唤醒，并重新把数据返回给用户空间。

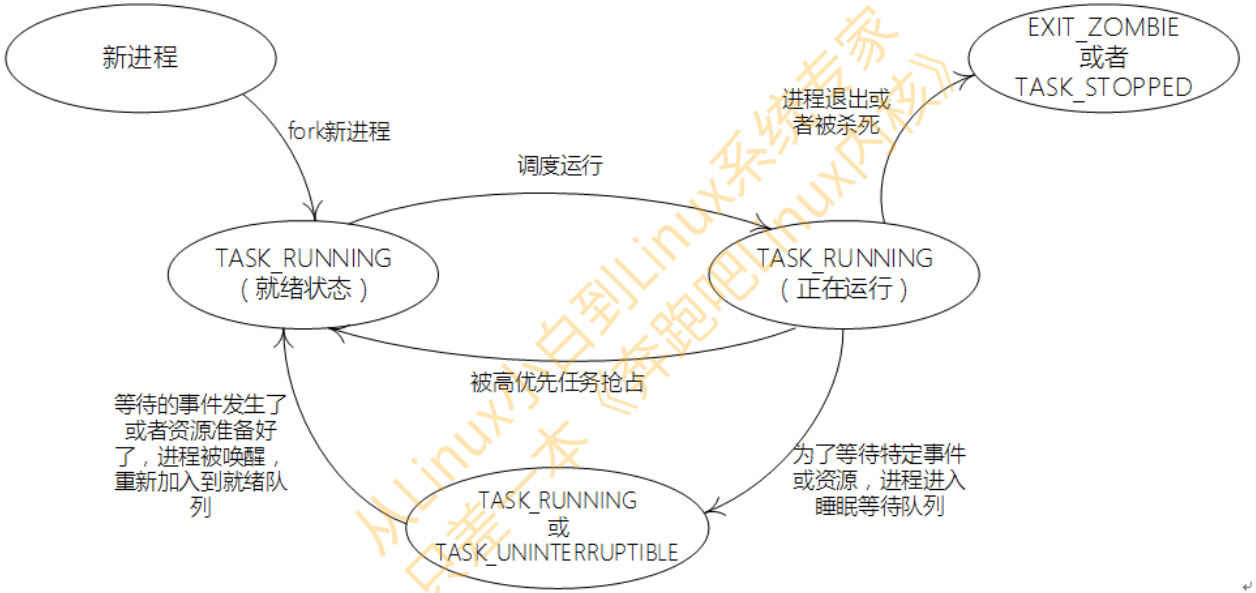
实验5：把虚拟设备驱动改成非阻塞模式

- 实验目的
  - ✓ 1）学习如何在字符设备驱动中添加非阻塞I/O操作。
- 实验详解
  - open()函数有一个flags参数，这些标志位通常用来表示文件打开的属性。
    - ✓ O_RDONLY：只读打开。
    - ✓ O_WRONLY：只写打开。
    - ✓ O_RDWR：读写打开。
    - ✓ O_CREAT：若文件不存在，则创建它。
    - ✓ O_NONBLOCK的标志位，用来设置访问文件的方式为非阻塞模式。

实验6：把虚拟设备驱动改成阻塞模式

- 实验目的
  - ✓ 1）学习如何在字符设备驱动中添加阻塞I/O操作。
- 实验详解

进程状态



等待队列（wait queue）的机制

- 等待队列
  - ✓ 需要睡眠等待的进程可以放到等待队列里
- 数据结构定义
  - ✓ 等待队列头wait_queue_head_t
  - ✓ 等待元素wait_queue_t
- API：
  - init_waitqueue_head(&my_queue);
  - wait_event(wq, condition)
  - wait_event_interruptible(wq, condition)
  - wait_event_timeout(wq, condition, timeout)
  - wait_event_interruptible_timeout()
  - wake_up(x)
  - wake_up_interruptible(x)

```
39 struct __wait_queue_head {
40         spinlock_t              lock;
41         struct list_head        task_list;
42 };
43 typedef struct __wait_queue_head wait_queue_head_t;
```

等待队列头的定义，include/ linux/wait.h

```
20 struct __wait_queue {
21         unsigned int            flags;
22         void                    *private;
23         wait_queue_func_t       func;
24         struct list_head        task_list;
25 };
26
```

等待队列头的元素，include/ linux/wait.h

等待队列（wait queue）的机制

- 初始化等待队列头
  - ✓ DECLARE_WAIT_QUEUE_HEAD(name)
  - ✓ init_waitqueue_head(&name);
- 睡眠等待
  - wait_event(wq, condition)
  - wait_event_interruptible(wq, condition)
  - wait_event_timeout(wq, condition, timeout)
  - wait_event_interruptible_timeout()
- 唤醒：
  - wake_up(x)
  - wake_up_interruptible(x)

实验7：向虚拟设备中添加I/O多路复用支持

- 实验目的
  - ✓ 1）对虚拟设备的字符驱动添加I/O多路复用的支持。
  - ✓ 2）编写应用程序对I/O多路复用进行测试。

- 实验详解

实验8：为什么不能唤醒读写进程

- 实验目的
  - ✓ 1）本实验是在5.6.2节实验7中故意设置的一个错误。希望读者通过发现问题和深入调试来解决问题，找到问题的根本原因，对字符设备驱动有一个深刻的认识。
- 实验详解

实验9：向虚拟设备中添加异步通知

- 实验目的
  - ✓ 1）学会如何给一个字符设备驱动程序添加异步通知功能。
- 实验详解

BACKUP

第1季旗舰篇课程目录

课程名称	时长
序言一： Linux内核学习方法论	0:09:13
序言二： 学习前准备	
序言2.1 Linux发行版和开发板的选择	0:13:56
序言2.2 搭建Qemu+gdb单步调试内核	0:13:51
序言2.3 搭建Eclipse图形化调试内核	0:10:59
实战运维1：查看系统内存信息的工具（一）	0:20:19
实战运维2：查看系统内存信息的工具（二）	0:16:32
实战运维3：读懂内核log中的内存管理信息	0:25:35
实战运维4：读懂 proc meminfo	0:27:59
实战运维5：Linux运维能力进阶线路图	0:09:40
实战运维6：Linux内存管理参数调优(一)	0:19:46
实战运维7：Linux内存管理参数调优(二)	0:31:20
实战运维8：Linux内存管理参数调优(三)	0:22:58
运维高级如何单步调试RHEL— CENTOS7的内核一	0:15:45
运维高级如何单步调试RHEL— CENTOS7的内核二	0:41:28
vim:打造比source insight更强更好用的IDE(一)	0:24:58
vim:打造比source insight更强更好用的IDE(二)	0:20:28
vim:打造比source insight更强更好用的IDE(三)	0:23:25
实战git项目和社区patch管理	
2.0 Linux内存管理背景知识介绍	
奔跑2.0.0 内存管理硬件知识	0:15:25
奔跑2.0.1 内存管理总览一	0:23:27
奔跑2.0.2 内存管理总览二	0:07:35
奔跑2.0.3 内存管理常用术语	0:09:49
奔跑2.0.4 内存管理究竟管些什么东西	0:28:02
奔跑2.0.5 内存管理代码框架导读	0:38:09
2.1 Linux内存初始化	
奔跑2.1.0 DDR简介	0:06:47
奔跑2.1.1 物理内存三大数据结构	0:19:39
奔跑2.1.2 物理内存初始化	0:11:13
奔跑2.1 内存初始化之代码导读一	0:43:54
奔跑2.1 内存初始化之代码导读二	0:23:31
2.2 页表的映射过程	
奔跑2.2.0 ARM32页表的映射	0:08:54
奔跑2.2.1 ARM64页表的映射	0:10:58
奔跑2.2.2 页表映射例子分析	0:11:59
奔跑2.2.3 ARM32页表映射那些奇葩的事	0:09:42
2.3 内存布局图	
奔跑2.3.1 内存布局一	0:10:35
奔跑2.3.2 内存布局二	0:13:30
2.4 分配物理页面	
奔跑2.4.1 伙伴系统原理	0:10:10
奔跑2.4.2 Linux内核中的伙伴系统和碎片化	0:11:14
奔跑2.4.3 Linux的页面分配器	0:21:37
2.5 slab分配器	
奔跑2.5.1 slab原理和核心数据结构	0:18:36
奔跑2.5.2 Linux内核中slab机制的实现	0:16:56
2.6 vmalloc分配	
奔跑2.6 vmalloc分配	0:15:48
2.7 VMA操作	
奔跑2.7 VMA操作	0:16:42
2.8 malloc分配器	
奔跑2.8.1 malloc的三个迷惑	0:17:41
奔跑2.8.2 内存管理的三个重要的函数	0:17:38
2.9 mmap分析	
奔跑2.9 mmap分析	0:23:14
2.10 缺页中断处理	
奔跑2.10.1 缺页中断一	0:31:07
奔跑2.10.2 缺页中断二	0:16:58
2.11 page数据结构	
奔跑2.11 page数据结构	0:29:41
2.12 反向映射机制	
奔跑2.12.1 反向映射机制的背景介绍	0:19:01
奔跑2.12.2 RMAP四部曲	0:07:31
奔跑2.12.3 手撕Linux2.6.11上的反向映射机制	0:07:35
奔跑2.12.4 手撕Linux4.x上的反向映射机制	0:10:08
2.13 回收页面	
奔跑2.13 页面回收一	0:16:07
奔跑2.13 页面回收二	0:11:41
2.14 匿名页面的生命周期	0:26:16
2.15 页面迁移	0:19:07
2.16 内存规整	0:24:03
2.17 KSM	0:28:17
2.20 Meltdown漏洞分析	
奔跑2.20.1 Meltdown背景知识	0:10:13
奔跑2.20.2 CPU体系结构之指令执行	0:11:25
奔跑2.20.3 CPU体系结构之乱序执行	0:11:03
奔跑2.20.4 CPU体系结构之异常处理	0:03:48
奔跑2.20.5 CPU体系结构之cache	0:10:56
奔跑2.20.6 进程地址空间和页表及TLB	0:17:39
奔跑2.20.7 Meltdown漏洞分析	0:06:04
奔跑2.20.8 Meltdown漏洞分析之x86篇	0:12:07
奔跑2.20.9 ARM64上的KPTI解决方案	0:25:39
代码导读	
奔跑2.1 内存初始化之代码导读一	0:43:54
奔跑2.1 内存初始化之代码导读二	0:23:31
奔跑2.1 代码导读C语言部分（一）	0:27:34
奔跑2.1 代码导读C语言部分（二）	0:21:28
代码导读3页表映射	1:12:40
代码导读4分配物理页面	0:55:57
git入门和实战	
git入门与实战：节目总览	0:08:48
git入门与实战1：建立本地的git仓库	0:30:53
git入门与实战2：快速入门	0:12:45
git入门与实战3：分支管理	0:24:27
git入门与实战4：冲突解决	0:20:20
git入门和实战5：提交更改	0:12:15
git入门和实战6：远程版本库	0:13:26
git入门和实战7：内核开发和实战	0:15:52
git入门和实战8：实战rebase到最新Linux内核代码	0:18:07
git入门和实战9：给内核发补丁	0:13:57

第2季旗舰篇课程目录	
课程名称	时长
进程管理	
进程管理1基本概念	0:52:16
进程管理2进程创建	0:53:24
进程管理3进程调度	0:54:51
进程管理4多核调度	0:49:38
中断管理	
中断管理1基本概念	1:04:27
中断管理2中断处理part1	0:46:28
中断管理2中断处理part2	0:10:19
中断管理3下半部机制	0:55:57
中断管理4面试题目	1:13:57
锁机制	
锁机制入门1基本概念	0:56:16
锁机制入门2-Linux常用的锁	0:54:01

实战死机专题课程目录	
课程名称	时长
上集x86_64	
实战死机专题（上集）part1-kdump+crash介绍	0:30:09
实战死机专题（上集）part2-crash命令详解	0:28:15
实战死机专题（上集）part3-实战lab1	0:12:38
实战死机专题（上集）part4-实战lab2	0:11:03
实战死机专题（上集）part4-实战lab3	0:06:48
实战死机专题（上集）part4-实战lab4	0:15:28
实战死机专题（上集）part4-实战lab5	0:12:21
实战死机专题（上集）part4-实战lab6	0:24:07
实战死机专题（上集）part4-实战lab7	0:59:34
下集arm64	
实战死机专题(下集)part1	0:13:19
实战死机专题(下集)part2	0:20:47
实战死机专题(下集)part3	0:11:22
实战死机专题(下集)part4	0:33:01

全程约5小时高清，140多页ppt，8大实验，基于x86_64的**Centos 7.6**和**arm64**，提供全套实验素材和环境。全面介绍kdump+crash在死机黑屏方面的实战应用，全部案例源自线上云服务器和嵌入式产品开发实际案例！

微店 奔跑吧Linux内核

扫码识别

微店二维码

淘宝店二维码

《奔跑吧Linux内核 * 入门篇》相关的免费视频，或者更多更精彩更in的内容，请关注奔跑吧Linux社区微信公众号