深入理解 Linux 内核 第三章进程

Contents

第三章 进程

进程是任何多道程序设计的**操作系统**中的基本概念。通常把进程定义为程序执行的一个实例。在 **Linux** 中，常把进程称为任务 task 或线程 thread。

进程、轻量级进程和线程

进程类似人类：它们被产生，有或多或少的生命，可以产生一个或多个子进程，最终都要死亡。一个微小的差异是进程之间没有性别差异——每个进程都只有一个父亲。

当一个进程创建时，它几乎与父进程相同。它接受父进程地址空间的一个逻辑拷贝，并从进程创建系统调用的下一条指令开始执行与父进程相同的代码。尽管父子进程可以共享含有程序代码的页，但是它们各自有独立的数据拷贝（栈和堆），因此子进程对一个内存单元的修改对父进程是不可见的，反之亦然。

现代 Unix 系统支持多线程应用程序，一个进程由几个用户线程组成，每个线程都代表进程的一个执行流。大部分多线程应用程序都是用 pthread（POSIX thread）库的标准库函数集编写的。

Linux 使用轻量级进程 lightweight process 多多线程应用程序提供更好的支持。两个轻量级进程基本上可以共享一些资源，诸如地址空间、打开的文件等等。

进程描述符

为了管理进程，内核必须对每个进程所做的事情进行清楚的描述。这正是进程描述符 process descriptor 的作用，进程描述符都是 task_struct 类型结构，它的字段包含了与一个进程相关的所有信息。

进程状态

进程描述符中的 state 字段描述了进程当前所处的状态。它由一组标志组成，其中每个标志描述一种可能的进程状态。下面是进程可能的状态：

- 可运行状态 TASK_RUNNING，进程要么在 CPU 上执行，要么准备执行。
- 可中断的等待状态 TASK_INTERRUPTIBLE，进程被挂起（睡眠），直到某个条件变为真。产生一个硬件中断，释放进程正等待的系统资源，或传递一个信号都是可以唤醒进程的条件（把进程的状态方回到 TASK_RUNNING）。
- 不可中断的等待状态 TASK_UNINTERRUPTIBLE，和可中断的等待状态类似，但是有一个例外，把信号传递到睡眠进程不能改变它的状态。
- 暂停状态 TASK_STOPPED，进程的执行被暂停，当进程接收到 SIGSTOP、SIGTSTP、SIGTTIN 或 SIGTTOU 信号后，进入暂停状态。
- 跟踪状态 TASK_TRACED，进程的执行已由 debugger 程序暂停。

还有两个状态即可以存放在进程描述符的 state 字段中，也可以存放在 exit_state 字段中。

- 僵死状态 EXIT_ZOMBIE，进程的执行被终止，但是父进程还没有发布 wait4()或 waitpid()系统调用来返回有关死亡进程的信息。这个时候内核不能丢弃该进程进程描述符中的数据，因为父进程可能还需要使用。
- 僵死撤销状态 EXIT_DEAD，最终状态：由于父进程发出了 wait4 或 waitpid 系统调用，因而进程由系统删除。

标识一个进程

进程和进程描述符之间有非常严格的一一对应关系，类 Unix 操作系统允许用户使用一个叫做进程标识符 process ID 或 PID 的数来标识进程，PID 存放在进程描述符的 pid 字段中。PID 被顺序编号，其值有上限，当达到这个上限之后就必须开始循环使用已闲置的小 PID 号。默认最大的 PID 号是 32767，可通过/proc/sys/kernel/pid_max 进行修改。

Linux 引入线程组的表示。一个线程组中的所有线程使用和该线程组的领头线程 thread group leader 相同的 PID，它被存入进程描述符的 tgid 字段中。线程组的领头线程其 tgid 值与 pid 值相同。

进程描述符处理

对于每个进程来说，Linux 都把两个不同的**数据结构**紧凑地存放在一个单独为进程分配的存储区域内：一个是内核的进程堆栈，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，这块存储区的大小通常为 8192 个字节。

esp 寄存器是 CPU 的栈指针，用来存放栈顶单元的地址。

标识当前进程

内核很容易从 esp 寄存器的值获得当前在 CPU 上正在运行进程的 thread_info 结构体的地址。调用 current 宏可获取进程描述符的地址。

双向链表

对每个链表，必须实现一组原语操作：初始化链表，插入和删除一个元素，扫描链表等等。为了方便，Linux 内核定义了 list_head 数据结构，字段 next 和 prev 分别表示通用双向链表向前向后的指针元素。

可使用 LIST_HEAD(list_name)来创建一个 list_name 的链表，其 prev 和 next 字段指向 list_name 变量本身。

几个常见的函数和宏：

- list_add(n,p)

把 n 指向的元素插入 p 所指向的特定元素之后。

- list_add_tail(n,p)

把 n 指向的元素插到 p 所指向的特定元素之前。

- list_del(p)

删除 p 所指向的元素。

- list_empty(p)

检查由第一个元素的地址 p 指定的链表是否为空。

- list_for_each(p,h)

对表头地址 h 指定的链表进行扫描，在每次循环时，通过 p 返回指向链表元素的 list_head 结构的指针。

- list_for_each_entry(p,h,m)

与 list_for_each 类似，但是返回包含了 list_head 结构的数据结构的地址，而不是 list_head 结构本身的地址。

Linux 内核还支持另一种双向链表，它不是循环链表，主要用于散列表。表头存放在 hlist_head 数据结构中。

进程链表

每个 task_struct 结构都包含一个 list_head 类型的 tasks 字段，这个类型的 prev 和 next 字段分别指向前面和后面的 task_struct 元素。

进程链表的头是 init_task 描述符，它是所谓 0 进程 process 0 或 swapper 进程的进程描述符。

TASK_RUNNING 状态的进程链表

为了提高调度程序运行速度，Linux2.6 之后版本建立了多个可运行进程链表，每种进程优先级对应一个不同的链表。每个 task_struct 描述符包含一个 list_head 类型的字段 run_list。如果进程的优先级为 k（取值范围 0-139），run_list 字段把该进程链入优先级为 k 的可运行进程的链表中。此外，在多处理器系统中，每个 CPU 都由它自己的运行队列，即它自己的进程链表集。

进程描述符的 prio 字符存放进程的动态优先权。
所有这些链表都由一个单独的 prio_array_t 数据结构来实现。

类型	字段	描述
int	nr_active	链表中进程描述符的数量
unsigned log[5]	bitmap	优先权位图，当且仅当每个优先权的进程链表不为空时设置响应的位标志
struct list_head[140]	queue	140 个优先权队列的头节点

进程间的关系

程序创建的进程具有父子关系，如果一个进程创建多个子进程，则子进程之间具有兄弟关系。
- real_parent 指向创建了 P 的进程的描述符，如果 P 的父进程不存在，就指向了进程 1（init）的描述符
- parent 指向 P 的当前父进程
- children 链表的头部，链表中的所有的元素都是 P 创建的子进程
- sibling 指向兄弟进程链表中的下一个元素或钱一个元素的指针，这些兄弟进程的父进程都是 P

进程之间还存在其他关系： 一个进程可能是一个进程组或登陆会话的领头进程，也可能是一个线程组的领头进程，还可能跟踪其他进程的执行。
- group_leader P 所在进程组的领头进程的描述符指针
- signal->pgrp P 所在进程组的领头进程的 PID
- tgid P 所在线程组的领头进程的 PID
- signal->session P 的登陆会话领头进程的 PID
- ptrace_children 链表的头，该链表包含所有被 debugger 程序跟踪的 P 的子进程
- ptrace_list 指向所跟踪进程其实际父进程链表的钱一个和下一个元素

pidhash 表及链表

顺序扫描进程链表并检查进程描述符的 pid 字段是可行但相当低效的。为了加快查找，引入了 4 个散列表。

Hash 表的类型	字段名	说明
PIDTYPE_PID	pid	进程的 PID
PIDTYPE_TGID	tgid	线程组领头进程的 PID
PIDTYPE_PGID	gprp	进程组领头进程的 PID
PIDTYPE_SID	session	会话领头进程的 PID

散列 hash 函数并不能总能确保 PID 与表索引一一对应，两个不同 PID 散列到相同的表索引称为冲突 colliding。

如何组织进程

运行队列链表把处于 TASK_RUNNING 状态的所有进程组织在一起。当要把其他状态的进程分组时，不同的状态要求不同的处理，Linux 选择了下列方式之一：

- 没有为处于 TASK_STOPPED、EXIT_ZOMBIE 或 EXIT_DEAD 状态的进程建立专门的链表。由于对处于暂停、僵死、死亡状态的进程的访问比较简单，或者通过 PID，或者通过特定父进程的子进程链表，所以不必对这三种状态进程分组。

等待队列

等待队列在内核中有很多用途，尤其用在终端处理、进程同步及定时。等待队列实现了在事件上的条件等待：希望等待特定事件的进程把自己放进合适的等待队列，并放弃控制权。因此，等待队列表示一组睡眠的进程，当某一条件变为真时，由内核唤醒它们。

等待队列由双向链表实现，其元素包括指向进程描述符的指针。每个等待队列都有一个等待队列头 wait queue head，等待队列头是一个类型为 wait_queue_head_t 的数据结构。

```
struct __wait_queue_head {
        spinlock_t lock;
        struct list_head task_list;
};
typedef struct __wait_queue_head wait_queue_head_t;
```


同步是通过等待队列头中 lock 自旋锁达到的。task_list 字段是等待进程链表的头。

等待队列链表中的元素类型为 wait_queue_t

```
struct __wait_queue {
        unsigned int flags;
#define WQ_FLAG_EXCLUSIVE       0x01
        void *private;
        wait_queue_func_t func;
        struct list_head task_list;
};
typedef struct __wait_queue wait_queue_t;
```

等待队列中每个元素代表一个睡眠进程，该进程等待某一事件的发生；它的描述符地址存放在 private 字段中。task_list 字段中包含的是指针，由这个指针把一个元素连接到等待相同时间的进程链表中。

有两种睡眠进程：互斥进程（等待队列元素的 flags 字段为 1）由内核有选择地唤醒，而非互斥进程（flags 为 0）总是有内核在时间发生时唤醒。等待访问临界资源的进程就是互斥进程的典型例子。

等待队列的操作

略，见《wait queue 简介.md》

进程的资源限制

每个进程都由一组相关的资源限制，限制指定了进程能使用的系统资源数量。对当前进程的资源限制存放在 current->signal->rlim 字段，即进程的信号描述符的一个字段。

进程切换

为了控制进程的执行，内核必须有能力挂起正在 CPU 上运行的进程，并回复以前挂起的某个进程的执行。这种行为被称为进程切换 process switch、任务切换 task switch 或上下文切换 context switch。

硬件上下文

尽管每个进程可以拥有属于自己的地址空间，但所有进程必须共享 CPU 寄存器。因此，在回复一个进程的执行之前，内核必须确保每个寄存器装入了挂起进程时的值。这组寄存器数据称为硬件上下文 hardware context。在 Linux 中，进程硬件上下文一部分存放在 TSS 段，而剩余部分存放在内核堆栈中。

由于进程切换经常发生，减少保存和装入硬件上下文所花的时间是非常重要的。Linux2.6 使用软件执行进程切换：通过一组 mov 指令逐步执行切换，这样能较好地控制所能装入数据的合法性。尤其是，这使检查 ds 和 es 段寄存器的值称为可能，这些值有可能被恶意用户伪造。

任务状态段

尽管 Linux 不使用硬件上下文切换，但是强制它为系统每个不同的 CPU 创建一个 TSS task state segment，这样做的两个主要理由为：

- 当一个 CPU 从用户态切换到内核态时，从 TSS 中获取内核态堆栈的地址。
- 当用户态进程试图通过 in 或 out 指令访问一个 I/O 端口时，CPU 需要访问存放在 TSS 中的 I/O 许可权位图以检查进程是否有访问端口的权力。

每个 TSS 都由它自己 8 字节的任务状态段描述符 Task State Segment Descriptor，TSSD。这个描述符包括指向 TSS 起始地址的 32 为 Base 字段，20 位 Limit 字段。

由 Linux 创建的 TSSD 存放在全局描述符表 GDT 中，GDT 的基地址存放在每个 CPU 的 gdtr 寄存器中。每个 CPU 的 tr 寄存器包含相应 TSS 的 TSSD 选择符，也包含了两个隐藏的非编程字段：TSSD 的 Base 字段和 Limit 字段。这样，处理器就能直接对 TSS 寻址问不用从 GDT 中检索 TSS 的地址。

thread 字段

每个进程描述符包含一个类型为 thread_struct 的 thread 字段，只要进程被切换出去，内核就把其硬件上下文保存在这个结构中。

执行进程切换

进程切换可能只发生在精心定义的点：schedule 函数。从本质上来说，每个进程切换由两步组成：

- 切换页全局目录以安装一个新的地址空间。
- 切换内核态堆栈和硬件上下文。

switch_to 宏

进程切换的第二步有 switch_to 宏执行。该宏有三个参数：prev，next 和 last。prev 和 next 分表表示被替换进程和新进程描述符的地址。最后一个参数 last 是输出参数，表示宏把进程 C 的描述符写在内存的什么位置了。这个地方有点复杂，待分析。

创建进程

传统的 Unix 操作系统以统一的方式对待所有的进程：子进程复制父进程所拥有的资源。该方法使进程的创建非常慢且效率低。现代 Unix 内核通过引入三种不同的机制解决了这个问题：

- 写时复制技术允许父子进程读相同的物理页。只要两者中有一个试图写一个物理页，内核就把这个页的内容拷贝到一个新的物理页，并把这个新的物理页分配给正在写的进程。
- 轻量级进程允许父子进程共享每进程在内核的很多数据结构，如页表、打开文件表及信号处理。
- vfork 系统调用创建的进程能共享其父进程的内存地址空间。为了防止父进程重写子进程需要的数据，阻塞父进程的执行，一直到子进程退出或执行一个新的程序为止。

clone、fork、vfork 系统调用

Linux 中，轻量级进程由 clone 函数创建。该函数有如下参数：

- fn

指定一个由新进程执行的函数。当这个函数返回时，子进程终止。函数返回一个整数，表示子进程的退出代码。

- arg

指向传递给 fn 函数的数据。

- flags

各种各样的信息。低字节指定子进程结束时发送到父进程的信号代码，通常选择 SIGCHLD 信号。剩余的 3 个字节给 clone 标志组用于编码，如下表所示。

- child_stack

表示把用户态堆栈指针赋给子进程的 esp 寄存器。调用进程应该总是为子进程分配新的堆栈。

- tls

表示线程局部存储段 TLS 数据结构的地址，该结构是为新轻量级进程定义的。只有在 CLONE_SETTLS 标志被设置时才有意义。

- ptid

表示父进程的用户态变量地址，该父进程具有与新轻量级进程相同的 PID。只有在 CLONE_PARENT_SETTID 标志被设置时才有意义。

- ctid

表示新轻量级进程的用户态变量地址，该进程具有这一类进程的 PID。只有在 CLONE_CHILD_SETTID 标志被设置时才有意义。

标志名称	说明
CLONE_VM	共享内存描述符和所有的页表
CLONE_FS	共享根目录和当前工作目录所在的表，以及用于屏蔽新文件初始许可权的位掩码值

标志名称	说明
CLONE_FILES	共享打开文件表
CLONE_SIGHAND	共享信号处理程序的表、阻塞信号表和挂起信号表
CLONE_PTRACE	如果父进程被跟踪，子进程也被跟踪
CLONE_VFORK	在发出 vfork 系统调用时设置
CLONE_PARENT	设置子进程的父进程为调用进程的父进程
CLONE_THREAD	把子进程插入到父进程的统一线程组中，并迫使子进程共享父进程的信号描述符。也设置子进程的 tgid 字段和 group_leader 字段
CLONE_NEWNS	当 clone 需要自己的命名空间时设置这个标志。
CLONE_SYSVSEM	共享 System V IPC 取消信号量的操作
CLONE_SETTLS	为轻量级进程创建新的线程局部存储段 TLS
CLONE_PARENT_SETTID	把子进程的 PID 写入有 ptid 参数所指向的父进程的用户态变量
CLONE_CHILD_SLEARTID	内核建立一种触发机制，用在子进程要退出或要开始执行新程序时。内核将清除有参数 tgid 所指向的用户态变量，并唤醒等待这个事件的任何进程
CLONE_DETACHED	遗留标志，内核会忽略
CLONE_UNTRACED	内核设置这个标志使 CLONE_PTRACE 标志失去作用
CLONE_CHILD_SETTID	把子进程的 PID 写入有 ctid 参数所指向的子进程的用户态变量中
CLONE_STOPPED	强迫子进程开始于 TASK_STOPPED 状态

实际上 clone 是在 **C 语言**库中定义的一个封装函数，它负责建立新轻量级进程的堆栈并且调用对编程者隐藏的 clone 系统调用。而 clone 系统调用的 sys_clone 服务例程并没有 fn 和 arg 参数。封装函数将 fn 和 arg 存放在封装函数本身返回地址处，当封装函数结束时，CPU 从堆栈中取出返回地址，然后执行 fn(arg)函数。

fork 系统调用在 Linux 中是用 clone 实现的，其 flags 参数指定为 SIGCHLD 信号及所有请 0 的 clone 标志，而它的 child_stack 参数是父进程当前的堆栈指针。

vfork 系统调用也是用 clone 实现的，其 flags 指定为 SIGCHLD 信号和 CLONE_VM 及 CLONE_VFORK 标志，child_stack 参数是父进程当前的堆栈指针。

do_fork 函数

do_fork 函数负责处理 clone、fork 和 vfork 系统调用，执行时使用下列参数：

- clone_flags 与 clone 的参数 flags 相同
- stack_start 与 clone 的参数 child_stack 相同
- regs 指向通用寄存器值的指针，通用寄存器的值是在从用户态切换到内核态时被保存到内核态堆栈中的
- stact_size 未使用，0
- parent_tidprt,child_tidptr 与 clone 中的对应参数 ptid 和 ctid 相同

do_fork 利用辅助函数 copy_process 来创建进程描述符以及子进程执行所需要的所有其他内核数据结构。下面是 do_fork 执行的主要步骤：

- 通过查找 pidmap_array 位图，为子进程分配新的 PID。
- 检查父进程的 ptrace 字段 current->ptrace：如果它的值不等于 0，说明有另外一个进程正在跟踪父进程，因而，do_fork 检查 debugger 程序是否自己向跟踪子进程。
- 调用 copy_process 复制进程描述符。如果所有必须的资源都是可用的，该函数返回刚创建的 task_struct 描述符的地址。
- 如果设置了 CLONE_STOPPED 标志，或者必须跟踪子进程，即在 p->ptrace 中设置了 PT_PTRACED 标志，那么子进程的状态被设置称 TASK_STOPPED，并为子进程增加挂起 SIGSTOP 信号，直到另外一个进程把子进程的状态恢复为 TASK_RUNNING。
- 如果没有设置 CLONE_STOPPED 标志，则调用 wake_up_new_task 函数以执行下述操作：
- 调整父进程和子进程的调度参数
- 如果子进程将和父进程运行在同一个 CPU 上，而且父进程和子进程不能共享同一组页表，那么，就把子进程插入父进程运行队列，插入时让子进程恰好在父进程前面，因此而迫使子进程先于父进程运行。
- 否则，如果子进程与父进程运行在不同的 CPU 上，或者父进程和子进程共享同一组页表，就把子进程插入父进程运行队列的队尾。
- 如果 CLONE_STOPPED 标志被设置，则把子进程置为 TASK_STOPPED 状态。
- 如果父进程被跟踪，则把子进程的 PID 存入 current 的 ptrace_message 字段并调用 ptrace_notify，该函数使当前进程停止运行，并向当前进程的父进程发送 SIGCHLD 信号。子进程的祖父进程是跟踪父进程的 debugger 进程。SIGCHLD 信号通知 debugger 进程：current 已经创建了一个子进程，可以通过查找 current->ptrace_message 字段获得子进程的 PID。
- 如果设置了 CLONE_VFORK 标志，则把父进程插入等待队列，并挂起父进程直到子进程释放自己的内存地址空间。
- 结束并返回子进程的 PID。

copy_process 函数

该函数创建进程描述符以及子进程执行所需要的所有其他数据结构。它的参数与 do_fork 的参数相同，外加子进程的 PID。其主要步骤如下：

- 检查产生 clone_flags 的合法性。
- 调用 security_task_create 以及稍后调用的 security_task_alloc 执行所有附加的安全检查。
- 调用 dup_task_struct 为子进程获取进程描述符。
- 检查存放在 current->signal->rlim[RLIMIT_NPROC].rlim_cur 变量中的值是否小于或等于用户所拥有的进程数。如果是，返回错误码。
- 递增 user_struct 结构的使用计数器和用户所拥有的进程的计数器。
- 检查系统中的进程数量是否超过 max_threads 变量的值。该变量的缺省值取决于系统内存容量的大小，总的原则是，所有 thread_info 描述符和内核栈所占用的空间不能超过物理内存大小的 1/8。可以通过/proc/sys/kernel/thread-max 修改这个值。
- 如果实现新进程的执行域和可执行格式的内核函数都包含在内核模块中，则递增他们的使用计数器。
- 设置与进程状态相关的几个关键字段
- ……

内核线程

在 Linux 中，内核线程在一下几个方面不同于普通进程：

- 内核线程只运行在内核态，而普通进程既可以运行在内核态，也可以运行在用户态。
- 因为内核线程只运行在内核态，它们只使用大于 PAGE_OFFSET 的线性地址空间。不管在用户态还是在内核态，普通进程可以用 4GB 的线性地址空间。

创建一个内核线程

kernel_thread 函数穿件一个新的内核线程，它接收的参数有：所要执行的内核函数的地址 fn，要传递给函数的参数 arg，一组 clone 标准 flags。该函数会调用 do_fork 函数：

```
    do_fork(falgs|CLONE_VM|CLONE_UNTRACED, 0, pregs, 0, NULL, NULL);
```

进程 0

所有进程的祖先叫做进程 0，idle 进程或因为历史原因叫做 swapper 进程，它是在 Linux 初始化阶段从无到有攒国家的一个内核线程。这个祖先进程使用下列静态分配的数据结构（其他进程的数据结构都是动态分配的）：

- 存放在 init_task 变量中的进程描述符，有 INIT_TASK 宏完成对它的初始化。
- 存放在 init_thread_union 变量中的 thread_info 描述符和内核堆栈，由 INIT_THREAD_INFO 宏完成对它们的初始化。
- 由进程描述符指向的下列表：
- init_mm 由 INIT_MM 宏初始化
- init_fs 由 INIT_FS 宏初始化
- init_files 由 INIT_FILES 宏初始化
- init_signals 由 INIT_SIGNALS 宏初始化
- init_sighand 由 INIT_SIGHAND 宏初始化
- 主内核页全局目录存放在 swapper_pg_dir 中。

start_kernel 函数初始化内核需要的所有数据结构，激活中断，创建另一个叫进程 1 的内核线程（init）：

```
kernel_thread(init, NULL, CLONE_FS|CLONE_SIGHAND);
```

新创建内核线程的 PID 为 1，并与进程 0 共享每进程所有的内核数据结构。此外，当调度程序选择到它时，init 进程开始执行 init 函数。在系统关闭之前，init 进程一直存活，因为它创建和监控在操作系统外层执行的所有进程的活动。

创建 init 进程后，进程 0 执行 cpu_idle 函数，该函数本质上是在开中断的情况下重复执行 hlt 汇编语言指令。只有当没有其他进程处于 TASK_RUNNING 状态时，调度程序才选择进程 0。

其他内核线程

Linux 使用很多其他内核线程。有一些在初始化阶段创建，一直运行到系统关闭，而其他一些在内核必须执行一个任务时按需创建，这种任务在内核的执行上下文中得到很好的执行。

一些内核线程的例子：

- keventd 执行 keventd_wa 工作队列中的函数。
- kapmd 处理与高级电压管理相关的事件。
- kswapd 执行内存回收。
- pdflush 刷新脏缓冲区中的内容到磁盘以回收内存。
- kblockd 执行 kblockd_workqueue 工作队列中的函数。实质上，它周期性地激活块设备驱动程序。
- ksoftirqd 运行 tasklet，系统中每个 CPU 都由这样一个内核线程。

撤销进程

进程终止的一般方式是调用 exit 库函数，该函数释放 C 函数库所分配的资源，执行编程者所注册的每个函数，并结束从系统回收进程的那个系统调用。C 编译程序总是把 exit 函数插入到 main 函数的最后一条语句之后。

进程终止

两个终止用户态应用的系统调用：

- exit_group 系统调用，它终止整个线程组，即整个基于多线程的应用。do_group_exit 是实现这个系统调用的主要内核函数。这是 C 库函数 exit 应该调用的系统调用。

- exit 系统调用，它终止某一个线程，而不管该线程所属线程组中的所有其他进程。do_exit 是实现这个系统调用的主要内核函数。这是被诸如 pthread_exit 的 Linux 线程库的函数所调用的系统调用。

do_group_exit 函数

杀死属于 current 线程组的所有进程。它接受进程终止代号作为参数，进程终止代号可能是系统调用 exit_group 指定的一个值，也可能是内核提供的一个错误代号。

do_exit 函数

所有进程的终止都是由 do_exit 函数来处理的，这个函数从内核数据结构中删除对终止进程的大部分引用。

进程删除

release_task 函数从僵死进程的描述符中分离出最后的数据结构，对僵死进程的处理有两种可能的方式：如果父进程不需要接收来自子进程的信号，就调用 do_exit；如果已经给父进程发送了一个信号，就调用 wait 等系统调用。在后一种情况下，函数还将回收进程描述符所占用的内存空间，在前一种情况下，内存的回收将由进程调度程序来完成。