ULK Linux Kernel 2.6.11 Note 笔记

基于 ULK Linux Kernel 2.6.11 学习

ElegantLaTeX Program

Victory won't come to us unless we go to it.

作者：Miao

时间：July 23, 2023

邮箱: chenmiao.ku@gmail.com

版本: 0.10

目　　录

第 1 章 绪论

1.1 操作系统基本概念

任何计算机系统都包含一个名为操作系统的基本程序集合。在这个集合中，最重要的程序称为内核 (kernel)。启动后，内核中包含了系统运行所必不可少的很多核心过程 (procedure)，和其他一些不太重要的实用程序。

系统根本的样子和能力还是由内核决定，内核也为操作系统中所有事情提供了主要功能，并决定高层软件的很多特性。

操作系统必须完成两个主要目标：

1) 与硬件部分交互，为包含在硬件平台上的所有底层可编程部件提供服务
2) 为运行在计算机系统上的应用程序提供执行环境

类 *Unix* 系统把与计算机物理组织相关的所有底层细节都对用户运行的程序隐藏起来，硬件为 *CPU* 引入了至少两种不同的执行模式：非特权模式 *(*用户态 *(User Mode)* 和特权模式 *(Kernel Mode))*。

1.2 多用户系统

多用户系统 *(multiuser system)* 就是一台能并发和独立地执行分别属于两个或多个用户的若干应用程序的计算机。

并发 (concurrently) 意味着几个应用程序能够同时处于活动状态并竞争各种资源。独立 (independently) 意味着每个应用程序能够执行自己的任务，而无需考虑其他用户的应用程序在做什么。

多用户操作系统必须包含：

1) 核实用户身份的认证机制
2) 防止有错误的用户程序妨碍其他应用程序在系统中运行的保护机制
3) 防止有恶意的用户程序干涉或窥视其他用户的活动的保护机制
4) 限制分配给每个用户的资源数的记账机制

1.3 用户和组

操作系统必须保证用户空间的私有部分仅仅对于其拥有者是可见的。

所有的用户由一个唯一的数字来表示，这个数字叫用户标识符 (User ID, UID)。

为了和其他用户有选择地共享资料，每个用户是一个或多个用户组的一名成员，组由唯一的用户组标识符 (user group ID) 标识。每个文件也恰好与一个组相对应。

任何类 *Unix* 操作系统都有一个特殊的用户，*root(超级用户 (superuser))*。系统管理员能够通过 root 账号登陆，值得一提的是：*root* 几乎无所不能，其能访问系统中的每一个文件，能干涉每一个正在执行的用户程序。

1.4 进程

所有的操作系统都有一种基本的抽象：进程 *(process)*。一个进程可以定义为：”程序运行时的一个实例”，或者一个运行程序的”执行上下文”。

传统的操作系统中，一个进程在地址空间中 *(address space)* 执行一个单独的指令序列。现代操作系统允许具有多个执行流的进程，也就是在相同的地址空间可执行多个指令序列。

允许进程并发活动的系统称为多道程序系统 *(multiprogramming)* 或多处理系统 *(multiprocessing)*。值得注意的是：几个进程能够并发地执行同一个程序，而同一个进程能顺序的执行几个程序。

调度程序 (scheduler) 的部分决定哪个进程能执行，一些操作系统只允许有非抢占式 (nonpreemptable) 进程，也就是说，只有当进程自愿放弃时，调度程序才能被调用。但是，多用户系统中的进程必须是抢占式的 *(preemptable)*。

1.5 内核体系结构

大部分 Unix 内核都是单块结构：每一个内核层都被集成到整个内核程序中，并代表当前进程在内核态下运行。

微内核 *(microkernel)* 操作系统只需要内核有一个很小的函数集 *(*几个同步原语[1]，一个简单的调度程序和进程间通信*)*。

Linux 内核提供了模块 (module) 用于达到微内核理论上的很多优点且不影响性能。模块是一个目标文件，其代码可以在运行时链接到内核或从内核解除链接。这种目标代码通常由一组函数组成，用来实现文件系统、驱动程序或其他内核上层功能。

使用模块的主要优点：

1) 模块化方法

   任何模块都能在运行时被链接或解除链接。这要求程序员提出良定义的软件接口以访问由模块处理的数据结构

2) 平台无关性

   即使模块依赖于某些特殊的硬件特点，但它不依赖于某个固定的硬件平台

[1]原语（primitive）是计算机科学中的一个概念，它指的是一组基本的操作或指令，可以直接在计算机硬件上执行。原语通常是由计算机硬件提供的，用于支持高级编程语言或操作系统的功能

3) 节省内存使用

当需要模块时，就链接；不需要时，则解除

4) 无性能损失

模块的目标代码一旦被链接进内核，起作用与静态链接的内核的目标代码完全对等。因此无需显式的进行消息传递[1]

1.6 Unix 文件系统概述

1.6.1 文件

Unix 文件是以字节序列组成的信息载体 (container)，内核不解释文件的内容。

从用户的观点来看，文件被组织在一个树结构的命名空间内：

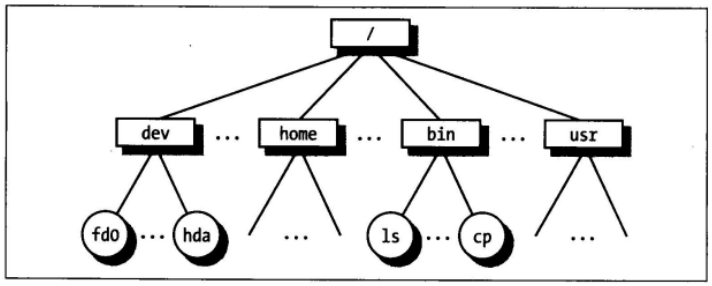

图 1.1: 目录树结构

除叶节点外，所有节点都表示目录名。目录节点包含它下面文件及目录的所有信息。

Unix 每个进程都有一个当前工作目录，属于进程执行上下文 (execution context)，标识出进程所用的当前目录。

路径名 (pathname) 由斜杠及一列指向文件的目录名交替组成。如果第一个字符是斜杠，那么就是所谓的绝对路径；否则就是所谓的相对路径。

当标识文件名时，用符号"." 和".." 分别标识当前工作目录和父目录。

1.6.2 硬链接和软连接

包含在目录中的文件名就是一个文件的硬链接 (hard link)，或简称连接 (link)。

使用 Unix 命令：

```
1 $ ln P1 P2
```

[1]模块被链接或解除时，都有一定的性能下降。但是在微内核中也是如此

用来创建一个新的硬链接，即为由路径 P1 标识的文件创建一个路径名为 P2 的硬链接。

硬链接有两方面的限制：

1) 不允许给目录创建硬链接，这可能使得目录树编程环形图从而无法通过名字定位一个文件
2) 只有在同一文件系统中的文件之间才能创建链接。

为了克服限制，引入软链接 *(soft link)[*也称符号链接 *(symbolic link)]*，符号链接是短文件，这些文件包含另一个文件的任意一个路径名。

值得注意的是：路径名可以指向位于任意一个文件系统的任意文件或目录 *(*哪怕它不存在*)*。

Unix 命令：

```
1 $ ln -s P1 P2
```

创建一个路径名为 P2 的新软连接，P2 指向路径名 P1。当执行命令时，文件系统抽取 P2 的目录部分，并在此创建 P2 的符号链接属性的新项。因此，任何对 P2 的引用都可以自动被转换为指向 P1 的引用。

1.6.3 文件类型

Unix 命令文件可以是以下类型:

- 普通文件 (regular file)
- 目录
- 符号链接
- 面向块的设备文件 (block-oriented device file)
- 面向字符的设备文件 (character-oriented device file)
- 管道 (pipe) 和命名管道 (named pipe)(也叫 FIFO)
- 套接字 (socket)

1.6.4 文件描述符与索引节点

除了设备文件和特殊文件系统外，每个文件都由字符序列组成。文件内容不包括任何控制信息，如文件长度或文件结束符 *(end-of-file, EOF)*。

文件系统处理文件需要的所有信息都包含在一个名为索引节点 *(inode)* 的数据结构中，文件系统用索引节点来标识文件。

索引节点 (inode) 至少包括：

- 文件类型
- 与文件相关的硬链接个数

- 以字节为单位的文件长度
- 设备标识符 (即包含文件的设备的标识符)
- 文件系统中标识的索引节点号
- 文件拥有者的 UID
- 文件的用户组 ID
- 几个时间戳 (改变时间、最后访问时间、最后修改时间)
- 访问权限和文件模式

1.6.5 访问权限和文件模式

文件的潜在用户分为三种类型：

- 文件所有者
- 同组用户 (不含所有者)
- 其他用户

同时，拥有三种类型的访问权限————读、写以及执行。因此就有九种组合不同的二进制来标记，还有三种额外的标记

- suid

  进程执行一个文件时通常保持进程拥有者的 UID，若设置 suid，进程就可以获取该文件拥有者的 UID

- sgid

  进程执行一个文件时保持进程组的用户组 ID，若设置 sgid，进程就可以获得该文件用户组 ID

- sticky

  设置 sticky 标志位相当于向内核发出请求，当程序结束后仍保留在内存[1]

1.6.6 文件操作的系统调用

当用户访问一个普通文件或目录文件的内容时，实际上访问的是在硬件块设备上的一些数据。

打开文件

```
1 /**
  * @param path 表示被打开文件的路径
3 * @param flag 指定文件打开的方式，也可以创建一个不存在的文件
  * @param mode 指定新创建文件的访问权限
5 */
```

[1]该标记已经过时，已被其他方法取代

```
fd = open(path, flag, mode)
```

该系统调用返回文件描述符 (file descriptor) 的标识符。一个打开文件对象包括：

- 文件操作的一些数据结构；表示文件当前位置的 offset 字段等等
- 进程可以调用的一些内核函数指针，由参数 flag 决定

访问打开的文件

对于 *Unix* 普通文件，可以顺序或随机访问；但设备文件和命名管道文件一般只能顺序访问。

内核把文件指针存放在打开文件对象中，也就是，当前位置就是下一次进行读或写的位置。为了修改文件指针的值，必须显式调用 lseek() 系统调用。

```
/**
 * @param fd 表示打开文件的文件描述符
 * @param offset 指定一个有符号整数，用于计算文件指针的新位置
 * @param whence 指定文件指针新位置的计算方式：offset加0，表示从文件头移动；offset加文件指针当前位置，表示从当前移动等
 */
newoffset = lseek(fd, offset, whence);

/**
 * @param fd 表示打开文件的文件描述符
 * @param buf 指定在进程地址空间中缓冲区的地址，所读的数据放在该缓冲区
 * @param count 表示所读的字节数
 */
nread = read(fd, buf, count);
```

返回值 nread 的值就是实际所读的字节数。

关闭文件

当进程无需访问文件时，则可以关闭文件资源

```
res = close(fd)
```

更名与删除文件

重新命名和删除并不需要进程打开文件

```
1 res = rename(oldpath, newpath);
```

改变文件链接的名字：

```
1 res = unlink(pathname)
```

文件真正的被删除时：链接数为 *0* 才会被删除。

1.7 Unix 内核概述

Unix 内核提供了应用可以运行的执行环境。因此，内核需要实现服务与对应接口。

1.7.1 进程/内核模式

程序在用户态下执行时，不能直接访问内核数据结构或内核的程序。CPU 都为从用户态到内核态的转换提供了特殊的指令。程序大部分时间都在用户态，只有需要内核提供特殊服务时会切换到内核态。

进程是动态的实体，在系统内通常只有有限的生存期。创建、撤销及同步现有进程的任务都委托给内核中的一组例程。内核本身不是进程，而是进程的管理者。

规定：请求内核服务的进程使用所谓系统调用 *(system call)* 的特殊编程机制。每个系统调用都设置了一组识别进程请求的参数，然后执行用户–内核态转换。

Unix 系统包括几个内核线程 (kernel thread)：

- 以内核态运行在内核地址空间
- 不与用户直接交互
- 在系统启动时创建，一直活跃到系统关闭

在单系统中，任何时候只有一个进程在运行，要么处于用户态，要么处于内核态。

Unix 中，可以通过以下方式激活内核例程：

- 调用系统调用
- 进程发出异常 (exception) 信号
- 外围设备发出中断 (interrupt) 信号通知一个事件的发生。每个中断信号都是由中断处理程序 (interrupt handler) 处理的
- 内核线程被执行

1.7.2 进程实现

为了让内核管理进程，每个进程由一个进程描述符 (process descriptor) 表示，该描述符包含有关进程当前状态的信息。

内核暂停一个进程的执行，需要保存其相关信息 (将相关寄存器保存在进程描述符中)：

- 程序计数器 (PC) 和栈指针 (SP)
- 通用寄存器
- 浮点寄存器
- 状态寄存器 (处理器状态字，Processor Status Word)
- 用于跟踪进程对 RAM 访问的内存管理寄存器

由于保存 PC 的原因，进程会从它停止的地方恢复执行。Unix 内核可以区分很多等待状态，由进程描述符队列实现。

1.7.3 可重入内核

所有的 Unix 内核都是可重入的[1](reentrant)，这意味着若干进程可以同时在内核态下执行。

提供可重入的一种方式是编写函数，以便这些函数只能修改局部变量，而不能修改全局数据结构，这样的函数叫做可重入函数。可重入内核可以包含非重入函数，并且利用锁机制保证一次只有一个进程执行一个非重入函数。

如果一个硬件中断发生，可重入内核能挂起当前正在执行的进程，即使这个进程处于内核态。

内核控制路径[2](kernel control path) 表示内核处理系统调用、异常或中断执行的指令序列。

最简单的情况下，CPU 从第一条指令到最后一条指令顺序地执行内核控制路径，但当下述事件之一发生，CPU 交错执行内核控制路径：

- 进程调用一个系统调用，但对应的内核控制路径无法立即满足，且投入一个新的进程运行。因此进程发生切换，则两条控制路径交替执行
- CPU 检测到一个异常，第一个控制路径挂起，执行合适的异常处理。处理结束后，继续执行控制路径
- CPU 执行中断的控制路径。第一个控制路径执行时，CPU 执行另一个控制路径来处理中断，终止后恢复执行第一个控制路径。

[1]具体来说，可重入函数是指在多个线程同时调用时，不会出现竞争条件或数据不一致的情况。这意味着函数的执行不依赖于外部状态，并且可以在任何时间点被中断和恢复，而不会影响函数的正确性。

[2]指操作系统内核在执行期间所经过的代码路径。它代表了操作系统的执行流程，包括中断处理、系统调用、任务切换等。内核控制路径通常是由硬件中断或软件触发的事件引发的。当一个事件发生时，例如外部设备的中断请求或用户程序的系统调用，操作系统内核会通过相应的中断处理程序或系统调用处理程序来响应和处理这个事件。

- 在支持抢占调度的内核中，一个更高优先级的进程加入就绪队列中，中断开始，优先执行高优先级。执行完毕后，恢复刚开始的控制路径

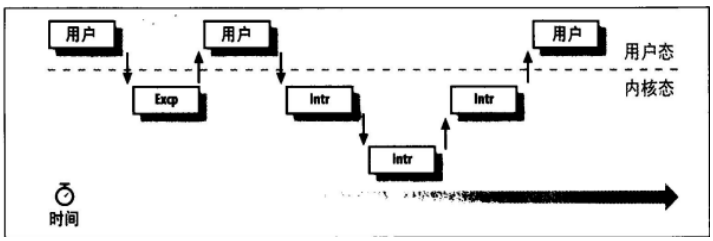

图 1.2: 内核控制路径的交错执行

上图考虑了以下三种 CPU 状态：

- 在用户态下运行进程 (User)
- 运行异常处理或系统调用 (Excp)
- 运行中断处理 (Intr)

1.7.4 进程地址空间

每个进程运行在它的私有地址空间。进程 *(用户态)* 涉及到私有栈、数据区和代码区。进程 *(内核态)* 涉及数据区和代码区，使用另外的私有栈。

因为内核可重入，因此内核控制路径引用自己的私有内核栈。虽然看上去每个进程访问自己的私有地址空间，但是也有共享部分地址空间。

Linux 支持 mmap() 系统调用，允许存放在块设备上的文件或信息的一部分映射到进程的部分地址空间。

1.7.5 同步和临界区

实现可重入内核需要利用同步机制：如果内核控制路径对某个内核数据结构进行操作时被挂起，那么，其他内核控制路径就不应该对该数据结构再操作。除非被设置成一致性[1] *(consistent)* 状态。否则会破坏所存储的信息。

当某个计算结果取决于如何调度两个或多个进程时，相关代码就是不正确的。也就是存在一种竞争条件 (race condition)

一般地，对全局变量的安全访问通过原子操作 (atomic operation) 来保证。临界区[2](critical region) 就是这样的一段代码，进入这段代码的进程必须完成，之后另一个进程才能进入。

[1]一致性状态（*Consistency State*）是指在分布式系统中，所有副本或节点之间的数据副本保持一致的状态。

[2]临界区（*Critical Section*）是指在并发编程中，一段代码或一段程序片段，在同一时间只能被一个线程执行的区域。

非抢占式内核

在以前，大多数传统 Unix 内核都是非抢占式的。因此，进程不会被轻易挂起，也不能被随意代替。(单处理器上) 中断或异常处理不能修改所有内核数据结构，内核对它们的访问都是安全的。

内核态的进程需要自愿放弃 CPU，但是，必须确保所有的数据结构都处于一致性状态。

非抢占式在多处理器系统上时低效的，因为运行在不同 CPU 上的内核控制路径本可以并发访问相同的数据结构。

禁止中断

(单处理器系统的另一种同步机制) 进入临界区之前禁止所有的硬件中断，离开时再启动中断。尽管机制简单，但是不是最佳的。如果临界区过大，那么在相对长的时间内持续禁止中断会导致硬件活动处于冻结。

信号量

信号量 (semaphore) 是一种广泛的机制，其仅仅是一个与数据结构相关的计数器。所有内核线程在试图访问该数据结构之前，需要检查该信号量。信号量的组成如下：

- 一个整数变量
- 一个等待进程的链表
- 两个原子方法: down() 和 up()

down() 方法对信号量减一，如果新值小于 0，则就把正在允许的进程加入信号量链表，然后阻塞该进程 (即调用调度程序)。

up() 方法对信号量加一，如果大于等于 0，则激活信号量链表中的一个或多个进程。

每个要保护的数据结构都有自己的信号量，其初始值为 1。当内核控制路径希望访问时，调用 down()，信号量为非负数允许访问该数据结构。

自旋锁

多处理器中，信号量并不是解决同步的最佳方案。系统不允许在不同 *CPU* 上允许的内核控制路径同时访问某些内核数据结构，这种情况下，修改数据结构所需的事件比较短，而信号量需要加入信号量链表挂起并唤醒，是更为耗时的。

因此，多处理器系统使用自旋锁 (spin lock)。自旋锁与信号量十分相似，但没有进程链表，当进程发现锁被另一个进程使用时，自身就不停” 旋转”。

注意：自旋锁在单处理器上是无效的，因为没有机会释放该自旋锁。

避免死锁

与其他控制路径同步的进程或内核控制路径很容易进入死锁 *(deadlock)* 状态。

也就是说：进程 p1 获得访问数据结构 a 的权限，进程 p2 获得访问 b 的权限，而 p1 在等待 b，p2 等待 a。

1.7.6 信号和进程间通信

Unix 信号 (signal) 提供了把系统事件报告给进程的一种机制。每种事件都有自己的信号编号，通常用符号常量表示。有两种系统事件：

- 异步通告

  例如，用户在中断按下中断键，即向前台发送中断信号 SIGINT
- 同步错误或异常

  例如，进程访问内存地址非法，内核向进程发送 SIGSEGV 信号

POSIX 标准定义了大约 *20* 种不同的信号，其中两种是用户自定义的，可以当作进程通信和同步原语机制。一般地，进程可以对两种对接方式信号做出反应：

- 忽略该信号
- 异步地执行一个指定的过程 (信号处理程序)

若进程不指定选择何种方式，内核根据编号执行一个默认动作：

- 终止进程
- 将执行上下文和进程地址空间的内容写入一个文件 (核心转储，core dump)，并终止进程
- 忽略信号
- 挂起进程
- 如果进程曾被暂停，则恢复执行

1.7.7 进程管理

Unix 在进程与执行的程序中做了一个清晰的划分。fork() 和 _exit() 系统调用分别用来创建和终止进程。exec() 类系统调用则是装入新程序。

执行 fork() 的进程是父进程，产生的是子进程。父子进程能够互相找到对方，因为每个描述进程的数据结构都包含两个指针。

当前实现 fork() 的技术是依赖硬件分页单元的内核采用写时复制 (Copy-On-Write) 技术，即把页的复制延迟到最后 (直到父子需要时才写进)。

_exit() 调用终止进程。通过释放进程所拥有的资源并向父进程发送 SIGCHILD 信号。

僵尸进程 (zombie process)

wait() 系统调用允许进程等待，直到一个子进程结束，其返回已终止进程的进程标识符 (Process ID，PID)

引入僵尸进程的特殊状态是为了表示终止的进程：父进程未执行 *wait()* 调用前，其子进程已经终止 *()*。

系统调用处理程序从进程描述符字段中获取有关资源的数据，一旦获取，就可以释放进程描述符。

若父进程终止而没有发出 wait() 调用，这就会导致该进程一直停留在内存中，无法被使用也无法被清除。因此，可以使用名为 init 的特殊系统进程 (在系统初始化时被创建)。当一个进程终止时，内核改变其现有子进程的进程描述符指针，将其称为 *init* 的子进程，*init* 监视所有子进程的执行，并发布 *wait()*，其作用就是为了除掉所有的僵尸进程。

进程组和登录会话

现代 Unix 系统引入了进程组 (process group) 的概念，以表示一种” 作业 (job)” 的抽象：

```
1 $ ls | sort | more
```

Shell 中为这三个相应的进程创建了一个新的组，就好像它们时一个单独的实体 (作业)。每个进程描述符包括一个包含进程组 ID 的字段。每个进程组可以有一个领头进程 (即 PID 与进程组 ID 一致)。

现代 Unix 系统也引入了登录会话 (login session)。非正式的说，一个登录会话包含在指定中断已经开始工作的进程的所有后代进程。

通常，登录会话时 shell 为用户创建的第一条命令，进程组中的所有进程必须在同一登录会话中。

1.8 内存管理

1.8.1 虚拟内存

虚拟内存 *(virtual memory)* 作为一种逻辑层，处于应用程序的内存请求与硬件内存管理单元 *(Memory Management Uint，MMU)* 之间。其有很多用途和优点：

- 进程可以并发地执行
- 应用所需内存大于可用物理内存时也能运行
- 程序只有部分代码装入内存时进程可以执行

- 允许每个进程访问可用物理内存的子集内存映像
- 程序是可重定位的，也就是可以把程序放在物理内存的任何地方
- 程序员可以编写与机器无关的代码，不必关心有关物理内存的组织结构

虚拟内存子系统的主要成分是虚拟地址空间 *(virtual address space)*。进程所用的一组内存地址不同于物理内存地址。当使用虚拟地址[1]时，内核和 *MMU* 协同定位其在内存中的实际物理地址。

1.8.2 随机访问存储器 (RAM) 的使用

所有 Unix 系统都将 RAM 毫无疑义地划分为两部分，其中若干兆字节用于存放内存映像 (内核代码和内核静态数据结构)。其余部分通常由虚拟内存系统来处理：

- 满足内核对缓冲区，描述符及其他动态内核数据结构的请求
- 满足进程对一般内存区的请求及文件内存映射的请求
- 借助于高速缓存从磁盘及其他缓冲设备获得较好的性能

每种请求类型都是重要的。但需要做出平衡，当可用内存达到临界阈值时，可用调用页框回收 (page-frame-reclaiming) 算法释放其他内存。

虚拟内存必须解决的一个问题便是内存碎片。

1.8.3 内核内存分配器

内核内存分配器 *(Kernel Memory Allocator，KMA)* 是一个子系统，其试图满足系统中所有部分对内存的请求。一个好的 KMA 应该具有：

- 必须快。这是最重要的属性，因为由所有的内核子系统 (包括中断处理) 调用
- 必须把内存的浪费减少到最少
- 必须努力减轻内核的碎片 (fragmentation) 问题
- 必须能与其他内存管理子系统合作，以便借用和释放页框

基于各种不同的算法，已经由以下的 KMA：

- 资源图分配算法 (allocator)
- 2 的幂次方空闲链表
- McKisick-Karels 分配算法
- 伙伴 (Buddy) 算法
- Mach 的区域 (Zone) 分配算法
- Dynix 分配算法
- Solaris 的 Slab 分配算法

[1]在不同体系结构中叫法不一，Intel 中叫做逻辑地址

1.8.4 进程虚拟地址空间处理

进程的虚拟地址空间包括了进程可用引入的所有虚拟内存地址。内核通常用一组内存区描述符描述进程虚拟地址空间。

- 程序的可执行代码
- 程序的初始化数据
- 程序的未初始化数据
- 初始程序站
- 所需共享库的可执行代码和数据
- 堆

所有的现代 Unix 系统都采用了所谓请求调页 (demand paging) 的内存分配策略。有了请求调页，进程可以在它的页还没在内存时就开始执行。

当进程访问一个不存在的页，MMU 产生一个异常，异常处理程序找到受影响的内存区，分配一个空闲页。

1.8.5 高速缓存

物理内存的一大优势就是用作磁盘和其他块设备的高速缓存。

磁盘是非常慢的，因此，其通常是影响系统性能的瓶颈。早期的一种解决方案：尽可能地推迟写磁盘的时间，因此，从磁盘读入内存中的数据即使任何进程都不再使用它们，也需要继续留在 *RAM* 中。

新进程请求从磁盘读或写的数据，就是被撤销进程曾拥有的数据。当一个进程请求访问磁盘，内核会首先检查进程请求的数据是否在缓存中，如果缓存命中，则先为进程请求提供服务。

sync() 系统调用把所有” 脏” 的缓冲区 (即缓冲区内容与对应磁盘块内容不一致) 写入磁盘来强制磁盘同步。

1.8.6 设备驱动程序

内核通过设备驱动程序 (device driver) 与 I/O 设备交互。设备驱动程序包含在内核中，由控制一个或多个设备的数据结构和函数组成，包括硬盘、键鼠、监视器、网络接口及 *SCSI* 总线上的设备。其具有以下优点:

- 可以把特定设备的代码封装在特定的模块中
- 可以在不了解内核源码只知道接口规范的情况下，增加新设备
- 以统一的方式对待所有设备，并通过相同的接口访问
- 可以把设备驱动程序写成模块，并动态进行载入

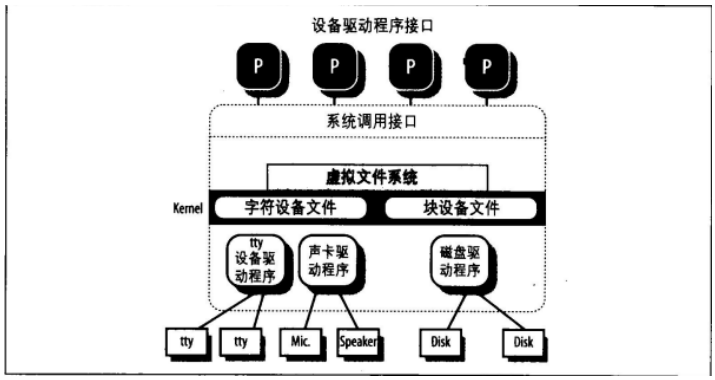

图 1.3: 设备驱动程序接口

第 2 章 内存寻址

2.1 内存地址

在使用 80x86 微处理器时，我们必须区分以下三种不同的地址：

- 逻辑地址 (logical address)

  包含在机器语言指令中用来指定一个操作数或一条指令的地址。每一个逻辑地址都由一个段 *(segment)* 和偏移量 *(offset* 或 *displacement)* 组成，偏移量指明了从段开始的地方到实际地址之间的距离。

- 线性地址 (linear address)(也称虚拟地址 virtual address)

  是一个 32 位无符号整数，可以用来表示高达 4GB 的地址。

- 物理地址 (physical address)

  用于内存芯片级内存单元寻址。从微处理器的地址引脚发送到内存总线上的电信号相对应。

内存控制单元 *(MMU)* 通过分段单元 *(segmentation unit)* 的硬件电路把一个逻辑地址转换成线性地址，然后分页单元 *(paging unit)* 的硬件电路把线性地址转化为物理地址。

在多处理器中，所有 CPU 都共享同一内存：这意味着 RAM 可以由独立的 CPU 并发访问。但因为 RAM 上的读写必须串行执行，因此需要内存仲裁器 (memory arbiter) 的硬件电路插在总线和 RAM 芯片之间。

内存仲裁器的作用是：如果某个 *RAM* 空闲，就准予 *CPU* 访问。若 *RAM* 忙于另一个 *CPU* 提出的请求服务，就延迟这个 *CPU* 的访问

2.2 硬件中的分段

从 80286 模型开始，Intel 处理器以实模式[1](real mode) 和保护模式 (protected mode) 执行地址转换。

[1]实模式（*Real Mode*）是 *x86* 体系结构中的一种工作模式，它是早期 *x86* 处理器的默认工作模式。在实模式下，处理器以 *16* 位的方式进行操作，可以直接访问 *1MB* 的物理内存。在实模式下，内存寻址是通过段地址和偏移地址的组合来实现的。段地址由段寄存器（如 *CS*、*DS*、*ES* 等）保存，偏移地址由指令中的操作数给出。通过将段地址左移 *4* 位后与偏移地址相加，可以计算出实际的物理地址

2.2.1 段选择符和段寄存器

一个逻辑地址由一个段标识符 *(16 位长的字段，段选择符 (Segment Selector))* 和一个指定段内相对地址的偏移量 *(32 位长的字段)* 组成。

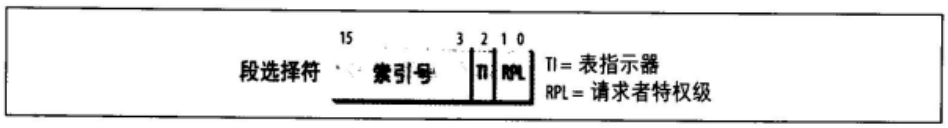

图 2.1: 段选择符格式

为了方便段选择符，段寄存器用于存放段选择符。其拥有六个寄存器：

cs	代码段寄存器，指向包含程序指令的段
ss	栈段寄存器，指向包含当前程序栈的段
ds	数据段寄存器，指向包含静态数据或者全局数据段
es, fs, gs	一般用途，可以指向任意数据段

cs 寄存器：包含一个两位的字段，用以指明 *CPU* 的当前特权级[1]*(Current Privilege Level，CPL)*。

2.2.2 段描述符

每个段由一个 8 字节的段描述符 (Segment Descriptor) 表示，描述了段的特征。

段描述符放在全局描述符表 (Global Descriptor Table，GDT) 或局部描述符表 (Local Descriptor Table，LDT) 中。

通常只定义一个 GDT，每个进程除了存放在 GDT 中的段之外如果还需附加，就可以使用 LDT。GDT 在主存中的地址和大小存放在 gdtr 控制寄存器中，LDT 地址和大小存放在 ldtr 控制寄存器中。

有几种不同类型的段以及对应的段描述符：

- 代码段描述符

  表示这个段描述符表示代码段，S 标志置 1
- 数据段描述符

  表示这个段描述符表示数据段，S 标志置 1
- 任务状态段描述符 (TSSD)

  表示这个段描述符表示任务状态段 (Task State Segment, TSS)，也就是该段用于保存处理器寄存器的内容。*只能出现在 GDT 中，根据相应进程是否在 CPU 上，其 Type 字段值为 11 或 9，S 标志置零*

[1]*值为 0 表示最高优先级，值为 3 为最低。Linux 只用 0 和 3 表示内核态和用户态。*

表 2.1: 段描述符字段

字段名	描述
Base	包含段的首字节的线性地址
G	粒度标志；若清零，则段大小以字节为单位，否则以 4096 字节倍数计
Limit	存放段中最后一个内存单元的偏移量，从而决定段的长度。
S	系统标志；置零表示系统段，否则是普通代码段或数据段
Type	描述了段的类型特征和存取权限
DPL	描述符特权级 (Dsecriptor Privilege Level) 字段；用于限制这个段的存取。其表示为访问这个段要求的 CPU 最小优先级。若 DPL 设为 0 的段只能当 CPL 为 0 时 (即内核态) 可访问
P	Segment-Present 标志；等于 0 表示段当前不在主存中 *Linux 总是把这个标志设置为 1，因为其从不把整个段交换到磁盘*
D 或 B	取决于是代码段还是数据段。D 或 B 的含义在两种情况下略有区别 如果偏移量 32 位则置 1，否则清零

- 局部描述符表描述符 (LDTD)

  表示这个段描述符包含一个 LDT 段，其只出现在 GDT 中。相应的 Type 字段值为 2，S 标志置 0

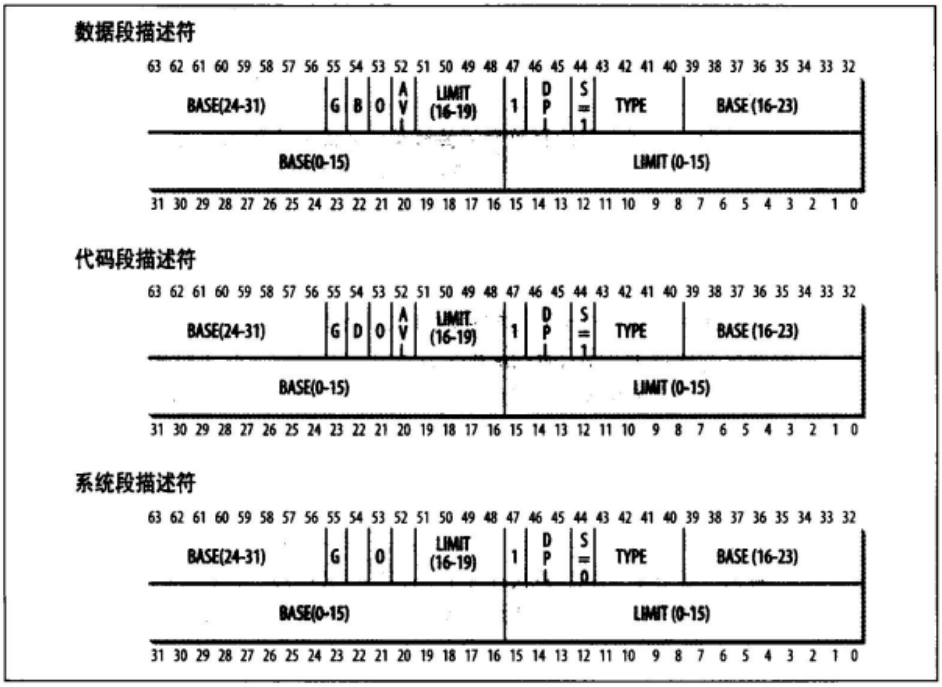

图 2.2: 段描述符格式

2.2.3 快速访问段描述符

逻辑地址由 16 位段选择符和 32 位偏移量组成，段寄存器仅仅存放段选择符。

为了加速逻辑地址 → 线性地址，80x86 处理器提供了附加的非编程的寄存器，供 6 个可编程的段寄存器使用。每一个非编程的寄存器含有八个字节的段描述符，由对应的段寄存器中的段选择描述符来指定。

每当一个段选择符被装入段寄存器，相应的段描述符就由内存装入对应的非编程 CPU 寄存器。这时，针对该段的逻辑地址转换就可以仅访问该非编程寄存器即可 (除非段寄存器内容发生更改)。

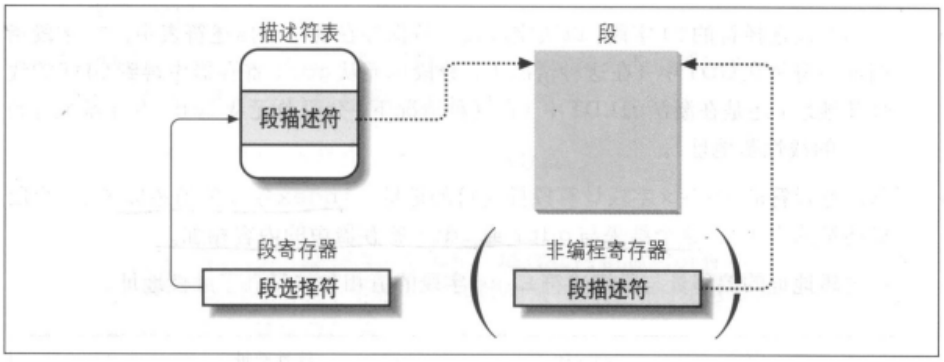

图 2.3: 段选择符和段描述符

表 2.2: 段选择符字段

字段名	描述
index	指定了描述放在 GDT/LDT 中对应的段描述符入口
TI	TI((Table Indicator) 标志，指明段描述符是在 GDT(TI = 0) 中或 LDT(TI = 1) 中)
RPL	请求者特权级，当相应的段选择符装入到 cs 寄存器中时指示出 CPU 当前的特权级还可以用于访问数据段时选择地削弱特权级

由于段描述符是 8 字节长，因此在 GDT/LDT 中的相对地址由段选择符的最高 13 位数值乘以 8 得到

GDT 的第一项总是设置为 0，这就确保了空段选择符的逻辑地址会被认为是无效的。因此引起一个处理器异常。

2.2.4 分段单元

分段单元 (segmentation unit) 执行以下步骤：

- 首先检查 TI 字段以决定段描述符保存在哪个描述符表。
- 从段选择符的 index 字段计算段描述符的地址
- 把逻辑地址的偏移量与段描述符 Base 字段的值相加就得到了线性地址

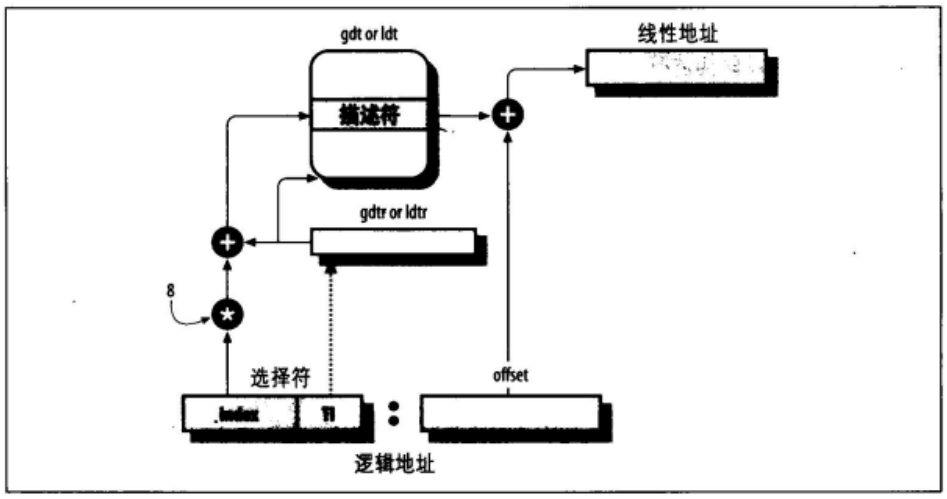

图 2.4: 逻辑地址的转换

注意：如果有了不可编程寄存器，只有当段寄存器内容被改变时才需执行前两个操作。

2.3 Linux 中的分段

实际上分段和分页在某种程度上有点多余，因为都可以划分进程的物理地址空间：分段可以给每个进程分配不同的线性地址空间，而分页可以把同一线性地址空间映射到不同的物理空间。Linux 更喜欢分页：

- 当所有进程使用相同段寄存器值，内存管理变得更简单，也就是其能共享同样的一组线性地址
- RISC 架构对分段的支持有限[1]

运行到用户态的所有 Linux 进程都使用一对相同的段来对指令和数据寻址。也就是用户代码段和用户数据段。相对地，也有内核代码段和内核数据段。

表 2.3: 四个主要的 Linux 段的描述符字段的值

段	*Base*	*G*	*Limit*	*S*	*Type*	*DPL*	*D/B*	*P*
用户代码段	0x00000000	1	0xfffff	1	10	3	1	1
用户数据段	0x00000000	1	0xfffff	1	2	3	1	1
内核代码段	0x00000000	1	0xfffff	1	10	0	1	1
内核数据段	0x00000000	1	0xfffff	1	2	0	1	1

相应的段选择符由宏 __USER_CS，__USER_DS，__KERNEL_CS 和 __KERNEL_DS 分别定义。

[1] *2.6version 的 Linux 只有 80x86 架构才需要使用分段*

所有的段都从 *0x00000000* 开始，也就是说，*Linux* 下逻辑地址与线性地址是一致的，即逻辑地址的偏移量字段与对应的线性地址的值总是一致的。

对指令或数据结构的指针进行保存时，内核不需要设置逻辑地址的段选择符，因为 CS 寄存器含有当前的段选择符。因为已经隐含在 CS 寄存器中。” 在内核态执行” 的段只有一种，叫做代码段，由 __KERNEL_CS 定义，因此当切换为内核态时，只需要将该宏装入 CS 即可。

同样的，对于指向内核数据结构的指针，隐式使用 DS，有宏 __KERNEL_DS。

2.3.1 Linux GDT

多处理器中，每个 CPU 对应一个 GDT。所有的 GDT 都存放在 *‘cpu_gdt_table‘*数组中，而所有的 GDT 地址和大小 (初始化 gdtr 寄存器时使用) 被存放在 *‘cpu_gdt_descr‘*数组中。

```
1  // for i386 GDT_ENTRIES
   #define GDT_ENTRIES 32
3
   // for x86_64 GDT_ENTRIES
5  #define GDT_ENTRIES 16

7  // 8 byte segment descriptor
   struct desc_struct {
9      u16 limit0;
       u16 base0;
11     unsigned base1 : 8, type : 4, s : 1, dpl : 2, p : 1;
       unsigned limit : 4, avl  : 1, l : 1, d   : 1, g : 1, base2 : 8;
13 } __attribute__((packed));

15 extern struct desc_struct cpu_gdt_table[NR_CPUS][GDT_ENTRIES];

17 struct Xgt_desc_struct {
       unsigned short size;
19     unsigned long address __attribute__((packed));
       unsigned short pad;
21 } __attribute__ ((packed));

23 extern struct Xgt_desc_struct idt_descr, cpu_gdt_descr[NR_CPUS];
```

如下是 GDT 的布局示意图，每个 GDT 都包含 18 个描述符和 14 个空的，未使用的，或保留的。插入未使用的项目的是为了使经常一起访问呢的描述符能够处于同一个 *32* 字节的硬件高速缓存行中

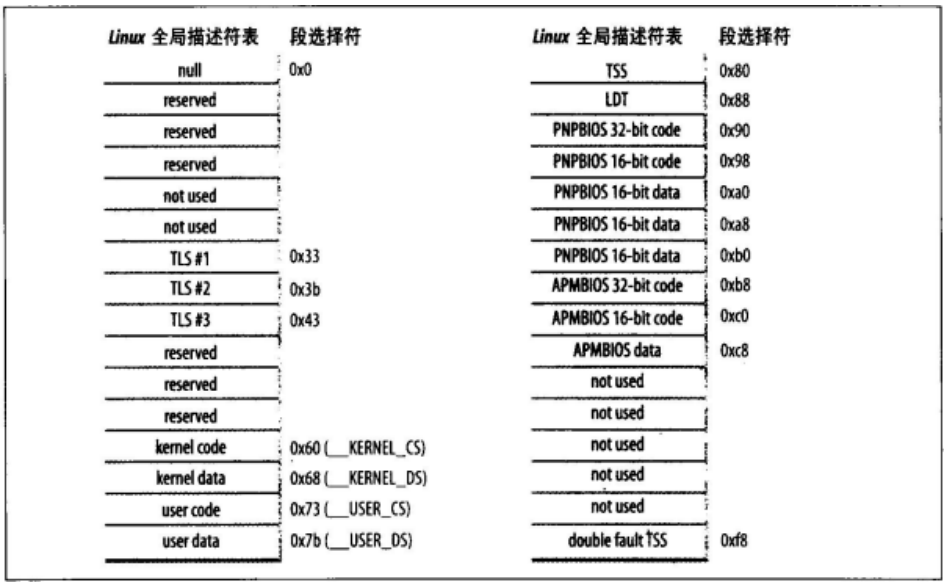

图 2.5: 全局描述符表

每一个 GDT 中包含的 18 个段描述符指向下列的段：

- 用户态和内核态下的代码段和数据段共四个
- 任务状态段 (TSS)，每个处理器一个。

  TSS 相应的线性地址空间都是内核数据段相应线性空间的一个小子集。所有的状态任务段都顺序地放在 init_tss 数组中。注意：第 *n* 个 *CPU* 的 *TSS* 描述符的 *Base* 字段指向 *init_tss* 数组的第 *n* 个元素，*G* 标志清零，*Limit* 字段置为 *0xeb*，*Type* 字段置为 *9* 或 *11* 且 *DPL* 置 *0*
- 1 个包括缺省局部描述符的段，通常被所有进程共享的段
- 3 个局部线程存储 (Thread-Local Storage, TLS) 段

  该机制允许多线程应用程序使用最多 3 个局部线程的数据段。系统调用 set_thread_area() 和 get_thread_area() 分别创建和撤销一个 TLS 段
- 与高级电源管理 (AMP) 相关的三个段

  由于 BIOS 代码使用段，所以 Linux APM 驱动程序调用 BIOS 函数来获取或设置 APM 时，可以自定义的代码段和数据段
- 与支持即插即用 (PnP) 功能的 BIOS 服务程序相关的五个段

  当 PnP 设备驱动程序调用 BIOS 函数来检查 PnP 设备使用的资源时，就可以使用自定义的代码段和数据段
- 被内核用来处理” 双重错误” 异常[1]的特殊 TSS 段

[1]处理异常的程序引发了另一个异常，产生的双重错误

2.3.2 Linux LDT

大多数用户态 Linux 不使用局部描述符表，内核定义了一个缺省的 LDT 供进程共享。缺省的局部描述符放在 *'default_ldt'* 数组中，其包含五个项，但内核仅仅有效地使用了其中两个项。

- 用于 iBCS 执行文件的调用门
- Solaris/x86 可执行文件的调用门

调用门时 80x86 微处理器提供的一种机制，可以在调用与预定义函数时改变 CPU 的特权级。

```
// 8 byte segment descriptor
struct desc_struct {
    u16 limit0;
    u16 base0;
    unsigned base1 : 8, type : 4, s : 1, dpl : 2, p : 1;
    unsigned limit : 4, avl  : 1, l : 1, d   : 1, g : 1, base2 : 8;
} __attribute__((packed));

extern struct desc_struct default_ldt[];
```

但是，如同 Wine 这样的程序，需要创建局部描述符。modify_ldt() 系统调用允许进程创建自己的局部描述符。

任何被 modify_ldt() 创建的自定义局部描述符仍然需要自己的段。

```
/*
* ldt.h
*
* Definitions of structures used with the modify_ldt system call.
*/
#ifndef _LINUX_LDT_H
#define _LINUX_LDT_H

/* Maximum number of LDT entries supported. */
#define LDT_ENTRIES 8192
/* The size of each LDT entry. */
#define LDT_ENTRY_SIZE  8

#ifndef __ASSEMBLY__
struct user_desc {
    unsigned int  entry_number;
```

```
17      unsigned long base_addr;
        unsigned int  limit;
19      unsigned int  seg_32bit:1;
        unsigned int  contents:2;
21      unsigned int  read_exec_only:1;
        unsigned int  limit_in_pages:1;
23      unsigned int  seg_not_present:1;
        unsigned int  useable:1;
25  };

27  #define MODIFY_LDT_CONTENTS_DATA      0
    #define MODIFY_LDT_CONTENTS_STACK     1
29  #define MODIFY_LDT_CONTENTS_CODE      2

31  #endif /* !__ASSEMBLY__ */
    #endif
```

2.4 硬件中的分页

分页单元 (paging unit) 把线性地址转换为物理地址。其中的一个关键任务是把所请求的访问类型与线性地址的访问权限比较，若内存访问无效，则产生缺页异常。

线性地址被分成以固定长度为单位的组，称为页 *(page)*。页内部连续的线性地址被映射到连续的物理地址中。内核就能够指定一个页的物理地址和存取权限，而不用指定页所包含的全部线性地址的存取权限。

一般地，使用属于” 页” 既指一组线性地址，又指包含在这组地址中的数据。

分页单元把所有的 RAM 分成固定长度的页框 (page frame)。每个页框包含一个页，也就是说，一个页框的长度和页的长度是一致的。

把线性地址映射到物理地址的数据结构称为页表 (page table)。

从 80386 开始，所有的 80x86 处理器都支持分页，通过设置 cr0 寄存器的 PG 标志启用。当 PG=0 时，线性地址被解释为物理地址

2.4.1 常规分页

32 位的线性地址被分成 3 个域：

- Directory(目录)

  最高 10 位

- Table(页表)

  中间十位

- Offset(偏移量)

  最低 12 位

线性地址的转换分两步，每一步都基于一种转换表。第一种转换表称为页目录表 (page directory)，第二种转换表称为页表 (page table)

使用这种二级模式的目的在于减少每个进程页表所需 *RAM* 的数量。

正在使用的页目录的物理地址存放在控制寄存器 cr3 中。

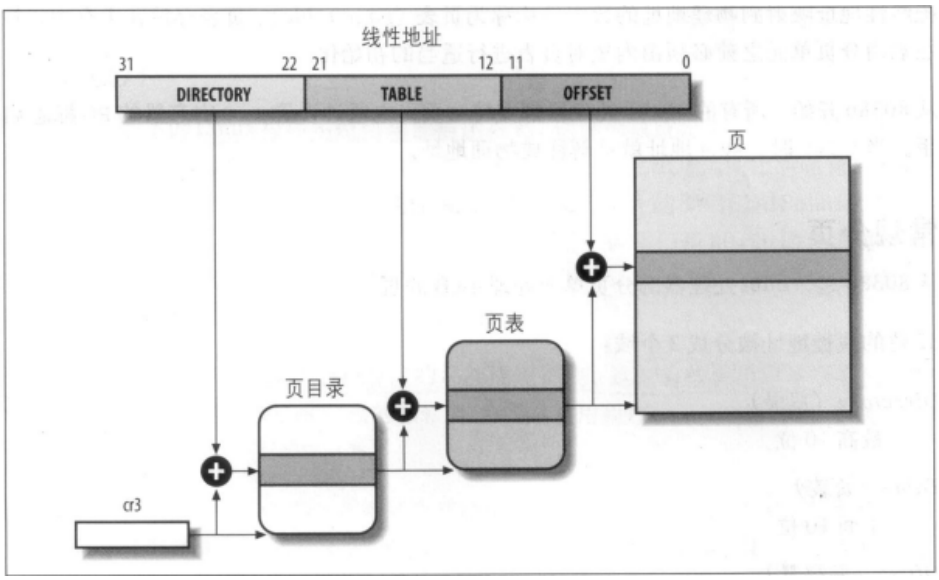

图 2.6: 80x86 处理器分页

线性地址内的 Directory 字段决定页目录中的目录项，而目录项指向适当的页表。地址的 Table 字段依次又决定页表中的表项，表项含有页所在页框的物理地址。Offset 字段决定页框内的相对位置。

Directory 字段和 Table 字段都是 10 位长，因此页目录和页表都可以多达 1024 项。

页目录项和页表有同样的结构，每项都包含下面的字段：

- Present 标志

  若被置 1，所在页就在主存中；若地址转换所需的页表项或页目录项中 Present 标志清零，则分页单元就把该线性地址放在控制寄存器 cr2 中，产生 14 号缺页异常

- 包含页框物理地址最高 20 位的字段

  一个页框 4KN 大小，因此物理地址是 4096 的倍数 (低 12 位总是 0).

- Accessed 标志

  每当分页单元对相应页框进行寻址时设置。必须被操作系统设置。

- Dirty 标志

  只用于页表项。每当一个页框进行写操作时设置。必须被操作系统设置。

- Read/Write 标志

  含有页或页表的存取权限

- User/Supervisor 标志

  含有访问页或页表所需的特权级

- PCD 和 PWT 标志

  控制硬件高速缓存处理页或页表的方式

- Page Size 标志

  只用于页目录项。

- Global 标志

  只用于页表项。用来防止页从 TLB 高速缓存中刷新出去。只有 cr4 寄存器在页全局启用 (Page Global Enable, PGE) 标志置位时才有效

2.4.2 扩展分页

80x86 引入了扩展分页 (extended paging)，允许页框大小为 4MB 而不是 4KB。扩展分页用于把大段连续的线性地址转换成相应的物理地址 *(内核可以不用中间页表进行地址转换，从而节省内存并保存 TLB 项)*

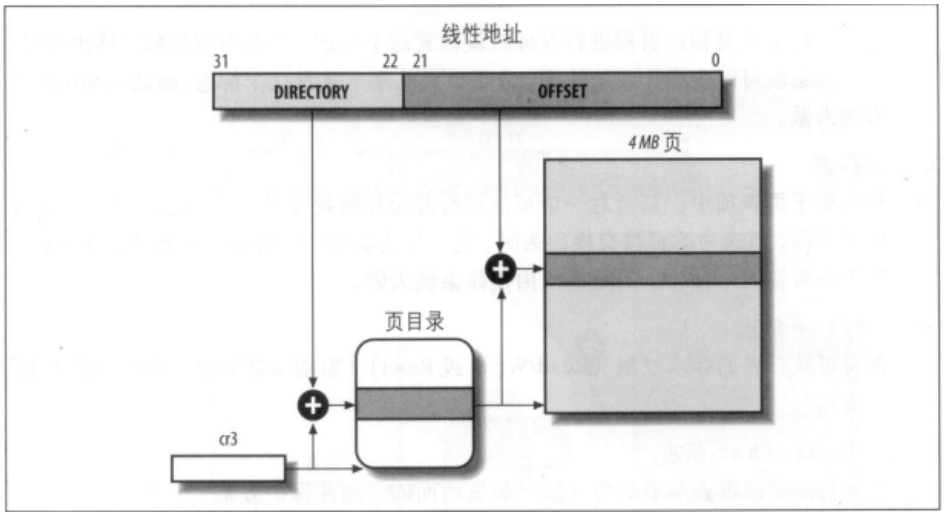

图 2.7: 扩展分页

通过设置 Page Size 字段启用扩展分页：

- Directory

  最高 10 位

- Offset

  最高 22 位

- 扩展分页与正常分页页目录项基本相同，除了

  Page Size 必须被设置

  20 位物理地址字段只有最高 10 位是有意义的

2.4.3 硬件保护方案

页与页表的特权级只有两个，由 User/Supervisor 标志控制。页的存储权限只有两种，若 Read/Write 标志为 0，说明是只读的，否则是可读可写的。

2.4.4 物理地址扩展 (PAE) 分页机制

处理器所支持的 *RAM* 容量受连接到地址总线上的地址管脚数限制。

Intel 引入物理地址扩展 (Physical Address Extension，PAE) 机制，另外一种叫做页大小扩展机制在 Pentium 中引入。

通过设置 cr4 控制寄存器中的物理地址扩展 PAE 标志激活 PAE，为了支持 PAE，因此分页机制需要更改：

- 64GB 的 RAM 分为 2^{24} 个页框，页表项物理地址字段扩展到 24 位。
- 引入页目录指针表 (Page Directory Pointer Table，PDPT) 的页表新级别，由四个 64 位表项组成
- cr3 控制寄存器包含一个 27 位的页目录指针表 (PDPT) 基地址字段。
- 把线性地址映射到 4KB 的页时，32 位线性地址按下列方式解释：

  cr3：指向一个 PDPT

  位 31 - 30：指向 PDPT 中 4 个项中的一个

  位 29 - 21：指向页目录中 512 个项的一个

  位 20 - 12：指向页表中 512 项中的一个

  位 11 - 0：4KB 页中的偏移量
- 把线性地址映射到 2MB 的页时，32 位线性地址按下列方式解释：

  cr3：指向一个 PDPT

  位 31 - 30：指向 PDPT 中 4 个项中的一个

  位 29 - 21：指向页目录中 512 个项的一个

  位 20 - 0：2MB 页中的偏移量

2.4.5 硬件高速缓存

为了缩小 *CPU* 与 *RAM* 之间的速度不匹配，引入了硬件高速缓存内存 *(hardware cache memory)*。

硬件高速缓存基于局部性原理[1]*(locality principle)*, 这说明程序的循环结构及相关

[1] *1.* 时间局部性（*Temporal Locality*）：指在程序执行过程中，刚刚访问过的数据或指令很可能在不久的将来再次被访问。这是因为程序中的循环、迭代和函数调用等结构会导致对同一数据或指令的重复访问。通过利用时间局部性，计算机系统可以将频繁访问的数据或指令缓存到高速缓存（*Cache*）中，以提高访问速度。

2. 空间局部性（*Spatial Locality*）：指在程序执行过程中，与当前访问的数据或指令在空间上相邻的数据或指令很可能在不久的将来被访问。这是因为程序中的数组、结构体和代码块等通常具有连续的存

数组可以组成线性数组，最近最常用的相邻地址在最近的将来可能又会用到。

为了适应这种机制，80x86 引入了行的新单位。行由几十个连续字节组成，以脉冲突发模式[2](*burst mode*) 在慢速 *DRAM* 和快速 *SRAM* 之间传送，实现高速缓存。

在极端情况下，高速缓存可以直接映射，此时主存中的一个行总是放在高速缓存中完全相同的位置。另一种情况，高速缓存时全相联的 (fully associative)，意味着主存中任意一个行可以存放在高速缓存中的任意位置。大多数高速缓存时 N-路组关联的 (N-way set associative)，意味着任意一个行能够放在 N 行中的任意一行

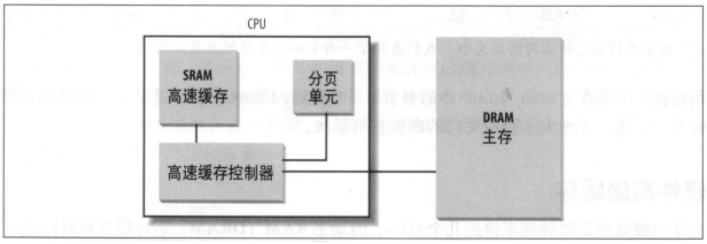

图 2.8: 高速缓存器

高速缓存单元插在分页单元和主存之间。包含一个高速缓存内存和高速缓存控制器。高速缓存内存存放内存中真正的行，控制器存放一个表项数组，每个表项对应高速缓存内存中的行。

每个表项都有一个标签 (tag) 和描述高速缓存行状态的标志 (flag)。

访问 RAM 时，CPU 从物理地址中提取子集的索引号并把子集种所有行的标签与物理地址的高几位相比较，若相同，则称缓存命中 (cache hit)，否则缓存未命中 (cache miss)。

当命中时，高速缓存控制器根据存取类型采取不同的操作。

- 读操作

  控制器从行种选择数据并送往 *CPU*，不需要访问 *RAM*

- 写操作

  控制器有两种方案：通写 (write-through) 和回写 (write-back)

  通写中，控制器既写 *RAM* 也写高速缓存行

  回写中，控制器只更新高速缓存行，不更改 *RAM*，但回写结束后，*RAM* 必须被更新

当缓存为命中，高速缓存行被写回 RAM 中，且有必要的情况下，将正确的行写入高速缓存表项中

在多处理器中，每一个处理器都有自己的硬件高速缓存，因此需要额外电路来保持缓存内容的同步。

储结构。通过利用空间局部性，计算机系统可以预取和预加载与当前访问的数据或指令相邻的数据或指令，以提高内存访问效率。

[2]脉冲突发模式是指在通信系统中，用于携载业务和控制信道的信息的一种模式。

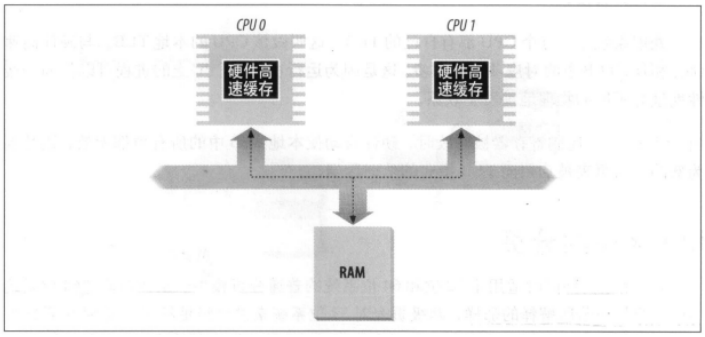

图 2.9: 双处理器中的高速缓存

高速缓存监听 (cache snooping) 用于检查 CPU 是否需要更新，当然这些属于硬件级，内核并不需要关心。

处理器上的 cr0 寄存器的 CD 标志位用来启用或禁用高速缓存电路，该寄存器的 NW 标志指明高速缓存是使用通写还是回写。

2.4.6 转换后援缓冲器 (TLB)

80x86 中包含了一个 TLB(Translation Lookaside Buffer) 的高速缓存用于加快线性地址的转换。

当一个线性地址第一次使用时，通过慢速访问 RAM 的页表计算出对应的物理地址，同时该物理地址会被存放在 TLB 表项 (entry) 以便同一个线性地址可以快速转换。

注意：当 cr3 寄存器更改后，硬件自动使本地 TLB 中的所有项都无效，因为此时 TLB 是旧数据。

2.5 Linux 中的分页

Linux 为了能够同时适用于 32 位和 64 位系统的普通分页模型，因此采用了三级分页模型，但是从 2.6.11 版本开始，采用四级分页模型：

- 页全局目录 (Page Global Directory)
- 页上级目录 (Page Upper Directory)
- 页中间目录 (Page Middle Directory)
- 页表 (Page Table)

对于没有启用物理地址扩展的 32 位系统，两级页表已经足够，Linux 使页上级目录和页中间目录为 0，从而取消该字段。启用了物理地址扩展的 32 位系统采用了三级页表，页全局目录对应页目录指针表，取消了页上级目录。

Linux 的进程处理很大程度上依赖于分页，线性地址到物理地址的自动转换使得下面的设计目标变得可行：

- 给每个进程分配一块不同的物理地址空间

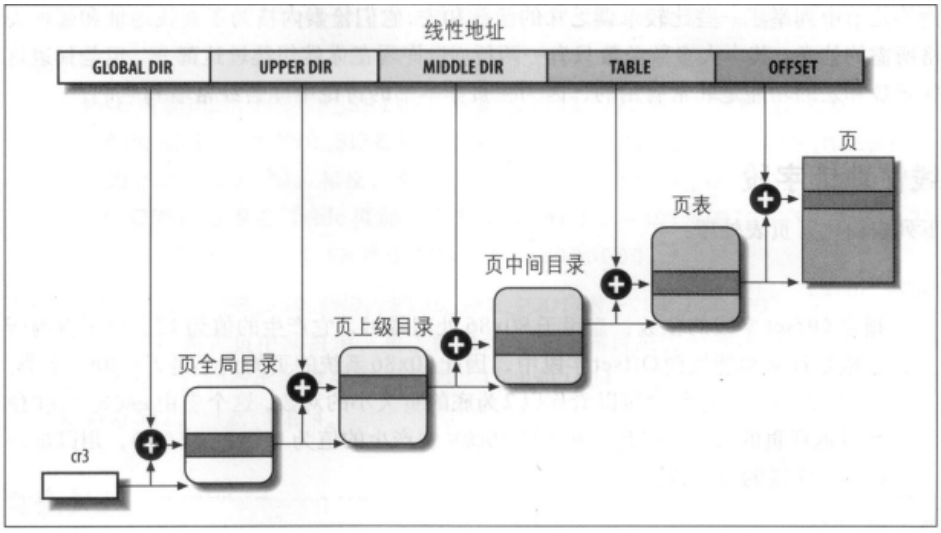

图 2.10: Linux 的分页模式

- 区别页和页框的不同，这使得存放在某个页框中的一个页，可以保存在磁盘上，然后重新装入不同页框内 (虚拟机内存机制的基本要素)

2.5.1 线性地址字段

下列宏简化了页表处理 (全在 include/asm-x86_64/pgtable.h 中)

- PAGE_SHIFT

  指定 Offset 字段的位数。当使用 80x86 处理器时，默认的指为 12。

  该宏由 PAGE_SIZE 使用以返回页大小,最后由 PAGE_MASK 产生的指 0xfffff000，来屏蔽 Offset 字段

- PMD_SHIFT

  指定线性地址的 Offset 字段和 Table 字段的总位数。

  PMD_SIZE 用于计算由页中间目录的一个单独表项所映射的区域大小,PMD_MASK 用于屏蔽 Offset 和 Table 字段的所有位

  当 PAE 被禁用时，PMD_SHIFT 的指为 22(Offset 的 12 位 + Table 的 10 位)；PMD_SIZE 的值为 2^{22}，PMD_MASK 的值为 0fffc00000

  当 PAE 被激活时，PMD_SHIFT 的指为 21(Offset 的 12 位 + Table 的 9 位)；PMD_SIZE 的值为 2^{21}，PMD_MASK 的值为 0fffe00000

- PUD_SHIFT

  确定页上级目录项能映射的区域大小的对数

  PUD_SIZE 用于计算页全局目录中的一个单独表项所能映射的区域,PUD_MASK 用于频闭 Offset 字段、Table、Middle Air 字段和 Upper Air 字段

- PGDIR_SHIFT

  去欸的那个页全局目录能映射的大小的对数

  PGDIR_SIZE 用于计算页全局目录中一个单独的表项所能映射的区域,PGDIR_MASK

用于用于频闭 Offset 字段、Table、Middle Air 字段和 Upper Air 字段

2.5.2 页表处理

pte_t、pmd_t、pud_t 和 pgd_t 分别描述页表项、页中间目录项、页上级目录项和页全局目录项的格式。*当 PAE 被激活时，都是 64 位的数据类型，否则为 32 位数据类型*

pgprot_t 是另一个 64 位 (32 位) 的数据类型，表示与一个单独表项相关的保护标志

五种类型转换红 (__pte, __pmd, __pud, __pgd 和 __pgrot) 把一个无符号整数转换为所需的类型，另外五种类型转换宏 (pte_val, pmd_val, pud_val, pgd_val 和 pgrot_val) 执行相反的转换。

```
// 如果对应表项为0，则返回值为1
#define pte_none(x) (!pte_val(x))
#define pmd_none(x) (!pmd_val(x))
#define pud_none(x) (!pud_val(x))
#define pgd_none(x) (!pgd_val(x))

// 原子的写入指定值
#define set_pte(pteptr, pteval) (*(pteptr) = pteval)
#define set_pte_atomic(pteptr, pteval) set_pte(pteptr,pteval)

// 若a和b指向同一页且指定相同访问优先级
#define pte_same(a, b)      ((a).pte == (b).pte)
#define pte_same(a, b)      ((a).pte_low == (b).pte_low)

// 通过输入参数传递来检查页中间目录项，若指向一个不能使用的页表，宏值为1
#define pmd_bad(x)  \
    ((pmd_val(x) & (~PTE_MASK & ~_PAGE_USER)) != _KERNPG_TABLE )
```

```
static inline int pte_user(pte_t pte)    // 读User/Supervisor标志
    { return pte_val(pte) & _PAGE_USER; }
extern inline int pte_read(pte_t pte)    // 读User/Supervisor标志
    { return pte_val(pte) & _PAGE_USER; }
extern inline int pte_exec(pte_t pte)    // 读User/Supervisor标志
    { return pte_val(pte) & _PAGE_USER; }
extern inline int pte_dirty(pte_t pte)   // 读Dirty标志
```

```
    { return pte_val(pte) & _PAGE_DIRTY; }
extern inline int pte_young(pte_t pte)  // 读Accessed标志
    { return pte_val(pte) & _PAGE_ACCESSED; }
extern inline int pte_write(pte_t pte)  // 读Read/Write标志
    { return pte_val(pte) & _PAGE_RW; }
static inline int pte_file(pte_t pte)   // 读Dirty标志
    { return pte_val(pte) & _PAGE_FILE; }
```

Listing 2.1: 读页标志的函数

```
extern inline pte_t pte_mkread(pte_t pte)       // 设置U/S标志
extern inline pte_t pte_mkexec(pte_t pte)       // 设置U/S标志
extern inline pte_t pte_rdprotect(pte_t pte)    // 清除U/S标志
extern inline pte_t pte_exprotect(pte_t pte)    // 清除U/S标志

extern inline pte_t pte_mkold(pte_t pte)        // 清除Accessed标志
extern inline pte_t pte_mkyoung(pte_t pte)      // 设置Accessed标志

extern inline pte_t pte_wrprotect(pte_t pte)    // 清除R/W标志
extern inline pte_t pte_mkwrite(pte_t pte)      // 设置R/W标志

extern inline pte_t pte_mkdirty(pte_t pte)      // 设置Dirty标志
extern inline pte_t pte_mkclean(pte_t pte)      // 清除Dirty标志

// 与pte_mkclean类似，但作用于指向页表项的指针，并返回旧值
static inline int ptep_test_and_clear_dirty(pte_t *ptep)
// 与pte_mkold类似，但作用于指向页表项的指针，并返回旧值
static inline int ptep_test_and_clear_young(pte_t *ptep)
// 与pte_wrprotect类似，但作用于指向页表项的指针
static inline void ptep_set_wrprotect(pte_t *ptep)
// 与pte_mkdirty类似，但作用于指向页表项的指针
static inline void ptep_mkdirty(pte_t *ptep)

// 将页表项p访问权限设置为指定值v
extern inline pte_t pte_modify(pte_t pte, pgprot_t newprot)

// 设置页表项中的Page Size和Present
#define mk_pte_huge(entry) \
    (pte_val(entry) |= _PAGE_PRESENT | _PAGE_PSE)
```

Listing 2.2: 设置页标志的函数

```
1 暂时略过
```

Listing 2.3: 对页表项操作的宏

2.5.3 物理内存布局

在初始化阶段，内核必须建立在一个物理地址映射来指定哪些物理地址范围对内核可用。

内核将下列页框记为保留：

- 在不可用的物理地址范围内的页框
- 含有内核代码和已初始化的数据结构的页框

保留的页框中的页绝不能被动态分配或交换到磁盘上。

Linux 通常从 0x00100000 开始的地方启动，因为在之前的 RAM 需要留给：

- 页框 0 由 BIOS 使用，存放加电自检 (Power-On Self-Test，POST) 期间检查到的系统硬件配置
- 物理地址从 0x000a0000 到 0x000fffff 的范围通常留给 BIOS 例程，并且映射 ISA 图形卡的内部内存。
- 第一个 MB 内的其他页框可能由特定计算机模型保留

在启动过程早期,内核询问 BIOS 并了解物理内存大小,随后内核执行 machie_specific_-memory_setup() 函数，建立物理地址映射

表 2.4: BIOS 提供的物理地址映射举例

开始	结束	类型
0x00000000	0x0009ffff	Usable
0x000f0000	0x000fffff	Reserved
0x00100000	0x07fefffff	Usable
0x07ff0000	0x07ff2ffff	ACPI data
0x07ff3000	0x07ffffff	ACPI NVS
0xfff0000	0xffffffff	Reserved

为了避免把内核装入一组不连续的页框内，Linux 更愿意跳过 RAM 中的第一个 MB

2.5.4 进程页表

进程的线性地址空间被分为两部分：

- 从 0x0 到 0xbfffffff 的线性地址，无论进程运行在用户态还是内核态都可以寻址
- 从 0xc00000000 到 0xffffffff 的线性地址，只有内核态的进程能够寻址

变量名称	说明
num_physpages	最高可用页框的页框号
totalram_pages	可用页框的总数量
min_low_pfn	RAM 中在内核映像后第一个可用页框的页框号
max_pfn	最后一个可用页框的页框号
max_low_pfn	被内核直接映射的最后一个页框的页框号（低地址内存）
totalhigh_pages	内核非直接映射的页框的总数（高地址内存）
highstart_pfn	内核非直接映射的第一个页框的页框号
highend_pfn	内核非直接映射的最后一个页框的页框号

图 2.11: 描述内核物理内存布局的变量

2.5.5 内核页表

内核维持着一组自己使用的页表，驻留在主内核页全局目录中。

内核初始化自身的页表分为两个阶段：

- 第一阶段创建一个有限的地址空间，包括内核的代码段和数据段、初试页表和用于存放动态数据结构的共 128KB 大小的空间
- 第二阶段利用剩余的 RAM 并适当的建立分页表

临时内核页表

略

2.5.6 固定映射的线性地址

固定映射的线性地址 (fix-mapped linear address) 基本上是一种类似于 0xffffc000 这样的常量线性地址，其对应的物理地址不必定于线性地址减去 0xc000000。

每个固定映射的线性地址由 enum fixed_addresses 数据结构中的整形索引来表示：

```
enum fixed_addresses {
    VSYSCALL_LAST_PAGE,
    VSYSCALL_FIRST_PAGE = \
        VSYSCALL_LAST_PAGE + \
        ((VSYSCALL_END-VSYSCALL_START) >> PAGE_SHIFT) - 1,
    VSYSCALL_HPET,
    FIX_HPET_BASE,
#ifdef CONFIG_X86_LOCAL_APIC
    FIX_APIC_BASE,  /* local (CPU) APIC) -- required for SMP or not
    */
#endif
#ifdef CONFIG_X86_IO_APIC
    FIX_IO_APIC_BASE_0,
    FIX_IO_APIC_BASE_END = FIX_IO_APIC_BASE_0 + MAX_IO_APICS-1,
```

```
#endif
    __end_of_fixed_addresses
};


#define __fix_to_virt(x)    (FIXADDR_TOP - ((x) << PAGE_SHIFT))
extern void __this_fixmap_does_not_exist(void);

/*
 * 'index to address' translation. If anyone tries to use the idx
 * directly without translation, we catch the bug with a NULL-
   deference
 * kernel oops. Illegal ranges of incoming indices are caught too.
 */
extern inline unsigned long fix_to_virt(const unsigned int idx)
{
    /*
     * this branch gets completely eliminated after inlining,
     * except when someone tries to use fixaddr indices in an
     * illegal way. (such as mixing up address types or using
     * out-of-range indices).
     *
     * If it doesn't get removed, the linker will complain
     * loudly with a reasonably clear error message..
     */
    if (idx >= __end_of_fixed_addresses)
        __this_fixmap_does_not_exist();

    return __fix_to_virt(idx);
}
```

第 3 章 进程

进程是任何躲到程序涉及的操作系统中的基本概念

3.1 进程、轻量级进程和线程

OS 教科书的常规定义：进程是程序执行时的一个实例。每一个进程都只有一个父亲。

从内核的观点来看：进程的目的就是担当分配系统资源 *(CPU time、memory)* 的实体。

对于一个进程的创建，其原理是接受父进程地址空间的一个 (逻辑) 拷贝。尽管父子进程可以共享含有程序代码 *(正文)* 的页，但是拥有各自独立的数据拷贝，因此子进程对内存单元的修改对父进程是不可见的 *(反之亦然)*

对于现代 Unix 系统来说，需要支持多线程应用————拥有很多相对独立执行流的用户程序共享应用的大部分数据结构。这样的系统中，进程由多个线程组成，每个线程都代表进程的一个执行流。

目前而言，大多数多线程应用都是基于 pthread(POSIX thread) 库的标准库函数集编写的

Linux 使用轻量级进程 (lightweight process) 对多线程进行更好的支持。对于轻量级进程而言，可以共享资源，只要其中一个修改共享资源，另一个就会立即查看并同步。

3.1.1 进程描述符

进程描述符 (process descriptor，由 task_struct 类修饰)，其字段包含了与进程相关的所有信息。

图 3-1 右边的六个数据结构涉及进程拥有的所有特殊资源，目前我们只讨论两种：进程的状态以及进程的父子关系

3.1.2 进程状态

进程描述符中的 state 字段描述了当前进程所处的状态。其由一组标志组成，其中每一个标志描述了一种可能的进程状态。

值得注意的是：状态之间是互斥的，严格意义上只允许存在一种状态。

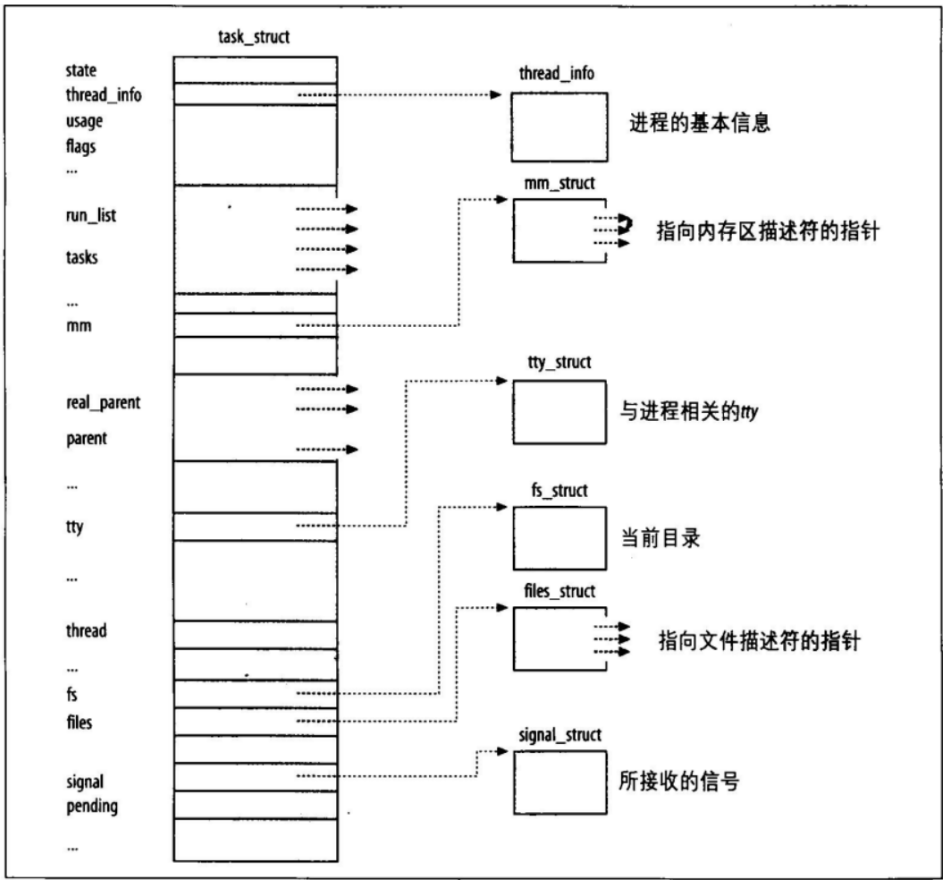

图 3.1: Linux 进程描述符

```
#define TASK_RUNNING          0
#define TASK_INTERRUPTIBLE    1
#define TASK_UNINTERRUPTIBLE  2
#define TASK_STOPPED          4
#define TASK_TRACED           8
#define EXIT_ZOMBIE           16
#define EXIT_DEAD             32
```

Listing 3.1: 进程状态一览

- TASK_RUNNING

  进程要么已经执行，要么准备执行

- TASK_INTERRUPTIBLE

  进程被挂起，知道某个条件为真。产生硬件中断，释放进程正在等待的系统资源或传递一个信号都是可以唤醒进程的条件 (也就是恢复到 TASK_RUNNING)

- TASK_UNINTERRUPTIBLE

  与上一个状态类似，但传递信号无法唤醒。一般用于特殊情况下，进程必须等待，直到一个不能被中断的实践发生

- TASK_STOPPED

  进程的执行被暂停，当接收到 SIGSTOP、SIGTSTP、SIGTTIN 和 SIGTTOU 信号后，进入暂停

- TASK_TRACED

  进程的执行由 debugger 程序暂停。当一个进程被另一个进程监控时，任何信号都可以把这个进程置于 TASK_TRACED 状态

- EXIT_ZOMBIE

  进程的执行被终止，但是父进程未发布 wait() 类系统调用来返回有关死亡进程的信息

- EXIT_DEAD

  最终状态，由于父进程刚发布 wait() 类系统调用，因而进程被系统删除，但防止其他线程也执行 wait() 类系统调用，为了防止冲突，因而将僵死状态设置为僵死撤销 (将亡) 状态

一般的，进程状态可以由以下简单的赋值语句设置，也可以由宏设置：

```
p->state = TASK_RUNNING;

#define __set_task_state(tsk, state_value)      \
    do { (tsk)->state = (state_value); } while (0)
#define set_task_state(tsk, state_value)        \
    set_mb((tsk)->state, (state_value))

#define __set_current_state(state_value)                \
    do { current->state = (state_value); } while (0)
#define set_current_state(state_value)      \
    set_mb(current->state, (state_value))
```

3.1.3 标识一个进程

进程和进程描述符之间有非常严格的一一对应关系，这使得进程描述符标识进程称为一种方便的方式。

另一方面，类 Unix 操作系统允许用户使用进程标识符 process ID(PID) 来标识进程，PID 存放在进程描述符的 pid 字段中。

PID 被顺序编号，新创建的进程通常是前一个 PID 加 1。PID 的值通常有一个上线，缺省情况下，最大 PID 号是 32767(PID_MAX_DEFAULT - 1)

由于循环使用 PID，因此需要通过与个 pidmap_array 位图来标识当前已分配的 PID 和闲置的 PID。

Linux 希望同一个线程组内有同一个 PID，因此，一个线程组中的所有线程使用和该线程组的领头线程的 *PID*，被存放在进程描述符中的 *tgid* 字段。系统调用 getpid() 实际上是返回的 tgid 的值，而非 PID。

3.1.4 进程描述符处理

进程是动态实体，因此内核必须能够同时处理多进程，并把进程描述符存放在动态内存中，而非在永久分配给内核的内存区。

对于每个进程来说，Linux 把内核态的进程堆栈和进程描述符中的 thread_info(线程描述符) 紧凑的存放在一个单独的存储区域内。

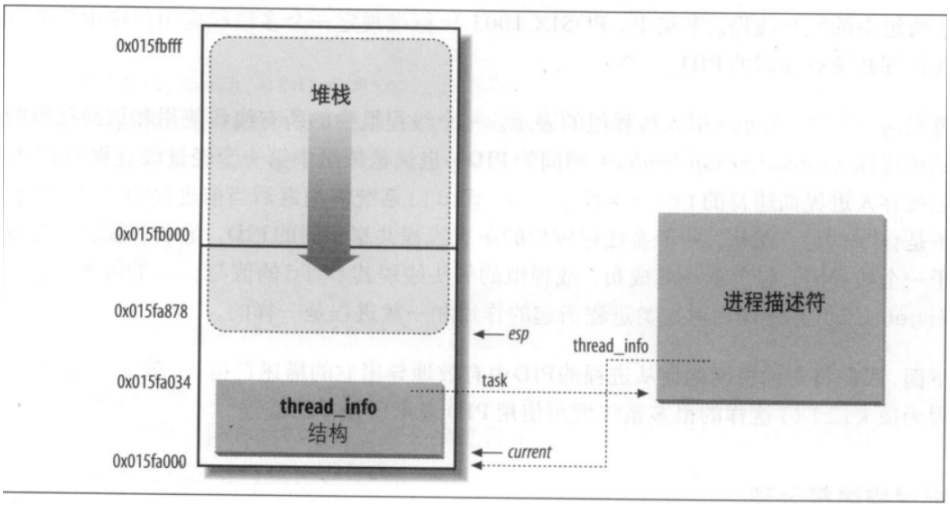

图 3.2: thread_info 结构和进程内核栈存放在两个连续的页框中

线程描述符驻留于该内存区的开始，栈从末端向下开始增长。

esp 寄存器是 CPU 的栈指针 (x86 架构)，用来存放栈顶单元的地址。栈起始于末端，并朝该内存区开始的地方开始增长。从用户态切换到内核态后，进程的内核总是空的

在内核中，使用 union 对该结构进行描述：

```
#define THREAD_SIZE        (4096)
#else
#define THREAD_SIZE        (8192)
#endif

union thread_union {
    struct thread_info thread_info;
    unsigned long stack[THREAD_SIZE/sizeof(long)];
};
```

3.1.5 标识当前进程

从效率上看：*thread_info* 结构体与内核态堆栈之间的紧密结合提供的主要好处为：内核可以轻易从 *esp* 的值获取 *CPU* 正在运行进程的 *thread_info* 的地址

事实上，如果 thread_union 结构长度是 8K(2^{13})，则内核屏蔽掉 esp 的低 13 位有效位就能够获取 thread_info 的基地址。这项工作由以下函数实现：

```
/* how to get the thread information struct from C */
static inline struct thread_info *current_thread_info(void)
{
    struct thread_info *ti;
    __asm__("andl %%esp,%0; ":"=r" (ti) : "0" (~(THREAD_SIZE - 1)))
   ;
    return ti;
}
```

也就是说，通过该函数内联汇编：

```
movl $0xffffe000, %ecx
andl %esp, %ecx
movl %ecx, p
```

这样执行后，p 就包含了 CPU 当前运行进程的 thread_info 结构的指针。

但是，进程中最常用的是进程描述符的地址而非 thread_info 的地址，因此，内核需要调用 current 宏，其等价于

```
current_thread_info()->task;


movl $0xffffe000, %ecx
andl %esp, %ecx
movl (%ecx), p
```

查看源码可以知道，task 在 thread_info 上的偏移量是 0，因此上述三条指令就能够将进程描述符指针赋值到 p 上。

3.1.6 双向链表

对于每一个链表，都需要实现一组原语操作：初始化，插入和删除，遍历等操作。Linux 内核定义了 list_head 数据结构，字段 next 和 prev 分别表示通用黄翔链表向前和

向后的指针元素。

```
struct list_head {
    struct list_head *next, *prev;
};
```

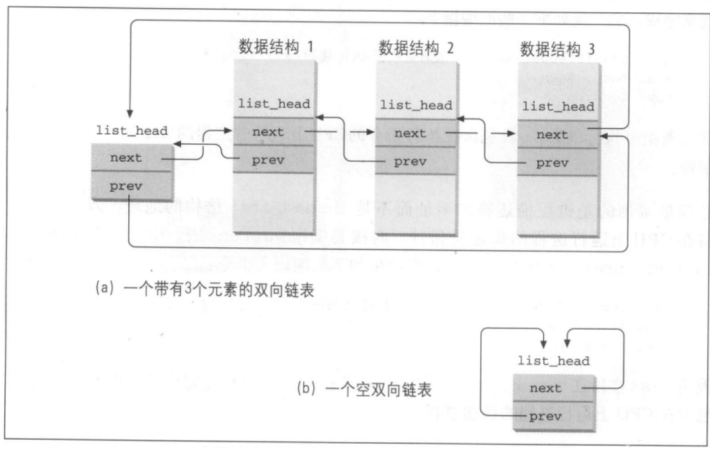

图 3.3: list_head 构造双向链表

对于新链表而言，是用 LIST_HEAD(list_name) 宏创建的，其声明类型为 list_head 的新变量 list_name。见上图中 b 图。Linux 内核为链表提供了一些原语操作：

```
// 初始化链表
#define LIST_HEAD_INIT(name) { &(name), &(name) }
#define LIST_HEAD(name) \
    struct list_head name = LIST_HEAD_INIT(name)
// 链表头插
static inline void list_add(
    struct list_head *new, struct list_head *head)
{
    __list_add(new, head, head->next);
}
// 链表尾插
static inline void list_add_tail(
    struct list_head *new, struct list_head *head)
{
    __list_add(new, head->prev, head);
}
// 链表删除
static inline void list_del(struct list_head *entry)
{
```

```
    __list_del(entry->prev, entry->next);
    entry->next = LIST_POISON1;
    entry->prev = LIST_POISON2;
}
// 链表判空
static inline int list_empty(const struct list_head *head)
{
    return head->next == head;
}
// 获取链表元素
#define list_entry(ptr, type, member) \
    container_of(ptr, type, member)
// 遍历链表
#define list_for_each(pos, head) \
for (pos = (head)->next; prefetch(pos->next), pos != (head); \
        pos = pos->next)

#define list_for_each_entry(pos, head, member)                \
for (pos = list_entry((head)->next, typeof(*pos), member);   \
        prefetch(pos->member.next), &pos->member != (head);      \
        pos = list_entry(pos->member.next, typeof(*pos), member))
```

进程链表

进程链表把所有的进程描述符链接起来。每一个 task_struct 结构都包含一个 list_head 类型的 tasks 字段，prev 和 next 分贝指向前后的 task_struct 元素。

进程链表的头是 init_struct 描述符，也就是 0 进程 (process 0) 或 swapper 进程的进程描述符。

SET_LINKS 和 REMOVE_LINKS 分别用于从进程链表中插入和删除一个进程描述符，且考虑了父子进程间关系。同时也有宏 for_each_process，遍历整个进程链表。

```
#define next_task(p)    \
    list_entry((p)->tasks.next, struct task_struct, tasks)
#define prev_task(p)    \
    list_entry((p)->tasks.prev, struct task_struct, tasks)
#define for_each_process(p) \
    for (p = &init_task ; (p = next_task(p)) != &init_task ; )
```

TASK_RUNNING 状态的进程链表

当内核寻找新进程运行时，只需要考虑处于 *TASK_RUNNING* 状态的进程

早先的 Linux 把可运行进程都放在运行队列 (runqueue) 的链表中，但是维持链表中进程优先级排序的开销过大，因此早期的调度程序不得不为选择最佳可运行程序而扫描整个进程。

而 2.6 实现的运行队列，目的是为了让调度程序在固定时间内选出最佳可运行进程，与队列中可运行的进程数无关。

提高调度程序运行速度的诀窍是建立多个可运行进程链表，每种进程优先级对应一个不同的链表。

每个 task_struct 描述符包含一个 list_head 类型的字段 run_list，如果进程优先级等于 k(取值为 0 139)，run_list 就把该进程链入优先级为 k 的可运行进程队列中。

在内核中，运行队列的主要数据结构还是组成运行队列的进程描述符链表，所有这些链表都由一个单独 prio_array_t 数据结构来实现

```
struct prio_array {
    // 链表中进程描述符的数量
    unsigned int nr_active;
    // 优先权位图，仅当某个优先权的进程链表不为空时设置
    unsigned long bitmap[BITMAP_SIZE];
    // 140个优先权队列的头节点
    struct list_head queue[MAX_PRIO];
};

typedef struct prio_array prio_array_t;

static void enqueue_task(struct task_struct *p, prio_array_t *array
    )
{
    sched_info_queued(p);
    list_add_tail(&p->run_list, array->queue + p->prio);
    __set_bit(p->prio, array->bitmap);
    array->nr_active++;
    p->array = array;
}
```

enqueue_task 把进程描述符插入某个运行队列的链表，进程描述符的 prio 字段存放进程的动态优先权，array 时一个指针，指向当前队列的 prio_array_t 数据结构。

3.1.7 进程间的关系

程序创建的进程具有父/子关系，而一个进程创建多个子进程，子进程之间具有兄弟关系。

因此，进程描述符中引入几个字段表示这些关系。进程 *0* 和进程 *1* 由内核创建

```
/* real parent process (when being debugged) */
struct task_struct *real_parent;
struct task_struct *parent; /* parent process */
/*
* children/sibling forms the list of my children plus the
* tasks I'm ptracing.
*/
/* list of my children */
struct list_head children;
/* linkage in my parent's children list */
struct list_head sibling;
```

下图中演示了一组进程间的关系：

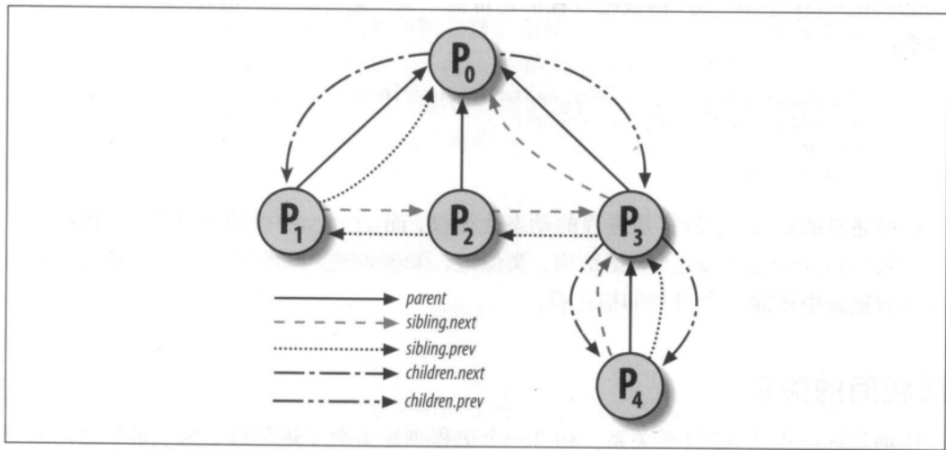

图 3.4: 五个进程间的亲属关系

值得注意的是，一个进程可能是一个进程组或登录会话的领头进程，也可能是线程组的临潼进程，还可能是跟踪其他进程执行

```
group_leader        // P所在进程组的领头进程描述符指针
signal->pgrp        // P所在进程组的领头进程PID
tgid                // P所在线程组的领头进程PID
signal->session     // P的登录会话领头进程PID
ptrace_children     // 链表头部，包含所有被debugger程序跟踪子线程
ptrace_list         // 跟踪进程父进程链表的前后元素
```

pidhash 表及链表

在一些情况下，内核必须从进程的 PID 导出对应的进程描述符指针：为 kill() 系统调用提供服务

顺序扫描进程链表并检查进程描述符 PID 的做法可行，但是相当低效。为了快速查找，引入了 4 个散列表。引入 4 个散列表的原因是：进程描述符包含了表示不同类型的 PID 字段，每种 PID 都需要自己的散列表

表 3.1: 散列表和进程描述符中的相关字段

Hash 表的类型	字段名	说明
PIDTYPE_PID	pid	进程的 PID
PIDTYPE_TGID	tgid	进程组领头进程的 PID
PIDTYPE_PGID	pgrp	线程组领头进程的 PID
PIDTYPE_SID	session	会话领头进程的 PID

内核初始化期间动态地为 4 个散列表分配空间，并把它们的地址存入 *pid_hash* 数组

用 pid_hashfn 宏把 PID 转化为表索引：

```
#if BITS_PER_LONG == 32
/* 2^31 + 2^29 - 2^25 + 2^22 - 2^19 - 2^16 + 1 */
#define GOLDEN_RATIO_PRIME 0x9e370001UL
#elif BITS_PER_LONG == 64
/*  2^63 + 2^61 - 2^57 + 2^54 - 2^51 - 2^18 + 1 */
#define GOLDEN_RATIO_PRIME 0x9e37fffffffc0001UL
#else
#error Define GOLDEN_RATIO_PRIME for your wordsize.
#endif

#define pid_hashfn(nr) hash_long((unsigned long)nr, pidhash_shift)

static inline unsigned long hash_long(unsigned long val, unsigned
    int bits)
{
     unsigned long hash = val;

```

```
#if BITS_PER_LONG == 64
    /* Sigh, gcc can't optimise this alone like it does for 32
   bits. */
    unsigned long n = hash;
    n <<= 18; hash -= n;
    n <<= 33; hash -= n;
    n <<= 3; hash += n;
    n <<= 3; hash -= n;
    n <<= 4; hash += n;
    n <<= 2; hash += n;
#else
    /* On some cpus multiply is faster, on others gcc will do
   shifts */
    hash *= GOLDEN_RATIO_PRIME;
#endif

    /* High bits are more random, so use them. */
    return hash >> (BITS_PER_LONG - bits);
}
```

但是，我们清楚：散列 *(hash)* 函数并不总能确保 *PID* 与表的索引一一对应，两个不同的 *PID* 散列到相同的表索引被称为冲突 *(colliding)*

Linux 利用链表来处理冲突的 PID：每个表项是由冲突的进程描述符组成的双向链表

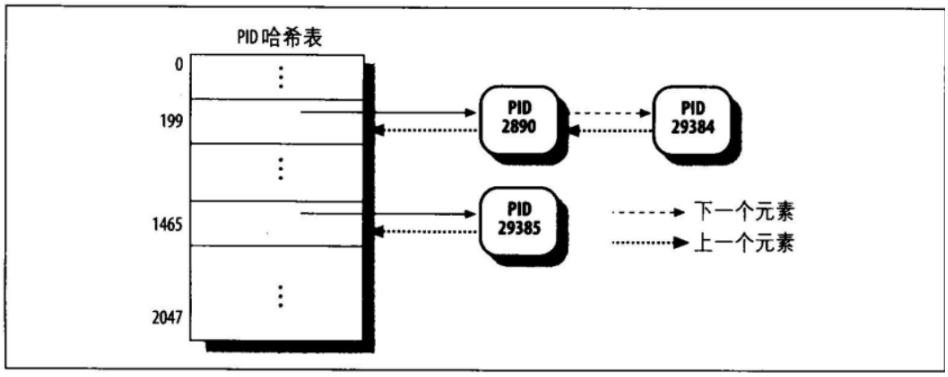

图 3.5: pidhash 表及链表

可以看见，PID2890 和 PID29384 被散列到第 200 的元素处，然后通过双向链表进行链接。

具有链表的散列法比 *PID* 到表索引的线性转换更为优越，因为系统中的进程数总是远远小于 *32768*。

PID 散列表的数据结构解决了上述的难题，最主要的数据结构是四个 PID 结构的素组，其在进程描述符的 PID 字段中：

```
enum pid_type
{
    PIDTYPE_PID,
    PIDTYPE_TGID,
    PIDTYPE_PGID,
    PIDTYPE_SID,
    PIDTYPE_MAX
};

struct pid
{
    // pid的数值
    int nr;
    // 链接散列表的前后元素
    struct hlist_node pid_chain;
    // 每一个pid的进程链表头
    struct list_head pid_list;
};
```

下面是处理 PID 散列表的函数和宏：

- do_each_task_pid(who, type, task)
- while_each_task_pid(who, type, task)

  循环作用在 PID 值等于 nr 的 PID 链表上，类型由 type 给出，task 参数指出当前被扫描的元素的进程描述符
- task_t *find_task_by_pid_type(int type, int nr)

  在 type 类型的散列表中查找 pid 等于 nr 的进程，返回所匹配的进程描述符指针
- find_task_by_pid(nr)
- int fastcall attach_pid(task_t *task, enum pid_type type, int nr)

  把 task 指向的 PID 等于 nr 的进程描述符插入 type 类型的散列表中。如果已经存在，则只把 task 插入已有 PID 的进程链表
- void fastcall detach_pid(task_t *task, enum pid_type type)

  删除 task 所指向的进程描述符，若删除后链表未变空，则函数终止；否则还要从 type 类型的散列表中删除进程描述符，若该 PID 在任何散列表中不存在，则清除 PID 位图中的对应位

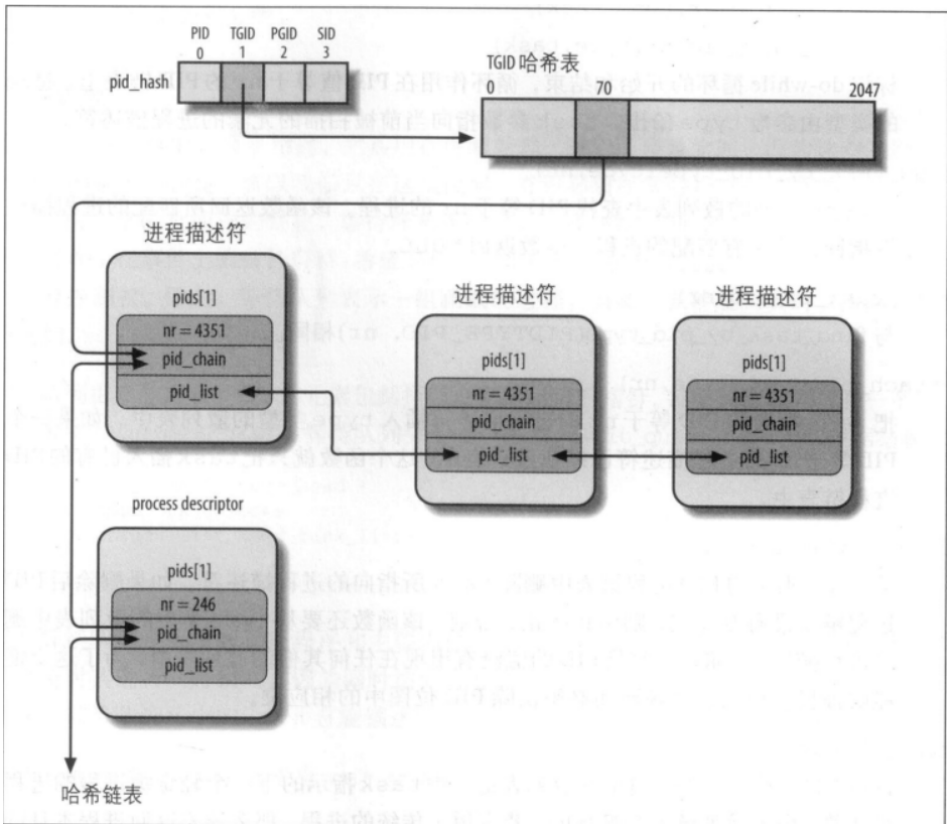

图 3.6: pid 散列表

- task_t fastcall *next_thread(const task_t *p)

  返回 TGID 类型中的散列表链接中 task 知识的下一个轻量级进程的进程描述符

3.1.8 如何组织进程

运行队列链表把处于 TASK_RUNNING 状态的所有进程都组织再一起，但对于其他状态：

- 没有为 TASK_STOPPED、EXIT_ZOMBIE 或 EXIT_DEAD 状态的进程专门建立链表

等待队列

等待队列再中断处理、进程同步和定时方面尤为重要。

进程必须经常等待某些事件的发生，例如的带一个磁盘操作的弘治，的带释放系统资源，或等待时间经过固定的间隔。等待队列实现了在事件上的等待

希望等待特定的事件的进程将自己放入合适的等待队列，并放弃控制权，一旦某条件为真，由内核唤醒。

等待队列由双向列表实现，其元素包括指向进程描述符的指针

```
struct __wait_queue_head {
    spinlock_t lock;
    struct list_head task_list;
};
typedef struct __wait_queue_head wait_queue_head_t;

struct __wait_queue {
    unsigned int flags;
#define WQ_FLAG_EXCLUSIVE   0x01
    struct task_struct * task;
    wait_queue_func_t func;
    struct list_head task_list;
};
typedef struct __wait_queue wait_queue_t;
```

等待队列是由中断处理程序和主要内核函数修改而来，因此必须对该链表进程保护以免被同时访问，因此通过自旋锁来确保。

等待队列链表中的每个元素代表一个睡眠进程，该进程等待某一事件的发生。然而要唤醒等待队列中的所有睡眠进程有时并不方便，因此有两种睡眠进程。

互斥进程 (等待队列元素的 flags 字段为 1) 由内核有选择地唤醒；而非互斥进程 (flags 值为 0) 总是由内核在事件发生时唤醒。

等待队列的操作

```
#define __WAITQUEUE_INITIALIZER(name, tsk) {                \
.task       = tsk,                          \
.func       = default_wake_function,                \
.task_list  = { NULL, NULL } }

#define DECLARE_WAITQUEUE(name, tsk)                        \
    wait_queue_t name = __WAITQUEUE_INITIALIZER(name, tsk)
```

可以用 DECLEAR_WAIT_QUEUE_HEAD 宏定义一个新等待队列的头，其静态声明一个叫 name 的等待队列的头变量并对该变量的 lock 和 task_list 字段进行初始化。

函数 init_waitqueue_head() 可以用来初始化动态分配的等待队列的头变量

```
static inline void init_waitqueue_head(wait_queue_head_t *q)
{
```

```
3     q->lock = SPIN_LOCK_UNLOCKED;
      INIT_LIST_HEAD(&q->task_list);
5 }
```

非互斥进程将由 default_wake_function() 唤醒。

一旦定义了一个元素，必须将其插入等待队列，add_wait_queue() 函数把一个非互斥进程插入等待队列链表的第一个位置。add_wait_queue_exclusive() 函数把一个互斥进程插入等待队列的最后一个位置。

```
1 static inline void __add_wait_queue(
      wait_queue_head_t *head, wait_queue_t *new)
3
  static inline void add_wait_queue_exclusive_locked(
5     wait_queue_head_t *q, wait_queue_t *wait)
```

要等待特定条件的进程可以调用如下中的任何一个函数：

- void fastcall __sched sleep_on(wait_queue_head_t *q)

  该函数把当前进程状态设置为 TASK_UNINTERRUPTIBLE，并把其插入到特定的等待队列。然后调度程序重新开始另一个程序的执行

  当睡眠进程被唤醒，则调度程序重新执行 sleep_on() 函数，并把该进程从队列中删除
- __sched interruptible_sleep_on_timeout(wait_queue_head_t *q, long timeout)

  和上一个函数类似，但该函数将状态设置为 TASK_INTERRUPTIBLE，因此接受一个信号就可以唤醒当前进程
- long fastcall __sched sleep_on_timeout(wait_queue_head_t *q, long timeout)

  和上一个函数类似，但其允许定义一个时间间隔，过了该间隔，由内核唤醒，因此调用 schedule_timeout() 函数
- 在 2.6kernel 下引入了 prepare_to_wait()、prepare_to_wait_exclusive() 和 finish_wait()，提供了另一种途径来使当前进程在一个等待队列中睡眠

```
1 void fastcall prepare_to_wait(
      wait_queue_head_t *q, wait_queue_t *wait, int state)
3 void fastcall prepare_to_wait_exclusive(
      wait_queue_head_t *q, wait_queue_t *wait, int state)
5 void fastcall finish_wait(wait_queue_head_t *q, wait_queue_t *wait)

7 DEFINE_WAIT(wait);
  prepare_to_wait_exclusive(&wq, &wait, TASK_INTERRUPTIBLE);
```

```
...
if (!condition)
    schedule();
finish_wait(&wq, &wait);
```

prepare_to_wait() 和 prepare_to_wait_exclusive() 中的第三个参数设置进程的状态，然后把等待队列元素的互斥标志 flag 设置为 0 或 1，最后把等待元素 wait 插入到 wq 为头的等待队列的链表中。

wait_event 和 wait_event_interruptible 宏使它们的调用进程在等待队列上睡眠，一直到修改了给定条件为止

```
#define __wait_event(wq, condition)                                    \
do {                                                                   \
    DEFINE_WAIT(__wait);                                               \
                                                                       \
    for (;;) {                                                         \
        prepare_to_wait(&wq, &__wait, TASK_UNINTERRUPTIBLE);           \
        if (condition)                                                 \
            break;                                                     \
        schedule();                                                    \
    }                                                                  \
    finish_wait(&wq, &__wait);                                         \
} while (0)

#define wait_event(wq, condition)                                      \
do {                                                                   \
    if (condition)                                                     \
        break;                                                         \
    __wait_event(wq, condition);                                       \
} while (0)
```

值得注意的使：*sleep_on() 类函数在以下条件下不能使用，那就是必须测试条件且该条件还没有得到验证时，又让进程去睡眠，这样是竞争条件产生的根源。*

内核可以通过下面任何一个宏唤醒等待队列中的进程并把其状态设置为 RUNNING:wake_up、wake_up_nr、wake_up_all、wake_up_interruptible、wake_up_interruptible_nr、wake_up_interruptible_all、wake_up_interruptible_sync、wake_up_locked。

- 所有宏都考虑到处于 INTERRUPTIBLE 状态的睡眠进程;若宏的名字中不含”interruptible”，那么处于 UNINTERRUPTIBLE 的进程也会被考虑到

- 所有宏都具有唤醒请求状态的所有非互斥进程
- 名字中含有”nr” 字符串的宏唤醒给定数具有请求状态的互斥进程；”all” 唤醒所有具有请求状态的互斥进程，都不含的唤醒一个互斥进程
- 名字中不含”snyc” 的宏检查被唤醒进程的优先级是否高于系统中正在运行的进程优先级，并在必要时调用 schedule()
- wake_up_locked 需要自旋锁被持有

```
void fastcall __wake_up(wait_queue_head_t *q, unsigned int mode,
    int nr_exclusive, void *key)
{
    unsigned long flags;

    spin_lock_irqsave(&q->lock, flags);
    __wake_up_common(q, mode, nr_exclusive, 0, key);
    spin_unlock_irqrestore(&q->lock, flags);
}
#define wake_up(x)                  \
    __wake_up(                      \
        x, TASK_UNINTERRUPTIBLE | TASK_INTERRUPTIBLE, 1, NULL)
```

3.1.9 进程资源限制

每一个进程都有一组相关的资源限制 *(resource limit)*，限定了进程能使用的系统资源数论。

```
/* Allow arch to control resource order */
#ifndef __ARCH_RLIMIT_ORDER
#define RLIMIT_CPU          0   /* CPU time in ms */
#define RLIMIT_FSIZE        1   /* Maximum filesize */
#define RLIMIT_DATA         2   /* max data size */
#define RLIMIT_STACK        3   /* max stack size */
#define RLIMIT_CORE         4   /* max core file size */
#define RLIMIT_RSS          5   /* max resident set size */
#define RLIMIT_NPROC        6   /* max number of processes */
#define RLIMIT_NOFILE       7   /* max number of open files */
#define RLIMIT_MEMLOCK      8   /* max locked-in-memory address
    space */
#define RLIMIT_AS           9   /* address space limit */
#define RLIMIT_LOCKS        10  /* maximum file locks held */
```

```
14 #define RLIMIT_SIGPENDING    11    /* max number of pending signals */
   #define RLIMIT_MSGQUEUE      12    /* maximum bytes in POSIX mqueues
      */
16
   #define RLIM_NLIMITS         13
18 #endif
```

对当前进程的资源限制存放在 current->signal->rlim 字段中，即进程的信号描述符的一个字段，该字段是类型为 rlimit 结构的数组

```
   struct rlimit {
2      unsigned long    rlim_cur;
       unsigned long    rlim_max;
4  };
```

rlim_cur 是当前资源限制，rlim_max 是资源限制所允许的最大值。

利用 getrlimit() 和 setrlimit() 系统调用，用户能够把一些资源的 rlim_cur 限制增加。当然的，只有管理员权限才能使用。

3.2 进程切换

为了控制进程的执行，内核必须有能力挂起 *CPU* 上允许的进程，并恢复以前挂起某个进程的执行，这种行为被称为进程切换 *(process switch)*、任务切换 *(task switch)* 或上下文切换 *()*

3.2.1 硬件上下文

尽管每个进程都可以拥有自己的地址空间，但所有进程必须共享 *CPU* 寄存器。

进程恢复执行前必须装入寄存器的一组数据被称为硬件上下文 (hardware context)。硬件上下文是进程可执行上下文的一个子集，在 Linux 中，进程硬件的上下文的一部分存放在 TSS 段，剩余部分存放在内核态的堆栈中。

在下面的描述中，我们假定用 prev 表示切出的进程描述符，next 表示切入的进程描述符：保存 prev 硬件上下文，用 next 硬件上下文代替 prev。

在早期的 Linux 中，通过 far jmp 指令[1]跳到 next 进程 TSS 描述符的选择符来执行进程切换：

[1]在 *x86* 中，该指令既能修改 *cs* 寄存器，又能修改 *eip* 寄存器，而简单的 *jmp* 指令只能修改 *eip* 寄存器。

- 通过一组 mov 指令逐步执行切换 (这样能够较好的控制装入数据的合法性)

进程切换只发生在内核态，在执行进程切换之前，用户态进程使用的所有寄存器内部都已保存在内核堆栈上，也包括 ss 和 esp 这对寄存器的内容

任务状态段

任务状态段 (Task State Segment，TSS) 用来存放硬件上下文。Linux 并不使用硬件上下文切换，但强制为不同 CPU 创建一个 TSS：

- 当 CPU 从用户态切换到内核态时，可以从 TSS 获取内核态堆栈的地址
- 当用户态进程试图通过 in 或 out 指令访问一个 I/O 端口时，CPU 需要访问 TSS 中的 I/O 许可权位图 (Permission Bitmap) 以检查该进程是否拥有访问端口的权力

  检查 eflags 寄存器中的 2 位 IOPL 字段，如果该字段为 3，控制单元就执行 I/O 指令，否则执行下一个检查

  访问 tr 寄存器以确定当前 TSS 和相应的 I/O 许可权位图

  检查 I/O 指令中指定的 I/O 端口在 I/O 许可权位图中对应的位

tss_struct 结构描述了 TSS 的格式，每一个 CPU 都存放一个 TSS(使用 init_tss 数组描述)。

每个 TSS 都有自己的 8 字节的任务状态段描述符 (Task State Segment Descriptor，TSSD)。该描述符包括指向 TSS 起始地址的 32 位 Base 字段，20 位 Limit 字段和其他字段。

thread 字段

每次切换进程时，被替换进程的硬件上下文必须保存在别处，因此每个进程描述符中包含一个类型为 thread_struct 的 thread 字段，只要进程被切换，内核就把硬件上下文保存在该结构中。

```
struct thread_struct {
    unsigned long   rsp0;
    unsigned long   rsp;
    unsigned long   userrsp;    /* Copy from PDA */
    unsigned long   fs;
    unsigned long   gs;
    unsigned short  es, ds, fsindex, gsindex;
/* Hardware debugging registers */
    unsigned long   debugreg0;
    unsigned long   debugreg1;
    unsigned long   debugreg2;
    unsigned long   debugreg3;
    unsigned long   debugreg6;
```

```
14      unsigned long   debugreg7;
    /* fault info */
16      unsigned long   cr2, trap_no, error_code;
    /* floating point info */
18      union i387_union    i387  __attribute__((aligned(16)));
    /* IO permissions. the bitmap could be moved into the GDT, that
       would make
20      switch faster for a limited number of ioperm using tasks. -AK
       */
        int     ioperm;
22      unsigned long   *io_bitmap_ptr;
        unsigned io_bitmap_max;
24  /* cached TLS descriptors. */
        u64 tls_array[GDT_ENTRY_TLS_ENTRIES];
26  } __attribute__((aligned(16)));
```

3.2.2 执行进程切换

进程切换可能只发生在：schedule() 函数上。但是此处仅关注如何执行一个进程切换

从本质上说，每个进程切换由两步组成：

- 切换页全局目录以安装一个新的地址空间
- 切换内核太堆栈和硬件上下文

switch_to 宏

```
    #define switch_to(prev,next,last) do {                  \
2   unsigned long esi,edi;                          \
    asm volatile("pushfl\n\t"                       \
4           "pushl %%ebp\n\t"                       \
            "movl %%esp,%0\n\t" /* save ESP */      \
6           "movl %5,%%esp\n\t" /* restore ESP */   \
            "movl $1f,%1\n\t"       /* save EIP */      \
8           "pushl %6\n\t"      /* restore EIP */   \
            "jmp __switch_to\n"             \
10          "1:\t"                          \
            "popl %%ebp\n\t"                        \
12          "popfl"                         \
```

```
        :"=m" (prev->thread.esp),"=m" (prev->thread.eip),    \
14      "=a" (last),"=S" (esi),"=D" (edi)                    \
        :"m" (next->thread.esp),"m" (next->thread.eip), \
16      "2" (prev), "d" (next));                             \
} while (0)
```

进程切换的第二步由 switch_to 宏执行。

首先，该宏有三个参数，分别表示被替换进程，替换进程和表示把进程 C 的描述符地址写在了内存中的什么位置 (方便重新引用)

也就是说，实际上进程切换涉及了三个进程 (并非两个)。当 schedule 暂停进程 A，激活 B 时，switch_to 宏使得 A 进程被冻结，如果内核想再次激活 A，那么必须暂停另一个进程 C(通常不同于 B)，于是就需要暂停 C，激活 B。但目前 next 仍指向 B 的描述符，因此 C 的引用就被丢失了。

那么，last 参数就是一个输出参数，用于表示宏把进程 C 的描述符地址写在哪个地方了[1]。

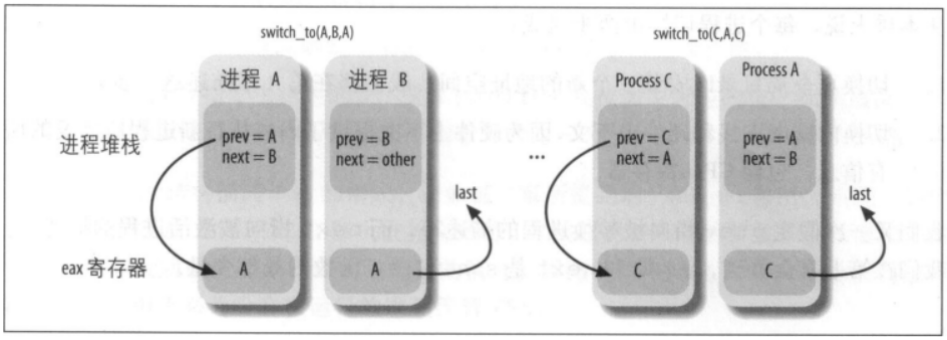

图 3.7: 通过对进程切换保存对进程 C 的引用

由于 switch_to 采用扩展内联汇编，因此可读性较差

1) 在 eax 和 dx 寄存器分别保存 prev 和 next
2) 把 eflags 和 ebp 寄存器的内容保存在 prev 内核栈中，因为这些值在更换后不变
3) 把 esp 的内容保存在 prev->thread.esp 中以使得该字段指向 prev 内核栈的栈顶
4) 把 next->thread.esp 装入 esp，此时内核开始在 next 的内核栈工作。因此，改变内核栈意味着改变当前进程。
5) 把标记为 1 的地址存入 prev->thread.eip。当被替换的进程重新恢复时，进程执行被标记为 1 的指令
6) 宏把 next->thread.eip 的值 (绝大多数情况时被标记为 1 的地址) 压入 next 的内核栈
7) 跳转到 __switch_to() 函数

[1] 一般而言，B 进程执行后，可能会执行其他进程，因此后面才会是 C 进程暂停，激活 A 进程。

8) 被进程 B 替换的进程 A 再次获得 CPU：执行保存 eflags 和 ebp 寄存器的指令，这两条指令的第一条指令被标记为 1

9) 拷贝 eax 寄存器的内容到 switch_to 宏的第三个参数 last 标识的内存区域中[1]

```
movl prev, %eax
movl next, %ebx
pushfl
pushl %ebp
movl %esp, 484(%ebp)
movl 484(%edx), %esp
movl $1f, 480(%eax)
pushl 480(%edx)
jmp __switch_to

1:
    popl %ebp
    popfl
movl %eax, last
```

Listing 3.2: 对于 switch_to 的过程描述

__switch_to() 函数

__switch_to() 函数执行大多数开始于 switch_to 宏的进程切换，这个函数作用于 prev_p 和 next_p 参数，这两个参数标识前一个进程和新进程。为了强迫函数从寄存器中获取参数，内核利用 __attribute__ 和 regpram 关键字，由 gcc 编译程序实现

```
struct task_struct fastcall * __switch_to(
    struct task_struct *prev_p, struct task_struct *next_p)
```

函数的执行步骤如下：

- 执行由 __unlazy_fpu() 宏产生的代码，以有选择地保存 prev_p 进程的 FPU、MMX 和 XMM 寄存器的内容
- 执行 smp_processor_id() 宏获得本地 (local)CPU 的下标，即执行代码的 CPU。
- 把 next_p->thread.esp0 装入对应于本地 CPU 的 TSS 的 esp0 字段
- 把 next_p 进程使用的线程局部存储 (TLS) 段装入本地 CPU 的全局描述符，三个段选择描述符保存在进程描述符内的 tls_array 数组中
- 把 fs 和 gs 段寄存器的内容分别存放在 prev_p->thread.fs 和 prev_p->thread.gs 中

[1] *eax 寄存器指向刚被替换进程的描述符，当前指向的 schedule 函数重新使用了 prev 的局部变量：movl %eax, prev*

- 用 next_p->thread.debugreg 数组的内容装在 dr0,...,dr7 中的 6 个调试寄存器
- 如果有必要，更新 TSS 中的 I/O 位图。等 *next_p* 或 *prev_p* 有其自己的定制 *I/O* 权限位图时必须这么做
- 终止 __switch_to 通过使用返回语句

3.2.3 保存和加载 FPU、MMX 及 XMM 寄存器

算术浮点单元 (floating-point unit，FPU) 被继承到 CPU 中。MMX 用于加速多媒体应用程序的执行，其指令作用于 FPU。

80x86 微处理器并不在 TSS 中自动保存 FPU、MMX 和 XMM 寄存器，而是通过硬件支持由 cr0 寄存器中的一个 TS(Task Switching) 标志组成，遵循：

- 每当执行硬件上下文切换时，设置 TS 标志
- 每当 TS 标志被设置时，执行 ESCAPE、MMX、SSE 或 SSE2 指令，控制单元就产生一个”Device not available” 异常

TS 标志使得内核只有在真正需要时才保存和恢复 FPU、MMX、XMM 寄存器。其被描述的数据结构在进程描述符中的 thread.i387 子字段中：

```
union i387_union {
    struct i387_fsave_struct    fsave;
    struct i387_fxsave_struct   fxsave;
    struct i387_soft_struct soft;
};
```

可以看见，该结构允许选择其中一种结构使用：i387_soft_struct 由无数学协处理器的 CPU 模型使用，Linux 内核通过软件模拟协处理器来支持这些老式芯片。i387_fsave_struct 支持协处理器，而最后一种则是支持 SSE 和 SSE2 扩展的 CPU 模型。

进程描述符包含两个附加的标志：

- 包含在 thread_info 中的 TS_USEDFPU 标志，用于标识当前执行过程中是否使用 FPU、MMX 和 XMM 寄存器
- 包含在 task_struct 中的 flags 字段中的 PF_USED_MATH 标志。该标志标识 thread.i387 字段是否有意义

  当进程调用 execve() 时，thread.i387 不再有意义

  当在用户态执行一个进程执行一个信号处理程序时，不再有意义

保存 FPU 处理器

可以看见，__unlazy_fpu 宏检查 prev 中的标志位，如果 TS_USEDFPU 标志被设置，就说明 prev 在本次执行中使用了 FPU、MMX、SSE 或 SSE2 指令，因此内核需要保存其上下文

```
#define __unlazy_fpu( tsk ) do { \
    if ((tsk)->thread_info->status & TS_USEDFPU) \
        save_init_fpu( tsk ); \
} while (0)
```

- 把 FPU 中的内容转储到 prev 进程描述符中，然后重新初始化 FPU。
- 重置 prev 的 TS_USEDFPU 标志
- 使用 stts() 宏设置 cr0 的 TS 标志

```
static inline void __save_init_fpu( struct task_struct *tsk )
{
    if ( cpu_has_fxsr ) {
        asm volatile( "fxsave %0 ; fnclex"
                      : "=m" (tsk->thread.i387.fxsave) );
    } else {
        asm volatile( "fnsave %0 ; fwait"
                      : "=m" (tsk->thread.i387.fsave) );
    }
    tsk->thread_info->status &= ~TS_USEDFPU;
}
```

装载 FPU 寄存器

当 next 进程刚恢复执行时，浮点寄存器的内容还没有被恢复，因为 cr0 中的 TS 标志已被设置，因此 next 进程第一次试图执行 ESCAPE、MMX 或 SSE/SSE2 指令时，控制单元产生"Device not available" 异常，内核 (异常处理程序) 运行 math_state_restore() 函数

```
asmlinkage void math_state_restore(struct pt_regs regs)
{
    struct thread_info *thread = current_thread_info();
    struct task_struct *tsk = thread->task;

    clts();      /* Allow maths ops (or we recurse) */
    if (!tsk_used_math(tsk))
        init_fpu(tsk);
    restore_fpu(tsk);
```

```
    thread->status |= TS_USEDFPU; /* So we fnsave on switch_to() */
}
```

该函数清除 cr0 中的 TS 标志，且如果 thread.i387 字段无效，则调用 init_fpu() 重新设置该字段，并将 PF_USED_MATH 标志设置为 1.

在内核态使用 FPU、MMX 和 SSE/SSE2 单元

在内核态使用时，需要注意：

- 在使用协处理器之前，如果用户态使用了 FPU，内核必须调用 kernel_fpu_begin()，保存 fpu 并重置 TS 标志
- 使用完成后，必须使用 kernel_end_fpu 设置 TS 标志

3.3 创建进程

Unix 系统依赖进程创建来满足用户的需求。

传统 Unix 通过统一的方式对待所有进程：子进程复制父进程所有资源。

现代 Uni 引入三种不同的机制解决该问题：

- 写时复制技术允许父子进程读相同的物理页。只要二者中有一个试图写一个物理页，内核就把该页的内容拷贝到新页，并分配给正在写的进程
- 轻量级进程允许父子进程共享每个进程在内核的数据结构
- vfork() 创建的进程能共享其父进程的内存地址空间

3.3.1 clone、fork、vfork 系统调用

Linux 系统中，轻量级进程由 clone() 系统调用创建的：

- fn

  指定一个由新进程执行的函数
- arg

  指向传递 fn 函数的参数
- flags

  clone flags 信息
- child_stack

  把用户态堆栈指针赋值给子进程的 esp 寄存器，调用进程应该总是为子进程分配新的堆栈
- tls

表示线程局部存储段 (TLS) 的地址，只有 CLONE_SETTLS 标志有效才有意义

- ptid

  表示父进程的用户态变量地址，该父进程具有与新轻量级进程一样的 PID，只有在 CLONE_PARENT_SETTID 有效才有意义

- ctid

  表示轻量级进程用户态变量地址，只有 CLONE_CHILD_SETTID 标志有效才有意义

```
1  /* set if VM shared between processes */
   #define CLONE_VM      0x00000100
3  /* set if fs info shared between processes */
   #define CLONE_FS      0x00000200
5  /* set if open files shared between processes */
   #define CLONE_FILES 0x00000400
7  /* set if signal handlers and blocked signals shared */
   #define CLONE_SIGHAND    0x00000800
9  /* set if we want to let tracing continue on the child too */
   #define CLONE_PTRACE     0x00002000
11 /* set if the parent wants the child to wake it up on mm_release */
   #define CLONE_VFORK 0x00004000
13 /* set if we want to have the same parent as the cloner */
   #define CLONE_PARENT     0x00008000
15 /* Same thread group? */
   #define CLONE_THREAD     0x00010000
17 /* New namespace group? */
   #define CLONE_NEWNS 0x00020000
19 /* share system V SEM_UNDO semantics */
   #define CLONE_SYSVSEM    0x00040000
21 /* create a new TLS for the child */
   #define CLONE_SETTLS     0x00080000
23 /* set the TID in the parent */
   #define CLONE_PARENT_SETTID 0x00100000
25 /* clear the TID in the child */
   #define CLONE_CHILD_CLEARTID     0x00200000
27 /* Unused, ignored */
   #define CLONE_DETACHED       0x00400000
29 /* set if the tracing process can't force CLONE_PTRACE on this
      clone */
   #define CLONE_UNTRACED       0x00800000
```

```
31 /* set the TID in the child */
   #define CLONE_CHILD_SETTID   0x01000000
33 /* Start in stopped state */
   #define CLONE_STOPPED        0x02000000
```

Listing 3.3: clone flags

事实上，clone 是 C 语言库中的一个封装函数，负责建立新轻量级进程的堆栈并且调用对编程者隐藏的 clone 系统调用。

实现 clone 的 sys_clone 服务例程没有 fn 和 arg 参数。封装函数把 fn 指针放在子进程堆栈中的某个位置，该位置就是该封装函数本身返回地址存放的位置。arg 指针正好存放在子进程堆栈中 fn 的下方

传统的 fork 系统调用是由 clone 实现的，其中 clone 的 flags 参数指定为 SIGCHLD 信号及所有清零的 clone 标志，child_stack 由父进程当前的堆栈指针。因此，父子进程暂时共享一个用户态堆栈 (但是由于写时拷贝机制，通常只要其中一个进程更改栈，就会立即得到各自的用户态堆栈拷贝)

```
long sys_clone(
2    unsigned long clone_flags, unsigned long newsp,
     int __user *parent_tid, int unused, int __user *child_tid)
```