《操作系统》

课程讲稿

第4章 第4讲

下半部

软件所制

第四章 第4讲 下半部

**学时：1**学时

**教学目的：**对中断处理机制中的下半部机制进行学习……

**课程时间线：**

章节简介

下半部概览

下半部处理

tasklet

工作队列

5min

10min

25min

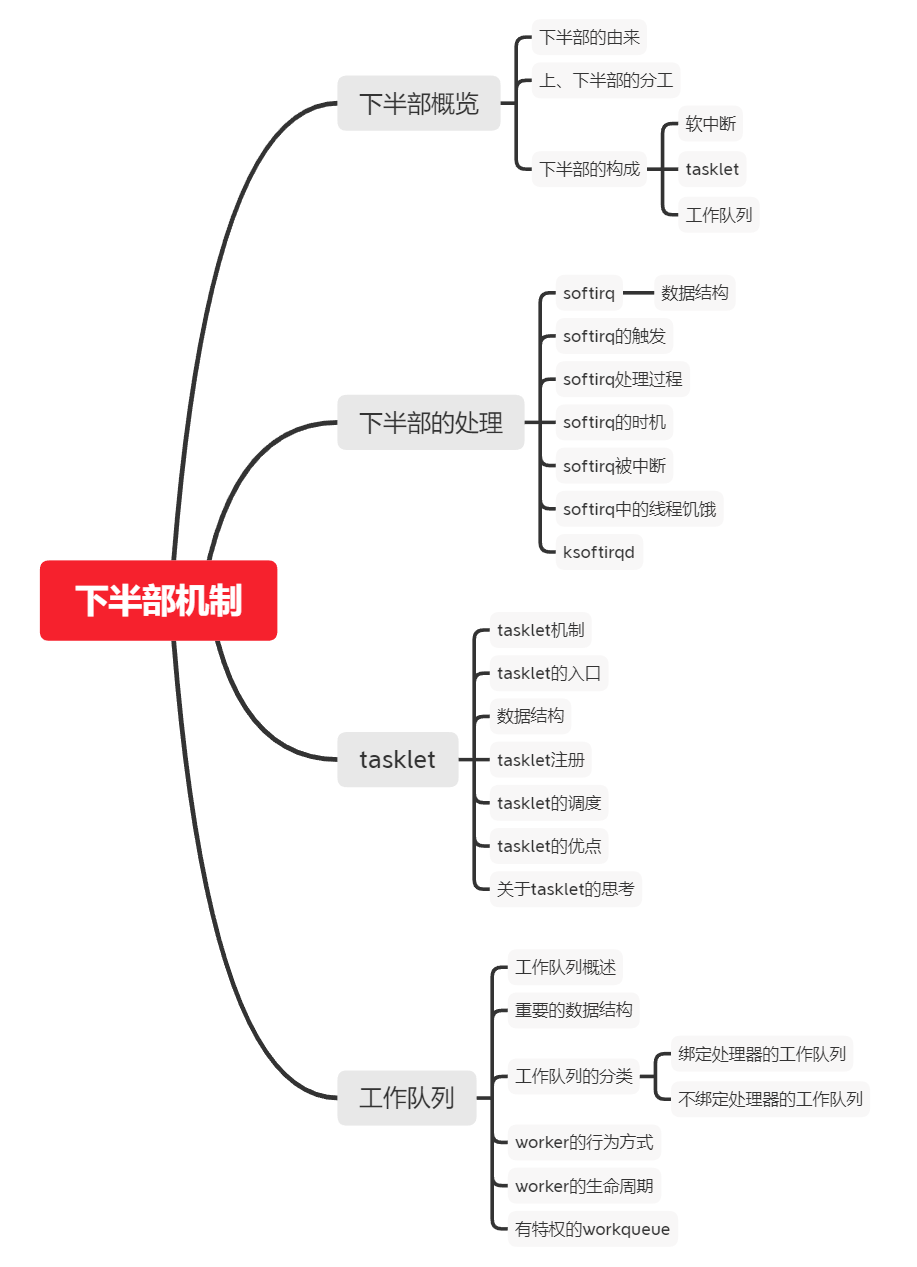
35min

45 min

**课外参考读物：**

《深入理解LINUX内核》

**知识框图：**

****

**PPT讲稿：**

1. 上一节我们学习了中断处理过程，但是还遗留了一部分，就是今天要讲的——下半部。
2. 下半部是相对于上半部来讲的，而上半部就是我们上一节所讲的内容。
3. 本节中我们将分为四块内容来介绍下半部机制，分别是初始化下半部概览、下半部的处理、tasklet和工作队列，我们先来看看下半部概览这一部分。
4. 在上一章我们对中断处理或称中的下半部机制做了一点铺垫，我们知道下半部是在do\_IRQ函数的末尾开启的。在这节课我们对下半部机制进行更加细致的讲解。在学习一个东西之前我们首先要了解为什么要有这个东西，有时候了解为什么要这样做比如何这样做更有价值。一个中断产生之后，内核在中断处理函数中可能需要完成很多工作。但是中断处理函数的处理是关闭了中断的。也就是说在响应中断时，系统不能再次响应外部的其它中断。这样可能会造成外部中断的丢失。于是linux内核设计出了一种机制，中断函数需要处理的任务分为两部分，一部分在中断处理函数中执行，这时系统关闭中断。另外一部分在软件中断中执行，这个时候开启中断，系统可以响应外部中断。
5. 为了解决中断处理程序执行过长和中断丢失的问题，Linux 将中断处理过程分成了两个阶段，也就是“上半部”和“下半部”：
   1. 上半部用来快速处理中断，它在中断禁止模式下运行，主要处理跟硬件紧密相关的或时间敏感的工作。
   2. 下半部用来延迟处理上半部未完成的工作，通常以内核线程的方式运行。

在图中我们可以看到在上半部完成之后紧接着就开始了下半部的处理，而下版本有几个重要的部分：softirq、tasklet、workqueue和内核线程。这些将在后面讲到。

1. 以上种种原因促使了系统程序员开发了下半部机制，因此我们可以得出设计下半部机制的原则：
   1. 如果该任务对时间比较敏感，将其放在上半部中执行。
   2. 如果该任务和硬件相关，一般放在上半部中执行。
   3. 如果该任务要保证不被其他中断打断，放在上半部中执行（因为这是系统关中断）。
   4. 其他不太紧急的任务， 一般考虑在下半部执行。因为下半部可以延后操作，让其他更紧急的操作线完成。
2. 构成下半部机制主要有三种技术：
   1. 软中断
      1. 在内核中仿造硬中断实现软中断向量表（softirq\_vec），作用与中断描述符表类似
      2. 在硬中断的中断处理程序结束时候给出相应的软中断向量，也就是在do\_IRQ结束的时候开启软中断。
      3. 一个软中断不会去抢占另一个软中断，只有硬件中断才可以抢占软中断，这意味着所有软中断的优先级都是相同的。
   2. tasklet
      1. 软中断的分配是静态的（编译时定义），tasklet的分配和初始化可以在运行时进行（动态），这一点是它和软中断之间的最大的不同
      2. 软中断是可重入函数而且必须明确地使用自旋锁保护其数据结构。相同类型的tasklet总是被串行执行
      3. 将tasklet组成一个链表挂接到cpu上进行调度（注意，tasklet不是进程）
3. 第三种技术就是工作队列：
   1. 工作队列的实体是内核线程，在延后处理的时候将处理任务交给内核线程去完成，这是与tasklet最大的不同
   2. 把推后的工作交由一个内核线程去执行，因此工作队列的优势就在于它允许和线程一样被调度甚至睡眠。
4. 对下半部有个整体性的了解后，我们再来看看细节方面的东西，首先看看下半部是如何进行处理的。
5. 为了了解下半部的工作机制，我们还是先来看看期内部比较重要的数据结构。在上一节课我们讲到了软中断数组softirq\_vec，它是用来放置softirq\_action的，为了便于多核的并行处理，它还被设计成了per-cpu类型的数组，也就是每个处理器对应一个。而每个softirq\_action对应一个软中断处理函数。
6. 下面来看看内核如何触发一个软中断，也就是告诉处理器：当前我要给你挂接一个软中断，你有空的时候来处理一下！内核使用一个位图\_\_softirq\_pending（目前只使用了11个位）来代表某位对应的软中断是否被触发（有正在等待处理的软中断，触发某位软中断即为告诉内核此位有软中断正在等待被处理），若为1则表示有，否则没有。触发软中断的任务是由raise\_softirq\_irqoff函数来完成的。
7. 在上一章我们了解到do\_IRQ函数在他的最后开启了软中断的执行，其实经过层层调用后，最终执行软中断的任务是由\_\_do\_softirq函数来完成的。但是进入\_\_do\_softirq函数还有两个前提，
   1. 1.调用in\_interrupt()判断当前是否还处于硬中断、软中断或者屏蔽了软中断，是的话则否定（不用进入\_\_do\_softirq()）；
   2. 2. 查看位图\_\_softirq\_pending是否有等待的softirq，没有则否定（不用进入\_\_do\_softirq()）。
8. \_\_do\_softirq的主要动作就是扫描当前cpu的\_\_softirq\_pending位图，看是否有待处理的软中断啊，如果有，则调用softirq\_vec数组中对应项的action函数进行软中断的处理。一直持续这个过程，直到\_\_softirq\_pending每个位都为0。
9. 我们看看第二步，这里说执行优先级较高的待处理softirq，我们并没有定义各个软中断的优先级，那优先级如何体现呢？这是因为在之前我们扫描\_\_softirq\_pending位图时是从高位往低位扫描的，排在越前面的软中断当然优先级就越高了。第三步我们发现当待处理的软中断超过一定数量时，则交给后台软中断处理进程（ksoftirqd）执行，这里我们先埋个伏笔，留到后面再讲。
10. 前面我们说软中断是在do\_IRQ函数要结束时才开始进行处理的，事实上处理软中断的时机不止这样一个，还有如ppt上所述的这几个时机。可以后面的路径来找到对它的调用。
11. 我们需要注意一下，软中断就是为了延后操作而生的，因此在如那终端处理过程中可以被其他中断所打断，但是不能被其他软中断打断。而每个CPU都对应一个softirq的栈和一个hardirq的栈，这样就保证了处理完硬中断后能正确返回被中断的softirq。这一点和函数调用或硬中断返回有点类似。
12. 硬中断打断软中断的过程与中断处理的过程非常类似，也是用栈来保存上下文，而后也要进行返回。
13. 同时，若系统当中有较多的软中断待处理的话，就会一直进行软中断的处理，而系统当中的普通进程则就收到了冷落而长时间得不到运行，这就叫做线程饥饿。
14. 为了应对这种饥饿现象，内核对\_\_do\_softirq()的处理时间和次数都进行了限制，其中时间限定为2ms(用MAX\_SOFTIRQ\_TIME表示)，次数限制为10次(用MAX\_SOFTIRQ\_RESTART表示)，当超过了规定的时间或次数的话，就将处理软中断的任务交给一个叫做ksoftirqd的内核线程去处理。
15. 每个cpu的rq上都有一个内核线程ksoftirqd，当将要发生线程饥饿时就唤醒ksoftirqd。而在ksoftirqd依旧是调用前面所讲的\_\_do\_softirq()函数去处理这些待处理软中断。但是这个处理的过程不是由内核来完成，而是由内核线程来完成，这就意味着这个过程可以想线程一样参与调度了。
16. 学习完下半部的处理过程，接下来我们进入tasklet机制（有时候也称作小任务）的学习。
17. tasklet其实就是软中断的一种。就像下图描述的一样，在进入软中断机制之后才能进入tasklet机制。他出现的原因如ppt上所讲：随着中断数的不停增加，软中断不够用了，于是下半部又做了进化，可以动态增加软中断类型。
18. pending位图中的HI\_SOFTIRQ和TASKLET\_SOFTIRQ就是tasklet的入口。其中由于在位图种的位置关系HI\_SOFTIRQ的优先级比TASKLET\_SOFTIRQ要高。在初始化的时候将HI\_SOFTIRQ和TASKLET\_SOFTIRQ分别指向了tasklet\_hi\_action和tasklet\_action，在这两个函数内进运行tasklet任务。
19. 来看看tasklet机制中主要的数据结构。一个tasklet\_struct就代表一个tasklet任务，在内核中有一个tasklet\_struct实例组成的链表，代表待处理的tasklet任务。tasklet\_struct里面的func就是待处理的任务，即处理函数，data就是传入的参数的指针。
20. 内核用tasklet\_vec来指向系统中由tasklet\_struct构成的链表，head指向表头，tail指向表尾。使用tasklet\_schedule函数来向tasklet\_vec中插入tasklet\_struct。每个tasklet在执行完毕之后会被从tasklet\_vec取下，因此每个tasklet只能执行一次。
21. Tasklet的注册机制非常简单，首先使用test\_and\_set\_bit函数检查TASKLET\_STATE\_SCHED位，这个TASKLET\_STATE\_SCHED是个位图，作用类似软中断里的\_\_softirq\_pending位图。最后将\_\_softirq\_pending的TASKLET\_SOFTIRQ位置1，也就是向系统报告当前有tasklet正在等待执行。tasklet\_hi\_schedule与tasklet\_schedule机制基本一样
22. tasklet 的调度机制也非常简单，首先按照软中断的机制走，检查到\_\_softirq\_pending位图的时候发现HI\_SOFTIRQ或者TASKLET\_IRQ为1，则说明有tasklet任务待处理，接着按链表顺序检查的tasklet\_vec中的各个tasklet\_struct，若TASKLET\_STATE\_SCHED位为1且TASKLET\_STATE\_RUN为0，则对其进行处理。
23. 最后看看tasklet的优点：
    1. 无类型数量限制（由来）；
    2. 效率高，无需循环查表；
    3. 支持SMP机制；
    4. softirq是在编译链接时静态定义的，也就是提前分配的，而tasklet是在运行时动态创建和注册的，这就提供了很大的灵活性。
    5. 一个softirq的处理函数能够并行地运行在多个CPU 上，因此softirq的处理函数对共享资源操作时需要同步。然而同一个tasklet处理函数不能同时运行在不同CPU上，因此不用在CPU间进行同步。
24. 下面来思考几个问题，前面讲软中断的时候我们讨论过软中断的调度时机，那么tasklet的调度时机呢？tasklet的调度时机与softirq的调度时机相同，这又是为什么呢？因为tasklet的入口与softirq相同，进入tasklet首先要进入前面所讲的\_\_do\_softirq()函数，因此它们的调度时机肯定是相同的。
25. 讲完了软中断机制和tasklet机制，我们再来学习下半部的最后一种技术——工作队列。
26. 工作队列（workqueue）是使用内核线程来异步处理后半段任务的机制。与前两种技术不同，工作队列是一开始就将任务交给内核线程来处理，而软中断和tasklet是有一定数量时才交给内核线程处理。
27. Linux内核预定义了一些默认的工作队列，当然我们也可以根据需要来自己创建专用的工作队列
28. 工作队列中涉及的一些基本概念较多，下面我们来看看这些基本概念都有什么。
    1. work ：工作，在这里理解为下半部所要处理的任务任务。其实就是一段用于后半段任务的函数。
    2. workqueue ：work的集合，不同种类的work可以组成不同的workqueue。其中workqueue 和 work 是一对多的关系。
    3. worker ：工人。在代码中 worker 对应一个 work\_thread() 内核线程。
    4. worker\_pool：worker的集合（工人池）。worker\_pool 和 worker 是一对多的关系，有活跃的worker也有空闲的worker，一个CPU上的所有worker线程共同构成了一个worker\_pool
    5. pwq(pool\_workqueue)：中间人 / 中介，负责建立起 workqueue 和 worker\_pool 之间的关系。workqueue 和 pwq 是一对多的关系，pwq 和 worker\_pool 是一对一的关系。
29. 下面看看工作队列涉及的几个重要的数据结构。首先是work\_struct，一个work\_struct实例就对应一个work，里面有func，也就是此工作对应的处理函数的句柄，响应肯定还有要传递给这个处理函数的参数指针data，最后的entry用于组织同一个工人池的work。
30. 接下来是工人对应的结构——worker，刚才我们说过，每个工人都属于一个工人池里面，因此worker里面自然有指向期工人池的成员，工人要处理work，自然也有指向work的指针，和其对应的函数。Worker是一个内核线程，那就肯定有对应的task\_struct。为了快速访问，还有指向其work所在的worqueue的指针。
31. 工作队列主要分为两大类，绑定处理器的工作队列和不绑定处理器的工作队列。在没有特殊的标注下，内核在创建工作队列时默认就是创建的绑定处理器的工作队列，在这种工作队列下的所有工人线程都绑定到了特定的CPU上运行。而不绑定处理器的工作队列在创建时需要特殊标注，在这个工作队列里的work可以根据需要在不同的cpu之间迁移。
32. 关于绑定处理器的工作队列，右边这张图描述的非常清楚。
    1. 内核为每个工作队列创建一个workqueue\_struct实例来管理一个workqueue
    2. 在每个workqueue\_struct实例中有若干个pool\_workqueue成员（每个CPU对应一个）
    3. 一个pool\_workqueue实例中有一个工人池，一个工人池对应一个未处理工作链表、一个工人链表。
33. 而不绑定处理器的工作队列则与绑定处理器的工作队列有点不同，绑定处理器的工作队列中的pool\_workqueue不再是每个CPU对应一个，而是一个内存节点对应一个。其他方面与绑定处理器的工作队列相同。
34. Worker的行为方式很有意思，工人会不断找这当前工人池里面地待处理work，当work数量太多而worker不够时，内核会新创建worker来进行处理；相反，当work太少而空闲worker太多时，会kill掉一部分空闲worker。虽然很残忍，但是却非常高效。
35. Worker实际上就是线程，因此他也有自己的生命周期，如同一半的进程和线程一样，工人也有运行态、阻塞态、就绪态（由于就绪态和运行态之间关系的特殊性，这里将它们画在一起了）。和一半的线程不同，工人线程还有空闲态和破坏态、因为它们和0号进程一样允许处于空闲，以及可以直接被内核杀掉。
36. 在最后我们对三种后半部机制进行一下比较
    1. 工作队列是使用内核线程实现的，因而可以睡眠，这是他和前两个机制最大的不同
    2. 软中断是静态的，而tasklet是动态的，具有更大的灵活性。
    3. 软中断和tasklet在一定数量内由内核直接来处理，但超过一定数量之后就交由一个叫ksoftirqd的内核线程来负责处理，整个系统只有一个线程来处理这些软中断和tasklet；而工作队列在一开始就用多个内核线程来处理多个任务，因此获得cpu的几率较高。