**[linux进程调度与进程切换](http://blog.csdn.net/clp_csdnid/article/details/51161769)**

**一、[Linux](http://lib.csdn.net/base/linux" \o "Linux知识库" \t "_blank)进程调度时机主要有**：

（1）主动调度：

* 进程的运行状态发生变化时，例如等待某些事件而进入睡眠态；
* 设备驱动程序

       主动调度随时都可以进行，一个进程可以调用schedule() 启动一次调度。从应用的角度来看，用户空间放弃运行是可见的，而在内核空间放弃运行是不可见的，它隐藏在其他可能受阻的系统调用中。几乎所有设计外设的的系统调用都可能受阻，如read(),write()等。

       进程要调用sleep()或exit()等函数进行状态转换，这些函数会主动调用调度程序进行进程调度

当设备驱动程序执行长而重复的任务时，直接调用调度程序。在每次反复循环中，驱动程序都检查need\_resched的值，如果必要，则调用调度程序schedule()主动放弃CPU。

（2)被动调度：

* 当前进程的时间片用完（会发生一个时钟中断）
* 进程从中断、异常及系统调用返回到用户态时

       不管是从中断、异常还是系统调用返回，最终都调用ret\_from\_sys\_call（），由这个函数进行调度标志的检测，如果必要，则调用调用调度程序。在进入内核处理的时候，可能发生嵌套，这段时间可能使一些在睡眠态中等待的进程进入就绪态，如果这些进程的优先级比当前进程的优先级高，那么在从内核态回到用户态的时候自然就需要让出CPU，让高优先级的任务运行。

       每个时钟中断发生时，由三个函数协同工作，共同完成进程的选择和切换，它们是：schedule()、do\_timer()及ret\_form\_sys\_call()。我们先来解释一下这三个函数：

schedule（）：进程调度函数，由它来完成进程的选择

do\_timer（）：启动定时器，在时钟中断服务程序中被调用，是时钟中断服务程序的主要组成部分，该函数被调用的频率就是时钟中断的频率即每秒钟100次（简称100赫兹或100Hz）；

ret\_from\_sys\_call（）：系统调用返回函数。当一个系统调用或中断完成时，该函数被调用，用于处理一些收尾工作，例如信号处理、核心任务等等。

ret\_from\_sys\_call()函数中有如下几行：

cmpl $0, \_need\_resched

jne reschedule

……

restore\_all:

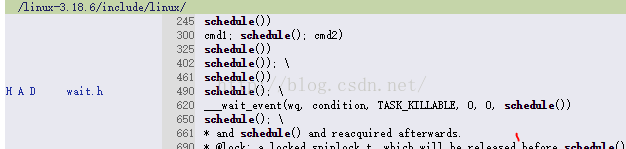
RESTORE\_ALL

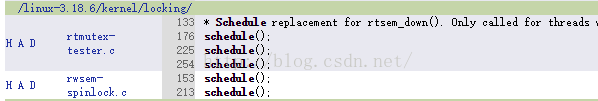
reschedule:

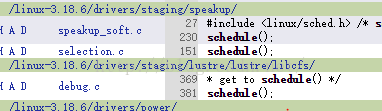
call SYMBOL\_NAME(schedule)

jmp ret\_from\_sys\_call

在[内核代码](http://codelab.shiyanlou.com/s?n=25&start=225&sort=relevancy&q=schedule&project=linux-3.18.6" \t "_blank)中搜索schedule() 可以找到调度函数被调用的位置：

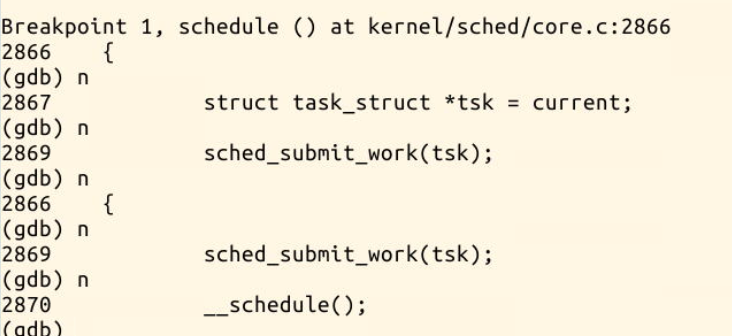






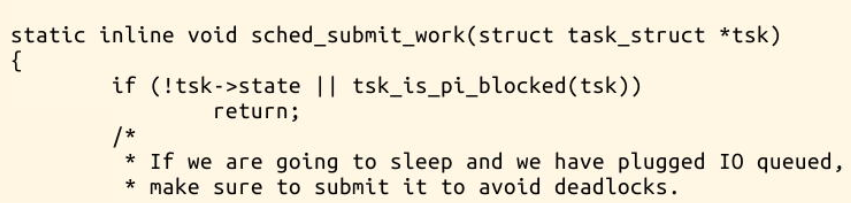
http://img.blog.csdn.net/20160415200538889?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQv/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/Center

**二、GDB追踪schedule()函数的执行**

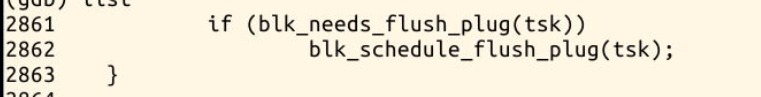


进程调度时，首先进入schedule()函数，将一个task\_struct结构体的指针tsk赋值为当前进程。 然后调用sched\_submit\_work(tsk) 我们进入这个函数，查看一下做了什么工作

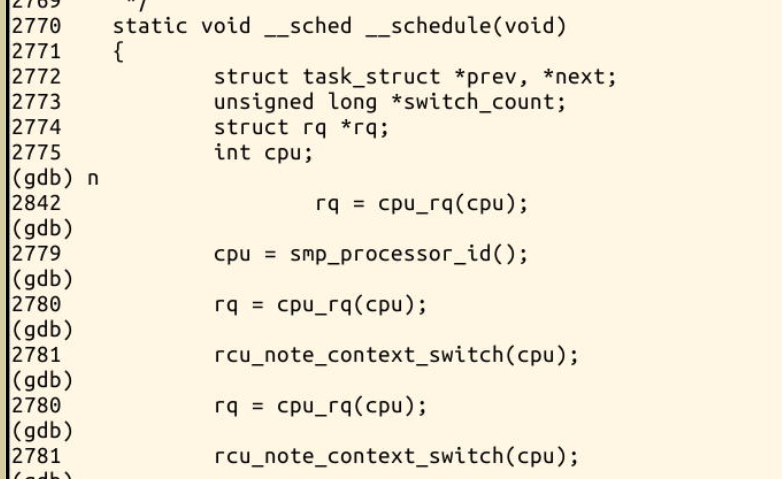
在执行到sched\_submit\_work时，输入si进入函数。



可以看到这个函数时检测tsk->state是否为0 （runnable）若为运行态时则返回， tsk\_is\_pi\_blocked(tsk),检测tsk的死锁检测器是否为空，若非空的话就return。



然后检测是否需要刷新plug队列，用来避免死锁。 sched\_submit\_work主要是来避免死锁。   
然后我们进入\_\_schedule()函数。



**三、schedule函数分析：**

1、在进程却换前，scheduler做的事情是用某一个进程替换当前进程。  
（1）关闭内核抢占，初始化一些局部变量。

need\_resched:

preempt\_disable( );

prev = current;

rq = this\_rq( );

当前进程current被保存在prev,和当前CPU相关的runqueue的地址保存在rq中。  
（2）检查prev没有持有big kernel lock.  
   if (prev->lock\_depth >= 0)  
    up(&kernel\_sem);  
   Schedule没有改变lock\_depth的值，在prev唤醒自己执行的情况下，假如lock\_depth的值不是负的，prev需要重新获取kernel\_flag自旋锁。所以大内核锁在进程却换过程中是自动释放的和自动获取的。  
（3）调用sched\_clock( )，读取TSC，并且将TSC转换成纳秒，得到的timestamp保存在now中，然后Schedule计算prev使用的时间片。

now = sched\_clock( );

run\_time = now - prev->timestamp;

if (run\_time > 1000000000)

    run\_time = 1000000000;

（4）在察看可运行进程的时候，schedule必须关闭当前CPU中断，并且获取自旋锁保护runqueue.  
            spin\_lock\_irq(&rq->lock);  
（5）为了识别当前进程是否已终止，schedule检查PF\_DEAD标志。  
            if (prev->flags & PF\_DEAD)    prev->state = EXIT\_DEAD;  
（6）Schedule检查prev的状态，假如他是不可运行的，并且在内核态没有被抢占，那么从runqueue删除他。但是，假如prev有非阻塞等待信号 并且他的状态是TASK\_INTERRUPTBLE,配置其状态为TASK\_RUNNING，并且把他留在runqueue中。该动作和分配CPU给 prev不相同，只是给prev一个重新选择执行的机会。

if (prev->state != TASK\_RUNNING &&

    !(preempt\_count() & PREEMPT\_ACTIVE)) {

    if (prev->state == TASK\_INTERRUPTIBLE && signal\_pending(prev))

        prev->state = TASK\_RUNNING;

    else {

        if (prev->state == TASK\_UNINTERRUPTIBLE)

            rq->nr\_uninterruptible++;

        deactivate\_task(prev, rq);

    }

}

deactivate\_task( )是从runqueue移除进程：

rq->nr\_running--;

dequeue\_task(p, p->array);

p->array = NULL;

（7）检查runqueue中进程数，  
   A: 假如有多个可运行进程，调用dependent\_sleeper( )函数。一般情况下，该函数立即返回0，但是假如内核支持超线程技术，该函数检查将被运行的进程是否有比已运行在同一个物理CPU上一个逻辑CPU上的兄 弟进程的优先级低。假如是，schedule拒绝选择低优先级进程，而是执行swapper进程。

if (rq->nr\_running) {

if (dependent\_sleeper(smp\_processor\_id( ), rq))

{        next = rq->idle;

         goto switch\_tasks;

}

}

   B:假如没有可运行进程，调用idle\_balance( )，从其他runqueue队列中移动一些进程到当前runqueue，idle\_balance( )和load\_balance( )相似。

if (!rq->nr\_running) {    idle\_balance(smp\_processor\_id( ), rq);

if (!rq->nr\_running) {        next = rq->idle;

 rq->expired\_timestamp = 0;

wake\_sleeping\_dependent(smp\_processor\_id( ), rq);

if (!rq->nr\_running)            goto switch\_tasks;

}

}

   假如idle\_balance( )移动一些进程到当前runqueue失败，schedule( )调用wake\_sleeping\_dependent( )重新唤醒空闲CPU的可运行进程。  
   假设schedule( )已决定runqueue中有可运行进程，那么他必须检查可运行进程中至少有一个进程是激活的。假如没有，交换runqueue中active 和expired域的内容，任何expired进程变成激活的，空数组准备接受以后expire的进程。

if (unlikely(!array->nr\_active)) {

       / \* Switch the active and expired arrays.  \*/

       schedstat\_inc(rq, sched\_switch);

       rq->active = rq->expired;

       rq->expired = array;

       array = rq->active;

       rq->expired\_timestamp = 0;

       rq->best\_expired\_prio = MAX\_PRIO;

    }

（8）查找在active prio\_array\_t数组中的可运行进程。Schedule在active数组的位掩码中查找第一个非0位。当优先级列表不为0的时候，相应的位掩码 北配置，所以第一个不为0的位标示一个有最合适进程运行的列表。然后列表中第一个进程描述符被获取。

idx = sched\_find\_first\_bit(array->bitmap);

    queue = array->queue + idx;

    next = list\_entry(queue->next, task\_t, run\_list);

   现在next指向将替换prev的进程描述符。  
（9）检查next->activated，他标示唤醒进程的状态。  
（10）假如next是个普通进程，并且是从TASK\_INTERRUPTIBLE 或TASK\_STOPPED状态唤醒。Scheduler在进程的平均睡眠时间上加从进程加入到runqueue开始的等待时间。

if (!rt\_task(next) && next->activated > 0) {

        unsigned long long delta = now - next->timestamp;

        if (unlikely((long long)(now - next->timestamp)

            delta = 0;

        if (next->activated == 1)

            delta = delta \* (ON\_RUNQUEUE\_WEIGHT \* 128 / 100) / 128;

        array = next->array;

        new\_prio = recalc\_task\_prio(next, next->timestamp + delta);

        if (unlikely(next->prio != new\_prio)) {

            dequeue\_task(next, array);

            next->prio = new\_prio;

            enqueue\_task(next, array);

        } else

            requeue\_task(next, array);

    }

    next->activated = 0;

   Scheduler区分被中断或被延迟函数唤醒的进程和被系统调用服务程式或内核线程唤醒的进程。前者，Scheduler加整个runqueue等待时间，后者只加一部分时间。  
2、进程却换时，Scheduler做的事情：  
现在，Scheduler已确定要运行的进程。  
（1）访问next的thread\_info,他的地址保存在next进程描述符的顶部。

switch\_tasks:

  if (next == rq->idle)

     schedstat\_inc(rq, sched\_goidle);

  prefetch(next)

（2）在替换prev前，执行一些管理工作

clear\_tsk\_need\_resched(prev);

rcu\_qsctr\_inc(task\_cpu(prev));

   clear\_tsk\_need\_resched清除prev的TIF\_NEED\_RESCHED，该动作只发生在Scheduler是被间接调用的情况。  
（3）减少prev的平均睡眠时间到进程使用的cpu时间片。

    prev->sleep\_avg -= run\_time;

    if ((long)prev->sleep\_avg

        prev->sleep\_avg = 0;

    prev->timestamp = prev->last\_ran = now;

（4）检查是否prev和next是同一个进程，假如为真，放弃进程却换，否则，执行（5）

   if (prev == next) {

     spin\_unlock\_irq(&rq->lock);

     goto finish\_schedule;

   }

（5） 真正的进程却换  
       next->timestamp = now;  
       rq->nr\_switches++;  
       rq->curr = next;  
       ++\*switch\_count;  
       prepare\_task\_switch(rq, next);  
       prev = context\_switch(rq, prev, next);  
   context\_switch 建立了next的地址空间，进程描述符的active\_mm指向进程使用的地址空间描述符，而mm指向进程拥有的地址空间描述符，通常二者是相同的。但是 内核线程没有自己的地址空间，mm一直为NULL。假如next为内核线程，context\_switch确保next使用prev的地址空间。假如 next是个正常的进程，context\_switch使用next的替换prev的地址空间。

    struct mm\_struct \*mm = next->mm;

    struct mm\_struct \*oldmm = prev->active\_mm;

    if (unlikely(!mm)) {

       next->active\_mm = oldmm;

       atomic\_inc(&oldmm->mm\_count);

       enter\_lazy\_tlb(oldmm, next);

    } else

    switch\_mm(oldmm, mm, next);

  假如prev是个内核线程或正在退出的进程，context\_switch在runqueue的prev\_mm中保存prev使用的内存空间。

    if (unlikely(!prev->mm)) {

       prev->active\_mm = NULL;

       WARN\_ON(rq->prev\_mm);

       rq->prev\_mm = oldmm;

    }

        调用switch\_to(prev, next, prev)进行prev和next的转换。

 3、进程转换后的工作

（1）finish\_task\_switch()：

struct mm\_struct \*mm = rq->prev\_mm;

unsigned long prev\_task\_flags;

rq->prev\_mm = NULL;

 prev\_task\_flags = prev->flags;

finish\_arch\_switch(prev);

finish\_lock\_switch(rq, prev);

if (mm)

      mmdrop(mm);

 if (unlikely(prev\_task\_flags & PF\_DEAD))

     put\_task\_struct(prev)

   假如prev是内核线程，runqueue的prev\_mm保存prev的内存空间描述符。 Mmdrop减少内存空间的使用数，假如该数为0，该函数释放内存空间描述符，连同和之相关的页表和虚拟内存空间。 finish\_task\_switch()还释放runqueue的自选锁，开中断。

（2）最后

prev = current;

if (unlikely(reacquire\_kernel\_lock(prev)

      goto need\_resched\_nonpreemptible;

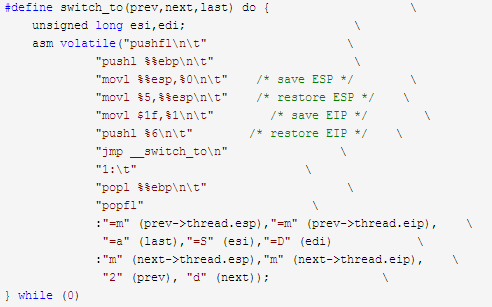
preempt\_enable\_no\_resched();

if (unlikely(test\_thread\_flag(TIF\_NEED\_RESCHED)))

    goto need\_resched;

   schedule获取大内核块，重新使内核能够抢占，并且检查是否其他进程配置了当前进程的TIF\_NEED\_RESCHED，假如真，重新执行schedule，否则该程式结束

**三、switch\_to 汇编代码分析：**



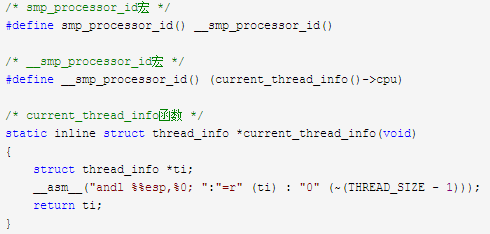
该宏的工作步骤大致如下：

1. prev的值送入eax，next的值送入edx（这里我从代码中没有看出来，原著上如是写，可能是从调用switch\_to宏的switch\_context或schedule函数中处理的）。
2. 保护prev进程的eflags和ebp寄存器内容，这些内容保存在prev进程的内核堆栈中。
3. 将prev的esp寄存器中的数据保存在prev->thread.esp中，即将prev进程的内核堆栈保存起来。
4. 将next->thread.esp中的数据存入esp寄存器中，这是加载next进程的内核堆栈。
5. 将数值1保存到prev->thread.eip中，该数值1其实就是代码中"1:\t"这行中的1。为了恢复prev进程执行时用。
6. 将next->thread.eip压入next进程的内核堆栈中。这个值往往是数值1。
7. 跳转到\_\_switch\_to函数处执行。
8. 执行到这里，prev进程重新获得CPU，恢复prev进程的ebp和eflags内容。
9. 将eax的内容存入last参数（这里我也没看出来，原著上如是写，只是在\_\_switch\_to函数中返回prev，该值是放在eax中的）。

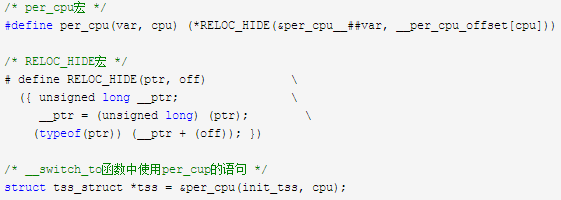
**\_\_switch\_to函数**

　　\_\_switch\_to函数采用FASTCALL调用模式，利用eax和edx传入两个参数的值。由于\_\_switch\_to中用了很多其他函数，这里首先介绍相关函数和宏，然后再讨论\_\_switch\_to函数。

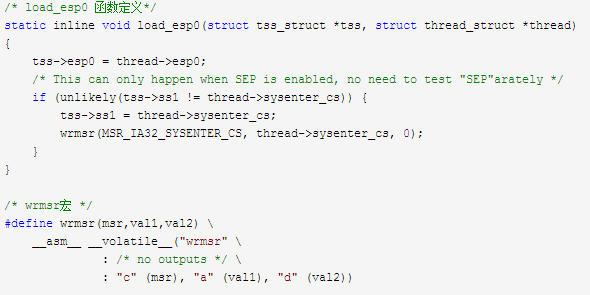
   smp\_process\_id宏展开如下。该宏得到当前代码运行在哪个CPU上，返回CPU编号。current\_thread\_info函数返回当前运行着的进程的thread\_info结构地址，该函数中让esp的值与上（THREAD\_SIZE - 1）的逆，实际上，THREAD\_SIZE - 1 = 8192 - 1 = 8191 = 0x1FFF，取反就是0xE000，就是让esp低13位清零，这样就通过内核堆栈得到thread\_info结构地址。然后从该结构中得到cpu编号。



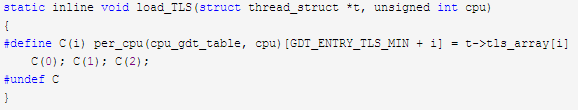
per\_cup宏展开如下。\_\_switch\_to函数中传入init\_tss参数和CPU编号cpu给per\_cpu宏，然后得到该CPU上的TSS指针。



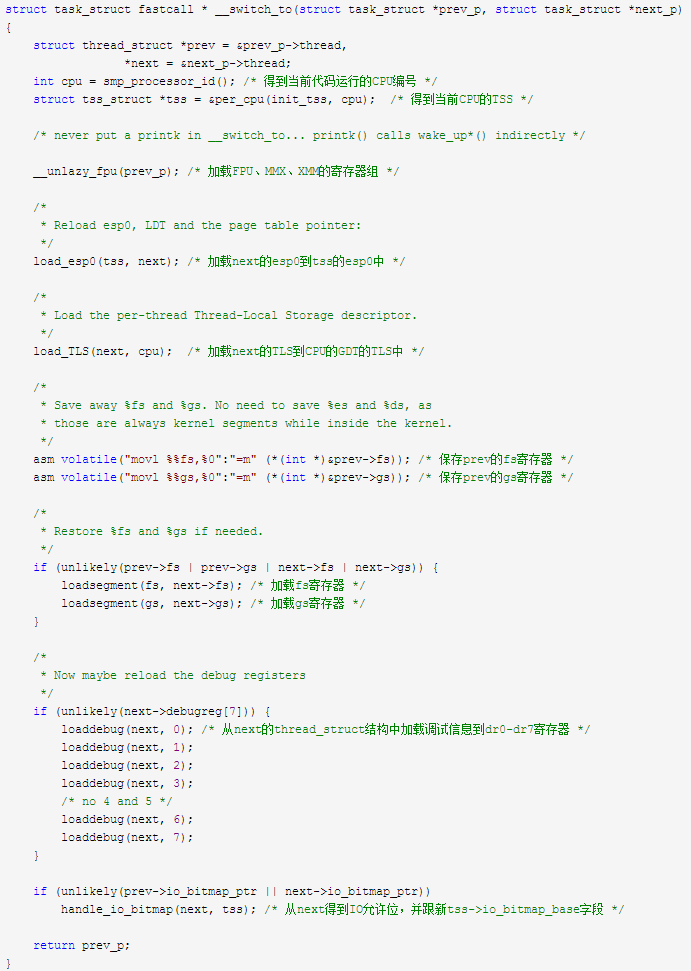
load\_esp0函数定义如下。这里从thread\_struct结构中加载esp0到tss中，即即将执行进程的esp0。当SEP打开时，还用wrmsr写入新的CS段选择子——即sysenter指令执行后的代码段。



　　Load\_TLS函数定义如下。该函数中使用宏C(i)和per\_cpu，先通过per\_cpu得到CPU的GDT所在内存的地址，然后将3个thread\_struct结构中的tls\_array加载到GDT的TLS段中。其中GDT\_ENTRY\_TLS\_MIN=6，这正是3个TLS在GDT中的索引，因此Load\_TLS就加载了GDT中3个线程局部段（TLS）。



现在来看\_\_switch\_to函数，其定义和注释如下。



四、总结

1、Linux 调度器将进程分为三类：

**交互式进程**

  此类进程有大量的人机交互，因此进程不断地处于睡眠状态，等待用户输入。典型的应用比如编辑器 vi。此类进程对系统响应时间要求比较高，否则用户会感觉系统反应迟缓。

**批处理进程**

  此类进程不需要人机交互，在后台运行，需要占用大量的系统资源。但是能够忍受响应延迟。比如编译器。

**实时进程**

  实时对调度延迟的要求最高，这些进程往往执行非常重要的操作，要求立即响应并执行。比如视频播放软件或飞机飞行控制系统，很明显这类程序不能容忍长时间的调度延迟，轻则影响电影放映效果，重则机毁人亡。

  根据进程的不同分类 Linux 采用不同的调度策略。对于实时进程，采用 FIFO 或者 Round Robin 的调度策略。对于普通进程，则需要区分交互式和批处理式的不同。传统 Linux 调度器提高交互式应用的优先级，使得它们能更快地被调度。而 CFS 和 RSDL 等新的调度器的核心思想是“完全公平”。这个设计理念不仅大大简化了调度器的代码复杂度，还对各种调度需求的提供了更完美的支持。

2、调度的发生主要有两种方式：

**主动式调度(自愿调度)**

  在内核中主动直接调用进程调度函数schedule()，当进程需要等待资源而暂时停止运行时，会把状态置于挂起（睡眠），并主动请求调度，让出cpu。

**被动式调度**（抢占式调度、强制调度）

  用户抢占（在中断返回等过程中，从内核空间回到用户空间的时候）

  内核抢占（在内核进程中发生抢占）

3、schedule() --> context\_switch() --> switch\_to --> \_\_switch\_to()

4、、jmp和call 的区别

  call会把他的下一条指令的地址压入堆栈，然后跳转到他调用的开始处，同时ret会自动弹出返回地址。

  JMP只是简单的跳转

  call的本质相当于push+jmp  ret的本质相当于pop+jmp