openEuler内核编程

课程讲稿

第6章 第2讲

《内核定时器》

软件所制

第6章 第2讲内核定时器

**学时：**1学时

**教学目的：**学习内核定时器的原理，了解定时器的竞争条件，学习定时器在实际应用中的作用。

**课程时间线：**

定时器的概念和原理

定时器的竞争条件

基于定时器的应用

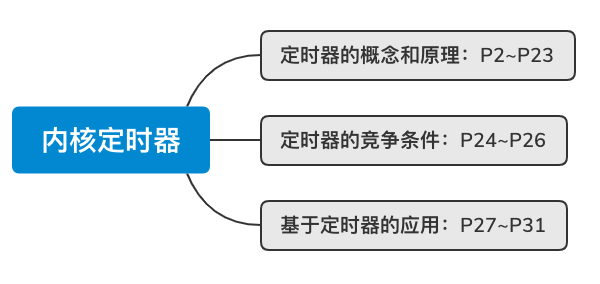
25 min

35 min

45 min

**课外参考读物：**

**知识框图：**

****

**PPT讲稿：**



这一讲的内容是Linux内核定时器。



课程内容分类三个方面。第一是定时器的概念和原理，第二是定时器的竞争条件，第三是讲解基于定时器的应用。



通过上一节课的讲解我们知道计算机内部有一个时钟，每一次计算机启动，这个时钟便从零开始计时。除了计算机启动按顺序执行和用户手动去执行的软件，还有一部分软件是需要定期执行的。比如按时启动的闹钟，每隔一段时间自动清理计算机垃圾的程序。这一部分程序都需要一种叫做定时器的工具来完成计时工作。具体来说定时器是一种软件功能，即在将来的某一时刻调用用户指定的功能。这里的术语超时（timeout）指的是申请的程序运行时间段结束的那一个时刻。

定时器主要分为两种，一种是动态定时器，这一种定时器并不会循环运行，一旦超时就会自动销毁。在内核中，动态定时器不断地被创建和销毁，这也是它被称为动态定时器的原因。另一种定时器是间隔定时器，所谓间隔定时器就是指定时器采用“间隔”值（interval）来作为计时方式，当定时器启动后，间隔值interval将不断减小，当interval值减到0时，我们就说该间隔定时器到期。与内核动态定时器相比，二者最大的区别在于定时器的计时方式不同。我们主要讲动态定时器。



动态定时器在内核中是没有个数限制的，需要多少个创建多少个就可以，到期会自动销毁。我们来看一下内核中定时器的结构定义，位置在linux/timer.h中。第一个参数是entry，表示定时器链表的入口。系统会维护一个双向循环链表，这个链表根据定时器的另一个变量expires来对定时器分组存放。当定时器到期的时候，会自动被踢出链表。这里的expires是定时器的时值，以节拍为基本单位。比如我们要设置一个2秒的定时器，那么就设置为jiffies+2\*HZ就可以了。



我们继续看这个结构体的其他参数，第三个参数是function。当当前的jiffies大于expires的时候会运行function所指向函数，这个函数我们可以自定义，同时函数还有给一个参数，也就是data。下面是一个合法的函数原型。这个data数据是一个无符号长整型，可以根据具体的需要来随便自定义。比如存放设备ID，或者内存地址等等。最下边的base参数一般用户不要使用，是定时器的内部值。



使用定时器也非常简单，总共分四步，第一定义定时器，第二设定超时时间，第三指定超时后要运行的函数，第四激活定时器。Linux内核也提供了相应的函数来简化这些程式化的操作。这些函数的定义大家在linux目录下的timer.h文件中就能看到。下面我们具体的来讲一下定时器的使用流程。



首先，通过timer\_list来定义一个定时器，然后通过内核提供的init\_timer函数来初始化定时器的内部值。接着对定时器内部的变量进行赋值，包括超时时长，超时时要运行的函数，以及一个传递给function 的data参数。紧接着调用add\_timer函数，激活定时器，实际上就是将定时器插入到合适的链表里，由内核管理这个定时器。



这样我们一个定时器就新建完成并开始工作了。当jiffies大于或等于超时值就会运行函数。那么为什么会有大于的情况发生呢？因为可延迟函数的存在，有可能会延误定时器的执行。另一方面定时器也可能会延迟到超时的下一个节拍执行，所以不能用来执行硬实时任务。



有的时候我们拿到一个已激活的定时器，可能会有改变超时值的需求。这个时候就需要用到mod\_time函数。使用的时候只需要传入现有定时器地址以及新的超时值就可以了。同时如果定时器已经初始化但没有激活，用mod\_time函数修改超时时间后，同样会被激活。



假设某一个定时器在超时前我们就不需要了，可以使用del\_timer函数来将其删除。不论定时器有没有激活都是删除。



好，说完了定时器的使用后我们来思考一个问题。由于我们的内核或者用户会有大量任务需要用到定时器，这就意味着操作系统需要对这些定时器进行维护。那么什么样的数据结构可以高效的完成定时器的维护问题呢？Linux系统设计了一个专用的数据结构tvec\_base\_t，用来管理系统所有的内核动态定时器。



我们来简单看一下这个数据结构的源码。一眼望过去好像很复杂，不过没关系，我们一点点分析。第一个是同步锁。第二个是timer\_jiffies字段,表示需要检查的动态定时器的最早到期时间：大于或等于jiffies，说明延迟函数没有积压；小于jiffies，说明前几个节拍相关的可延迟函数必须处理。在多处理器系统中，字段running\_timer指向由本地CPU当前正处理的动态定时器的timer\_list数据结构。



这个数据结构中存在着非常显眼的5个变量，tv1到tv5。这五个变量都是数组，长度是64，其中tv1比较特殊，长度是256。我们前面知道定时器是存储在链表中的，这里的数组就是用来存储一个又一个的链表头的。通过这样的数据结构系统就可以高效的管理所有的定时器。那么我们如何来决定把哪链表头放到哪一个数组里呢？Linux内核把定时器按照即将到时的时长进行了分类，分为了五类，对应时长的定时器链表会被放在指定的数组里。



我们把tv1到tv5单独拿出来看一下。每一个变量都有一个数值范围，这个范围对应的就是我们定时器的时长。每一段的长度以2的六次方指数增长，一直到tv5为止。我们注意到tv5的上限是2的32次方减1，这个刚好是32位系统下jiffies的取值上限。所以这一组变量可以完整的覆盖任何定时器的时长。定时器每时每刻都在前进，随着时间的推移，定时器可能已经不适合刚刚加入时的时间段，这时系统会自动将定时器移动到相应的时间段中。



前面我们看到tv1的数组长度是256，剩下的tv长度都是64，那为什么这样设计呢？我们知道tv1存储的是最近255节拍内将到时的定时器。所以是离到时最近的一批定时器，系统需要集中处理这些数据。为了高效的处理，将每一个不同的expires值分别对应一个链表，所以tv1的长度是256。而剩下的tv并没有那么紧急，所以每一个链表存储的可以超时值不同的定时器。不然的话，如果对每一个不同超时值的定时器都创建一个链表的话，一共需要2的32次方个链表，显然是不现实的。所以Linux系统选择了这种兼顾空间和时间的解决方案。

从术语上来讲，tv1属于严格语义上的定时器向量。而tv2~tv5属于定义的比较松散的定时器向量。有了这样的结构，Linux系统就可以轻松管理所有的定时器了。



这里我们可以直观的看到每一个tv的数据结构。数组按顺序排列，存放的都是定时器链表的头地址。通过这个地址可以找到所有已存在的定时器。每一个链表上存在的定时器数量不一定，当然也有可能不存在任何定时器。



说完了数据结构，我们再来探究一下实际运行中是怎样的。这样的结构要怎么样使用才能达到最优的效率。系统首先会查看tv1里的所索引的定时器。tv1里存在的256个定时器链表是接下来紧接着要到时的计时器，所以必须优先处理。所以系统会循环的去处理tv1里所有的定时器，待处理完毕后再去从tv2里面找即将到时，也就是255个节拍以内的定时器，将这些定时器移动到t1内，tv3，tv4，tv5以此类推。越往后面的tv由于间隔时间长，所以很久才会被检索一次。这样的设计避免了无穷无尽的轮询，可以在实现目标的情况下保持高效率。



**略**



对于内核来说，除了定时器以外，还需要其他的方法来实现任务的延迟进行。这一类延迟往往非常短，主要有一下几种方法：忙等待、段延迟、和schedule\_timeout函数。我们简单讲解一下这三个方法。



忙等待是最简单的一种延迟方法。首先设定一个超时值，然后下面接一个while循环，用来判断当前节拍和超时值之间的关系，一旦超时循环就结束。我们看到这段代码非常精简，对于在程序中需要简短延迟的操作是非常方便的。那么以秒为单位的延时，直接以HZ为基本单位就可以了。这样还可以规避掉不同系统下节拍率不同的情况。



那这种简短的代码有没有什么缺陷呢？当然有，而且是显然易见的。一旦进入了忙等待，我们的CPU就会一直做这一件事情，就是判断超时发没发生，而其他任务是没有办法插入进来执行的。相当于发生了阻塞，降低了CPU的利用率。这是引入cond\_resched函数可以解决这个问题。加入该函数，可以让渡CPU给其他程序运行。忙等待就讲这么多。



下面看短延迟。前面的忙等待又一个特点，就是等待的最小单位是一个节拍，但是内核部分的代码经常会要求更加精细的等待时间，同时要求也更精确，比如设备的驱动程序。所以忙等待无法满足这类需求。不过好在内核提供了三种精度更高的延迟函数，分别是mdelay,udelay,ndelay。分别可以处理毫秒、微秒、纳秒级的延迟。但是这种延迟也仍然是忙等待，也就是等待过程中CPU无法处理其他程序。



最后看schedule\_timeout。看函数名也能看出来，是有计划的延迟。在程序中加入这个函数，参数为延迟的节拍数。运行到此处后任务就被放置在睡眠队列中，直到超时才会被唤醒。同时函数还可以设置可中断参数，在运行的过程中可以被激活唤醒。



接下来讲一下定时器的竞争条件。



我们先来看上面这一段代码。首先删除定时器，紧接着去把相关的资源进行释放。按理讲顺序执行的代码应该不会有什么问题，但是如果遇到了多核CPU的情况，可就不太一样了。调用del\_timer的时候定时器可能运行在其他的CPU上。为了避免这种情况的发生，内核提供了del\_timer\_sync函数，除了删除定时器外，还会检查定时器是否在其他CPU上运行。如果是，del\_timer\_sync()就等待，直到定时器函数运行结束，才真正返回删除成功。



再看另一个情景。对于一个已经初始化的定时器，现在打算修改时间。这个程序员首先删除了这个定时器，然后改这个定时器更新超时时间。接着再把定时器加入到定时器链表中。那么这样操作肯定是不对的，前面刚说过del\_timer在多CPU的情况下是不安全的。另一方面，直接修改expires变量也会带来不可预期的错误。所以修改定时器还是要用mod\_timer函数。



再来看一下利用定时器我们能实现什么应用。



这里以系统函数nanosleep为例。这个函数可以按规定的时间来暂停某个进程，到期后再恢复。函数原型如图所示，有两个参数，req和rem，都是timespec结构体。这个结构体由两部分组成，一个是秒，一个是纳秒。调用nanosleep函数后，进程会进入一个中断状态。在中断状态中的进程，是可以接收相应的激活信号而重新运行的。也就是说很有可能我们的定时器还没有到期，进程就被额外的信号激活而重新运行。这种情况下函数会返回-1。同时剩余的还没走完的时间量会被记录在rem变量里。



在当前的进程进入中断状态后，nanosleep会调用schedule\_timeout函数来实现进程的延时。



下面是schedule\_timeout的实现。前面几行和之前讲的定时器用法一模一样。添加完计时器后，调用了schedule函数。这时候就会运行其他待运行的进程。当定时器到时候，定时器会被删除。最后返回一个值。0表示进程是在定时器超时后被唤醒的，大于零表示的进程收到激活信号而被唤醒，返回的是剩余的节拍数。



定时器到期后会运行process\_timeout函数，主要作用就是唤醒被挂起的进程，继续运行。那么如果进程不是到期被唤醒，而是中途被信号唤醒的，那是不是延时时间就不够用了？其实一旦发生唤醒的情况，系统会继续执行sys\_nanosleep，一直到schedule\_timeout返回值为零，即真的到期。否则就会不断重启计时，知道进程延时到期为止。



**略**