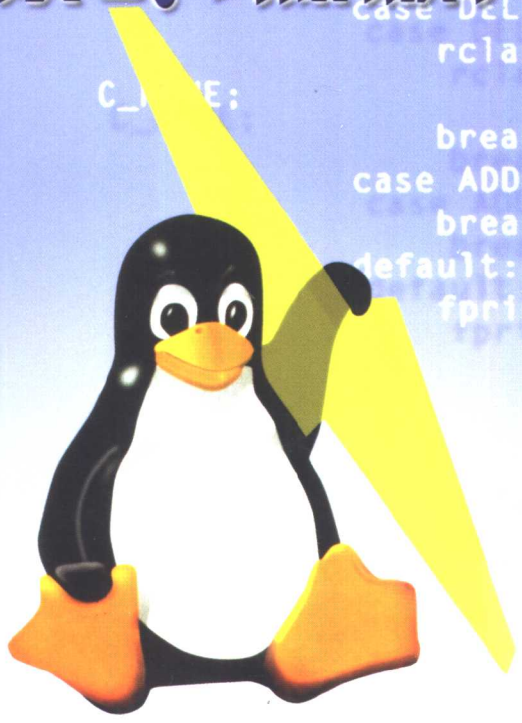
HZ BOOKS

Linux与自由软件资源丛书

Linux

内核2.4版源代码分析大全



李善平 刘文峰 李程远
王焕龙 王伟波 编著

机械工业出版社
China Machine Press

Linux与自由软件资源丛书

Linux内核2.4版源代码分析大全

李善平　刘文峰　李程远　王焕龙　王伟波　编著

机械工业出版社
China Machine Press

本书对 Linux 2.4 内核源代码进行了详细的介绍，它有效地解决了 Linux 操作系统长期以来“抽象”的现象，将 Linux 操作系统的概念、算法、原理等与实际操作统一起来。书中源代码与注释相对应，是大专院校师生与 Linux 操作系统管理人员及编程人员不可缺少的参考书。

图书在版编目(CIP)数据

Linux 内核 2.4 版源代码分析大全/李善平等编著. - 北京：机械工业出版社，2002.1
(Linux 与自由软件资源丛书)
ISBN 7-111-09344-5

Ⅰ.L... Ⅱ.李 ... Ⅲ.Linux 操作系统—程序分析 Ⅳ.TP316.89

中国版本图书馆 CIP 数据核字(2001)第 066138 号

机械工业出版社(北京市西城区百万庄大街 22 号 邮政编码 100037)
责任编辑：张金梅
北京第二外国语学院印刷厂印刷·新华书店北京发行所发行
2002 年 1 月第 1 版第 1 次印刷
787mm×1092mm 1/16 · 53.5 印张
印数：0 001 - 4000 册
定价：79.00 元

前　言

笔者自 1994 年起一直承担计算机系“操作系统”课程的教学，感觉学习该课程有两大难处。其一是抽象。很难将书本中学到的操作系统概念、算法、原理等，与实际的操作系统加以印证。操作系统本来是所有软件中使用最广泛的，每台计算机必须安装操作系统，有的甚至不止一套。但它们只不过是一只只“黑匣子”，因为读不到，或读不懂操作系统的源代码。绝大多数用户只了解它们的外部功能和性能，却不知道如何实现这样的功能和达到这样的性能。其二是脱离实际。计算机专业课非常强调上机实习，而学习“操作系统”却难以实习。一方面因为操作系统是所有软件中最复杂的，编制这样的系统牵涉到方方面面；编程者既要有扎实的软件基础，又要了解将操作的硬件，难度可想而知。另一方面，由于种种原因，并不是大多数软件工程师都拥有参与编制实际操作系统的经历或机会。

因此，为了教好、学好“操作系统”和“操作系统实验”课程，实验平台的选择就变得至关重要。通过检索国内外、网内外资料，并在对 UNIX、Microsoft Windows、MINIX、XINU、Naches 等知名系统进行比较的基础上，我们选择了 Linux。因为 Linux 兼具如下三大优点：

- Linux 不同于 Windows NT、UNIX 等，它的源代码是公开的。只要你有兴趣，可以仔细阅读、分析、探究它是怎样实现现代操作系统常规功能且达到如此优良性能的，还可以与其他有关操作系统原理书中介绍的方法逐一比较。
- Linux 不同于 MINIX，它实现了虚拟存储管理，当然也支持进程、多处理器、多文件系统等现代操作系统的特征。
- Linux 不同于 Naches 等实验型操作系统，它是一个具有极强生命力的实用操作系统。成千上万种应用软件都可以在 Linux 平台上运行。Linux 版本升级非常快，编写此教程时，最新版本号是 2.4.0，然而截止 2001 年 4 月 28 日，已升至 2.4.4 版本。

正是在“操作系统”课程教学的过程中，使我们逐步深入了解了 Linux。正是在学习过程中，令我们以及浙江大学计算机系绝大多数上过“操作系统”、“高级操作系统”课的研究生和大学生（包括第二学位的计算机专业本科生）对 Linux 爱不释手。可以说，编写此教程，也是希望读者分享我们的乐趣。

在本书之前，承蒙机械工业出版社华章公司的支持，我曾与同行合作编写过《Linux 操作系统及实验教程》。该教程分三大部分，分别是关于 Linux 的使用及维护，Linux 内核 2.0.35 版本源代码的系统分析，Linux 的实验。限于时间，Linux 源码中绝大多数函数的注释，以及有些重要模块还来不及编入，实验内容也比较简单。而另一方面，Linux 的常识性内容与目前出版的大多数参考书重复。因此，编写本书时，我们放弃了第一、第三部分，突出 Linux 内核 2.4 版本的源代码分析。而且，这里只讨论 Linux 系统的单个 Intel CPU 版本（其实 Linux 本身支持多种 CPU 体系结构，且支持 SMP）。顺便提一下，如果你同时读 Linux 内核 2.0.35 版和 2.4.0 版，你肯定发现，这两个版本之间，差异已经非常明显。为方便读者在阅读本书过程中分析 Linux 源

代码，本书随时给出所讨论主题在源代码中的位置。给出的路径均相对于/usr/src/linux。例如，进程的 PCB 由文件 include/linux/sched.h 定义，也就是说，该文件的路径是/usr/src/linux/include/linux/sched.h。

本书主要内容包括 Linux 进程管理、Linux 存储管理、Linux 文件系统管理、Linux 设备管理、Linux 系统初始化、Linux 网络实现、Linux 的模块、Linux 内核源码的调试等内容，由我和我的学生刘文峰、李程远、王焕龙、王伟波合作完成。其中，李程远、李善平编写第 1 章、第 3 章、第 5 章，王焕龙、李善平编写第 2 章，王伟波、李善平编写第 4 章，刘文峰编写第 6 章、第 7 章、第 8 章。李善平审校了全书。由于作者水平有限，书中难免存在各种问题，敬请读者指正。笔者的邮箱地址为：Lishanping 2001@yahoo.com.cn。

还有一点有必要强调一下，本书内容在浙江大学计算机系 96 级、97 级、98 级、99 级、2000 级研究生，以及 96 级、97 级、98 级本科生累计约 2200 位同学的使用过程中得到了不断的补充和完善。本书能在今天与大家见面，也有他们的贡献。当然，本书中一切编写或理解方面的错误与他们无关。

李善平

写于浙江大学 104 年校庆之际

2001 年 3 月 27 日

目　录

第 1 章　Linux 进程管理

Linux 作为一个多用户操作系统，支持多道程序设计、分时处理和“软”实时处理，也带有某些微内核的特征(如 module 机制)。从技术上讲，Linux 在不久的将来支持线程也不成问题。为实现上述目标，毫无疑问须引入进程的概念。

INTEL 版本的 Linux 利用 INTEL 386 体系结构的保护模式、特权级等特征，把每个进程分为内核态(特权级 0)和用户态(特权级 3)两种级别。中断和系统调用是内核态向用户态提供服务的重要途径。

Linux 的进程控制块是 Linux 最复杂的数据结构之一，内容包括进程调度、信号处理、进程队列指针、进程标识和用户标识、定时控制、信号量处理、上下文切换、文件系统管理、内存管理、进程状态和进程标志等等信息，累计约 80 多种属性。Linux 的进程控制块常驻内存，它集中了所有关于进程个体特性的描述信息，是 Linux 实现任何系统功能的基础。因此，进程控制块将作为本章首先讨论的数据结构。

进程也是一个动态的概念，有它自己的生命周期。鉴于计算机系统的数字化特征，进程的生命周期是离散描述的。描述进程各主要阶段的就是进程状态。Linux 的进程状态分“执行”(包括就绪状态)、“可打断睡眠”、“不可打断睡眠”、“僵死”、“暂停”、“交换”等 6 种，它们的转换关系参见 1.2 节的图 1-4 及相应说明。因为进程从一个状态向另一个状态过渡的过程需要运行大段可中断、可重新调度的程序，处于过渡过程中的进程，其状态不明确，所以 Linux 另外定义了 11 种进程标志，同样可参见 1.2 节的内容。

Linux 进程管理由进程控制块、进程调度、中断处理、任务队列、定时器、bottom half 队列、系统调用、进程通信等部分组成，它是 Linux 存储管理、文件管理、设备管理的基础。

进程调度分实时进程调度和非实时进程调度两种。前者调度时，可以采用基于动态优先级的轮转法，也可以采用先进先出算法。后者调度时，一律采用基于动态优先级的轮转法。一个进程采用何种调度策略，由该进程控制块的 policy、priority、rt _priority、counter 属性规定。注意，没有一个专门的系统进行处理进程调度等事宜。Linux 的进程调度操作由 schedule 函数执行。这是一个只在内核态运行的函数，函数代码为所有进程所共享。任何进程，当它从系统调用返回时，都会转入 schedule()。或者，当中断产生时，大多数中断服务程序(包括 Linux 基准时间的中断服务程序)在完成它们的中断响应后，也会转入 schedule()。关于进程调度程序的分析，参见 1.3 节。

中断是现代计算机系统普遍采用和支持的概念。Linux 能够处理所有常用的中断源。因为中断响应的本质是“窃取”一段 CPU 时间，必然与进程有密切的联系，所以把与中断相关的主题都归入本章讨论，参见 1.4 节。为了及时捕获中断信号，避免过多地占用进程的 CPU 时间，中断响应往往要求在短期内完成。但有些中断服务客观上必须占用相对较多的 CPU 时间。对此，Linux 设计的 bottom half 队列和各种任务队列很好地解决了这对矛盾。作为一个实

用的例子，1.4 节还详细分析了 Linux 时钟中断和各类定时器的响应机制。

系统调用是操作系统为应用程序以及系统的外围支持软件提供的传统的界面。第 5 节集中给出了 Linux 关于系统调用的数据结构和函数，系统调用命令的转换，int 0x80 软中断，Linux 系统调用功能初始化，系统调用响应流程等方面的实现内幕。请特别关注系统调用返回前执行的 ret_from_sys_call 程序段。

在本章的剩余部分还特别对 Linux 突出的功能模块进行了分析。例如，1.5 节关于等待队列，1.14 节关于进程的创建和终止，1.15 节关于程序的装入和执行。读者可以根据兴趣选择。

1.1 进程和进程控制块

对于进程，Linux 沿用了操作系统教科书对进程概念的解释，没什么新的涵义。

由于进程是操作系统的最小调度单位。进程控制块（PCB）需表达方方面面的信息，Linux 的进程控制块（称作 task_struct，在 include/linux/sched.h 中定义）是系统中最重要的数据结构之一，2.4.0 版本中，每个 task_struct 结构占 1680 字节。在每个进程创建的时候申请一个 task_struct 结构空间。系统中的最大进程数由系统的物理内存大小决定（这一点与以前的内核版本有很大的不同）。大体数目的计算方法定义在函数 fork_init()中：

kernel/fork.c

```
66 void __init fork_init(unsigned long mempages)
67 {
68         /*
69          * The default maximum number of threads is set to a safe
70          * value: the thread structures can take up at most half
71          * of memory.
72          */
73         max_threads = mempages / (THREAD_SIZE/PAGE_SIZE) / 2;
74
75         init_task.rlim[RLIMIT_NPROC].rlim_cur = max_threads/2;
76         init_task.rlim[RLIMIT_NPROC].rlim_max = max_threads/2;
77 }
```

kernel/fork.c

task_struct（include/linux/sched.h）的定义及各主要部分的解释如下：

include/linux/sched.h

```
277 struct task_struct {
278     /*
279      * offsets of these are hardcoded elsewhere - touch with care
280      */
281         volatile long state;            /* -1 unrunnable, 0 runnable, >0 stopped */
282         unsigned long flags;            /* per process flags, defined below */
283         int sigpending;
284         mm_segment_t addr_limit;        /* thread address space:
285                                            0-0xBFFFFFFF for user-thead
286                                            0-0xFFFFFFFF for kernel-thread
287                                         */
288     struct exec_domain *exec_domain;
289     volatile long need_resched;
290     unsigned long ptrace;
```

```
291
292         int lock_depth;                 /* Lock depth */
293
294 /*
295  * offset 32 begins here on 32-bit platforms. We keep
296  * all fields in a single cacheline that are needed for
297  * the goodness() loop in schedule().
298  */
299         long counter;
300         long nice;
301         unsigned long policy;
302         struct mm_struct *mm;
303         int has_cpu, processor;
304         unsigned long cpus_allowed;
305         /*
306          * (only the 'next' pointer fits into the cacheline, but
307          * that's just fine.)
308          */
309         struct list_head run_list;
310         unsigned long sleep_time;
311
312         struct task_struct *next_task, *prev_task;
313         struct mm_struct *active_mm;
314
315 /* task state */
316         struct linux_binfmt *binfmt;
317         int exit_code, exit_signal;
318         int pdeath_signal;  /* The signal sent when the parent dies */
319         /* ??? */
320         unsigned long personality;
321         int dumpable:1;
322         int did_exec:1;
323         pid_t pid;
324         pid_t pgrp;
325         pid_t tty_old_pgrp;
326         pid_t session;
327         pid_t tgid;
328         /* boolean value for session group leader */
329         int leader;
330         /*
331          * pointers to (original) parent process, youngest child, younger sibling,
332          * older sibling, respectively. (p->father can be replaced with
333          * p->p_pptr->pid)
334          */
335         struct task_struct *p_opptr, *p_pptr, *p_cptr, *p_ysptr, *p_osptr;
336         struct list_head thread_group;
337
338         /* PID hash table linkage. */
339         struct task_struct *pidhash_next;
340         struct task_struct **pidhash_pprev;
341
342         wait_queue_head_t wait_chldexit;  /* for wait4() */
343         struct semaphore *vfork_sem;  /* for vfork() */
344         unsigned long rt_priority;
345         unsigned long it_real_value, it_prof_value, it_virt_value;
```

```
346         unsigned long it_real_incr, it_prof_incr, it_virt_incr;
347         struct timer_list real_timer;
348         struct tms times;
349         unsigned long start_time;
350         long per_cpu_utime[NR_CPUS], per_cpu_stime[NR_CPUS];
351 /* mm fault and swap info: this can arguably be seen as either mm-specific or thread-specific */
352         unsigned long min_flt, maj_flt, nswap, cmin_flt, cmaj_flt, cnswap;
353         int swappable:1;
354 /* process credentials */
355         uid_t uid,euid,suid,fsuid;
356         gid_t gid,egid,sgid,fsgid;
357         int ngroups;
358         gid_t groups[NGROUPS];
359         kernel_cap_t cap_effective, cap_inheritable, cap_permitted;
360         int keep_capabilities:1;
361         struct user_struct * user;
362 /* limits */
363         struct rlimit rlim[RLIM_NLIMITS];
364         unsigned short used_math;
365         char comm[16];
366 /* file system info */
367         int link_count;
368         struct tty_struct * tty; /* NULL if no tty */
369         unsigned int locks; /* How many file locks are being held */
370 /* ipc stuff */
371         struct sem_undo * semundo;
372         struct sem_queue * semsleeping;
373 /* CPU-specific state of this task */
374         struct thread_struct thread;
375 /* filesystem information */
376         struct fs_struct * fs;
377 /* open file information */
378         struct files_struct * files;
379 /* signal handlers */
380         spinlock_t sigmask_lock; /* Protects signal and blocked */
381         struct signal_struct * sig;
382
383         sigset_t blocked;
384         struct sigpending pending;
385
386         unsigned long sas_ss_sp;
387         size_t sas_ss_size;
388         int (* notifier)(void * priv);
389         void * notifier_data;
390         sigset_t * notifier_mask;
391
392 /* Thread group tracking */
393         u32 parent_exec_id;
394         u32 self_exec_id;
395 /* Protection of (de-)allocation: mm, files, fs, tty */
396         spinlock_t alloc_lock;
397 };
```

include/linux/sched.h

1.1.1　调度数据成员

1. volatile long state;

表示进程的当前状态：

- TASK_RUNNING。正在运行或在就绪队列 run-queue 中准备运行的进程，实际参与进程调度。
- TASK_INTERRUPTIBLE。处于等待队列中的进程，待资源有效时唤醒，也可由其他进程通过信号(signal)或定时中断唤醒后进入就绪队列 run-queue。
- TASK_UNINTERRUPTIBLE。处于等待队列中的进程，待资源有效时唤醒，不可由其他进程通过信号和定时中断唤醒。
- TASK_ZOMBIE。表示进程结束但尚未消亡的一种状态(僵死状态)。此时，进程已经结束运行且释放大部分资源，但尚未释放进程控制块。
- TASK_STOPPED。进程被暂停，通过其他进程的信号才能唤醒。导致这种状态的原因有二，或者是对收到 SIGSTOP、SIGSTP、SIGTTIN 或 SIGTTOU 信号的反应，或者是受其他进程的 ptrace 系统调用的控制而暂时将 CPU 交给控制进程。

2. unsigned long flags;

进程标志：

- PF_ALIGNWARN　正在打印“对齐”警告信息
- PF_STARTING　正在创建进程
- PF_EXITING　进程正在退出
- PF_FORKNOEXEC　进程刚创建，但还没执行
- PF_SUPERPRIV　使用超级用户特权
- PF_DUMPCORE　dumped core
- PF_SIGNALED　进程被信号(Signal)终止
- PF_MEMALLOC　正在分配内存
- PF_VFORK　对于用 vfork 创建的进程，退出前正在唤醒父进程
- PF_USEDFPU　该进程使用 FPU(SMP only)

3. unsigned long ptrace;

- PF_DTRACE　延迟跟踪(存储器为 68kB)
- PF_TRACESYS　正在跟踪
- PF_PTRACED　被 ptrace 系统调用监控

4. long priority;

进程优先级，priority 的值给出进程每次获取 CPU 后，可使用的时间(按 jiffy 计)。优先级可通过系统调用 sys_setpriority()改变(kernel/sys.c)。

5. unsigned long rt_priority;

rt_priority 给出实时进程的优先级，rt_priority + 1000 给出进程每次获取 CPU 后，可使用的时间(同样按 jiffy 计)。实时进程的优先级可通过系统调用 sys_sched_setscheduler()改变，不过

实际的工作是由 setscheduler()完成的,这两个函数均定义在 kernel/sched.c 中。

6.long counter;

在轮转法(round robin)调度时表示进程当前还可运行多久。在进程开始运行时被赋为 priority 的值,以后每隔一个 tick(时钟中断)递减 1,减到 0 时引起新一轮调度。重新调度将从 runqueue 队列选出 counter 值最大的就绪进程获得 CPU,因此 counter 起到了进程的动态优先级的作用(priority 则是静态优先级)。

7.unsigned long policy;

该进程的进程调度策略,可以通过系统调用 sys_sched_setscheduler()更改(kernel/sched.c)。调度策略有:

- SCHED_OTHER 0 非实时进程,基于优先权的轮转法(round robin)
- SCHED_FIFO 1 实时进程,用先进先出算法
- SCHED_RR 2 实时进程,用基于优先权的轮转法

1.1.2 信号处理

1.sigset_t blocked;

进程所能接收信号的位掩码。置位表示屏蔽,复位表示不屏蔽。

2.struct signal_struct * sig;

因为 signal 和 blocked 都是 32 位的变量,Linux 最多只能接受 32 种信号。对每种信号,各进程可以由 PCB 的 sig 属性选择使用自定义的处理函数,或是系统的缺省处理函数。指派各种信息处理函数的结构定义在 include/linux/sched.h 中。对信号的检查安排在系统调用结束后,以及在"慢速型"中断服务程序结束后(参见 5.5 节)。

3.int exit_code, exit_signal;

系统强行退出时发出的信号。

4.int pdeath_signal;

当父进程结束时所发出的信号。

1.1.3 进程队列指针

1.struct task_struct * next_ask, * prev_task;

所有进程(以 PCB 的形式)组成一个双向链表。next_task 和 prev_task 就是链表的前后向指针。链表的头和尾都是 init_task(即 0 号进程)。通过宏 for_each_task 可以很方便地搜索所有进程:

include/linux/sched.h

```
824 #define for_each_task(p) \
825     for (p = &init_task ; (p = p->next_task) != &init_task ; )
```

include/linux/sched.h

我们可以看到搜索进程时以 init_task 作为开头与结尾,这样做是比较安全的,因为 init_task 是永远不会退出的。

2. struct task_struct * p_opptr, * p_pptr;
   struct task_struct * p_cptr, * p_ysptr, * p_osptr;

以上分别是指向原始父进程(original parent)、父进程(parent)、子进程(youngest child)及新老兄弟进程(younger sibling, older sibling)的指针。相关的操作宏参见 kernel/linux/sched.h。它们之间的关系由图 1-1 给出。

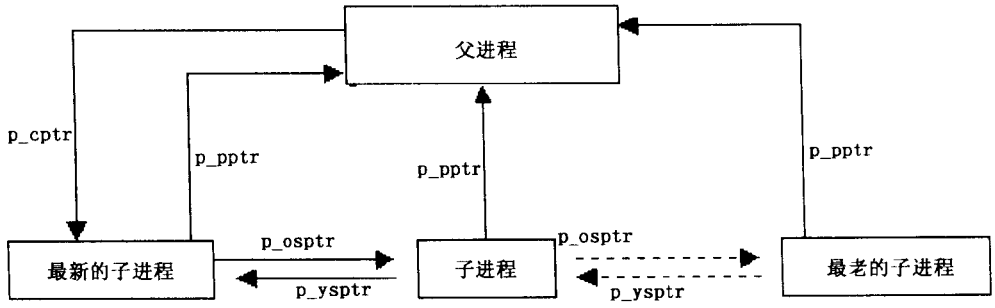


图1-1　父子进程间关系

3. struct task_struct * pidhash_next;
   struct task_struct ** pidhash_pprev;

用于链入进程 hash 表的前后指针。系统进程除了链入双向链表外,还被加到 hash 表中。Hash 表的定义如下:

kernel/fork.c

```
35 struct task_struct * pidhash[PIDHASH_SZ];
```

kernel/fork.c

PIDHASH_SZ 在 include/linux/sched.h 中定义,其值为 1024。

系统根据进程的进程号求得 hash 值,加到 hash 表中。

include/linux/sched.h

```
498 # define pid_hashfn(x) ((((x) > > 8) ^ (x)) & (PIDHASH_SZ-1))
```

include/linux/sched.h

若知道进程号,那么可以通过 hash 表很快地找到该进程,查找函数如下:

include/linux/sched.h

```
517 static inline struct task_struct * find_task_by_pid(int pid)
518 {
519     struct task_struct * p, ** htable = &pidhash[pid_hashfn(pid)];
520
521     for(p = * htable; p && p->pid != pid; p = p->pidhash_next)
522         ;
523
524     return p;
525 }
```

include/linux/sched.h

1.1.4 进程标识

1. uid_t uid, euid, suid, fsuid;
   gid_t gid, egid, sgid, fsgid;

uid 和 gid 分别是运行进程的用户标识和用户组标识。

euid 和 egid 又称为有效的 uid 和 gid。出于系统安全权限的考虑，运行程序时要检查 euid 和 egid 的合法性。通常，uid 等于 euid，gid 等于 egid。有时候，系统会赋予一般用户暂时拥有 root 的 uid 和 gid(作为用户进程的 euid 和 egid)，以便于进行运作。

suid 和 sgid 是根据 POSIX 标准引入的，在系统调用改变 uid 和 gid 时，用于保留真正的 uid 和 gid。

fsuid 和 fsgid 称为文件系统的 uid 和 gid，用于对文件系统操作时的合法性检查，是 Linux 独特的标识类型。它们一般分别和 euid 和 egid 一致，但在 NFS 文件系统中 NFS 服务器需要作为一个特殊的进程访问文件，这时只修改客户进程的 fsuid 和 fsgid 即可。

2. gid_t groups[NGROUPS];

与多数现代 UNIX 操作系统一样，Linux 允许进程同时拥有一组用户组号。在进程访问文件时，这些组号可用于合法性检查。

3. struct list_head thread_group;

线程组链表接口。

4. pid_t pid;
   pid_t pgrp;
   pid_t tty_old_pgrp;
   pid_t session;
   pid_t tgid;

进程标识号，进程的组标识号及 session 标识号，相关系统调用见程序 kernel/sys.c，有 sys_setpgid、sys_getpgid、sys_getpgrp、sys_setpgrp、sys_getsid 及 sys_setsid。

5. int leader;

是否是 session 的主管，布尔量。

6. struct user_struct * user;

进程使用的用户信息，包括该用户使用的进程数目、打开文件数等。user_struct 的结构定义在 include/linux/sched.h 中：

```
struct user_struct {
atomic_t __count;     /* reference count */
atomic_t processes;       /* How many processes does this user have? */
atomic_t files;       /* How many open files does this user have? */

/* Hash table maintenance information */
struct user_struct *next, **pprev;
uid_t uid;
};
```