

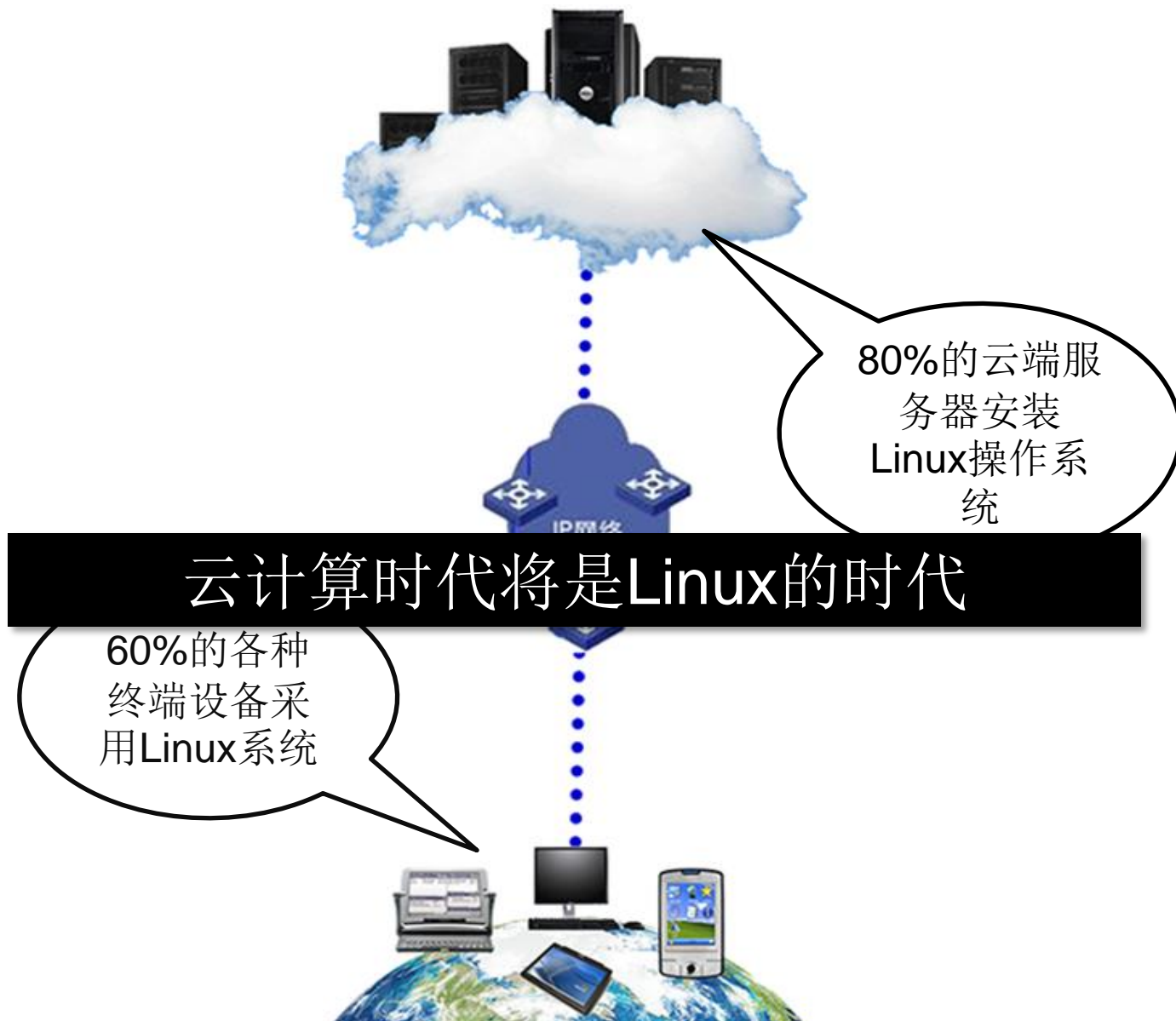
第5章 ARM-Linux内核



目 录

- 5.1 ARM-Linux概述
- 5.2 ARM-Linux进程管理
- 5.3 ARM-Linux内存管理
- 5.4 ARM-Linux模块
- 5.5 ARM-Linux中断管理
- 5.6 ARM-Linux系统调用

Linux系统发展



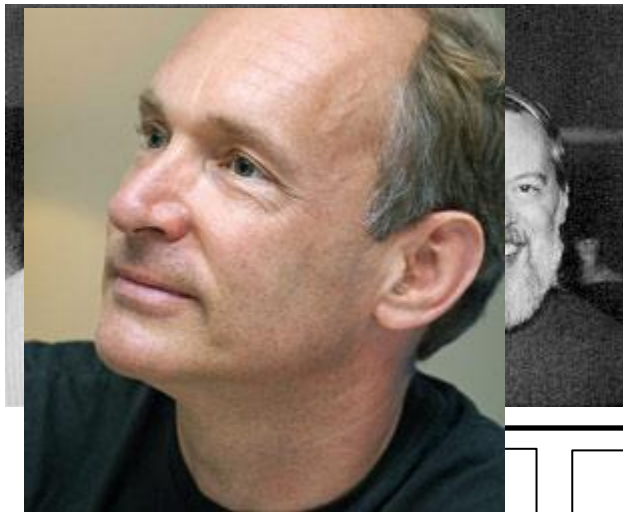
Linux操作系统诞生于1991年，可安装在各种计算机硬件设备中，比如各种智能移动终端、路由器、台式计算机、大型机和超级计算机等。Linux支持包括x86、ARM、MIPS和POWER-PC等在内的多种硬件体系结构。

Linux存在着许多不同的版本，但它们都使用了Linux内核。

Linux是一个一体化内核（monolithic kernel）系统。

这里的“内核”指的是一个提供硬件抽象层、磁盘及文件系统控制、多任务等功能的系统软件，一个内核不是一套完整的操作系统。一套建立在Linux内核的完整操作系统叫作Linux操作系统，或是GNU/Linux。

Linux系统概况——诞生



UNIX

时间：
地点：
人物：
事件：

Linux诞生



时间：1969年
地点：贝尔实验室
人物：k. Thompson,
Dennis Richie
事件：诞生unix家族



时间：1984年
地点：美国
人物：理查德·斯托曼
事件：开启了自由软件时代

时间：1988年
地点：美国IEEE
人物：理查德·斯托曼
事件：诞生了POSIX标准

时间：1990年
地点：欧洲核研究所
人物：蒂姆·伯纳斯·李
事件：宣告互联网的来临

时

与Linux相关的协议和标准

- **FSF (Free Software Foundation)**：自由软件基金会
- **自由软件**：自由使用、复制、研究、修改和传播
- **GNU(GNU is Not Unix)计划**：目标是创建一套完全自由的网络操作系统和应用软件
- **GPL协议**：**GNU General Public License**，即**GNU通用公共许可证**
- **LGPL协议**：**Libraray General Public License**即**程序库公共许可证**
- **POSIX (Portable Operating System Interface)** 标准：**可移植性操作系统接口**

Linux的主要版本

- Linux版本介绍
 - RedHat系列(红帽公司的产品，来自美国),产品分支主要有：
 - Red Hat Linux
 - Red Hat Enterprise Linux
 - Fedora Core
 - CentOS
 - Debian及衍生版
 - Debian
 - Ubuntu: 20.04 LTS
 - Mint Linux
 - 其他版本
 - SUSE Linux Enterprise
 - RedFlag

Linux精髓

- 代表一种开源文化
 - 免费软件，开放源代码
 - 自由软件，可在原有程序基础上开发自己的程序
 - GNU/Linux
 - Linux仅指Linux内核
 - Linux系统的大部分应用都建立在GNU软件之上
- 核心结构
 - Linux内核
 - Linux Shell
 - Linux文件系统
 - X-Window
 - GNU Tools

*Linux*操作系统的灵魂是*Linux*内核，

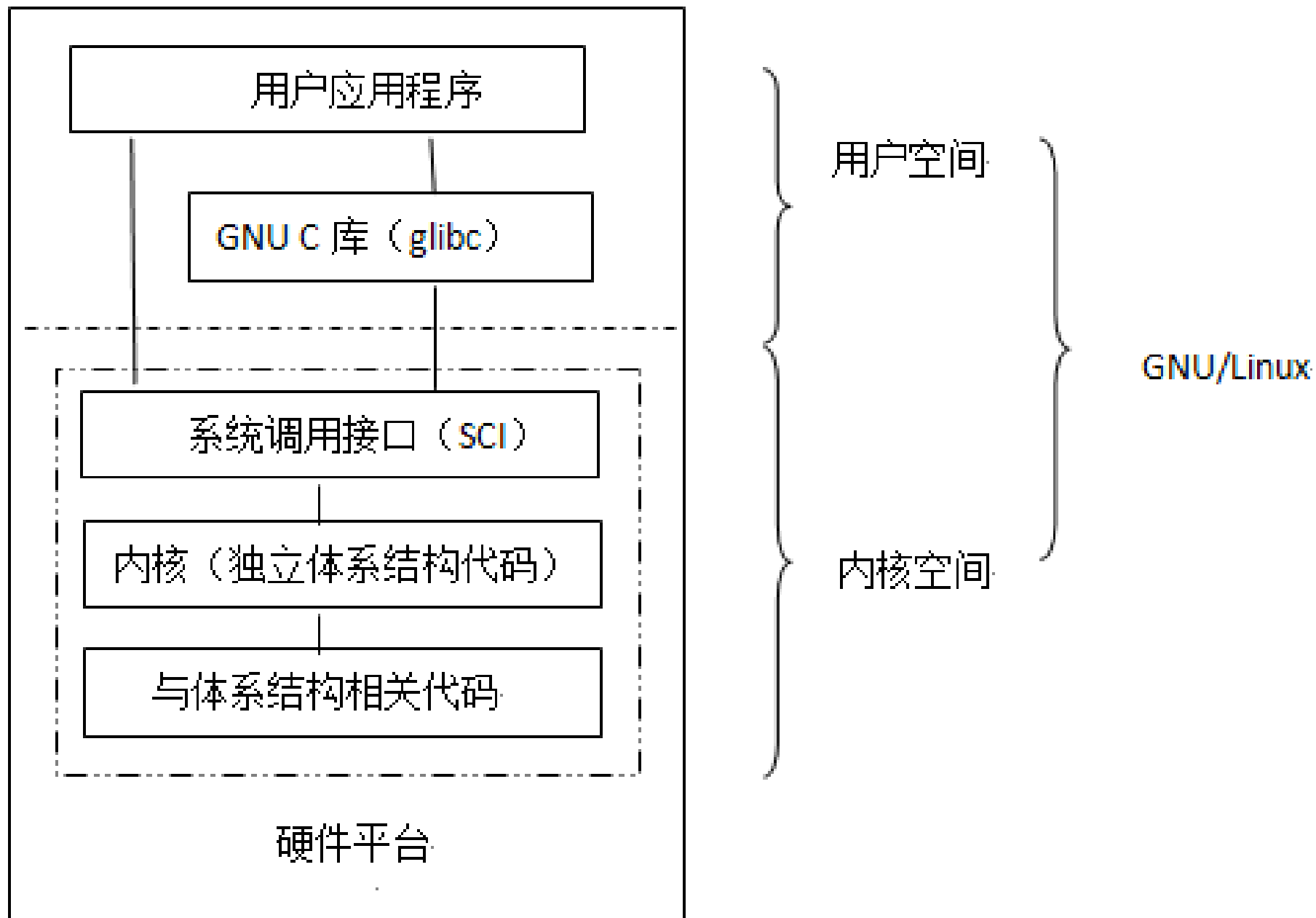
内核为系统其他部分提供系统服务。
ARM-Linux内核是专门适应ARM体系
结构设计的Linux内核，它负责整个系
统的：

进程管理和调度、
内存管理、
文件管理、
设备管理和网络管理等主要系统功能。

5.1 Part One

ARM-Linux概述

5.1.1 GNU/Linux 操作系统的基本体系结构



用户空间包括用户应用程序和GNU C库（glibc库），负责执行用户应用程序。

内核空间可以进一步划分成 3 层：

系统调用接口（System Call Interface），它是用户空间与内核空间的桥梁

独立于体系结构的内核代码

依赖于体系结构的代码，构成了通常称为板级支持包BSP（Board Support Package）的部分

5.1.2 ARM-Linux内核版本及特点

ARM-Linux内核版本变化与Linux内核版本变化保持同步。

嵌入式linux系统内核（如ARM_Linux内核）往往在标准linux基础上通过安装patch实现，如ARM_Linux内核就是对linux安装rmk补丁形成的，只有安装了这些补丁，内核才能顺利地移植到ARM_Linux上。

当然也可以通过已经安装好补丁的内核源码包实现。

在2.6版本之前，Linux内核版本的命名格式为“A.B.C”。

数字 A 是内核版本号，

数字 B 是内核主版本号，主版本号根据传统的奇-偶系统版本编号来分配：奇数为开发版，偶数为稳定版；


数字 C 是内核次版本号，次版本号是无论在内核增加安全补丁、修复bug、实现新的特性或者驱动时都会改变。

2011年5月29号，设计者Linus Torvalds宣布为了纪念Linux发布 20周年，在 2.6.39 版本发布之后，内核版本将升到3.0。Linux继续使用在2.6.0版本引入的基于时间的发布规律，但是使用第二个数——例如在3.0发布的几个月之后发布3.1，同时当需要修复bug和安全漏洞的时候，增加一个数字（现在是第三个数）来表示，如 3.0.18。

Mainline是主线版本， Stable 是稳定版， Longterm是长期支持版

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

Latest Stable Kernel:

 4.19.1

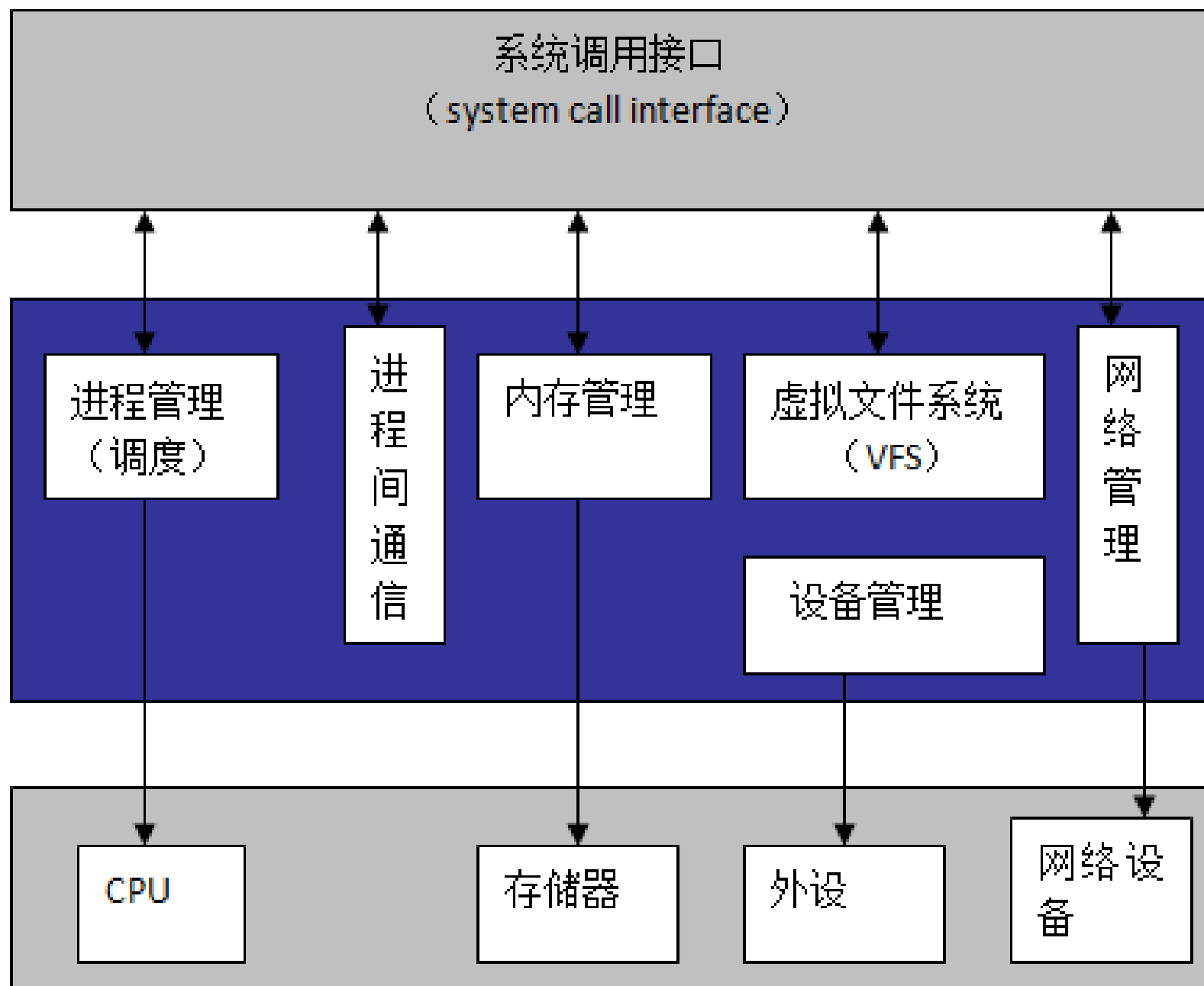
mainline:	4.20-rc1	2018-11-04	[tarball]	[patch]	[view diff]	[browse]			
stable:	4.19.1	2018-11-04	[tarball]	[pgp]	[patch]	[view diff]	[browse]	[changelog]	
stable:	4.18.17	2018-11-04	[tarball]	[pgp]	[patch]	[inc. patch]	[view diff]	[browse]	[changelog]
longterm:	4.14.79	2018-11-04	[tarball]	[pgp]	[patch]	[inc. patch]	[view diff]	[browse]	[changelog]
longterm:	4.9.135	2018-10-20	[tarball]	[pgp]	[patch]	[inc. patch]	[view diff]	[browse]	[changelog]
longterm:	4.4.162	2018-10-20	[tarball]	[pgp]	[patch]	[inc. patch]	[view diff]	[browse]	[changelog]
longterm:	3.18.124 [EOL]	2018-10-13	[tarball]	[pgp]	[patch]	[inc. patch]	[view diff]	[browse]	[changelog]
longterm:	3.16.60	2018-10-21	[tarball]	[pgp]	[patch]	[inc. patch]	[view diff]	[browse]	[changelog]
linux-next:	next-20181107	2018-11-07						[browse]	

Linux是一个内核运行在单独的内核地址空间的**单内核**，但是**汲取了微内核的精华**如模块化设计、抢占式内核、支持内核线程以及动态装载内核模块等特点。

以2.6版本为例，其主要特点有：

- (1) 支持动态加载内核模块机制。
- (2) 支持对称多处理机制（SMP）。
- (3) O(1)的调度算法。
- (4) linux内核可抢占，linux内核具有允许在内核运行的任务优先执行的能力。
- (5) linux不区分线程和其他一般的进程，对内核来说，所有的进程都一样(仅部分共享资源)。
- (6) linux提供具有设备类的面向对象的设备模块、热插拔事件，以及用户空间的设备文件系统。

5.1.3 ARM-Linux内核的主要架构及功能



根据内核的核心功能，Linux内核具有5个主要的子系统，分别负责如下的功能：进程管理、内存管理、虚拟文件系统、进程间通信和网络接口。

1.进程管理

进程管理负责管理CPU资源，以便让各个进程能够以尽量公平的方式访问CPU。进程管理负责进程的创建和销毁，并处理它们和外部世界之间的连接(输入输出)。除此之外，控制进程如何共享的调度器也是进程管理的一部分。概括来说，内核进程管理活动就是在单个或多个CPU上实现了多个进程的抽象。进程管理源码可参考./linux/kernel目录。

2.内存管理

Linux内核所管理的另外一个重要资源是内存。内存管理策略是决定系统性能好坏的一个关键因素。内核在有限的可用资源之上为每个进程都创建了一个虚拟空间。内存管理的源代码可以在 `./linux/mm` 中找到。

3.虚拟文件系统

文件系统在Linux 内核中具有十分重要的地位，用于对外设的驱动和存储，隐藏了各种硬件的具体细节。Linux引入了虚拟文件系统

（Virtual File System）为用户提供了统一、抽象的文件系统界面，以支持越来越繁杂的具体的文件系统。Linux内核将不同功能的外部设备，例如Disk设备、输入输出设备、显示设备等，抽象为可以通过统一的文件操作接口来访问。Linux中的绝大部分对象都可以视为文件并进行相关操作。

4.进程间通信

不同进程之间的通信是操作系统的基本功能之一。

Linux内核通过支持**POSIX**规范中标准的**IPC**

（**Inter Process Communication**，相互通信）机制和其他许多广泛使用的**IPC**机制实现进程间通信。**IPC**不管理任何的硬件，它主要负责**Linux**系统中进程之间的通信。比如**UNIX**中最常见的管道、信号量、消息队列和共享内存等。另外，信号

（**signal**）也常被用来作为进程间的通信手段。**Linux**内核支持**POSIX**规范的信号及信号处理并广泛应用。

5.网络管理

网络管理提供了各种网络标准的存取和各种网络硬件的支持，负责管理系统的网络设备，并实现多种多样的网络标准。网络接口可以分为网络设备驱动程序和网络协议。

这五个系统相互依赖，缺一不可，但是相对而言进程管理处于比较重要的地位，其他子系统的挂起和恢复进程的运行都必须依靠进程调度子系统的参与。

调度程序的初始化及执行过程中需要内存管理模块分配其内存地址空间并进行处理；

进程间通信需要内存管理实现进程间的内存共享；而内存管理利用虚拟文件系统支持数据交换，交换进程（**swapped**）定期由调度程序调度；

虚拟文件系统需要使用网络接口实现网络文件系统，而且使用内存管理子系统实现内存设备管理，

同时虚拟文件系统实现了内存管理中内存的交换。

5.1.4 linux内核源码目录结构

Arch目录包括了所有和体系结构相关的核心代码。它下面的每一个子目录都代表一种Linux支持的体系结构

Include目录包括编译核心所需要的大部分头文件，

Init目录包含核心的初始化代码，需要注意的是该代码不是系统的引导代码。

Mm目录包含了所有的内存管理代码。与

Drivers目录中是系统中所有的设备驱动程序。

lpc目录包含了核心进程间的通信代码。

Modules目录存放了已建好的、可动态加载的模块。

Fs目录存放Linux支持的文件系统代码。不同的文件系统有不同的子目录对应，如jffs2文件系统对应的就是jffs2子目录。

Kernel内核管理的核心代码放在这里。另外与处理器结构相关代码都放在**arch/*/kernel**目录下。

Net目录里是核心的网络部分代码。

Lib目录包含了核心的库代码，但是与处理器结构相关的库代码被放在**arch/*/lib/**目录下。

Scripts目录包含用于配置核心的脚本文件。

Documentation目录下是一些文档，是对目录作用的具体说明。

5.2

Part Two

ARM-Linux进程管理

进程是处于执行期的程序以及它所管理的资源的总称，这些资源包括如打开的文件、挂起的信号、进程状态、地址空间等。程序并不是进程，实际上两个或多个进程不仅有可能执行同一程序，而且还有可能共享地址空间等资源。

进程管理是Linux内核中最重要的子系统，它主要提供对CPU的访问控制。由于计算机中，CPU资源是有限的，而众多的应用程序都要使用CPU资源，所以需要“进程调度子系统”对CPU进行调度管理。

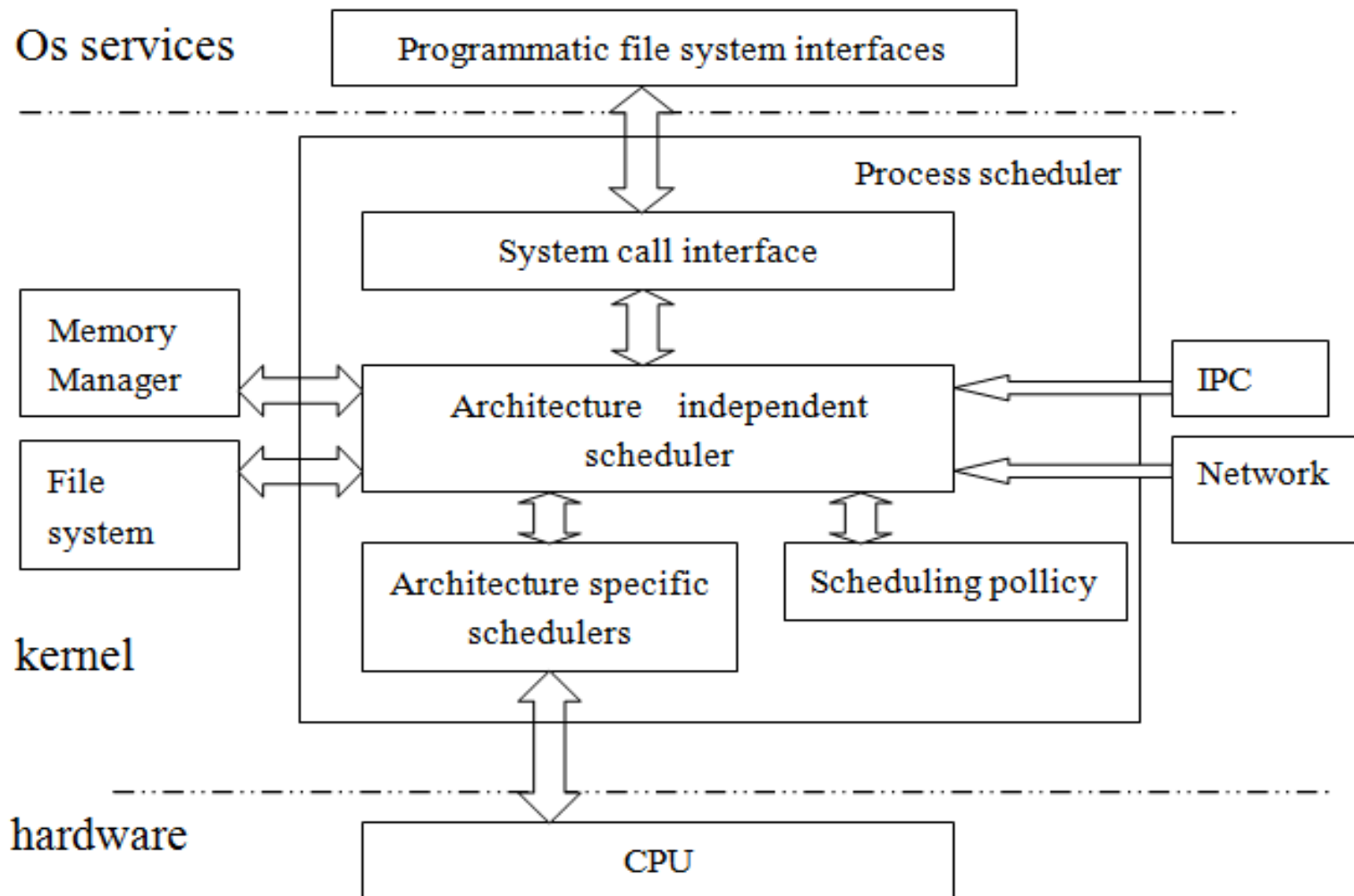


图5-4 Linux进程管理调度子系统基本架构

Scheduling Policy模块。该模块实现进程调度的策略，它决定哪个（或者哪几个）进程将拥有CPU资源。

Architecture-specific Schedulers模块。该模块涉及体系结构相关的部分，用于将对不同CPU的控制抽象为统一的接口。这些控制功能主要在suspend和resume进程时使用，包含CPU的寄存器访问、汇编指令操作等。

Architecture-independent Scheduler模块。该模块涉及体系结构无关的部分，会和“Scheduling Policy模块”沟通，决定接下来要执行哪个进程，然后通过“Architecture-specific Schedulers模块”指定的进程予以实现。

System Call Interface，系统调用接口。进程调度子系统通过系统调用接口将需要提供给用户空间的接口开放出去，同时屏蔽掉不需要用户空间程序关心的细节。

5.2.1 进程的表示和切换

Linux内核通过一个被称为进程描述符的`task_struct`结构体（也叫进程控制块）来管理进程，这个结构体记录了进程的最基本的信息，

进程描述符中不仅包含了许多描述进程属性的字段，而且还包含一系列指向其他数据结构的指针。内核把每个进程的描述符放在一个叫做任务队列的双向循环链表当中，它定义在`./include/linux/sched.h`文件中。

系统中的每个进程都必然处于以下所列进程状态中的一种。

TASK_RUNNING表示进程要么正在执行，要么正要准备执行。

TASK_INTERRUPTIBLE表示进程被阻塞（睡眠），直到某个条件变为真。条件一旦达成，进程的状态就被设置为**TASK_RUNNING**。

TASK_UNINTERRUPTIBLE的意义与**TASK_INTERRUPTIBLE**基本类似，除了不能通过接受一个信号来唤醒以外。

__TASK_STOPPED表示进程被停止执行，当进程接收到**SIGSTOP**、**SIGTTIN**、**SIGTSTP**或者**SIGTTOU**信号之后就会进入该状态。

__TASK_TRACED表示进程被**debugger**等进程监视。

TASK_DEAD：一个进程在退出时，**state**字段被置于该状态，表明进程处于退出过程中，可能成为**EXIT_ZOMBIE** 或**EXIT_DEAD**态。

EXIT_ZOMBIE：表示进程的执行被终止，但是其父进程还没有使用**wait()**等系统调用来获知它的终止信息。

EXIT_DEAD：进程在系统中被删除时将进入该状态。该状态表示进程的最终状态。

TASK_WAKEKILL：该状态是当进程收到致命错误信号时唤醒进程。类似于前面的**TASK_UNINTERRUPTIBLE**，但可接收**fatal signals**。

TASK_WAKING：该状态说明该任务正在唤醒。

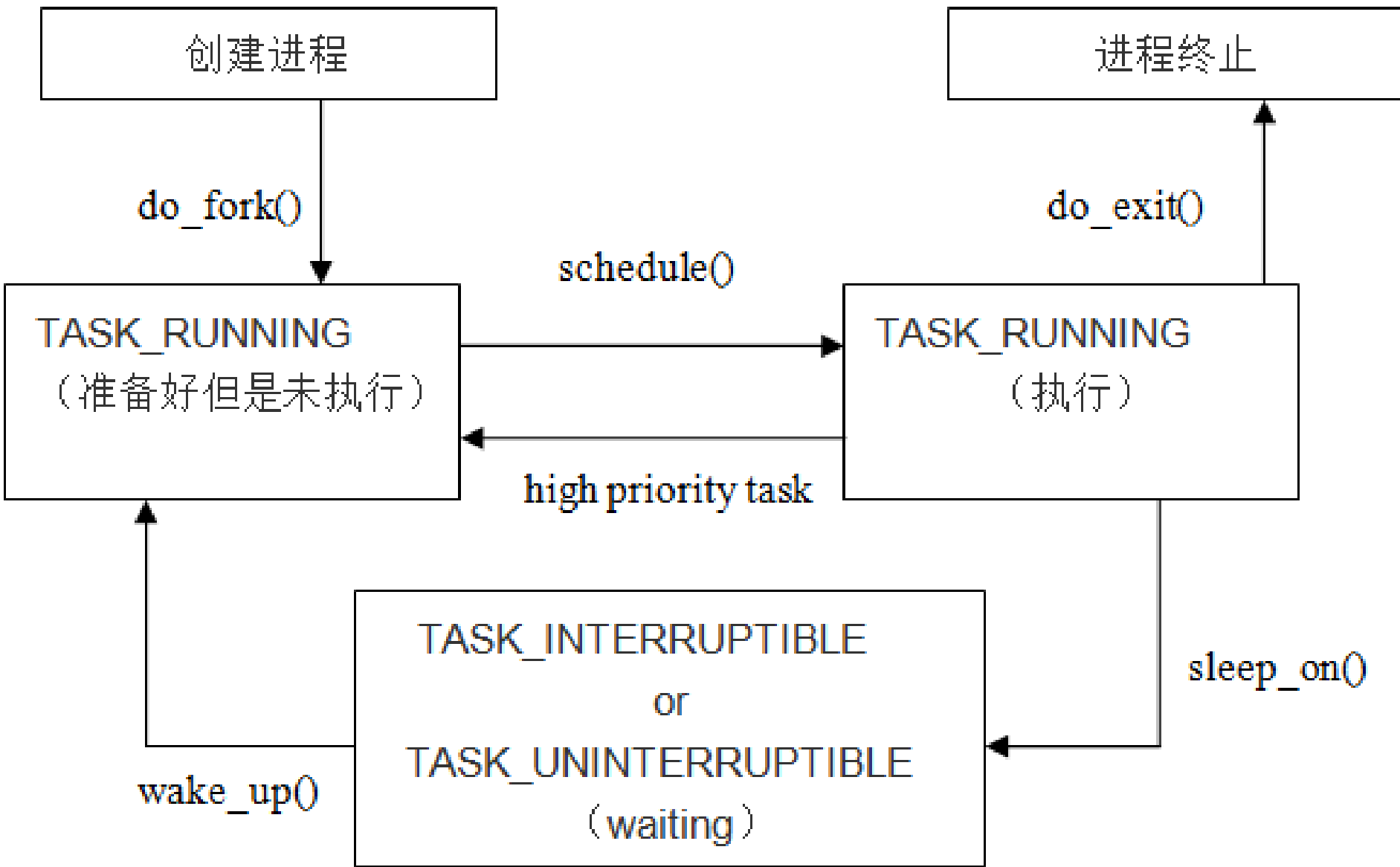


图5-5 进程状态的切换

Linux命令 ps

Linux中的ps命令是Process Status的缩写。ps命令用来列出系统中当前运行的那些进程及其信息。

ps aux命令显示状态信息：

- R 正在运行，或在队列中的进程
- D 不可中断
- S 处于休眠状态
- T 停止或被追踪
- Z 僵尸进程
- X 死掉的进程

- < 高优先级

- N 低优先级

- L 有些页被锁进内存

- s 包含子进程

- + 位于后台的进程组；

- l 多线程，克隆线程 multi-threaded (using CLONE_THREAD, like NPTL pthreads do)

Linux进程状态(ps stat)之R、S、D、T、Z、X

Linux进程状态：R (TASK_RUNNING), 可执行状态。

只有在该状态的进程才可能在CPU上运行。而同一时刻可能有多个进程处于可执行状态，这些进程的task_struct结构（进程控制块）被放入对应CPU的可执行队列中（一个进程最多只能出现在一个CPU的可执行队列中）。进程调度器的任务就是从各个CPU的可执行队列中分别选择一个进程在该CPU上运行。

很多操作系统教科书将正在CPU上执行的进程定义为RUNNING状态、而将可执行但是尚未被调度执行的进程定义为READY状态，这两种状态在linux下统一为 TASK_RUNNING 状态。

Linux进程状态：S (TASK_INTERRUPTIBLE)，可中断的睡眠状态。

处于这个状态的进程因为等待某某事件的发生（比如等待socket连接、等待信号量），而被挂起。这些进程的task_struct结构被放入对应事件的等待队列中。当这些事件发生时（由外部中断触发、或由其他进程触发），对应的等待队列中的一个或多个进程将被唤醒。

通过ps命令我们会看到，一般情况下，进程列表中的绝大多数进程都处于TASK_INTERRUPTIBLE状态（除非机器的负载很高）。毕竟CPU就这么一两个，进程动辄几十上百个，如果不是绝大多数进程都在睡眠，CPU又怎么响应得过来。

Linux进程状态：D (TASK_UNINTERRUPTIBLE)，不可中断的睡眠状态。

与TASK_INTERRUPTIBLE状态类似，进程处于睡眠状态，但是此刻进程是不可中断的。不可中断，指的并不是CPU不响应外部硬件的中断，而是指进程不响应异步信号。绝大多数情况下，进程处在睡眠状态时，总是应该能够响应异步信号的。否则你将惊奇的发现，kill -9竟然杀不死一个正在睡眠的进程了！于是我们也很好理解，为什么ps命令看到的进程几乎不会出现TASK_INTERRUPTIBLE状态，而总是TASK_UNINTERRUPTIBLE状态。

而TASK_UNINTERRUPTIBLE状态存在的意义就在于，内核的某些处理流程是不能被打断的。在进程对某些硬件进行操作时（比如进程调用read系统调用对某个设备文件进行读操作，而read系统调用最终执行到对应设备驱动的代码，并与对应的物理设备进行交互），可能需要使用TASK_UNINTERRUPTIBLE状态对进程进行保护，以避免进程与设备交互的过程被打断，造成设备陷入不可控的状态。这种情况下的TASK_UNINTERRUPTIBLE状态总是非常短暂的，通过ps命令基本上不可能捕捉到。

linux系统中也存在容易捕捉的TASK_UNINTERRUPTIBLE状态。执行vfork系统调用后，父进程将进入TASK_UNINTERRUPTIBLE状态，直到子进程调用exit或exec。通过下面的代码就能得到处于TASK_UNINTERRUPTIBLE状态的进程：

```
#include
```

```
void main() {
```

```
if (!vfork()) sleep(100);
```

```
}
```

编译运行，然后ps一下：

```
$ ps -ax | grep a.out
```

```
4371 pts/0  D+    0:00 ./a.out
```

```
4372 pts/0  S+    0:00 ./a.out
```

```
4374 pts/1  S+    0:00 grep a.out
```

然后我们可以试验一下TASK_UNINTERRUPTIBLE状态的威力。不管kill还是kill -9，这个TASK_UNINTERRUPTIBLE状态的父进程依然屹立不倒。

Linux进程状态：T (TASK_STOPPED or TASK_TRACED)，暂停状态或跟踪状态。

向进程发送一个SIGSTOP信号，它就会因响应该信号而进入TASK_STOPPED状态（除非该进程本身处于TASK_UNINTERRUPTIBLE状态而不响应信号）。

向进程发送一个SIGCONT信号，可以让其从TASK_STOPPED状态恢复到TASK_RUNNING状态。

当进程正在被跟踪时，它处于TASK_TRACED这个特殊的状态。“正在被跟踪”指的是进程暂停下来，等待跟踪它的进程对它进行操作。比如在gdb中对被跟踪的进程下一个断点，进程在断点处停下来的时候就处于TASK_TRACED状态。而在其他时候，被跟踪的进程还是处于前面提到的那些状态。

而TASK_TRACED状态相当于在TASK_STOPPED之上多了一层保护，处于TASK_TRACED状态的进程不能响应SIGCONT信号而被唤醒。只能等到调试进程通过ptrace系统调用执行PTRACE_CONT、PTRACE_DETACH等操作（通过ptrace系统调用的参数指定操作），或调试进程退出，被调试的进程才能恢复TASK_RUNNING状态。

Linux进程状态：Z (TASK_DEAD - EXIT_ZOMBIE)，退出状态，进程成为僵尸进程。

进程在退出的过程中，处于TASK_DEAD状态。

在这个退出过程中，进程占有的所有资源将被回收，除了task_struct结构（以及少数资源）以外。于是进程就只剩下task_struct这么个空壳，故称为僵尸。之所以保留task_struct，是因为task_struct里面保存了进程的退出码、以及一些统计信息。

父进程可以通过wait系列的系统调用（如wait4、waitid）来等待某个或某些子进程的退出，并获取它的退出信息。然后wait系列的系统调用会顺便将子进程的尸体（task_struct）也释放掉。

子进程在退出的过程中，内核会为其父进程发送一个信号，通知父进程来“收尸”。这个信号默认是SIGCHLD，但是在通过clone系统调用创建子进程时，可以设置这个信号。

通过下面的代码能够制造一个EXIT_ZOMBIE状态的进程：

```
#include
void main() {
if (fork())
while(1) sleep(100);
}
```

编译运行，然后ps一下：

```
$ ps -ax | grep a.out
```

```
10410 pts/0  S+   0:00 ./a.out
```

```
10411 pts/0  Z+   0:00 [a.out]
```

```
10413 pts/1  S+   0:00 grep a.out
```

只要父进程不退出，这个僵尸状态的子进程就一直存在。那么如果父进程退出了呢，谁又来给子进程“收尸”？

当进程退出的时候，会将它的所有子进程都托管给别的进程（使之成为别的进程的子进程）。托管给谁呢？可能是退出进程所在进程组的下一个进程（如果存在的话），或者是1号进程。所以每个进程、每时每刻都有父进程存在。除非它是1号进程。1号进程，pid为1的进程，又称init进程。

Linux进程状态：X (TASK_DEAD - EXIT_DEAD)，退出状态，进程即将被销毁。

而进程在退出过程中也可能不会保留它的task_struct。比如这个进程是多线程程序中被detach过的进程。或者父进程通过设置SIGCHLD信号的handler为SIG_IGN，显式的忽略了SIGCHLD信号。此时，进程将被置于EXIT_DEAD退出状态，这意味着接下来的代码立即就会将该进程彻底释放。所以EXIT_DEAD状态是非常短暂的，几乎不可能通过ps命令捕捉到。

。

5.2.2 进程、线程和内核线程

在Linux内核中，内核是采用进程、线程和内核线程统一管理的方法实现进程管理的。

内核将进程、线程和内核线程一视同仁

- 1.即内核使用唯一的数据结构`task_struct`来分别表示它们；
- 2.内核使用相同的调度算法对这三者进行调度；
- 3.内核也使用同一个函数`do_fork()`来分别创建这三种执行线程（thread of execution）。

执行线程通常是指任何正在执行的代码实例，比如一个内核线程，一个中断处理程序或一个进入内核的进程。

进程是系统资源分配的基本单位，线程是程序独立运行的基本单位

线程有时候也被称作小型进程

如果内核要对线程进行调度，那么线程必须如同进程那样在内核中对应一个数据结构。进程在内核中有相应的**进程描述符，即task_struct结构**。

在内核中还有一种特殊的线程，称之为**内核线程**（Kernel Thread）。由于在内核中进程和线程不做区分，因此也可以将其称为内核进程。内核线程在内核中也是通过**task_struct**结构来表示的

内核线程和普通进程的不同:

1. 内核线程永远都运行在内核态，而普通进程既可以运行在用户态也可以运行在内核态。从地址空间的使用角度来讲，内核线程只能使用大于**3GB**的地址空间，而普通进程则可以使用整个**4GB**的地址空间。
2. 内核线程只能调用内核函数无法使用用户空间的函数，而普通进程必须通过系统调用才能使用内核函数。

5.2.3 进程描述符task_struct的几个特殊字段

(1) mm字段：指向mm_struct结构的指针，该类型用来描述进程整个的虚拟地址空间。其数据结构如下：

```
struct mm_struct *mm, *active_mm;
#ifdef CONFIG_COMPAT_BRK
    unsigned brk_randomized:1;
#endif
#if defined(SPLIT_RSS_COUNTING)
    struct task_rss_stat  rss_stat;
#endif
```

5.2.3 进程描述符task_struct的几个特殊字段

(2) fs字段：指向fs_struct结构的指针，该字段用来描述进程所在文件系统的根目录和当前进程所在的目录信息。

3) files字段：指向files_struct结构的指针，该字段用来描述当前进程所打开文件的信息。

(4) signal字段：指向signal_struct结构（信号描述符）的指针，该字段用来描述进程所能处理的信号。其数据结构如下；

```
/* signal handlers */
struct signal_struct *signal;
struct sighand_struct *sighand;
sigset_t blocked, real_blocked;
sigset_t saved_sigmask; /* restored if set_restore_sigmas
k() was used */
struct sigpending pending;
unsigned long sas_ss_sp;
size_t sas_ss_size;
int (*notifier)(void *priv);
void *notifier_data;
sigset_t *notifier_mask;
```

对于普通进程来说，上述字段分别指向具体的数据结构以表示该进程所拥有的资源。对应每个线程而言，内核通过轻量级进程与其进行关联。轻量级进程之所轻量，是因为它与其它进程共享上述所提及的进程资源。

举例：

进程A创建了线程B，则B线程会在内核中对应一个轻量级进程。这个轻量级进程对应一个进程描述符，而且B线程的进程描述符中的某些代表资源指针会和A进程中对应的字段指向同一个数据结构，这样就实现了多线程之间的资源共享。

5.2.4 do_fork()函数

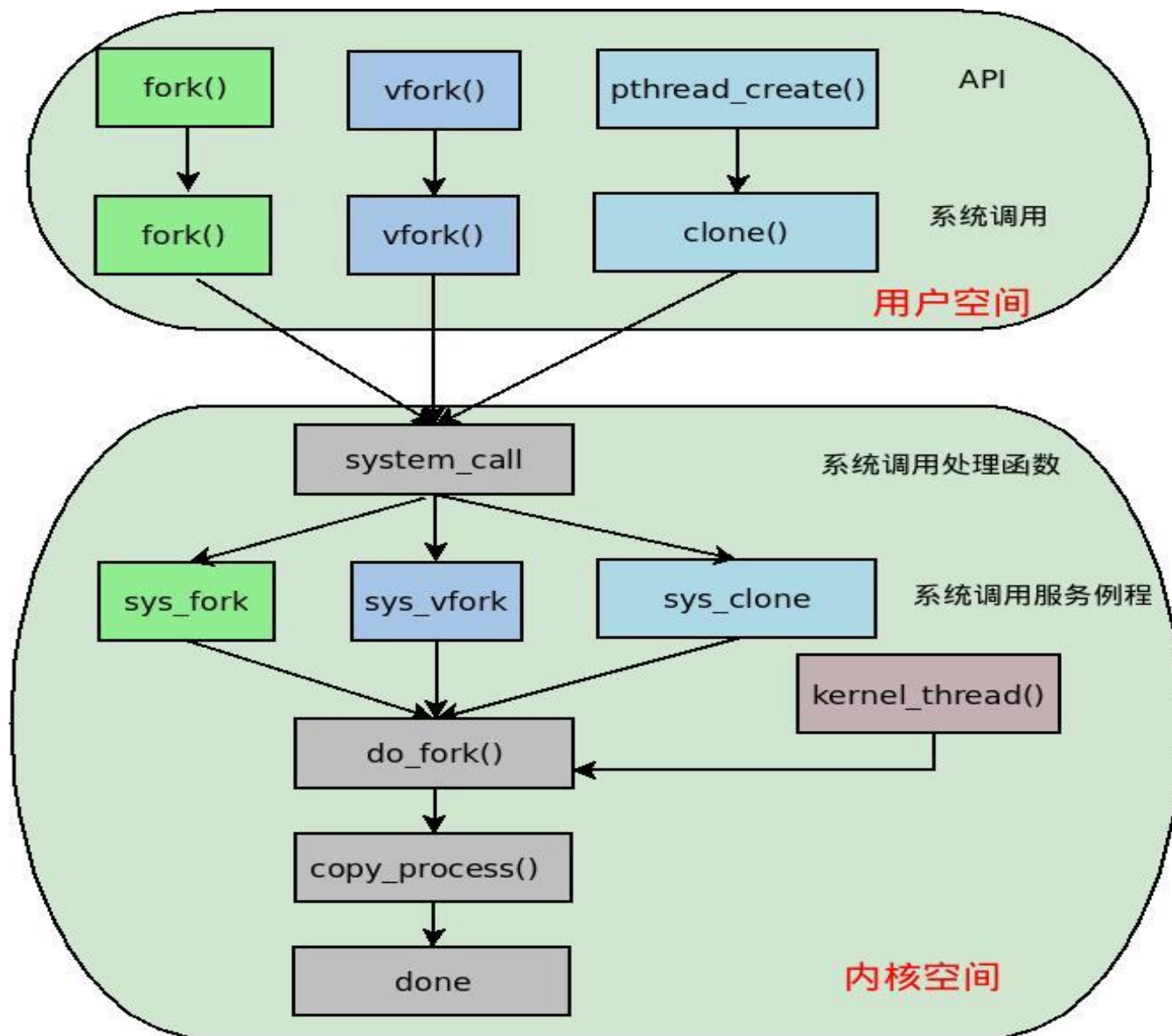


图5-6
`do_fork()`函数
对于进程、线程以及
内核线程的应用

内核中创建进程的核心函数即为`do_fork()`，该函数的原型如下：

```
long do_fork(unsigned long clone_flags,  
             unsigned long stack_start,  
             struct pt_regs *regs,  
             unsigned long stack_size,  
             int __user *parent_tidptr,  
             int __user *child_tidptr)
```

该函数的参数的功能说明如下：

clone_flags：代表进程各种特性的标志。低字节指定子进程结束时发送给父进程的信号代码，一般为SIGCHLD信号，剩余三个字节是若干个标志或运算的结果。

stack_start：子进程用户态堆栈的指针，该参数会被赋值给子进程的esp寄存器。

regs: 指向通用寄存器值的指针，当进程从用户态切换到内核态时通用寄存器中的值会被保存到内核态堆栈中。

stack_size: 未被使用，默认值为0。

parent_tidptr: 该子进程的父进程用户态变量的地址，仅当CLONE_PARENT_SETTID被设置时有效。

child_tidptr: 该子进程用户态变量的地址，仅当CLONE_CHILD_SETTID被设置时有效。

一个问题：既然进程、线程和内核线程在内核中都是通过do fork()完成创建的，那么do fork()是如何体现其功能的多样性？

clone_flags参数在这里起到了关键作用，通过选取不同的标志，从而保证了do_fork()函数实现多角色——创建进程、线程和内核线程——功能的实现。

下面只介绍其中几个主要的标志。

CLONE_VIM: 子进程共享父进程内存描述符和所有的页表。

CLONE_FS: 子进程共享父进程所在文件系统的根目录和当前工作目录。

CLONE_FILES: 子进程共享父进程打开的文件。

CLONE_SIGHAND: 子进程共享父进程的信号处理程序、阻塞信号和挂起的信号。使用该标志必须同时设置CLONE_VM标志。

如果创建子进程时设置了上述标志，那么子进程会共享这些标志所代表的父进程资源。

5.2.5 进程的创建

在用户态程序中，可以通过**fork()**、**vfork()**和**clone()**三个接口函数创建进程，这三个函数在库中分别对应同名的系统调用。系统调用函数通过128号软中断进入内核后，会调用相应的系统调用服务例程。分别是**sys_fork()**、**sys_vfork()**和**sys_clone()**。

```
int sys_fork(struct pt_regs *regs)  
{  
    return do_fork(SIGCHLD, regs->sp, regs, 0,  
NULL, NULL);  
}
```

```
int sys_vfork(struct pt_regs *regs)  
{  
    return do_fork(CLONE_VFORK | CLONE_VM |  
SIGCHLD, regs->sp, regs, 0, NULL, NULL);  
}
```

```
Long sys_clone(unsigned long clone_flags, unsigned long
newsp,
void __user *parent_tid, void __user *child_tid, struct
pt_regs *regs)
{
    if (!newsp)
        newsp = regs->sp;
    return do_fork(clone_flags, newsp, regs, 0,
parent_tid, child_tid);
}
```

三个系统服务例程内部都调用了 *do_fork()*，主要差别在于第一个参数所传的值不同。

下面予以说明：

(1) **fork()**：由于**do_fork()**中**clone_flags**参数除了子进程结束时返回给父进程的**SIGCHLD**信号外并无其他特性标志，因此由**fork()**创建的进程不会共享父进程的任何资源。

子进程会完全复制父进程的资源，也就是说父子进程相对独立。不过由于写时复制技术（**Copy On Write**）的引入，子进程可以只读父进程的物理页，只有当父进程或者子进程去写某个物理页时，内核此时才会将这个页的内容拷贝到一个新的物理页，并把这个新的物理页分配给正在写的进程。

(2) `vfork()`: `do_fork()`中的`clone_flags`使用了`CLONE_VFORK`和`CLONE_VM`两个标志。

`CLONE_VFORK`标志使得子进程先于父进程执行，父进程会阻塞到子进程结束或执行新的程序。

`CLONE_VM`标志使得子进程可以共享父进程的内存地址空间（父进程的页表项除外）。在写时复制技术引入之前，`vfork()`适用于子进程形成后立即执行`execv()`的情形。

因此，`vfork()`现如今已经没有特别的使用之处，因为写时复制技术完全可以取代它创建进程时所带来的高效性。

(3) **clone()**: **clone**通常用于创建轻量级进程。通过传递不同的标志可以对父子进程之间数据的共享和复制作精确的控制，一般**flags**的取值为**CLONE_VM|CLONE_FS|CLONE_FILES|CLONE_SIGHAND**。由上述标志可以看到，轻量级进程通常共享父进程的内存地址空间、父进程所在文件系统的根目录以及工作目录信息、父进程当前打开的文件以及父进程所拥有的信号处理函数。

5.2.6 线程和内核线程的创建

每个线程在内核中对应一个轻量级进程，两者的关联是通过线程库完成的。因此通过`pthread_create()`创建的线程最终在内核中是通过`clone()`完成创建的，而`clone()`最终调用`do_fork()`。

一个新内核线程的创建是通过在现有的内核线程中使用 `kernel_thread()` 而创建的，其本质也是向 `do_fork()` 提供特定的 `flags` 标志而创建的。

```
Int kernel_thread(int (*fn)(void*), void *arg, unsigned long
flags)
{
return do_fork(flags|CLONE_VM|CLONE_UNTRACED, 0,
&regs, 0, NULL, NULL);
}
```

从上面的组合的 `flags` 标志可以看出，新的内核线程至少会共享父内核线程的内存地址空间。

5.2.7 进程的执行---exec函数族

在linux中中exec函数族提供了一个在进程中启动另一个程序执行的方法。

它可以根据指定的文件名或目录名找到可执行文件，并用它来取代原调用进程的数据段、代码段和堆栈段，在执行完之后，原调用进程的内容除了进程号外，其他全部被新的进程替换了。

在Linux中使用exec函数族主要有两种情况：

(1) 当进程认为自己不能再为系统和用户做出任何贡献时，就可以调用exec函数族中的任意一个函数让自己重生。

(2) 如果一个进程希望执行另一个程序，那么它就可以调用fork()函数新建一个进程，然后调用exec函数族中的任意一个函数，这样看起来就像通过执行应用程序而产生了一个新进程。

实际上，在Linux中并没有exec()函数，而是有6个以exec开头的函数

所需头文件	#include <unistd.h>
函数原型	int execl(const char *path, const char *arg, ...)
	int execv(const char *path, char *const argv[])
	int execlp(const char *path, const char *arg, ..., char *const envp[])
	int execve(const char *path, char *const argv[], char *const envp[])
	int execlp(const char *file, const char *arg, ...)
	int execvp(const char *file, char *const argv[])
函数返回值	-1：出错

事实上，这6个函数中真正的系统调用只有**execve()**，其他5个都是库函数，它们最终都会调用**execve()**这个系统调用。这里简要介绍**execve()**执行的流程：

- (1) 打开可执行文件，获取该文件的*file*结构；
- (2) 获取参数区长度，将存放参数的页面清零；
- (3) 对*linux_binprm*结构的其它项作初始化。这里的*linux_binprm*结构用来读取并存储运行可执行文件的必要信息。

5.2.8 进程的终止

当进程终结时，内核必须释放它所占有的资源，并告知其父进程。进程的终止可以通过以下三个事件驱动：

正常的进程结束

信号

`exit()` 函数的调用。

进程的终结最终都要通过`do_exit()`来完成。

`exit()` 函数所需的头文件为`#include <stdlib.h>`，函数原型是：

```
void exit(int status)
```

do_exit () 的执行过程:

- (1)**将task_struct中的标志成员设置PF_EXITING，表明该进程正在被删除，释放当前进程占用的mm_struct，如果没有别的进程使用，即没有被共享，就彻底释放它们
- (2)**如果进程排队等候IPC信号，则离开队列
- (3)**分别递减文件描述符、文件系统数据、进程名字空间的引用计数。如果这些引用计数的数值降为0，则表示没有进程在使用这些资源，可以释放。
- (4)**向父进程发送信号：将当前进程的子进程的父进程重新设置为线程组中的其他线程或者init进程，并把进程状态设成TASK_ZOMBIE
- (5)**切换到其他进程，处于TASK_ZOMBIE状态的进程不会再被调用。此时进程占用的资源就是内核堆栈、thread_info结构、task_struct结构。此时进程存在的唯一目的就是向它的父进程提供信息。父进程检索到信息后，或者通知内核那是无关的信息后，由进程所持有的剩余内存被释放，归还给系统使用。

5.2.9 进程的调度

Linux进程调度分为**主动调度**和**被动调度**两种方式：

主动调度随时都可以进行，内核里可以通过`schedule()`启动一次调度，当然也可以将进程状态设置为`TASK_INTERRUPTIBLE`、`TASK_UNINTERRUPTIBLE`，暂时放弃运行而进入睡眠，用户空间也可以通过`pause()`达到同样的目的；如果为这种暂时的睡眠放弃加上时间限制，内核态有`schedule_timeout`，用户态有`nanosleep()`用于此目的。

被动调度发生在系统调用返回、**中断异常处理返回**或用户态处理软中断返回。

从Linux 2.6内核后，linux实现了**抢占式内核**，

linux2.6内核之前的版本会恢复原进程的运行，直到该进程退出内核态才会引发调度程序；

而linux2.6抢占式内核，在处理完中断后，会立即引发调度，切换到高权值进程。

为支持内核代码可抢占，在2.6版内核中通过采用禁止抢占的**自旋锁 (spin unlock mutex)**来保护临界区。

2. 进程调度的一般原理

调度程序运行时，要在所有可运行的进程中选择最值得运行的进程。选择进程的依据主要有：

进程的调度策略（policy）、
静态优先级（priority）、
动态优先级（counter）、
实时优先级（rt-priority）。

Policy是进程的调度策略，用来区分实时进程和普通进程

Counter是实际意义上的进程动态优先级，它是进程剩余的时间片，起始值就是priority的值。

在linux中，用函数goodness（）综合四项依据及其他因素，赋予各影响因素权重（weight），调度程序以权重作为选择进程的依据。

3. Linux O(1)调度

O(1) 调度程序：内核实现了一种新型的调度算法，不管有多少个线程在竞争 CPU，这种算法都可以在**固定时间**内进行操作。这个名字就表示它调度多个线程所使用的时间和调度一个线程所使用的时间是相同的。

Linux2.6实现O(1)调度，每个CPU都有两个进程队列，采用**优先级为基础**的调度策略。内核为每个进程计算出一个反映其运行“资格”的权值，然后挑选权值最高的进程投入运行。在运行过程中，当前进程的资格随时间而递减，从而在下一次调度的时候原来资格较低的进程可能就有资格运行了。到所有进程的资格都为零时，就重新计算。

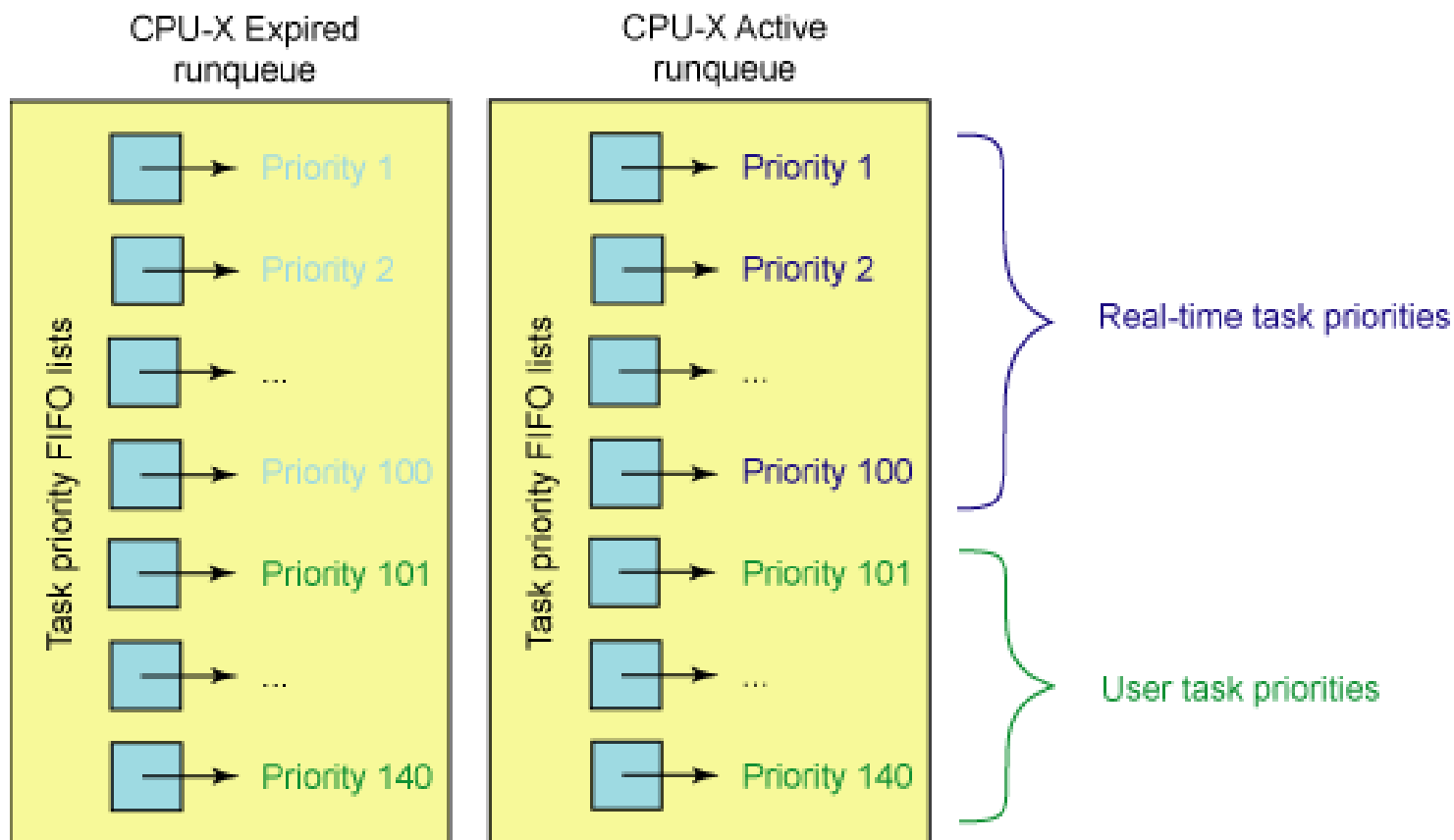
活动进程和过期进程

对于系统内处于可运行状态的进程，我们可以分为三类：

- 首先是正处于执行状态的那个进程；
- 其次，有一部分处于可运行状态的进程则还没有用完他们的时间片，他们等待被运行；
- 剩下的进程已经用完了自己的时间片，在其他进程没有用完它们的时间片之前，他们不能再被运行。

据此，我们将进程分为两类，活动进程，那些还没有用完时间片的进程；过期进程，那些已经用完时间片的进程。因此，调度程序的工作就是在活动进程集合中选取一个最佳优先级的进程，如果该进程时间片恰好用完，就将该进程放入过期进程集合中。

调度器为每一个CPU维护了两个进程队列数组：指向活动运行队列的**active**数组和指向过期运行队列的**expire**数组。数组中的元素着保存某一优先级的进程队列指针。系统一共有**140**个不同的优先级，因此这两个数组大小都是**140**。它们是按照先进先出的顺序进行服务的。被调度执行的任务都会被添加到各自运行队列优先级列表的末尾。每个任务都有一个时间片，这取决于系统允许执行这个任务多长时间。运行队列的前**100**个优先级列表保留给实时任务使用，后**40**个用于用户任务，参见下图：



当需要选择当前最高优先级的进程时，2.6调度器不用遍历整个runqueue，而是直接从active数组中选择当前最高优先级队列中的第一个进程。假设当前所有进程中最高优先级为50（换句话说，系统中没有任何进程的优先级小于50）。则调度器直接读取 `active[49]`，得到优先级为50的进程队列指针。该队列头上的第一个进程就是被选中的进程。这种算法的复杂度为 $O(1)$ ，从而解决了2.4调度器的扩展性问题。为了实现 $O(1)$ 算法active数组维护了一个由5个32位的字（140个优先级）组成的bitmap，当某个优先级级别上有进程被插入列表时，相应的比特位就被置位。

Schedule()函数是完成进程调度的主要函数，并完成进程切换的工作。它在/**kernel/sched.c**中的定义如下：

```
/*schedule() is the main scheduler function. */
```

```
asmlinkage void __sched schedule(void)
```

```
{
```

```
    struct task_struct *prev, *next;
```

```
    unsigned long *switch_count;
```

```
    struct rq *rq;
```

```
    int cpu;
```

```
need_resched:
```

```
    preempt_disable();
```

```
    cpu = smp_processor_id();
```

```
    rq = cpu_rq(cpu);
```

```
    rcu_sched_qs(cpu);
```

```
    prev = rq->curr;
```

```
    switch_count = &prev->nivcsw;
```

```
    release_kernel_lock(prev);
```

schedule的主要工作可以分为两步。

首先是找到**next**。

1.**schedule()**检查**prev**的状态。

2.检查本地运行队列中是否有进程。

3.若本地运行队列中有进程，但活动进程队列为空集。这时把活动进程队列改为过期进程队列，把原过期进程队列改为活动进程队列。空集用于接收过期进程。

4.在活动进程队列中搜索一个可运行进程。

5.检查**next**是否是实时进程以及是否从**TASK_INTERRUPTIBLE**或**TASK_STOPPED**状态中被唤醒。

找到next后，就可以实施进程切换了。

- 1.把next的进程描述符第一部分字段的内容装入硬件高速缓存。
- 2.清除prev的TIF_NEED_RESCHED的标志。
- 3.设置prev的进程切换时刻。
- 4.重新计算并设置prev的平均睡眠时间。
- 5.如果prev != next，切换prev和next硬件上下文。

这时，CPU已经开始执行next进程了。

5.3

Part Three

ARM- Linux内存管理

5.3.1 ARM-Linux内存管理概述

内存管理是Linux内核中最重要的子系统之一，它主要提供对内存资源的访问控制机制。这种机制主要涵盖了：

内存的分配和回收。

地址转换。

内存扩充。

内存的共享与保护。

Linux系统会在硬件物理内存和进程所使用的内存（称作虚拟内存）之间建立一种映射关系，这种映射是以进程为单位，因而不同的进程可以使用相同的虚拟内存，而这些相同的虚拟内存，可以映射到不同的物理内存上。

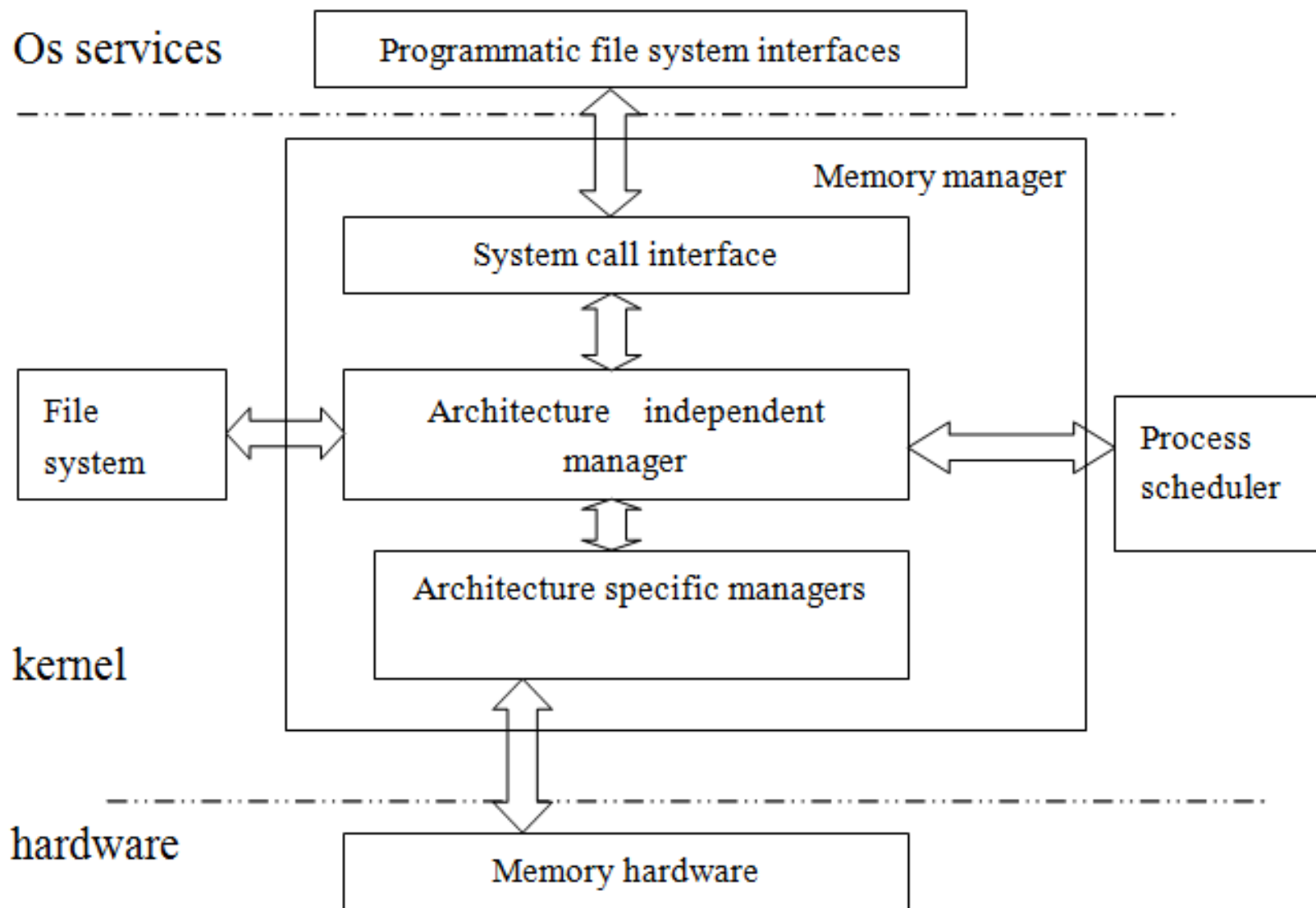


图5-7 内存管理主要子系统架构

Architecture Specific Managers子模块，涉及体系结构相关部分，提供用于访问硬件Memory的虚拟接口。

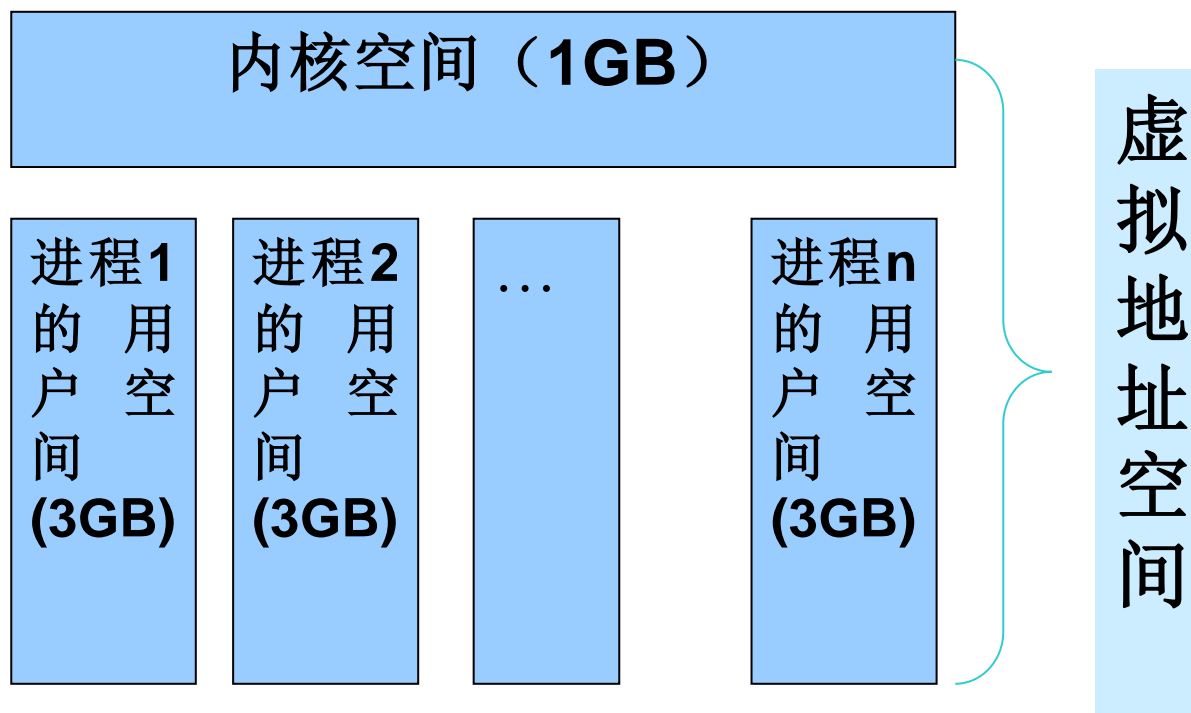
Architecture Independent Manager子模块，涉及体系结构无关部分，提供所有的内存管理机制，包括以进程为单位的memory mapping、虚拟内存的交换技术Swapping等。

System Call Interface系统调用接口。通过该接口，向用户空间程序应用程序提供内存的分配、释放，文件的映射等功能。

ARM-Linux内核的内存管理功能是采用请求调页式的虚拟存储技术实现的。ARM-Linux内核根据内存的当前使用情况动态换进换出进程页，通过外存上的交换空间存放换出页。内存与外存之间的相互交换信息是以页为单位进行的，这样的管理方法具有良好的灵活性，并具有很高的内存利用率。

5.3.2 ARM- Linux虚拟存储空间及分布

在系统空间，即在内核中，虚拟地址与物理地址在数值上是相同的，至于用户空间的地址映射是动态的，根据需要分配物理内存，并且建立起具体进程的虚拟地址与所分配的物理内存间的映射。需要值得注意的是，系统空间的一部分不是映射到物理内存，而是映射到一些I/O设备，包括寄存器和一些小块的存储器。



0XC0000000

ARM-Linux 内核

4GB

内核空间

3GB

环境变量

参数

栈

堆

...

(bss)

数据段

代码段

0X00000000



栈：栈是用户存放程序临时创建的局部变量

堆 (heap)：堆是用于存放进程运行中被动态分配的内存段，

用户空间

BSS段：BSS段包含了程序中未初始化的全局变量，

代码段：代码段是用来存放可执行文件的操作指令

数据段：数据段用来存放可执行文件中已初始化全局变量

BSS段:

- **BSS段**（**bss segment**）通常是指用来存放程序中未初始化的全局变量的一块内存区域。**BSS**是英文**Block Started by Symbol**的简称。**BSS**段属于静态内存分配。

数据段:

- 数据段（**data segment**）通常是指用来存放程序中已初始化的全局变量的一块内存区域。数据段属于静态内存分配。

代码段:

- 代码段（code segment/text segment）通常是指用来存放程序执行代码的一块内存区域。这部分区域的大小在程序运行前就已经确定，并且内存区域通常属于只读，某些架构也允许代码段为可写，即允许修改程序。在代码段中，也有可能包含一些只读的常数变量，例如字符串常量等。

堆 (*heap*) :

- 堆是用于存放进程运行中被动态分配的内存段，它的大小并不固定，可动态扩张或缩减。当进程调用**malloc**等函数分配内存时，新分配的内存就被动态添加到堆上（堆被扩张）；当利用**free**等函数释放内存时，被释放的内存从堆中被剔除（堆被缩减）

栈(stack):

- 栈又称堆栈， 是用户存放程序临时创建的局部变量，也就是说我们函数括弧“{}”中定义的变量（但不包括**static**声明的变量， **static**意味着在数据段中存放变量）。除此以外，在函数被调用时，其参数也会被压入发起调用的进程栈中，并且待到调用结束后，函数的返回值也会被存放回栈中。由于栈的先进先出特点，所以栈特别方便用来保存/恢复调用现场。从这个意义上讲，我们可以把堆栈看成一个寄存、交换临时数据的内存区。

例一】

- 用cl编译两个小程序如下：

程序1:

- ```
int ar[30000];
void main()
{
.....
}
```

程序2:

- ```
int ar[300000] = {1, 2, 3, 4, 5, 6 };  
void main()  
{  
.....  
}
```

- 发现程序2编译之后所得的.exe文件比程序1的要大得多。手工编译了一下，并使用了/FAs编译选项来查看了一下其各自的.asm，发现在程序1.asm中ar的定义如下：
_BSS SEGMENT
?ar@@@3PAHA DD 0493e0H DUP (?) ; ar
_BSS ENDS
- 而在程序2.asm中，ar被定义为：
_DATA SEGMENT
?ar@@@3PAHA DD 01H ; ar
DD 02H
DD 03H
ORG \$+1199988
_DATA ENDS

说明：

- 区别很明显，一个位于**.bss**段，而另一个位于**.data**段，两者的区别在于：全局的未初始化变量存在于**.bss**段中，具体体现为一个占位符；全局的已初始化变量存于**.data**段中；而函数内的自动变量都在栈上分配空间。**.bss**是不占用**.exe**文件空间的，其内容由操作系统初始化（清零）；而**.data**却需要占用，其内容由程序初始化，因此造成了上述情况。

说明：

- **bss**段（未手动初始化的数据）并不给该段的数据分配空间，只是记录数据所需空间的大小。
data（已手动初始化的数据）段则为数据分配空间，数据保存在目标文件中。
- 数据段包含经过初始化的全局变量以及它们的值。
。**BSS**段的大小从可执行文件中得到，然后链接器得到这个大小的内存块，紧跟在数据段后面。
当这个内存区进入程序的地址空间后全部清零。
包含数据段和**BSS**段的整个区段此时通常称为数据区。

【例二】

如下程序（test.cpp）

- ```
#include <stdio.h>

#define LEN 1002000

int inbss[LEN];
float fA;
int indata[LEN]={1,2,3,4,5,6,7,8,9};
double dbB = 100.0;
const int cst = 100;

int main(void)
{
 int run[100] = {1,2,3,4,5,6,7,8,9};
 int *ii;
 ii = (int *) malloc(sizeof(int));
 for(int i=0; i<LEN; ++i)
 printf("%d ", inbss[i]);
 return 0;
}
```

程序中变量各  
处于什么段？

## 5.3.3 进程空间描述

### 关键数据结构描述

一个进程的虚拟地址空间主要由两个数据结来描述。一个是最高层次的：`mm_struct`，一个是较高层次的：`vm_area_structs`。

**最高层次的`mm_struct`结构描述了一个进程的整个虚拟地址空间。**每个进程只有一个`mm_struct`结构，在每个进程的`task_struct`结构中，有一个指向该进程的`mm_struct`结构的指针，每个进程与用户相关的各种信息都存放在`mm_struct`结构体中，其中包括本进程的页目录表的地址，本进程的用户区的组成情况等重要信息。

**可以说，`mm_struct`结构是对整个用户空间的描述。**

**vm\_area\_struct**是描述进程地址空间的基本管理单元，Linux内核中对应进程内存区域的数据结构是：**vm\_area\_struct**，内核将每个内存区域作为一个单独的内存对象管理，相应的操作也都一致。

**每个进程的用户区是由一组vm\_area\_struct结构体组成的链表来描述的。**用户区的每个段（如代码段、数据段和栈等）都由一个**vm\_area\_struct**结构体描述，其中包含了本段的起始虚拟地址和结束虚拟地址，也包含了当发生缺页异常时如何找到本段在外存上的相应内容（如通过**nopage**函数）。



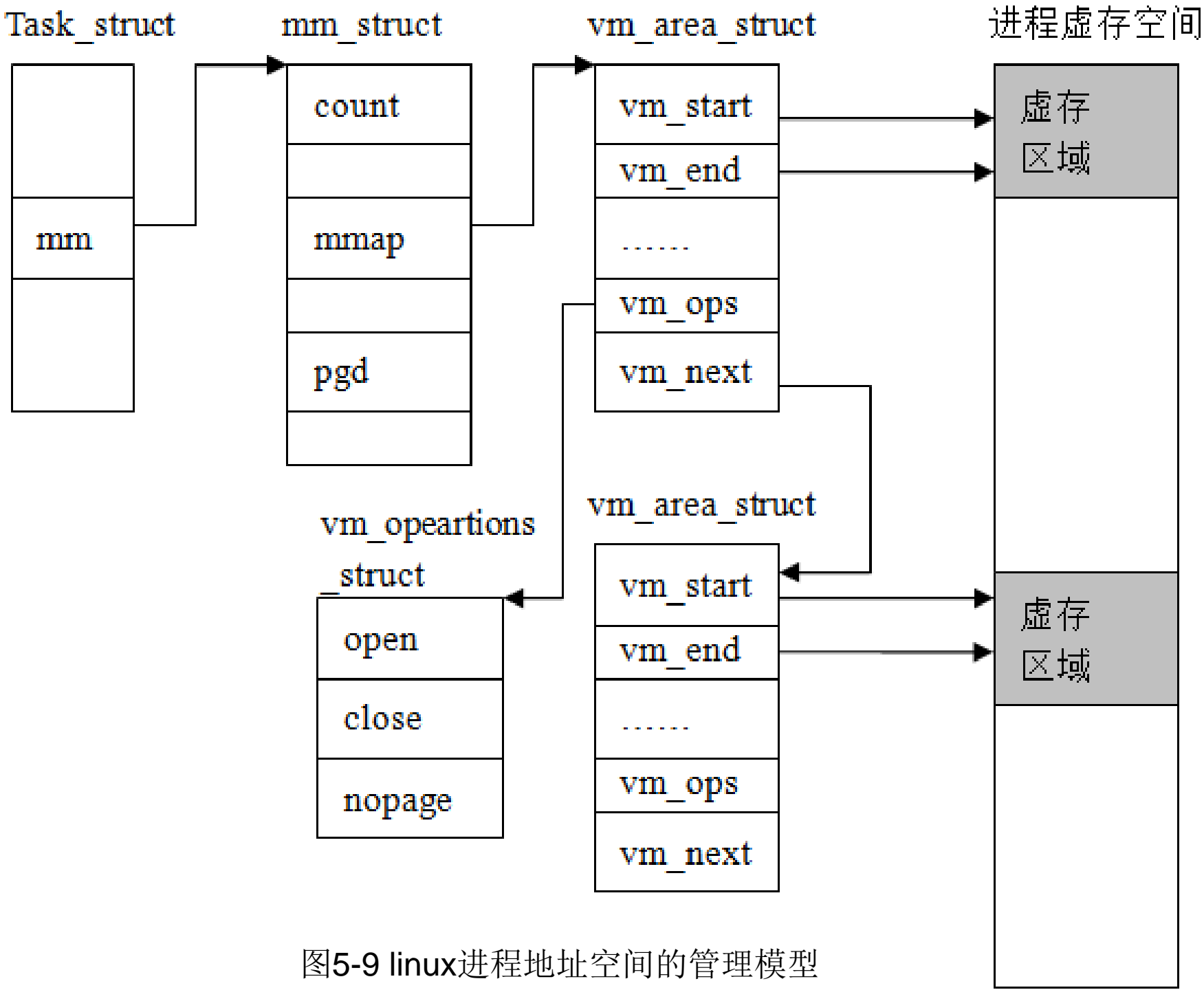
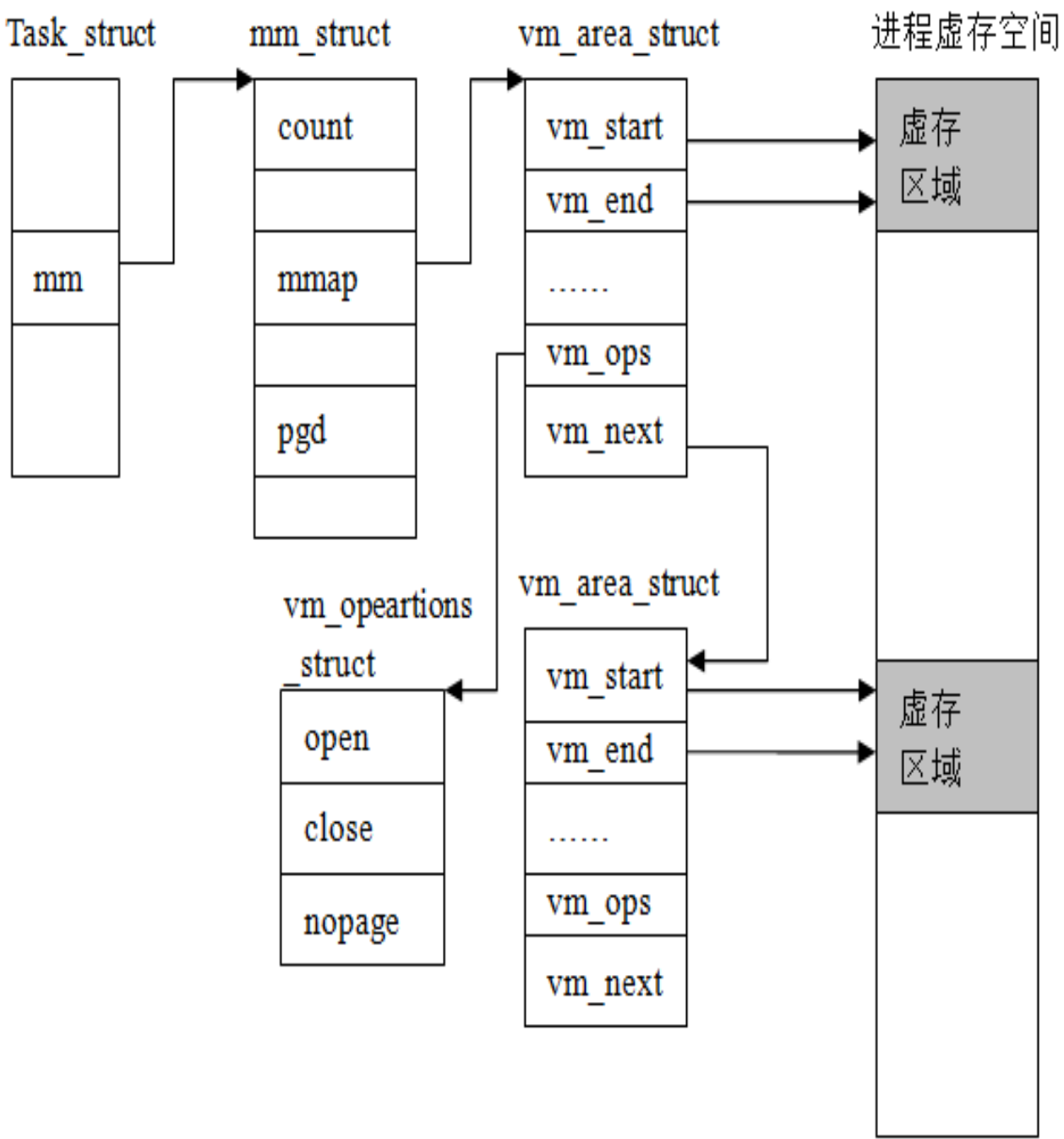


图5-9 linux进程地址空间的管理模型

内存映射(mmap)是Linux操作系统的一个很大特色，它可以将系统内存映射到一个文件（设备）上，以便可以通过访问文件内容来达到访问内存的目的。



vm\_area\_struct结构体描述如下:

```
struct vm_area_struct {
 struct mm_struct * vm_mm; /* The address space we belong to. */
 unsigned long vm_start; /* Our start address within vm_mm. */
 unsigned long vm_end; /* The first byte after our end address
 within vm_mm. */

 struct vm_area_struct *vm_next, *vm_prev;
 pgprot_t vm_page_prot; /* Access permissions of this VMA. */
 unsigned long vm_flags; /* Flags, see mm.h. */
 struct rb_node vm_rb;
 union {
 struct {
 struct list_head list;
 void *parent; /* aligns with prio_tree_node parent */
 struct vm_area_struct *head;
 } vm_set;
 struct raw_prio_tree_node prio_tree_node;
 } shared;
};
```

## 2. Linux的分页模型

在Linux2.6中，Linux采用了通用的四级页表结构，四种页表分别称为：页全局目录、页上级目录、页中间目录、页表。

为了实现跨平台运行Linux的目标（如在ARM平台上），设计者提供了一系列转换宏使得Linux内核可以访问特定进程的页表。该系列转换宏实现逻辑页表和物理页表在逻辑上的一致。这样内核无需知道页表入口的结构和排列方式。采用这种方法后，在使用不同级数页表的处理器架构中，Linux就可以使用相同的页表操作代码了

分页机制将整个线性地址空间及整个物理内存看成由许多大小相同的存储块组成的，并把这些块作为**页**（虚拟空间分页后每个单位称为页）或**页帧**（物理内存分页后每个单位称为页帧）进行管理。

### 5.3.3.物理内存管理（页管理）

内核分配物理页面时为了尽量减少不连续情况，采用了“**伙伴**”（**buddy**）**算法**来管理空闲页面。

Linux 系统采用伙伴算法管理系统页框的分配和回收，该算法对不同的管理区使用单独的伙伴系统管理。伙伴算法把内存中的所有页框按照大小分成10组不同大小的页块，每块分别包含1、2、4、...、512个页框。每种不同的页块都通过一个free\_area\_struct结构体来管理。系统将10个free\_area\_struct结构体组成一个free\_area[]数组。

```
typedef struct free__area__
struct
{
 struct list__head free__list ;
 unsigned long *map ;
} free__area__t ;
```

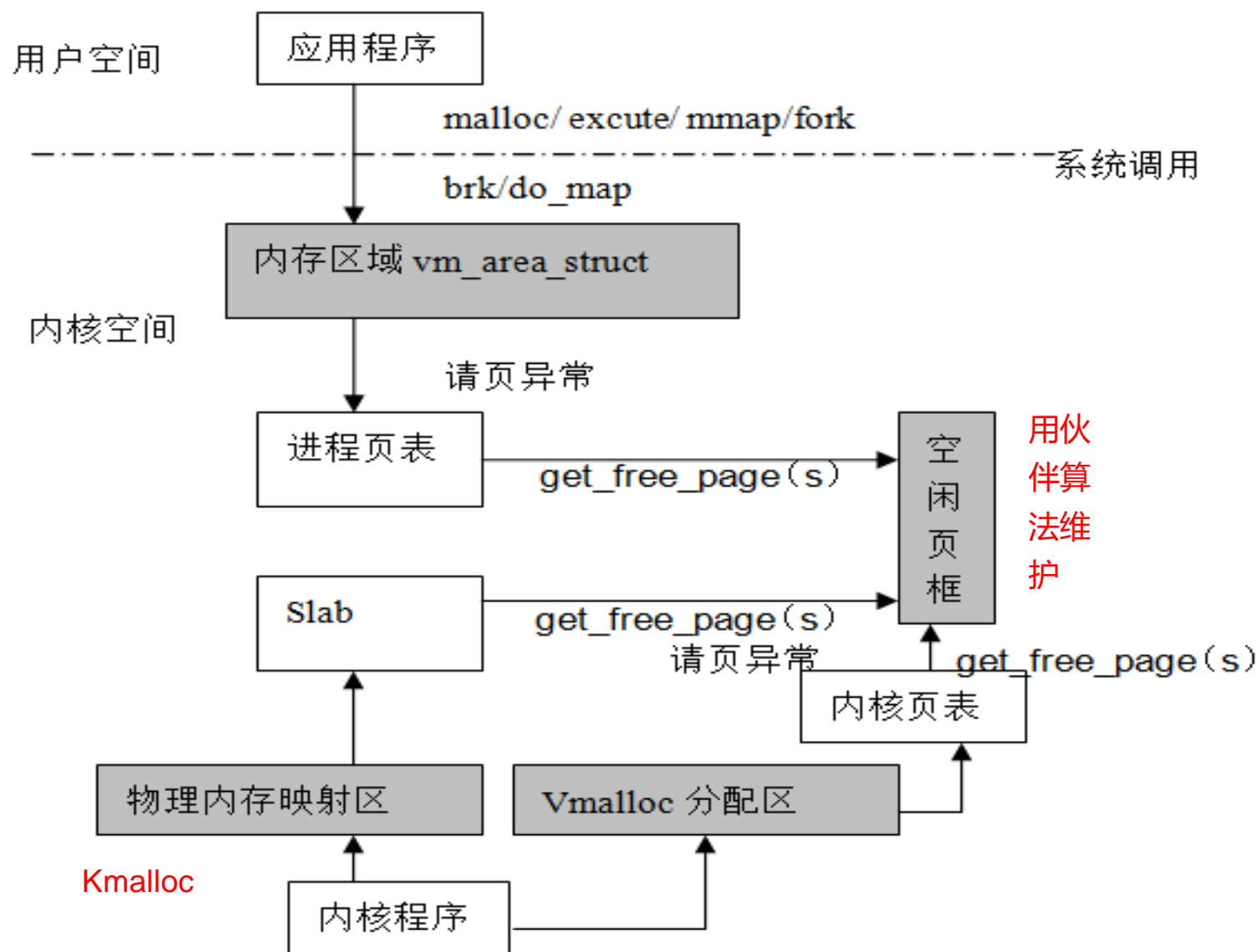


图5-10 内核空间物理页分配技术

ARM-Linux内核中分配空闲页面的基本函数是 `get_free_page/get_free_pages`，它们或是分配单页或是分配指定的页面（2、4、8...512页）。

值得注意的是：`get_free_page`是在内核中分配内存，不同于 `malloc`函数在用户空间中分配方法。

`malloc`函数利用堆动态分配，实际上是调用 `brk()` 系统调用，但一般仅分配虚拟空间，物理页面在缺页中断时分配。



## 5.3.4 基于Slab分配器的管理技术

类似 `task_struct`、`mm_struct` 等结构被内核中被频繁分配和释放，同时创建和销毁这些结构会产生一定的开销(**overhead**)。二者累计起来导致大量开销的产生。为了满足内核对这种小内存块的需要，Linux系统采用了一种被称为**slab分配器(slab allocator)**的技术。**Slab**并非是脱离伙伴关系而独立存在的一种内存分配方式，**slab**仍然是建立在页面基础之上。

**slab分配器主要的功能就是对频繁分配和释放的小对象提供高效的内存管理。**它的**核心思想是实现一个缓存池**，分配对象的时候从缓存池中取，释放对象的时候再放入缓存池。

**slab**分配器是基于对象类型进行内存管理的，每一种对象被划分为一类，例如索引节点对象是一类，进程描述符又是一类等等。每当需要申请一个特定的对象时，就从相应的类中分配一个空白的对象出去；当这个对象被使用完毕时，就重新“插入”到相应的类中

```
struct slab {
 union {
 struct {
 struct list_head list;
 unsigned long colouroff;
 void *s_mem;
 unsigned int inuse;
 kmem_bufctl_t free;
 unsigned short nodeid;
 };
 struct slab_rcu __slab_cover_slab_rcu;
 };
};
```

**Slab**分配器不仅仅只用来存放内核专用的结构体，它还被用来处理内核对小块内存的请求。一般来说内核程序中对小于一页的小块内存的请求才通过**Slab**分配器提供的接口**kmalloc**来完成（虽然它可分配32到131072字节的内存）。从内核内存分配的角度来讲，**kmalloc**可被看成是**get\_free\_page**（s）的一个有效补充，内存分配粒度更灵活了。

### 5.3.5 内核非连续内存分配（Vmalloc）

slab分配器使得一个页面内包含的众多小块内存可独立被分配使用，避免了内部分片，节约了空闲内存。伙伴关系把内存块按大小分组管理，一定程度上减轻了外部碎片的危害，但并未彻底消除。

Linux内核允许内核程序在内核地址空间中分配虚拟地址，同样也利用页表（内核页表）将虚拟地址映射到分散的内存页上。以此完美地解决了内核内存使用中的外部碎片问题。

内核提供 `vmalloc` 函数分配内核虚拟内存，该函数不同于 `kmalloc`，它可以分配较 `kmalloc` 大得多的内存空间（可远大于 128K，但必须是页大小的倍数），但相比 `Kmalloc` 来说，`vmalloc` 需要对内核虚拟地址进行重映射，必须更新内核页表，因此分配效率上相对较低。

**vmalloc**分配的内核虚拟内存与**kmalloc/get\_free\_page**分配的**内核虚拟内存**位于不同的区间，不会重叠。因为**内核虚拟空间**被分区管理，各司其职。

主要函数说明如下：

(1) **void\* vmalloc(unsigned long size)**该函数的作用是申请**size**大小的虚拟内存空间，发生错误时返回0，成功时返回一个指向大小为**size**的线性地址空间的指针。

(2) **void vfree(void \* addr)**该函数的作用是释放一个由**vmalloc()**函数申请的内存，释放内存的基地址为**addr**。

(3) **void \*vmap(struct page \*\*pages, unsigned int count, unsigned long flags, pgprot\_t prot)**该函数的作用是映射一个数组（其内容为页）到连续的虚拟空间中。第一个参数**pages**为指向页数组的指针；第二个参数**count**为要映射页的个数；第三个参数为**flags**为传递**vm\_area->flags**值；第四个参数**prot**为映射时页保护。

(4) **void vunmap(void \*addr)** 该函数的作用是释放由**vmap**映射的虚拟内存，释放从**addr**地址开始的连续虚拟区域。

### 5.3.6 页面回收简述

页面回收的方法大体上可分为两种：

**一是主动释放。**就像用户程序通过**free**函数释放曾经通过**malloc**函数分配的内存一样，页面的使用者明确知道页面的使用时机。前文所述的伙伴算法和**slab**分配器机制，一般都是由内核程序主动释放的。对于直接从伙伴系统分配的页面，这是由使用者使用**free\_pages**之类的函数主动释放的，页面释放后被直接放归伙伴系统；从**slab**中分配的对象（使用**kmem\_cache\_alloc**函数），也是由使用者主动释放的（使用**kmem\_cache\_free**函数）。

。

二是通过linux内核提供的**页框回收算法（PFRA）**进行回收。页面的使用者一般将页面当作某种缓存，以提高系统的运行效率。缓存一直存在固然好，但是如果缓存没有了也不会造成什么错误，仅仅是效率受影响而已。页面的使用者不需要知道这些缓存页面什么时候最好被保留，什么时候最好被回收，这些都交由**PFRA**来负责

# 5.4 Part Four

## ARM\_Linux模块

---



可装载模块LKM（Loadable Kernel Module）也被称为模块，即可在内核运行时加载到内核的一组目标代码

**LKM最重要的功能包括内核模块在操作系统中的加载和卸载两部分。**内核模块是一些在启动操作系统内核时如有需要可以载入内核执行的代码块，这些代码块在不需要时由操作系统卸载。模块扩展了操作系统的内核功能却不需要重新编译内核和启动系统。

## 5.4.1 LKM 的编写和编译

### 1. 内核模块的基本结构

一个内核模块至少包含两个函数，模块被加载时执行的初始化函数 `init_module()` 和模块被卸载时执行的结束函数 `cleanup_module()`。在版本 2.6 中，两个函数可以起任意的名字，通过宏 `module_init()` 和 `module_exit()` 实现。唯一需要注意的地方是函数必须在宏的使用前定义。例如：

```
static int __init hello_init(void){}
static void __exit hello_exit(void){}
module_init(hello_init);
module_exit(hello_exit);
```

*宏 `__init` 的作用是在完成初始化后收回该函数占用的内存，宏 `__exit` 用于模块被编译进内核时忽略结束函数。*

# Hello.c:

```
#include <linux/module.h>
MODULE_LICENSE("GPL");
int init_module(void){
 printk("Hello World\n");
 return 0;
}
void cleanup_module(void){
 printk("Bye,Bye\n");
}
```

## 2. 内核模块的编译

内核模块编译时需要提供一个makefile 来隐藏底层大量的复杂操作，使用户通过make 命令就可以完成编译的任务。

下面列举一个简单的编译hello.c 的makefile 文件:

```
obj-m += hello.o
```

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

```
PWD := $(shell pwd)
```

```
default:
```

```
$(MAKE) - C $(KDIR) SUBDIRS=$(PWD) modules
```

编译后获得可加载的模块文件hello.ko。

## 5.4.2 LKM版本差异比较（略）

可装载模块在 Linux 2.6 与 2.4 之间就存在巨大差异，其最大区别就是模块装载过程变化：在 Linux 2.6 中可装载模块是在内核中完成连接的。其他一些变化大致有：

模块的后缀及装载工具的变化

模块信息的附加过程的变化

模块的标记选项的变化

## 5.4.3 模块的加载与卸载

### 1. 模块的加载

模块的加载一般有两种方法，  
第一种是使用insmod 命令加载，  
另一种是当内核发现需要加载某个模块时， 请求内核后台进程kmod 加载适当的模块。

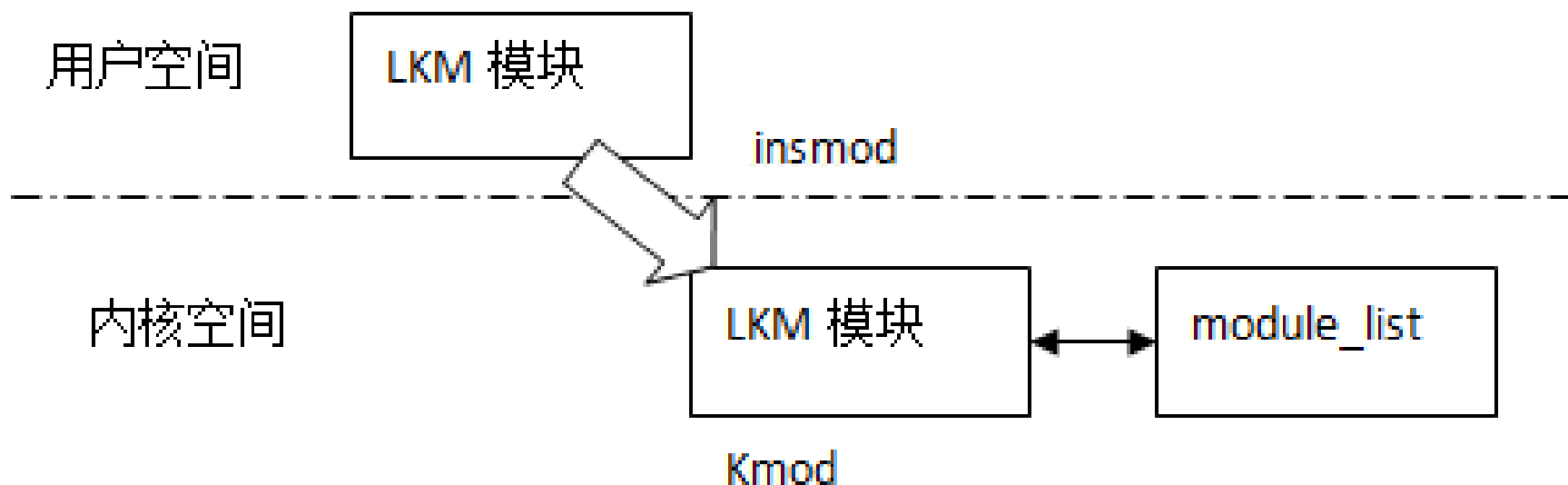


图5-11 LKM模块的加载

## 2. 模块的卸载

可以使用rmmod命令删除模块，这里有个特殊情况是请求加载模块在其使用计数为0时，会自动被系统删除。

用户空间

rmmod

内核空间

Kmod

deleted

LKM 模块

内存回收

清除关联

内核中其他部分还在使用的模块不能被卸载。

## 5.4.4工具集module-init-tools

在 Linux 2.6 中，工具 insmod 被重新设计并作为工具集 module-init-tools 中的一个程序，其通过系统调用 sys\_init\_module（可查看头文件 include/asm-generic/unistd.h）衔接了模块的版本检查，模块的加载等功能。Module-init-tools 是为 2.6 内核设计的运行在 Linux 用户空间的模块加卸载工具集，其包含的程序 rmmod 用于卸载当前内核中的模块。



| 名称     | 说明            | 使用方法示例                                                          |
|--------|---------------|-----------------------------------------------------------------|
| insmod | 装载模块到当前运行的内核中 | <code>#insmod [/full/path/module_name]<br/>[parameters]</code>  |
| rmmod  | 从当前运行的内核中卸载模块 | <code>#rmmod [-fw] module_name</code><br>-f:强制将该模块删除掉，不论是否正在被使用 |

|       |                                |                                                          |
|-------|--------------------------------|----------------------------------------------------------|
| lsmod | 显示当前内核已加装的模块信息，可以和 grep 指令结合使用 | <code>#lsmod</code><br>或者 <code>#lsmod   grep XXX</code> |
|-------|--------------------------------|----------------------------------------------------------|

|          |                              |                                                                                                                                                     |
|----------|------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------|
| modprobe | 利用 depmod 创建的依赖关系文件自动加载相关的模块 | <code>#modprobe [-lcfwr] module_name</code><br>-c:列出目前系统上面所有的模块<br>-l:列出目前在 /lib/modules/`uname -r` /kernel 当中的所有模块完整文件名<br>-f:强制加载该模块<br>-r:删除某个模块 |
|----------|------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------|

# 5.5 Part Five

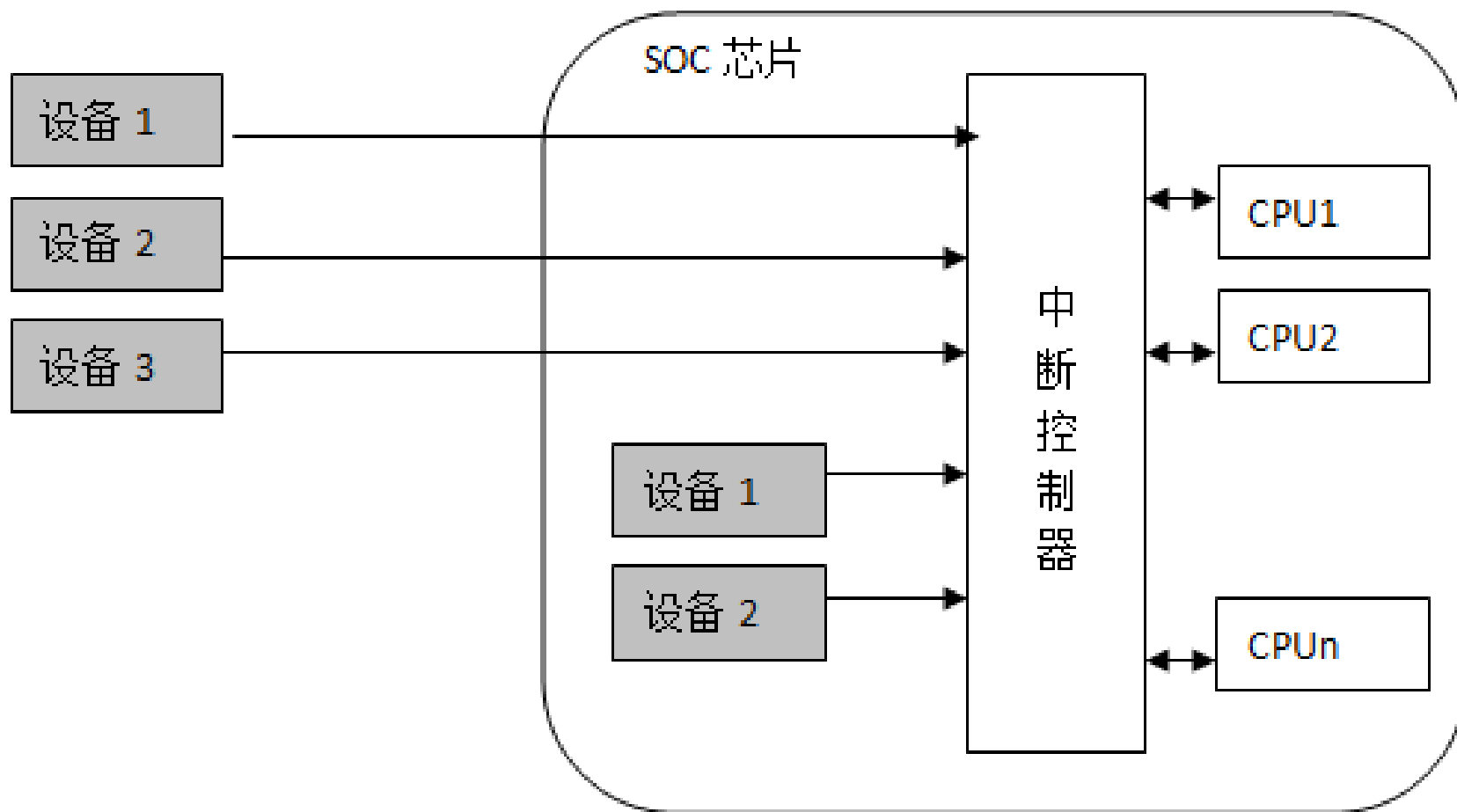
## ARM\_Linux中断管理

---

## 5.5.1 ARM\_Linux中断的一些基本概念

### 1. 设备、中断控制器和CPU

一个完整的设备中，与中断相关的硬件可以划分为3类，它们分别是：**设备**、**中断控制器**和**CPU本身**，



**设备：** 设备是发起中断的源，当设备需要请求某种服务的时候，它会发起一个硬件中断信号，通常，该信号会连接至中断控制器，由中断控制器做进一步的处理。

**中断控制器：** 中断控制器负责收集所有中断源发起的中断，ARM架构的soc，使用较多的中断控制器是VIC（Vector Interrupt Controller），进入多核时代以后，GIC（General Interrupt Controller）的应用也开始逐渐变多。

**CPU：** cpu是最终响应中断的部件，它通过对可编程中断控制器的编程操作，控制和管理者系统中的每个中断。当中断控制器最终判定一个中断可以被处理时，它会根据事先的设定，通知其中一个或者是某几个cpu对该中断进行处理，虽然中断控制器可以同时通知数个cpu对某一个中断进行处理，

## 2. IRQ编号

系统中每一个注册的中断源，都会分配一个唯一的编号用于识别该中断，称之为**IRQ**编号。**IRQ**编号贯穿在整个Linux的通用中断子系统中。在移动设备中，每个中断源的**IRQ**编号都会在arch 相关的一些头文件中，例如 `arch/xxx/mach-xxx/include/irqs.h`。驱动程序在请求中断服务时，它会使用**IRQ**编号注册该中断，中断发生时，`cpu`通常会从中断控制器中获取相关信息，然后计算出相应的**IRQ**编号，然后把该**IRQ**编号传递到相应的驱动程序中。

## 5.5.2 内核异常向量表的初始化

ARM-linux 内核启动时，首先运行的是 arch/arm/kernel/head.S，进行一些初始化工作，然后调用 main.c->start\_kernel() 函数，进而调用 trap\_init()（或者调用 early\_trap\_init() 函数）以及 init\_IRQ() 函数进行中断初始化，建立异常向量表。

```
asmlinkage void __init start_kernel(void)
{

 trap_init();

 early_irq_init();
 init_IRQ();

}
```

接着系统会建立异常向量表。首先会将ARM处理器异常中断处理程序的入口安装到各自对应的中断向量地址中。

在ARM V4及V4T以后的大部分处理器中，中断向量表的位置可以有两个位置：

一个是0x00000000，另一个是0xffff0000。

要说明的是Cortex-A8处理器支持通过设置协处理CP15的C12寄存器将异常向量表的首地址设置在任意地址。可以通过CP15协处理器c1寄存器中V位(bit[13])控制。

V位和中断向量表的对应关系如下：

V=0      ~      0x00000000~0x0000001C

V=1      ~      0xffff0000~0xffff001C

**early\_trap\_init**函数的主要功能就是将中断处理程序的入口拷贝到中断向量地址。其中

```
extern char __stubs_start[], __stubs_end[];
extern char __vectors_start[], __vectors_end[];
extern char __kuser_helper_start[], __kuser_helper_end[];
```

这三个变量是在汇编源文件中定义的，在源代码包里定义在**entry-armv.S**中。

**\_\_vectors\_start:**

```
swi SYS_ERROR0
b vector_und + stubs_offset
ldr pc, .LCvswi + stubs_offset
b vector_pabt + stubs_offset
b vector_dabt + stubs_offset
b vector_addrxcptn + stubs_offset
b vector_irq + stubs_offset
b vector_fiq + stubs_offset
.globl __vectors_end
```

**\_\_vectors\_end:**



这里要说明的是在采用了MMU内存管理单元后，异常向量表放在哪个具体物理地址已经不那么重要了，只需要将它映射到0xffff0000的虚拟地址即可。在中断前期处理函数中会根据IRQ产生时所处的模式来跳转到不同的中断处理流程中。

Init\_IRQ(void)函数是一个特定于体系结构的函数，对于ARM体系结构来说该函数定义如下：

```
void __init init_IRQ(void)
{
 int irq;
 for (irq = 0; irq < NR_IRQS; irq++)
 irq_desc[irq].status |= IRQ_NOREQUEST | IRQ_NOPROBE;

 init_arch_irq();
}
```

这个函数将irq\_desc[NR\_IRQS]结构数组各个元素的状态字段设置为IRQ\_NOREQUEST | IRQ\_NOPROBE，也就是未请求和未探测状态。然后调用特定机器平台的中断初始化init\_arch\_irq()函数。

## 5.5.3 Linux中断处理

### 1.中断申请并响应

ARM处理器的中断是由处理器内部或者外部的中断源产生，通过**IRQ或者FIQ中断请求**线传递给处理器。在ARM模式下中断可以配成IRQ模式或者FIQ模式。但是在Linux系统里面，所有的中断源都被配成了IRQ中断模式。

要想使设备的驱动程序能够产生中断，则首先需要调用**request\_irq()**来注册中断服务

**request\_irq()**函数的定义:

```
include/linux/interrupt.h
```

```
static inline int __must_check
```

```
request_irq(unsigned int irq, irq_handler_t handler,
unsigned long flags,
```

```
 const char *name, void *dev)
```

```
{
```

```
 return request_threaded_irq(irq, handler, NULL,
flags, name, dev);
```

```
}
```

在通过**request\_irq()**函数注册中断服务程序的时候，将会把设备中断处理程序添加进系统，使在中断发生的时候调用相应的中断处理程序。

内核用这个函数来完成分配中断线的工作，其主要参数说明如下：

**irq**，要注册的硬件中断号；

**handler**，向系统注册的中断处理函数，它是一个回调函数，

**irqflags**，中断类型标志，。

**devname**，一个声明的设备的ascii名字，与中断号相关联的名称，在/proc/interrupts文件中可以看到此名称。

**dev\_id**，I/O设备的私有数据字段，thread\_fn，由irq handler线程调用的函数，

## 2.保存现场

处理中断时要保存现场，然后才能处理中断，处理完之后还要把现场状态恢复后才能返回到被中断的地方继续执行。这里说明在指令跳转到中断向量的地方开始执行之前，由**CPU**自动完成了必要工作之后，每当中断控制器发出产生一个中断请求，则**CPU**总是到异常向量表的中断向量处取指令来执行。

### 3.中断处理

ARM Linux对中断的处理主要分为内核模式下的中断处理模式和用户模式下的中断处理模式。

**内核模式**下的中断处理，也就是调用**\_\_irq\_svc**例程，**\_\_irq\_svc**例程在文件**arch/arm/kernel/entry-armv.S**中定义，首先来看这个例程的定义：

```
__irq_svc:
 svc_entry
```

```
#ifdef CONFIG_PREEMPT
 get_thread_info tsk
 ldr r8, [tsk, #TI_PREEMPT] @ get preempt count
 add r7, r8, #1 @ increment it
 str r7, [tsk, #TI_PREEMPT]
#endif
```

***irq\_handler***

◦ ◦ ◦ ◦ ◦ ◦

用户模式下的中断处理流程。中断发生时，CPU处于用户模式下，则会调用\_\_irq\_usr例程，

```
.align 5
__irq_usr:
 usr_entry
 kuser_cmpxchg_check
 get_thread_info tsk
#ifdef CONFIG_PREEMPT
 ldr r8, [tsk, #TI_PREEMPT] @ get preempt count
 add r7, r8, #1 @ increment it
 str r7, [tsk, #TI_PREEMPT]
#endif
 irq_handler
#ifdef CONFIG_PREEMPT
 ldr r0, [tsk, #TI_PREEMPT]
 str r8, [tsk, #TI_PREEMPT]
 teq r0, r7
 ARM(strne r0, [r0, #-r0])
 THUMB(movne r0, #0)
 THUMB(strne r0, [r0])
#endif
#ifdef CONFIG_TRACE_IRQFLAGS
 bl trace_hardirqs_on
#endif
 mov why, #0
 b ret_to_user
UNWIND(.fnend)
ENDPROC(__irq_usr)
```

由该汇编代码可知，如果在用户模式下产生中断的话，在返回的时候，会根据需要进行进程调度，而如果中断发生在管理等内核模式下的话是不会进行进程调度的。



## 4.中断返回

中断返回在前文已经分析过，这里不再赘述。这里只补充说明一点：如果是从用户态中断进入的则先检查是否需要调度，然后返回。如果是从系统态中断进入的则直接返回。

5.5.4 内核版本2.6.38后的中断处理系统的一些改变----通用中断子系统（Generic irq）

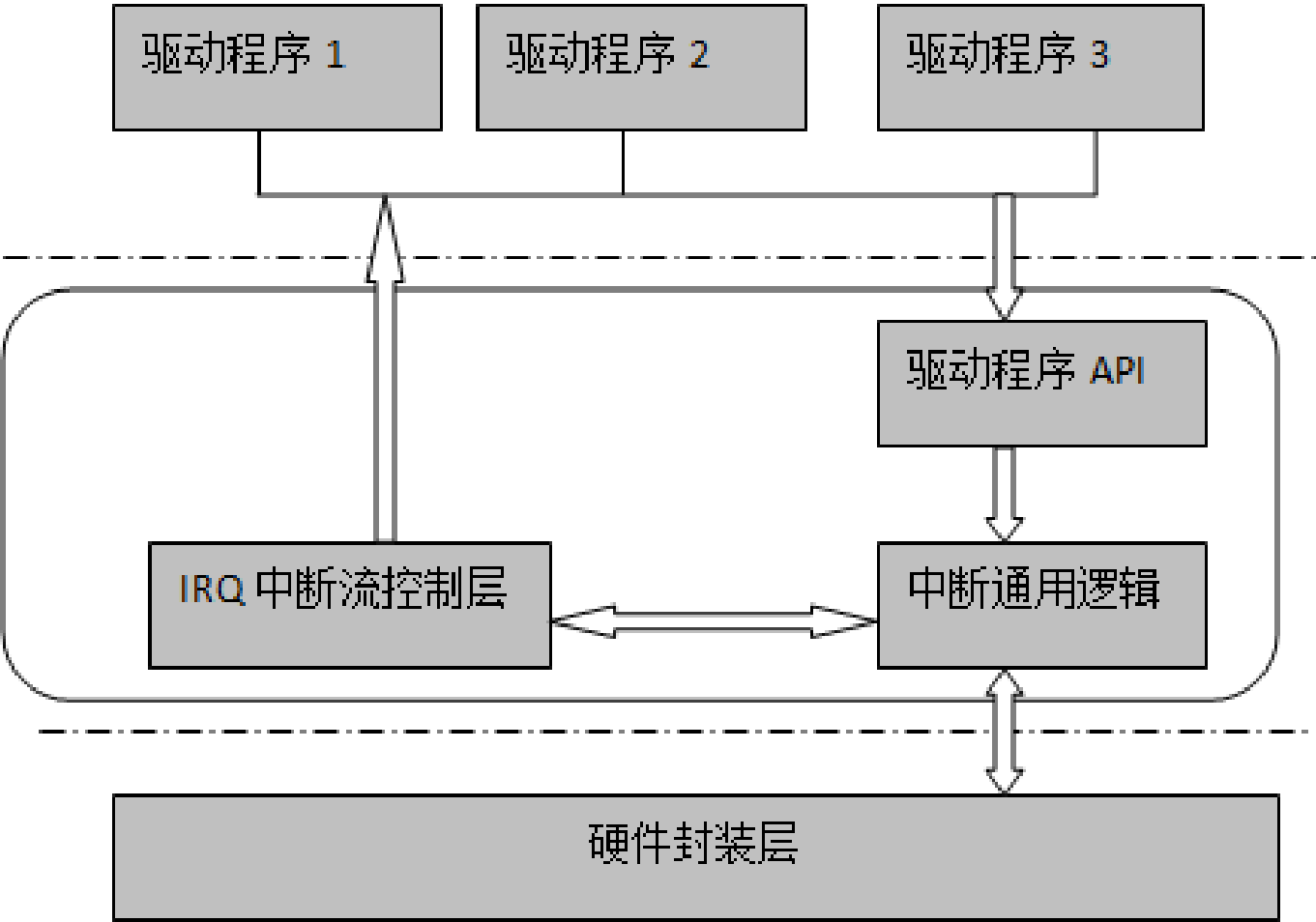


图5-14  
通用中  
断子系  
统的层  
次结构

**硬件封装层：** 它包含了体系架构相关的所有代码，包括中断控制器的抽象封装，**arch**相关的中断初始化，以及各个**IRQ**的相关数据结构的初始化工作，**cpu**的中断入口也会在**arch**相关的代码中实现。

**中断流控制层：** 所谓中断流控制是指合理并正确地处理连续发生的中断

**中断通用逻辑层：** 该层实现了对中断系统几个重要数据的管理，并提供了一系列的辅助管理函数。同时，该层还实现了中断线程的实现和管理，共享中断和嵌套中断的实现和管理，另外还提供了一些接口函数，它们将作为硬件封装层和中断流控层以及驱动程序**API**层之间的桥梁。

**驱动程序API：** 该部分向驱动程序提供了一系列的**API**，用于向系统申请/释放中断，打开/关闭中断，设置中断类型和中断唤醒系统的特性等操作。

# 5.6

Part Six

## ARM-Linux系统调用

---

Linux系统利用SWI指令来从用户空间进入内核空间。SWI指令用于产生软件中断，从而实现从用户模式到管理模式的变换，CPSR保存到管理模式的SPSR，执行转移到SWI向量。在其他模式下也可使用SWI指令，处理器同样地切换到管理模式。指令格式如下：

**SWI{cond} immmed\_24**

*Cond域：是可选的条件码，. immmed\_24域：范围从 0 到 224-1的表达式，（即0-16777215）， immmed\_24为软中断号（服务类型）。*

使用SWI指令时，通常使用以下两种方法进行传递参数，如：

(1) 指令中的24位立即数指定了用户请求的服务类型，参数通过通用寄存器传递。

**mov r0, #34 ;设置子功能号位34**  
**SWI 12 ;调用12号软中断**

(2) 指令中的24位立即数被忽略，用户请求的服务类型有寄存器R0的值决定，参数通过其他的通用寄存器传递。如：

**mov r0, #12 ;调用12号软中断**  
**mov r1, #34 ;设置子功能号位34**  
**SWI 0**