

Linux 驱动系列课-指导手册

(V1.1)

2023 年 4 月 30

修改历史

版本	作者	修改	备注
V1.0		内容创建	
V1.1		格式重排。	

目 录

第一章 《操作系统基础与 LINUX 系统导论》	5
课程目标.....	5
01 操作系统基础.....	5
1.1 什么是操作系统.....	5
1.2 操作系统分类.....	5
1.3 操作系统与 CPU 的关系.....	7
1.4 国产 Linux 操作系统.....	8
02 LINUX 系统.....	10
2.1 Linux 历史.....	10
2.2 Linux 系统.....	12
03 LINUX 内核.....	16
3.1 Linux 内核.....	16
3.2 Linux 内核源码.....	17
重点回顾.....	23
思考练习.....	23
第二章 《LINUX 系统调用及文件 IO 编程》	23
课程目标.....	23
01 文件描述符.....	23
1.1 Linux 中的文件.....	24
1.2 文件描述符.....	26
02 文件 IO 与标准 IO.....	27
2.1 Linux 系统编程.....	27
2.2 IO 常见函数.....	29
03 系统调用与 POSIX 标准.....	35
3.1 系统调用.....	35
3.2 POSIX 标准.....	37
重点回顾.....	38
思考练习.....	38
第三章 《LINUX 内核模块编程》	38
课程目标.....	38
01 内核模块概述.....	38
1.1 什么是内核模块.....	38
1.2 编程三要素.....	39
02 内核模块编程三步法.....	39
2.1 编程：内核模块编程.....	39
2.2 编译：内核模块编译.....	43
2.3 运行：内核模块加载.....	44
重点回顾.....	44
思考练习.....	45
第四章 《LINUX 设备驱动基础与接口实现》	45
课程目标.....	45
01 LINUX 设备驱动基础.....	45
1.1 设备驱动.....	45
1.2 设备文件.....	49
02 LINUX 字符设备驱动.....	50
2.1 基础概念.....	50

2.2 重要结构体.....	51
2.3 字符设备驱动流程分析.....	55
03 字符设备驱动注册	57
3.1 chrdev 版注册.....	57
3.2 cdev 版注册.....	58
04 字符设备驱动接口	61
4.1 字符设备驱动接口概述.....	61
4.2 打开和关闭设备.....	62
4.3 控制设备 (ioctl)	63
4.4 读写设备 (read/read)	66
重点回顾.....	69
思考练习.....	69

第一章 《操作系统基础与 linux 系统导论》

课程目标

- 了解操作系统的基本概念
- 了解 Linux 系统的发展历史
- 理解 Linux 系统框架
- 了解 Linux 内核

01 操作系统基础

1.1 什么是操作系统

- 广义的操作系统包括：计算机（PC、工作站、服务器）系统、移动端系统、嵌入式系统等。狭义的是指计算机操作系统。
- 计算机操作系统的功能角色：作为用户和计算机硬件资源之间的交互，管理调度硬件资源，为应用软件提供运行环境。
- 操作系统属于基础软件，是系统级程序的汇集，为用户屏蔽底层硬件复杂度，并提供编程接口和操作入口。操作系统控制处理器 (CPU) 调度系统资源，控制应用程序执行的时机，决定各个程序分配的处理器时间 (CPU time)。
- 操作系统需要兼容底层硬件和应用软件，才能实现计算机的功能。

1.2 操作系统分类

1.2.1 按应用领域划分

按应用领域划分：桌面、服务器、移动端操作系统



1.2.2 按代码是否开源划分

根据核心代码是否向开放，操作系统可划分为两类：闭源系统、开源系统。



1、闭源操作系统—Windows 系统

闭源操作系统：代码不开放（以微软 Windows 系统为代表）

- 微软公司内部的研发团队开发 Windows 操作系统，并开发配套的应用软件，比如 Office。在生态建设方面，Intel 和 Windows 长期合作形成 Wintel 体系，在 PC 端市场占有率全球领先。
- Windows 系统的访问分为 User mode(用户模式)和 Kernel mode(内核模式)。用户级的应用程序在用户模式中运行，而系统级的程序在内核模式中运行。内核模式允许访问所有的系统内存和 CPU 指令。

- Windows 系统最大的优势在于图形界面，使得普通用户操作起来非常便利。相比大部分 Linux 系统，windows 的常用软件安装和系统设置不需要以命令行的方式去输入系统指令，只需要点击“按钮”即可完成。如今，绝大多数常见软件、专用软件和底层硬件都支持 Windows 操作系统，形成了 Window 强大的生态整体。
- 微软 windows 核心技术是过于封闭的，不会对外开放，用户数据信息的安全与隐私无法得到保障。

2、开源操作系统—Linux 系统

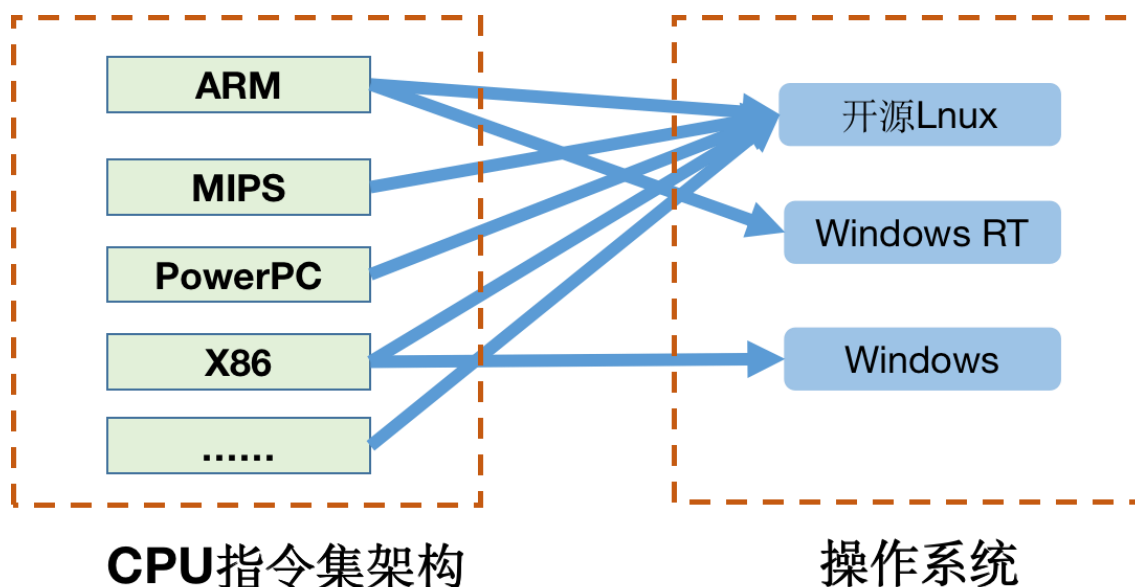
开源操作系统：代码免费开放，（以 Linux 操作系统为代表）

- Linux kernel 是开源项目，由全球范围的开发者(企业、团体、独立开发者)共同贡献源代码。
- Linux 的官方组织是 Linux 基金会，作为非盈利的联盟，协调和推动 Linux 系统的发展以及宣传、保护和规范 Linux。
- Linux 基金会由开源码发展实验室（Open Source Development Labs, OSDL）与自由标准组织（Free Standards Group ,FSG）于 2007 年联合成立。

1.3 操作系统与 CPU 的关系

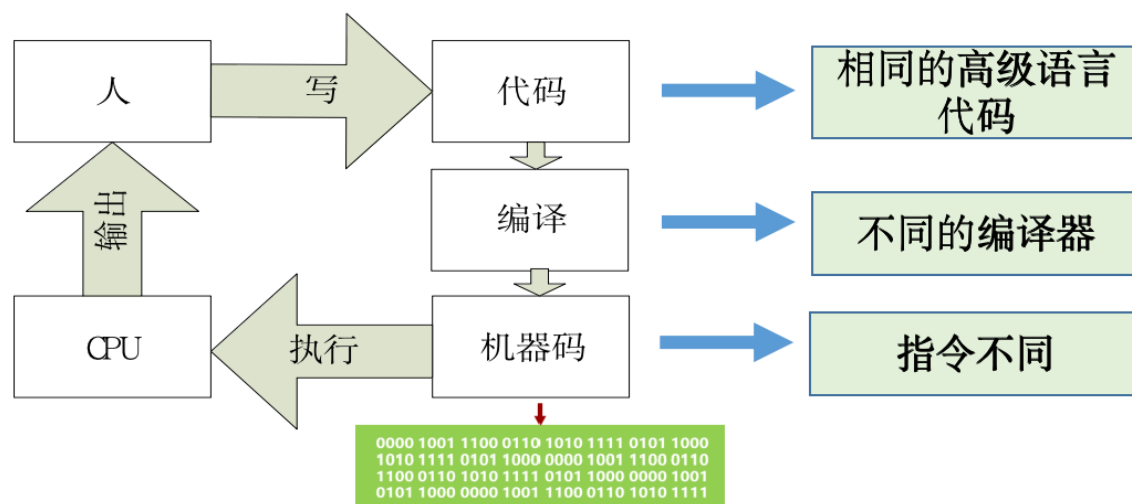
1.3.1 操作系统和 CPU 之间的关系

- 操作系统和 CPU 之间的关系：Linux 操作系统支持多种指令集，WINDOWS 操作系统只支持 x86 指令集。
- CPU 指令集：ARM 、X86、MIPS 等。
- 操作系统： 开源 LINUX 和闭源 WINDOWS 等。



1.3.2 应用程序的编译流程

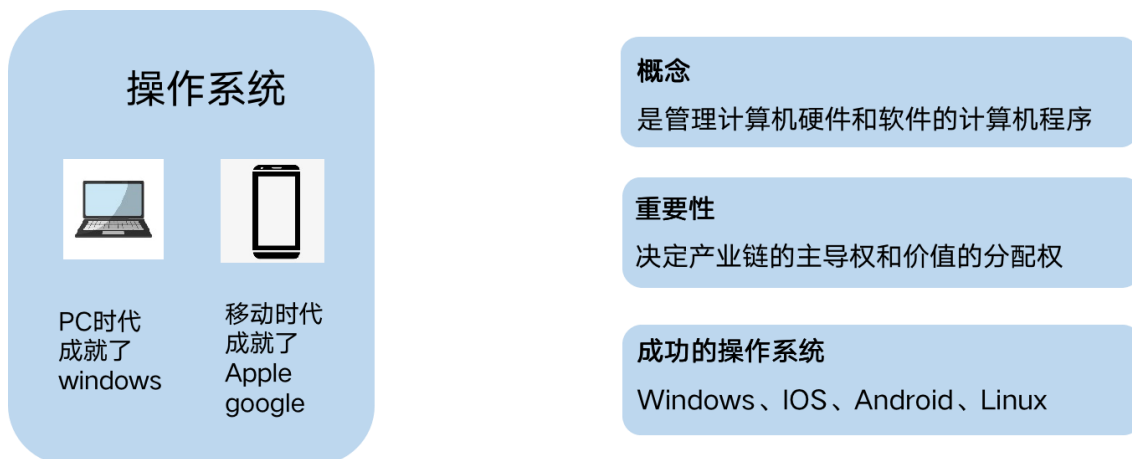
- 应用程序的编译流程：应用程序→操作系统→CPU 指令。
- eg：一个.c 应用程序，经操作系统编译为 CPU 指令，在 CPU 架构上执行。
- 注意：一个应用程序，由操作系统编译为 ARM 指令，就只能在 ARM 体系架构上运行；编译为 x86 指令，就只能在 x86 体系架构上运行。



1.4 国产 Linux 操作系统

1.4.1 背景：信创产业的根基


- 操作系统是连接硬件和数据库、中间件、应用软件、常见外设的中间环节，是计算机生态的重要组织部分。
- 主要作用：对计算机或服务器进行资源管理，为用户提供方便的操作界面。
- CPU 和操作系统是整个信创产业的根基，没有 CPU 和操作系统的可控，整个信创产业就是无根之木，无源之水。



1.4.2 技术：国产操作系统的内核

- 目前主流操作系统主要有 PC 端的 windows、linux 等，服务器操作系统 Unix/Linux，Windows Server。
- 目前国产操作系统均是基于 Linux 内核进行的二次开发。

The Linux Kernel Archives




[About](#)
[Contact us](#)
[FAQ](#)
[Releases](#)
[Signatures](#)
[Site news](#)

Protocol	Location
HTTP	https://www.kernel.org/pub/
GIT	https://git.kernel.org/
RSYNC	rsync://rsync.kernel.org/pub/

Latest Release

5.19.1



	Version	Date	tarball	pgp	patch	inc. patch	view diff	browse	changelog
mainline:	6.0-rc1	2022-08-14	[tarball]		[patch]			[browse]	
stable:	5.19.1	2022-08-11	[tarball]	[pgp]	[patch]		[view diff]	[browse]	[changelog]
stable:	5.18.17	2022-08-11	[tarball]	[pgp]	[patch]	[inc. patch]	[view diff]	[browse]	[changelog]
longterm:	5.15.60	2022-08-11	[tarball]	[pgp]	[patch]	[inc. patch]	[view diff]	[browse]	[changelog]
longterm:	5.10.136	2022-08-11	[tarball]	[pgp]	[patch]	[inc. patch]	[view diff]	[browse]	[changelog]
longterm:	5.4.210	2022-08-11	[tarball]	[pgp]	[patch]	[inc. patch]	[view diff]	[browse]	[changelog]
longterm:	4.19.255	2022-08-11	[tarball]	[pgp]	[patch]	[inc. patch]	[view diff]	[browse]	[changelog]
longterm:	4.14.290	2022-07-29	[tarball]	[pgp]	[patch]	[inc. patch]	[view diff]	[browse]	[changelog]
longterm:	4.9.325	2022-07-29	[tarball]	[pgp]	[patch]	[inc. patch]	[view diff]	[browse]	[changelog]
linux-next:	next-20220816	2022-08-16						[browse]	

1.4.3 历史：国产操作系统发展 30 年

1、国产操作系统发展历程

国产操作系统发展任重道远，在国产操作系统的发展历程中先后出现了十几家厂商。



2、国产操作系统发展 30 年

国产通用操作系统发展的几个阶段：

- 1989 年之前：无自主通用操作系统 CCDOS、汉化版 UNIX、Xenix 等；
- 1989 年~1999 年：中软公司牵头自主研发操作系统及国际合作探索阶段（十年萌芽期），COSA / COSIX、COSIX64；
- 1999 年~2009 年：开源软件（十年成长期），中软 Linux、红旗 Linux、Xteam、蓝点 Linux 等；
- 2010 年~2019 年：国产软硬件加速发展（十年成熟期），中标软件、天津麒麟等生态体系在实际应用中加速完善，成熟度日益提升；
- 2020 年~ ：国产基础软件做大做强阶段，冰火麒麟合并，麒麟软件应运而生。

02 Linux 系统

2.1 Linux 历史

2.1.1 Unix 的历史

1、Unix 诞生

- Unix：1969 年，Unix 系统的第一个版本（由 Ken Thompson 在 AT&T 贝尔实验室实现）

2、C 语言诞生

- 1973 年 Ken Thompson 与 Dennis Ritchie 用他们重新发明的 C 语言重写了 Unix 的第三版内核。

3、当今 Unix 的两大主流

- Unix System V：随着 Unix 的广泛应用 AT&T 开始认识到 Unix 的价值，1979 年成立了专门的 Unix 实验室（USL），并且 AT&T 同时宣布了对 Unix 的拥有权和商业化。
- BSD Unix：20 世纪 70 年代末，AT&T 成立 Unix 系统实验室，CSRG（加州大学伯克利分校计算机系统研究小组）使用 Unix 对操作系统进行研究，最终有了伯克利自己的版本：BSD Unix。

2.1.2 Linux 两大人物

1、Richard Stallman

- 1983年 GNU计划
- 1985年 FSF基金会
- 1990年 Emacs、GCC（c语言的编译器）、程序库
- 1991年 Stallman去找Linus，商谈让Linux加入其开源计划（GNU计划）
- 1992年 GNU/Linux

- copyleft: 代表无版权。copyright: 则代表有版权。
- GPL: 通用版权许可证协议。
- opensource free: 源代码开放，使用GPL协议保护。



理查德·马修·斯托曼（Richard Matthew Stallman, RMS），于1953年出生，自由软件运动的精神领袖、GNU计划以及自由软件基金会（Free Software Foundation）的创立者、著名黑客。

2、Linux

- 1990年，Linus Torvalds首次接触MINIX
- 1991年，Linus开始在MINIX上编写各种驱动程序等操作系统内核组件
- 1991年底，Linus公开了Linux内核
- 1993年，Linux1.0版本发行，Linux转向GPL版权协议
- 1994年，Linux的第一个商业发行版Slackware问世
- 1996年，美国国家标准技术局的计算机系统实验室确认Linux版本1.2.13符合POSIX标准
- 1999年，Linux的简体中文发行版相继问世



林纳斯·托瓦兹：当今世界最著名的程序员、黑客，开源操作系统Linux之父。1969年生于芬兰，毕业于赫尔辛基大学，1997年~2003年任职于美国加州硅谷的全美达公司，现受聘于开放源代码开发实验室，全力开发Linux内核。

2.1.3 Linux 起源的核心要素

Linux 操作系统的诞生、发展和成长过程中三个核心因素：Minix 操作系统、GNU 计划、POSIX 标准。

Minix操作系统	GNU计划	POSIX标准
		

2.1.4 Linux 崛起的核心优势

1、优势领域

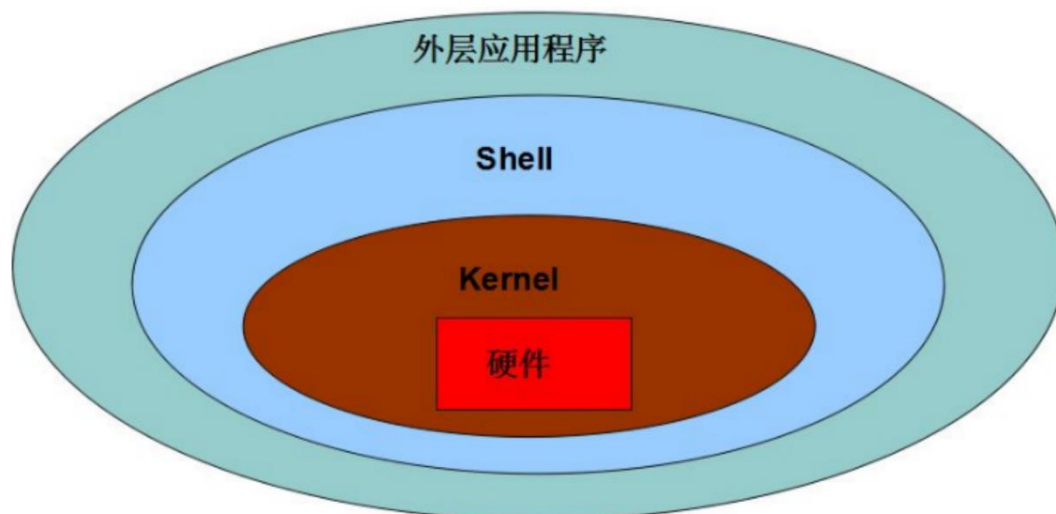
- Linux 操作系统主要的优势领域是服务器和嵌入式。
- 据 Linux 基金会统计，全球 90%的公有云平台采用了 Linux 系统，99%的超算和 62%的嵌入式设备也都是基于 Linux。全球公有云平台运行的所有应用，超过 54%是运行在 Linux 虚拟机上。
- 根据 IDC 在 2017 年的统计数据，全球服务器操作系统使用份额(免费+付费)中， 68%是 Linux 服务器操作系统。

2、核心优势

- 创新方面，集思广益。全球开发者对 Linux 内核保持了持续的更新，提供了充足的创新动力。
- 系统代码可以修改和自定义，用户可调用计算机资源的自由度极高。相比 Windows 等闭源系统，Linux 支持了使用者极大的使用自由度。
- 运行效率高，运维成本低。Linux 系统在服务器上运行效率较高，相对比较轻量化，天然支持虚拟化。在服务器集群上运维成本较低。
- 安全。Linux 从发展根源上就是针对多用户系统设计的，系统管理员和 root 用户具有系统管理权限，全球开发者多次的筛查和更新。

2.2 Linux 系统

2.2.1 计算机系统组成

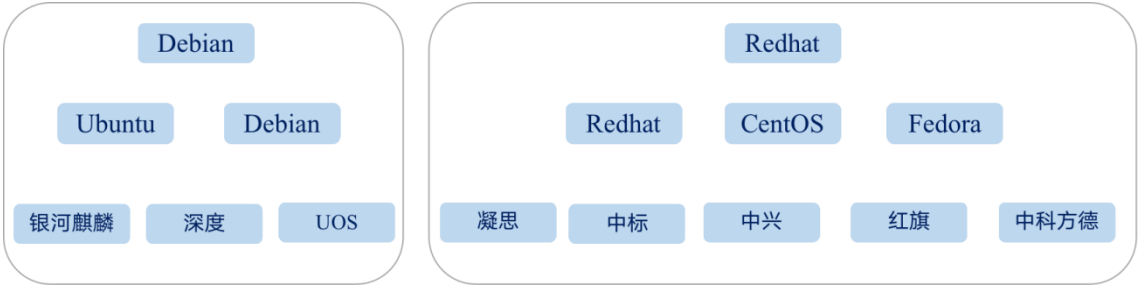


App（应用程序）	应用的范围涵盖了从桌面工具和编程语言到多用户业务套件等各种软件。大多数 Linux 发行版都会提供一个应用软件仓库，用于搜索和下载其他应用。
Shell（GNU 等系统工具）	系统级任务（如配置和软件安装）的管理层。它包括 shell（或称为命令行）、守护进程（在后台运行的进程）和桌面环境。
Kernel（内核）	内核管理着系统的资源，并与硬件进行通信。它负责内存、进程和文件的管理。
HW（硬件）	CPU、存储器、外设

2.2.2 Linux 发行版

1、什么是 Linux 发行版

- Linux 发行版：是一个由 Linux 内核、GNU 工具、附加软件和软件包管理器组成的操作系统，它也可能包括显示服务器和桌面环境，以用作常规的桌面操作系统。
- Linux 发行版可以大体分为两类：以 Redhat 为代表和以 Debian 为代表。
- 主要国产操作系统：麒麟 OS、UOS、凝思、红旗、一铭软件、中科方德等。
- 我们常说的“Linux 系统、操作系统”指的是一个完整的发行版。



2、Linux 发行版与 Linux 内核的区别

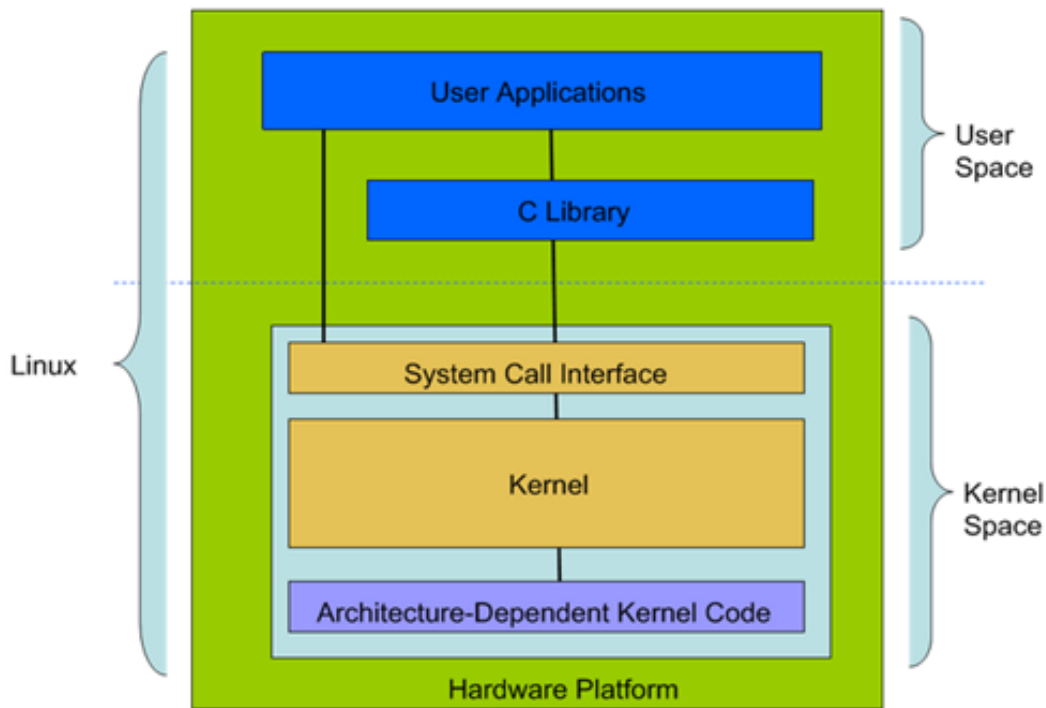
- 我们常说的“Linux”指的是 Linux 内核。
- Linux 内核：内核是一个操作系统的核心，你不能直接使用内核，只能通过应用程序和 shell 与它交互。
- 例如：一个 Linux 发行版可以看作是一个汽车制造商（比如丰田或福特）为你提供的现成的汽车，就像 Ubuntu 或 Fedora 发行版的发行商为你提供一个基于 Linux 的现成操作系统一样。

3、Linux 与 GNU/Linux 的区别

- Linux 只是一个内核，为了与 Linux 内核进行交互，你需要在 shell 中运行一些命令来完成一些工作。
- GNU 项目实现了许多流行的 Unix 实用程序，如 cat、grep、awk、shell (bash)，同时还开发了自己的编译器 (GCC) 和编辑器 (Emacs)。
- 由于 Linux 与 GNU 工具集成得很深，几乎是完全依赖于 GNU 工具，所以也有些人称它为 GNU Linux (写成 GNU/Linux)。

2.2.3 Linux 系统组成结构

Linux 体系结构从大的方面可以分为用户空间 (User Space) 和内核空间 (Kernel Space)，如下图所示：



User Space	1、用户程序：最上面是用户（或应用程序）空间。这是用户应用程序执行的地方。 2、函数库：它提供了连接内核的系统调用接口，还提供了在用户空间应用程序和内核之间进行转换的机制。GNU C Library（glibc）也在这里。
Kernel Space	3、系统调用：系统调用接口，它实现了一些基本的功能，例如 read 和 write。内核和用户空间的应用程序使用的是不同的保护地址空间。每个用户空间的进程都使用自己的虚拟地址空间，而内核则占用单独的地址空间。 4、 内核子系统：设备管理、内存管理、文件系统 5、硬件相关代码：厂商

	BSP、device driver
Hardware	6、CPU、存储、外设

2.2.4 Linux 系统文件目录

- `/`: 根目录, 所有的目录、文件、设备都在`/`之下, `/`就是 Linux 文件系统的组织者, 也是最上级的领导者。
- `/bin`: `bin` 就是二进制 (binary) 英文缩写。在一般的系统当中, 都可以在这个目录下找到 linux 常用的命令。系统所需要的那些命令位于此目录。
- `/boot`: Linux 的内核及引导系统程序所需要的文件目录, 比如 `vmlinuz` `initrd.img` 文件都位于这个目录中。在一般情况下, GRUB 或 LILO 系统引导管理器也位于这个目录。
- `/etc`: `etc` 这个目录是 linux 系统中最重要目录之一。在这个目录下存放了系统管理时要用到的各种配置文件和子目录。要用到的网络配置文件, 文件系统, x 系统配置文件, 设备配置信息, 设置用户信息等都在这个目录下。
- `/lib`: `lib` 是库 (library) 英文缩写。这个目录是用来存放系统动态连接共享库的。几乎所有的应用程序都会用到这个目录下的共享库。因此, 千万不要轻易对这个目录进行什么操作, 一旦发生问题, 系统就不能工作了。
- `/dev`: `dev` 是设备 (device) 的英文缩写。这个目录对所有的用户都十分重要。因为在这个目录中包含了所有 linux 系统中使用的外部设备。但是这里并不是放的外部设备的驱动程序。这一点和常用的 windows, dos 操作系统不一样。它实际上是一个访问这些外部设备的端口。可以非常方便地去访问这些外部设备, 和访问一个文件, 一个目录没有任何区别。
- `/proc`: 可以在这个目录下获取系统信息。这些信息是在内存中, 由系统自己产生的。
- `/home`: 如果建立一个用户, 用户名是“xx”, 那么在`/home`目录下就有一个对应的`/home/xx`路径, 用来存放用户的主目录。
- `/usr`: 这是 linux 系统中占用硬盘空间最大的目录。用户的很多应用程序和文件都存放在这个目录下。在这个目录下, 可以找到那些不适合放在`/bin`或`/etc`目录下的额外的工具
- `/usr/local`: 这里主要存放那些手动安装的软件, 即不是通过“新立得”或 `apt-get` 安装的软件。它和`/usr`目录具有相类似的目录结构。让软件包管理器来管理`/usr`目录, 而把自定义的脚本 (scripts) 放到`/usr/local`目录下面。
- `/usr/share` : 系统共用的东西存放地, 比如 `/usr/share/fonts` 是字体目录, `/usr/share/doc` 和 `/usr/share/man` 帮助文件。
- `/root`: Linux 超级权限用户 `root` 的家目录。
- `/sbin`: 这个目录是用来存放系统管理员的系统管理程序。大多是涉及系统管理的命令的存放, 是超级权限用户 `root` 的可执行命令存放地, 普通用户无权限执行这个目录下的命令, 这个目录和`/usr/sbin`; `/usr/X11R6/sbin` 或 `/usr/local/sbin` 目录是相似的, 凡是目

录 `sbin` 中包含的都是 `root` 权限才能执行的。

- `/cdrom`: 这个目录在刚刚安装系统的时候是空的。可以将光驱文件系统挂在这个目录下。例如: `mount /dev/cdrom /cdrom`
- `/mnt`: 这个目录一般是用于存放挂载储存设备的挂载目录的, 比如有 `cdrom` 等目录。可以参看 `/etc/fstab` 的定义。
- `/media`: 有些 linux 的发行版使用这个目录来挂载那些 `usb` 接口的移动硬盘 (包括 U 盘)、`CD/DVD` 驱动器等。
- `/lost+found`: 在 `ext2` 或 `ext3` 文件系统中, 当系统意外崩溃或机器意外关机, 而产生一些文件碎片放在这里。当系统启动的过程中 `fsck` 工具会检查这里, 并修复已经损坏的文件系统。有时系统发生问题, 有很多的文件被移到这个目录中, 可能会用手工的方式来修复, 或移到文件到原来的位置上。
- `/opt`: 这里主要存放那些可选的程序。
- `/selinux` : 对 `SElinux` 的一些配置文件目录, `SElinux` 可以让 `linux` 更加安全。
- `/srv` 服务启动后, 所需访问的数据目录, 举个例子来说, `www` 服务启动读取的网页数据就可以放在 `/srv/www` 中。
- `/tmp`: 临时文件目录, 用来存放不同程序执行时产生的临时文件。有时用户运行程序的时候, 会产生临时文件。`/tmp` 就用来存放临时文件的。`/var/tmp` 目录和这个目录相似。
- `/var`: 这个目录的内容是经常变动的, 看名字就知道, 可以理解为 `vary` 的缩写, `/var` 下有 `/var/log` 这是用来存放系统日志的目录。`/var/ www` 目录是定义 `Apache` 服务器站点存放目录; `/var/lib` 用来存放一些库文件, 比如 `MySQL` 的, 以及 `MySQL` 数据库的的存放地。

03 Linux 内核

3.1 Linux 内核

3.1.1 什么是 Linux 内核

- 操作系统位于应用与硬件之间, 负责在所有软件与相关的物理资源之间建立连接。所以, 我们又称 `Linux` 操作系统为 `Linux` 内核。
- `linus` 说: `Linux` (内核) 就是为上层应用程序提供运行环境并管理整个系统软硬件资源的一个程序 (管理和服务程序) 。

3.1.2 Linux 内核作用

我们进一步看看 `Linux` 系统结构中内核的位置和作用

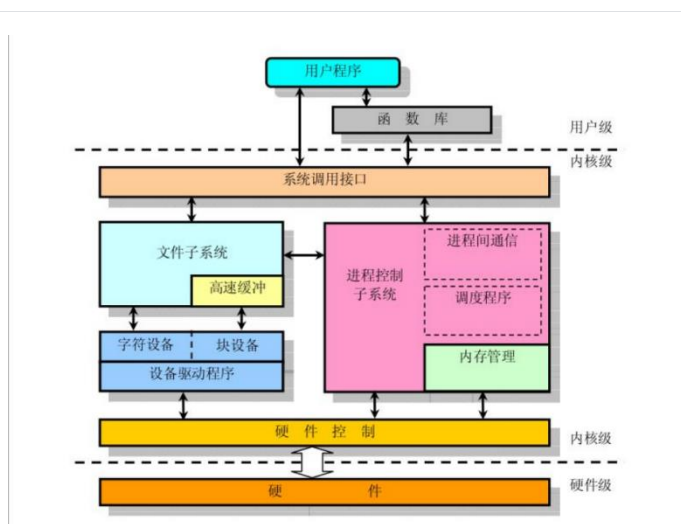
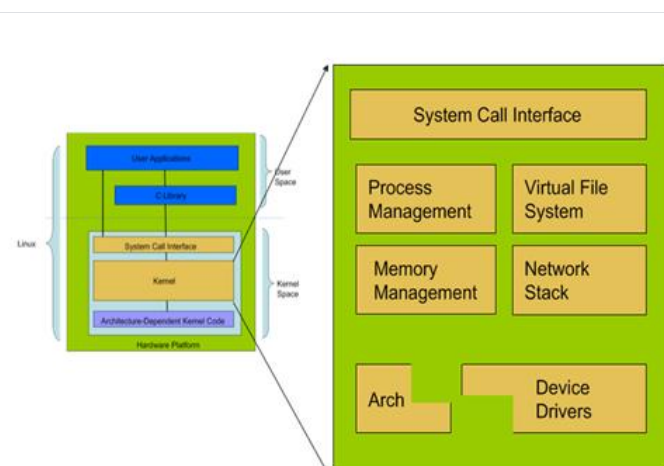


图 2-4 内核结构框图

- 用户程序：最上面是用户（或应用程序）空间。这是用户应用程序执行的地方。
- 函数库：GNU C Library（glibc 也在这里）它提供了连接内核的系统调用接口。
- 系统调用：系统调用接口，它实现了一些基本的功能，例如 read 和 write。
- 硬件控制：通常称为 BSP（Board Support Package）

3.1.3 Linux 内核组成



内核主要完成如下功能：

- SCI：系统调用接口
- PM：进程管理
- VFS：虚拟文件系统
- MM：内存管理
- Network Stack：内核协议栈
- Arch：体系架构
- DD：设备驱动

3.2 Linux 内核源码

3.2.1 代码量及大小

1、代码量

- 2020 年 1 月 1 日，Linux 内核 Git 源码树中的代码达到了 2780 万行。
- phoronix 网站统计了 Linux 内核在进入 2020 年时的一些源码数据并作了总结。从统计数据来看，Linux 内核源码树共有：27852148 行（包括文档、Kconfig 文件、树中的用户空间实用程序等）、887925 次 commit、21074 位不同的作者，2780 万行代码分布在 66492 个文件中。

GitStats - linux

[General](#)[Activity](#)[Authors](#)[Files](#)[Lines](#)[Tags](#)**Project name:**

linux

Generated:

2020-01-01 09:00:10 (in 8567 seconds)

Generator:[GitStats](#) (version 55c5c28), git version 2.20.1, gnuplot 5.2 patchlevel 6**Report Period:**

1969-12-31 19:00:01 to 2085-06-18 11:57:19

Age:

42173 days, 5474 active days (12.98%)

Total Files:

66492

Total Lines of Code:

27852148 (52733303 added, 24881155 removed)

Total Commits:

887925 (average 162.2 commits per active day, 21.1 per all days)

Authors:

21074 (average 42.1 commits per author)

2、代码大小

以 linux-4.1.15 为例，整个内核源码一共约 793M。

模块	大小
/drivers	驱动相关的代码占了大概一半，大约 380M
/arch	体系相关的代码大约 134M
/net	网路子系统相关的代码 26M
/fs	文件系统相关的代码 37M
/kernel	linux 内核核心代码大约 6.8M

3.2.2 获取内核源代码

1、下载源码

Linux 内核官方网站: <https://www.kernel.org/>

官网下载经常速度太慢, 无法下载, 提供 2 个国内镜像下载源

<https://mirror.bjtu.edu.cn/kernel/linux/kernel/>

<http://ftp.sjtu.edu.cn/sites/ftp.kernel.org/pub/linux/kernel/>

2、解压源码

下载 linux-4.4.131.tar.gz 包, 然后解压

```
$ tar zxvf linux-x.y.z.tar.gz
```

内核源代码通常都会安装到/usr/src/linux 下, 但在开发的时候最好不要使用这个源代码树, 因为针对你的 C 库编译的内核版本通常也链接到这里的。

3.2.3 版本号与内核补丁

1、内核版本号

内核版本号由 3 个数字组成: A.B.C。各数字含义如下:

A: 内核主版本号。这是很少发生变化, 只有当发生重大变化的代码和内核发生才会发生。

B: 内核次版本号。是指一些重大修改的内核。**偶数表示稳定版本; 奇数表示开发中版本。**

C: 内核修订版本号。是指轻微修订的内核。这个数字当有安全补丁, bug 修复, 新的功能或驱动程序, 内核便会有变化。

查看 Linux 内核版本相关命令

```
uname -a
```

```
uname -r
```

```
cat /proc/version
```

2、内核补丁

1) 打补丁

```
cd linux-2.6.10
patch -p1 < ../2.6.13.patch
说明：-p1 忽略一级目录
```

2) 恢复

```
patch -R -p1 < ../2.6.13.patch
```

3) 制作补丁

```
diff -Nur linux-2.6.10 linux-2.6.13 > 2.6.13.patch
说明：制作 linux-2.6.10 到 2.6.13 的补丁
```

3.2.4 内核源码文件及目录

1、源码目录

1) 主要目录说明

目录	说明
arch	包含和硬件体系结构相关的代码，每个架构的 CPU 都对应一个目录，如 i386、arm、arm64、powerpc、mips 等。
arch/mach	具体的 machine/board 相关的代码。
arch/boot/dts	对应开发平台的设备树文件。
arch/include/ asm	体系结构相关的头文件。
drivers	设备驱动（设备驱动占了内核 50%的代码量），每一类的驱动对应一个子目录，如 drivers/block 为块设备驱动、drivers/char 为字符设备驱动程序、driver/mtd 为 NOR Flash/Nand Flash 等存储设备的驱动。
include	与系统相关的头文件。
lib	实现需要在内核中使用的公用的库函数，例如 printk 等。与处理器相关的库函数代码位于 arch/*/lib 目录下。

init	内核初始化代码。内核引导后运行的第一个函数 <code>start_kernel()</code> 就位于 <code>init/main.c</code> 文件中。
------	---

2) 内核子系统目录说明

目录	说明
kernel	Linux 内核的核心代码，包含了进程调度子系统，以及和进程调度相关的模块。而和处理器相关的部分代码放在 <code>arch/*/kernel</code> 目录下。
ipc	IPC（进程间通信）子系统。
mm	内存管理子系统。而和处理器相关的一部分代码放在 <code>arch/*/mm</code> 目录下。
fs	VFS 子系统。
net	不包括网络设备驱动的网络子系统。（实现各种常见的网络协议）
block	不包括块设备驱动的块设备子系统。（管理块设备的代码）
sound	音频相关的驱动及子系统，可以看作“音频子系统”。

3) 其他目录说明

目录	说明
crypto	加密、解密相关的库函数。
security	提供安全特性（SELinux）。
virt	提供虚拟化技术（KVM 等）的支持。
usr	用于生成 <code>initramfs</code> 的代码。（实现用于打包和压缩的 <code>cpio</code> 等）
firmware	保存用于第三方设备的固件。
samples	一些示例代码。
tools	一些常用工具，如性能剖析、自测试等。

scripts	用于内核编译的配置文件、脚本等。（Kconfig, Kbuild, Makefile）
Documentation	内核帮助文档。（文档手册相关目录，想了解 Linux 某个功能模块或驱动架构的功能，先在 Documentation 目录中查找有没有对应的文档）

2、源码自带文件

1）重要文件（配置文件）

文件名	说明
Makefile	Linux 顶层 Makefile
Kbuild	Makefile 会读取此文件
Kconfig	图形化配置界面的配置文件

2）其他文件

名称	说明
README	内核说明文档。
.gitignore	git 工具相关文件
.mailmap	邮件列表
COPYING	版权声明
CREDITS	Linux 贡献者
MAINTAINERS	维护者名单
REPORTING-BUGS	BUG 上报指南

3）编译生成文件

文件	功能
.config	Linux 最终使用的配置文件，编译 Linux 的时候会读取此文件中的配置信息。最终根据配置信息来选择编译 Linux 哪些模

	块，哪些功能
System.map	符号表
vmlinux	编译出来的、未压缩的 ELF 格式 Linux 内核文件
vmlinux.o	编译生成中间文件
Module.xx modules.xx	一系列文件，和模块有关

重点回顾

- Linux 系统是什么
- Linux 内核是什么
- Linux 系统系统结构图（重点）
- Linux 内核源码目录结构

思考练习

- Linux 发行版、Linux 内核的区别
- 下载内核并浏览内核目录结构

第二章 《Linux 系统调用及文件 IO 编程》

课程目标

- 能够理解文件描述符的用途
- 能够理解什么是系统调用
- 能够完成 Linux 文件 IO 编程实验

01 文件描述符

1.1 Linux 中的文件

1.1.1 Linux 一切皆文件

与 windows 不同，Linux 操作系统都是基于文件概念的（Linux 一切皆文件）。比如：

- 存储数据的文件，可以运行的二进制程序，层次化的目录结构等等。这些文件都是使用基于磁盘硬件的文件系统（比如 ext2/3/4，xfs 等）来管理的，就是说它们都是存储在真实磁盘设备上的。在 Linux 中大部分文件都是这种类型的。
- 操作设备的文件：文件是以字符序列构成的信息载体，根据这一点，可以把 I/O 设备当做文件来处理。因此，与磁盘上的普通文件进行交互所用的同一系统调用可以直接用于 I/O 设备。这样大大简化了系统对不同设备的处理，提高了效率。
- 还有一部分文件是虚拟文件，并不存储在真实的磁盘硬件设备上，例如：sys，proc，cgroup 等。这些虚拟机文件是内核运行时生成的，从而提供了从用户态通过 VFS 来和内核态通信的方式。

Everything is a file，虽然有一些例外，但在 Linux 上大部分资源确实都是文件，而且都是通过 VFS 来访问的。

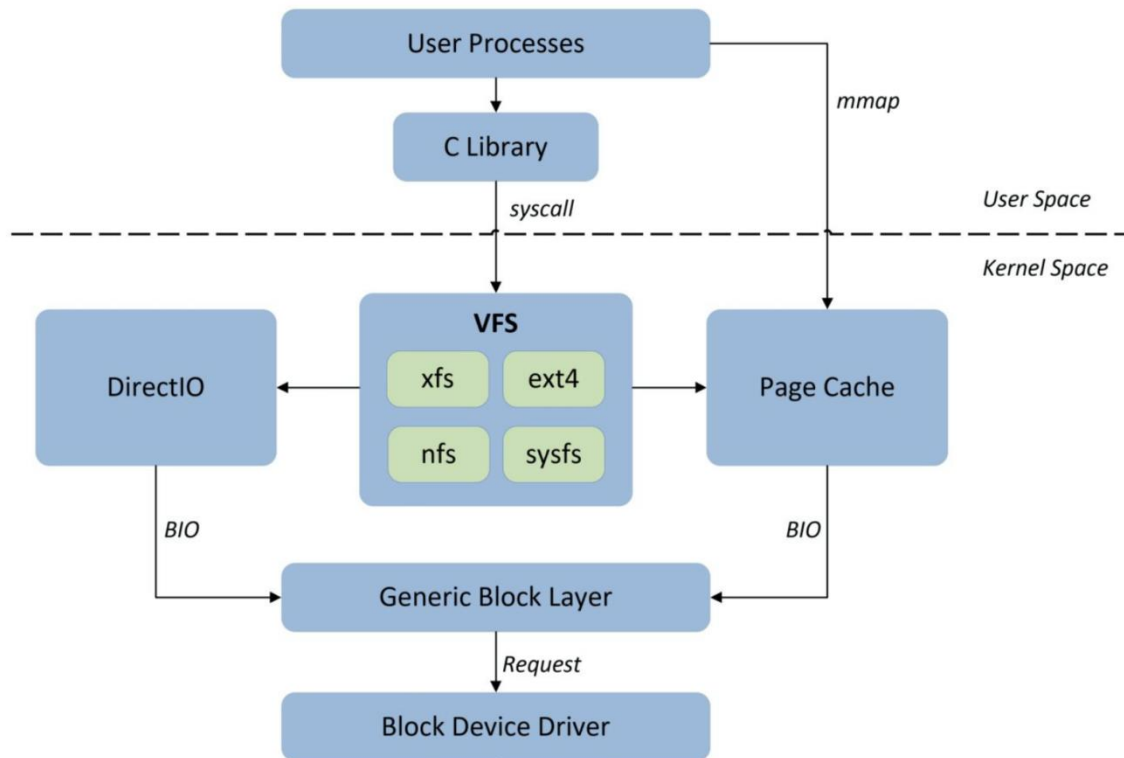
1.1.2 Linux 系统的文件类型

普通文件（regular file）	<p>普通文件（regular file）：就是一般存取的文件，由 <code>ls -al</code> 显示出来的属性中，第一个属性为 <code>[-]</code>，例如 <code>[-rwxrwxrwx]</code>。另外，依照文件的内容，又大致可以分为：</p> <ul style="list-style-type: none">• 1、纯文本文件（ASCII）：这是 Unix 系统中最多的一种文件类型，之所以称为纯文本文件，是因为内容可以直接读到的数据，例如数字、字母等等。设置文件几乎都属于这种文件类型。举例来说，使用命令“<code>cat ~/.bashrc</code>”就可以看到该文件的内容（<code>cat</code> 是将文件内容读出来）。• 2、二进制文件（binary）：系统其实仅认识且可以执行二进制文件（binary file）。Linux 中的可执行文件（脚本，文本方式的批处理文件不算）就是这种格式的。举例来说，命令 <code>cat</code> 就是一个二进制文件。• 3、数据格式的文件（data）：有些程序在运行过程中，会读取某些特定格式的文件，那些特定格式的文件可以称为数据文件（data file）。举例来说，Linux 在用户登入时，都会将登录数据记录在 <code>/var/log/wtmp</code> 文件内，该文件是一个数据文件，它能够通过 <code>last</code> 命令读出来。但使用 <code>cat</code> 时，会读出乱码。因为它是属于一种特殊格式的文件。
目录文件	目录文件（directory）：就是目录，第一个属性为 <code>[d]</code> ，例如

(directory)	[drwxrwxrwx]。
连接文件 (link)	连接文件 (link)：类似 Windows 下面的快捷方式。第一个属性为 [l]，例如 [lrwxrwxrwx]。
设备文件 (device)	<p>设备与设备文件 (device)：与系统外设及存储等相关的一些文件，通常都集中在 /dev 目录。通常又分为两种：</p> <ul style="list-style-type: none">• 块设备文件：就是存储数据以供系统存取的接口设备，简单而言就是硬盘。例如一号硬盘的代码是 /dev/hda1 等文件。第一个属性为 [b]。• 字符设备文件：即串行端口的接口设备，例如键盘、鼠标等等。第一个属性为 [c]。
套接字 (sockets)	套接字 (sockets)：这类文件通常用在网络数据连接。可以启动一个程序来监听客户端的要求，客户端就可以通过套接字来进行数据通信。第一个属性为 [s]，最常在 /var/run 目录中看到这种文件类型。
管道 (FIFO pipe)	FIFO 也是一种特殊的文件类型，它主要的目的是，解决多个程序同时存取一个文件所造成的错误。FIFO 是 first-in-first-out (先进先出) 的缩写。第一个属性为 [p]。

1.1.3 虚拟文件系统 (VFS)

为了支持各种各样文件系统，Linux 在用户进程和文件系统实例中间引入了一个抽象层，对不同文件系统的访问都使用相同的方法，并提供了文件的统一视图。不同的文件系统的底层实现方式可能有很大的差异，但 VFS 并不关心这些。通过提供公共组件和统一框架，VFS 对上层系统调用屏蔽了具体文件系统实现之间的差异性，为所有文件的访问提供了相同的 API，并遵循相同的调用语义。



1.2 文件描述符

1.2.1 什么是文件描述符

文件描述符是一个非负整数，当打开一个已有文件或创建一个新文件时，内核向进程返回一个文件描述符。当读、写一个文件时，用 `open` 或 `create` 返回的文件描述符标识该文件，将其作为参数传给 `read` 或 `write`。

在 POSIX 应用程序中，标准输入、标准输出、标准错误使用整数 0、1、2 表示，整数 0、1、2 被替换成符号常数 `STDIN_FILENO`、`STDOUT_FILENO` 和 `STDERR_FILENO`。这些常数都定义在头文件 `<unistd.h>` 中。文件描述符的范围是 0-`OPEN_MAX`。Linux 为 1024。

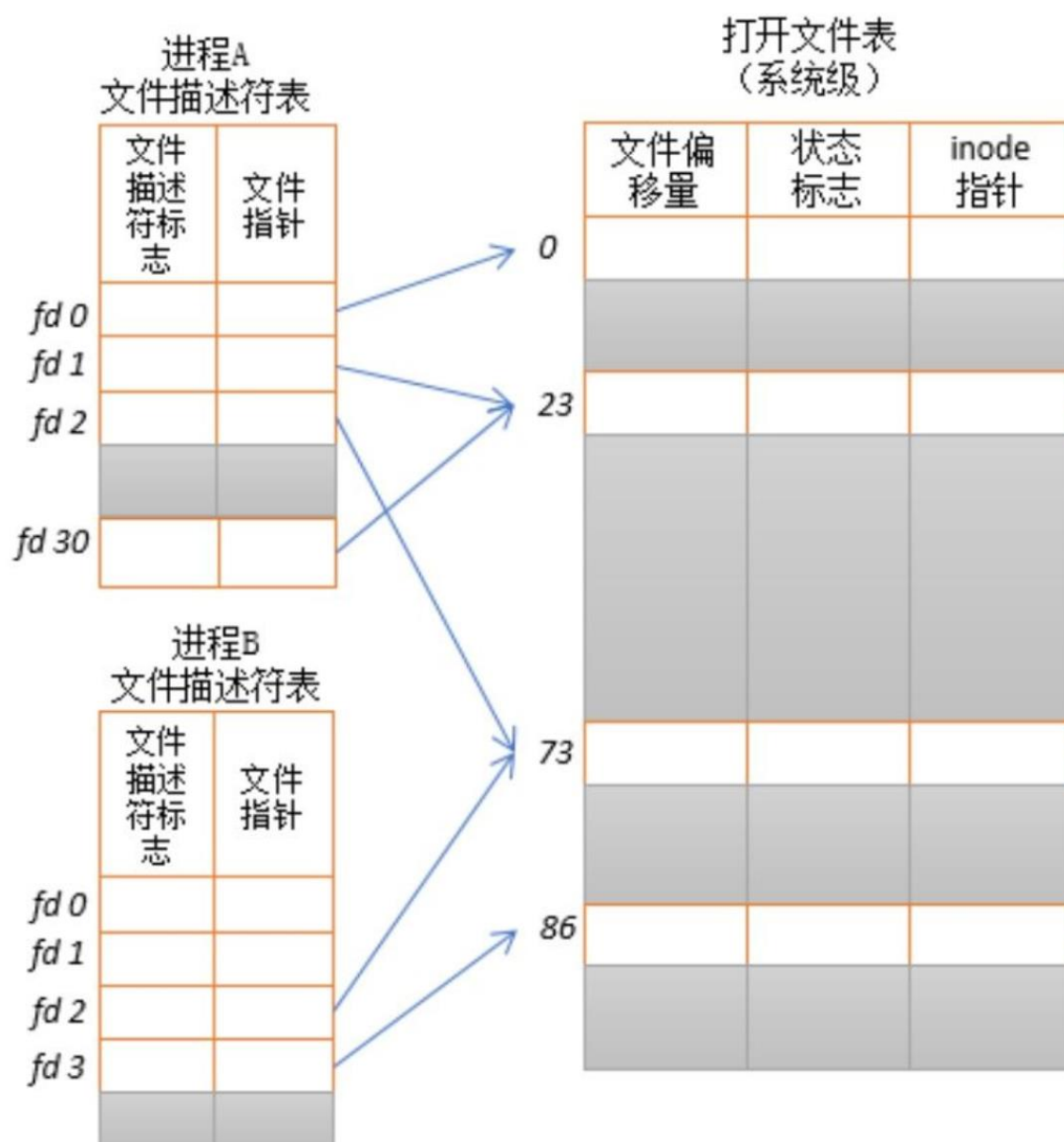
举例：每当进程用 `open()` 函数打开一个文件，内核便会返回该文件的文件操作符（一个非负的整形值），此后所有对该文件的操作，都会以返回的 `fd` 文件操作符为参数。

```

fd = open(pathname, flags, mode)
// 返回了该文件的 fd
rlen = read(fd, buf, count)
// IO 操作均需要传入该文件的 fd 值
wlen = write(fd, buf, count)
status = close(fd)
  
```

1.2.2 文件描述符表

Linux 系统中，把一切都看做是文件，当进程打开现有文件或创建新文件时，内核向进程返回一个文件描述符，文件描述符就是内核为了高效管理已被打开的文件所创建的索引，用来指向被打开的文件，所有执行 I/O 操作的系统调用都会通过文件描述符。



02 文件 IO 与标准 IO

2.1 Linux 系统编程

2.1.1 Linux 系统编程概述

典型的嵌入式产品的研发过程有两大步：

- 第一步，让某个系统在硬件上跑起来（系统移植、设备驱动）；
- 第二步，基于这个系统来开发应用程序实现产品功能。

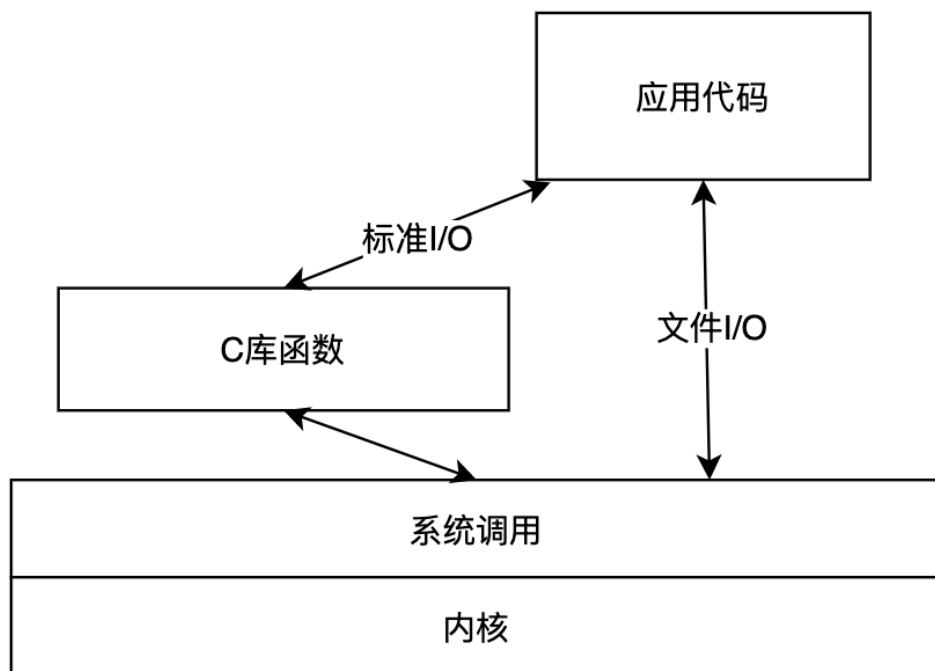
基于系统来开发应用程序实现产品功能，其实就是通过调用系统的 API 和一些库来进行编程。在 Linux 系统上一切皆文件，所以 Linux 系统应用编程主要就是对文件进行操作。

Linux 下对文件操作有两种方式：**系统调用 (system call)** 和 **库函数调用 (Library functions)**。

- 1) 系统调用：系统调用是通向操作系统本身的接口，是面向底层硬件的。通过系统调用，可以使得用户态运行的进程与硬件设备(如 CPU、磁盘、打印机等)进行交互，是操作系统留给应用程序的一个接口。
- 2) 库函数：库函数 (Library function) 是把函数放到库里，供别人使用的一种方式。方法是把一些常用到的函数编完放到一个文件里，供不同的人进行调用。一般放在 .lib 文件中。

其中，**系统调用**对应的是**文件 I/O** 操作、**库函数**对应的是**标准 I/O** 操作。

- 文件 I/O：直接调用内核提供的系统调用函数
- 标准 I/O：间接调用系统调用函数



2.1.2 文件 I/O 与标准 I/O

Linux 文件 I/O 操作有两套大类的操作方式：不带缓存的文件 I/O 操作，带缓存的标准 I/O 操

作。

不带缓存的属于直接调用系统调用（system call）的方式，高效完成文件输入输出。它以文件标识符（整型）作为文件唯一性的判断依据。这种操作不是 ASCII 标准的，与系统有关，移植有一定的问题。带缓存的是在不带缓存的基础之上封装了一层，维护了一个输入输出缓冲区，使之能跨 OS，成为 ASCII 标准，称为标准 IO 库。

不带缓存的方式频繁进行用户态和内核态的切换，高效但是需要程序员自己维护；带缓冲的方式因为有了缓冲区，不是非常高效，但是易于维护。由此，不带缓冲区的通常用于文件设备的操作（文件 IO），而带缓冲区的通常用于普通文件的操作（标准 IO）。

	文件 IO	标准 IO
定义	文件 I/O 是不带缓冲的 IO，是系统提供的 API。	标准 I/O 是带缓冲的 IO，是一个标准 C 函数库。
头文件	<unistd.h>	<stdio.h>
函数	man 2 open	man 3 fopen
标准	POSIX 标准	ANSI C 标准
缓冲	通过文件 I/O 读写文件时，每次操作都会执行相关系统调用。这样处理的好处是直接读写实际文件，坏处是频繁的系统调用会增加系统开销。	标准 I/O 可以看成是在文件 I/O 的基础上封装了缓冲机制。先读写缓冲区，必要时再访问实际文件，从而减少了系统调用的次数。
文件	文件 I/O 中用文件描述符表现一个打开的文件，可以访问不同类型的文件如普通文件、设备文件和管道文件等。	标准 I/O 中用 FILE（流）表示一个打开的文件，通常只用来访问普通文件。

说明：Linux 中使用的是 glibc 库，它是标准 C 库的超集。不仅包含 ANSI C 中定义的函数，还包括 POSIX 标准中定义的函数。因此，Linux 下既可以使用标准 I/O，也可以使用文件 I/O。

2.2 IO 常见函数

2.2.1 文件 IO 常见函数

常用的文件 IO 函数有 5 个：open、close、read、write、ioctl，可以通过 man 2 查看对应参数。

1、open

open 用于创建一个新文件或打开一个已有文件，返回一个非负的文件描述符 fd。

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

//成功返回文件描述符，失败返回-1
int open(const char *pathname, int flags, ... /* mode_t mode */);
```

flags 参数一般在 O_RDONLY、O_WRONLY 和 O_RDWR 中选择指定一个，还可以根据需要或上以下常值：

- O_CREAT：若文件不存在则创建它，此时需要第三个参数 mode，mode 可设的值及含义如下图所示。
- O_APPEND：每次写时都追加到文件的尾端
- O_NONBLOCK：如果 pathname 对应的是 FIFO、块特殊文件或字符特殊文件，则该命令使 open 操作及后续 IO 操作设定为非阻塞模式

2、close

close 用于关闭一个已打开文件。

```
#include <unistd.h>

//成功返回 0，失败返回-1
int close(int fd);
```

进程终止时，内核会自动关闭它所有的打开文件，应用程序经常利用这一点而不显式关闭文件。

3、read

read 用于从文件中读数据。

```
#include <unistd.h>

//成功返回读到的字节数；若读到文件尾则返回 0；失败返回-1
ssize_t read(int fd, void *buf, size_t count);
```

read 操作从文件的当前偏移量处开始，在成功返回之前，文件偏移量将增加实际读到的字

节数。

有几种情况可能导致实际读到的字节数少于要求读的字节数：

- 读普通文件时，在读到要求字节数之前就到达了文件尾。例如，离文件尾还有 30 字节，要求读 100 字节，则 read 返回 30，下次在调用 read 时会直接返回 0；
- 从网络读时，网络中的缓冲机制可能造成返回值少于要求读的字节数。

4、write

write 用于向文件写入数据。

```
#include <unistd.h>

//成功返回写入的字节数，失败返回-1
ssize_t write(int fd, const void *buf, size_t count);
```

- write 的返回值通常与参数 count 相同，否则表示出错；
- 对于普通文件，write 操作从文件的当前偏移量处开始；
- 若指定了 O_APPEND 选项，则每次写之前先将文件偏移量设置到文件尾；
- 成功写入之后，文件偏移量增加实际写的字节数。

5、ioctl

ioctl 提供了一个用于控制设备及其描述符行为和配置底层服务的接口。

```
#include <sys/ioctl.h>

//出错返回-1，成功返回其他值
int ioctl(int fd, int cmd, ...);
```

- ioctl 对描述符 fd 引用的对象执行由 cmd 参数指定的操作；
- 每个设备驱动程序都可以定义它自己专用的一组 ioctl 命令。

2.2.2 标准 IO 常见函数

常用的标准 IO 函数分为以下几大类：

- 打开和关闭流
- 定位流
- 读写流：包括文本 IO、二进制 IO 和格式化 IO 三种

1、打开和关闭流

```
//成功返回文件指针，失败返回 NULL
FILE *fopen(const char *pathname, const char *type);

//成功返回 0，失败返回 EOF
void fclose(FILE *fp);
```

fopen 打开由 pathname 指定的文件，type 指定读写方式：

- 使用字符 b 作为 type 的一部分，使得标准 IO 可以区分文本文件和二进制文件；
- Linux 内核不区分文本和二进制文件，因此在 Linux 系统下使用字符 b 实际上没有作用，只读、只写、读写分别指定为“r”、“w”、“rw”即可；
- Windows 中，文本文件只读、只写、读写分别为“r”、“w”、“rw”，二进制文件只读、只写、读写分别为“rb”、“wb”、“rb+”；
- 只读方式要求文件必须存在，只写或读写方式会在文件不存在时创建。

fclose 关闭文件，关闭前缓冲区中的输出数据会被冲洗，输出数据则丢弃。

2、定位流

流的定位类似于系统调用 IO 中获取当前文件偏移量，ftell 和 fseek 函数可用于定位流。

```
//成功返当前文件位置，出错返回-1
int ftell(FILE *fp);

//成功返回 0，失败返回-1
void fseek(FILE *fp, long offset, int whence);
```

offset 和 whence 含义及可设的值与系统调用 IO 中的 lseek 相同，不再赘述，但如果是在非 Linux 系统，则有一点需要注意：

- 对于二进制文件，文件位置严格按照字节偏移量计算，但对于文本文件可能并非如此；
- 定位文本文件时，whence 必须是 SEEK_SET，且 offset 只能是 0 或 ftell 返回值。

3、文本 IO

文本 IO 有两种：

- 每次读写一个字符
- 每次读写一行字符串

1) 每次读写一个字符


```

/*
 * 每次读写一个字符
 */

//成功返回下一个字符，到达文件尾或失败返回 EOF
int getc(FILE *fp);          //可能实现为宏，因此不允许将其地址作为参数传给另一个函数，因为宏没有地址
int fgetc(FILE *fp);         //一定是函数
int getchar();               //等价于 getc(stdin)

//成功返回 c，失败返回 EOF
int putc(int c, FILE *fp);    //可能实现为宏，因此不允许将其地址作为参数传给另一个函数，因为宏没有地址
int fputc(int c, FILE *fp);   //一定是函数
int putchar(int c);           //等价于 putc(c, stdout)

```

2) 每次读写一行字符串

```

/*
 * 每次读写一行字符串
 */

//成功返回 str，到达文件尾或失败返回 EOF
char *fgets(char *str, int n, FILE *fp); //从 fp 读取直到换行符（换行符也读入），str 必须以'\0' 结尾，故包括换行符在内不能超过 n-1 个字符

//成功返回非负值，失败返回 EOF
int fputs(const char *str, FILE *fp);     //将字符串 str 输出到 fp，str 只要求以'\0' 结尾，不一定含有换行符

```

4、二进制 IO

二进制 IO 就是 fread 和 fwrite。

```

//返回读或写的对象数
size_t fread(void *ptr, size_t size, size_t nobj, FILE *fp);
size_t fwrite(const void *ptr, size_t size, size_t nobj, FILE *fp);

```

二进制 IO 常见的用法包括：

- 读写一个二进制数组
- 读写一个结构

上述两种用法结合起来，还可以实现读写一个结构数组。

```

struct Item
{
    int id;
    char text[100];
};

int data[10];
struct Item item;
struct Item item[10];

//读写二进制数组
fread(&data[2], sizeof(int), 4, fp);
fwrite(&data[2], sizeof(int), 4, fp);

//读写结构
fread(&item, sizeof(item), 1, fp);
fwrite(&item, sizeof(item), 1, fp);

//读写结构数组
fread(&item, sizeof(item[0]), sizeof(item) / sizeof(item[0]), fp);
fwrite(&item, sizeof(item[0]), sizeof(item) / sizeof(item[0]), fp);

```

5、格式化 IO

格式化 IO 包括输入函数族和输出函数族，我们剔除了不常用的与文件指针 fp、文件描述符 fd 相关的 API，仅保留常用的 3 个输出函数和 2 个输入函数。

```

//成功返回输出或存入 buf 的字符数（不含'\0'），失败返回负值
int printf(const char *format, ...);
int sprintf(char *buf, const char *format, ...);
int snprintf(char *buf, size_t n, const char *format, ...);

```

- sprintf 和 snprintf 会自动在 buf 末尾添加字符串结束符'\0'，但该字符不包括在返回值中
- snprintf 要求调用者保证缓冲区 buf 长度 n 足够大

```

//成功返回输入的字符数，到达文件尾或失败返回 EOF
int scanf(const char *format, ...);
int sscanf(const char *buf, const char *format, ...);

```

PS: sscanf 在实际工程中有一个实用的小技巧：串口接收的一条报文，可以根据串口协议，使用 sscanf 提取各个字段，从而快速便捷的进行报文解析。

03 系统调用与 POSIX 标准

3.1 系统调用

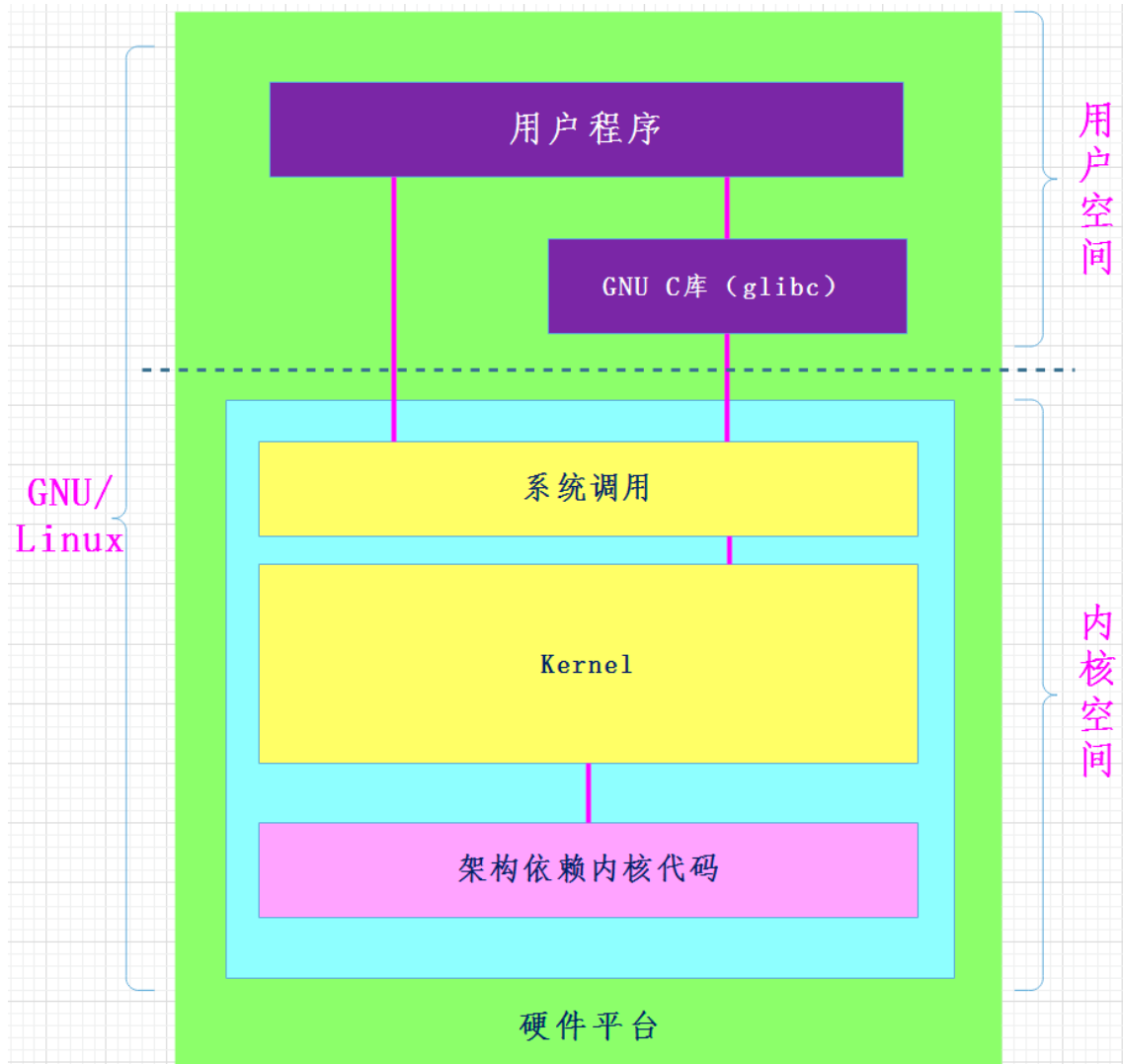
3.1.1 什么是系统调用

操作系统负责管理和分配所有的计算机资源。为了更好地服务于应用程序，操作系统提供了一组特殊接口——系统调用。通过这组接口用户程序可以使用操作系统内核提供的各种功能。例如分配内存、创建进程、实现进程之间的通信等。

3.1.2 什么是库函数

库函数可以说是对系统调用的一种封装，因为系统调用是面对的是操作系统，系统包括 Linux、Windows 等，如果直接系统调用，会影响程序的移植性，所以这里使用了库函数，比如说 C 库，这样只要系统中安装了 C 库，就都可以使用这些函数，比如 `printf()` `scanf()` 等，C 库相当于对系统函数进行了翻译，使我们的 APP 可以调用这些函数。

3.1.3 系统调用与库函数



如上图所示：

- (1) 库函数是 C 语言或应用程序的一部分，而系统调用是内核提供给应用程序的接口，属于系统的一部分。
- (2) 库函数在用户地址空间执行，系统调用是在内核地址空间执行，库函数运行时间属于用户时间，系统调用属于系统时间，库函数开销较小，系统调用开销较大。
- (3) 系统调用依赖于平台，库函数并不依赖。

	函数库调用	系统调用
定义差别	在所有的ANSI C编译器版本中，C库函数是相同的	各个操作系统的系统调用是不同的
调用差别	它调用函数库中的一段程序（或函数）	调用系统内核的服务
与系统关系	与用户程序相联系	操作系统的一个入口点
运行空间	在用户地址空间执行	在内核地址空间执行
运行时间	它的运行时间属于“用户时间”	它的运行时间属于“系统”时间
开销	属于过程调用，开销较小	需要在用户空间和内核上下文环境间切换，开销较大
个数	在C函数库libc中有大约300个函数	在LNIX中大约有100多个系统调用
典型调用	fprintf、fread、malloc	chdir、fork、write、brk

3.2 POSIX 标准

3.2.1 什么是 POSIX 标准

POSIX 是可移植操作系统接口（Portable Operating System Interface of UNIX）的缩写。它是一个 IEEE 1003.1 标准，其定义了应用程序（以及命令行 Shell 和实用程序接口）和 UNIX 操作系统之间的语言接口。

该标准的目的是定义了标准的基于 UNIX 操作系统的系统接口和环境来支持源代码级的可移植性。

3.2.1 为什么需要 POSIX 标准

不同内核提供的系统调用是不同的。

例如：创建进程，linux 下是 fork 函数，windows 下是 creatprocess 函数。我在 linux 下写一个程序，用到 fork 函数，那么这个程序该怎么往 windows 上移植？我需要把源代码里的 fork 通通改成 creatprocess，然后重新编译...

posix 标准的出现就是为了解决这个问题。

例如：linux 和 windows 都按 posix 标准实现了创建进程的函数，linux 把 fork 函数封装成 posix_fork（随便起的名字），windows 把 creatprocess 函数也封装成 posix_fork，

都声明在 `unistd.h` 里。这样，程序员编写普通应用时候，只用包含 `unistd.h`，调用 `posix_fork` 函数，程序就在源代码级别可移植了。

重点回顾

- 文件描述符
- 系统调用
- 文件 IO 操作

思考练习

- 【实验 1】编写一个用户程序打开创建文件，并写入 “hello world” 到文件中

第三章 《Linux 内核模块编程》

课程目标

- 1、理解内核编程与用户态编程的区别
- 2、理解内核模块的作用
- 3、掌握内核模块的编程、编译、运行方法

01 内核模块概述

1.1 什么是内核模块

1.1.1 内核模块的作用

- 内核模块是具有独立功能的程序。它可以被单独编译，但是不能单独运行，它的运行必须被链接到内核作为内核的一部分在内核空间中运行。
- 模块编程和内核版本密切相连，因为不同的内核版本中某些函数的函数名会有变化。因此模块编程也可以说是内核编程。
- 模块本身不被编译进内核映像，从而控制了内核的大小； 模块一旦被加载，就和内核中的其他部分完全一样。

1.1.2 内核模块编程注意事项

- 内核的 C 语言标准 GNU C

- 内核中不能使用 libc 库
- 内核中不要使用浮点运算
- 内核代码尽量写的可移植性强
- 内核编程中随时并发和竞态的情况

1.2 编程三要素

以 C 语言的第一个程序 “hello world” 为例

1.2.1 编程

入口和 API

```
#include <stdio.h>
int main(void)
{
    printf("hello world\n");
    return 0;
}
```

1.2.2 编译

目标平台可执行的文件

```
gcc -o hello hello.c
```

1.2.3 运行

把执行文件放到目标平台运行起来

```
./hello
```

02 内核模块编程三步法

2.1 编程：内核模块编程

2.1.1 第一个内核程序

几乎所有程序员在学习一门新语言时都会编写的程序：输出 “hello world”，我们本节课的目标是在内核态编程输出 “hello world”，下面是一段完整的内核代码示例，然后我们逐步拆解代码，了解内核态编程的世界。

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

static int test_init(void)
{
    printk("hello world\n");
    return 0;
}

static void test_exit(void)
{
    printk("bye\n");
}

module_init(test_init);
module_exit(test_exit);

MODULE_LICENSE("GPL");
```

2.1.2 内核程序分析

1、加载/卸载模块

1) 模块加载: module_init

```
int test_init(void)
{
    ...
    return 0;
}

module_init(test_init);
```

说明

- 作用：告诉内核你编写模块程序从哪里开始执行。
- 入口：module_init() 中的参数就是入口函数的函数名。

2) 模块卸载: module_exit

```
void test_exit(void)
{
    ...
}
```



```
}  
module_exit(test_exit);
```

说明

- 作用：告诉内核你编写模块程序从哪里离开。
- 出口：module_exit() 中的参数名就是卸载函数的函数名。

2、头文件

```
#include <linux/init.h> // init&exit 相关宏  
#include <linux/module.h> //所有模块都需要的头文件
```

注意：由于内核层编程和用户层编程所用的库函数不一样，所以它的头文件也和我们在用户层编写程序时所用的头文件也不一样。

内核层头文件的位置：/usr/src/linux-x.y.z/include/

用户层头文件的位置：/usr/include/

3、打印函数

printk 是内核态信息打印函数，功能和标准 C 库的 printf 类似。不同的是 printk 可以附加不同的日志级别，从而可以根据消息的严重程度分类。

1) 内核日志

printk 打印的信息通过内核日志查看

方法一：运行命令 dmesg

方法二：cat /var/log/messages

2) 显示级别

查看当前控制台的显示级别：cat /proc/sys/kernel/printk

该文件有 4 个数字值，它们根据日志记录消息的重要性，定义将其发送到何处，上面显示的 4 个数据分别对应如下：

- 控制台日志级别：优先级高于该值得消息将被打印到控制台；
- 默认的消息日志级别：将用该优先级来打印没有优先级的消息；
- 最低的控制台日志级别：控制台日志级别可被设置的最小值（最高优先级）；

- 默认的控制台日志级别：控制台日志级别的缺省值。

以上的数值设置，数值越小，优先级越高。

上面这个 4 个值的定义在 kernel/printk.c，如下：

```
int console_printk[4] = {
    CONSOLE_LOGLEVEL_DEFAULT, /* console_loglevel */
    MESSAGE_LOGLEVEL_DEFAULT, /* default_message_loglevel */
    CONSOLE_LOGLEVEL_MIN,     /* minimum_console_loglevel */
    CONSOLE_LOGLEVEL_DEFAULT, /* default_console_loglevel */
};
```

如果需要在系统的终端处修改显示级别，可以通过下面的命令：

```
echo 7 4 1 7 > /proc/sys/kernel/printk
```

3) 打印级别

内核通过 printk() 输出的信息具有日志级别，内核中提供了 8 种不同的日志级别，在 linux/kernel_levels.h 中有相应的宏定义，如下：

```
KERN_EMERG "<0>": 紧急情况
KERN_ALERT "<1>": 需要立即被注意到的错误
KERN_CRIT  "<2>": 临界情况
KERN_ERR   "<3>": 错误
KERN_WARNING"<4>": 警告
KERN_NOTICE "<5>": 注意
KERN_INFO  "<6>": 非正式的消息
KERN_DEBUG "<7>": 调试信息(冗余信息)
```

因此，printk() 可以这样来使用：printk(KERN_INFO"Hello World\n");

当未指定日志级别的 printk() 采用默认的级别是 DEFAULT_MESSAGE_LOGLEVEL，这个宏定义为整数 4。

如果想要在内核启动过程中打印少的信息，可以根据自己的需要在 kernel/printk.c 中修改以上的数值，重新编译后，烧写内核镜像。

4、模块声明

```
MODULE_LICENSE("GPL");
```

模块声明描述内核模块的许可权限，如果不声明 LICENSE，模块被加载时，将收到内核的警告。

声明习惯上放在文件最后，可以用 modinfo 查看模块声明信息。

2.2 编译：内核模块编译

2.2.1 模块编译 Makefile

```
obj-m := file.o

KDIR :=/lib/modules/$(shell uname -r)/build

all:
    make -C $(KDIR) M=$(PWD) modules
clean:
    make -C $(KDIR) M=$(PWD) clean
```

注意：all 和 clean 下面的命令要严格用“tab”

说明：如果是交叉编译，通过 CROSS_COMPILE 指定交叉编译器、ARCH 指定体系结构。例如：make -C \$(KDIR) M=\$(PWD) module CROSS_COMPILE=/xx/gcc-6.3-arm64-linux/bin/aarch64-linux-gnu- ARCH=arm64

2.2.2 Makefile 说明

参数	说明
obj-m	<ul style="list-style-type: none">obj-m := file.o 表示编译连接后将生成 file.ko 模块
KDIR	<ul style="list-style-type: none">/lib/modules/\$(shell uname -r)/build 是内核 Kbuild Makefile 所在路径。Linux 系统使用 Kbuild Makefile 来编译内核或模块。\$(shell uname -r):是调用 shell 命令显示内核版本。
-C \$KDIR and M=\$PWD	<ul style="list-style-type: none">-C 指定内核 Kbuild Makefile 所在路径。M=指定模块所在路径。 <p>详细介绍参考内核文档/kbuild/modules.txt</p>
target	<ul style="list-style-type: none">make modules: 编译模块make clean: 清除模块编译产生的文件，相当于 rm -f *.ko *.o *.mod.o *.mod.c *.symvers *.ordermake modules_install: 拷贝安装模块到指定目录

2.2.3 编译模块

在模块代码和模块 Makefile 所在目录执行 “make”

2.3 运行：内核模块加载

2.3.1 加载/卸载模块

1、加载模块

```
sudo insmod test.ko
```

2、卸载模块

```
sudo rmmmod hello
```

注意：rmmmod 后面要加用 lsmod 查看到的模块名字'test'而不是'test.ko'

2.3.2 查看内核日志

运行命令：dmesg

内核日志显示信息如下：

```
[156596.317933] hello world.  
[156604.933930] hello exit!
```

2.3.3 模块相关命令

查看本机模块：lsmod

查看模块信息：modinfo

重点回顾

- 编程：内核编程的入口、头文件、打印函数
- 编译：模块编译和安装的 Makefile
- 运行：掌握模块相关命令

思考练习

- 作业：根据《实验手册》按编程、编译、运行三步法，开发第一个内核程序。

第四章 《Linux 设备驱动基础与接口实现》

课程目标

- 理解 Linux 设备驱动基础概念（设备类型、设备文件）
- 理解 Linux 字符设备驱动重要结构体
- 理解 Linux 字符设备驱动工作流程
- 掌握 Linux 字符设备驱动注册方法
- 掌握 Linux 字符设备驱动接口实现方法

01 Linux 设备驱动基础

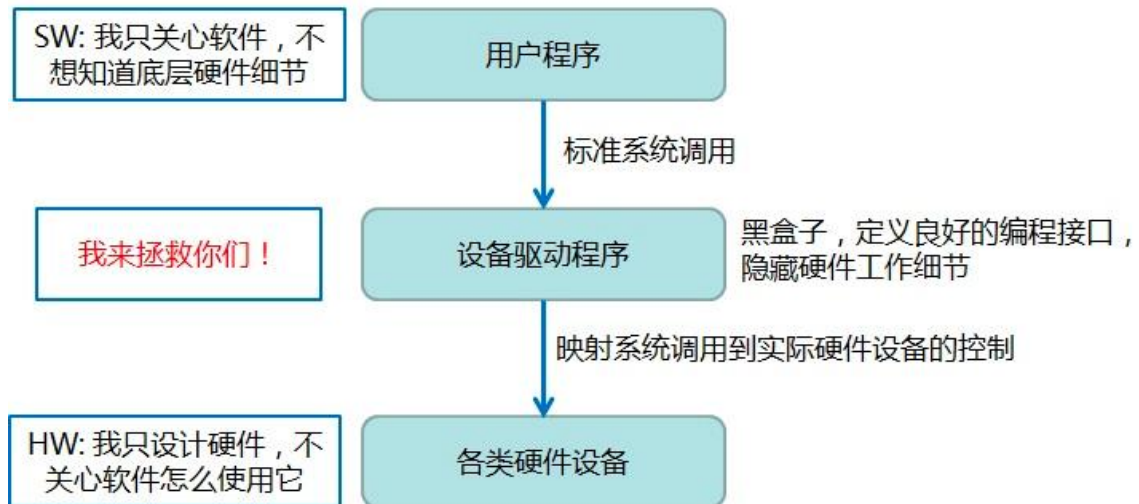
1.1 设备驱动

1.1.1 设备驱动概述

1、为什么需要设备驱动

在任何一个计算机系统中，大至服务器、PC 机，小至手机、mp3/mp4 播放器，无论是复杂的大型服务器系统还是一个简单的流水灯单片机系统都离不开驱动程序的身影。没有硬件的软件是空中楼阁，没有软件的硬件只是一堆废铁，硬件是底层的基础，是所有软件得以运行的平台，代码最终会落实到硬件上的逻辑组合。

硬件与软件之间存在一个驳论：为了快速、优质的完成软件功能设计，应用软件程序工程师不想也不愿关心硬件，而硬件驱动工程师也很难有功夫去处理软件开发中的一些应用。例如：软件工程师在调用 printf 的时候，不许也不用关心信息到底是通过什么样的处理，走过哪些通路显示在该显示的地方，硬件工程师在写完了一个 4*4 键盘驱动后，无需也不必管应用程序在获得键值后做哪些处理及操作。



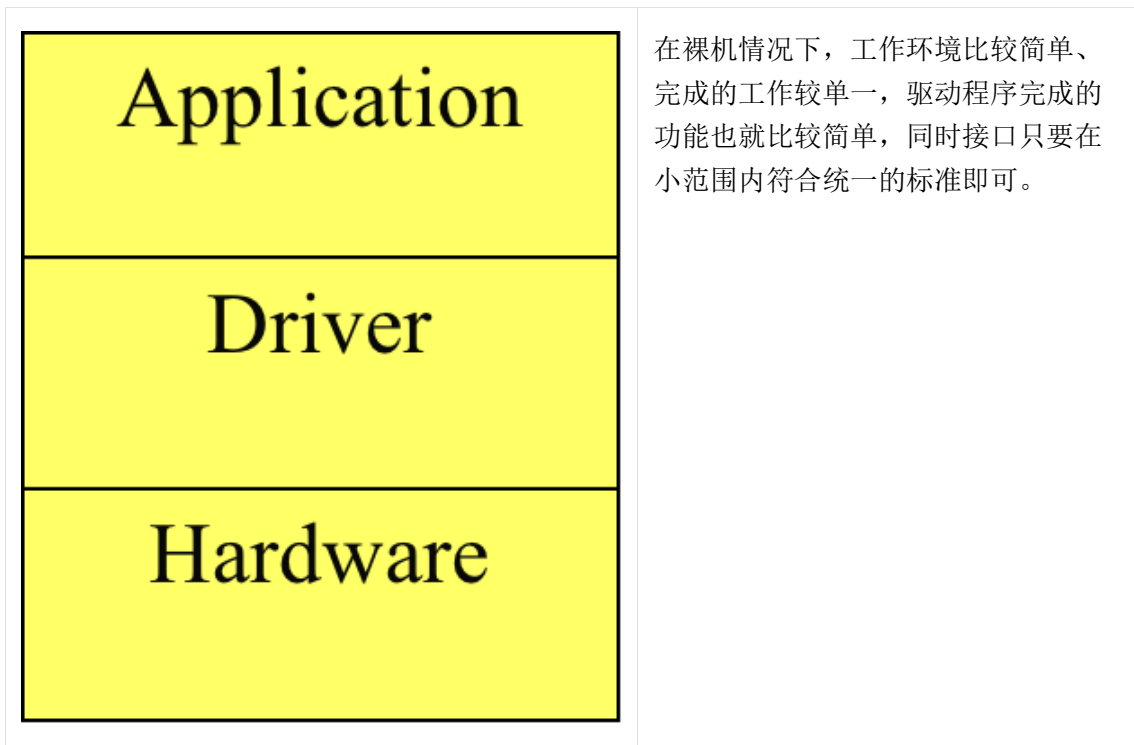
2、什么是设备驱动

软件工程师需要看到一个没有硬件的纯软件世界，硬件必须透明的提供给他，谁来实现这一任务？答案是驱动程序。驱动程序从字面解释就是：“驱使硬件设备行动”。驱动程序直接与硬件打交道，按照硬件设备的具体形式，驱动设备的寄存器，完成设备的轮询、中断处理、DMA 通信，最终让通信设备可以收发数据，让显示设备能够显示文字和画面，让音频设备可以完成声音的存储和播放。

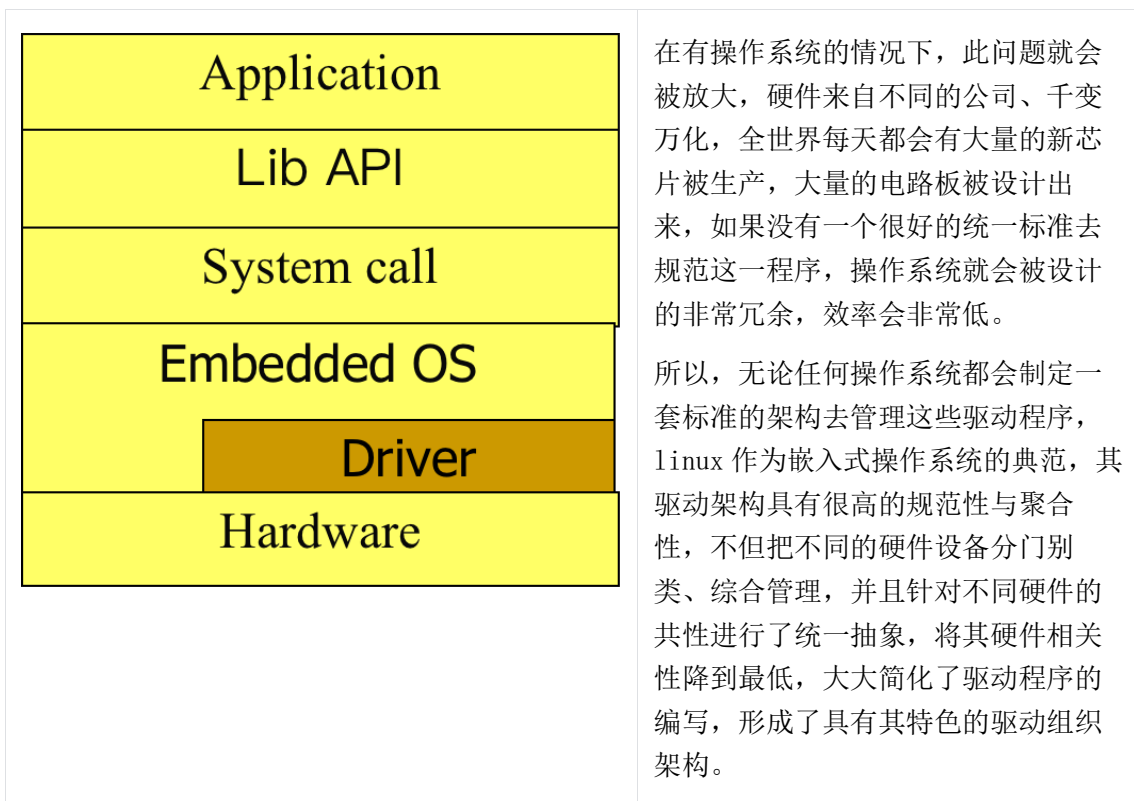
3、如何实现设备驱动

设备驱动程序充当了硬件和软件之间的枢纽，因此驱动程序的表现形式就是一些标准的、事先约定好的 API 函数，驱动工程师只需要去完成相应函数的填充，应用工程师只需要调用相应的接口完成相应的功能。

1) 无操作系统的设备驱动



2) 有操作系统的设备驱动



1.1.2 驱动程序与应用程序

1、驱动程序的职责

- 对设备初始化和释放资源
- 把数据从内核传送到硬件和从硬件读取数据
- 读取应用程序传送给设备文件的数据和回送应用程序请求的数据
- 检测和处理设备出现的错误(底层协议)

2、驱动程序与应用程序的区别

区别类型	区别描述
入口	<ul style="list-style-type: none">• 应用程序以 main 开始• 驱动程序没有 main，它以一个模块初始化函数作为入口
执行	<ul style="list-style-type: none">• 应用程序从头到尾执行一个任务• 驱动程序加载完成注册之后不再运行，等待系统调用
库	<ul style="list-style-type: none">• 应用程序可以使用 glibc 等标准 C 函数库• 驱动程序不能使用标准 C 库

3、驱动程序与应用程序的关系

1、应用程序	<ul style="list-style-type: none">• 应用程序以文件形式访问各种硬件设备，调用一系列函数库，对文件进行操作完成不同功能。
2、函数库	<ul style="list-style-type: none">• 部分函数需要硬件操作或内核的支持，通过系统调用由内核完成对应功能。
3、内核（系统调用）	<ul style="list-style-type: none">• 内核处理系统调用，根据设备文件类型、设备号，调用设备驱动程序；
4、驱动程序	<ul style="list-style-type: none">• 实现对应系统调用 API，对硬件寄存器操作，驱使硬件设备行动。

总结：设备驱动程序为应用程序屏蔽了硬件的细节，这样在应用程序看来，硬件设备只是一个设备文件，应用程序可以像操作普通文件一样对硬件设备进行操作。

1.1.3 固件工程师与应用工程师

Linux 软件工程师大致可分为两个层次：

- Linux 应用软件工程师（Application Software Engineer）

- Linux 固件工程师 (Firmware Engineer)

1、Linux 应用软件工程师 (Application Software Engineer)

主要利用 C 库函数和 Linux API 进行应用程序的编写；从事这方面的开发工作，主要需要掌握符合 Linux Posix 标准的 API 函数及系统调用，Linux 的多任务编程技巧（多进程、多线程、进程间通信、多任务之间的同步互斥等），嵌入式数据库，QT 图形编程等。

2、Linux 固件工程师 (Firmware Engineer)

主要进行 Bootloader、Linux 的移植及 Linux 设备驱动程序的开发工作。

一般而言，固件工程师的要求要高于应用软件工程师，而其中的 Linux 设备驱动编程又是 Linux 程序设计中比较复杂的部分，究其原因，主要包括如下几个方面：

- 设备驱动属于 Linux 内核的部分，编写 Linux 设备驱动需要有一定的 Linux 操作系统内核基础；需要了解部分 Linux 内核的工作机制与系统组成。
- 编写 Linux 设备驱动需要对硬件的原理有相当的了解，大多数情况下我们是针对一个特定的嵌入式硬件平台编写驱动的。
- Linux 设备驱动中广泛涉及到多进程并发、同步、互斥等，容易出现 bug；因为 Linux 本身是一个多任务的工作环境，不可避免的会出现在同一时刻对同一设备产生并发操作。
- 由于属于内核的一部分，Linux 设备驱动的调试也相当复杂。Linux 设备驱动没有一个很好的 IDE 环境进行单步、变量查看等调试辅助工具；Linux 驱动跟 Linux 内核工作在同一层次，一旦发生问题，很容易造成内核的整体崩溃。

1.2 设备文件

1.2.1 设备类型

Linux 系统将设备分成三种基本类型：字符设备、块设备、网络设备。

1、字符设备

- 字符(char)设备是个能够像字节流（类似文件）一样被访问的设备。
- 对字符设备发出读/写请求时，实际的硬件 I/O 操作一般紧接着发生。

2、块设备

- 一个块设备驱动程序主要通过传输固定大小的数据（一般为 512 或 1k）来访问设备。
- 块设备通过 buffer cache(内存缓冲区)访问，可以随机存取，即：任何块都可以读写，不必考虑它在设备的什么地方。

3、网络设备

- 访问网络接口的方法仍然是给它们分配一个唯一的名字（比如 eth0），但这个名字在文件系统中不存在对应的节点。

- 内核调用一套和数据包传输相关的函数（socket 函数）而不是 read、write 等。

字符设备和块设备的区别

- 传输数据大小
- 是否有缓冲区

1.2.2 设备文件

在 Linux 操作系统中，每个驱动程序在应用层的/dev 目录下都会有一个设备文件和它对应，并且该文件会有对应的主设备号和次设备号。

1、查看设备文件

每个设备文件都有其文件属性（c 或者 b）。

使用 `ls /dev -l` 的命令查看

2、创建设备文件

使用 `mknod` 手工创建设备文件。

```
mknod filename type major minor
```

1.2.3 设备号

用户进程是通过设备文件来与实际的硬件打交道。每个设备文件都有两个设备号，第一个是主设备号，标识驱动程序。第二个是从设备号，标识使用同一个设备驱动程序的不同的硬件设备；比如：有两个软盘，就可以用从设备号来区分他们。设备文件的主设备号必须与设备驱动程序在登记时申请的主设备号一致，否则用户进程将无法访问到驱动程序。

02 Linux 字符设备驱动

2.1 基础概念

2.1.1 字符设备

字符设备通过字符（一个接一个的字符）以流方式向用户程序传递数据，就像串行端口那样。对字符设备发出读/写请求时，实际的硬件 I/O 操作一般紧接着发生。

2.1.2 字符设备文件

字符设备驱动通过操作/dev 目录下的字符设备文件在设备和用户应用程序之间交换数据，也可以通过它来控制实际的物理设备。这也是 Linux 的基本概念，一切皆文件。

```
创建字符设备文件：mknod /dev/test c 241 0
```

```
查看字符设备文件：ls -l /dev/test
```

```
crw-r--r-- 241, 0 Nov 17 2013 /dev/test
```

2.1.3 字符设备驱动

字符设备驱动程序是内核源码中最常用的设备驱动程序。在 Linux 内核实现对一个设备的驱动一般要完成 2 件事：字符设备注册、字符设备操作。

2.2 重要结构体

2.2.1 inode 结构体

在 Linux 文件系统中，每个文件都用一个 `struct inode` 结构体来描述，这个结构体里面记录了这个文件的所有信息。例如：文件类型，访问权限等。

`struct inode` 结构体包括的重要成员：

成员	描述
<code>dev_t i_rdev</code>	设备文件的设备号
<code>struct cdev *i_cdev</code>	代表字符设备的数据结构

说明：内核使用 `inode` 结构体在内核内部表示一个文件。因此，它与表示一个已经打开的文件描述符的结构体（即 `file` 文件结构）是不同的，我们可以使用多个 `file` 文件结构表示同一个文件的多个文件描述符，但此时，所有的这些 `file` 文件结构全部都只能指向一个 `inode` 结构体。（`struct inode` 是代表一个“静态文件”，通常 `struct inode` 描述的是文件的静态信息，即这些信息很少会改变。）

2.2.2 file 结构体

`file` 结构体指示一个已经打开的文件（设备对应于设备文件），每个打开的文件在内核空间都有一个相应的 `struct file` 结构体，它由内核在打开文件时创建，并传递给在文件上进行操作的任何函数，直至文件被关闭。如果文件被关闭，内核就会释放相应的数据结构。

`struct file` 结构体包括的重要成员：

成员	描述
<code>fmode_t f_mode</code>	此文件模式通过 <code>FMODE_READ</code> , <code>FMODE_WRITE</code> 识别了文件为可读的，可

	写的，或者是二者。在 open 或 ioctl 函数中可能需要检查此域以确认文件的读/写权限，你不必直接去检测读或写权限，因为在进行 octl 等操作时内核本身就需要对其权限进行检测。
loff_t f_pos	当前读写文件的位置。如果想知道当前文件当前位置在哪，驱动可以读取这个值而不会改变其位置。对 read/write 来说，当其接收到一个 loff_t 型指针作为其最后一个参数时，他们的读写操作便作更新文件的位置，而不需要直接执行 filp ->f_pos 操作。而 llseek 方法的目的就是用于改变文件的位置。
unsigned int f_flags	文件标志，如 O_RDONLY, O_NONBLOCK 以及 O_SYNC。在驱动中还可以检查 O_NONBLOCK 标志查看是否有非阻塞请求。其它的标志较少使用。特别地注意的是，读写权限的检查是使用 f_mode 而不是 f_flags。
struct file_operations *f_op	与文件相关的各种操作。当应用程序需要对文件进行各种操作时，调用对应的系统调用 API，内核将与这个文件有关的系统调用 API 用这个成员里对应的函数指针来填充，从而实现了应用对文件的打开，读，写等功能实现。 file_operation 结构体详见： file_operations 结构体
void *private_data	在驱动调用 open 方法之前，open 系统调用设置此指针为 NULL 值。你可以很自由的将其做为你自己需要的一些数据域或者不管它。例如，你可以将其指向一个分配好的数据，但是你必须记得在 file struct 被内核销毁之前在 release 方法中释放这些数据的内存空间。private_data 用于在系统调用期间保存各种状态信息是非常有用的。

struct file 是代表一个打开的“动态文件”，通常 struct file 描述的是动态信息，即在对文件的操作的时候，struct file 里面的信息经常会发生变化。典型的是 struct file 结构体里面的 f_pos(记录当前文件的位移量)，每次读写一个普通文件时 f_ops 的值都会发生改变。

2.2.3 chrdevs 结构体

通过数据结构 struct inode 中的 i_cdev 成员可以找到 cdev，而所有的字符设备都在 chrdevs 数组中。

下面先看一下 chrdevs 的定义：

```
#define CHRDEV_MAJOR_HASH_SIZE 255
```

```
static DEFINE_MUTEX(chrdevs_lock);

static struct char_device_struct {
    struct char_device_struct *next;
    unsigned int major;
    unsigned int baseminor;
    int minorct;
    char name[64];
    struct cdev *cdev;
} *chrdevs[CHRDEV_MAJOR_HASH_SIZE];
```

可以看到全局数组 `chrdevs` 包含了 255(`CHRDEV_MAJOR_HASH_SIZE` 的值)个 `struct char_device_struct` 的元素，每一个对应一个相应的主设备号。

如果分配了一个设备号，就会创建一个 `struct char_device_struct` 的对象，并将其添加到 `chrdevs` 中；这样，通过 `chrdevs` 数组，我们就可以知道分配了哪些设备号。

通过 `cat /proc/devices` 可以看到哪些设备号被注册了

2.2.4 cdev 结构体

在 Linux 内核中，使用 `cdev` 结构体来描述一个字符设备。

`cdev` 结构体的定义如下：

```
<include/linux/cdev.h>

struct cdev {
    struct kobject kobj;
    struct module *owner;
    const struct file_operations *ops; //成员 file_operations 来定义字符设备驱动提供
    供给 VFS 的接口函数，如常见的 open()、read()、write() 等；
    struct list_head list; //内核链表
    dev_t dev; //成员 dev_t 来定义设备号（分为主、次设备号）以确定字符设备的唯一
    性；
    unsigned int count; //次设备号个数
};
```

2.2.5 file_operations 结构体

Linux 下的设备驱动程序被组织为一组完成不同任务的函数的集合，通过这些函数使得设备操作犹如文件一般。在应用程序看来，硬件设备只是一个设备文件，应用程序可以象操作普通文件一样对硬件设备进行操作，如 `open()`、`close()`、`read()`、`write()` 等。

file_operations 是把系统调用和驱动程序关联起来的关键数据结构。这个结构体的每一个成员都对应着一个系统调用 API。

用户进程利用在对设备文件进行诸如 read/write 操作的时候，系统调用通过设备文件的主设备号找到相应的设备驱动程序，然后读取这个数据结构相应的函数指针，接着把控制权交给该函数，这是 Linux 的设备驱动程序工作的基本原理。

struct _file_operations 在 fs.h 这个文件里面被定义的，如下所示：

```
struct file_operations {

    struct module *owner; //拥有该结构的模块的指针，一般为 THIS_MODULE

    loff_t (*llseek) (struct file *, loff_t, int); //用来修改文件当前的读写位置
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *); //从设备
    中同步读取数据

    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *); //
    向设备发送数据

    ssize_t (*aio_read) (struct kiocb *, const struct iovec *, unsigned long,
    loff_t); //初始化一个异步的读取操作

    ssize_t (*aio_write) (struct kiocb *, const struct iovec *, unsigned long,
    loff_t); //初始化一个异步的写入操作

    int (*readdir) (struct file *, void *, filldir_t); //仅用于读取目录，对于设
    备文件，该字段为 NULL

    unsigned int (*poll) (struct file *, struct poll_table_struct *); //轮询函
    数，判断目前是否可以非阻塞的读写或写入

    int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
    //执行设备 I/O 控制命令

    long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long); //不
    使用 BLK 文件系统，将使用此种函数指针代替 ioctl

    long (*compat_ioctl) (struct file *, unsigned int, unsigned long); //在 64
    位系统上，32 位的 ioctl 调用将使用此函数指针代替

    int (*mmap) (struct file *, struct vm_area_struct *); //用于请求将设备内存
    映射到进程地址空间
```

```
int (*open) (struct inode *, struct file *); //打开设备

int (*release) (struct inode *, struct file *); //关闭设备

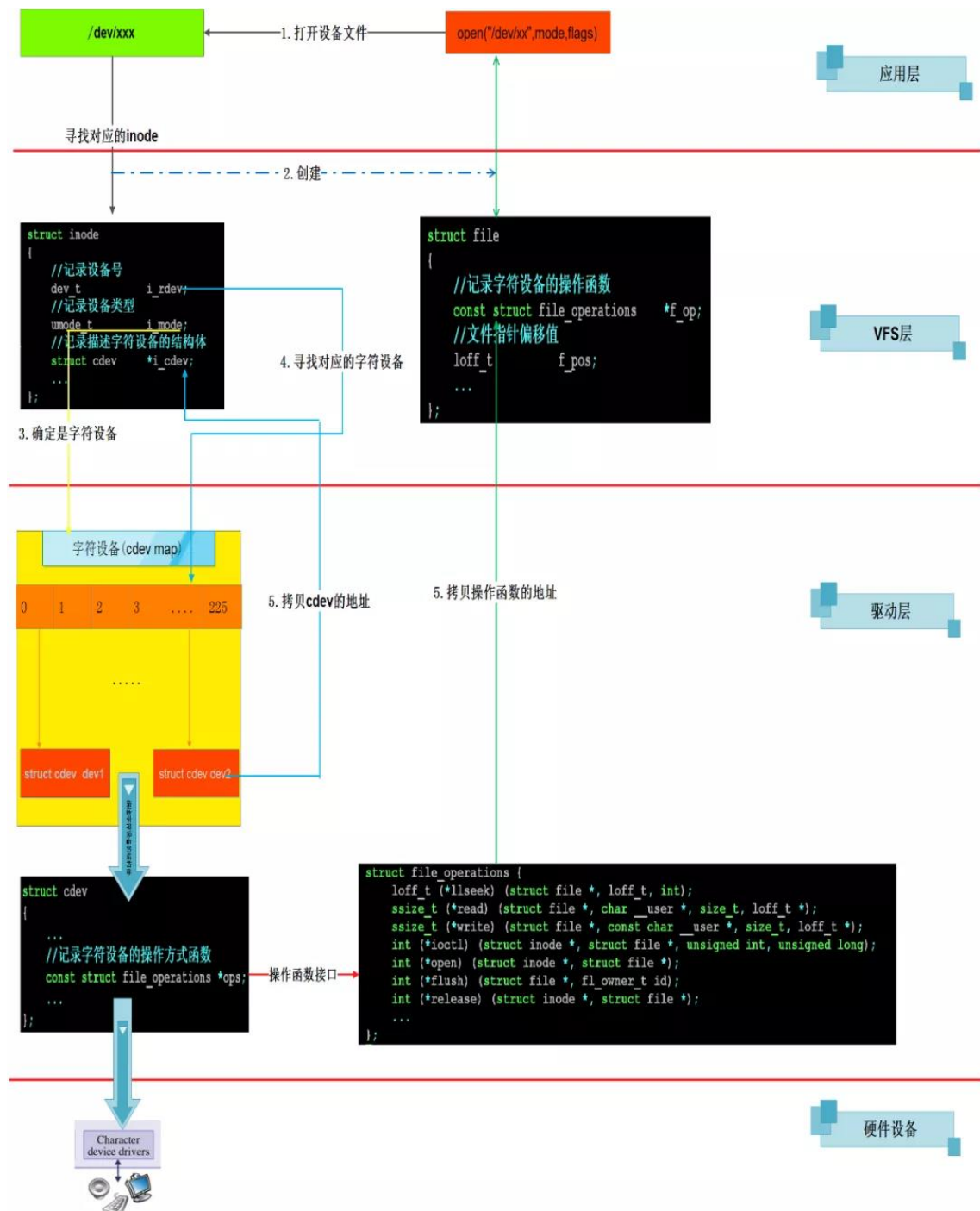
int (*fsync) (struct file *, struct dentry *, int datasync); //刷新待处理
的数据

int (*aio_fsync) (struct kiocb *, int datasync); //异步刷新待处理的数据

int (*fasync) (int, struct file *, int); //通知设备 FASYNC 标志发生变化
};
```

2.3 字符设备驱动流程分析

通过下图我们可以知道，如果想访问底层设备，就必须打开对应的设备文件。在这个打开的过程中，Linux 内核将应用程序打开的设备文件和对应的驱动程序关联起来，接下来应用程序对设备文件的操作即为驱动对硬件设备的操作。



1、创建设备文件时

当执行“`mknod test_dev c 241 0`”建立设备文件后，内核对应生成一个 `inode` 结构体来表示这个文件。

2、插入设备驱动时

当执行“insmod test_dev_driver.ko”插入“test_dev”设备驱动后，内核会将“test dev driver”驱动注册的 241 设备号及 test cdev 存放到 chrdevs 字符设备表。

3、应用程序执行时

当执行 “./app” 应用程序后，执行 `open("/dev/test_dev")` 打开设备文件时：

- 1) 内核可以根据设备文件对应的 `struct inode` 结构体描述的信息，知道接下来要操作的设备类型（字符设备还是块设备）。还会分配一个 `struct file` 结构体。
- 2) 接下来，内核通过 `inode` 的文件类型判断是字符设备，然后找到字符设备表 `chrdevs`。通过 `inode` 中的设备号去索引 `chrdevs` 表，找到设备号对应的字符设备结构体 “`test_cdev`”。
- 3) 找到 `cdev` 后，内核将 “`test_cdev`” 结构体记录在 `struct inode` 结构体的 `i_cdev` 成员中。然后，将 `i_cdev` 成员的 `fops` 记录在打开这个文件的 `struct file` 结构体的 `f_op` 成员中。（即：将 `struct cdev` 结构体中的函数操作接口 `fops` 填充 `struct file` 结构体的 `f_op` 成员中。）
- 4) 最后，内核会给应用层返回一个文件描述符 `fd`，这个 `fd` 是和 `struct file` 结构体对应的。接下来上层的应用程序就可以通过 `fd` 来找到 `struct file`，然后再由 `struct file` 中的 `f_op` 找到操作字符设备 `cdev` 中的 `fops` 函数接口了。

03 字符设备驱动注册

3.1 chrdev 版注册

3.1.1 注册字符设备驱动

```
#include <linux/fs.h>
static inline int register_chrdev(unsigned int major, const char *name, const
struct file_operations *fops)
```

major: 设备号	<ul style="list-style-type: none">在设备管理中，除了设备类型外，内核还需要一对被称为主从设备号的参数，才能唯一标识一个设备，类似人的身份证号。利用 <code>cat /proc/devices</code> 查看设备驱动申请到的设备名，设备号。
name: 名字	<ul style="list-style-type: none">设备驱动名，用 <code>cat /proc/devices</code> 查看已注册设备，用 <code>mknod</code> 建立设备文件。
fops: 操作接口	<ul style="list-style-type: none"><code>file_operations</code> 是把系统调用和驱动程序关联起来的关键数据结构。这个结构体的每一个成员都对应着一个系统调用 API。用户进程利用在对设备文件进行诸如 <code>read/write</code> 操作的时候，系统调

	用通过设备文件的主设备号找到相应的设备驱动程序，然后读取这个数据结构相应的函数指针，接着把控制权交给该函数，这是 Linux 的设备驱动程序工作的基本原理。
返回值	<ul style="list-style-type: none">• 0: 成功• 非 0: 失败

3.1.2 注销字符设备驱动

```
#include <linux/fs.h>
static inline void unregister_chrdev(unsigned int major, const char *name)
```

major: 设备号	<ul style="list-style-type: none">• 与注册函数的主设备号一致
name: 名字	<ul style="list-style-type: none">• 与注册函数的设备名称一致

3.1.3 分配设备号

1、静态分配设备号

查看内核主设备号文档: Documentation/device.txt

240-254 char LOCAL/EXPERIMENTAL USE

```
//静态分配代码
int ret;
ret = register_chrdev(241, "test-driver", &test_fops); //返回 0 成功, 返回非零失败
```

2、动态分配设备号

```
//动态分配代码
int test_major;
test_major = register_chrdev(0, "test-driver", &test_fops); //返回大于 0 的主设备号成功, 返回非零失败
```

3.2 cdev 版注册

3.2.1 申请设备号

1、设备号

一个字符设备或块设备都有一个主设备号和一个次设备号。主设备号用来标识与设备文件相连的驱动程序，用来反映设备类型。次设备号被驱动程序用来辨别操作的是哪个设备，用来区分同类型的设备。

linux 内核中，设备号用 `dev_t` 来描述，定义如下：

- `typedef u_long dev_t;`

内核也为我们提供了几个方便操作的宏实现 `dev_t`：

1) 从设备号中提取 `major` 和 `minor`

- `MAJOR(dev_t dev);`
- `MINOR(dev_t dev);`

2) 通过 `major` 和 `minor` 构建设备号

- `MKDEV(int major, int minor);`

设备号操作宏定义如下：

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

2、静态申请（方式一）

【申请函数】

```
int register_chrdev_region(dev_t from, unsigned count, const char *name)
```

参数：

- 1、`from`：设备号
- 2、`count`：申请注册的设备数量
- 3、`name`：名字

返回值：Return value is zero on success, a negative error code on failure.

【申请代码】

```
#include <linux/fs.h>
devno = MKDEV(major, minor);
register_chrdev_region(devno, 1, "test_driver");
```

3、动态申请（方式二）

【申请函数】

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

参数：

- 1、dev: 内核自动分配设备号，并通过这个指针返回
- 2、baseminor: 你期望自动分配的次设备号从几开始
- 3、count: the number of minor numbers required
- 4、name: the name of the associated device or driver

返回值: Return value is zero on success, a negative error code on failure.

【申请代码】

```
#include <linux/fs.h>
int major, minor;
int devno;
alloc_chrdev_region(&devno, 0, 1, "test_driver");
printf("ma[%d], mi[%d]\n", MAJOR(devno), MINOR(devno));
```

3.2.2 注册 cdev

1、头文件

<linux/cdev.h>

2、注册函数

1) cdev_alloc

```
struct cdev *cdev_alloc(void);
```

该函数主要分配一个 struct cdev 结构，动态申请一个 cdev 内存

2) cdev_init

```
void cdev_init(struct cdev *, const struct file_operations *);
```

该函数主要对 struct cdev 结构体做初始化，最重要的就是建立 cdev 和 file_operations 之间的连接。

3) cdev_add

```
int cdev_add(struct cdev *p, dev_t dev, unsigned count);
```

该函数向内核注册一个 struct cdev 结构，即正式通知内核由 struct cdev *p 代表的字符设备已经可以使用了。

参数说明：注册的设备号 dev 和数量 count，这两个参数直接赋值给 struct cdev 的 dev 成员和 count 成员。

3.2.3 注销 cdev

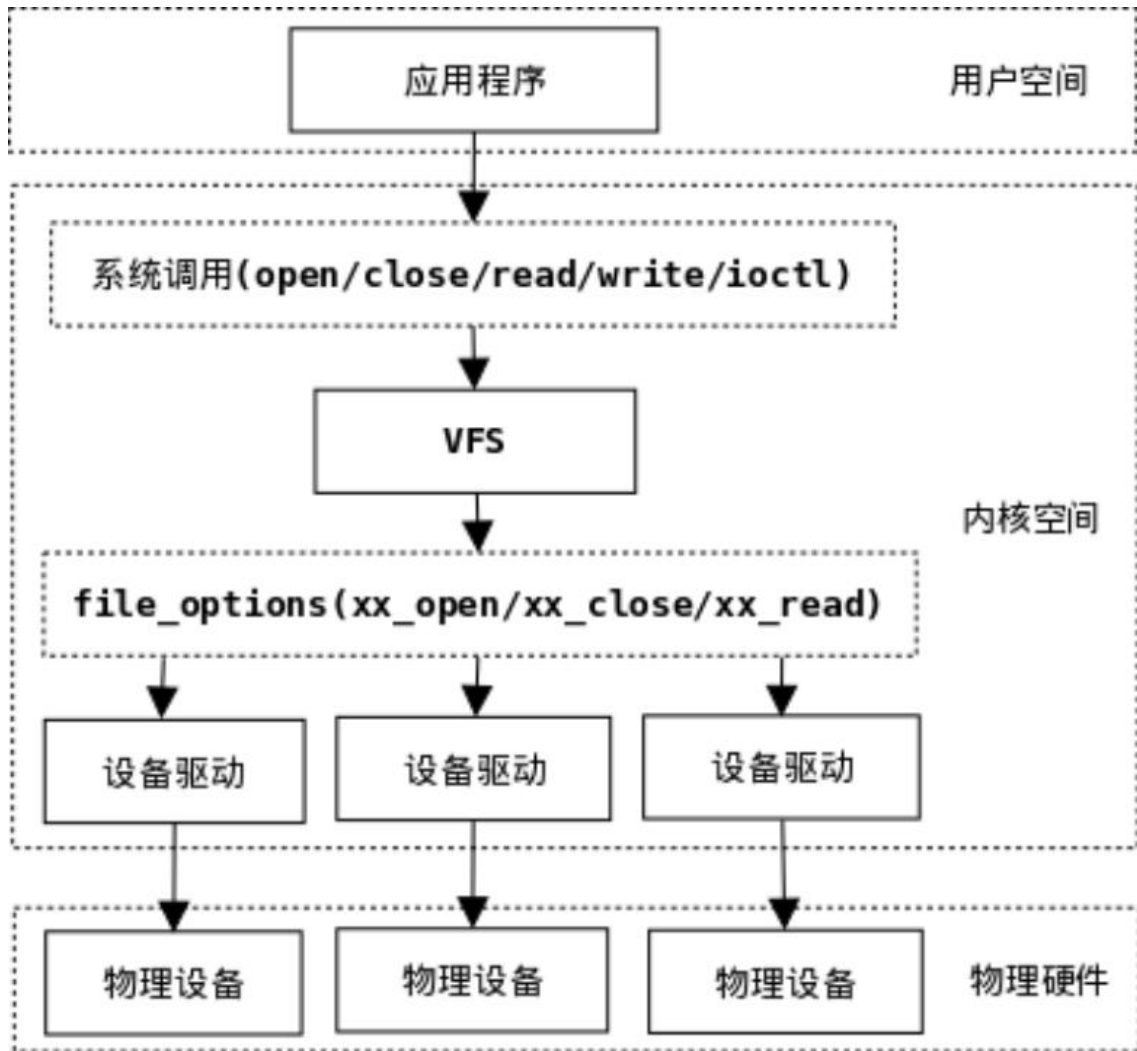
```
void cdev_del(struct cdev *p);
```

该函数向内核注销一个 struct cdev 结构，即正式通知内核由 struct cdev *p 代表的字符设备已经不可以使用了。

04 字符设备驱动接口

4.1 字符设备驱动接口概述

应用程序和内核之间的接口是系统调用，内核定义的 file_operations 结构体中成员函数是字符设备驱动与内核的接口，是用户空间对 linux 进行系统调用的最终落实者，这个结构体包含对文件打开，关闭，读写，控制的一系列成员函数。file_operations 就是实现字符设备驱动接口的核心。



`struct file_operations` 是一个字符设备把驱动的操作和设备号联系在一起的纽带，是一系列指针的集合，每个被打开的文件都对应于一系列的操作，用 `file_operations` 用来执行一系列的系统调用。

4.2 打开和关闭设备

4.2.1 open 方法

当用户进程执行 `open()` 系统调用的时候，内核将调用驱动程序 `open()` 函数。

`open` 方法提供给驱动程序以初始化的能力，在大部分驱动程序中 `open` 应该完成以下工作：

- 检查特定设备的错误，例如设备是否准备就绪等硬件问题；
- 如果设备是首次打开，则对其进行初始化；

`open` 方法的原型如下：

```
int (*open) (struct inode *, struct file *);
```

其中，inode 参数在其 i_cdev 字段中包含了我们所需要的信息，即我们先前设置的 cdev 结构；

与 open() 函数对应的是 release() 函数。

4.2.2 release 方法

当最后一个打开设备的用户进程执行 close() 系统调用的时候，内核将调用驱动程序 release() 函数。

release 函数的主要任务是清理未结束的输入输出操作，释放资源，用户自定义排他标志的复位等。

release 方法的原型如下：

```
int (*release) (struct inode *, struct file *);
```

4.2.3 示例

```
int test_open(struct inode *node, struct file *filp)
{
    printk("%s\n", __FUNCTION__);
    return 0;
}

int test_close(struct inode *node, struct file *filp)
{
    printk("%s\n", __FUNCTION__);
    return 0;
}

struct file_operations test_fops = {
    .open = test_open,
    .release = test_close,
};
```

4.3 控制设备 (ioctl)

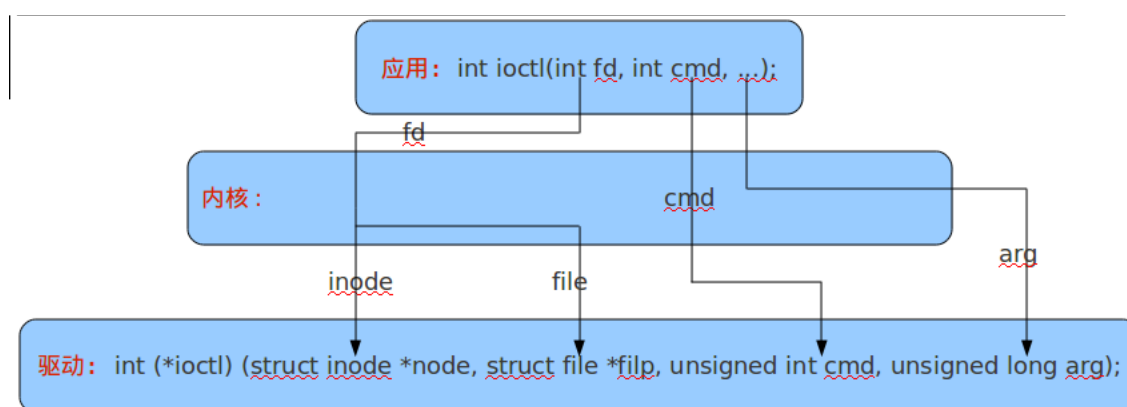
4.3.1 ioctl 概述

1、为什么需要 ioctl

虽然在结构体“struct file_operations”中有很多对应的操作函数，但硬件操作的方法 fops 函数集无法完全覆盖，所以可以通过 ioctl 来自定义硬件操作命令。

- 例如：CD-ROM 的驱动，想要一个弹出光驱的操作，这种操作并不是所有的字符设备都需要的，所以文件操作结构体也不会有对应的函数操作。
- 例如：针对串口设备，还需提供对串口波特率、奇偶校验位、终止位的设置，这些配置信息需要从应用层传递一些基本数据，仅仅是数据类型不同。

2、应用层与驱动函数的 ioctl 之间的联系



4.3.2 ioctl 函数

1、头文件

```
include/linux/ioctl.h
```

2、接口函数

```
long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
```

1) 参数

- file: 指针对应的是应用程序传递的文件描述符 fd 的值，和传递给 open 方法的相同参数。
- cmd: cmd 参数从用户那里不改变地传下来，由驱动工程师自定义。
- arg: 可选的参数 arg 参数以一个 unsigned long 的形式传递，不管它是否由用户给定为一个整数或一个指针。

2) 返回值

- 如果传入的非法命令，ioctl 返回错误号-EINVAL。
- 内核中的驱动函数返回值都有一个默认的方法，只要是正数，内核就会认为这是正确的返回，并把它传给应用层，如果是负值，内核就会认为它是错误号了。

4.3.3 ioctl 进阶

1、命令 (cmd)

如果有两个不同的设备，但它们的 ioctl 的 cmd 却一样的。为了防止这样的事情发生，内核对 cmd 又有了新的定义，规定了 cmd 都应该不一样。

具体用法参考<asm-generic/ioctl.h>和 ioctl-number.txt 这两个文档。

2、传参 (arg)

一般会有两种的传参方法：

- 整数：直接使用就可以了。
- 指针：通过指针，可以传任何类型，非常灵活。例如：unsigned long = 结构体地址。

4.3.4 ioctl 示例

1、ioctl 驱动

```
#include <asm/uaccess.h>
#include "test_cmd.h"

void arm_on(void)
{
    //驱动硬件寄存器代码

    printk("%s\n", __FUNCTION__);
}

void arm_off(void)
{
    printk("%s\n", __FUNCTION__);
}

long test_unlocked_ioctl (struct file *filp, unsigned int cmd, unsigned long args)
{
    switch(cmd) {
        case TEST_ON:
            arm_on();
            break;
        case TEST_OFF:
```

```
                arm_off();
                break;
            default:
                printk("unknow ioctl cmd\n");
                return -1;
        }
        return 0;
}

struct file_operations test_fops = {
    .unlocked_ioctl = test_unlocked_ioctl,
};
```

2、ioctl 命令自定义

```
//test_cmd.h
#define TEST_ON 1
#define TEST_OFF 2
```

3、ioctl 应用测试

```
#include "test_cmd.h"

int main(int argc, char *argv[])
{
    int fd;
    char buf[10];
    fd = open("/dev/test_dev", O_RDWR);
    if(fd < 0){
        perror("open failed\n");
        return -1;
    }
    ioctl(fd, TEST_ON);

    return 0;
}
```

4.4 读写设备 (read/read)

4.4.1 读设备 (read)

用途：从设备中读取数据。

场景：当对设备文件进行 read() 系统调用时，将调用驱动程序 read() 函数：

```
ssize_t read(struct file *filp, char __user *buff, size_t count, loff_t
*offp);
```

当该函数指针被赋为 NULL 值时，将导致 read 系统调用出错并返回 -EINVAL (“Invalid argument, 非法参数”)。函数返回非负值表示成功读取的字节数（返回值为 “signed size” 数据类型，通常就是目标平台上的固有整数类型）。

4.4.2 写设备 (write)

用途：向设备发送数据。

场景：当对设备文件进行 write () 系统调用时，将调用驱动程序的 write () 函数：

```
ssize_t write(struct file *filp, const char __user *buff, size_t count, loff_t
*offp);
```

当该函数指针被赋为 NULL 值时，write 系统调用会向调用程序返回一个 -EINVAL。如果返回值非负，则表示成功写入的字节数。

4.4.3 交换数据函数

```
#include <asm/uaccess.h>
unsigned long copy_to_user(void __user *to, const void *from, unsigned long
count);
unsigned long copy_from_user(void *to, const void __user *from, unsigned long
count);
```

read 方法从设备拷贝数据到用户空间(使用 copy_to_user)

write 方法从用户空间拷贝数据到设备(使用 copy_from_user)

备注：每个 read 或 write 系统调用请求一个特定数目字节的传送。不管这些方法传送多少数据，它们通常应当更新 *offp 中的文件位置来表示在系统调用成功完成后当前的文件位置。

4.4.4 示例

```
//第一步：添加读写有关头文件
#include <asm/uaccess.h>           //添加 copy_to_user 与 copy_from_user 的头文件
#define BUF_SIZE          1024     //内核 buffer 的 size
```

```
static char tmpbuf[BUF_SIZE]; //内核空间的数据读写数组
```

//第二步：添加设备驱动操作接口函数 write 和 read，这两个函数在应用程序调用 write 及 read 时调用。

```
static ssize_t test_chardev_read(struct file *file, char __user *buf, size_t
const count, loff_t *offset)
{
    if(count < BUF_SIZE)    //读写大小检查
    {
        if(copy_to_user(buf, tmpbuf, count))//执行完成后返回还需拷贝的字
节数。成功为 0
        {
            printk(KERN_ALERT "copy to user fail \n");
            return -EFAULT;
        }
    }else{
        printk(KERN_ALERT "read size must be less than %d\n",
BUF_SIZE);
        return -EINVAL;
    }

    *offset += count;    //文件位置指针更新
    return count;    //返回文件成功读写的字节数
}
```

```
static ssize_t test_chardev_write(struct file *file, const char __user
*buf, size_t const count, loff_t *offset)
{
    if(count < BUF_SIZE)    //读写大小检查
    {
        if(copy_from_user(tmpbuf, buf, count)) //执行完成后返回还需拷贝
的字节数。成功为 0
        {
            printk(KERN_ALERT "copy from user fail \n");
            return -EFAULT;
        }
    }else{

        printk(KERN_ALERT "size must be less than %d\n", BUF_SIZE);
        return -EINVAL;
    }

    *offset += count;    //文件位置指针更新
```

```

        return count;    //返回文件成功读写的字节数
    }

//第三步：完成 file_operations 的赋值，为该字符设备添加读写操作接口
static struct file_operations chardev_fops={
    .owner = THIS_MODULE,
    .read = test_chardev_read,
    .write = test_chardev_write,
};

```

重点回顾

1、在 Linux 内核中：

- 使用 cdev 结构体来描述字符设备；
- 通过其成员 dev_t 来定义设备号（分为主、次设备号）以确定字符设备的唯一性；
- 通过其成员 file_operations 来定义字符设备驱动提供给 VFS 的接口函数，如常见的 open()、read()、write() 等；

2、在 Linux 字符设备驱动中：

- 模块加载函数通过 register_chrdev_region() 或 alloc_chrdev_region() 来静态或者动态获取设备号；
- 通过 cdev_init() 建立 cdev 与 file_operations 之间的连接，通过 cdev_add() 向系统添加一个 cdev 以完成注册；
- 模块卸载函数通过 cdev_del() 来注销 cdev，通过 unregister_chrdev_region() 来释放设备号；

3、用户空间访问该设备的程序：

- 通过 Linux 系统调用，如 open()、read()、write()，来调用 file_operations 来定义字符设备驱动提供给 VFS 的接口函数；

思考练习

- 【实验 1】编写 chrdev 版字符设备驱动的注册程序
- 【实验 2】编写 cdev 版字符设备驱动的注册程序
- 【实验 3】打开（open）设备实验
- 【实验 4】控制（ioctl）设备实验