Linux内核架构源代码解析

（基于MIPS32架构）

龙传慧 著

二0二0年八月

# 自 序

作者男，1984年出生，江西赣州人，中共党员，2006年本科毕业于南昌大学电子信息工程专业，获得工学学士学位，2011年中国科学院光技术研究所硕士毕业，获得工学硕士学位。高考太紧张发挥的不好，本科上的大学不是很好，学的东西也比较杂，模电、数电、C语言、C++、通信、信号处理等等，什么都学了点，什么也没学通，还是稀里糊涂。本科做毕业设计时，自学了一下80C51单片机，毕业设计做的是一个单片机的简单的控制电路，那时自己去电子市场花了一百多块钱买了个开发板，写了程序往里烧，程序跑起来的，控制了几个LED灯，那时觉得挺好玩，我也很好奇单片机里面倒底是怎么工作的。大四那年，因为觉得上的本科不是很好，所以想努力考个好一点学校的研究生深造一下，结果选择了报考中国科学院光技术研究所（因为那时研究所不用交学费！！），结果居然还考上了。研究生阶段做的是一个模拟高压放大电路用于驱动压电陶瓷驱动器，虽然本科是学过模电、数电，但是那时真没学明白，一切感觉从零开始。研究生学习阶段学习环境的自由，让我有时间慢慢的从新开始学习电路方面的知识，打牢基础，建立了模电、数电、处理器之间的联系，理解了其本质，最终在导师的指导下我顺利毕业了。到现在我都非常感激和感谢研究所给我提供的良好的学习环境，感谢导师们的指导，读研的三年感觉比我读四年大学学到的知识多多了。

研究生毕业回到了工作岗位，空闲时间总想学点什么，要不然我学的知识真的要荒废了。一直以来我都有一个梦想，梦想自己能在某一行当成为专家（小时候的理想是当科学家，现在估计是当不成了，成中年大叔了），曾经学过C++、VB等编程语言，还学过英语，甚至还学过几天俄语，现在感觉好搞笑！！研究生阶段做模电，买了一堆电子元器件，搭了些模拟电路玩，后来感觉玩不出什么名堂来，因为要做个印制电路板就得花一千多块钱，自己玩不起。想了想还是搞软件，买个开发板玩玩比较靠谱，于是我又回到了大学时的情景，买了个8051单片机开发板玩。写了几天程序后，感觉单片机实在是太低级了，速度慢，内存小，连显示个小小的图片都要半天。后来，知道了ARM很火，又买了个ARM开发板，买回来发现开发资料里只有介绍怎么在Linux操作系统里写程序的教程，而我向来是个喜欢寻根问底的人，玩单片机时知道处理器是怎么执行每一条指令的，现在弄个操作系统把底层都屏蔽了，所以我把操作系统给刷了，写了一段时间的裸机程序。

在玩单片机的时候（大概是2012年的时候）我接触到了一本书，那是周立功写的一本书，书名我忘了，书的内容是讲在单片机中实现一个简单的操作系统。当时我感觉很惊讶，我知道的Windows操作系统装到电脑上要占好几个G的空间，小小的单片机如何实现操作系统？看完这本书之后，纠正了我错误的观念，Windows是操作系统，但是操作系统不仅仅只有Windows，凡是能实现进程调度，能让处理器运行多个程序的都是操作系统。后来我又了解到，Windows只是商业上最成功的操作系统，国外讲操作系统的教材根本就没Windows什么事，C、C++语言也不一定非要在Window系统下编写，我发现我们的眼界在大学的时候就被限制在了Windows操作系统上。讲操作系统必然要讲到Unix和Linux，后者是前者的继承和发展，Linux内核是一个开源的项目，任何人都可以直接从其官网上下载源代码，Linux内核加上一些系统应用、用户应用软件就成了跟Windows一样的发行版操作系统了。Linux内核尤其广泛地应用在嵌入式系统中，因为它是免费的、开源的、可裁剪的，我买的ARM开发板装的就是Linux系统。

ARM开发板写了一段时间的裸机程序后，我觉得必须要学操作系统，因为现在所有应用程序都是在操作系统之上运行的，连单片机都能跑操作系统，那么操作系统应该不难学，从此我就踏上了学习Linux内核的不归路。

要自学操作系统又是谈何容易，我先是看几本有关操作系统原理的书，后面还接触了一下用于教学用的Minix系统，想先从一个简单的操作系统开始，后面再学复杂的Linux。但是很不幸，我发现简单的我也学不来，还是直接回到Linux。Linux系统在加载到内存运行前，系统要运行一段引导加载程序（如电脑中的BIOS），用于将内核加载到内存，当时的想法是先学引导加载程序，后学Linux内核，掌握整个系统的运作。嵌入式系统中最常用的引导加载程序是U-boot，我几乎是花了将近一年的时间学习U-boot，最终还是没有学完。最后我还是放弃了，以我的能力不可能求全，还是直接学Linux内核吧，能把内核学好就不错了。

Linux内核的书买了一堆，最开始看的是《深入Linux内核架构》，我天真的以为直接学习内核源代码就能直接成为专家了，结果是书基本上看不懂。后来大概是到了2016、2017年的时候，反反复复的看书感觉有点眉目了，开始阅读内核的源代码。刚开始，内核数据结构之间错综复杂的关系，让我非常的迷茫，甚至买了一个笔记本来专门画出数据结构之间的关系。

随着学习的深入，我产生了写笔记的想法，毕竟好记性不如烂笔头，而且内核方方面面的内容庞杂，如果不做记录，过几天自己都忘了，又等于白学。于是，本书的雏形就在我脑子里酝酿，最终2017年7月的时候我决定开设微信公众号，把学习的心得和经验跟大家一起分享，一起学习、进步。在笔记的撰写过程中，不断的促进了对内核的学习，逼着自己去弄清内核的工作机理和组织结构。经过几年的努力，现在终于感觉有种拨云见日的感觉，内核的架构在脑海里也清晰了起来。曾经也迷茫、困惑过，怀疑过写作笔记的意义，甚至想到过放弃，最终还是庆幸在众多关注微友的支持下坚持了下来，另外也要非常感谢家人对我的支持，坚持就是胜利。

近几年在学习Linux内核的过程中，我也不断地关注了国产处理器、操作系统方面的信息，操作系统离不开处理器。目前，电脑系统基本是英特尔和微软的天下，手机则是ARM和安卓、IOS的天下，没有一样是国产的，这给我们的信息安全带来了非常大的隐患，加上2018年的中兴事件，更是坚定了我们发展自主处理器和操作系统的决心。因此我又抛弃了ARM的学习，选择了国产龙芯处理器（MIPS架构），支持国产才是正道，虽然个人的能力有限，但也要力所能及的支持一下。

回顾几年的内核学习经验，最开始是好奇，觉得很神奇，希望搞明白内核是如何工作的，接下来是困惑，真正接触内核了就不知道从何处下手，不知道内核是什么结构，各组成部分、数据结构是什么关系，后面是欣喜，感觉掌握点眉目了，可以顺藤摸瓜了，最后是淡定和从容，理解了内核架构，整个内核在脑海里有一个整体的架构，可以去思考和研究各部分是怎么工作的，相互关系是什么，可以去研究某个函数做了些什么工作。学习内核最终的目的是将内核移植到目标系统中，我正在为此努力。

我看了很多关于内核方面的书，国外讲内核源代码的书虽然写得很好，如《深入Linux内核架构》等，但讲解的内核版本太低，多是2.6.X版本，现在内核版本都更新到5.0了。国内讲内核方面的书，多是讲驱动的移植，个人觉得讲内核都讲的不够深入，只知然不知其所以然。所以我决定写一本自己的书，讲解内核的架构和主要骨干源代码的实现，我始终认为只有深入透彻的理解内核，才能更好地应用内核。

内核代码量巨大，各部分内容庞杂，关系错综复杂，新手如何快速入门呢？内核说到底是编译链接成的一个单一的目标文件，系统启动时加载到内存中开始运行，内核启动完成后负责为用户进程提供服务。内核更多的是充当一个资源管理器的功能，通过某些数据结构管理和调配系统各种软、硬件资源。学习内核要先了解内核的组织架构，各功能部件的主要功能和相互关系，避免一开始就迷茫在代码细节中。本书在讲解内核的过程中，尽量地在章节的开始就建立本部分的组织架构，让读者对本部分的功能和管理结构有整体的认识，再顺着结构去阅读和理解内核源代码。

本书主要从内核组织架构及源代码讲解内核内存管理、进程管理、文件系统、驱动程序、网络等各功能组件的工作机理，以MIPS32架构为基础介绍内核中特定于处理器架构的代码实现，并以龙芯1B处理器为基础介绍Linux内核移植的相关知识。

作者在Windows系统中采用CodeLite集成开发环境阅读源代码，CodeLite是一款开源，支持Windows、Linux等操作系统的集成开发环境。移植内核前，读者需要在您的电脑上安装上Linux发行版操作系统，以便对内核源代码进行配置、编译和链接，作者安装的是国产Deepin操作系统，具体安装方法读者可参考Deepin操作系统官网。

学习本书之前读者应先了解操作系统基本知识、Linux操作系统（发行版）的安装和常用命令、MIPS架构（或其它处理器架构）、以及C语言和汇编语言（特定于处理器）等的相关知识。由于作者水平有限，难免会有错误之处，敬请读者批评指正。

本书结构如下：

1. 前言：简要介绍操作系统组成，Linux内核框架和和组成部分的功用，移植内核的流程，以及内核源代码中常用的通用数据结构。

第2章 启动内核：介绍内核启动阶段初始化函数功能。

第3章 物理内存管理：介绍内核对物理内存的分配、释放等管理工作。

第4章 虚拟内存管理：介绍内核对进程虚拟地址空间的管理，以及内核自身地址空间的管理。

第5章 进程管理：介绍进程创建、调度、进程间通信等相关内容。

第6章 内核活动：介绍异常（中断）的处理、系统调用实现以及内核时间管理等。

第7章 文件系统：介绍虚拟文件系统的实现，几种具体文件系统类型的实现；

第8章 通用驱动模型：介绍内核对设备和驱动程序进行管理的驱动模型，以及常用总线驱动实现。

第9章 字符设备驱动程序：介绍常见字符设备驱动程序框架及实现。

第10章 块设备驱动程序：介绍块设备驱动程序框架及实现。

第11章 内存与块设备交互：介绍内核页缓存和块缓存、数据同步以及页回收和页交换的实现。

第12章 网络与设备驱动：介绍内核网络栈的实现，网络数据包接收和发送的流程，以及网络设备驱动程序的实现。

第13章 内核移植：以龙芯1B处理器为例，介绍移植公版linux-4.2.4内核到龙芯1B开发板的步骤和方法。

扫码关注微信公众号：



# 目 录

# 

# 前言

翻开本书的读者，相信你已经对计算机的基本结构，计算机编程等已经有了初步的了解。计算机就是能够自动执行事先编制好指令的机器。程序就是由指令组成的集合，程序加载到内存后处理器依次执行指令，完成人类赋予的工作。

最初，计算机只能依次一个一个地执行程序，执行完一个后再执行下一个，这时的计算机没有操作系统，由人工输入一个程序，执行完后再由人工输入下一个要执行的程序。后来，计算机有了批处理系统，即人工输入多个程序，多个程序排成队列，计算机执行完一个之后，自动调入下一个程序执行。再后来，计算机有了操作系统，系统内可以同时执行多个程序（执行中的程序称为进程），处理器在多个程序间跳转，每个程序执行一段时间，再跳到下一个程序执行，以达到多个程序同时执行的效果。

操作系统不仅管理着运行中的程序，还管理着系统的其它资源，如外部设备，文件系统等，程序本身也是操作系统管理的对象。操作系统内核执行系统资源、进程的管理和操作等核心功能，用户进程需要操作系统资源时，向内核提出申请，由内核代为执行，最后将结果返回给用户进程。内核本身可以看成一个程序，它提供多种功能供用户进程调用。内核加上一些必要的用户程序，如桌面管理程序，常用系统工具，文件系统等，就组成了一个完整的操作系统。

## 1.1操作系统

操作系统最核心的功能就是管理多个同时执行的程序（进程），以及对系统软硬件资源的管理和操作。下面先介绍处理器如何执行一个程序，再介绍如何同时执行多个程序，最后介绍操作系统和内核的功用。

### 1.1.1运行单个程序

计算机由五大部分组成：运算器、控制器、存储器、输入设备和输出设备。运算器和控制器位于处理器里，运算器主要完成算术和逻辑运算，里面包含寄存器和运算单元等，寄存器用于暂存运算数据。控制器用于控制指令、数据的读取，数据的写出，指令的解码，流水线控制等。存储器用于存放指令和数据。输入设备和输出设备称为外部设备，通常映射到处理器的虚拟地址空间，处理器可以像访问内存一样访问外部设备。

程序的执行如下图所示，程序的代码和数据保存在物理内存中，程序代码中的虚拟地址通过MMU（内存管理单元）映射到处理器物理内存空地址。代码和数据在物理内存中可以是分散存储的，MMU可以将其映射到连续的虚拟地址区域。由于程序代码中使用的是虚拟地址，所以程序觉察不到物理内存的分散。



处理器有一个程序指针PC，在硬件时钟的驱动下，顺序取出保存在代码区的指令，在控制器的作用下送入运算器执运算操作，指令执行完后通过控制器将结果写出内存或外设映射地址（寄存器）。

处理器只能识别二进制的机器指令，每条指令对应一个二进制编码，每条指令表示要执行的一个动作，如从内存取数据到运算器中的指定寄存器，将运算器指定寄存器中数据左移一位等。每条指令需要硬件逻辑的支持，每种类型的处理器只能识别、执行固定的一些指令，这些指令称为指令集（ISA）。指令集加上处理器中的控制逻辑的规范，中断处理的规范，处理器本身控制的一些规范等就组成了处理器体系结构（架构）。体系结构可视为实现具体处理器的规范或标准，流行的处理器体系结构有x84\_64、ARM等，本书关注的是国产龙芯处理器采用的MIPS体系结构，读者可参考相关手册。

通常我们用C、C++等高级语言编写应用程序，所有高级编程语言编写的程序代码都需要通过编译器编译成处理器的汇编源程序，然后通过汇编器将汇编源程序转换成能被处理器识别的二进制指令文件，最后通过链接器将各二进制指令文件合并成一个二进制可执行文件，以便加载到内存供处理器执行。在链接过程中通常将所有的代码段链接在一起，所有的数据段链接在一起，用户也可以在源代码中指定链接属性将代码或数据链接到指定的段。

### 1.1.2运行多个程序

如果处理器只运行一个程序，那就不需要操作系统了。因为系统的所有资源都属于这个进程，不需要进行管理。当有多个程序同时运行时，首先需要解决的是处理器在各程序之间的跳转问题。下图示意了系统中同时运行两个程序时的情形：



上图中程序1和程序2的代码和数据都加载到了内存中，分别位于内存中的不同位置。运行程序时，处理器通过虚拟地址访问物理内存，对于处理器来说系统中运行着两个进程，它们有着相同的虚拟地址空间，但是分别执行着程序1和程序2的代码。每个进程有自已的页表，通过MMU将相同的虚拟地址空间映射到不同的物理内存，以执行不同的代码。某一时刻，某个处理器核只能执行一个进程的代码（硬件超线程除外），如上图中所示，处理器在执行进程1时使用页表1访问物理内存，在执行进程2时使用页表2访问物理内存，以达到程序间的隔离。系统内需要保存页表1和页表2的信息，处理器在进程间切换时要保存当前进程的上下文信息，主要是此时处理器运算器中寄存器的信息，以便下次再次执行本进程时能够恢复运行，以及进程使用页表的位置。切换到下一个进程运行前，恢复下一个进程的上下文信息至处理器，并切换使用下一个进程的页表，而后处理器就可以执行下一个进程了。

实现运行多个进程的一个简单方法是将多个程序加载到内存，并建立页表，在系统中设置一个定时器，在定时器中断处理程序中保存当前进程的上下文信息，恢复下一个进程上下文信息至处理器，切换页表，中断返回。中断返回后执行的就是下一个进程了，也就是说由定时器中断来控制进程间的切换。

### 1.1.3操作系统实现

系统中如果运行着多个进程，如果没有有效的管理势必会带来混乱，而且如何来实现进程的动态创建和退出呢？为此，系统中引入了一个进程，称为内核，用于对系统内运行的进程进行有效管理。内核负责进程的创建、退出、进程间的切换、调度等，内核还有一项重要的任务是对系统资源的管理，如物理内存的管理，文件系统管理和操作，外部设备的管理和操作等。

从内核自身的角度来说，内核可视为一个系统资源的管理器，它管理着系统内的软硬件资源。进程需要某项资源或对某项资源进行操作时，向内核发出申请，由内核代为执行，并将执行结果返回给进程。由内核统一执行对系统资源的操作可提高操作的效率、避免冲突，并简化用户程序的编程。

从（用户）进程的角度来说，内核相当于一个函数库，提供各项操作的接口函数，进程需要执行某项操作时，调用内核提供的接口函数即可。内核提供的接口函数称为系统调用，用户进程通过中断（异常）的方式进入内核代码，执行系统调用。内核与用户进程的关系如下图所示：



现代处理器地址空间通常分为内核空间和用户空间，处理器只有处于核心态时才能访问内核空间，处理器处于用户态时只能访问用户空间。处理器处于核心态时才具有访问系统特权的资源（如外设，文件系统等），内核在处理器的核心态下运行，内核代码映射到处理器内核空间，以保证只有内核才能访问系统特权资源，确保安全。

在系统内，内核只有一个，因此内核地址空间至物理内存（内核镜像、内核代码）的映射是固定的，而用户进程有多个，不同的进程页表将用户地址空间映射到不同的物理内存。

用户进程通过指定的异常（中断）进入内核空间运行，此操作称为系统调用，在系统调用中用户进程会传递系统调用编号及参数。内核定义此异常（中断）处理函数，通过系统调用编号调用不同的实现函数，并接受用户空间传递的参数。

除了向用户进程提供系统调用，对进程本身的管理也是内核一项非常重要的工作，例如：创建进程、进程执行新的代码、进程切换、进程调度（何时切换以及切换至哪一个进程）、进程资源的管理、进程间通信、进程退出等等。

内核只封装了最基本和基础的资源管理和操作功能。发行版的操作系统在内核的基础上实现了一些常用和通用的系统工具，如创建目录、创建文件等，以方便用户空间编程。具有图形操作界面的操作系统还提供了一个桌面管理程序，以方便用户的操作。操作系统组成如下图所示：



本书主要介绍Linux内核（操作系统内核）的组成架构和功能，通过解读内核源代码使读者理解内核实现的机制，讲解设备驱动程的实现框架和内核移植的相关知识。

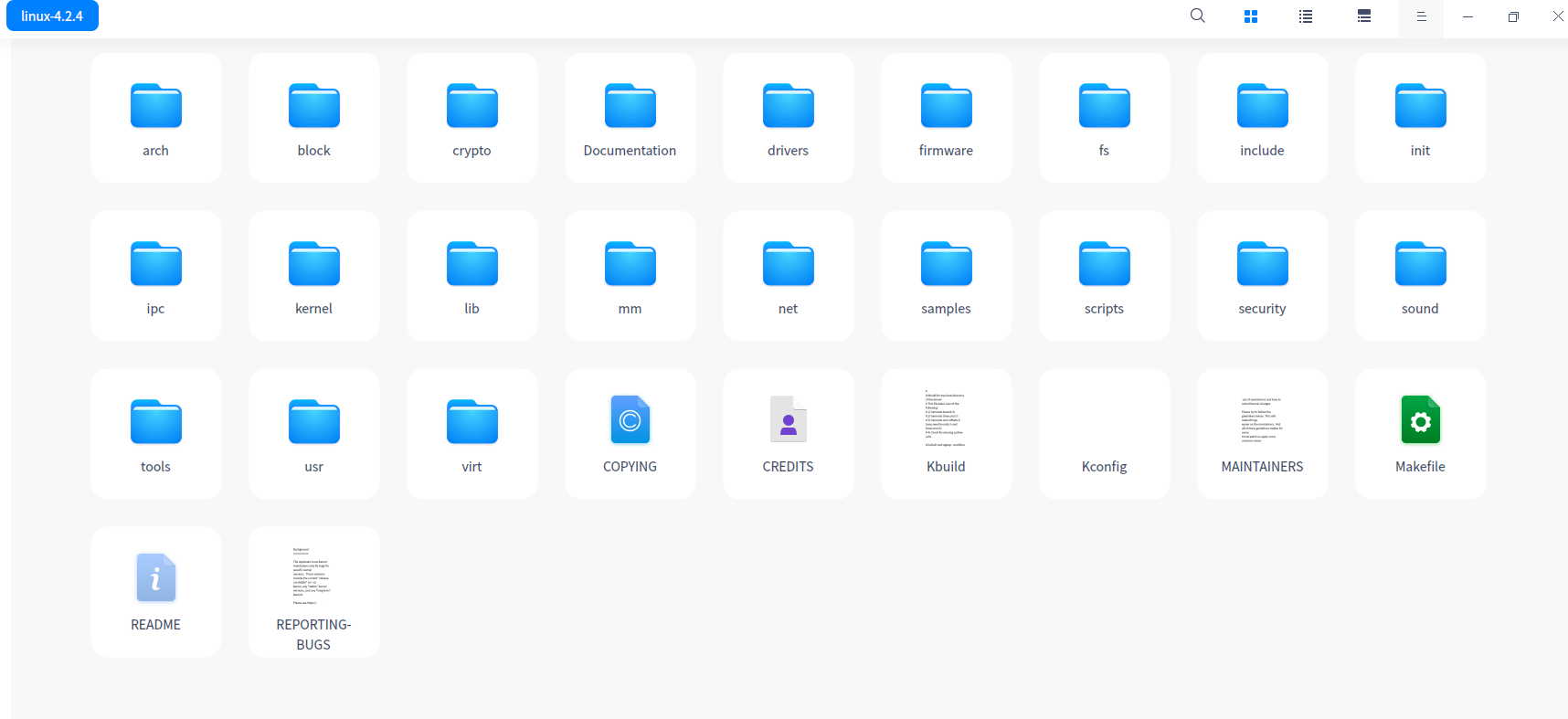
## 1.2 Linux内核

内核可视为一个程序，它由一组源代码文件通过编译、链接生成一个单一的可执行目标文件（二进制指令文件），在系统启动时加载到内存中运行。

本节先介绍Linux内核源代码文件的组成，然后介绍内核主要组成部分的功能。

### 1.2.1源代码目录

Linux内核源代码读者可从www.kernel.org网站直接下载，解压后得到内核源代码树，如下所示：



源代码主目录下包含的主要子目录名称及源代码内容简介如下：

|  |  |
| --- | --- |
| /arch | 体系结构相关代码，每种体系结构对应其下一个子目录，如/arch/mips。 |
| /block | 块设备驱动程序通用层代码。 |
| /crypto | 加密层文件，包含了各种密码算法的实现。 |
| /drivers | 设备驱动程序代码，包含通用驱动层代码和特定字符、块设备驱动程序代码。 |
| /firmware | 固件代码。 |
| /fs | 虚拟文件系统及具体文件系统类型代码。 |
| /include | 头文件。 |
| /init | 内核初始化代码。 |
| /ipc | 进程间通信机制实现代码。 |
| /kernel | 内核核心功能代码，如进程调度、中断等。 |
| /lib | 内核通用库例程代码，如基本数据结构的实现。 |
| /mm | 内存管理代码。 |
| /net | 网络协议层代码。 |
| /samples | 内核功能举例。 |
| /scripts | 编译、链接内核或进行其它任务的脚本和实用程序。 |
| /security | 安全框架和密钥管理代码。 |
| /sound | 声卡驱动程序代码。 |
| /tools | 内核配置编译过程中使用的工具程序。 |
| /usr | 用户层代码，主要用于编译根文件系统。 |
| /Documentation | 内核说明文档。 |

### 1.2.2内核架构

系统在启动时，将由引导加载程序（固件中的程序）将内核目标文件（镜像）从外部存储介质中加载到内存中并运行，内核将执行初始化操作，初始化操作完成后将创建并运行用户进程。

内核是系统中的大总管，管理、调配系统中的所有软、硬件资源，管理和调度进程的运行。内核在启动初始化阶段，将会创建管理各种资源的数据结构，如红黑树、链表、根文件系统等，并接受各种资源的注册和初始化函数，以实现对资源的管理。内核通过系统调用为用户进程提供资源使用、操作的申请，并负责执行这些申请。

Linux内核功能架构如下图所示：



Linux内核按功能可分为进程管理、内存管理、文件系统、设备驱动程序、网络层等组成部分。下面分别简要介绍各部分的功能。

#### 1进程管理

内核最终是为进程服务的，首先是对进程本身进行管理，其次是为进程提供各种资源及操作接口。进程起初是不存在的，内核在启动后期将创建第一个用户进程，并由此进程创建其它的用户进程。每个进程在内核中由task\_struct结构体实例表示，此结构体管理了进程所有的资源和信息。内核创建进程要为其创建虚拟地址空间管理结构，并从物理内存中为其分配物理内存（建立页表），从外部文件系统中导入程序代码和数据至内存，并将CPU控制权交给此进程，此进程才能得以运行。

除了内存外，进程另一个重要的资源是文件，Linux/UNIX奉行万物皆文件的哲学。进程通过文件接口访问外部介质文件系统中的文件、外部设备以及网络等资源。内核要为进程提供打开、关闭、操作文件的接口，普通文件系统、设备驱动程序、网络层都要实现文件操作的接口。

对进程本身的管理是影响系统性能的重要因素。内核提供了进程创建、退出、进程睡眠、进程间通信的机制。进程调度是进程管理的一项重要工作，进程调度是指何时需要进行进程切换，如何进行进程切换，以及切换的下一个进程如何选择。系统内运行着多个进程，不同的进程重要性、紧迫性不同，CPU运行时间如何在各种类型进程间分配是一个重要问题。

此外，内核还需要解决不同进程访问同一共享资源的竞争问题，以及进程其它资源的管理。

#### 2内存管理

物理内存是程序运行的基础，用于存放代码和数据，CPU从中读取指令和数据。系统中的进程在不断地创建和退出，因此需要不断地申请和释放内存。内核在运行过程中也需要不断地申请和释放内存。

内核为管理物理内存，将物理内存按页进行划分，页大小通常为4KB或8KB，此物理内存页称为页帧。内核在初始化阶段为每个页帧分配了page结构体实例（数组），用于管理和跟踪页帧。在page实例的基础上内核通过伙伴系统分配和释放物理上连续的页帧，一次分配/释放页帧的数量为2order，order称为分配阶，例如：order为0表示分配1个页帧，order为3表示分配8个连续的页帧。

内核总是按页（单页）为用户进程分配物理内存，内核自身可按不同的分配阶申请物理内存，只要不超过内核设置的最大分配阶即可。另外，内核运行过程中需要为各资源管理结构分配数据结构实例，此实例通常较小，用不了一页的物理内存。为此内核实现了分配小块内存的slub/slab/slob分配器，配置内核时在三者中选择其一，其中slub适用于大型系统，slob适用于资源有限的嵌入式系统，slab比较适中。这三个分配器都是从伙伴系统中申请内存，再划分成小块分配给内存各子系统。

以上介绍的伙伴系统和slub/slab/slob分配器是对物理内存的管理。

用户进程在处理器用户态下运行，访问用户地址空间，用户地址空间总是通过页表转换映射到物理内存。实际进程通常并不需要使用到全部的用户地址空间，内核将进程使用到的用户地址空间区域称为虚拟内存域，并对进程的虚拟内存域进行管理。用户进程的虚拟内存域通常并不是在进程运行时就全部映射到物理内存，而是在进程访问到此虚拟内存域时，再按需创建到物理内存的映射（按页创建）。用户地址空间的映射通常在缺页异常处理程序中完成。另外还可以将文件、外部设备等映射到进程虚拟内存域，以简化进程对文件和外部设备的操作。

内核在处理器内核态下运行，访问内核地址空间，内核地址空间分为直接映射区和间接映射区。直接映射区线性映射到（连续）的低端物理内存（物理内存底部），不通过页表转换（MIPS体系结构是这样，其它的不一定，如ARM还是需要通过页表映射）。间接映射区通过页表可映射到物理内存的任意位置，也是按页建立映射。直接映射区和间接映射区都可以通过伙伴系统为其分配映射页帧，不过直接映射区只能在低端物理内存中分配页帧，间接映射区可以在任意位置分配页帧，间接映射区分配页帧后需要修改内核页表，而直接映射区不需要。另外，slub/slab/slob分配器也只能在低端内存中分配页帧，分配的数据结构实例地址将线性映射到内核直接映射区。

用户进程虚拟内存域的管理、映射的创建和解除，以及内核直接映射区映射的创建和解除，统称为虚拟内存管理。

#### 3文件系统

Linux奉行万物皆文件的哲学，系统内的许多资源都通过文件来表示，例如：普通文件、外部设备、网络等。内核通过虚拟文件系统（VFS）管理系统的文件资源，并提供文件操作的接口。虚拟文件系统中包含一个由目录项（由dentry结构表示）构成的树状层次结构，称为根文件系统，目录项可表示普通的目录，也可表示文件名称。内核在初始化阶段会创建树状目录的根目录，名称为“/”。每个目录项关联到表示对应文件的节点（inode）结构体实例（VFS中文件由节点表示），普通的目录也关联节点，因为普通目录也是文件，只不过文件的内容是目录项而已。

根文件系统相当于系统中文件的仓库，目录项关联的文件节点结构中包含文件操作的接口，即file\_operations结构体实例，内核通过此实例中的操作函数对文件进行操作。某项资源如果想要内核通过文件接口对其进行操作，需要在根文件系统中创建对应的目录项和文件节点，并定义file\_operations实例，从而内核就可以通过通用的文件操作接口对其进行操作了。

不同类型的文件具有不同的读写操作方法，例如：外部介质中的文件系统具有不同的类型，具体文件系统类型需要按照file\_operations结构体的标准实现各操作函数。文件系统在被内核识别前需要挂载到根文件系统中某个目录项下，挂载操作会创建超级块和挂载点结构体实例，超级块结构中定义了将具体文件系统中文件导入到内核根文件系统中的方法，即创建目录项和文件节点的方法，并将file\_operations实例赋予表示文件的节点，以实现内核对文件的操作接口。也就是说，具体文件系统类型的代码需要实现将实际文件系统中的文件/目录导入到内核根文件系统中的方法，并定义文件/目录操作的接口。

进程打开文件实际就是到内核根文件系统（文件仓库）中去查找所需的文件，查找文件时按路径（目录项）在根文件系统中搜索，如果文件尚未被打开过，则需要从实际文件系统（介质文件系统）中导入文件至内核根文件系统。进程通过file结构体实例建立进程与根文件系统中文件inode节点之间的关联，而进程task\_struct结构体中包含打开文件file实例指针数组，数组索引值表示文件描述符（一个正整数），因此对于用户进程来说，文件按名称打开后，就通过文件描述符对其进行访问了。

内核支持几十种实际的文件系统类型，例如：ext2、ext3、ext4、FAT等。另外，内核定义了许多基于内存、内核数据结构实例的文件系统类型，用于向用户进程提供操作内核、获取内核信息的渠道。有了虚拟文件系统这个框架，向内核增加新的文件系统类型变得容易，内核越来越倾向于通过文件系统与用户进程进行通信。

#### 4设备驱动程序

外部设备在内核中也由文件表示，称为设备文件。按照虚拟文件系统的要求，驱动程序的主要工作就是创建设备文件，并实现文件操作接口file\_operations实例即可。对于内核来说，所有文件都是一样的，都采用相同的接口，它不会去区分是普通文件还是设备文件。

内核将外部设备分为字符设备和块设备两种类型，对每个设备赋予一个设备号，设备号由主设备号和从设备号组成。在创建设备文件时，设备号将写入设备文件节点inode实例中。内核分别为字符设备和块设备建立了设备数据库，字符设备由cdev结构体实例表示，块设备由gendisk结构体实例表示（两个结构体中包含设备号）。cdev和gendisk结构体实例由设备驱动程序创建和注册，cdev结构体中包含字符设备文件操作file\_operations实例，块设备gendisk结构体实例中需要实现读写块设备的队列和执行IO操作的方法。

字符设备驱动程序需要向内核申请设备号，创建并初始化cdev实例，定义file\_operations实例并赋予cdev实例，最后向内核注册cdev实例（注册实例时通常创建设备文件）。

块设备是按数据块进行读写的设备，块设备驱动程序中需要申请设备号，创建并初始化gendisk结构体实例，实现块设备操作的请求队列，最后注册gendisk实例。在注册gendisk实例的过程中内核将会扫描块设备中的分区，并为各分区创建设备文件。

内核可以直接通过块设备文件对块设备进行操作，也可以通过文件系统对块设备进行操作。通过文件系统进行操作时，具体文件系统类型的代码会将对文件内容的操作转化成对块设备中数据块的操作。最终，所有对块设备的操作都会转化成对块设备中数据块的操作，并封装成请求，提交到块设备请求队列，由块设备驱动程序实现数据块的传输。由于块设备读写操作比较缓慢，内核在内存中建立了块设备（文件内容）的缓存，以提高访问效率，缓存由内存页组成，由地址空间结构进行管理。

内核在打开设备文件时，根据节点中保存的设备类型和设备号，在设备数据库中查找cdev或gendisk结构体实例（或分区结构体实例）。对于字符设备，cdev实例中直接包含文件操作file\_operations实例，将其赋予进程文件file实例即可，此后进程通过此实例访问字符设备。对于块设备，内核定义了统一的文件操作file\_operations实例，对块设备的操作将会转换成对数据块的操作，以请求的形式提交到块设备请求队列。

#### 5网络层

网络设备是比较特殊的设备，内核不是通过设备文件访问网络设备，而是通过套接字访问网络。在创建套接字时也会关联到file实例，并赋予file\_operations实例，返回文件描述符。进程除了可以通过file\_operations实例访问网络外，还定义了一些网络专用的系统调用，以实现不能合并到file\_operations结构体中的操作。

内核网络层主要实现进程与进程间的通信，包括本机进程间的通信和不同主机进程间的通信。计算机网络用于将不同的主机连接在一起，以实现数据传输。不同的网络类型具有不同的数据传输协议，网络层主要是实现网络数据传输协议。

套接字对上层实现进程操作网络的接口，对下层连接网络传输协议，实现数据格式化和传输。网络设备负责将网络协议下传的数据发送出去，以及将接收到的数据提交给网络协议。网络协议层屏蔽了数据传输的细节，对于用户进程来说可以像操作普通文件一样操作网络数据。

最著名的计算机网络就是因特网，因特网将世界上数以亿计的主机连接在一起。因特网网络协议采用了分层的结构，主要分为应用层、传输层、网络层和数据链路层，内核主要实现后面三层，应用层由用户进程实现。传输层协议主要有UDP和TCP，用于实现主机进程间的通信，主机内的进程通过端口号进行识别。网络层协议为IPv4/IPv6，用于实现主机之间的通信，各主机通过IP地址寻址，网络中的路由器用于将数据发送到指定IP地址的主机上。数据链路层用于实现网络相邻节点间的数据传输，常用的协议有以太网、WiFi等，数据链路层各节点用设备MAC地址寻址。

发送网络数据时，通常需要指定目的主机的IP地址和进程端口号，即将数据发送到哪个主机上的哪个进程。因特网通过IP地址寻址到目的主机，在主机内由端口号寻找到接收进程，并将数据传递给该进程。

### 1.2.3移植内核

内核相当于一个运行中的程序，内核目标文件（镜像）由内核源代码编译、链接生成。Linux内核支持多种体系结构，以及许多的特性和设备驱动，在编译内核前需要对内核源代码进行配置，然后进行编译、链接生成单一的可执行目标文件。

要移植内核，要在用户主机上安装Linux发行版操作系统，下载内核压缩包，并解压获得内核源代码树。内核配置就是要选择哪些源文件进行编译、选择函数实现方式、源文件是持久编译入内核还是编译成模块、选择设备驱动程序，以及内核属性的选择等等。内核提供了Kconfig机制用于配置内核，配置选项由Kconfig语言编写，各内核源代码目录下几乎都包含一个Kconfig文件，用于设置内核的配置选项。在源代码目录下执行make menuconfig等命令将启动内核配置，内核将向用户提供一个配置界面，用于选择配置选项，配置完成后保存配置结果。

内核构建就是根据配置结果，对内核源代码文件进行选择性编译和链接，最终生成可执行目标文件和模块。内核通过Kbuild机制实现内核的构建，几乎每个源代码目录下都有一个Makefile文件，文件内根据配置结果，确定要编译本目录和子目录下的哪些文件，确定是持久编译入内核目标文件还是编译成模块，或是不编译。最终，所有目录下编译生成的目标文件将链接成一个单一的内核目标文件，用于加载到目标机内存中运行。

在移植内核过程中，最主要的工作是为目标机外部设备选择或编写设备驱动程序。另外，还需要在目标机外部存储介质中制作根文件系统，根文件系统中需包含第一个用户进程目标文件、系统配置文件、系统常用工具等。

## 1.3基本数据结构

在分析内核源代码之前，有必要先了解内核中常用的基本数据结构类型，它们在内核代码中广泛使用。内核并没有使用很复杂的数据结构，就是的简单的单链表、双链表、散列表、红黑树、基数树等。数据结构用来管理内核中的对象，主要是各种类型的数据结构实例，如表示进程的task\_srtuct结构、表示内核文件的inode结构等等。本小节对以上数据结构做简要的介绍，为后面阅读内核源代码做准备。

### 1.3.1容器

container\_of(ptr, type, member)宏定义在/include/linux/kernel.h（L812）头文件：

#define container\_of(ptr, type, member) ({ \

const typeof( ((type \*)0)->member ) \*\_\_mptr = (ptr); \

(type \*)( (char \*)\_\_mptr - offsetof(type,member) );})

member为type类型数据结构中成员名，ptr为指向type实例中member成员的指针。container\_of()宏用于由成员指针获取包含它的type类型数据结构实例的指针。

例如：vm\_area\_struct 结构体中包含红黑树节点rb\_node结构体成员，成员名称为vm\_rb：

struct vm\_area\_struct {

...

struct rb\_node vm\_rb;

...

}

ptr为指向vm\_area\_struct 结构实例中vm\_rb成员的指针，container\_of(ptr,vm\_area\_struct,vm\_rb)宏返回对应的vm\_area\_struct 结构实例指针，如下图所示：



### 1.3.2链表

链表是内核广泛使用的简单数据结构，包括单链表、双链表、内核链表、散列表等。

#### **1单链表**

单链表用于将结构实例链接成单向的链表，单向链表数据结构定义在/include/linux/llist.h头文件：

struct llist\_head {

struct llist\_node \*first; /\*第一个节点指针\*/

};

llist\_node结构表示链表头，表头只包含一个指向节点的first指针成员。单链表节点数据结构定义如下：

struct llist\_node {

struct llist\_node \*next; /\*指向下一节点指针\*/

};

节点成员中也只包含指向下一节点的指针成员，节点结构通常嵌入到被管理数据结构中。单链表结构如下图所示：



内核/include/linux/llist.h头文件声明了单链表的通用操作函数：

#define LLIST\_HEAD\_INIT(name) { NULL }

#define LLIST\_HEAD(name) struct llist\_head name = LLIST\_HEAD\_INIT(name)

/\*定义并初始化单链表头实例\*/

●bool llist\_empty(const struct llist\_head \*head)：测试链表是否为空。

●struct llist\_node \*llist\_next(struct llist\_node \*node)：获取node下一个链表元素。

●bool llist\_add(struct llist\_node \*new, struct llist\_head \*head)：插入节点。

●struct llist\_node \*llist\_del\_first(struct llist\_head \*head)：删除第一个节点。

#### **2双链表**

双链表是内核中大量使用的简单数据结构，用于将对象串联进行管理。双链表表头及节点数据结构相同，定义在/include/linux/types.h头文件内：

struct list\_head {

struct list\_head \*next, \*prev; /\*next：指向下一节点，prev：指向前一节点\*/

};

结构体中包含两个成员，分别是指向前一个节点和后一个节点的指针。list\_head结构通常嵌入被管理数据结构中，内核通常会单独定义表示链表头的list\_head结构实例。双链表结构如下图所示，双向链表的首尾成员也会相互连接组成环形链表：



内核在/include/linux/list.h头文件中定义了双链表的基本操作函数，例如：

#define LIST\_HEAD\_INIT(name) { &(name), &(name) } /\*双链表节点初值，两个成员都指向自身\*/

#define LIST\_HEAD(name) \

struct list\_head name = LIST\_HEAD\_INIT(name) /\*定义并初始化双链表节点\*/

#define list\_entry(ptr, type, member) container\_of(ptr, type, member)

/\*由list\_head指针获取被管理对象实例指针\*/

#define list\_for\_each\_entry(pos, head, member) \

for (pos = list\_first\_entry(head, typeof(\*pos), member); \

&pos->member != (head); \

pos = list\_next\_entry(pos, member)) /\*遍历head表头表示链表的管理对象实例，pos为指针\*/

●void list\_add(struct list\_head \*new, struct list\_head \*head)：将new节点插入到head节点**后面**。

●void list\_add\_tail(struct list\_head \*new, struct list\_head \*head)：将new节点插入到head节点**前面（与字面意思不同）**。

●void list\_del(struct list\_head \*entry)：从双链表移除entry节点。

#### 3内核链表

内核链表可理解为带自旋锁的双链表，链表头由klist结构表示，定义在/include/linux/klist.h头文件：

struct klist {

**spinlock\_t k\_lock**; /\*自旋锁\*/

**struct list\_head k\_list**; /\*双链表头\*/

void (\*get) (struct klist\_node \*); /\*增加成员引用计数\*/

void (\*put) (struct klist\_node \*); /\*减少成员引用计数\*/

} \_\_attribute\_\_ ((aligned (sizeof(void \*))));

表头结构中带有保护链表的自旋锁的k\_lock成员，用于防止内核多条路径同时修改链表，get和put为回调函数指针，用于增加或减少节点成员的引用计数。

内核链表节点结构为klist\_node，定义在/include/linux/klist.h头文件：

struct klist\_node {

void \*n\_klist; /\*指向表头klist结构实例，不可直接访问\*/

**struct list\_head n\_node**; /\*双链表节点成员\*/

struct kref n\_ref; /\*链表成员的引用计数，原子变量\*/

};

内核在/include/linux/klist.h头文件定义了初始化链表头的宏：

#define KLIST\_INIT(\_name, \_get, \_put) \

{ \

.k\_lock = \_\_SPIN\_LOCK\_UNLOCKED(\_name.k\_lock), \

.k\_list = LIST\_HEAD\_INIT(\_name.k\_list), \

.get = \_get, \

.put = \_put, \

}

#define DEFINE\_KLIST(\_name, \_get, \_put) \ /\*定义并初始化链表头\*/

struct klist \_name = KLIST\_INIT(\_name, \_get, \_put)

内核在/lib/klist.c文件内定义了链表操作的通用函数，例如：

●void klist\_add\_head(struct klist\_node \*n, struct klist \*k)：将n节点添加到k表示链表的头部。

●void klist\_add\_tail(struct klist\_node \*n, struct klist \*k)：将n节点添加到k表示链表的尾部。

●void klist\_del(struct klist\_node \*n)：从链表中删除节点n 。

内核在/include/linux/klist.h头文件还定义了klist\_iter结构，内部包含指向内核链表头的指针成员，以及指向当前链表节点的指针成员，用于迭代定义如下：

struct klist\_iter {

struct klist \*i\_klist; /\*内核链表头指针\*/

struct klist\_node \*i\_cur; /\*链表中当前节点指针\*/

};

在引入klist\_iter结构后的内核链表，结构如下图所示：



●struct klist\_node \*klist\_next(struct klist\_iter \*i)：用于将前一节点引用计数减1，后一节点引用计数加1，返回后一节点指针。

#### 4散列表

散列表（哈希表）是内核中大量使用的数据结构，散列表类似于双链表数组。管理对象按某一特性（散列函数）进行分类，散列表中每一个链表项通过双链表链接同一类型的对象。散列表结构如下图所示：



散列表头及散列表节点数据结构定义在/include/linux/types.h头文件内：

struct hlist\_head { /\*散列表头\*/

struct hlist\_node \*first; /\*指向节点的指针\*/

};

散列表节点数据结构定义如下：

struct hlist\_node {

struct hlist\_node \*next, \*\*pprev; /\*next指向下一个节点，pprev指向前一节点的next成员\*/

};

散列表头hlist\_head结构中只包含指向节点hlist\_node结构的指针，在散列表（数组）较大时可减小内存占用量。散列表节点成员为hlist\_node结构，其中next成员指向下一节点hlist\_node结构，而pprev成员是指向指针的指针，其值保存的是前一节点hlist\_node结构next成员的地址。

散列表节点hlist\_node结构通常内嵌到被管理对象数据结构的内部。内核在用散列表管理对象时，通常定义散列表头数组，例如：hlist\_head[n]，数组中包含n个散列表头，被管理对象数据结构实例，通过某种算法关联到某一数组项对应链表中。例如：以对象实例地址address对n取模确定对象链接的链表头，address%n取模函数称为散列函数。散列表相当于将对象分类进行管理。

内核在/include/linux/list.h头文件中定义了hlist链表的基本操作函数，例如：

#define HLIST\_HEAD\_INIT { .first = NULL } /\*散列表头初始值\*/

#define HLIST\_HEAD(name) struct hlist\_head name = { .first = NULL } /\*定义并初始化散列表头\*/

#define INIT\_HLIST\_HEAD(ptr) ((ptr)->first = NULL) /\*初始化指定散列表头\*/

#define hlist\_entry(ptr, type, member) container\_of(ptr,type,member) /\*hlist\_node转对象数据结构\*/

#define hlist\_for\_each(pos, head) \

for (pos = (head)->first; pos ; pos = pos->next) /\*遍历散列表头链表中hlist\_node成员\*/

#define hlist\_for\_each\_entry(pos, head, member) \

for (pos = hlist\_entry\_safe((head)->first, typeof(\*(pos)), member);\

pos; \

pos = hlist\_entry\_safe((pos)->member.next, typeof(\*(pos)), member))

/\*遍历散列表头链表管理的对象数据结构实例，pos为指向对象指针\*/

●void hlist\_add\_head(struct hlist\_node \*n, struct hlist\_head \*h)：将n指向节点添加到h散列表头链表的头部。

●void hlist\_del(struct hlist\_node \*n)：从散列链表中移除n指向的节点。

内核还定义了散列表的一个改进版本，即对散列表头成员加锁，在操作表头指向的链表时表头指针成员最低位置1，其它时候清零。数据结构定义在/include/linux/list\_bl.h头文件内（bl意为bit lock）：

struct hlist\_bl\_head {

struct hlist\_bl\_node \*first;

};

struct hlist\_bl\_node {

struct hlist\_bl\_node \*next, \*\*pprev;

};

向链表添加删除成员的函数如下：

●void hlist\_bl\_add\_head(struct hlist\_bl\_node \*n,struct hlist\_bl\_head \*h)：插入节点操作，函数内部在操作前对表头加锁，在操作完成后解锁。

●void hlist\_bl\_del(struct hlist\_bl\_node \*n)：删除节点操作，函数内部在操作前对表头加锁，在操作完成后解锁。

#define hlist\_bl\_for\_each\_entry(tpos, pos, head, member) \ /\*遍历链表管理的对象实例\*/

for (pos = hlist\_bl\_first(head); \

pos && \

({ tpos = hlist\_bl\_entry(pos, typeof(\*tpos), member); 1;}); \

pos = pos->next)

### 1.3.3基数树

基数树是内核中以库方式实现的数据结构，基数树是一种用来管理对象数据结构的树结构，基数树通过数据结构的冗余来提高查找管理对象的速度。内核中使用基数树的典型例子就是地址空间，基数树结构如下图所示：



基数树中数据结构包括两种一种是作为树节点的radix\_tree\_node结构，以及树管理的对象数据结构，例如内存页结构page。图中radix\_tree\_root结构一般内嵌于其它结构中，用于指向树根节点。

基数树各数据结构定义在/include/linux/radix-tree.h头文件，树根结点结构定义如下：

struct radix\_tree\_root {

unsigned int height; /\*树的高度\*/

gfp\_t gfp\_mask; /\*指定从哪个内存域分配内存用于创建节点数据结构\*/

struct radix\_tree\_node \_\_rcu \***rnode**; /\*指向树的根节点\*/

};

基数树节点数据结构定义如下：

struct radix\_tree\_node {

unsigned int path; /\*从底部开始计算树的高度\*/

unsigned int count; /\*节点中已经使用的数组项数目\*/

union {

struct {

struct radix\_tree\_node \*parent; /\*指向父节点\*/

void \*private\_data; /\* For tree user \*/

};

struct rcu\_head rcu\_head; /\* Used when freeing node \*/

};

struct list\_head private\_list; /\* For tree user \*/

**void \_\_rcu \*slots[RADIX\_TREE\_MAP\_SIZE]**; /\*数据项指针数组\*/

**unsigned long tags[RADIX\_TREE\_MAX\_TAGS][RADIX\_TREE\_TAG\_LONGS]**; /\*标签数组\*/

};

节点结构中比较重要的成员是数据项指针数组slot[RADIX\_TREE\_MAP\_SIZE]及标签二维数组tags[][]。

数据项指针数组可指向节点radix\_tree\_node结构或管理对象结构，只有基数树最底部的节点数据项指针才会指向管理结构（称为叶子），中间层节点都指向节点结构。数据项指针数组项数为RADIX\_TREE\_MAP\_SIZE，定义如下：

#ifdef \_\_KERNEL\_\_

#define **RADIX\_TREE\_MAP\_SHIFT (CONFIG\_BASE\_SMALL ? 4 : 6)**

#else

#define RADIX\_TREE\_MAP\_SHIFT 3 /\* For more stressful testing \*/

#endif

#define **RADIX\_TREE\_MAP\_SIZE (1UL << RADIX\_TREE\_MAP\_SHIFT)**

/\*结点对象指针数量\*/

#define RADIX\_TREE\_MAP\_MASK (RADIX\_TREE\_MAP\_SIZE-1)

数组项数目RADIX\_TREE\_MAP\_SIZE为RADIX\_TREE\_MAP\_SHIFT个二进制位能表示的最大值加1，例如：RADIX\_TREE\_MAP\_SHIFT取值为3时，数组项数目为23=8，三位二进制数能表示的最大值为7，数组项最大索引值也为7。

基数树结构中采用整型数来索引管理的对象，对象在树的底部依次按索引值从左至右排列。这里以RADIX\_TREE\_MAP\_SHIFT为6为例来解释对象在基数树中的索引过程，如下图所示。由于树的每一层都需要RADIX\_TREE\_MAP\_SHIFT个二进制位来表示节点（对象）索引号，因此取值为6时树的最大高度值为5（32/6），基数树最大管理对象数量为25\*6。

因此对索引id位域进行如下图中所示的划分，bit[29...24]表示要根节点slots[]数组中的索引值，依次类推bit[5...0]表示对象在最后第5层slot[]数组中的索引值。



基数树刚建立初始化时并没有创建最大的高度而是在插入对象时按需扩大树的层级。向基数树插入管理对象的函数：

int radix\_tree\_insert(struct radix\_tree\_root \*root,unsigned long index, void \*item)：向基数树添加管理对象数据项，root表示根节点，index表示对象索引值，item为对象指针，成功返回0。

在插入对象前函数会判断索引值index有没有超过当前树管理的最大对象数目，如果超过则需要扩展树，扩展树的方法是在根节点和后层节点中插入新的节点以提高树的高度，随后再找到对象正确的位置建立关联，节点结构也是动态创建的。

void radix\_tree\_delete(struct radix\_tree\_root \*root, unsigned long index)：删除某一索引值对象数据项。

void \*radix\_tree\_lookup(struct radix\_tree\_root \*root, unsigned long index)：由索引值查找查找对象数据项，返回对象指针。查找工作原理上面介绍过了，只需将index整型数位域按RADIX\_TREE\_MAP\_SHIFT宽度进行划分，从高到低将位域值作为树层的指针数组索引，即可快速查找到所需对象。

基数树节点radix\_tree\_node结构中另一个重要成员是tags[][]二维标签数组：

struct radix\_tree\_node {

...

**void \_\_rcu \*slots[RADIX\_TREE\_MAP\_SIZE]**;

**unsigned long tags[RADIX\_TREE\_MAX\_TAGS][RADIX\_TREE\_TAG\_LONGS]**; /\*标签数组\*/

};

标签数组项中每一位对应节点指针数组slots[RADIX\_TREE\_MAP\_SIZE]中的一项，内核支持的标签类型数量为3，定义如下：

#define RADIX\_TREE\_MAX\_TAGS 3 /\*标签类型\*/

#define RADIX\_TREE\_TAG\_LONGS \ /\*每类标签位图所占的整型数数量\*/

((RADIX\_TREE\_MAP\_SIZE + BITS\_PER\_LONG - 1) / BITS\_PER\_LONG)

标签数组与指针数组对应关系如下图所示：



基数树中每个radix\_tree\_node实列中都包含二维的标签数组成员。若某一叶节点具有某项属性，则需要设置叶节点所在radix\_tree\_node实列中对应该对象的相应类型的标签位，并且沿着根节到本叶节点的所有radix\_tree\_node实列中相应类型的标签位都需要设置。清除标签位则需要slots[]数组项下的所有子节点（叶节点）都不具有此类型属性时才能清除。设置和清除标签位的函数如下：

●radix\_tree\_tag\_set(struct radix\_tree\_root \*root,unsigned long index, unsigned int tag)：设置标签标记，tag参数为标记类型索引值，index为基数树中对象（叶节点）索引值，下同。

●radix\_tree\_tag\_clear(struct radix\_tree\_root \*root,unsigned long index, unsigned int tag)：清除标签标记。

●radix\_tree\_tag\_get(struct radix\_tree\_root \*root,unsigned long index, unsigned int tag)：获取标签标记值，返回0或1。

内核在start\_kernel(void)函数内调用radix\_tree\_init(void)函数对基数树数据结构进行初始化，代码如下：

void \_\_init radix\_tree\_init(void)

{

radix\_tree\_node\_cachep = kmem\_cache\_create("radix\_tree\_node",sizeof(struct radix\_tree\_node), 0,

SLAB\_PANIC | SLAB\_RECLAIM\_ACCOUNT,radix\_tree\_node\_ctor); /\*创建slub缓存\*/

**radix\_tree\_init\_maxindex()**; /\*初始化最大管理对象数组，/lib/radic\_tree.c\*/

hotcpu\_notifier(radix\_tree\_callback, 0);

}

### 1.3.4 idr

idr是一种与基数树类似的数据结构，也是用于通过对象ID来管理对象，idr结构中相当于建立了稀疏的数组结构。由前面基数树的结构做基础，我们先来看一下idr数据结构管理对象的结构，如下图所示。idr结构也是分层的结构，idr结构表示根节点，管理对象位于最底层，每一层的数据结构为idr\_layer，该结构中主要包含ary[]指针数组，数组项数IDR\_SIZE为1 << IDR\_BITS，其中IDR\_BITS为8。其实就是将id号按每8位进行分段，每个位段代表该层对象的索引。idr结构理论上最多可管理4G数量的对象。



idr及idr\_layer数据结构定义在/include/linux/idr.h头文件：

struct idr {

struct idr\_layer \_\_rcu \*hint; /\* the last layer allocated from \*/

struct idr\_layer \_\_rcu \*top;

int layers; /\* only valid w/o concurrent changes \*/

int cur; /\* current pos for cyclic allocation \*/

spinlock\_t lock;

int id\_free\_cnt;

struct idr\_layer \*id\_free;

};

struct idr\_layer {

int prefix; /\* the ID prefix of this idr\_layer \*/

int layer; /\* distance from leaf \*/

struct idr\_layer \_\_rcu \*ary[1<<IDR\_BITS];

int count; /\* When zero, we can release it \*/

union {

DECLARE\_BITMAP(bitmap, IDR\_SIZE); /\*位图，表示当层及下层指针数组空闲状态\*/

struct rcu\_head rcu\_head;

};

};

内核在/lib/idr.c文件内定义了idr数据结构的操作函数，idr\_init\_cache()函数在内核启动时调用，定义如下：

void \_\_init idr\_init\_cache(void)

{

idr\_layer\_cache = kmem\_cache\_create("idr\_layer\_cache", \

sizeof(struct idr\_layer), 0, SLAB\_PANIC, NULL); /\*为idr\_layer结构创建slab缓存\*/

}

使用idr数据结构时，需要定义idr结构实例，并调用idr\_init()函数对其进行初始化：

void idr\_init(struct idr \*idp)

{

memset(idp, 0, sizeof(struct idr));

spin\_lock\_init(&idp->lock);

}

内核也定义了定义和初始化idr实例的宏：

**#define DEFINE\_IDR(name) struct idr name = IDR\_INIT(name)**

#define IDR\_INIT(name) \

{ \

.lock = \_\_SPIN\_LOCK\_UNLOCKED(name.lock), \

}

向idr结构插入对象的函数为：

**int idr\_alloc(struct idr \*idp, void \*ptr, int start, int end, gfp\_t gfp\_mask)**：idp为idr实例指针，ptr为管理对象数据结构实例指针，[start,end)表示可分配id号的区间，gfp\_mask为分配内存掩码，函数成功返回对象分配的id号。

**void \*idr\_find\_slowpath(struct idr \*idp, int id)**：从idr结构中由id号查找管理对象，返回对象指针，若对象为空，返回NULL。

**void idr\_remove(struct idr \*idp, int id)：**从idr中移出id表示的对象。

### 1.3.5红黑树

红黑树是内核使用的一种非常重要的数据结构，非常多的内核重要的数据结构都由红黑树管理，它在速度和实现复杂度之间提供了一种很好的平衡。红黑树是同时具有以下特征的自平衡二叉查找树：

（1）树中节点是红色或者黑色；

（2）根节点是黑色的；

（3）红节点的所有子节点是黑色的；

（4）每条从根节点到叶节点的简单路径上包含相同数目的黑节点。

二叉树中每个节点最多只有两个子节点，且节点的左子节点键值比节点键值小，右子节点键值比节点键值大。红黑树满足二叉树的条件，另外每个节点还带着色，或红或黑。树中不能有两个连续的红色节点（父子节点不能同时为红色），但可以有任意连续的黑色节点。在红黑树中插入和删除节点时需要保持树的平衡性，即不能有某一节点的深度远大于其它节点。下图示意了一个红黑树的例子，节点旁边数字是节点表管理对象的键值：



内核在/include/linux/rbtree.h头文件内定义了红黑树根、节点数据结构：

struct rb\_root { /\*红黑树根节点指针\*/

struct rb\_node \*rb\_node; /\*指向根节点\*/

};

struct rb\_node { /\*红黑树节点\*/

unsigned long \_\_rb\_parent\_color; /\*父节点颜色，包含父节点指针\*/

struct rb\_node \*rb\_right; /\*指向右子节点\*/

struct rb\_node \*rb\_left; /\*指向左子节点\*/

} \_\_attribute\_\_((aligned(sizeof(long))));

红黑树节点结构\_\_rb\_parent\_color成员最低位用于表示父节点颜色（0红，1黑），屏蔽低两位后表示父节点指针。

内核中以库函数的形式实现红黑树的通用操作函数，如插入、删除节点等，函数实现在/lib/rbtree.c文件内，下面列出其中主要的通用操作函数：

●void rb\_link\_node(struct rb\_node \*node, struct rb\_node \*parent,struct rb\_node \*\*rb\_link)：关系父子节点，node节点父节点设为parent，rb\_link指向nod实例的指针地址（\*rb\_link = node）。

●void rb\_insert\_color(struct rb\_node \*, struct rb\_root \*)：插入节点，插入前需调用rb\_link\_node()函数关联父节点。

●void rb\_erase(struct rb\_node \*, struct rb\_root \*)：从红黑树中删除节点。

●struct rb\_node \*rb\_next(const struct rb\_node \*)：查找指定节点的下一个节点，遍历树时使用。

●struct rb\_node \*rb\_prev(const struct rb\_node \*)：查找指定节点的前一个节点。

●struct rb\_node \*rb\_first(const struct rb\_root \*)：查找树中第一个节点。

●struct rb\_node \*rb\_last(const struct rb\_root \*)：查找树中最后一个节点。

红黑树节点结构rb\_node一般内嵌在被管理数据结构内部，如进程虚拟内存域结构：

struct vm\_area\_struct {

unsigned long vm\_start; /\*内存域起始地址\*/

unsigned long **vm\_end**; /\*内存域起始地址\*/

...

**struct rb\_node vm\_rb**; /\*红黑树节点\*/

...

}

由rb\_node实例指针通过容器机制可获得被管理数据结构实例指针：

#define rb\_entry(ptr, type, member) container\_of(ptr, type, member) /\*/include/linux/rbtree.h\*/

/\*ptr：rb\_node指针，type：管理结构指针，rb\_node在type内成员名称\*/

例如，假设ptr是指向rb\_node实例指针，rb\_entry(ptr,vm\_area\_struct,vm\_rb)返回的就是包含ptr指向的rb\_node实例的vm\_area\_struct实例指针。

管理对象数据结构中某一成员被当作键值，用作对象红黑树节点成员在红黑树中排序的依据。例如：虚拟内存域vm\_area\_struct实例中以结束地址vm\_end为键值，将所含的节点成员vm\_rb按照实例中vm\_end值从小到大在红黑树中从左至右排序。如下图所示，树中左侧节点vm\_end值最小，右侧节点vm\_end值最大。



查找红黑树节点操作由应用红黑树的代码实现，查找符合某一条件节点时，以给定键值为基准，从红黑树根节点开始查找管理对象实例，比给定键值小的对象在节点的左子树中，比给定键值大的对象在节点的右子树中，递归查找即可。例如：在vm\_area\_struct实例红黑树中查找结束地址在addr之后的第一个实例时，从根节点开始通过容器机制获取节点代表的vm\_area\_struct实例，再将实例中vm\_end成员值与addr比较，vm\_end比addr小的实例在节点左子树中，比addr大的实例在节点右子树中，如此递归查找即可。

将管理对象数据结构实例插入红黑树之前，先要依键值查找红黑树确定插入的位置，即父节点，而后依次调用rb\_link\_node()和rb\_insert\_color()函数将节点插入红黑树。

删除操作获取管理对象数据结构实例rb\_node成员指针后，调用rb\_erase()函数从红黑树中删除节点。在通用的插入/删除节点操作函数中，在插入、删除对象后需要对树进行旋转、改变节点颜色等操作，以满足红黑树的条件，有兴趣的读者可阅读/lib/rbtree.c文件内源代码。

若红黑树节点中包含额外的数据信息，可使用被称为扩展的红黑树。扩展的红黑树在向树中添加/删除节点时会回调用户提供的函数，以处理节点扩展的数据信息。若要使用扩展的红黑树，程序代码中需包含头文件/include/linux/rbtree\_augmented.h，在头文件中定义了传递回调函数的数据结构：

struct rb\_augment\_callbacks {

void (\*propagate)(struct rb\_node \*node, struct rb\_node \*stop);

void (\*copy)(struct rb\_node \*old, struct rb\_node \*new);

void (\*rotate)(struct rb\_node \*old, struct rb\_node \*new);

};

●propagate：更新node节点及其所有祖先节点的扩展信息（stop==NULL），或直至stop指向的节点；

●copy：将old节点的扩展信息复制给new的节点；

●rotate：将old节点的扩展信息复制给new的节点，并重新计算old节点的扩展信息。

扩展的红黑树与标准的红黑树之间的区别在于插入、删除节点的调用函数不同。

●void rb\_insert\_augmented(struct rb\_node \*node, struct rb\_root \*root,const struct rb\_augment\_callbacks \*augment)：插入节点接口函数，augment参数为rb\_augment\_callbacks实例指针，函数调用关系如下图所示：



函数内在执行标准的红黑树插入节点操作后，会调用rb\_augment\_callbacks实例内的rotate()函数。标准的红黑树插入节点函数rb\_insert\_color()在内部调用的是同样的执行函数\_\_rb\_insert()，只不过此时调用的rotate()函数为空函数。

●void rb\_erase\_augmented(struct rb\_node \*node, struct rb\_root \*root,const struct rb\_augment\_callbacks \*augment)：删除节点函数，内部可能会调用rb\_augment\_callbacks实例定义的三个函数。

内核在头文件/include/linux/rbtree\_augmented.h给出了定义rb\_augment\_callbacks实例的宏：

#define RB\_DECLARE\_CALLBACKS(rbstatic, rbname, rbstruct, rbfield, \

rbtype, rbaugmented, rbcompute) \

/\*

\*rbstatic：rb\_augment\_callbacks实例修饰符，通常为static；rbname：rb\_augment\_callbacks实例名称；

\*rbstruct：节点管理数据结构名称，如vm\_area\_struct；

\*rbfield：rbstruct结构中rb\_node成员名称，如vm\_rb；

\*rbtype：表示扩展信息的数据类型，如unsigned long；

\*rbaugmented：表示扩展信息的rbstruct数据结构成员，如rb\_subtree\_gap；

\*rbcompute：重新计算扩展信息的函数指针。

\*/

static inline void \

rbname ## \_propagate(struct rb\_node \*rb, struct rb\_node \*stop) \ /\*定义propagate()函数\*/

{ \

while (rb != stop) { \

rbstruct \*node = rb\_entry(rb, rbstruct, rbfield); \

rbtype augmented = rbcompute(node); \

if (node->rbaugmented == augmented) \

break; \

node->rbaugmented = augmented; \

rb = rb\_parent(&node->rbfield); \

} \

} \

static inline void \

rbname ## \_copy(struct rb\_node \*rb\_old, struct rb\_node \*rb\_new) \ /\*定义copy()函数\*/

{ \

rbstruct \*old = rb\_entry(rb\_old, rbstruct, rbfield); \

rbstruct \*new = rb\_entry(rb\_new, rbstruct, rbfield); \

new->rbaugmented = old->rbaugmented; \

} \

static void \

rbname ## \_rotate(struct rb\_node \*rb\_old, struct rb\_node \*rb\_new) \ /\*定义rotate()函数\*/

{ \

rbstruct \*old = rb\_entry(rb\_old, rbstruct, rbfield); \

rbstruct \*new = rb\_entry(rb\_new, rbstruct, rbfield); \

new->rbaugmented = old->rbaugmented; \

old->rbaugmented = rbcompute(old); \

} \

rbstatic const struct rb\_augment\_callbacks rbname = { \ /\*定义rb\_augment\_callbacks实例\*/

rbname ## \_propagate, rbname ## \_copy, rbname ## \_rotate \

};

宏内部定义了rb\_augment\_callbacks结构体实例及其内部的三个函数，rbcompute参数表示计算节点扩展信息的函数指针。应用扩展红黑树时，需定义rbcompute()函数，调用以上宏定义rb\_augment\_callbacks结构体实例，随后即可调用rb\_insert\_augmented()和rb\_erase\_augmented()函数插入和删除节点，注意需要将rb\_augment\_callbacks实例的指针传递给函数。第4章进程虚拟内存域vm\_area\_struct实例的管理就是应用扩展红黑树的一个例子。

## 1.4小结

本章首先介绍了作者对计算机、操作系统的一些理解，然后介绍了Linux内核的框架、各部分组成及功用，最后介绍了内核常用的数据结构，以便为后面内核源代码的分析打下基础。

通过本章的学习，希望读者能对内核的框架和主要功能有个整体的认识。内核源代码庞大而复杂，最好采取先整后分的学习方法，先对内核各部分（子系统）的框架和功用有个整体的认识，了解各部分之间的关联，然后再去详细研究各功能（子系统）函数代码的实现。没有整体的认识，很容易就迷失在众多的代码细节之中。

内核源代码的学习是一项艰苦而卓绝的工程，需要超强的耐心和毅力，以及时刻清醒的头脑。下一章将真正开始内核源代码的学习，你准备好了吗？准备好了就开始吧！