# 进程

Linux使用轻量级进程提供多线程支持。在内核中，线程也是一种特殊的进程。两个轻量级的进程可以共享一些资源，比如地址空间、打开的文件等等。其中一个修改了共享资源，另一个可以立即查看。将线程和轻量级进程关联，这样就做到了线程之间的数据共享，同时内核又可以独立调度每个线程。

## 进程描述符task\_struct结构

state成员：用来保存进程状态。

TASK\_RUNNING：可运行状态。要么在执行，要么准备执行。

TASK\_INTERRUPUTIBLE：可中断的等待状态。等待某个资源释放或者等待某个条件达成，回到TASK\_RUNNING状态。

TASK\_UNINTERRUNPTIBEL：不可中断状态。信号传递给这个状态的进程不能唤醒。

TASK\_STOPPED：暂停状态。收到SIGSTOP、SIGTSTP、SIGTTIN、SIGTTOU信号会进入这个状态。

TASK\_TRACED：跟踪状态。进程的执行被debugger程序暂停。

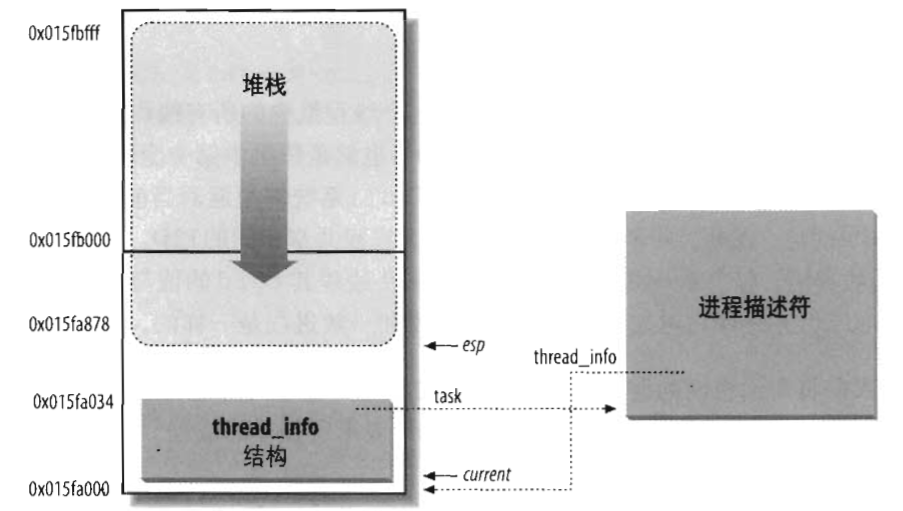
下面两种状态既可以保存在state成员也可以保存在exit\_state成员中。

EXIT\_ZOMBIE：僵死状态。进程的执行被终止，但父进程还没有通过wait系统调用来回收被终止进程的信息。

EXIT\_DEAD：僵死撤销状态。父进程调用了wait系统调用，为了防止该进程上的其他线程重复调用wait，把僵死状态的进程设置为这个状态。

## 进程标识符

task\_struct的pid成员，用来保存进程的编号。新创建进程的PID一般是前一个进程的PID+1。32位系统的进程pid最大值为32767。64位系统的pid最大值为4194303。



这张图中，2页内存一共8KB存放了thread\_info和进程在内核态的栈空间。thread\_info在内存的起始位置，栈空间则从大到小增长。同时thread\_info和task\_struct通过成员互相关联。

esp是CPU的栈指针寄存器，用来存放栈顶的地址。每次用户态切换到内核态，进程的内核栈都是空的。一旦数据写入内核栈，esp就递减。

在内核中通过一个联合体来表述整个内核栈空间：

union thread\_union {

struct thread\_info thread\_info;

unsigned long stack[2048]; //如果栈空间是4K，那么数组下标是1024

};

将thread\_info和栈空间放在一起的好处是，可以通过esp保存的栈顶指针快速的找到thread\_info结构。如果是8K，则屏蔽esp保存地址的低13位。如果是4K，则屏蔽esp保存地址的低12位。这就是current\_thread\_info()的实现。task\_struct在thread\_info中的偏移量为0，因此也就同样快速的找到了task\_struct。

## 进程链表

内核通过双向链表来管理所有的进程。进程链表的头部是init\_task。从linux2.6开始，为了能够更快速的找到当前最佳的可运行进程，内核根据优先级将可运行的进程放在多个不同优先级的进程链表中。用prio\_array\_t来管理这些进程链表，其成员有：

**int nr\_active;** //链表中进程描述符的数量

**unsigned long [5] bit\_map;** //优先级位图，如果某个优先级的链表不为空，则在相应的位上置1。

**struct list\_head [140] queue;** //一共140个不同优先级的链表数组。

## 进程关系

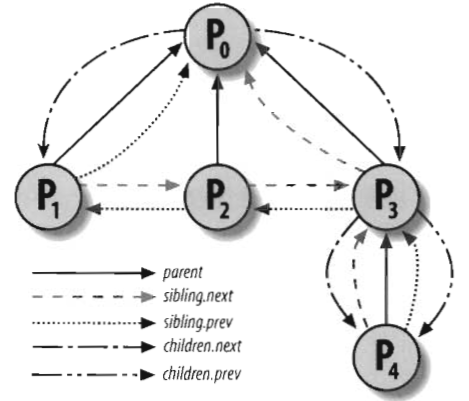
task\_struct通过以下成员来建立进程之间的关系。

**struct task\_struct \*real\_parent;** //指向创建当前进程的进程的描述符。对于子进程，就指向其父进程。对于父进程则指向进程1即init\_task的描述符。

**struct task\_struct \*parent;** //指向当前进程的父进程，子进程结束的时候必须向父进程发送信号。值通常与real\_parent一致，在部分特殊情况下不同，比如另一个进程发出ptrace()系统调用请求。

**struct list\_head children;** //当前进程创建的子进程。

**struct list\_head sibling;** //兄弟进程的链表。链表中进程的父进程和当前进程一样。



有些系统调用比如kill()要求能够通过进程标识符PID找到其进程描述符。为了提高查询效率，内核通过哈希表来保存PID和进程描述符的对应关系。

# 创建进程

linux创建进程时使用的三个技术：

**写时复制技术**：允许父子进程都相同的物理页。只有当两者有要写内存页的时候，才会拷贝新的内存页，并将其分配给需要写入的进程。

**轻量级进程**：允许父子进程共享数据，比如页表（用户态地址空间）、打开的文件、信号处理等。通过clone()系统调用创建。

**vfork()**：创建的子进程与父进程共享内存空间，父进程阻塞等待子进程执行完成。

相关的系统调用：

# 调度

进程可以分为I/O消耗型和CPU消耗型。I/O消耗型进程需要调度系统能够快速响应但不需要很长的执行时间，而CPU消耗型则要求能够尽量分配更多的执行时间，减少调度切换。Linux进程调度系统会更多的偏向I/O消耗型进程，以追求有更好的用户交互体验。

## Linux采用两种进程优先级：

第一种：nice值，范围-20~+19，nice值越大优先级越低。

第二种：实时优先级，可配置，范围0~99，值越大优先级越高。

实时优先级进程优先级一定高于普通进程。因此两种进程优先级不会冲突。

## Unix进程调度策略：

根据进程的nice值为进程分配时间片，这样做会产生以下问题：

1. nice值直接映射到进程时间片，很难保证时间分布的合理性。假设nice为0分配的时间片为100ms，nice为19分配的时间片为5ms。如果此时系统中的进程nice值都为19，那么5ms就会进行一次进程切换，进程切换的开销很大。
2. 可能会出现nice值为0和1的进程时间片为100ms和95ms的区分相差无几。而nice值为18和19的进程时间片为10ms和5ms，两者相差了一倍。导致nice值+1和-1的效果完全取决于初始nice值为多少。
3. 时间片会随着定时器的节拍而改变，并不是绝对的。这样就无法保证相同的nice值都对应绝对的时间片。
4. 为了能够更快的唤醒进程，需要对新的要唤醒的进程提升优先级，但是这可能会打破“公平性”。

## Linux进程调度策略：

完全公平调度(CFS)：针对普通进程的调度策略。根据系统中进程的总数计算每个进程能够使用的CPU时间。然后再根据进程的nice值对能够使用的CPU时间进行加权。

同时CFS也限制了每个进程能够使用的CPU时间的最小值（默认1ms）。以保证当系统中进程数量太多的时候，每个进程分配到的CPU使用时间过小，从而产生不可接受的进程切换消耗。

nice值的加权，不是根据nice值的绝对值，而是相对值来进行加权。即nice值为5和15之间的CPU使用时间差距与nice值为15和25之间的CPU使用时间差距一样。

nice值转为权重的公式为：

weight = 2014 / (1.25 ^ nice)

因此nice值对于加权的影响是指数上的，这个可以保证在nice值为负数的情况下，仍然可以得到正数的加权值。

## 任务状态段

Task State Segment（TSS）。linux为每个CPU创建了一个TSS。