# 软中断

软中断是内核提供的一种延迟机制，完全由软件触发。虽然是延迟机制，实际上，在大多数情况下，它比普通进程能够得到更快的响应。软中断也是内核其他机制的基础，如tasklet、高分辨率timer等。

软中断资料有限，目前内核中实现了10中类型的软中断

1. **enum**
2. {
3. HI\_SOFTIRQ=0,
4. TIMER\_SOFTIRQ,
5. NET\_TX\_SOFTIRQ,
6. NET\_RX\_SOFTIRQ,
7. BLOCK\_SOFTIRQ,
8. BLOCK\_IOPOLL\_SOFTIRQ,
9. TASKLET\_SOFTIRQ,
10. SCHED\_SOFTIRQ,
11. HRTIMER\_SOFTIRQ,
12. RCU\_SOFTIRQ,    /\* Preferable RCU should always be the last softirq \*/
14. NR\_SOFTIRQS
15. };

内核开发者不建议擅自增加软中断数量，如果需要新的软中断，尽可能它实现为基于tasklet形式

## ksoftirqd

early\_initcall(spawn\_ksoftirqd);---初始化的时候调用spawn\_ksoftirqd函数。

smpboot\_register\_percpu\_thread(&softirq\_threads) ---热插拔阶段为每个percpu上创建一个ksoftirqd守护进程

1. DEFINE\_PER\_CPU(**struct** task\_struct \*, ksoftirqd);
2. **static** **struct** smp\_hotplug\_thread softirq\_threads = {
3. .store          = &ksoftirqd,
4. .thread\_should\_run  = ksoftirqd\_should\_run, /\*判断是否应该运行处理软中断\*/
5. .thread\_fn      = run\_ksoftirqd,        /\*运行处理软中断\*/
6. .thread\_comm        = "ksoftirqd/%u",
7. };
8. **static** **void** run\_ksoftirqd(unsigned **int** cpu)
9. {
10. /\*关闭本地cpu中断\*/
11. local\_irq\_disable();
12. **if** (local\_softirq\_pending()) { /\*检查本地cpu上是否有软中断挂起\*/
13. /\*
14. \* We can safely run softirq on inline stack, as we are not deep
15. \* in the task stack here.
16. \*/
17. \_\_do\_softirq();/\*处理软中断\*/
18. local\_irq\_enable();/\*使能本地cpu中断\*/
19. cond\_resched(); /\*有条件的重新调度\*/
21. preempt\_disable();
22. rcu\_note\_context\_switch(cpu);
23. preempt\_enable();
25. **return**;
26. }
27. local\_irq\_enable();
28. }

## 结构

Irq\_cpustat\_t,多个软中断可以同时在多个cpu运行，就算是同一个软中断，也有可能同时在多个cpu上运行。内核为每个cpu都管理着一个待决软中断pedding，他就是Irq\_cpustat\_t

1. **typedef** **struct** {
2. unsigned **int** \_\_softirq\_pending;
3. } \_\_\_\_cacheline\_aligned irq\_cpustat\_t;
4. irq\_cpustat\_t irq\_stat[NR\_CPUS] \_\_\_\_cacheline\_aligned;
5. **struct** softirq\_action
6. {
7. **void**    (\*action)(**struct** softirq\_action \*);
8. };
9. **static** **struct** softirq\_action softirq\_vec[NR\_SOFTIRQS] \_\_cacheline\_aligned\_in\_smp;

\_\_softirq\_pending中每个bit，对应某一个软中断，某个Bit被置位，说明有相应的软总段等待处理。因此最多只能定义32个软中断类型。

## 软中断触发

想触发软中断，只需要调用raise\_softirq即可，它的实现简单先关闭本地cpu中断，然后调用raise\_softirq\_irqoff，再打开本地cpu中断。

1. **void** raise\_softirq(unsigned **int** nr)
2. {
3. unsigned **long** flags;
5. local\_irq\_save(flags);
6. raise\_softirq\_irqoff(nr);
7. local\_irq\_restore(flags);
8. }

再来看raise\_softirq\_irqoff

1. **inline** **void** raise\_softirq\_irqoff(unsigned **int** nr)
2. {
3. \_\_raise\_softirq\_irqoff(nr);
5. ......
6. **if** (!in\_interrupt())
7. wakeup\_softirqd();
8. }

先通过\_\_raise\_softirq\_irqoff设置cpu的软中断pending标志位(irq\_stat(NR\_CPUS))，然后通过in\_interrupt判断是否在中断上下文中，如果不成立，则唤醒软中断守护进程，在守护进程中执行软中断的回调函数。

## 软中断的执行

软中断有两种执行方式，一种是在中断调用结束时，一种是在ksoftirqd守护进程中

1. /\*
2. \* Exit an interrupt context. Process softirqs if needed and possible:
3. \*/
4. **void** irq\_exit(**void**)
5. {
6. #ifndef \_\_ARCH\_IRQ\_EXIT\_IRQS\_DISABLED
7. local\_irq\_disable();
8. #else
9. WARN\_ON\_ONCE(!irqs\_disabled());
10. #endif
12. account\_irq\_exit\_time(current);
13. preempt\_count\_sub(HARDIRQ\_OFFSET);
14. /\*
15. 在中断发生嵌套时，通过in\_interrupt能确保在最外层的中断Irq\_exit阶段
16. invoke\_softirq才会被调用
17. \*/
18. **if** (!in\_interrupt() && local\_softirq\_pending())
19. invoke\_softirq();
21. tick\_irq\_exit();
22. rcu\_irq\_exit();
23. trace\_hardirq\_exit(); /\* must be last! \*/
24. }

代码最终都会进入到\_\_do\_softirq中，执行软中断的重点都在该函数中。

1. asmlinkage **void** \_\_do\_softirq(**void**)
2. {
3. unsigned **long** end = jiffies + MAX\_SOFTIRQ\_TIME;
4. unsigned **long** old\_flags = current->flags;
5. **int** max\_restart = MAX\_SOFTIRQ\_RESTART;
6. **struct** softirq\_action \*h;
7. **bool** in\_hardirq;
8. \_\_u32 pending;
9. **int** softirq\_bit;
10. **int** cpu;
12. /\*
13. \* Mask out PF\_MEMALLOC s current task context is borrowed for the
14. \* softirq. A softirq handled such as network RX might set PF\_MEMALLOC
15. \* again if the socket is related to swap
16. \*/
17. current->flags &= ~PF\_MEMALLOC;
18. /\*
19. 复制软中断掩码到局部变量，这是必要的
20. 因为local\_softirq\_pending中的值在开中断后将不再可靠，必须先保存
21. \*/
22. pending = local\_softirq\_pending();
23. account\_irq\_enter\_time(current);
25. /\*
26. 标志下面的代码正在处理softirq
27. \*/
28. \_\_local\_bh\_disable\_ip(\_RET\_IP\_, SOFTIRQ\_OFFSET);
29. in\_hardirq = lockdep\_softirq\_start();
31. cpu = smp\_processor\_id();
32. restart:
33. /\* Reset the pending bitmask before enabling irqs \*/
34. set\_softirq\_pending(0);/\*清空pending\*/
36. local\_irq\_enable(); /\*打开本地中断\*/
38. /\*
39. 到这里已经打开了本地中断，下面在软中断处理执行过程中可能会被硬件中断抢占
40. \*/
41. /\*
42. 根据软中断标志位处理软中断
43. \*/
44. /\* softirq\_vec 存放action的结构体\*/
45. h = softirq\_vec;
47. **while** ((softirq\_bit = ffs(pending))) {
48. unsigned **int** vec\_nr;
49. **int** prev\_count;
51. h += softirq\_bit - 1;
53. vec\_nr = h - softirq\_vec;
54. prev\_count = preempt\_count();
56. kstat\_incr\_softirqs\_this\_cpu(vec\_nr);
58. trace\_softirq\_entry(vec\_nr);
59. h->action(h);
60. trace\_softirq\_exit(vec\_nr);
61. **if** (unlikely(prev\_count != preempt\_count())) {
62. pr\_err("huh, entered softirq %u %s %p with preempt\_count %08x, exited with %08x?\n",
63. vec\_nr, softirq\_to\_name[vec\_nr], h->action,
64. prev\_count, preempt\_count());
65. preempt\_count\_set(prev\_count);
66. }
67. rcu\_bh\_qs(cpu);
68. h++;
69. pending >>= softirq\_bit;
70. }
72. /\*关掉本地中断\*/
73. local\_irq\_disable();
74. /\*
75. 由于前面有打开过本地中断，因此这次可能会有新的软中断未处理，再检查处理，
76. \*/
77. pending = local\_softirq\_pending();
78. **if** (pending) {
79. **if** (time\_before(jiffies, end) && !need\_resched() &&
80. --max\_restart)
81. **goto** restart;
83. wakeup\_softirqd();
84. }

87. lockdep\_softirq\_end(in\_hardirq);
88. account\_irq\_exit\_time(current);
89. \_\_local\_bh\_enable(SOFTIRQ\_OFFSET);
90. WARN\_ON\_ONCE(in\_interrupt());
91. tsk\_restore\_flags(current, old\_flags, PF\_MEMALLOC);
92. }

## 软中断注册

void open\_softirq(int nr, void (\*action)(struct softirq\_action \*))