进程调度

多任务与进程调度

- 多任务操作系统能同时并发地交互执行多个进程的操作系统，多任务操作系统都能使多个进程处于堵塞或者睡眠状态。也就是说实际上不被投入执行直到工作确实准备就绪，因此现代Linux系统也许有100进程在内存，但是只有一个处于可运行状态
- 多任务系统可以划分为两个：非抢占式多任务和抢占式多任务
    - 非抢占式多任务：在非抢占式多任务模式下，除非进程自己停止运行，否则他会一直执行
    - 抢占式多任务：Linux也是采用这种模式，在此模式下，由调度程序来决定什么时候停止一个进程的运行

调度需要考虑的策略

IO

进程可以被分为IO消耗型和处理器消耗型

- IO消耗型的就是通常处于可运行状态，但都是运行短短的一小会，比如说是多数用户的GUI界面
- 处理器耗费型进程把时间大多用在执行代码上，除非被抢占，否则他们通常一直不停的运行，比如MATLAB
- 也有同时结合两者的，比如打字输入程序

进程优先级

Linux采用了两种不同的优先级范围，分别是nice值和实时优先级

- nice值
    - 他的范围为-20到19
    - 越大的nice值意味着更低的优先级
    - 在Linux系统中，nice值代表的是时间片的比例
- 实时优先级
    - 默认情况下它的变化范围为0到99.
    - 越高的实时优先级数值意味着进程优先级越高
    - 任何实时进程的优先级都高于普通进程

时间片

- 对于Linux来说,Linux的CFS调度器没有直接分配时间片到进程，他是将处理器的使用比划分给了进程。这样一来，进程所获得的处理器时间其实是和系统负载密切相关的，这个比例还会进一步受到nice值的影响
- CFS调度器其抢占时机取决于新的可运行程序消耗了多少处理器使用比，如果消耗的使用比比当前进程小，则新进程立刻投入运行，抢占当前进程，否则将推迟其运行

Linux的调度算法：CFS

CFS综述

Linux调度器是以模块的方式提供的，其提供的算法叫做CFS调度。

CFS调度出发点基于一个简单的概念：进程调度效果应该如同系统具备一个理想中的完美多任务处理器

- 完美的多任务处理器应该是这样：我们能在10ms内同时运行两个进程，他们各自使用处理器一半的能力

CFS利用nice值，他作为权重，去平衡对应的调度。

- 越高的nice值权重越低，运行的越慢
- 越低的nice值权重越高。-1--20为root权限才能进行分配的值
- 作为权重的体现如下
    - 假设有两个程序的nice值分别是10和15，计算方法如下：
        - 先获得相对比例，以0为基础：相对为0和5
        - 再看时间片：假设为20ms，那么就以nice值为0的为基准，通过公式计算获取nice值。
        - 然后根据nice值的比例去计算对应分配的处理器时间

CFS并不是一个完美的调度器，当运行任务的数量趋近于无限的时候，他们各自所获得的处理器使用比和时间线都将趋于0，因此CFS为此引入每个进程获得的时间片底线，这个底线被称之为最小粒度。默认情况下这个值为1ms

Linux调度实现有非常重要的四个部分

时间记账

- CFS里面实际上不存在时间片的概念 。当每次系统时钟节拍发生的时候，时间片都会被减少一个节拍周期。当一个进程的时间片被减少到0时，它就会被另外一个尚未减少到0的时间片可运行进程抢占
- CFS会保存一个虚拟时间：vruntime变量存放进程的虚拟运行时间。vruntime可以准确的测量给定进程的运行时间，而且可知道谁应该是下一个被运行的进程。他就是通过nice值和对应的权重，cpu运行时间计算出来的

进程选择

- CFS使用红黑树来组织可运行进程队列，并利用其迅速找到最小vruntime值的进程。他是一颗自平衡二叉树

调度器入口

- 进程调度的主要入口点函数是schedule(),他可以选择哪个进程可以运行，然后投入运行
- schedule()通常都需要和一个具体的调度类相关联（一般都是CFS类）
- 然后他会调用pick-next-task()会以优先级为序，从高到低，依次检查每一个调度类，并且从最高优先级的调度类中选择贵高优先级的进程

睡眠和唤醒

- 休眠的进程处于一个特殊的不可执行的状态
- 内核的基本操作是：进程把自己标记成为休眠状态，从可执行的红黑树中移出，唤醒的过程则刚好相反，进程被设置为可执行状态，然后再从等待队列移到可执行的红黑树中

休眠

- 等待队列
  - 休眠通过等待队列来处理，等待队列是由等待某些时间发生的进程组成的简单链表。具体操作如下
    - 创建一个等待队列的项
    - 把自己加入到队列中
    - 将进程变更为TASK_INTERRUPTIBLE或TASK_UNINTERRUPTIBLE
    - 如果状态被设置为TASK_INTERRUPTIBLE则信号唤醒进程
    - 当进程被唤醒的时候，他会再次检查条件是否为真，如果不是他会再一次调用schedule()并且一直重复这个操作

唤醒

- 唤醒通过函数wake_up进行
- 他将函数设置为TASK_RUNNING状态
- 然后将他放入红黑树中，如果被唤醒的进程优先级比当前正在执行的进程优先级高，还要设置need_resched标志

抢占和上下文切换

上下文切换

上下文切换，也就是从一个可执行进程切换到另外一个可执行进程，他完成了两项基本工作

- 首先把虚拟内存上一个进程映射到新的进程中国
- 然后从上一个进程的处理器状态切换到新进程的处理器状态。这包括保存，恢复栈信息和寄存器信息还有其他和结构体相关的信息，都必须以每个进程为对象进行保存

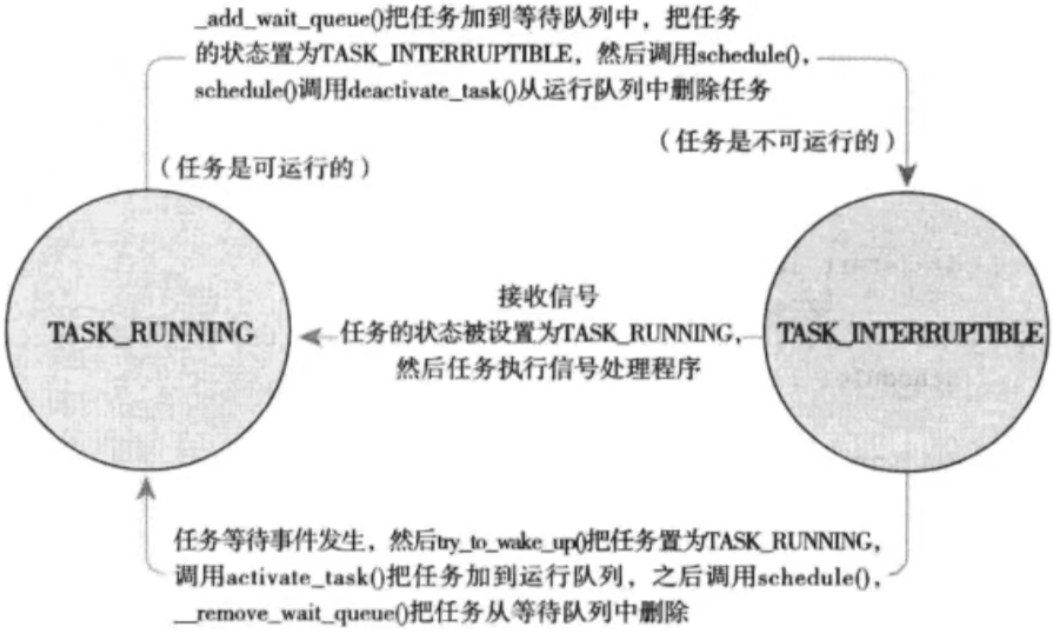

图 4-1　休眠和唤醒

- 内核必须知道什么时候调用schedule，内核提供了一个need-resched标志来表明是否需要重新执行一次调度。他在thread-info结构体里面，用一个特别的标志变量中的一位来表示

用户抢占

- 当内核即将返回用户空间的时候，如果need-resched标志被设置，会导致schedule()被调用，此时就会发生用户抢占。
- 在内核空间返回的时候，因为他知道自己是安全的，既然它可以去选择当前进程，那也可以选择一个新的也可以

内核抢占

Linux支持完整的内核抢占

- 首先Linux为每个进程的thread-info引入preempt-count计数，该计数器的初始值为0，每当使用锁时候数值加1，释放锁的时候数值-1，当数值为0的时候，内核就可以被抢占
- 内核会检查上述两个东西的值，如果会有的话，说明有一个更加重要的任务需要执行并且可以安全的抢占，此时调度程序会被启用

内核抢占一般发生在

- 中断程序正在执行，且返回内核空间之前
- 内核代码再一次具有抢占性质的时候
- 如果内核中任务显式的调用了schedule
- 如果内核中任务阻塞

强制绑定与放弃机制

Linux调度程序提供处理器绑定机制：这些进程无论如何必须在这个处理器上运行

Linux也提供了一种显式的系统调用，他是通过将进程从活动队列中移到过期队列实现的

系统调用

在现在操作系统中，内核提供了用户进程和内核进行交互的一组接口，这些接口让应用程序首先的访问硬件设备，提供了创建新进程并与已有进程进行通信的机制，也提供了申请操作系统其他资源的能力

与内核通信

系统调用在用户空间进程和硬件设备之间添加了一个中间层。该层的主要作用有三个

- 为用户空间提供了一种硬件的抽象接口
- 系统调用保证了系统的稳定和安全
- 每个进程都运行在虚拟系统中，为了提供一层公共接口

在Linux中，系统调用是用户空间访问呢的唯一手段。除了异常和陷入外，他是内核唯一的合法接口

API，POSIX，C库

一般情况下，应用程序通过在用户空间实现的应用编程接口API而不是直接通过系统调用来编程，他们的关系大致如下

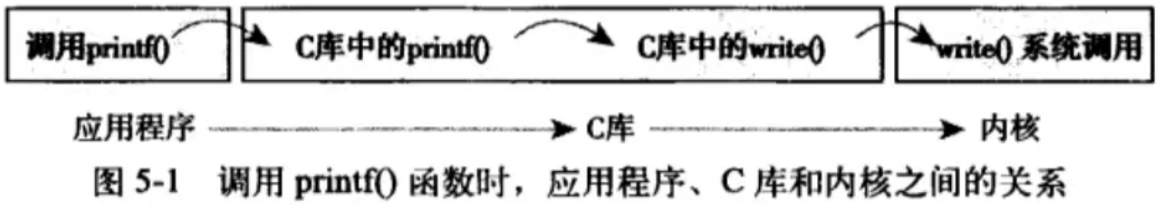

图 5-1　调用 printf() 函数时，应用程序、C 库和内核之间的关系

系统调用

什么是系统调用

要访问系统调用（system call）通常通过C库中的定义函数调用来进行。

在Linux系统中每个系统调用被赋予一个系统调用号，这样通过这个独一无二的号就可以关联系统调用。当用户空间的进程执行一个系统调用的时候，这个系统调用号就用来指明到底是要执行那个系统调用，进程不会体积系统调用的名字

Linux也有一个未实现的系统调用——sys_ni_syscall()他除了返回-ENOSYS外不做任何其他工作，这个错误号就是专门针对无效的系统调用设计的

为什么需要系统调用

用户空间的程序无法直接执行内核代码，它们不能直接调用内核空间的函数，因为内核驻留在受保护的空间上

所以我们在调用的时候，应用程序就会通过某种方式通知系统，告诉内核自己需要执行一个系统调用，希望系统切换到内核态，这样内核就可以代表应用程序在内核空间执行系统调用

通知内核的机制就是靠软中断实现的：通过引发一个异常来促使系统切换到内核态去执行异常处理程序——也就是系统调用处理程序。

在传入的时候，除了陷入内核，还需要传入系统调入好，和一些外部的参数，这些都放在寄存器里面，用单独的寄存器指向他

中断和中断处理

中断

中断使得硬件得以发出通知给处理器，不同设备对应的中断都不同，每一个中断都对应的唯一的数字标志

中断处理程序

在响应一个特定的中断的时候，内核会执行一个函数，该函数叫做中断处理程序/中断服务例程。产生中断的每个设备都有一个相应的中断处理程序

在Linux中，中断处理程序就是普通的C函数

中断随时可能发生，因此中断处理程序也就可以随时执行，所以必须保证中断处理程序能够快速执行，这样才能保证尽可能快的恢复中断代码的运行。因此操作系统能快速处理中断很重要，中断程序在尽可能短的时间内完成运行也同样重要

上半部和下半部

既然既要又要所以我们把中断处理程序分为两个部分——上半部和下半部

- 上半部在接收到应答的时候做快速的恢复
- 下部分完成一些可以稍微推后的工作

中断上下文

- 当执行一个中断处理程序的时候，内核处于中断上下文。
- 因为没有后备进程，所以中断上下文不可以睡眠，如果一个函数睡眠，就不能在中断处理程序中使用它
- 中断上下文的代码应该迅速简单，尽量不使用循环去处理对应的工作
- 有一点非常重要，中断处理程序打断了原先运行的代码，所以必须要简单，然后把一些稍后能做的工作都放在后面
- 中断处理程序是拥有自己的栈的，默认为一页，4kb
- 你的中断处理程序不必关心栈如何设置，或者内核栈的大小是多少，总而言之，尽量节约内核栈空间

中断处理机制的实现

中断处理系统在Linux中的实现是非常依赖于体系结构的。实现依赖于处理器，所使用的中断控制器的类型，体系结构的设计和机器本身

中断的具体过程如下所示

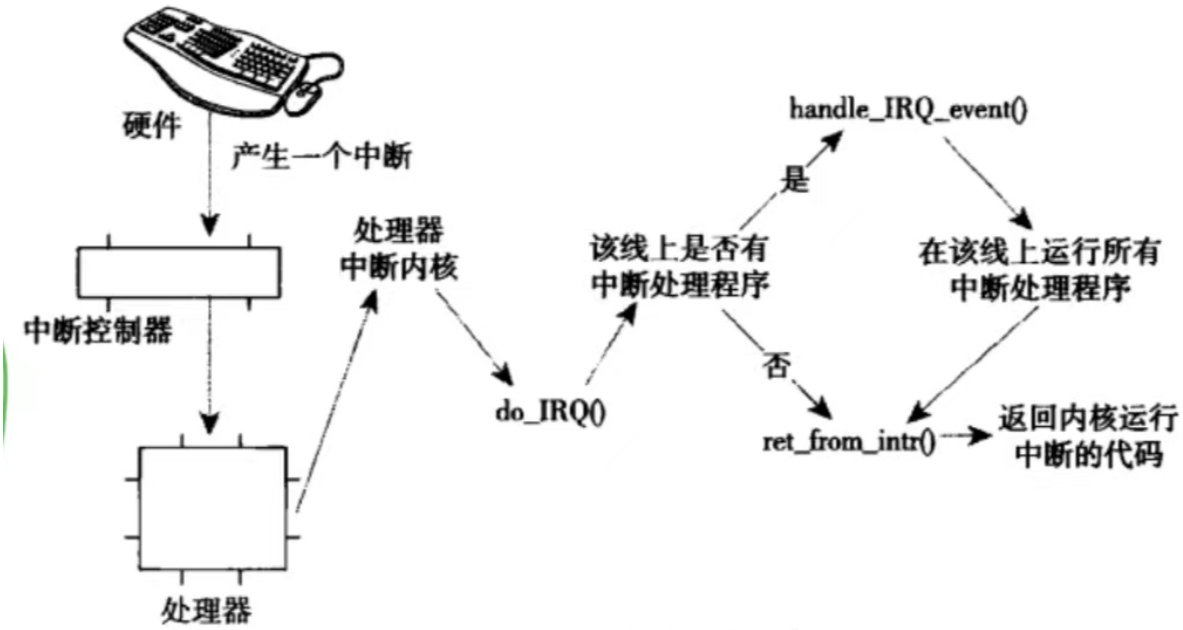

图 7-1　中断从硬件到内核的路由

- 在内核中，中断的旅程开始于预定义的入口点，这类似于系统调用通过预定义的异常句柄进入内核
  - 对于每条中断线，处理器都会跳到对应的一个唯一的位置
- 然后内核调用函数do IRQ()从这里开始，大多数中断处理代码使用C编写的了
- 接下来，需要确保这条中断线上有一个有效的处理程序，如果没有就需要注册一个
  - 首先因为处理器禁止中断，那么注册的时候也需要注意指定禁止标志

    - 然后对每个线都进行一次，如果非共享的话就执行一次就退出循环
    - 然后清理之后返回内核中断的代码，如果设置了need-resched，而且内核要返回内核空间，那么就调用schedule()
- 在schedule调用后，如果没有挂起的工作，一切就会被恢复，内核恢复到曾经中断的点

中断控制

Linux内核提供了一组接口用于操作机器上的中断状态，这些接口为我们提供了能够禁止当前处理器的中断系统或屏蔽掉整个机器一整条线的中断线的能力

一般来说，控制中断系统的原因归根结底是需要提供同步。通过禁止中断，可以确保某个中断处理程序不会抢占当前的代码。此外，禁止中断还可以禁止内核抢占

完全禁止激活中断

我们可以使用local_irq_disable()和local_irq_enable()这两个语句进行中断的禁止和激活。这两个函数通常使用汇编语言实现

当我们静止完成中断之后也需要恢复到原来的值，不能一直静止对应的中断

禁止指定的中断线

我们也可以选择去禁止其中一条中断线，我们也可以屏蔽其中一条。当然我们可以能先禁止中断的传递。Linux提供了四个接口

这些函数是可以嵌套的，如同加锁，而且不会睡眠

中断系统的状态

使用宏irqs_disable()可以了解中断的处理状态