

《嵌入式系统》

（第七讲）

厦门大学信息学院软件工程系 曾文华

2023年10月24日

第7章 ARM-Linux内核

- 7.1 ARM-Linux内核简介
- 7.2 ARM-Linux内存管理
- 7.3 ARM-Linux进程管理和调度
- 7.4 ARM-Linux模块机制
- 7.5 ARM-Linux系统启动和初始化

7.1 ARM-Linux内核简介

- **内核**：是一个**操作系统的核心**。是基于硬件的第一层软件扩充，提供操作系统的最基本的功能，是操作系统工作的基础，它负责管理系统的进程、内存、设备驱动程序、文件和网络系统，决定着系统的性能和稳定性。现代操作系统设计中，为减少系统本身的开销，往往将一些**与硬件紧密相关的**（如中断处理程序、设备驱动程序等）、基本的、公共的、运行频率较高的模块（如时钟管理、进程调度等）以及**关键性数据结构**独立开来，使之常驻内存，并对他们进行保护。通常把这一部分称之为操作系统的内核。
- **Linux内核**：Linux内核的主要模块（或组件）分以下几个部分：存储管理、CPU和进程管理、文件系统、设备管理和驱动、网络通信，以及系统的初始化（引导）、系统调用等。
- **ARM-Linux内核**：基于ARM处理器的Linux内核。

• 7.1.1 ARM-Linux内核和普通Linux内核的区别

- 相对于ARM Linux，我们说的**普通Linux**指的是**x86 Linux**，它们都是Linux系统，但是由于ARM和x86是不同的CPU架构，它们的指令集不同，所以软件编译环境不同，软件代码一般不能互用，一般需要进行兼容性移植。
- **x86**是经典的**CISC指令集**，指令集复杂，功能多，串行执行，但是也意味着执行效率低下，但性价比突出，所以称为民用终端的主流处理器内置指令集。Intel和AMD的家用户理器都是x86指令集。以x86为代表的CISC，理论并发线程1-2条。
- **ARM**是Advanced RISC Machine 的缩写。它的指令集比**RISC**还要精简。通常使用ARM架构处理器的机型，多为嵌入式或者便携机。主频通常不高，现在高通公司的ARM架构处理器有1.0GHz的，已经算相当高了。另外，ARM 7沿用冯·诺依曼结构；而从ARM 9以后，就都采用了哈佛结构。ARM的并发线程，理论上有4条左右，处理效率较X86高不少。

• 7.1.2 ARM-Linux的版本控制

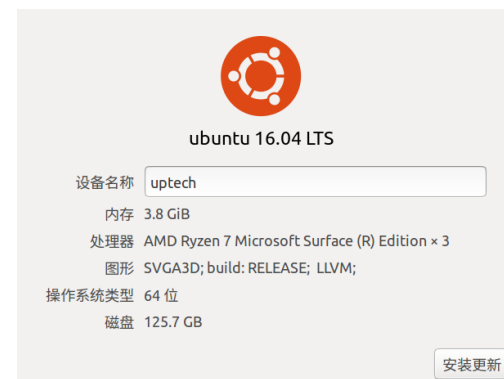
– Linux的版本号

- 主版本号：序号的第1位
- 次版本号：序号的第2位
- 修订号：序号的第3位
- 稳定版：序号的第2位（次版本号）为偶数
- 测试版：序号的第2位（次版本号）为奇数

– 查看Linux系统的版本号（Ubuntu）

- root@uptech-virtual-machine:/home/uptech# **cat /etc/issue**
 - Ubuntu **16.04.6 LTS** \n \l
- root@uptech-virtual-machine:/home/uptech# **cat /etc/os-release**
 - NAME="Ubuntu"
 - VERSION="**16.04.6 LTS** (Xenial Xerus)"
 - ID=ubuntu
 - ID_LIKE=debian
 - PRETTY_NAME="Ubuntu **16.04.6 LTS**"
 - VERSION_ID="**16.04**"
 - HOME_URL="http://www.ubuntu.com/"
 - SUPPORT_URL="http://help.ubuntu.com/"
 - BUG_REPORT_URL="http://bugs.launchpad.net/ubuntu/"
 - VERSION_CODENAME=xenial
 - UBUNTU_CODENAME=xenial

Ubuntu的版本号



– 查看Linux内核的版本号（Ubuntu）

Ubuntu Linux内核的版本号

- root@uptech-virtual-machine:/home/uptech# **cat /proc/version**
 - Linux version **4.15.0-142**-generic (buldd@lgw01-amd64-039) (gcc version 5.4.0 20160609 (Ubuntu 5.4.0-6ubuntu1~16.04.12)) #146~16.04.1-Ubuntu SMP Tue Apr 13 09:27:15 UTC 2021
- root@uptech-virtual-machine:/home/uptech# **uname -a**
 - Linux uptech **4.15.0-142**-generic #146~16.04.1-Ubuntu SMP Tue Apr 13 09:27:15 UTC 2021 x86_64 x86_64 x86_64 GNU/Linux
- root@uptech-virtual-machine:/home/uptech# **uname -r**
 - **4.15.0-142**-generic

– 查看Linux内核的版本号（实验箱）



```
imx6dlsabresd login: root
Password:
root@imx6dlsabresd:~# cat /proc/version
Linux version 4.9.88-1.0.0_ga+g91cf351 (uptech@uptech) (gcc version 4.9.1 (GCC) ) #180 SMP PREEMPT Mon Mar 25 15:56:23 CST 2019
root@imx6dlsabresd:~#
```

实验箱Linux内核的版本号

Linux内核的版本号（实验箱）

.config - Linux/arm 4.9.88 Kernel Configuration

Linux/arm 4.9.88 Kernel Configuration

Arrow keys navigate the menu. <Enter> selects submenus ---> (or empty submenus ----). Highlighted letters are hotkeys. Pressing <Y> includes, <N> excludes, <M> modularizes features. Press <Esc><Esc> to exit, <?> for Help, </> for Search. Legend: [*] built-in []

^(-)
Boot options --->
CPU Power Management --->
Floating point emulation --->
Userspace binary formats --->
Power management options --->
[*] Networking support --->
Device Drivers --->
Firmware Drivers --->
File systems --->
Kernel hacking --->

⌞(+)

<Select>

< Exit >

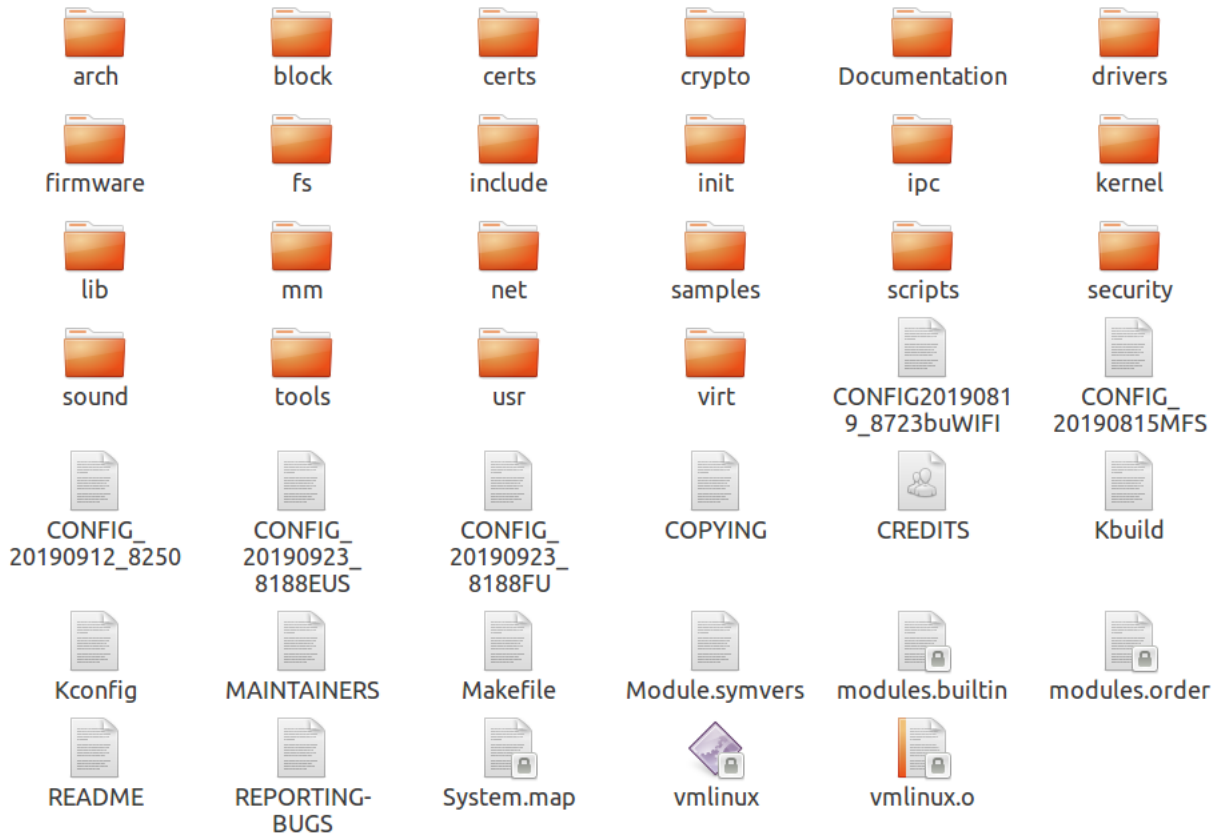
< Help >

< Save >

< Load >

• 7.1.3 ARM-Linux的代码结构

– 位于Ubuntu的/home/uptech/fsl-6dl-source/**kernel-4.9.88**/目录下



1. **arch** : 包含和硬件体系结构相关的代码, 每种平台 (处理器) 占一个相应的目录, 如**i386**、**arm**、**arm64**、**powerpc**、**mips** 等
2. **block**: 块设备驱动程序
3. **certs**: 存储认证和签名相关代码
4. **crypto**: 常用加密和散列算法 (如**AES**、**SHA** 等), 还有一些压缩和**CRC** 校验算法
5. **Documentation**: 内核各部分的通用解释和注释 (文档)
6. **drivers** : 设备驱动程序, 每个不同的驱动占用一个子目录, 如**char**、**block**、**net**、**mtd**、**i2c** 等
7. **firmware**: 固件, 包含了让计算机读取和理解从设备发来的信号的代码
8. **fs**: 所支持的各种文件系统, 如**EXT**、**FAT**、**NTFS**、**JFFS2** 等
9. **include**: 头文件, 与系统相关的头文件放置在**include/linux** 子目录下
10. **init**: 内核初始化代码, 著名的**start_kernel()** 就位于**init/main.c** 文件中
11. **ipc**: 进程间通信的代码

- 12. **kernel**：内核最核心的部分，包括进程调度、定时器等，而和平台（处理器）相关的一部分代码放在`arch/*/kernel`目录下
- 13. **lib**：库文件代码
- 14. **mm**：内存管理代码，和平台（处理器）相关的一部分代码放在`arch/*/mm`目录下
- 15. **net**：网络相关代码，实现各种常见的网络协议
- 16. **samples**：一些内核编程的范例
- 17. **scripts**：用于配置内核的脚本文件
- 18. **security**：Linux安全模型的代码
- 19. **sound**：ALSA、OSS 音频设备的驱动核心代码和常用设备驱动
- 20. **tools**：这个文件夹中包含了和内核交互的工具
- 21. **usr**：实现用于打包和压缩的`cpio` 等
- 22. **virt**：此文件夹包含了虚拟化代码，它允许用户一次运行多个操作系统

23. **Kbuild**: 这是一个设置一些内核设定的脚本

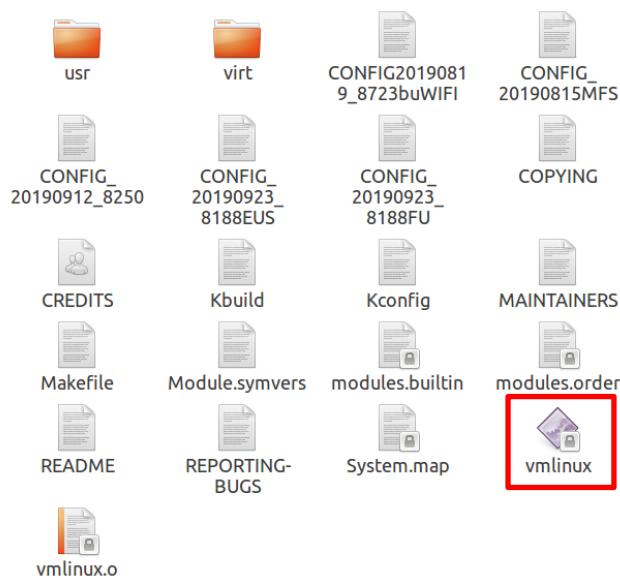
24. **Kconfig**: 内核配置选项文件

25. **Makefile**: 这个脚本是编译内核的主要文件，这个文件将编译参数和编译所需的文件和必要的信息传给编译器

26. **System.map**: 该文件可以帮助我们理解内核编译，它记录了所有代码的运行地址

27. **vmlinux**: 是可引导的、压缩的Linux内核

28. **vmlinux.o**: 是vmlinux的目标文件



arch

`/home/uptech/fsl-6dl-source/kernel-4.9.88/arch/`



alpha



arc



arm



arm64



avr32



blackfin



c6x



cris



frv



hexagon



ia64



m32r



m68k



metag



microblaze



mips



mn10300



openrisc



parisc



powerpc



s390



score



sh



sparc



tile



um



uncore32



x86



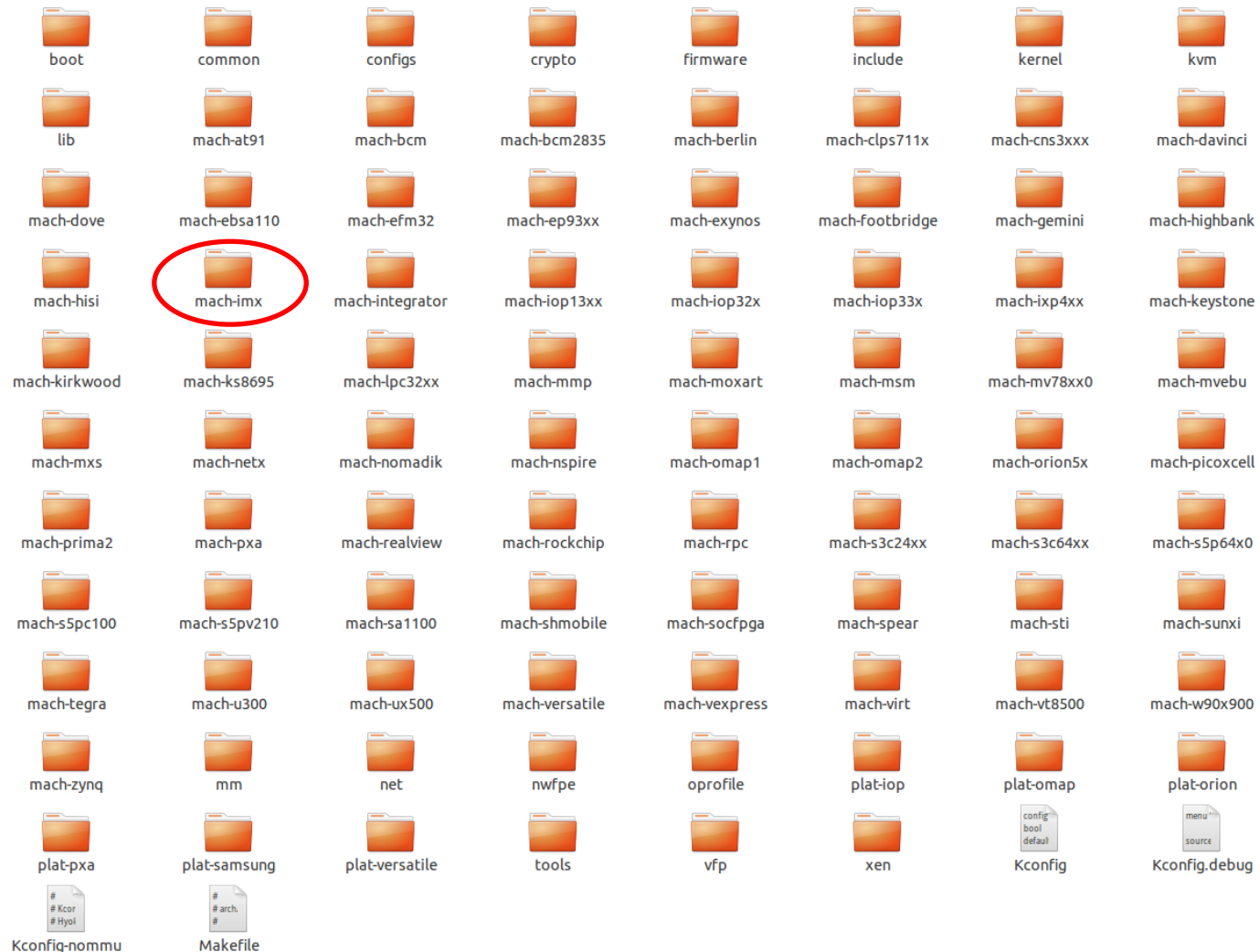
xtensa



Kconfig

arm

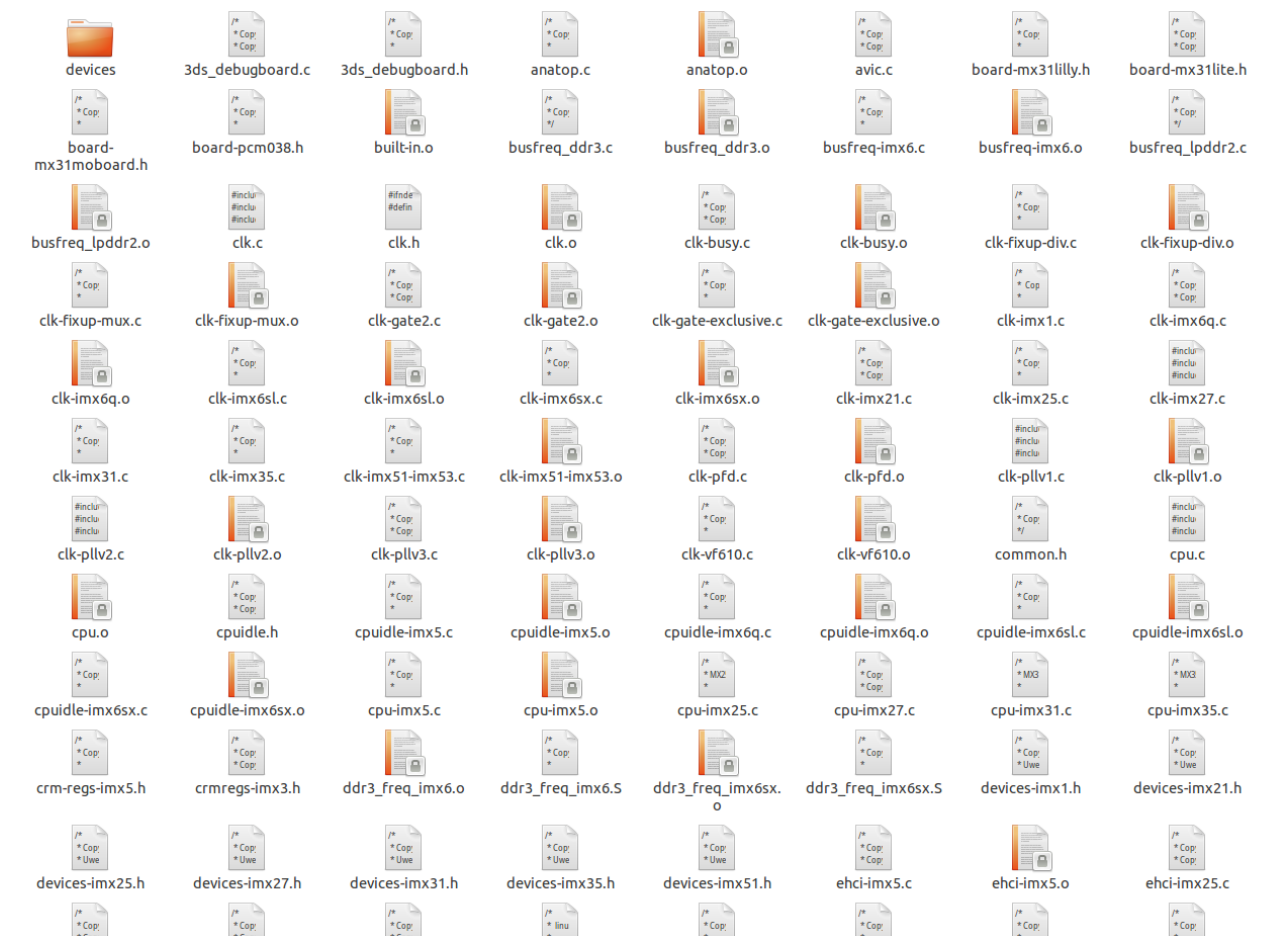
/home/uptech/fsl-6dl-source/kernel-4.9.88/arch/arm/



mach-imx

主CPU：采用飞思卡尔（Freescale）公司生产的基于ARM Cortex-A9 MPCore 的最新单核的IMX6DL 嵌入式微处理器（Freescale **IMX6DL**）

/home/uptech/fsl-6dl-source/kernel-4.9.88/arch/arm/mach-imx/



7.2 ARM-Linux内存管理

第3章 嵌入式Linux操作系统

3.2 内存管理

3.2.1 内存管理和MMU

3.2.2 标准Linux的内存管理

3.2.3 μ CLinux的内存管理

• 7.1.1 影响内存管理的两个方面

– Linux内核对内存的管理（Linux操作系统的内存管理）

- 内存管理是操作系统必不可少也是非常重要的一部分，包括：

- ① 地址映射
- ② 内存空间的分配
- ③ 地址访问的限制（即保护机制）
- ④ I/O地址的映射（I/O编址与内存编址相同）

– ARM体系结构对内存的管理（MMU）

- MMU（存储器管理单元）的主要作用有两个方面：

- ① 地址映射
- ② 对地址访问进行保护和限制

- MMU可以做在CPU芯片中，也可以作为一个协处理器（用协处理器实现）

• 7.2.2 ARM-Linux的存储机制

– 基于x86体系结构的Linux内核的存储空间

- 32位地址形成4GB的虚拟地址空间，被分为两部分：
 - ① 内核空间（系统空间）：位于高端的1GB，属于Linux操作系统
 - ② 用户空间：位于低端的3GB，属于应用程序

0xffffffff	1GB	} 内核空间
0xc0000000		
0xbfffffff	1GB	} 用户空间
0x80000000		
0x7fffffff	1GB	
0x40000000		
0x3fffffff	1GB	
0x00000000		

– ARM-Linux内核的存储空间

- 32位地址形成4GB的虚拟地址空间，也被分为两部分，但是内核空间（系统空间）和用户空间的具体划分，可以因CPU芯片和开发板（实验箱）而有所不同
- 另外，ARM将I/O也放在内存地址空间中

• 7.2.3 虚拟内存

- **虚拟内存（虚拟存储器）**：是计算机系统内存管理的一种技术，它使得应用程序认为它拥有连续的可用的内存（一个连续完整的地址空间），而实际上，它通常是被分隔成多个物理内存碎片，还有部分暂时存储在外部磁盘存储器上，在需要进行数据交换。
- **Linux虚拟内存的实现需要6种机制的支持：**
 - ① 地址映射机制
 - ② 请求页机制
 - ③ 内存分配回收机制
 - ④ 缓存和刷新机制
 - ⑤ 交换机制
 - ⑥ 内存共享机制

7.3 ARM-Linux进程管理和调度

- **进程**：也称为**任务**，是一个动态的执行过程，是**处于执行期的程序**，进程是系统资源分配的最小单位。
- **狭义定义**：进程是正在运行的程序的实例。
- **广义定义**：进程是一个具有一定独立功能的程序关于某个数据集合的一次运行活动。它是操作系统动态执行的基本单元，在传统的操作系统中，进程既是基本的分配单元，也是基本的执行单元。
- 进程的概念主要有两点：
 - ① **进程是一个实体**。每一个进程都有它自己的地址空间，一般情况下，包括文本区域（text region）、数据区域（data region）和堆栈（stack region）。文本区域存储处理器执行的代码；数据区域存储变量和进程执行期间使用的动态分配的内存；堆栈区域存储着活动过程调用的指令和本地变量。
 - ② **进程是一个“执行中的程序”**。程序是一个没有生命的实体，只有处理器赋予程序生命时（操作系统执行之），它才能成为一个活动的实体，我们称其为进程。

第3章 嵌入式Linux操作系统

3.3 进程管理

3.3.1 进程和进程管理

3.3.2 RT-Linux的进程管理

3.3.3 标准Linux的进程管理

3.3.4 μ CLinux的进程管理

• 7.3.1 进程的表示和生命周期

– 进程描述符：用task_struct{}数据结构表示

– 进程的状态：

- ① TASK_RUNNING：可执行状态，进程要么正在执行，要么准备执行
- ② TASK_INTERRUPTIBLE：可中断的睡眠状态
- ③ TASK_UNINTERRUPTIBLE：不可中断的睡眠状态
- ④ TASK_STOPPED：暂停状态，进程停止执行
- ⑤ TASK_TRACED：跟踪状态，进程被追踪
- ⑥ EXIT_ZOMBIE：僵尸状态的进程，表示进程被终止
- ⑦ EXIT_DEAD：进程的最终状态，进程死亡
- ⑧ TASK_DEAD：死亡
- ⑨ TASK_WAKEKILL：唤醒并杀死的进程
- ⑩ TASK_WAKING：唤醒进程

– 进程标识符：PID，Process ID（进程ID）

• 7.3.2 Linux进程的创建、执行和销毁

– Linux进程的**创建**:

- 通过fork函数创建进程
- 创建用户空间进程: vfork()、fork()
- 创建内核空间进程: copy_process()、kernel_thread()、do_fork()、sys_vfork()、sys_fork()、sys_clone()

– Linux进程的**执行**:

- 通过exec函数执行进程
- exec函数族: execl()、execvp()、execle()、execv()、execvp()、execve()

– Linux进程的**销毁**:

- 通过do_exit函数结束进程

• 7.3.3 Linux进程的调度

- Linux是一个多进程系统。多进程就是指计算机同时执行多个进程，即同时运行多个程序。对于多进程系统，就存在多个进程如何进行调度的问题，包括进程调度时机和进程调度依据。
- 进程调度时机：
 - ① 主动调度：随时可以进行。
 - ② 被动调度：发生在系统调用返回的前夕、中断异常处理返回前、用户态处理软中断返回前。
 - ③ 抢占式内核：处于内核态的进程也可能被调度出去。
- 进程调度依据：
 - 在所有处于可运行状态的进程中，如何选择最值得运行的进程投入运行，以下4项是选择的依据：
 - ① policy：进程的调度策略。
 - ② priority：进程的静态优先级。
 - ③ counter：进程剩余的时间片（进程的动态优先级）。
 - ④ rt_priority：用于实时进程间的选择。
- 进程调度函数：schedule()函数
 - ① 主动调度：主动调用schedule()函数。
 - ② 被动调度：被动调用schedule()函数。

7.4 ARM-Linux模块机制

- Linux是**单内核**的，单内核最大的优点是效率高，因为所有的内容都集中在一起，单内核也有**可扩展性差、可维护性差**的缺点。
 - **单内核**，是个很大的进程。它的内部又能够被分为若干模块（或是层次或其他）。但是在运行的时候，它是个单独的二进制大映像。其模块间的通讯是通过直接调用其他模块中的函数实现的，而不是消息传递。单内核结构的例子：传统的UNIX内核---例如伯克利大学发行的版本，**Linux**内核。
 - **微内核**结构由一个非常简单的硬件抽象层和一组比较关键的原语或系统调用组成，这些原语仅仅包括了建立一个系统必需的几个部分，如线程管理，地址空间和进程间通信等。微内核的例子：AIX，BeOS，L4微内核系列，.Mach中用于GNU Hurd和**Mac OS X**，Minix，MorphOS，QNX，RadiOS，VSTa，**鸿蒙OS**。
 - **混合内核**它很像微内核结构，只不过它的组件更多的在核心态中运行，以获得更快的执行速度。混合内核实质上是微内核，只不过它让一些微核结构运行在用户空间的代码运行在内核空间，这样让内核的运行效率更高些。混合内核的例子：BeOS 内核，DragonFly BSD，ReactOS 内核，**Windows** NT、Windows 2000、Windows XP、Windows Server 2003以及Windows Vista等基于NT技术的操作系统。
 - **外内核**系统，也被称为纵向结构操作系统，是一种比较极端的设计方法。外内核这种内核不提供任何硬件抽象操作，但是允许为内核增加额外的运行库，通过这些运行库应用程序可以直接地或者接近直接地对硬件进行操作。外核设计**还停留在研究阶段**，没有任何一个商业系统采用了这种设计。几种概念上的操作系统正在被开发，如剑桥大学的Nemesis，格拉斯哥大学的Citrix系统和瑞士计算机科学院的一套系统。麻省理工学院也在进行着这类研究。
- **模块机制**的引入就是为了弥补这一缺点（可扩展性差、可维护性差）。
- **模块（内核模块，动态可加载内核模块，Loadable Kernel Module，LKM）**是Linux内核向外部提供的一个插口。



华为鸿蒙系统

Harmony OS



- 鸿蒙系统是基于**微内核**的全场景分布式系统，可以按需扩展，低延时，实现更广泛的系统安全。
- 鸿蒙OS有三层架构，第一层是**微内核**，第二层是基础服务，第三层是程序框架。所谓的第二层基础服务应该那些从内核态空间移到用户态空间的基础服务程序。当然，也可能部分还是运行的内核态，更像是混合内核。



• 7.4.1 Linux模块概述

- Linux内核支持动态可加载模块（**Loadable Kernel Module, LKM**），模块是内核的一部分，**模块通常是设备驱动程序**，但是并没有编译到内核里面去。
- 与模块相关的命令：
 - ① **insmod**: 加载模块
 - ② **rmmod**: 卸载模块
 - ③ **lsmod**: 列出已经安装的模块
 - ④ **depmod**: 产生模块依赖的映射文件
 - ⑤ **modprob**: 根据**depmod**命令所产生的相依关系，决定要载入哪些模块

• 7.4.2 模块代码结构

– RS-485驱动程序（模块）的代码结构：

- 头文件

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/kernel.h>
```

位于Ubuntu的/home/uptech/fsl-6dl-source/kernel-4.9.88/drivers/char/**uptech485.c**

- 模块宏声明

```
MODULE_AUTHOR("uptech kzkuan");
MODULE_DESCRIPTION("uart 485 control");
MODULE_LICENSE("GSL");
```

- 模块初始化函数

```
static int __init gpio_uart485_init(void)
{
    printk("\n\nnkzkuan____%s\n\n", __func__);
    return platform_driver_register(&gpio_uart485_device_driver);
}
```

- 模块退出函数

```
static void __exit gpio_uart485_exit(void)
{
    printk("\n\nnkzkuan____%s\n\n", __func__);
    platform_driver_unregister(&gpio_uart485_device_driver);
}
```

- 入口、出口函数设置

```
module_init(gpio_uart485_init);
module_exit(gpio_uart485_exit);
```

uptech485.c

RS-485驱动程序（模块）

头文件

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/init.h>
#include <linux/errno.h>
#include <linux/fs.h>
#include <linux/cdev.h>
#include <linux/types.h>
#include <linux/device.h>
#include <asm/system.h>
#include <asm/uaccess.h>
#include <linux/platform_device.h>
#include <asm/irq.h>
#include <linux/of.h>
#include <linux/of_device.h>
#include <linux/of_gpio.h>
```

```
#define DRVNAME "UART485"
#define UART485_MAJOR 30
#define UART485_MINOR 0
#define UART485_TX 1
#define UART485_RX 0
```

//模块名

```
static unsigned int gpio_ctrl;  
static struct class *uart485_class;  
static struct cdev uart485cdev;
```

全局变量

```
static int Uart485PowerOpen(struct inode *inode, struct file *filp)  
{  
    return 0;  
}
```

打开RS485函数

```
static ssize_t Uart485PowerWrite(struct file *filp, char __user *buf, size_t count, loff_t *ppos)  
{  
    return 0;  
}
```

写RS485函数

```
static ssize_t Uart485PowerRead(struct file *filp, char __user *buf, size_t count, loff_t *ppos)  
{  
    return 0;  
}
```

读RS485函数

```
static int Uart485PowerIoctl(struct file *filp,unsigned int cmd,unsigned long arg)
{
    gpio_request(gpio_ctrl,"uart485Ctrl");

    if(cmd == UART485_TX) //RS485发送
    {
        gpio_direction_output(gpio_ctrl,1);
    }
    else if(cmd == UART485_RX) //RS485接收
    {
        gpio_direction_output(gpio_ctrl,0);
    }

    gpio_free(gpio_ctrl);
    return 0;
}
```

```
static const struct file_operations uart485_fops = {  
    .owner = THIS_MODULE,  
    .write = Uart485PowerWrite,  
    .read = Uart485PowerRead,  
    .open = Uart485PowerOpen,  
    .unlocked_ioctl = Uart485Powerioctl,  
};
```

RS485的文件操作结构体

```
static int Uart485Init(void)
```

```
{
```

```
    dev_t devt;
```

```
    int retval;
```

```
    devt = MKDEV(UART485_MAJOR,UART485_MINOR);
```

```
    retval = register_chrdev_region(devt,1,DRVNAME);
```

```
    if(retval>0)
```

```
        return retval;
```

```
    cdev_init(&uart485cdev,&uart485_fops);
```

```
    retval = cdev_add(&uart485cdev,devt,1);
```

```
    if(retval)
```

```
        goto error;
```

RS485初始化函数

RS485初始化函数

```
uart485_class = class_create(THIS_MODULE,"UART485");
if (IS_ERR(uart485_class)) {
    printk(KERN_ERR "Error creating raw class.\n"); //内核打印函数
    cdev_del(&uart485cdev);
    goto error;
}

device_create(uart485_class, NULL, MKDEV(UART485_MAJOR,UART485_MINOR), NULL,DRVNAME);
gpio_request(gpio_ctrl,"uart485Ctrl");
gpio_direction_output(gpio_ctrl,0);
gpio_free(gpio_ctrl);
return 0;

error:

unregister_chrdev_region(devt, 1);
return retval;

}
```

static void **Uart485Exit**(void)

RS485退出函数

```
{  
    device_destroy(uart485_class, MKDEV(UART485_MAJOR,UART485_MINOR));  
    class_destroy(uart485_class);  
    cdev_del(&uart485cdev);  
    unregister_chrdev_region(MKDEV(UART485_MAJOR,UART485_MINOR), 1);  
}
```



```

static int gpio_uart485_probe(struct platform_device *pdev)
{
    struct device *dev = &pdev->dev;
    struct device_node *of_node;
    of_node = dev->of_node;

    if (!of_node) {
        return -ENODEV;
    }

    gpio_ctrl = of_get_named_gpio(of_node,"uartctrl",0);

    if(!gpio_is_valid(gpio_ctrl))
    {
        return -ENODEV;
    }

    printk("\n\n\nkzkuan____%s\n\n\n",__func__);
    Uart485Init();
    return 0;
}

```

RS485探测函数

RS485移除函数

```
static int gpio_uart485_remove(struct platform_device *pdev)
{
    Uart485Exit();
    return 0;
}
```

```
static struct of_device_id gpio_uart485_of_match[] =
{
    { .compatible = "fsl,gpio-uart485-ctrl", },
    { },
};
```

设备ID结构体

```
MODULE_DEVICE_TABLE(of, gpio_uart485_of_match);
```

注册设备表函数

```
static struct platform_driver gpio_uart485_device_driver =
{
    .probe      = gpio_uart485_probe,
    .remove     = gpio_uart485_remove,
    .driver     = {
        .name   = "gpio-uart485-ctrl",
        .owner  = THIS_MODULE,
        .of_match_table = of_match_ptr(gpio_uart485_of_match),
    }
};
```

平台驱动结构体

```
static int __init gpio_uart485_init(void)
```

RS485 模块初始化函数

```
{  
    printk("\n\nnkzkuan____%s\n\n\n", __func__);  
    return platform_driver_register(&gpio_uart485_device_driver);  
}
```

```
static void __exit gpio_uart485_exit(void)
```

RS485 模块退出函数

```
{  
    printk("\n\nnkzkuan____%s\n\n\n", __func__);  
    platform_driver_unregister(&gpio_uart485_device_driver);  
}
```

```
module_init(gpio_uart485_init);
```

模块入口函数

```
module_exit(gpio_uart485_init);
```

模块出口函数

```
MODULE_AUTHOR("uptech kzkuang");
```

//作者

```
MODULE_DESCRIPTION("uart 485 control");
```

//描述

```
MODULE_LICENSE("GSL");
```

//许可证

• 7.4.3 模块的加载

– 模块加载的两种方式：

① 手工加载模块：通过`insmod`命令将模块加载到内核：

– `insmod led.o`：加载`led.o`模块

② 根据需要加载模块到内核：当内核发现需要某个模块时，内核守护进程（`kernelld`）加载该模块到内核。

- **7.4.4 模块的卸载**

- 使用**rmmod**命令卸载模块：

- **rmmod led**：卸载**led**模块

- 但是当内核在使用模块时，该模块是不能被卸载的。

- 7.4.5 版本依赖

- 模块代码一定要在连接到不同内核版本之前重新编译，因为模块是结合到某个特殊内核版本的数据结构和数据原型上，不同的内核版本的接口可能差别很大。
- 模块依赖于内核的版本。

7.5 ARM-Linux系统启动和初始化

• 7.5.1 使用Boot Loader将内核映像载入

- Boot Loader将Linux的内核加载到内存（SDRAM）后，将跳到函数 **start_kernel()** 进入初始化过程。
- **start_kernel()**函数：位于/home/uptech/fsl-6dl-source/kernel-4.9.88/init/**main.c**中。

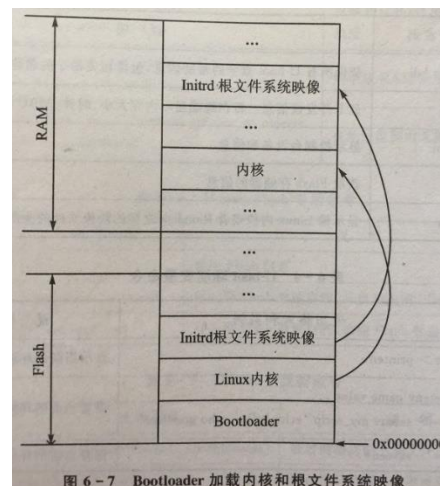



图 6-7 Bootloader 加载内核和根文件系统映像

start_kernel()函数

位于Ubuntu的/home/uptech/fsl-6dl-source/kernel-4.9.88/init/main.c

```
asmlinkage __visible void __init start_kernel(void)   
{  
    char *command_line;  
    char *after_dashes;  
  
    set_task_stack_end_magic(&init_task);  
    smp_setup_processor_id();  
    debug_objects_early_init();  
  
    /*  
     * Set up the the initial canary ASAP:  
     */  
    boot_init_stack_canary();  
  
    cgroup_init_early();  
  
    local_irq_disable();  
    early_boot_irqs_disabled = true;  
  
    /*  
     * Interrupts are still disabled. Do necessary setups, then  
     * enable them  
     */  
    boot_cpu_init();  
    page_address_init();  
    pr_notice("%s", linux_banner);  
    setup_arch(&command_line);  
    mm_init_cpumask(&init_mm);  
    setup_command_line(command_line);  
    setup_nr_cpu_ids();  
    setup_per_cpu_areas();  
    boot_cpu_state_init();  
    smp_prepare_boot_cpu(); /* arch-specific boot-cpu hooks */  
  
    build_all_zonelists(NULL, NULL);  
    page_alloc_init();  
}
```

• 7.5.2 内核引导第一部分：内核数据结构初始化

- `start_kernel()`函数中调用了一系列初始化函数，以完成内核（Kernel）本省的设置。
- `start_kernel()`函数最后启动init过程，创建第一个内核线程，调用 `init()`函数。

```
check_bugs();

acpi_subsystem_init();
sfi_init_late();

if (efi_enabled(EFI_RUNTIME_SERVICES)) {
    efi_late_init();
    efi_free_boot_services();
}

ftrace_init();

→ /* Do the rest non-__init'ed, we're now alive */
rest_init();
}
```

• 7.5.3 内核引导第二部分：外设初始化

- **init()**函数作为内核线程，首先锁定内核，然后调用**do_basic_setup()**函数，完成**外设**及其驱动程序的**加载**和**初始化**。

```
static void __init do_basic_setup(void) ←  
{  
    cpuset_init_smp();  
    shmem_init();  
    driver_init();  
    init_irq_proc();  
    do_ctors();  
    usermodehelper_enable();  
    do_initcalls();  
}
```

• 7.5.4 init进程和inittab脚本

- **init进程**是系统所有进程的起点，内核在完成核内引导以后，即在本线程（进程）空间内加载init程序，它的进程号是1。
- init程序需要读取Ubuntu的**inittab脚本文件**作为其行为指针。

• 7.5.5 rc启动脚本

- Linux系统运行后将启动rc脚本（**S06rc.local**）。
- rc启动脚本（**S06rc.local**）位于Ubuntu的/etc/rc2.d/目录下。

Ubuntu的rc启动脚本（S06rc.local）

```
#!/bin/sh
### BEGIN INIT INFO
# Provides:          rc.local
# Required-Start:    $all
# Required-Stop:
# Default-Start:     2 3 4 5
# Default-Stop:
# Short-Description: Run /etc/rc.local if it exist
### END INIT INFO

PATH=/sbin:/usr/sbin:/bin:/usr/bin

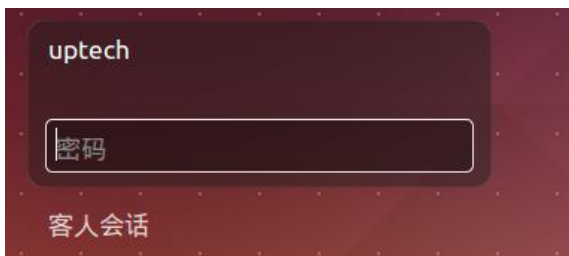
. /lib/init/vars.sh
. /lib/lsb/init-functions

do_start() {
    if [ -x /etc/rc.local ]; then
        [ "$VERBOSE" != no ] && log_begin_msg "Running local boot scripts (/etc/rc.local)"
        /etc/rc.local
        ES=$?
        [ "$VERBOSE" != no ] && log_end_msg $ES
        return $ES
    fi
}

case "$1" in
    start)
        do_start
        ;;
    restart|reload|force-reload)
        echo "Error: argument '$1' not supported" >&2
        exit 3
        ;;
    stop|status)
        # No-op
        exit 0
        ;;
    *)
        echo "Usage: $0 start|stop" >&2
        exit 3
        ;;
esac
```

• 7.5.6 Shell的启动

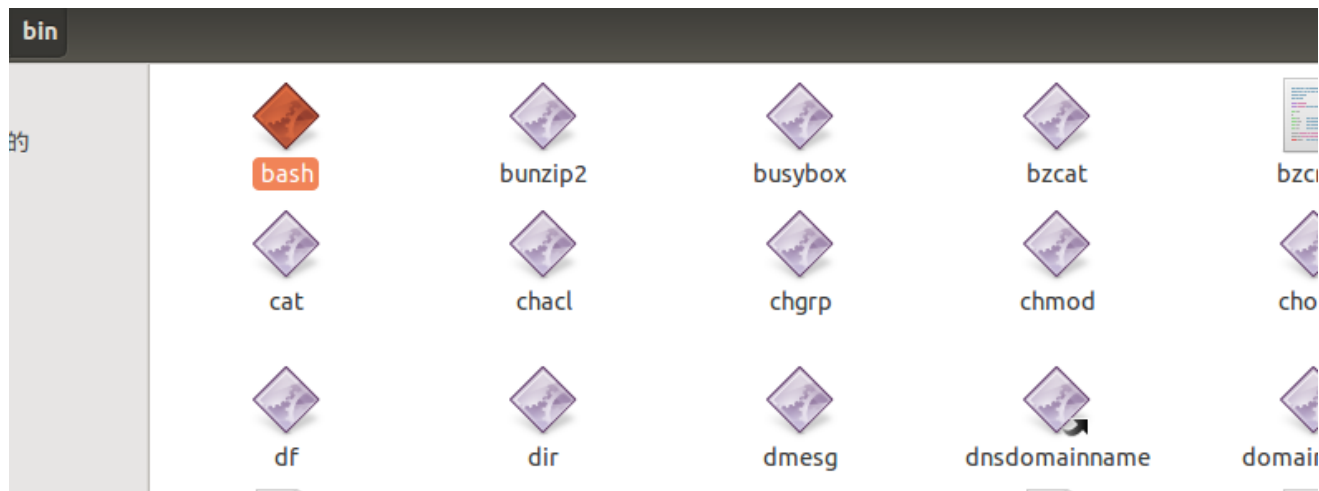
- Login用户（Ubuntu是**uptech**用户，实验箱是**root**用户）将启动一个用户指定的Shell，这个指定的Shell就是**/bin/bash**
 - **Shell**：俗称壳（用来区别于核），是指“为用户提供操作界面”的软件（命令解析器）。
- **bash**：是一个为GNU计划编写的Unix shell。它的名字是一系列缩写：**Bourne-Again SHell** — 这是关于Bourne shell（sh）的一个双关语（Bourne again / born again）。Bourne shell是一个早期的重要shell，由史蒂夫·伯恩在1978年前后编写，并同Version 7 Unix一起发布。**bash**则在1987年由布莱恩·福克斯创造。在1990年，Chet Ramey成为了主要的维护者。



Ubuntu的登录界面

```
Poky (Yocto Project Reference Distro) 1.7 imx6dlsabresd /dev/ttymx0  
imx6dlsabresd login: root
```

实验箱的登录界面



Ubuntu的Shell
(/bin/bash)

```
imx6dlsabresd login: root
```

```
Password:
```

```
root@imx6dlsabresd:~# cd /bin
```

```
root@imx6dlsabresd:/bin# ls
```

```
ash          dumpkmap      mkdir.coreutils  sed
base64       echo          mknod            sed.sed
bash         echo.coreutils mknod.coreutils  sh
busybox      egrep         mktemp           sleep
```

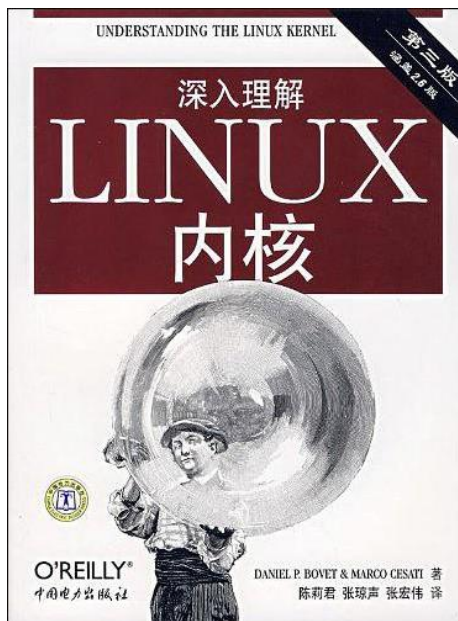
实验箱的Shell
(/bin/bash)

小结

- **ARM-Linux内核的：**
 - ① 内存管理（存储管理）
 - ② 进程管理和调度（多进程调度）
 - ③ 模块机制（模块：驱动程序）
 - ④ 系统启动和初始化

进一步探索

- 阅读《**Understanding the Linux Kernel**》一书，对Linux内核进行深入了解。
- 通过移植内核以及制作内核模块实验进行实践。



Thanks