Table of Contents

Linux内核读书笔记

目录

前言

原文出处：Linux内核读书笔记作者：crazyingbird

Linux内核读书笔记

记录了读书过程当中的一些收获、心得体会，与大家分享交流。持续更新中。。。

第一章 绪论（1）

一、Linux与其它类Unix内核的比较：

单块结构的内核：由几个逻辑上独立的成分构成，单块结构，大多数据商用Unix变体也是单块结构；

编译并静态连接的传统Unix内核：Linux能自动按需动态地装载和卸载部分内核代码（模块），而传统Unix内核仅支持静态连接；

内核线程：Linux以一种十分有限的方式使用内核线程来周期性地执行几个内核函数，而一些Unix内核则本身被组织成一组内核线程；

多线程应用程序支持：Linux定义了自己的轻量级进程版本，并以此来实现对多线程应用程序的支持，而商用Unix则都是基于内核线程来作为多线程应用程序的执行环境；

抢占式内核：Linux2.6提供了“可抢占的内核”的编译选项，当采用这种编译方式来编译内核时，Linux2.6可以随意交错执行处于特权模式的执行流，而一些传统的、通用的Unix，如Solaris则是完全的抢占式内核；

多处理器支持：一些Unix内核变体都利用了多处理器系统，Linux2.6支持不同存储模式的对称多处理，不仅可以使用多处理器，同时每个处理器可以毫无区别地处理任何一个任务；

注："对称多处理"（Symmetrical Multi-Processing）又叫SMP，是指在一个计算机上汇集了一组处理器(多CPU),各CPU之间共享内存子系统以及总线结构。

文件系统：Linux标准文件系统支持多种不同类型的文件系统，由于采用了面向对象虚拟文件系统技术，外部文件系统可以很容易移植到Linux内核上；

注：虚拟文件系统（VFS）是物理文件系统与服务之间的一个接口层，它对Linux的每个文件系统的所有细节进行抽象，使得不同的文件系统在Linux核心以 及系统中运行的其他进程看来，都是相同的。严格说来，VFS并不是一种实际的文件系统。它只存在于内存中，不存在于任何外存空间。VFS在系统启动时建 立，在系统关闭时消亡。

STREAMS：大部分Unix内核均包含STREAMS I/O子系统，作为编写设备驱动程序、终端驱动程序及网络协议的首选接口，但Linux无类似的子系统；

二、硬件依赖性：

Linux试图在硬件无关的源代码与硬件相关的源代码之间保持清晰的界限，为此，Linux为不同的硬件平台作了不同的支持，目前共对23种不同的硬件平台类型作了专门的支持。

三、操作系统基本概念：

当操作系统启动时，内核被装入到RAM中，内核中包含了系统运行所必须的很多核心过程。内核为系统中所有事情提供了主要功能，并决定高层软件的许多特性。

操作系统的两个主要目标：与硬件交互以及为运行在其上的应用程序提供执行环境。

多用户系统：

能并发和独立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”意味着几个应用程序能同时处于活动状态并竞争各种资源；

“独立”意味着每个应用程序能执行自己的任务，而无需考虑其他用户的应用程序在干些什么；

多用户操作系统必须包含的特点：

- 核实用户身份的认证机制；
- 防止有错误的应用程序妨碍到其它应用程序在系统中运行的保护机制；
- 防止有恶意用户程序干涉或窥视其它用户的活动的保护机制；
- 限制分配给每个用户的资源数的记账机制；

以上保护机制依赖与CPU特权模式相关的硬件保护机制。

用户和组：

所有的用户由一个唯一的用户标识符来标识。

为了实现资料的共享，引入用户组，组由唯一的用户组标识符来标识。

任何类Unix操作系统都有一个特殊的root用户，即超级用户，操作系统不对她使用通常的保护机制，可以访问系统中的任何一个文件，并干涉任何一个正在执行的用户程序活动。

进程：

程序执行的一个实例，一个运行程序的“执行上下文”。

Unix是具有抢占式进程的多处理器操作系统。

类Unix操作系统采用进程/内核模式，每个进程都自以为它是系统中唯一的进程，可以独占操作系统所提供的服务。只要进程发出系统调用，硬件就会把特权模式由用户态变成内核态，然后进程以非常有限的目的开始一个内核过程的执行。

内核体系结构：

大部分Unix内核是单块结构：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。

微内核操作系统只需要内核有一个很小的函数集，通常包括几个同步原语、一个简单的调度程序和进程间的通信机制。

微内核操作系统一般比单块内核的效率低，因此操作系统不同层次之间显式的消息传递要花费一定的代价。

微内核操作系统迫使系统程序员采用模块化方法，因为任何操作系统层都是一个相对独立的程序，这种程序必须通过定义明确而清晰的软件接口，以实现与其他层的交互，同时时方便移植。

微内核操作系统比单块内核更充分的利用了RAM。

Linux内核提供了模块。模块是一个目标文件，其代码可以在运行时链接到内核，或从内核上解除链接。这种目标代码通常由一组函数组成，用来实现文件系统、驱动程序或其它内核上层功能。

四、Unix文件系统概述：

文件：

Unix文件是以字节序列组成的信息载体，内核不解释文件的内容。

文件或目录名由除了“/”和“\0”之外的任意ASCII字符序列组成，通常不能超过255个字符。

当标识文件名时，“.”和“..”分别用来标识当前工作目录和父目录。

硬链接和软链接：

包含在目录中的文件名就是一个文件的硬链接，或简称链接。

软链接又称为符号链接，是短文件，这些文件包含有另一个文件的任意一个路径名。

文件类型：

Unix文件可以是下列类型之一：

- 普通文件
- 目录
- 符号链接
- 面向块的设备文件
- 面向字符的设备文件
- 管道和命令管道
- 套接字(Socket)

文件描述符与索引节点：

文件内容不包含任何控制信息。

文件系统处理文件需要的所有信息包含在一个名为索引节点的数据结构中。

索引节点至少提供以下属性：

- 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（包含文件的设备的标识符）
- 在文件系统中识别文件的索引节点号
- 文件拥有者的UID
- 文件的用户组ID
- 几个时间戳，表示索引节点状态改变的时间、最后访问时间及最后修改时间
- 访问权限和文件模式

文件操作的系统调用：

当用户访问一个变通文件或目录文件的内容时，实际是访问硬件设备上的一些数据，因此，文件系统是硬盘分区物理组织的用户级视图。而由于处于用户态的进程不能直接与低层硬件交互，因此，所有实际的文件操作都必须在内核态下进行。

当几个进程同时打开一个文件时，文件系统给每个文件分配一个单独的打开文件对象以及单独的文件描述符。在这种情况下，Unix文件系统对进程在同一文件上发出的I/O操作之间不提供任何形式的同步机制。

对普通Unix文件，可以顺序访问，也可随机访问，而对设备文件和命名管道文件，通常只能顺序访问。

第一章 绪论（2）

五、Unix内核概述：

Unix内核提供了应用程序可以运行的执行环境。因此，内核必须实现一组服务及相应的接口，应用程序使用这些接口，而且通常不会与硬件资源直接交互。

进程/内核模式：

所有标准的Unix内核都仅仅利用了内核态和用户态。

一个程序执行时，大部分时间都处在用户态下，只有需要内核所提供的服务时才切换到内核态。当内核满足了用户程序的请求后，便使程序回到用户态下。

内核本身不是进程，而是进程的管理者。

Unix系统包括几个称为内核线程的特权进程，它们通常在系统启动时创建，并以内核态运行在内核地址空间，一直处于活跃状态直到系统关闭。

进程实现：

每个进程由一个进程描述符表示。

当内核暂停一个进程的执行时，就把几个相关处理器寄存器的内容保存在进程描述符中。当内核决定恢复执行一个进程时，它用进程描述符中合适的字段来装载CPU寄存器，由于程序计数器中所存储的值指向下一条将要执行的指令，所以进程从它停止的地方恢复执行。

Unix内核可以区分很多等待状态，这些等待状态由进程描述符队列实现。每个队列对应一组等待特定事件的进程。

可重入内核：

可重入，意味着若干进程可以同时在内核态下执行，但在每个处理器上只有一个进程在真正运行，其它则可能在等待执行或处于阻塞状态。

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，这样可以提高发出中断的设备控制器的吞吐量。

进程地址空间：

在用户态下运行的进程涉及到私有栈、数据区和代码区。

在内核态下运行的进程，访问内核的数据区和代码区，但使用另外和私有栈。

如果一个程序由几个用户同时使用，则这个程序只被装入内存一次，其指令由所需要它的用户共享，当然，其数据不被共享。

进程间也能共享部分地址空间，以实现一种进程间的通信。

同步和临界区：

实现可重入内核需要利用同步机制，因为如果当内核控制路径对某个内核数据结构进行操作时被挂机，那么，其它内核控制路径就不应当再对该数据结构进行操作。

对全局变量的安全访问通过原子操作来保证。

如果内核支持抢占，如Linux，那么在应用同步机制时，确保进入临界区前禁止抢占，退出临界区时启用抢占。

信号量：

可以把信号量看成一个对象，其组成如下：

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

每个要保护的数据结构都有它自己的信号量，其初始值为1。当内核控制路径希望访问这个数据结构时，它在相应的信号量上执行down()方法，若信号量的当前值不是负数，则允许访问这个数据结构，否则，把执行内核控制路径的进程加入到这个信号量的链表并阻塞该进程。当另一个进程在那个信号量上执行up()方法是，允许信号量链表上的一个进程继续执行。

自旋锁：

自旋锁与信号量非常相似，但没有进程链表；当一个进程发现锁被另一个进程锁着时，它就不停的“旋转”，执行一个紧凑的循环指令直到锁打开。

自旋锁在单处理器环境下是无效的。因为：当内核控制路径试图访问一个上锁的数据结构时，它开始无休止循环，则内核控制路径可能因为正在修改受保护的数据结构而没有机会继续执行，也没有机会释放这个自旋锁，最后的结果可能是系统挂起。

Linux使用一个名为init的特殊系统进程，它在系统初始化的时候被创建。当一个进程终止时，内核改变其所有现有子进程的进程描述符指针，使这些子进程成为init的孩子。Init监控所有子进程的执行，并且按常规发布wait4()系统调用，除掉所有僵死的进程。

物理地址扩展（PAE）分页机制

Intel通过在处理器上把管脚数从32增加到36，以提高处理器的寻址能力，使其达到2^36=64GB，为此，需引入一种新的分页机制。

64GB的RAM被分为2^24个页框，页表项的物理地址字段从20位扩展到24位，每个页表项必须包含12个标志位（固定）和24个物理

地址位（36-12），共36位，因此，每个页表项须从32位扩展到64位（36位>32位，考虑到对齐，因此应将页表项扩大一倍到64位）。

在**4KB**的常规分页情况下，由于每个页表项大小为**64**为，因而，原有**210**大小的页表中，仅能包含**512**个表项，这占用了**32**

位线性地址中的**9**位，同理，由于页目录项与页表项具有同样的结构，高一级的页目录表中也仅能包含**512**个页表项，同样占用

了**32**位线性地址中的**9**位，此时，线性地址剩余位数为：**32**位（总位数）**-12**位（页内偏移量）**-9**位（指示页表中的索引）**-9**位

（指示页目录表中的索引）**=2**位，同时，**Linux**引入了一个页目录指针表（**PDPT**）的页表新级别，由**4**个**64**位表项构成，剩余

的**2**位即用来指向**PDPT**中**4**个项中的一个。

下面**4**张图详细说明了**4**种情况下的页表结构（引自**Wikipedia**）

未启用**PAE**下的**4K**分页的页表结构

未启用**PAE**下的**4M**分页的页表结构**

启用**PAE**下**4K**分页的页表结构**

启用**PAE**下**2M**分页的页表结构

第二章-内存寻址（1）

内存地址：

逻辑地址：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。

线性地址：一个32位无符号整数，也称虚拟地址。

物理地址：用于内存芯片级内存单元寻址，与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。

在多处理器系统中，所有CPU都共享同一内存，这意味着，RAM芯片可以由独立的CPU并发地访问。由于RAM芯片上的读或写操作必须串行地执行，

因此一种所谓内存仲裁器的硬件电路插在总线和每个RAM芯片之间，其作用是如果某一个RAM芯片空闲，就准予一个CPU访问，如果该芯片忙于为

另一个处理器提出的请求服务，就延迟这个CPU的访问。

即使在单处理器上也使用内存仲裁器。因为单处理器系统中包含一个叫做DMA控制器的特殊处理器，而且DMA控制器与CPU并发操作。

硬件中的分段：

一个逻辑地址由一个16位段标识符（或称为段选择符，如下）和一个32位段内偏移组成。

段选择符包含13位的索引号、1位TI表指示器以及2位RPL请求者特权级。

每个段由一个8字节的段描述符表示，段描述符存放在全局描述符表（GDT）或者局部描述符表（LDT）中。通常只定义一个GDT，而每个进程都拥

有自己的LDT。

GDT的第一项总是设为0，这就确保空段选择符的逻辑地址会被认为是无效的，并能引起一个处理器异常。由此，GDT中最大段描述符数目为213-1。

例：如果GDT在0x0020000（这个值保存在gdtr寄存器中），且由段选择符所指定的索引号为2（即段选择符高13位的值为2），则由于GDT第一项为0，

由该段选择符所指定的段描述符地址是0x0020000+2*8或者0x0020010。

逻辑地址转换成相应线性地址：分段单元先检查段选择符的TI字段，决定段描述符的保存位置（GDT或者LDT），若为GDT，分段单元从gdtr寄存器

中得到GDT的线性基址，若为激活的LDT，则分段单元从ldtr寄存器中得到LDT的线性基址；然后，从段选择符的index字段计划段描述符的地址，把

这个地址与逻辑地址中的偏移量字段值相加，得到线性地址。

Linux中的分段：

分段可以给每一个进程分配不同的线性地址空间，而分页则可以把同一线性地址空间映射到不同的物理空间。与分段相比，Linux更喜欢使用分页方式，

因为：当所有进程使用相同的段寄存器值时，内存管理变得更加简单；Linux设计目标之一是可移植到大多数处理器平台上，而RISC体系结构对分段的

支持很有限。

Linux主要使用了四个段：用户代码段、用户数据段、内核代码段、内核数据段。相应的段选择符由宏**USER_CS**、**__USER_DS**、KERNEL_CS、

__KERNEL_DS定义，内核只需把相应宏产生的值装入cs段寄存器即可对相应的段寻址。

当对指向指令或者数据结构的指针进行保存时，内核不需为其设置逻辑地址的段选则符，因为cs寄存器就含有当前的段选择符。

例：当内核调用一个函数时，它执行一条call汇编语言指令，该指令仅指定其逻辑地址的偏移量部分，而段选择符不用设置，因为“在内核态执行”

的段只有内核代码段，由__KERNEL_CS定义。

在单处理器系统中只有一个GDT，而多处理器系统中每个CPU对应一个GDT。

Linux中每一个GDT包含18个段描述符，指向下列段：

- 用户态和内核态下的代码段和数据段共4个；
- 任务状态段，每个处理器有1个；
- 1个包含缺省局部描述符表的段，这个段通常被所有进程共享；
- 3个局部线程存储（TLS）段；
- 3个与高级电池管理相关的段；
- 5个与支持即插即用功能的BIOS服务程序相关的段；
- 1个被内核用来处理“双重错误”异常的特殊TSS段；

注：处理一个异常时可能会引发另一个异常，这咱情况下产生双重错误。

大多数用户态下的Linux程序不使用LDT，这样内核就定义了一个缺省的LDT供大多数进程共享。

LINUX 和 WINDOWS 内核的区别

第二章-内存寻址(2)

硬件中的分页：

32位的线性地址被分成3个域：

高10位：页目录表

中间10位：页表

低12位：页表内偏移

使用二级页表模式的目的在于减少每个进程页表所需RAM数量。如果是一级页表，则需高达220个表项，而二级模式只为进程实际使用的那些虚拟内存区请求页表。

页目录项和页表项有同样的结构，均包含了一些属性字段。

评：段页属性字段的设置很有意义，分段、分页这种将内存结构化、组织化的方式，同时可以增加对某一段或页的属性描述信息，对于内存管理来说很有意义，这种组织内存的方法思路，正如行政区域的划分。

物理地址扩展（PAE）分页机制：

这一部分在我的另一篇博文中有提
到：http://blog.csdn.net/crazyingbird/article/details/7175559

硬件高速缓存：

为了缩小CPU和RAM之间的速度不匹配，引入了硬件调整缓冲内存。

多处理器系统的每一个处理器都有一个单独的硬件高速缓存，它们需要额外的硬件电路，用于保持不同CPU之间的高速缓存内容的同步。只要一个CPU修改了它的硬件高速缓存，它就必须检查同样的数据是否包含在其它的硬件高速缓存中，如果是，它必须通知其它CPU用适当的值对其更新，这种活动叫做高速缓存侦听。这些活动由硬件处理，内核无需关心。

Linux对于所有的页框都启用高速缓存，对于写操作总是采用加回写策略。

转换后援缓冲器：

记录上一次线性地址转换得到的相应的物理地址，以便以后对同一线性地址的引用可以快速地得到转换。

在多处理器系统中，每个CPU都有自己的TLB，称为该CPU的本地TLB，与硬件高速缓存相反，TLB中的对应项不必同步，这是因为运行在现有CPU上的进程可以使同一线性地址与不同物理地址发生联系。

Linux中的分页：

在2.6.10版本及之前，Linux采用三级分页模型，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，Linux取消了页上级目录和页中间目录，仅使用了两级页表。

启用了物理地址扩展的32位系统，Linux取消了上级目录，使用三级页表。

对于64位系统，Linux根据硬件对线性地址位的划分来决定采用三级或者四级页表。

物理内存布局：

Linux内核安装在RAM中从物理地址0x00100000开始的地方，也就是从第二个MB开始。

页框0由BIOS使用，存放加电自检期间检查到的系统硬件。

物理地址从0x000a0000到0x000fffff的范围通常留给BIOS全程，并且映射ISA图形卡上的内部内在。

第一个MB内的其它页框可能由特定计算机模型保留。

在启动过程的早期阶段，内核询问BIOS并了解物理内存的大小，随后，内核建立物理地址映射。

内核可能不会见到BIOS报告的所有物理内存：如果未启用PAE支持来编译，即使有更大的物理内存可供使用，内核也只能寻址4GB大小的RAM。

进程页表：

进程的线性地址空间分成两部分：

从0x00000000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0xc0000000到0xffffffff的线性地址，只有内核态的进程才能寻址；

当进程运行在用户态时，它产生的线性地址小于0xc0000000；当进程运行在内核态时，它执行内核代码，所产生的地址大于等于0xc0000000。但某些情况下，内核为了检索或存放数据必须访问用户态线性地址空间。

内核页表：

内核维持着一组自己使用的页表，驻留在所谓的主内核页全局目录中，系统初始化后，这组页表还从未被任何进程或者任何内核线程直接使用；确切的说，主内核页全局目录的最高目录项部分作为参考模型，为系统中每个普通进程对应的页全局目录提供参考模型。

那么，内核如何初始化自己的页表呢？内核映像刚刚被装入内存后，CPU仍然运行于实模式，所以分页功能没有被启用。

第一阶段，内核创建一个有限的地址空间，包括内核的代码段和数据段、初始页表和用于存放动态数据结构的共128KB大小的空间。这个最小限度的地址空间仅够将内核装入RAM和对其它初始化的核心数据结构。

第二阶段，内核充分利用剩余的RAM并适当建立分页表。

由内核页表所提供的最终映射必须把从0xc0000000，即第四个GB开始的线性地址转化为从0开始的物理地址。

固定映射的线性地址：

固定映射的线性地址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等于线性地址减去0xc000000，而是可以以任意方式建立。因此，每个固定映射的线性地址都映射一个物理内存的页框。内核使用固定映射的线性地址来代替指针变量。

每个固定映射的线性地址都存在线性地址第四个GB的末端。

处理硬件高速缓存：

为了使高速缓存的命中率达到最优化，内核在下列决策中考虑体系结构：

一个数据结构中最常使用的字段放在该数据结构内的低偏移部分，以便它们能够处于高速缓存的同一行中。

注：聚集存储的数据结构大小可能大于行的大小。

当为一大组数据结构分配空间时，内核试图把它们都存放在内存中，以便所有高速缓存行按同一方式使用。

关于“实模式”和“保护模式”

今天整理读书笔记，发现了一个之前没注意到的，或者自己一直忽略的地方：

《深入理解Linux内核》一书中，关于内存寻址这一章，提到了“硬件中的分段”、“Linux中的分段”、“硬件中的分页”和“Linux中的分页”四个概念，所谓的硬件上的分段、分页，是针对CPU在实模式下，即操作系统尚未加载启动之前所采用的内存寻址方式，而软件上的分段、分页，则是CPU在保护模式下，即操作系统启动后所采用的内存寻址方式。

在这里再引出”实模式“和”保护模式“的概念：

x86体系的处理器刚开始时只有**20**根地址线，寻址寄存器是**16**位。可以访问**64K**的地址空间，如果程序要想访问大于**64K**的内存，就需要把内存分段，每段**64K**，用段地址**+**偏移量的方式来访问，这样使**20**根地址线全用上，最大的寻址空间就可以到**1M**字节，这在当时已经是非常大的内存空间了。

实模式将整个物理内存看成分段的区域，程序代码和数据位于不同区域，系统程序和用户程序并没有区别对待，而且每一个指针都是指向实际的物理地址。这样一来，用户程序的一个指针如果指向了系统程序区域或其他用户程序区域，并修改了内容，那么对于这个被修改的系统程序或用户程序，其后果就很可能是灾难性的。再者，随着软件的发展，1M的寻址空间已经远远不能满足实际的需求了。最后，对处理器多任务支持需求也日益紧迫，所有这些都促使新技术的出现。

为了克服实模式下的内存非法访问问题，并满足飞速发展的内存寻址和多任务需求，处理器厂商开发出保护模式。在保护模式中，除了内存寻址空间大大提高；提供了硬件对多任务的支持；物理内存地址也不能直接被程序访问，程序内部的地址(虚拟地址)要由操作系统转化为物理地址去访问，程序对此一无所知。至此，进程(程序的运行态)有了严格的边界，任何其他进程根本没有办法访问不属于自己的物理内存区域，甚至在自己的虚拟地址范围内也不是可以任意访问的，因为有一些虚拟区域已经被放进一些公共系统运行库。这些区域也不能随便修改，若修改就会有出现linux中的段错误，或Windows中的非法内存访问对话框。

进程、轻量级进程和线程的一些点

从内核观点看，进程的目的就是担当分配系统资源（CPU时间、内存等）的实体。

进程是资源管理的最小单位，线程是程序执行的最小单位。在操作系统设计上，从进程演化出线程，最主要的目的就是更好的支持SMP以及减小（进程/线程）上下文切换开销。

最初的进程定义都包含程序、资源及其执行三部分，其中程序通常指代码，资源在操作系统层面上通常包括内存资源、IO资源、信号处理等部分，而程序的执行通常理解为执行上下文，包括对CPU的占用，后来发展为线程。在线程概念出现以前，为了减小进程切换的开销，操作系统设计者逐渐修正进程的概念，逐渐允许将进程所占有的资源从其主体剥离出来，允许某些进程共享一部分资源，例如文件、信号，数据内存，甚至代码，这就发展出轻量进程的概念。Linux内核在2.0.x版本就已经实现了轻量进程，应用程序可以通过一个统一的clone()系统调用接口，用不同的参数指定创建轻量进程还是普通进程。

Linux使用轻量级进程，对多线程应用程序提供更好的支持。两个轻量级进程基本上可以共享一些资源，诸如地址空间、打开的文件等等。只要其中一个修改共享资源，另一个就立即查看这种修改。当然，当两个线程访问共享资源时必须同步它们自己。

在Linux中，一个线程组基本上就是实现了多线程应用的一组轻量级进程。

第三章-进程(1)

进程、轻量级进程和线程：

从内核观点看，进程的目的就是担当分配系统资源（CPU时间、内存等）的实体。

进程是资源管理的最小单位，线程是程序执行的最小单位。在操作系统设计上，从进程演化出线程，最主要的目的就是更好的支持SMP以及减小（进程/线程）上下文切换开销。

最初的进程定义都包含程序、资源及其执行三部分，其中程序通常指代码，资源在操作系统层面上通常包括内存资源、IO资源、信号处理等部分，而程序的执行通常理解为执行上下文，包括对CPU的占用，后来发展为线程。在线程概念出现以前，为了减小进程切换的开销，操作系统设计者逐渐修正进程的概念，逐渐允许将进程所占有的资源从其主体剥离出来，允许某些进程共享一部分资源，例如文件、信号，数据内存，甚至代码，这就发展出轻量进程的概念。Linux内核在2.0.x版本就已经实现了轻量进程，应用程序可以通过一个统一的clone()系统调用接口，用不同的参数指定创建轻量进程还是普通进程。

当一个进程创建时，它几乎与父进程相同。它接受父进程地址空间的一个（逻辑）拷贝，并从进程创建系统调用的下一条指令开始执行与父进程相同的代码。尽管父子进程可以共巷含有程序代码的页，但是它们各自有独立的数据拷贝（栈和堆），因此子进程对一个内存单元的修改对父进程是不可见的。

Linux使用轻量级进程，对多线程应用程序提供更好的支持。两个轻量级进程基本上可以共享一些资源，诸如地址空间、打开的文件等等。只要其中一个修改共享资源，另一个就立即查看这种修改。

在Linux中，一个线程组基本上就是实现了多线程应用的一组轻量级进程。

进程描述符：

为了管理进程，内核必须对每个进程所做的事情进行清楚的描述。例如：内核必须知道进程的优先级，它是正在CPU上运行还是因为某些事件而被阻塞，给它分配了什么样的地址空间，允许它访问哪个文件等等，这正是进程描述符的作用。

进程状态：

可运行状态：进程要么在CPU上执行，要么准备执行。

可中断的等待状态：进程被挂起，直到某个条件变为真。

不可中断的等待状态：与可中断等待状态类似，但有一个例外：把信号传递到睡眠进程不能改变它的状态。

暂停状态：进程的执行被暂停。

跟踪状态：进程的执行已由Debugger程序暂停。当一个进程被另一个进程监控时，任何信号都可以把个进程置于跟踪状态。

僵死状态：进程的执行被终止，但父进程还没有发布wait4()或waitpid()系统调用来返回有关死忘进程的信息。在发布wait()类系统调用前，内核不能丢弃包含在死进程描述符中的数据，因此父进程可能还需要它。

僵死撤消状态：最终状态，由于父进程刚发出wait4()或waitpid()系统调用，因而进程由系统删除。为了防止其他执行线程在同一个进程上也执行wait()类系统调用，而把进程的状态由僵死状态改为僵死撤消状态。

标识一个进程：

内核对进程的大部分引用都是通过进程描述符指针进行的。

类Unix操作系统允许用户使用一个叫做进程标识符(PID)的数来标识进程，PID存放在进程描述符的pid字段中，PID被顺序编号，新创建进程的PID通常是前一个进程的PID加1。不过，PID的值有一个上限，当内核使用的PID达到这个上限值的时候必须开始循环使用已闲置的小PID号。缺省情况下，最大的PID号是32767。

由于循环使用PID号，内核必须通过管理一个pidmap_array位图来表示当前已分配的PID号和闲置的PID号。因为一个页框包含32768个位，所以32位体系结构中pidmap_array位置存放在一个单独的页中。

Linux把不同的PID与系统中每个进程或轻量级进程相关联。对于线程组，一个线程组中的所有线程使用和该线程组的领头线程相同的PID，即该组中第一个轻量级进程的PID，它被存入进程描述符的tgid字段。getpid()系统调用返回当前进程的tgid值而不是pid值。

进程间的关系：

进程0和进程1是由内核创建的，进程1（init）是所有进程的祖先。

进程描述符中表示进程亲属关系的字段的描述如下表：

字段名	说明
real_parent	指向创建了P的进程的描述符，如果P的父进程不再存在，就指向进程1的描述符
Parent	指向P的当前父进程（这种进程的子进程终止时，必须向父进程发信号）。它的值通常与real_parent一致，但也可以不同
Children	链表的头部，链表中的所有元素都是P创建的子进程
Sibling	指向兄弟进程链表中的下一个元素或前一个元素的指针，这些兄弟进程的父进程都是P

进程之间还存在其它关系：一个进程可能是一个进程组或登陆会话的领头进程，也可能是一个线程组的领头进程，它还可能跟踪其他进程的执行。

如何组织进程：

运行队列链表把处于TASK_RUNNING状态的所有进程组织在一起，但由于对处于暂停、僵死、死亡状态进程的访问比较简单，Linux并没有为处于TASK_RSTOPPED、EXIT_ZOMBILE或者EXIT_DEAD状态的进程建立专门的链表。

对于处于TASK_INTERRUPTIBLE或TASK_UNINTERRUPTIBLE状态的进程，根据不同的特殊事件被细分为许多类，每一类都对应某个特殊事件，引入等待队列。

等待队列表示一组睡眠的进程，当某一条件为真时，由内核唤醒它们。

等待队列由双向链表实现。因为等待队列是由中断处理程序和主要内核函数修改的，因此必须对其双向链表进行保护以免对其进行同时访问，因为同时访问会导致不可预测的后果。同步是通过等待队列头中的lock自旋锁来达到的。

如果有两个或者多个进程在等待互斥地访问某一资源时，由内核有选择地唤醒，而非互斥进程总是由内核在事件发生时唤醒。

因为所有的非互斥进程总是在双向链表的开始位置，而所有的互斥进程在双向链表的尾部，所以内核总是先唤醒非互斥进程然后再唤醒互斥进程。一个等待队列中同时包含互斥进程和非互斥进程的情况是非常罕见的。

第三章-进程(2)

进程切换：

为了控制进程的执行，内核必须有能力挂起正在CPU上运行的进程，并恢复以前挂起的某个进程的执行，这种行为被称为进程切换、任务切换或上下文切换。

硬件上下文：

尽管每个进程可以拥有属于自己的地址空间，但所有进程必须共享CPU寄存器。因此，在恢复一个进程的执行之前，内核必须确保每个寄存器装入了挂起进程时的值。

进程恢复执行前必须装入寄存器的一组数据称为硬件上下文。硬件上下文是进程可执行上下文的一个子集，因为可执行上下文包含进程执行时需要的所有信息。在Linux中，进程硬件上下文的一部分存放在任务状态（TSS）段，而剩余部分存放在内核态堆栈中。进程切换只发生在内核态。

执行进程切换：

从本质上说，每个进程切换由两步组成：

1、切换页全局目录以安装一个新的地址空间

2、切换内核态堆栈和硬件上下文

创建进程：

Unix操作系统依赖进程创建来满足用户的需求。

现代Unix内核引入了三种不同机制解决进程创建问题：

1、写时复制技术，允许子进程读相同的物理页。只要两者中有一个试图写一个物理页，内核就把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、轻量级进程允许父子进程共享每个进程在内核的很多数据结构

3、vfork()系统调用创建的进程能共享其父进程的内存地址空间。为了防止父进程重写子进程需要的数据，阻塞父进程的执行，一直到子进程退出或执行一个新的程序为止。

Clone()、fork()及vfork()系统调用：

在Linux中，轻量级进程是由名为clone()的函数创建的。clone()是在C语言库中定义的一个封装函数，它负责建立新的轻量级进程的堆栈，并调用对编程者隐藏的clone()系统调用。

传统的fork()以及vfork()系统调用在Linux中也是用clone()实现的。

do_fork()函数负责处理clone()、fork()和vfork()系统调用。

内核线程：

现代操作系统将一些重要的任务，如刷新磁盘高速缓存，交换出不用的页框，维护网络连接等，委托给内核线程，内核线程不受不必要的用户态上下文拖累。

在Linux中，内核线程在以下几方面不同于普通进程：

1、内核线程只运行在内核态，而普通进程既可以运行在内核态，也可以运行在用户态。

2、因为内核线程只运行在内核态，它只使用大于PAGE_OFFSET的线性地址空间，而不管在用户态还是内核态，普通进程可以使用4GB的线性地址空间。

kernel_thread()函数创建一个新的内核线程，它接受的参数有：所要执行的内核函数的地址、要传递给函数的参数、一组clone标志。该函数本质上调用do_fork()。

所有的进程的祖先叫做进程0，idle进程，或者因为历史原因叫做swapper进程，它是在Linux初始化阶段从无到有创建的一个内核线程。stark_kernel()函数初始化内核需要的所有数据结构，激活中断，创建另一个叫进程1的内核线程（init进程）。新创建内核线程的PID为1，并与进程0共享进程所有的内核数据结构。

创建init进程后，进程0执行cpu_idle()函数，该函数本质上是在开中断的情况下重复执行hlt汇编语言指令。只有当没有其它进程处于TASK_RUNNING状态时，调度程序才选择进程0。

在多处理器系统中，每个CPU都有一个进程0.只要打开电源，计算机的BIOS就会启动某一个CPU，同时禁用其它CPU。运行在CPU0上的swapper进程初始化内核数据结构，然后激活其它CPU，并通过copy_process()函数创建另外的swapper进程，把0传递给新创建的swapper进程作为它们的新PID。此外，内核把适当的CPU索引赋给内核所创建的每个进程的thread_info描述符的cpu字段。

由进程0创建的内核线程执行init()函数，init()依次完成内核初始化。init()调用execve()系统调用装入可执行程序init，结果，init内核线程变为一个普通进程，且拥有自己的每进程内核数据结构。在系统关闭之前，init进程一直存活，因为它创建和监控在操作系统外层执行的所有进程的活动。

撤消进程：

进程终止的一般方式是调用exit()库函数，该函数释放C函数库所分配的资源，执行编程者所注册的每个函数，并结束从系统回收进程的那个系统调用。

进程终止：

在Linux2.6中有两个终止用户态应用的系统调用：

1、exit_group()系统调用，它终止整个线程组，即整个基于多线程的应用。do_group_exit()是实现这个系统调用的主要内核函数。

2、exit()系统调用，它终止某一个线程，而不管该线程所属线程组中的所有其它进程。do_exit()是实现这个系统调用的主要内核函数。

进程删除：

Unix允许进程查询内核以获得其父进程的PID，或者其任何子进程的执行状态，包括终止状态。因此，不允许Unix内核在进程一终止后就丢弃包含在进程描述符字段中的数据，只有父进程发出了与被终止进程相关的wait()类系统调用之后，才允许这样做。这就是引入僵死状态的原因：尽管从技术上来说进程已死，但必须保存它的描述符，直到父进程得到通知。

如果发生父进程在子进程结束之前结束的情况，系统中会到处是僵死的进程，而且它们的进程描述符会永久占据着RAM，因此，必须强迫所有的孤儿进程成为init进程的子进程。这样，当init进程用wait()类系统调用检查其合法的子进程终止时，就会撤消僵死进程。

第四章-中断和异常(1)

中断（interrupt）通常被定义为一个事件，该事件改变处理器执行的指令顺序。这样的事件与CPU芯片内外部硬件电路产生的电信号相对应。

中断通常分为同步（synchronous）中断和异步（asynchronous）中断：

同步中断是当指令执行时由CPU控制单元产生的，之所以称为同步，是因为只有在一条指令终止执行后CPU才会发出中断。

异步中断是由其他硬件设备依照CPU时钟信号随机产生的。

在Intel微处理器手册中，把同步和异步中断分别称为异常（exception）和中断（interrupt）。

中断是由间隔定时器和I／O设备产生的，而异常是由程序的错误产生的，或者是由内核必须处理的异常条件产生的。

中断信号的作用：

中断信号提供了一种特殊的方式，使处理器转而去执行正常控制流之外的代码。当一个中断信号达到时，CPU必须停止它当前正在做的事情，并且切换到一个新的活动。为了做到这一点，就要在内核态堆栈保存程序计数器的当前值，并把与中断类型相关的一个地址放进程序计数器。

中断处理与进程切换有一个明显的差异：由中断或异常处理程序所执行的代码不是一个进程，更确切的说，它是一个内核控制路径，代表中断发生时正在运行的进程执行。作为一个内核控制路径，中断处理程序比一个进程要轻，中断的上下文很少，建立蒙昧无知中止中断处理所需要的时间很少。

中断处理是由内核执行的最敏感的任务之一，因为它必须满足以下约束：

当内核正打算去完成一些别的事情时，中断随时会到来。因此，内核的目标就是让中断尽可能快的处理完，尽其所能把更多的处理向后推迟。内核响应中断后需要进行的操作分为两部分：关键而紧急的部分，内核立即执行； 其余推迟的部分，内核随后执行。

因为中断随时会到来，所以内核可能正在处理其中一个中断时，另一个中断又发生了。因此，中断处理程序必须编写成使相应的内核控制路径能以嵌套的方式执行。当最后一个内核控制路径终止时，内核必须能恢复被中断进程的执行，或者，如果中断信号已导致了重新调度，内核能切换到另外的进程。

尽管内核在处理前一个中断时可以接受一个新的中断，但在内核代码中还是存在一些临界区，在临界区中，中断必须被禁止。必须尽可能地限制这样的临界区，因为，根据以前的要求，内核，尤其是中断处理程序，应该在大部分时间内以开中断的方式运行。

中断和异常：

Intel文档把中断和异常分为以下几类：

中断：

可屏蔽中断（maskable interrupt）：一个屏蔽的中断只要还是屏蔽的，控制单元就忽略它。

非屏蔽中断（nonmaskable interrupt）：非屏蔽中断总是由CPU辨认。

异常：

处理器探测异常（processor-detected exception）：当CPU执行指令时探测到一个反常条件所产生的异常，可分为三组：

故障（fault）：通常可以纠正，一旦纠正，程序就可以不失连贯的情况下重新开始。

陷阱（trap）：主要用于调试程序。在陷阱指令执行后立即报告，内核把控制权返回给程序后就可以继续它的执行而不失连贯性。

异常中止（abort）：发生一个严重错误，控制单元出了问题，不能在eip寄存器中保存引起异常的指令所在的确切位置。异常中止用于报告严重的错误，如硬件故障等。由控制单元发送的这个中断信号是紧急信号，用来把控制权切换到相应的异常中止处理程序，这个异常中止处理程序除了强制受影响的进程终止外，没有别的选择。

编程异常（programmed exception）：在编程者发出请求时发生。控制单元把编程异常作为陷阱来处理，编程异常通常也叫做软中断（software interrrupt）。这样的异常有两个用途：执行系统调用及给调试程序通报一个特定的事件。

每个中断和异常是由0～255之间的一个数来标识。Intel把这个8位无符号整数叫做一个向量（vector）。非屏蔽中断的向量和异常的向量是固定的，而可屏蔽中断的向量可以通过对中断控制器编程来改变。