目录

[进程管理 1](#_Toc141215318)

[进程描述符 1](#_Toc141215319)

[内核栈 1](#_Toc141215320)

[分配 进程描述符 2](#_Toc141215321)

[为什么需要thread\_info? 3](#_Toc141215322)

[thread\_union 3](#_Toc141215323)

[存放 进程描述符 4](#_Toc141215324)

[进程上下文 4](#_Toc141215325)

[进程家族树 5](#_Toc141215326)

[进程状态机 5](#_Toc141215327)

[进程创建 6](#_Toc141215328)

[写时拷贝 6](#_Toc141215329)

[fork() 6](#_Toc141215330)

[vfork() 7](#_Toc141215331)

[线程在Linux中的实现 7](#_Toc141215332)

[创建线程 7](#_Toc141215333)

[内核线程 8](#_Toc141215334)

[进程终结 8](#_Toc141215335)

[删除进程描述符 8](#_Toc141215336)

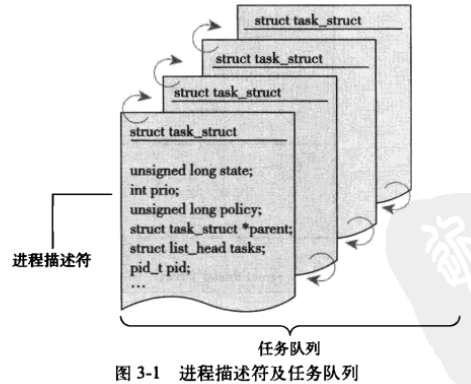
[孤儿进程造成的进退维谷 8](#_Toc141215337)

# 进程管理

1. 进程 进程是分配资源的最小单位，是处于执行器的程序以及相关资源的总称。
2. 线程 线程是处理器调度的最小单位，每个线程都拥有一个独立的程序计数器、进程栈和一组进程寄存器。

Linux系统的线程实现非常特别：它对线程和进程并不特别区分。对Linux而言，线程只不过是一种特殊的进程罢了。

## 进程描述符



task\_struct相对较大，在32位机器上大约有1.7KB.

### 内核栈

进程在内核态运行时需要自己的堆栈信息, 因此linux内核为每个进程都提供了一个内核栈kernel stack

struct task\_struct

{

// ...

void \*stack; // 指向内核栈的指针

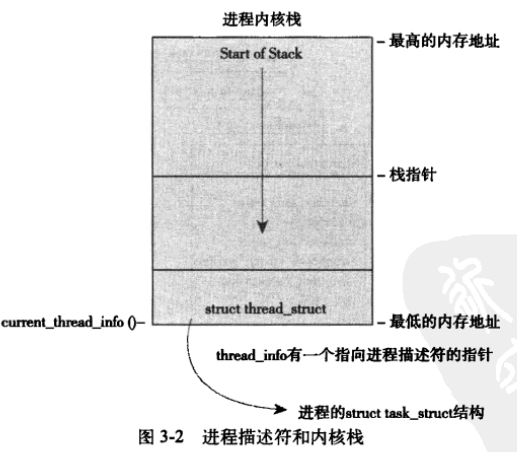
// ...

};

内核态的进程访问处于内核数据段的栈，这个栈不同于用户态的进程所用的栈。用户态进程所用的栈，是在进程线性地址空间中；而内核栈是当进程从用户空间进入内核空间时，特权级发生变化，需要**切换堆栈**，那么内核空间中使用的就是这个内核栈。因为内核控制路径使用很少的栈空间，所以只需要几千个字节的内核态堆栈。

### 分配 进程描述符

Linux通过slab分配器分配task\_struct结构，在内核栈栈底（对于向下增长的栈）或栈顶（对于向上增长的栈来说）创建一个新结构struct thread\_info，其中包含一个指向task\_struct的指针。



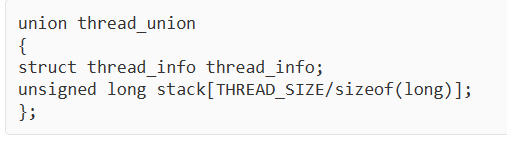
### 为什么需要thread\_info?

内核还需要存储每个进程的PCB信息, linux内核是支持不同体系的的, 但是不同的体系结构可能进程需要存储的信息不尽相同, 这就需要我们实现一种通用的方式, 我们将 **体系结构相关的部分和无关的部分进行分离**。用一种通用的方式来描述进程, 这就是struct task\_struct, 而 **thread\_info就保存了特定体系结构** 的汇编代码段需要访问的那部分进程的数据,我们在thread\_info中嵌入指向task\_struct的指针, 则我们可以很方便的通过thread\_info来查找task\_struct

### thread\_union

对每个进程，Linux内核都把两个不同的数据结构紧凑的存放在一个单独为进程分配的内存区域中：

1. 一个是内核态的进程堆栈stack
2. 另一个是紧挨着进程描述符的小数据结构thread\_info，叫做线程描述符。



1. thread\_info和内核栈虽然共用了thread\_union结构, 但是thread\_info大小固定, 存储在联合体的开始部分, 而内核栈由高地址向低地址扩展, 当内核栈的栈顶到达thread\_info的存储空间时, 则会发生栈溢出.
2. 系统的current指针指向了当前运行进程的thread\_union(或者thread\_info)的地址.
3. 进程task\_struct中的stack指针指向了进程的thread\_union(或者thread\_info)的地址, 在早期的内核中这个指针用struct thread\_info \*thread\_info来表示, 但是新的内核中用了一个更浅显的名字void \*stack, 即内核栈.

即，进程的thread\_info存储在进程内核栈的最低端

### 存放 进程描述符

内核通过一个唯一的进程标识符（process identification value）或PID来标识每个进程。为了与老版本Unix和Linux兼容，PID最大值是32768（short int短整型的最大值）。内核把每个进程的PID存放在它们各自的进程描述符中。  
在内核中，访问任务通常需要获得指向其task\_struct的指针。内核中大部分处理进程的代码都是直接通过task\_struct进行的。因此，通过current宏查找到当前正在运行进程的进程描述符的速度就显得尤为重要。硬件体系不同，该宏的实现也不同，必须针对专门的硬件体系结构做处理。

1. 有的硬件体系结构可以拿出一个专门寄存器存放当前进程的task\_struct指针，用于加快访问速度。如Power PC。
2. 而有些像x86这样的体系结构（寄存器并不富裕），在内存栈尾端创建thread\_info结构，通过计算偏移间接地查找task\_struct结构。

在x86系统中，current把栈指针后13位屏蔽掉，用来计算处thread\_info的偏移。该操作是通过current\_thread\_info()函数实现的。最后，current再从thread\_info的task域中提取并返回task\_struct的地址。

current\_thread\_info()->task;

### 进程上下文

首先用户态和内核态的区别是：此时执行的代码是用户程序代码还是内核程序代码，执行代码在内存中的位置也不同。

一般程序在用户空间执行，当执行系统调用或触发某个异常，就会陷入内核空间。此时，我们称内核“代表进程执行”并处于进程上下文中。在此上下文中current宏有效。系统调用和异常处理程序是对内核明确定义的接口。程序只有通过这些接口才能陷入内核执行——对内核的所有访问都必须通过这些接口。

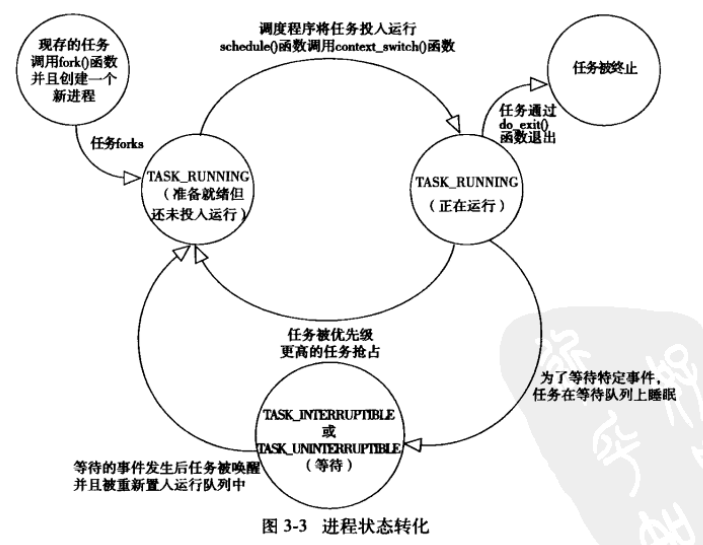
### 进程家族树

在Linux系统中，所有进程都是PID为1的init进程的后代。内核在系统启动的最后阶段启动init进程。该进程读取系统的初始化脚本并执行其他的相关程序，最终完成系统的启动。

系统中每个进程必有一个父进程，相应的也可以拥有零个或多个子进程。进程间的关系存放在进程描述符中。每个task\_struct都包含一个指向其父进程task\_struct、叫做parent的指针，还包含一个成为children的子进程链表。可以通过这种继承体系从系统的任何一个进程出发查找到任意指定的其他进程。

init进程的进程描述符是作为init\_task静态分配的。

### 进程状态机



TASK\_UNINTERRUPTIBLE（不可中断）：就算是接受信号也不会被唤醒或准备投入运行。这个状态通常在进程 **必须在等待时不受干扰** 或 **等待事件很快就会发生** 时出现。由于处于此状态的任务对信号不做响应，导致执行ps（1）命令时发现被标为D状态的进程不能被杀死，因为不可能发送给该进程SIGKILL信号。退一步讲，即使有方法，终结这样一个任务也不是一个明智选择，因为该任务可能正在执行重要的操作，甚至还可能持有一个信号量。

## 进程创建

许多其他操作系统都提供了产生（spawn）进程的机制：首先在新的地址空间里创建进程，读入可执行文件，最后开始执行。

Unix把上述步骤分解到两个单独的函数中去执行：fork()和exec()：首先fork（）通过拷贝当前进程创建一个子进程，仅仅PID、PPID（父进程的PID）和某些资源和统计量（如挂起的信号没有必要继承）。exec()负责负责可执行文件并将其载入地址空间开始运行。

### 写时拷贝

传统fork()直接把所有资源复制给新进程，简单且效率低，因为拷贝的数据不共享。更糟糕的是如果新进程打算执行新映像，所有拷贝都前功尽弃。

Linux的fork()使用写时拷贝页实现。写时拷贝是一种可以推迟甚至免除拷贝数据的技术：内核并不复制整个进程空间，而是共享一个拷贝。 **只有在写入的时候，数据才会复制** ，从而各个进程有各自的拷贝。在此之前，只是以只读方式共享，在页根本不会写入的情况下，它们就无需复制。因此fork()实际开销是复制父进程页表和创建子进程的进程描述符，这种优化可以 **避免拷贝大量根本不会被使用的数据** （地址空间常常包含数十兆的数据）。

### fork()

fork()、vfork()、\_clone()库函数都是根据自己需要的参数标志去调用clone()，然后clone()再去调用do\_fork()

fork()

clone()

do\_fork()

copy\_process()

1. dup\_task\_struct()：新内核栈、thread\_info、task\_struct，与当前进程值相同，此时父子进程描述符相同

2. 检查新建进程后当前用户进程数目没有超过给它分配的资源限制

3. 描述符许多成员被清0或设为初始值，主要是统计信息。描述符中大多数据仍为修改

4. 子进程状态设为TAK\_UNINTERTUPTIBLE，保证不会投入运行

5. copy\_flags()更新task\_struct的flags成员

6. alloc\_pid()：为新进程分配一个有效的PID

7. 根据传递给clone的参数标志，拷贝或共享打开文件、文件系统信息、信号处理函数、进程地址空间和命名空间等。

8. 做扫尾工作，返回一个指向子进程的指针

copy\_process()成功，新建子进程被唤醒并投入运行。

内核有意选择子进程首先执行，一般子进程会马上调用exec()函数，这样可避免写时拷贝的额外开销，如果父进程首先执行的话，有可能会像地址空间写入。

### vfork()

除不拷贝父进程的页表项，vfork()和fork()的功能相同，子进程作为父进程的单独线程在父进程地址空间运行，父进程阻塞，直到子进程退出或执行exec()，但是如果exec()调用失败父进程会被一直阻塞，所以理想情况下，系统最好不要调用vfork()，内核也不用去实现它，完全可以把vfork()实现成一个普普通通的fork()

## 线程在Linux中的实现

Linux从内核角度来说，并没有线程整个概念，把所有线程当作进程实现。内核并没有准备特别的调度算法或定义特别的数据结构来表征线程，线程仅仅被视为一个和其他进程共享某些资源的进程。每个线程都拥有唯一隶属于自己的task\_struct，所以在内核中看起来就像是一个普通的进程。

1. 在其他系统中，相较于重量级的进程，线程被抽象成一个耗费较少资源，运行迅速的指定单元，因此在这些系统中常常被称为“轻量级进程”。
2. 在Linux中，它只是一种进程间共享资源的手段（Linux进程本身就够轻量级）。 举个例子，有一个四线程的进程，在提供专门线程支持的系统中，通常会有 **一个包含** 指向四个不同线程的指针的进程描述符。该描述符负责描述共享资源，线程本身再去描述它独占的资源。在Linux中，仅仅创建四个进程并分配四个不同的task\_struct结构，建立着四个进程时指定他们共享某些资源。

### 创建线程

线程的创建和普通进程类似，但调用clone()时需要传递一些参数来指定需要共享的资源，例如下面的代码：

// 父子俩共享地址空间、文件系统资源、文件描述符和信号处理程序，换句话说，新建的进程和父进程就是流行的所谓线程

clone(CLONE\_VM | CLONE\_FS | CLONE\_FILES | CLONE\_SIGHAND, 0)

### 内核线程

内核经常需要在后台执行一些操作，可以通过内核线程（kernel thread）完成——独立运行在内核空间的标准线程。内核线程和普通进程间的区别在于内核线程没有独立的地址空间，只在内核空间运行，从不切换到用户空间。可以被调度也可以被抢占。  
内核线程只能由其他内核线程创建。内核是通过从kthreadd内核进程中衍生除所有新的内核线程来自动处理这一点。

1. 创建内核线程可以通过kthread\_create()创建，但新进程处于不可运行状态，如果不通过wake\_up\_process()主动唤醒，不会主动运行。
2. 新建并运行内核线程可以通过kthread\_run()函数达到，该函数以宏实现，简单调用kthread\_create()和wake\_up\_process()。

内核线程启动后一直运行到调用do\_exit()退出或内核的其他部分调用kthread\_stop()退出，参数为创建时返回的task\_struct结构体。

## 进程终结

1. 主动终结。进程显示或隐式调用exit()，可能从某个程序主函数返回（C语言编译器会在main()函数返回点后面放置exit()）
2. 被动终结。进程接收到它不能处理也不能忽略的信号或异常时，被动终结。

无论怎么终结，该任务大部分要靠do\_exit()完成。do\_exit()永不返回。调用后进程不可运行并处于EXIT\_ZOMBIE退出状态，占用的内存就是内核栈、thread\_info结构体和task\_struct结构体。此时进程存在的唯一目的就是向父进程提供信息，父进程检测到信息后，由进程持有的成语内存被释放，归还给系统。

### 删除进程描述符

wait()这一族函数都是通过唯一的一个系统wait4()来实现的。标准动作是：挂起调用进程，直到其中一个子进程退出，返回该子进程PID。

### 孤儿进程造成的进退维谷

父进程在子进程之前退出，必须有机制保证子进程找一个新父亲，否则将在退出时永远处于僵死状态，变成僵尸进程。对于该问题，解决方法是：

1. 给子进程在当前线程组内找一个线程做父亲
2. 如果1不行，就让init做它的父进程

do\_exit()->forget\_original\_parent()->find\_new\_reaper()来寻找父进程。init会例行调用wait()来检查其子进程，清除所有与其相关的僵死进程。