**进程的调度schedule和switch\_to解析**

2014年07月30日 14:47:31 [ToureYaya](https://me.csdn.net/charliewangg12) 阅读数：766

[Linux0.11内核--进程的调度](http://blog.csdn.net/geekcome/article/details/6371083)

[Linux任务切换代码（switch\_to）详解](http://blog.csdn.net/fpmystar/article/details/4100702)

[linux0.11中switch\_to理解](http://blog.csdn.net/smallmuou/article/details/6837087)

前言

在我学习进程调度的时候，是通过这三篇文章把这个问题搞明白的。所以这里贴

在一起，方便查看。第一篇主要是将进程的调度，可认为是个总体讲述，里面共

涉及两个函数schedule()和switch\_to()这篇把schedule()注释的比较清楚，但

switch\_to()我没太明白；第二篇和第三篇都是详解switch\_to（）的，其实第三篇是

讲的最清楚，我是看到这篇明白的，但是第二篇里面的几个图可以帮助理解几个

重要的概念，所以也贴了下来。

**第一篇 进程的调度**

**进程的调度**

linux系统中，一个进程有5种可能状态，在sched.c第19行处定义了状态的标识：

#define TASK\_RUNNING 0 // 正在运行或可被运行状态

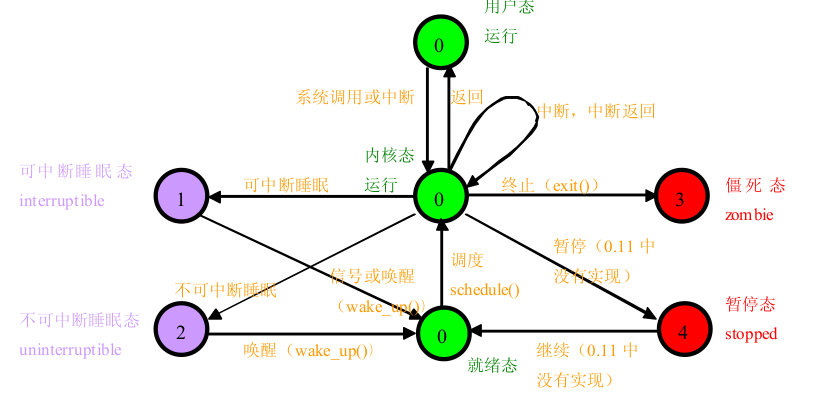
#define TASK\_INTERRUPTIBLE 1 // 可被中断睡眠状态

#define TASK\_UNINTERRUPTIBLE 2 // 不可中断睡眠状态

#define TASK\_ZOMBIE 3 // 僵死状态

#define TASK\_STOPPED 4 // 停止状态

各种状态的转换图如下：



**就绪态和运行态之间的转换**

当前占用CPU的进程调只有用了schedule()函数后，才可能会从运行态进入就绪态。Schedule()函数按照一定的选择策略选中处于TASK\_RUNNING态（包括用户运行态，内核运行态和就绪态）的某个进程，然后切换到该进程去执行。这时被选中的进程进入运行态，开始使用CPU资源。被选中的进程可能是刚刚调用schedule()函数的进程，也可能是其他进程。

schedule()函数在3种情况下会被调用

用户态时发生了时钟中断；

系统调用时相应的sys\_XXXX函数返回后；

睡眠函数内；

第一种情况发生在用户态。当时钟中断产生时，如果进程运行在用户态时并且时间片用完，中断处理函数do\_timer()会调用schedule()函数，这相当于用户态的运行被抢断了。如果进程处在内核态时发生时钟中断，do\_timer()不会调用schedule()函数，也就是内核态是不能被抢断的。当一个进程运行在内核态，除非它自愿调用schedule()函数而放弃CPU的使用权，它将永远占用CPU。由于schedule()不是系统调用，用户程序不能调用，所以在时钟中断中调用schedule()是必要的，这样保证用户态的程序不会独占CPU。

第二种情况就是为了对付运行在内核态的进程。应用程序一般通过系统调用进入内核态，因此linux 0.11在系统调用处理函数（sys\_XXXX()）结束后，int 0x80处理函数会检查当前进程的时间片和状态，如果时间片用完或状态不是TASK\_RUNNING，会调用schedule()函数。这时相当于内核态进程主动放弃对对CPU的占用。由此可见，如果某个系统调用处理函数或者中断异常处理函数永远不退出，比如进入死循环或者等待其他资源，整个系统死锁，任何进程都无法运行。

比较前两种情况，我们看到linux有保证用户态的程序不独占CPU的机制，却不能保证内核态程序不独占CPU。这也反映了系统级别开发和用户级别开发的不同之处。系统程序员需要考虑更多的问题。

第三种情况在下面一节*运行态（包括就绪态）和睡眠态之间的转换*中讨论。当进程等待的资源还不可用时，它进入睡眠态，并且调用schedule()让出CPU。

**switch\_to() （sched.h 第173行）**

**[cpp]**[**view plaincopy**](http://blog.csdn.net/geekcome/article/details/6371083)

1. **/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**
2. **/\* 功能：切换到任务号（即task[]数组下标）为n的任务                          \*/**
3. **/\* 参数：n 任务号                                                         \*/**
4. **/\* 返回：（无）                                                               \*/**
5. **/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**
6. **// 整个宏定义利用ljmp指令跳转到TSS段选择符来实现任务切换**
7. **#define switch\_to(n) {/**
8. **// \_\_tmp用来构造ljmp的操作数。该操作数由4字节偏移和2字节选择符组成。当选择符**
9. **// 是TSS选择符时，指令忽略4字节偏移。**
10. **// \_\_tmp.a存放的是偏移，\_\_tmp.b的低2字节存放TSS选择符。高两字节为0。**
11. **// ljmp跳转到TSS段选择符会造成任务切换到TSS选择符对应的进程。**
12. **// ljmp指令格式是 ljmp 16位段选择符：32位偏移，但如果操作数在内存中，顺序正好相反。**
13. **// %0   内存地址    \_\_tmp.a的地址，用来放偏移**
14. **// %1   内存地址    \_\_tmp.b的地址，用来放TSS选择符**
15. **// %2   edx         任务号为n的TSS选择符**
16. **// %3   ecx         task[n]**
17. **struct {long a,b;} \_\_tmp; /**
18. **\_\_asm\_\_("cmpl %%ecx,current/n/t" /  // 如果要切换的任务是当前任务**
19. **"je 1f/n/t" /                   // 直接退出**
20. **"movw %%dx,%1/n/t" /            // 把TSS选择符放入\_\_tmp.b中**
21. **"xchgl %%ecx,current/n/t" /     // 让current指向新进程的task\_struct**
22. **"ljmp \*%0/n/t" /                // 任务切换在这里发生，CPU会搞定一切**
23. **"cmpl %%ecx,last\_task\_used\_math/n/t" /  // 除进程第一次被调度外，以后进程从就绪**
24. **// 态返回运行态后，都从这里开始运行。因**
25. **// 而返回到的是内核运行态。**
26. **"jne 1f/n/t" /**
27. **"clts/n" /**
28. **"1:" /**
29. **::"m" (\*&\_\_tmp.a),"m" (\*&\_\_tmp.b), /**
30. **"d" (\_TSS(n)),"c" ((long) task[n])); /**
31. **}**

**schedule()（sched.c 第104行）**

**[cpp]**[**view plaincopy**](http://blog.csdn.net/geekcome/article/details/6371083)

1. **/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**
2. **/\* 功能：进程调度。                                                         \*/**
3. **/\*       先对alarm和信号进行处理，如果某个进程处于可中断睡眠状态，并且收 \*/**
4. **/\*       到信号，则把进程状态改成可运行。之后在处可运行状态的进程中挑选一个  \*/**
5. **/\*       并用switch\_to()切换到那个进程                                       \*/**
6. **/\* 参数：（无）                                                               \*/**
7. **/\* 返回：（无）                                                               \*/**
8. **/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**
9. **void schedule(void)**
10. **{**
11. **int i,next,c;**
12. **struct task\_struct \*\* p;**
13. **/\* check alarm, wake up any interruptible tasks that have got a signal \*/**
14. **// 首先处理alarm信号，唤醒所有收到信号的可中断睡眠进程**
15. **for(p = &LAST\_TASK ; p > &FIRST\_TASK ; --p)**
16. **if (\*p) {**
17. **// 如果进程设置了alarm，并且alarm已经到时间了**
18. **if ((\*p)->alarm && (\*p)->alarm < jiffies) {**
19. **// 向该进程发送SIGALRM信号**
20. **(\*p)->signal |= (1<<(SIGALRM-1));**
21. **(\*p)->alarm = 0; // 清除alarm**
22. **}**
23. **//可屏蔽信号位图BLOCKABLE定义在sched.c第24行，(~(\_S(SIGKILL) | \_S(SIGSTOP)))**
24. **// 说明SIGKILL和SIGSTOP是不能被屏蔽的。**
25. **// 可屏蔽信号位图 & 当前进程屏蔽的信号位图 = 当前进程实际屏蔽的信号位图**
26. **// 当前进程收到的信号位图 & ~当前进程实际屏蔽的信号位图**
27. **//                          = 当前进程收到的允许相应的信号位图**
28. **// 如果当前进程收到允许相应的信号，并且当前进程处于可中断睡眠态**
29. **// 则把状态改成运行态，参与下面的选择过程**
30. **if (((\*p)->signal & ~(\_BLOCKABLE & (\*p)->blocked)) &&**
31. **(\*p)->state==TASK\_INTERRUPTIBLE)**
32. **(\*p)->state=TASK\_RUNNING;**
33. **}**
34. **/\* this is the scheduler proper: \*/**
35. **// 下面是进程调度的主要部分**
36. **while (1) {**
37. **c = -1;**
38. **next = 0;**
39. **i = NR\_TASKS;**
40. **p = &task[NR\_TASKS];**
41. **while (--i) {       // 遍历整个task[]数组**
42. **if (!\*--p)      // 跳过task[]中的空项**
43. **continue;**
44. **// 寻找剩余时间片最长的可运行进程，**
45. **//  c记录目前找到的最长时间片**
46. **// next记录目前最长时间片进程的任务号**
47. **if ((\*p)->state == TASK\_RUNNING && (\*p)->counter > c)**
48. **c = (\*p)->counter, next = i;**
49. **}**
50. **// 如果有进程时间片没有用完c一定大于0。这时退出循环，执行 switch\_to任务切换**
51. **if (c) break;**
52. **// 到这里说明所有可运行进程的时间片都用完了，则利用任务优先级重新分配时间片。**
53. **// 这里需要重新设置所有任务的时间片，而不光是可运行任务的时间片。**
54. **// 利用公式：counter = counter/2 + priority**
55. **for(p = &LAST\_TASK ; p > &FIRST\_TASK ; --p)**
56. **if (\*p)**
57. **(\*p)->counter = ((\*p)->counter >> 1) +**
58. **(\*p)->priority;**
59. **// 整个设置时间片过程结束后，重新进入进程选择过程**
60. **}**
61. **// 当的上面的循环退出时，说明找到了可以切换的任务**
62. **switch\_to(next);**
63. **}**

当前进程只有调用了schedule()后才能发生[进程切换](https://www.baidu.com/s?wd=%E8%BF%9B%E7%A8%8B%E5%88%87%E6%8D%A2&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd)，因此当进程再次被选中执行后，都是从switch\_to()中ljmp后一条语句开始执行，即从"cmpl %%ecx,last\_task\_used\_math/n/t"语句继续，这时进程位于内核态。因此进程从就绪态进入的都是内核运行态。但有一个例外，进程产生后第一次被调度执行。

fork()产生的子进程会把父进程原cs、原eip当作初始的cs、eip，所以子进程刚刚创建时处于用户态。第一次进程被调度时，从就绪态进入的是用户运行态。以后进入的都是内核运行态。

第二篇  Switch\_to解析

最近看linux0.11源码时，看到任务切换函数switch\_to，感觉很晦涩，于是在网上查了一些资料，现在终于有些眉目，特记录于此，以方便大家参考，有什么错误或不足之处，还请大家指出~

switch\_to源码

/\*  
 \*    switch\_to(n) should switch tasks to task nr n, first  
 \* checking that n isn't the current task, in which case it does nothing.  
 \* This also clears the TS-flag if the task we switched to has used  
 \* tha math co-processor latest.  
 \*/  
#define switch\_to(n) {\  
struct {long a,b;} \_\_tmp; \  
\_\_asm\_\_("cmpl %%ecx,\_current\n\t" \  
    "je 1f\n\t" \  
    "movw %%dx,%1\n\t" \  
    "xchgl %%ecx,\_current\n\t" \  
    "ljmp %0\n\t" \  
    "cmpl %%ecx,\_last\_task\_used\_math\n\t" \  
    "jne 1f\n\t" \  
    "clts\n" \  
    "1:" \  
    ::"m" (\*&\_\_tmp.a),"m" (\*&\_\_tmp.b), \  
    "d" (\_TSS(n)),"c" ((long) task[n])); \  
}

大部分代码都很容易看懂，主要是：判断当前任务是否是要切换的任务，是则跳到标号1，即不做任何事；交换；调整等。。。

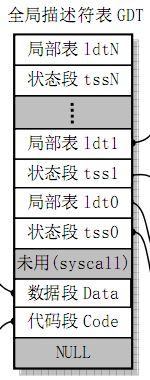
这里重点强调\_TSS(n) 和ljmp %0；

（2）\_TSS(n)，作用是生成TSS的段选择符

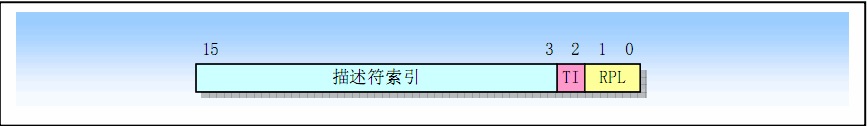
#define FIRST\_TSS\_ENTRY 4

#define \_TSS(n) ((((unsigned long) n)<<4)+(FIRST\_TSS\_ENTRY<<3))

当通过以上两个代码是不足以明白\_TSS(n)的机制，需要结合以下知识；



上图描述[linux内核](https://www.baidu.com/s?wd=linux%E5%86%85%E6%A0%B8&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd)中GDT的布局；0-nul, 1-cs, 2-ds, 3-syscall, 4-TSS0, 5-LDT0, 6-TSS1 等。。。



上图是段选择符，TI=0表示在GDT（全局描述符表）中，1表示在LDT（局部描述符表）中，RPL表示优先级；描述符索引就是在GDT中的索引；

通过上面两张图，下面分析代码，从图1可以看出第一个TSS位于索引为4的位置，于是#define FIRST\_TSS\_ENTRY 4；而FIRST\_TSS\_ENTRY<<3表示左移3位，因为TI和RPL总共占3为；((unsigned long) n)<<4为什么要左移4位呢？从图1可以看出TSS索引都是偶数，于是TI（1位）+RPL（2位）+偶数位（1）=4；通过上述组合就可以得到TSS选择子；

（）ljmp %0或（ljmp \*%0）

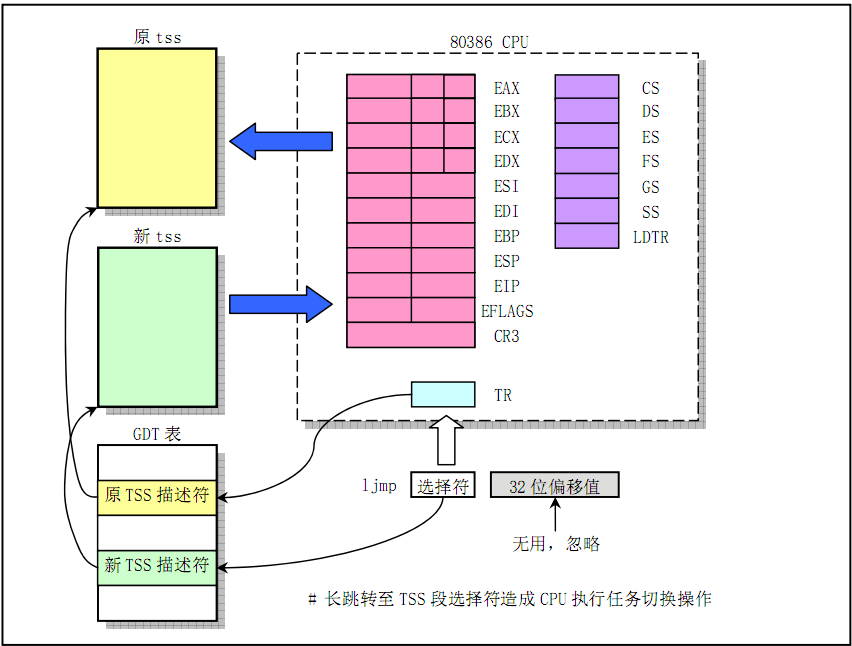
首先是为什么要加\*？这是gas语法，表示绝对跳转（与C中的\*是不同的），若程序没有加\*，则编译器会自己加上\*，可以在linux中测试；

ljmp用法说明：（很重要）

按AS手册，ljmp指令存在两种形式，即：  
   一、直接操作数跳转，此时操作数即为目标逻辑地址（选择子，偏移），即形如：ljmp $seg\_selector, $offset的方式；  
   二、使用内存操作数，这时候，AS手册规定，内存操作数必须用“\*”作前缀，即形如：ljmp \*mem48，其中内存位置mem48处存放目标逻辑地址: 高16bit存放的是seg\_selector，低32bit存放的是offset。注意：这条指令里的“\*”只是表示间接跳转的意思，与C语言里的“\*”作用完全不同。

回到源码上，ljmp %0用的ljmp的第二种用法，“ljmp \*%0”这条语句展开后相当于“ljmp \*\_\_tmp.a”，也就是跳转到地址&\_\_tmp.a中包含的48bit逻辑地址处。而按struct \_tmp的定义，这也就意味着\_\_tmp.a即为该逻辑地址的offset部分，\_\_tmp.b的低16bit为seg\_selector(高16bit无用)部分。由于在"ljmp %0"之前，"movw %%dx,%1"这条语句已经把状态段选择子"\_\_TSS(n)"的值赋给了\_\_tmp.b的低16bit。至于为什么要用\*&\_\_tmp.a，目前还不清楚，其实\*&\_\_tmp.a和\_\_tmp.a是一样的，通过汇编也可以看出；这里就先不用关心它了；

通过以上说明，可以知道了ljmp将跳转到选择子指定的地方，大致过程是，ljmp判断选择子为TSS类型，于是就告诉硬件要切换任务，硬件首先它要将当前的PC,esp,eax等现场信息保存在当前自己的TSS段描述符中,然后再将目标TSS段描述符中的pc,esp,eax的值拷贝至对应的寄存器中.当这些过程全部做完以后内核就实现了内核的切换；可以参考下图：



总结：

通过以上内容，可以大致了解到任务切换的流程，switch\_to中关键是ljmp %0；

*第三篇 switch \_to详解*

*Copyright 2009 (c) benzus*

以下代码来自Linux-1.0内核 include/linux/sched.h 文件。

（注意到Linux 0.11版的内核基本上也同样是这段代码，所以本文也同样适用于0.11内核）

01   #define switch\_to(n) { /   
02   struct (long a,b;} \_\_tmp; /   
03   \_\_asm\_\_("cmpl %%ecx,current /n/t" /   
04   "je 1f/n/t" /   
05   "xchgl %%ecx, current/n/t" /   
06   "movw %%dx, %1/n/t" /   
07   "ljmp \*%0/n/t" /   
08   "cmpl %%ecx, %2/n/t" /   
09   "jne 1f/n/t" /   
10   "clts/n" /   
11 "1:" /   
12 ::"m" (\*&\_\_tmp.a), "m" (\*&\_\_tmp.b), /   
13 "m" (last\_task\_used\_math),"d" \_TSS(n), "c" ((long) task[n])); /   
14 } 

**注释**：这是一个嵌入式汇编宏，作用是从当前任务切换到任务n，在进程调度程序中被调用。下面我来逐行注释它。

第2行定义了一个结构，包含2个long类型整数。

第3行将task[n]与current比较，其中task[n]是要切换到的任务，current是当前任务；

第4行说明，如果要切换到的任务是当前任务，则跳到标号1，即结束，什么也不做，否则继续执行下面的代码。

第5行交换两个操作数的值，相当于C代码的：current = task[n] ，ecx = 被切换出去的任务（原任务）；

第6行将新任务的TSS选择符赋值给 \_\_tmp.b；

第7行是理解任务切换机制的关键。长跳转至 \*&tmp，造成任务的切换。AT&T语法的ljmp相当于Intel语法的 jmp far SECTION : OFFSET，在这里就是将（IP）<-\_\_tmp.a,(CS)<-\_\_tmp.b,它的绝对地址之前加星号（"\*"）。当段间指令jmp所含指针的选择符指示一个可用任务状态段的TSS描述符时，将造成任务切换。那么CPU怎么识别描述符是TSS描述符而不是其他描述符呢？这是因为所有描述符（一个描述符是64位）中都有4位用来指示该描述符的类型，如描述符类型值是9或11都表示该描述符是TSS描述符。好了，CPU得到TSS描述符后，就会将其加载到任务寄存器TR中，然后根据TSS描述符的信息（主要是基址）找到任务的tss内容（包括所有的寄存器信息，如eip），根据其内容就可以开始新任务的运行。我们暂且把这个恢复所有寄存器状态的过程称为恢复寄存器现场。

第8~10行是判断原任务上次是否使用过协处理器，若是，则清除寄存器CR0的TS标志。

第2个难点是：在第7行执行后，完成任务切换（即切换到新的任务里执行）；当任务切换回来后才会继续执行第8行！下面详解其原因。

既然任务切换时CPU会恢复寄存器现场，那么它当然也会保存寄存器现场了。这些寄存器现场都会被写入原任务的tss结构里，值得注意的是，EIP会指向引起任务切换指令（第7行）的下一条指令（第8行），所以，很明显，当原任务[有朝一日](https://www.baidu.com/s?wd=%E6%9C%89%E6%9C%9D%E4%B8%80%E6%97%A5&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd)再次被调度运行时，它将从EIP所指的地方（第8行）开始运行。