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 14, 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：

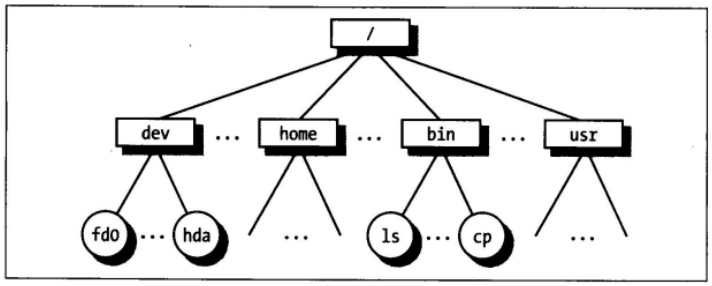

图 1.1: 目录树结构

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

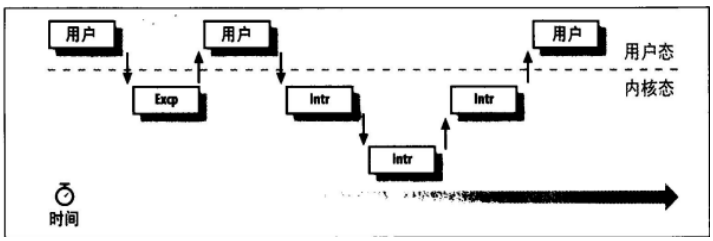

图 1.2: 内核控制路径的交错执行

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

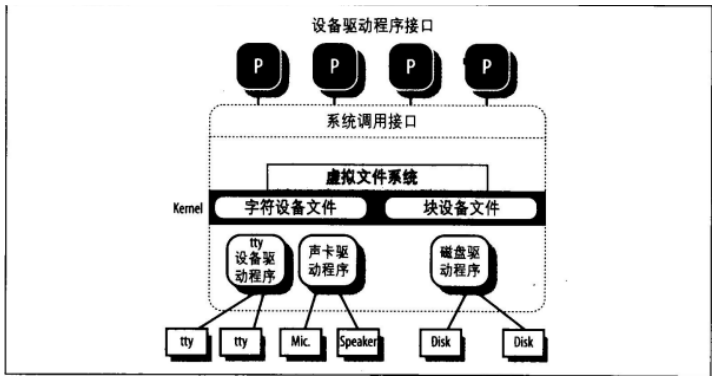

图 1.3: 设备驱动程序接口

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。

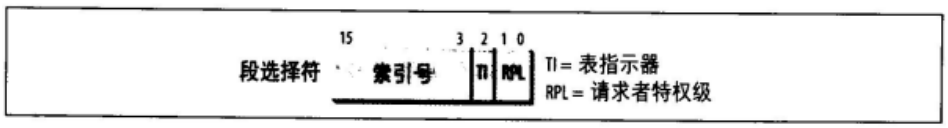

图 2.1: 段选择符格式

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

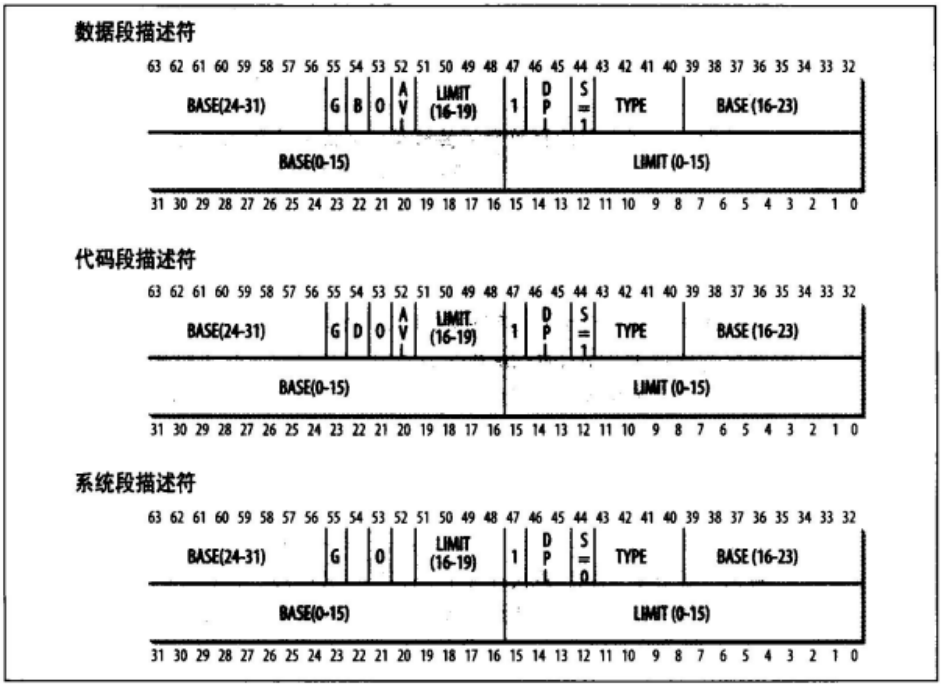

图 2.2: 段描述符格式

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)。

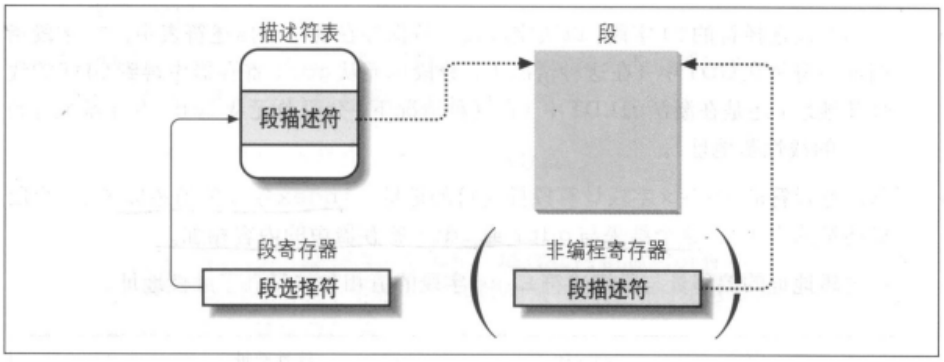

图 2.3: 段选择符和段描述符

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

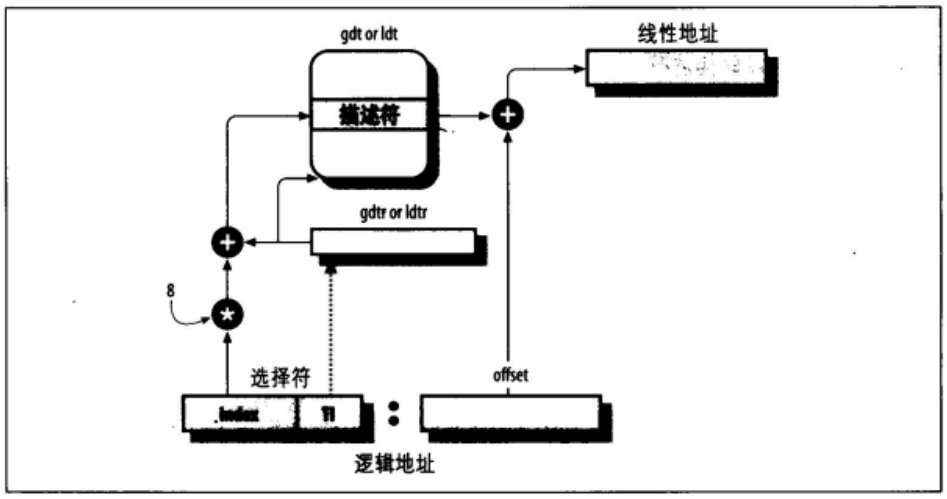

图 2.4: 逻辑地址的转换

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。

```
// for i386 GDT_ENTRIES
#define GDT_ENTRIES 32

// for x86_64 GDT_ENTRIES
#define GDT_ENTRIES 16

// 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 cpu_gdt_table[NR_CPUS][GDT_ENTRIES];

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

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

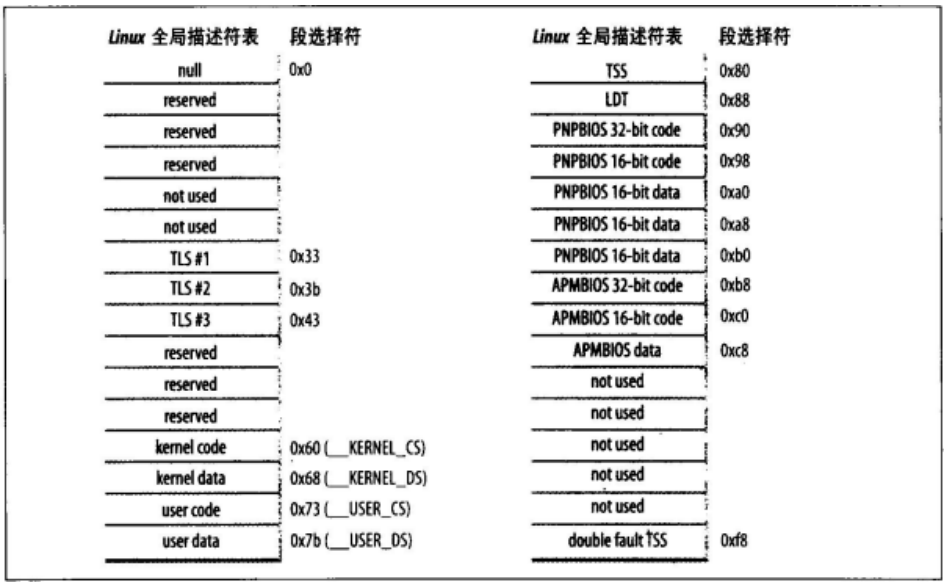

图 2.5: 全局描述符表

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。

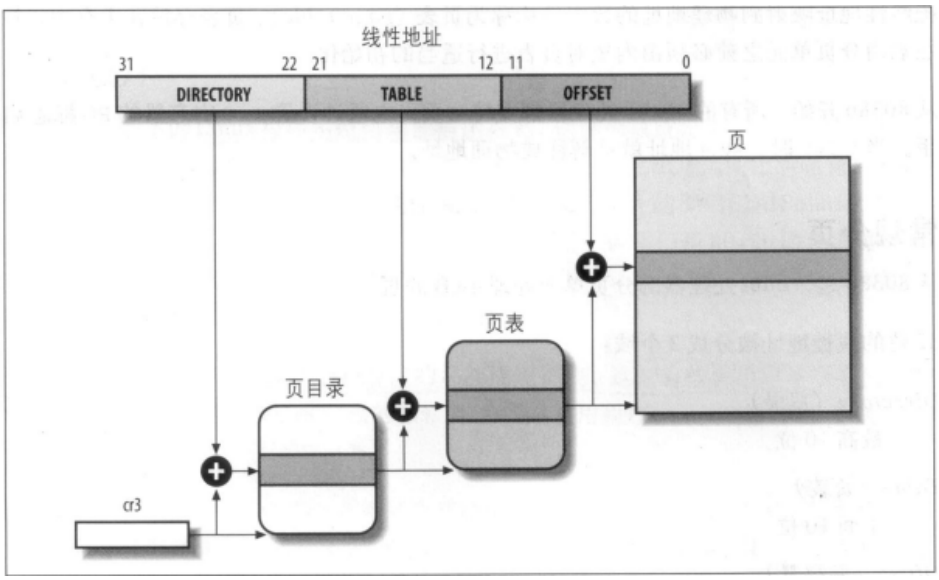

图 2.6: 80x86 处理器分页

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