《openEuler内核编程》

课程讲稿

第二章 第3讲

Kunpeng下的内核启动过程

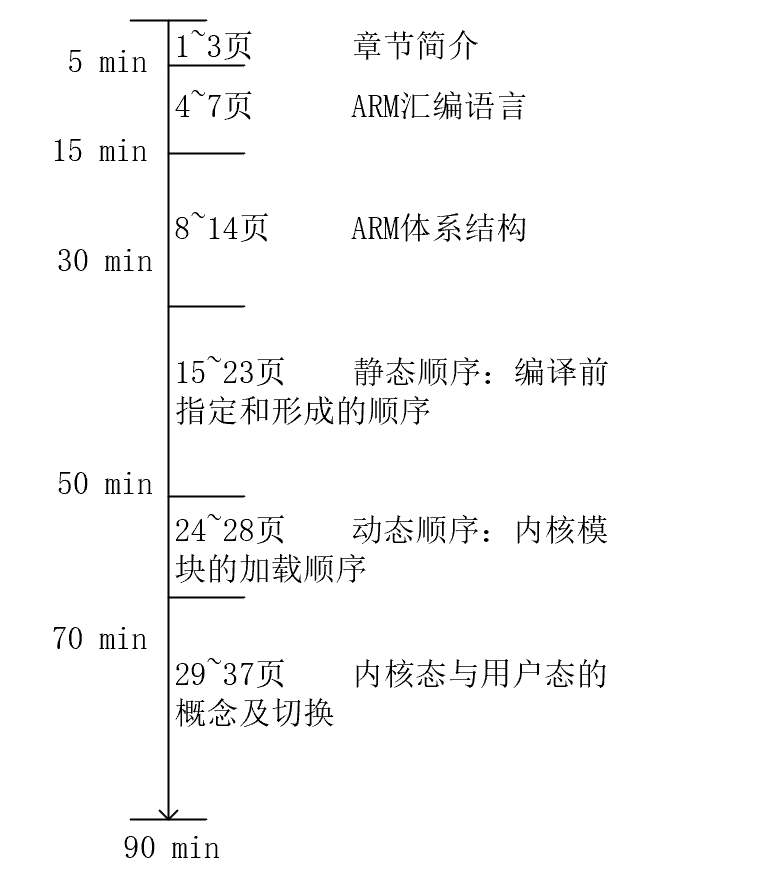
软件所制

第二章 第3讲 Kunpeng下的内核启动过程

**学时：**2学时

**教学目的：**系统学习kunpeng架构下的内核启动过程

**课程时间线：**

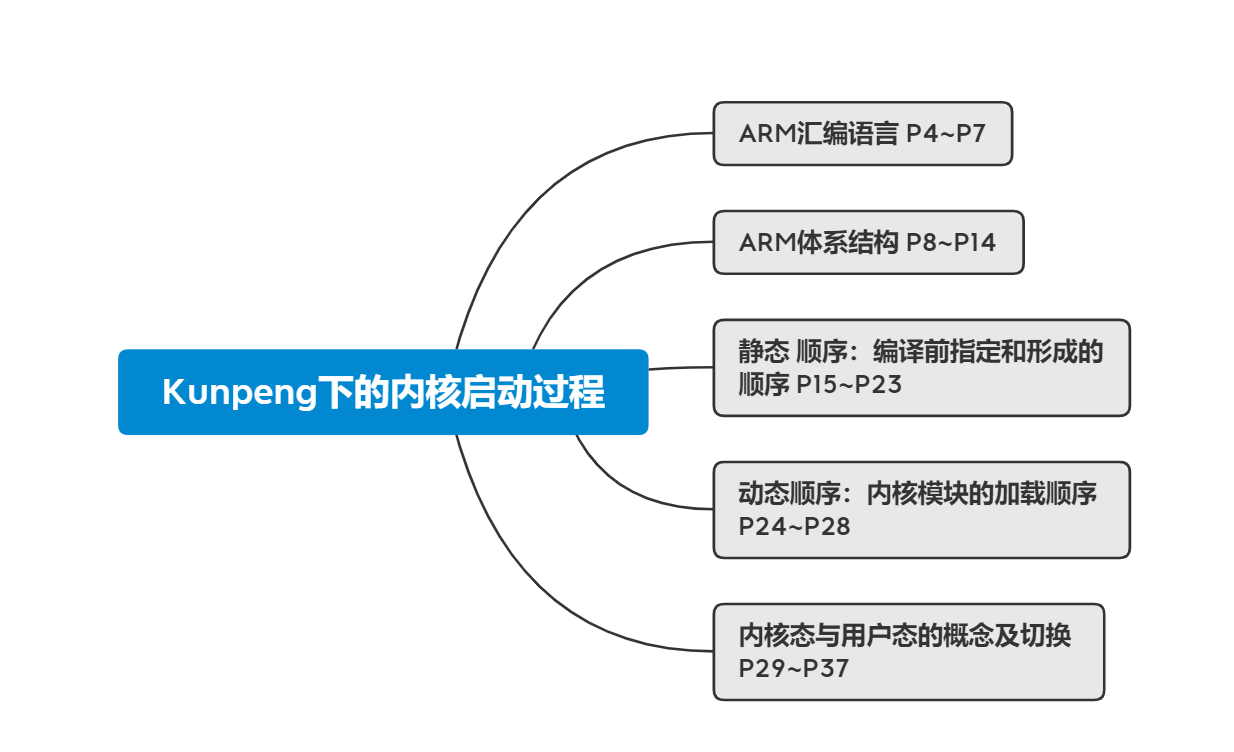


**课外参考读物：**

<http://blog.chinaunix.net/uid-69947851-id-5830505.html>

《Linux内核深度解析》

**知识框图：**



**PPT讲稿：**

1. 我们继续第二章内核启动的学习。
2. 这一节我们主要kunpeng下的内核启动过程。
3. 这一讲将从ARM汇编语言讲起。
4. 首先arm处理器与intel处理器有很多不同的地方，最主要的区别是指令集。Intel是复杂指令集（CISC）处理器，而ARM是精简指令集（RISC）处理器。CISC拥有功能更多更丰富的指令，允许对内存进行更复杂的操作。拥有更多指令操作、寻址模式，然而寄存器数量比ARM少。它主要应用在个人电脑、工作站、服务器中。而RISC拥有更简单的指令集和更多的通用寄存器。与Intel不同，ARM指令只操作寄存器，且只能使用Load/Store命令来读取和写入内存。Arm与x86相比也有很多不同，ARM中大部分指令都可以用作条件执行。x86和x86-64系列处理器使用小端（little-endian）地址格式。ARM架构在第三版以前是小端模式。之后变为大-小端（BI-endian)格式，允许大端或小端两种模式进行切换。
5. ARM汇编语法格式一般分为5个区域第1个域是4位[31-28]的条件码域；第2个域是[27-20]，除了指令编码外，还包含几个很重要的指令特征和可选后缀的编码；第3个域是地址基址，是4位[19-16],为R0-R15共16个寄存器编码；第4个域是目标或源寄存器Rd,是4位[15:12],为R0-R15共16个寄存器编码；第5个域是地址偏移或操作寄存器、操作数区[11-0]。
6. 具体的指令代码格式是<opcode>{<cond>}{S}<Rd>,<Rn>{,<OP2>}。其中，<opcode>是操作码，如ADD表示算数加法；{<cond>}表示指令执行的条件域，如EQ、NE等；{S}决定指令的执行结果是否影响CPSR的值，使用该后缀则指令执行的结果影响CPSR的值，否则不影响；<Rd>表示目的寄存器；<Rn>表示第一个操作数，为寄存器；<op2>表示第二个操作数，可以是立即数、寄存器或寄存器移位操作数；格式中<>的内容必不可少，{}中的内容可省略。
7. 上面的S值得注意一下，这是可选后缀，一般有两种可选后缀：S和！。指令的可选后缀："S"，当指令中使用"S"后缀时，指令执行后程序状态寄存器的条件标志位将被刷新，当不使用"S"后缀时，指令执行后程序状态寄存器的条件标志将不会发生变化。指令的可选后缀："!"，如果指令地址表达式中不含"!"后缀，则基址寄存器的地址值不会发生变化，如果指令中的地址表达式中含有"!"后缀时，指令执行后，基址寄存器中的地址值将发生变化，变化的结果如下:基址寄存器中的值(指令执行后) = 指令执行前的值 + 地址偏移量。
8. 接下来我们来看一下ARM的体系结构
9. ARM采用的RISC体系结构，采用RISC架构的ARM处理器一般具有如下特点：一是指令集——RISC减少了指令集种类，通常一个周期一条指令，采用固定长度的指令格式。可以通过几条指令完成一个复杂的操作。二是流水线——RISC采用单周期指令，且指令长度固定，便于流水线操作执行。三是寄存器——RISC的处理器拥有更多的通用寄存器，寄存器操作较多。最后是Load/Store结构——使用加载/存储指令批量从内存中读写数据，提高数据传输效率。
10. ARM处理器中共有37个寄存器，被分为若干个组，这些寄存器包括：31个通用寄存器和6个状态寄存器。这31个通用寄存器中包括程序计数器PC寄存器，这些寄存器都是32位的。其中又可以分为未分组寄存器 R0 - R7：对于未分组寄存器，它们没有被系统用于特别的用途，因此任何可采用通用寄存器的应用场合都可以使用未分组寄存器。但需要注意一点，未分组寄存器不会因为处理器模式的改变而更改指向的寄存器，因此在所有的处理器模式下未分组寄存器都指向同一个寄存器，当中断或异常处理造成处理器模式转换的时候，由于不同的处理器模式使用了相同的物理寄存器，这就有可能造成寄存器中的数据被破坏。分组寄存器R8~R14：分组寄存器R8～R14对应的物理寄存器取决于当前的处理器模式。若要访问特定的物理寄存器而不依赖当前的处理器模式，则要使用规定的名字。寄存器R8~R12各有两组物理寄存器：一组为FIQ模式，另一组为除了FIQ以外的所有模式。寄存器R8~R12没有任何指定的特殊用途，只是在作快速中断处理时使用。寄存器R13，R14各对应6个分组的物理寄存器，1个用于用户模式和系统模式，其它5个分别用于5种异常模式。寄存器R13通常用做堆栈指针，称为SP；寄存器R14用作子程序链接寄存器，也称为LR。而6个状态寄存器也是32位的，但是他们只用了其中的12位。在所有处理器模式下都可以访问当前的程序状态寄存器CPSR。CPSR包含条件码标志，中断禁止位，当前处理器模式以及其它状态和控制信息。每种异常模式都有一个程序状态保存寄存器SPSR。当异常出现时，SPSR用于保存CPSR的状态。
11. ARM处理器的工作模式有八种，用户模式（user）：正常程序执行模式；快速中断模式（FIQ）：高优先级的中断产生会进入该种模式，用于高速通道传输；外部中断模式（IRQ）：低优先级中断产生会进入该模式，用于普通的中断处理；特权模式（Supervisor）：复位和软中断指令会进入该模式；数据访问中止模式（Abort）：当存储异常时会进入该模式；未定义指令中止模式（Undefined）：执行未定义指令会进入该模式；系统模式（System）：用于运行特权级操作系统任务；监控模式（Monitor）：可以在安全模式和非安全模式之间切换。下图是在不同工作模式下能够使用的寄存器。
12. 与ARM7的三级流水相比，ARM9的五级流水效率更高。流水线的执行顺序：取指令->译码->执行->缓冲/数据->回写。取指令（Fetch）是从存储器读取指令；译码（Decode）是译码以鉴别它是属于哪一条指令；执行（Execute）是将操作数进行组合以得到结果或存储器地址；缓冲/数据（Buffer/data）是如果需要，则访问存储器以存储数据；回写：（Write-back）是将结果写回到寄存器组中。
13. ARM与x86在页面管理上最大的不同就是arm只支持分页管理而不支持分段。由于多任务系统的线性地址空间的总和要比真实的物理内存大很多，并且为了隔离多任务的地址空间，需要一种虚拟内存技术这种技术就是分页。分页的目的是将虚拟地址和物理地址划分成固定大小的页，然后在虚拟地址的页和物理地址的页之间动态建立映射关系。
14. 在32bit中的Linux内核中一般采用3层映射模型，第1层是页面目录（PGD)，第2层是页面中间目录（PMD）,第3层才是页面映射表（PTE）。但在ARM32系统中只用到两层映射，因此在实际代码中就要3层映射模型中合并一层。在ARM32架构中，可以按段（section）来映射，这时采用单层映射模式。使用页面映射需要两层映射结构，页面的选择可以是64KB的大页面或4KB的小页面。
15. 之后是静态顺序，这是内核编译前指定和形成的顺序。
16. vmlinux.lds.S主要是用来组织内核的每个函数存放在内核镜像文件的位置。编译内核源码生成内核文件的过程分两步，一个是“编译”，另一个是“链接”的过程，vmlinux.lds.S要做的就是告诉编译器如何链接编译好的各个内核.o文件。上图是一个删减了很多其他暂时不关心段的vmlinux.ld.s脚本文件，vmlinux.ld.s脚本文件的语法，这里不作出介绍， 关注的可以去查看相关文档。各个框中表明了各个框的作用。通过vmlinux.ld.s我们可以看到，bin文件的入口被设置为ENTRY(\_text),  因此，\_text即为入口函数地址，那么\_text又在那里呢？那么接着看这个脚本文件，我们发现在灰色框里面有一个\_text = .; 这说明\_text的地址就是.head.text的首地址。
17. 那么哪些数据被链接到.head.text段了呢？通过搜索发现目前只有kernel/head.S有代码被放置在了.head.text段，因此我们可以直接跳转到head.S中的\_\_HEAD处。
18. 之后我们找到位于/arch/arm64/head.S中的\_\_HEAD处，这里不管是if还是else最终都指向stext为止。其中，配置宏CONFIG\_EFI表示提供UEFI运行时支持，UEFI是统一的可扩展固件接口，用于取代BIOS。
19. 入口地址stext进入，可以看到执行了多个函数。preserve\_boot\_args是将uboot传入的参数 保存到bootargs[4] 全局变量里面。el2\_setup 是判断启动的模式是el2还是el1并进行相关级别的系统配置(armv8中el2是hypervisor模式，el1是标准的内核模式，具体的参考手册), 然后返回启动模式。set\_cpu\_boot\_mode\_flag是将启动模式保存到全局变量。\_\_create\_page\_tables是创建内存映射表，一共两张，一张存放在swapper\_pg\_dir（线性映射），一张存放在idmap\_pg\_dir（一对一映射）。\_\_cpu\_setup 是初始化处理器相关的代码，配置访问权限，内存地址划分等。最后一个\_\_primary\_switch是开启MMU, 准备0号进程和内核栈，然后跳转到start\_kernel运行。
20. 具体来看，el2\_setup作用是判断启动的模式是el2还是el1并进行相关级别的系统配置。其主要工作是如果异常级别是1，那么在异常级别1执行内核；如果异常级别是2，那么根据处理器是否支持虚拟化宿主扩展，决定是否需要降低到异常级别1，而对于支不支持虚拟机又可以考虑如果处理器支持虚拟化宿主扩展，那么在异常级别2执行内核；如果处理器不支持虚拟化宿主扩展，那么降低到异常级别1，在异常级别1执行内核。通常arm64处理器在异常级别0执行进程，在异常级别1执行内核。

而虚拟机是现在流行的虚拟化技术，在计算机上创建一个虚拟机，在虚拟机里面运行一个操作系统，运行虚拟机的操作系统称为宿主操作系统，在虚拟机里面的操作系统称为客户操作系统。现在常用的虚拟机时基于内核的虚拟机KVM，KVM的主要特点时直接在处理器上执行客户操作系统，因此虚拟机的执行速度非常快。KVM时内核的一个模块，把内核编程虚拟机监控程序，宿主操作系统中的进程在异常级别0运行 ，内核在异常级别1运行，KVM模块可以穿越级别1和2；客户操作系统中的进程在异常级别0运行，内核在异常级别1运行。

1. \_\_create\_page\_tables其主要工作是首先创建恒等映射，恒等映射的特点时虚拟地址和物理地址相同，是为了在开启处理器的内存管理单元的一瞬间能够平稳过度。函数\_\_enable\_mmu负责开启内存管理单元，内核把函数\_\_enable\_mmu附近的代码放在恒等映射代码节（.idmap.text）中，恒等映射代码节的起始地址存放在全局变量\_\_idmap\_text\_start中，结束地址存放在全局变量\_\_idmap\_text\_end中。其次是为内核镜像创建映射，在内核的页表中为内核镜像创建映射，内核镜像的起始地址是\_text，结束地址是\_end，swapper\_pg\_dir是内核的页全局目录的起始地址。
2. \_\_primary\_switch其主要工作是首先调用函数\_\_enable\_mmu以开启内存管理单元然后调用函数\_\_primary\_switched。\_\_enable\_mmu其主要工作是首先把转换表基准寄存器0（TTBR0\_EL1）设置为恒等映射的页全局目录的起始物理地址。然后把转换表基准寄存器1（TTBR1\_EL1）设置为内核的页全局目录的起始物理地址。再设置系统控制寄存器（SCTLR\_EL1），开启内存管理单元，以后执行程序时内存管理单元将会把虚拟地址转换成物理地址。
3. \_\_primary\_switched其主要工作是首先把当前异常级别的栈指针寄存器设置为0号线程内核栈顶部。然后把异常级别0的栈指针寄存器（SP\_EL0）设置为0号线程的结构体thread\_info的地址（init\_task.thread\_info）。然后把向量基准地址寄存器（VBAR\_EL1）设置为异常向量表的起始地址。再计算内核镜像的起始虚拟地址和物理地址的差值，保存在全局变量中。之后用0初始化内核的未初始化数据段。最后调用C语言函数start\_kernel进入C程序初始化部分。
4. 动态顺序是内核模块的加载顺序。
5. 内核模块是Linux内核向外部提供的一个插口，其全称为动态可加载内核模块(Loadable Kernel Module)，简称模块，这种模块通常由一组函数和数据结构组成，用来实现一种文件系统、一个驱动程序或其他内核上层的功能，模块可以被编译到内核中，或者作为独立的模块被动态加载。最终的Linux内核=vmlinux + kernel modules。
6. 将模块编译到内核中，是需要修改模块的Makefile文件的，Linux中Makefile能够指定是否编译该模块，以及是否编译到内核，或者作为独立模块。一般的obj-m表示编译生成可加载模块，而obj-y表示直接将模块编译到内核中。如果编译到内核中，那么模块的入口函数会被注册到内核的.initcall.init段中，也就是我们下面看到8中模块等级。
7. 将模块编译到内核中一共要经过一下的函数调用start\_kernel()--->rest\_init()--->kernel\_init()--->do\_basic\_setup()--->do\_initcalls()，再最后的do\_initcalls中通过for循环遍历所有注册的模块。
8. do\_initcalls()将在\_\_initcall\_start和\_\_initcall\_end之间定义的各个模块依次加载，其中.initcall.init段包含了这之间的内容。从中能够了解到这两个宏之间依次排列了14个等级的宏，由于这些宏是按先后顺序链接的，所以也就表示，这14个宏有优先级：0>1>1s>2>2s………>7>7s。在静态编译时module\_init就相当于device\_initcall。在的内核 中，gianfar\_device使用的是arch\_initcall，而gianfar\_driver使用的是module\_init，因为arch\_initcall的优先级大于module\_init，所以gianfar设备驱动的device先于driver在总线上添加。
9. 最后一节是内核态和用户态的概念和切换
10. 首先是概念，内核态是cpu可以访问内存的所有数据，包括外围设备，例如硬盘，网卡，cpu也可以将自己从一个程序切换到另一个程序。用户态是只能受限的访问内存，且不允许访问外围设备，占用cpu的能力被剥夺，cpu资源可以被其他程序获取。从这张linux系统架构图中可以看到，内核态和用户态之间夹着库函数，shell和系统调用，也就是说最终只有通过系统调用，用户才能从应用程序到内核中。
11. 系统调用是操作系统的最小功能单位。根据不同的应用场景，不同的Linux发行版本提供的系统调用数量也不尽相同，大致在240-350之间。这些系统调用组成了用户态跟内核态交互的基本接口。库函数就是屏蔽这些复杂的底层实现细节，减轻程序员的负担，从而更加关注上层的逻辑实现。它对系统调用进行封装，提供简单的基本接口给用户，这样增强了程序的灵活性，当然对于简单的接口，也可以直接使用系统调用访问资源，例如：open()，write()，read()等等。我们经常使用的各种函数如果最后要到内核态大都是库函数。使用起来非常方便。Shell顾名思义，就是外壳的意思。就好像把内核包裹起来的外壳。它是一种特殊的应用程序，俗称命令行。为了方便用户和系统交互，一般一个shell对应一个终端，呈现给用户交互窗口。当然shell也是编程的，它有标准的shell语法，符合其语法的文本叫shell脚本。很多人都会用shell脚本实现一些常用的功能，可以提高工作效率。
12. 内核态和用户态之间的切换。如果要从内核态到用户态只需要设置程序状态字PSW。而要从用户态到内核态唯一途径是通过中断、异常、陷入机制（访管指令）。一般的有两种方式：用户空间的应用程序，通过系统调用，进入内核空间。这个时候用户空间的进程要传递很多变量、参数的值给内核，内核态运行的时候也要保存用户进程的一些寄存器值、变量等。所谓的“进程上下文”，可以看作是用户进程传递给内核的这些参数以及内核要保存的那一整套的变量和寄存器值和当时的环境等。还有一种是硬件通过触发信号，导致内核调用中断处理程序，进入内核空间。这个过程中，硬件的一些变量和参数也要传递给内核，内核通过这些参数进行中断处理。所谓的“中断上下文”，其实也可以看作就是硬件传递过来的这些参数和内核需要保存的一些其他环境（主要是当前被打断执行的进程环境）。
13. 在内核初始化过程中也有一处需要从内核态切换到用户态，这个位置位于内核进程的创建。在start\_kernel()函数的最后，调用了rest\_init()函数，而在rest\_init()函数中，建立了两个内核进程，两个内核进程分别是kernel\_init和kthreadd。kernel\_init进程之后会转换为用户态的init进程，进程号为1；kthreadd进程的进程号为2，负责创建内核进程。在linux的终端中，通过ps命令可以看到这两个进程。还可以看到，其它的所有进程都是以这两个进程为父进程的。
14. kernel\_init最开始只是一个函数，这个函数作为进程被启动，但是之后它将读取根文件系统下的init程序，这个操作将完成从内核态到用户态的转变，而这个init进程是所有用户态进程的父进程，它生了大量的子进程，所以init进程将永远存在，其PID是1。随后，1号进程调用do\_execve运行可执行程序init，并演变成用户态1号进程，即init进程。init进程是linux内核启动的第一个用户级进程。init有许多很重要的任务，比如像启动getty（用于用户登录）、实现运行级别、以及处理孤立进程。它按照配置文件/etc/initab的要求，完成系统启动工作，创建编号为1号、2号…的若干终端注册进程getty。每个getty进程设置其进程组标识号，并监视配置到系统终端的接口线路。当检测到来自终端的连接信号时，getty进程将通过函数do\_execve（）执行注册程序login，此时用户就可输入注册名和密码进入登录过程，如果成功，由login程序再通过函数execv（）执行shell，该shell进程接收getty进程的pid，取代原来的getty进程。再由shell直接或间接地产生其他进程。上述过程可描述为：0号进程->1号内核进程->1号用户进程（init进程）->getty进程->shell进程。
15. kernel\_thread()函数的函数调用是pid\_t kernel\_thread(int (\*fn)(void \*), void \*arg, unsigned long flags)，它的第一个参数为一个函数的函数名第二个参数是arg为传递给函数的参数第三个参数是flags是一些创建进程时用到的标志，kernel\_thread函数的返回值是pid\_t的进程号，由此我们也可以看出这个函数是用来创建进程的，kernel\_thread的执行是通过调用do\_fork()函数实现的，do\_fork()函数是所有fork类的函数的最终调用的函数。
16. init进程创建完成后将继续进行初始化过程。其中smp\_prepare\_cpus()函数是在启动从处理器以前执行准备工作。do\_pre\_smp\_initcalls()函数是执行必须在SMP系统以前执行的早期初始化，即使使用宏early\_initcall注册的初始化函数。smp\_init()函数是初始化SMP系统，启动所有从处理器。do\_initcalls()函数是执行级别0~7的初始化。之后打开控制台的字符设备文件“/dev/console”，文件描述符0、1和2分别是标准输入、标准输出和标准错误，都是控制台的字符设备文件。prepare\_namespace()函数是挂载根文件系统，后面装载init程序时需要从存储设备上的文件系统中读文件。free\_initmem()函数是释放初始化代码和数据占用的内存。最后装载init程序（u-boot程序可以传递内核参数“init=”以指定init程序），从内核栈进程切换成用户空间的init进程。