## 4.1 进程四要素

（1）有一段程序供运行（2）有进程专用的系统堆栈空间（3）内核中有一个task\_struct结构（4）有专用的用户空间与用户堆栈

Intel在i386系统结构的设计中考虑了进程的管理和调度，包括新增设的段TSS，是一个104字节的数据结构，记录一个任务的关键信息；新增设的任务寄存器TR，指向当前任务的TSS

Linux内核实际不使用i386CPU硬件提供的任务切换装置，内核只是在开始阶段设置TR，使之指向TSS，以后就再也不改变TR的内容。内核不依靠TSS保存的每个进程切换时的寄存器副本，而是将寄存器副本保存在系统堆栈中。在Linux中只使用两个运行级别0级和3级，所以TSS中另外两个堆栈指针副本也失去意义。内核随着进程的切换，只改变TSS中的SS0和ESP0，所化开销比通过装入TR新的TSS小得多。

每个进程都有一个task\_struct结构和一个系统堆栈，二者的物理存储空间在一起，内核为他们分配了两个物理页面，两个页面的底部约1KB用作task\_struct结构，剩余用作系统堆栈。

每个进程都存在于三个静态的队列中：表征“家庭关系”的树形组织；为pid查询task\_struct方便的杂凑表；以及以init\_task进程为起点的一个线性队列。

## 4.2 进程三部曲：创建、执行与消亡

子进程复制出来后，就像父进程一样接收内核的调度，并且具有相同的返回地址。子进程从fork()“返回”时，其返回值为0；而父进程从fork()返回时的返回值为子进程的pid

子进程执行execve调用，其中必有一个exit()调用，执行完后不会执行execve后面的语句

## 4.3 系统调用fork()、vfork()与clone()

fork是全部复制，父进程所有的资源全部通过数据结构的复制“遗传”给子进程，而clone()则可以将资源有选择的复制给子进程，而没有复制数据结构则通过指针的复制让子进程共享。

fork()对有关的资源都要复制而不是通过指针共享，而对于vfork()父子进程共用虚存空间，并且当子进程释放其虚存空间时唤醒父进程，至于\_\_clone这一部分由调用者决定

do\_fork()中alloc\_task\_struct()为子进程分配两个连续的物理页面，低端用作子进程的task\_struct结构，高端则用作系统空间堆栈。随后父进程中整个task\_struct就复制到子进程的数据结构中。

每个进程的task\_struct结构中，有一个指针user，指向一个user\_struct结构。一个用户常常有多个进程，所以有关这个用户的信息都是由这个user\_struct结构来保存的。需要注意的是内核线程不属于任何线程，所以其指针user为0

在内核中有一个user\_struct结构指针的数组uidhash杂凑表，对用户名进行杂凑运算找到该用户的user\_struct结构

系统调用\_\_clone()的主要作用是创建一个线程，可以是内核线程，也可以是用户线程。创建用户空间线程时，可以给定子线程的用户空间堆栈的位置，作为参数child\_stack会出现在寄存器ecx中

进程的执行域下面有一个结构module，指向与这个进程相关的模块，如果模块的计数器表明还有进程需要使用这个模块，这些模块就不能释放。因而，在使用fork创建进程时，会递增模块中的计数。

当有标志位CLONE\_PID时，父子进程共有一个进程号。但是，只有0号进程，也就是系统中的原始进程，才允许这样调用\_\_clone()

由于do\_fork中的get\_pid()产生新的pid的操作必须是独占的，进程可能进不了临界区而睡眠，因而需要事先将进程状态设为TASK\_UNINTERRUPTIBLE

进程号0~299是为系统进程保留的，主要用于各种“保护神”进程

系统调用fork()通过sys\_fork()进入do\_fork()时，其clone\_flag为SIGCHLD，也就是所有标志位为0，所以copy\_files()、copy\_fs()、copy\_sighand()以及copy\_mm()全部执行，这四项资源全部复制。而系统调用vfork()通过sys\_vfork()进入do\_fork()时，其clone\_flag为VFORK|CLONE\_VM|SIGHLD，所以只执行了copy\_files()、copy\_fs()、以及copy\_sighand()，而copy\_mm由于CLONE\_VM为1，只是通过指针共享其父进程的存储空间。至于\_\_clone()取决于具体的参数

fork如何实现两次返回？

fork复制进程后，只是将子进程的返回值设为0，将子进程挂载到待运行队列中，实际没有两次返回，系统调用结束后系统调用一个进程运行。

## 4.4 系统调用execve()

首先会通过getname()将绝对路径名拷入系统空间，并分配一个物理页面作为缓冲区，注意系统空间堆栈+task\_struct结构不过8Kb，因此需要另外分配空间

内核为可执行程序的装入定义了一个数据结构linux\_binprm。以便将运行一个可执行文件时所需的信息组织在一起

不管是elf 还是a.out，在开头128字节都包含了可执行文件属性的必要而充分的信息

do\_execve在将可执行程序的所有必要信息收集到bprm结构中后，会通过search\_binary\_handler()找到对应的load\_binary

### 4.4.1 a.out格式目标文件的装载和投运

采用fork()函数创建的子进程用户空间，会在执行sys\_execve()->do\_execve()->search\_binary\_handle()->load\_aout\_binary->flush\_old\_execve时全部释放，因此可以考虑采用vfork()，然而由于此时父子进程共享用户空间，共享包括页面表在内的所有数据结构，就无法实现”copy\_on\_write”，子进程对用户堆栈的操作会直接影响父进程，所以在sys\_vfork()调用do\_fork()时结合使用CONE\_VFORK标志位，使父进程在创建完子进程后进入睡眠。

由于vfork公用父进程的地址空间，需要保证父进程在创建子进程后就进入睡眠，而且子进程需要通过执行execve执行另一个目标程序，或者通过exit()结束来释放其共享的内存空间,子进程不应该返回，否则会破坏父进程的返回地址

### 4.2.2 文字形式可执行文件的执行

以shell文件为例，实际上是将shell文件本身作为解释程序（即二进制程序）的参数来装载运行的。

## 4.5 系统调用exit()与wait4()

CPU在进入do\_exit()后，当前进程就结束了，不会从这个函数中返回

首先判断是否在中断或者是idle进程和init进程，其次通过del\_timer\_sync()将进程从定时器队列中删除，之后会释放各种临界资源

对于即将结束的进程保留最小额度资源—task\_struct结构和系统堆栈所在的两个页面？

1.也许在子进程的task\_struct结构中保留有有用的统计信息，让父进程来料理后事，不会造成信息丢失

2.也许在系统调度进程的间隙，发生中断或者异常需要使用到系统堆栈

父进程也存在生父与养父的关系，一个进程在创建之初生父与养父是一致的，当一个进程被另一个进程跟踪时，其养父变为那个跟踪的进程；如果一个进程先于子进程去世，那么它的子进程将托付给同一线程组中的下一个线程，否则托付给init进程

使用同一个控制终端的进程属于同一个session；在同一条shell命令或执行程序中启动多个进程，形成一个组；（session和组是不同的概念）不过每个session和进程组中都有一个为主的，最早创建的进程，这个进程的pid就是session和进程组的代号

实际上使exit()不返回的正是schedule()

sys\_wait4()中父进程将释放子进程的资源

## 4.6 进程的调度与切换

一个好的系统调度机制需要完成三种不同应用的需求

1.交互式应用，着重于系统响应速度 当延迟超过150ms，用户就会明显的感觉到

2.批处理应用，着重于任务完成时间

3.实时应用 ，着重于响应速度和系统能否在规定时间内完成

调度的时机：

1.自愿的调度：在内核中可以通过schedule()启动一次调度，在用户空间可以通过系统调用完成这一目的，包括可能受阻的系统调用open,close,write

2.非自愿的调度：强制的发生在每次从系统调用返回的前夕，以及每次从中断或者异常处理返回用户空间的前夕

注意：发生在内核中的中断或者异常不会引起系统调用

need\_resched何时设置

当前进程通过自愿让出运行以及在系统调用中由于某种原因受阻以外，主要原因就是当唤醒一个进程的时候，以及在时钟中断服务程序发现当前进程已经连续运行太久

调度的方式：有条件的可剥夺

1.当进程运行在用户空间，不管自不自愿，一旦有必要，内核就会剥夺

2.当进程运行在内核空间，尽管内核知道需要剥夺，实际上却不会发生，一直到回到用户空间的前夕才能剥夺

调度政策：以优先级为基础的调度除此之外还有三种不同的政策

1.SCHED\_FIFO适合时间要求比较强，但运行时间比较短的进程

2.SCHED\_RR 适合运行时间较长的进程

3.SCHED\_OTHER 适合交互式分时应用，为传统的调度策略

从嵌套中断返回不会调用schedule() 此时的中断返回并不返回到用户空间

在中断程序内部不能调用schedule() 只能通过将need\_resched字段设为1表达这种需求

\_\_cacheline\_aligned表示数据结构的起点应与高速缓存中的缓存线对齐

如果当前进程想要继续运行，那么在挑选候选进程时以当前进程的权值开始，这意味着对于相同权值的其他进程而言，当前进程优先

对于没有实时要求的进程SCHED\_OTHER，其权值取决于两个因数：剩余的时间配额以及进程优先级nice(19~-20),综合的权值在时间尚未用完时是两者之和

对于实时进程SCHED\_FIFO+SCHED\_RR，其权值为1000+rt\_priority（实时优先级），对于SCHED\_RR其权值与时间配额有关，当其为0时将移至队列末尾，同时由于实时进程有很大的基数，所以实时进程就绪时，非实时进程是没有机会的

当已经选择的进程的权值为0时，说明所有进程的时间配额以及耗完，这时需要重新计算每个进程的时间配额

对于新创建的进程，在调用schedule\_tail以后就直接转到ret\_from\_sys\_call返回用户空间了，没有进行进程的切换（堆栈的切换）

## 4.7 强制性调度

need\_resched何时设为1

1.在时钟中断服务程序中，发现当前进程连续运行时间过长

2.当唤醒一个睡眠中的进程时，发现被唤醒的进程比当前进程更有资格运行

3.一个进程通过系统调用改变调度政策或礼让

set\_setscheduler()改变进程的调度政策 sched\_yield进行礼让

## 4.8 系统调用nanosleep()和pause()

进程主动放弃CPU

1.几乎所有与外设有关的系统调用都有可能在执行的过程中受阻而进入睡眠，让出CPU

2.进入睡眠状态的系统调用

如果要求睡眠的时间小于2ms，而要求睡眠的进程又有实时要求（其调度政策为SCHED\_FIFO或SCHED\_RR），那这个进程不能进入睡眠，因为典型的时钟中断周期为10ms，进入睡眠可能需要10ms后才能唤醒，在这种情况下提供的是宏操作udelay()通过计数实现延时

定时器队列问题产生与解决方法

定时器队列组织最简单的方法就是将所有定时器按照到点先后链接在一起。然后每当jiffies改变就从队列头逐个检查并处理这些数据结构，直到发现第一个尚未到点的定时器，可是这样意味着每次加入就需要线性搜索，寻找恰当的链入位置

解决方案一：通过“杂凑”将这些数据结构组织成队列数组，根据定时器到点时间决定链入哪一个队列，但是如果只是简单的低位取与操作每个队列中会有许多不同到点的定时器

在linux 中采用的是时间轮的方法，共有5个杂凑队列，分别对应到点时间的5段，tv1对应到点时间的低8位，大小为2^8，也就是说每个队列对应同一个到点时间，其tv1结构中有一个index标记，每次时间中断会向前推进，当tv1结构为空后，会从对应较高6位的tv2中的一个将队列移下来，放到tv1中

## 4.9 内核中的互斥操作

互斥体(mutex)用于保护共享的易变代码，也就是，全局或静态数据。这样的数据必须通过互斥体进行保护，以防止它们在多个线程同时访问时损坏

信号量(semaphore)又称为信号灯，它是用来协调不同进程间的数据对象的，而最主要的应用是共享内存方式的进程间通信。本质上，信号量是一个计数器，它用来记录对某个资源（如共享内存）的存取状况。

从原理上讲，mutex实际上是count=1情况下的semaphore

无论是互斥锁，还是自旋锁，在任何时刻，最多只能有一个保持者，也就说，在任何时刻最多只能有一个执行单元获得锁。但是两者在调度机制上略有不同。对于互斥锁，如果资源已经被占用，资源申请者只能进入睡眠状态。但是自旋锁不会引起调用者睡眠，如果自旋锁已经被别的执行单元保持，调用者就一直循环在那里看是否该自旋锁的保持者已经释放了锁，"自旋"一词就是因此而得名。

自旋锁比较适用于锁使用者保持锁时间比较短的情况。自旋锁的效率远高于互斥锁。信号量和读写信号量适合于保持时间较长的情况，它们会导致调用者睡眠，因此只能在进程上下文使用，如果被保护的共享资源只在进程上下文访问，使用信号量保护该共享资源非常合适，但是如果被保护的共享资源需要在中断上下文访问（包括底半部即中断处理句柄和顶半部即软中断），就必须使用自旋锁。自旋锁保持期间是抢占失效的，而信号量和读写信号量保持期间是可以被抢占的。