2.2 示例：Linux 的 PCB

在 Linux 中，为了便于管理，使用 task_struct 结构来表示一个进程，每个进程都有自己独立的 task_struct。在这个结构体里，包含着这个进程的所有资源（或者到这个进程其他资源的链接）。task_struct 相当于进程在内核中的描述，以 2.6 内核为例的 task_struct 结构如下：

```
include/linux/sched.h, line 701
701 struct task_struct {
702         volatile long state;    /* -1 unrunnable, 0 runnable, >0 stopped */
703         struct thread_info *thread_info;
704         atomic_t usage;
705         unsigned long flags;    /* per process flags, defined below */
...
713         int prio, static_prio;
714         struct list_head run_list;
715         prio_array_t *array;
...
719         unsigned long sleep_avg;
720         unsigned long long timestamp, last_ran;
721         unsigned long long sched_time; /* sched_clock time spent running */
722         int activated;
723
724         unsigned long policy;
...
726         unsigned int time_slice, first_time_slice;
...
732         struct list_head tasks;
...
```

```
740         struct mm_struct *mm, *active_mm;
741
742 /* task state */
743         struct linux_binfmt *binfmt;
744         long exit_state;
745         int exit_code, exit_signal;
746         int pdeath_signal;  /*  The signal sent when the parent dies  */
748         unsigned long personality;
749         unsigned did_exec:1;
750         pid_t pid;
751         pid_t tgid;
752         /*
753          * pointers to (original) parent process, youngest child, younger sibling,
754          * older sibling, respectively.  (p->father can be replaced with
755          * p->parent->pid)
756          */
757         struct task_struct *real_parent; /* real parent process (when being debugged) */
758         struct task_struct *parent;     /* parent process */
759         /*
760          * children/sibling forms the list of my children plus the
761          * tasks I'm ptracing.
762          */
763         struct list_head children;      /* list of my children */
764         struct list_head sibling;       /* linkage in my parent's children list */
765         struct task_struct *group_leader;       /* threadgroup leader */
766
767         /* PID/PID hash table linkage. */
768         struct pid pids[PIDTYPE_MAX];
769
774         unsigned long rt_priority;
775         cputime_t utime, stime;
776         unsigned long nvcsw, nivcsw; /* context switch counts */
777         struct timespec start_time;
```

```
778 /* mm fault and swap info: this can arguably be seen as either mm-specific or thread-specific */
779         unsigned long min_flt, maj_flt;
780
781         cputime_t it_prof_expires, it_virt_expires;
782         unsigned long long it_sched_expires;
783         struct list_head cpu_timers[3];
784
785 /* process credentials */
786         uid_t uid,euid,suid,fsuid;
787         gid_t gid,egid,sgid,fsgid;
788         struct group_info *group_info;
789         kernel_cap_t   cap_effective, cap_inheritable, cap_permitted;
790         unsigned keep_capabilities:1;
791         struct user_struct *user;
...
797         int oomkilladj; /* OOM kill score adjustment (bit shift). */
798         char comm[TASK_COMM_LEN]; /* executable name excluding path
799                                      - access with [gs]et_task_comm (which lock
800                                        it with task_lock())
801                                      - initialized normally by flush_old_exec */
802 /* file system info */
803         int link_count, total_link_count;
804 /* ipc stuff */
805         struct sysv_sem sysvsem;
806 /* CPU-specific state of this task */
807         struct thread_struct thread;
808 /* filesystem information */
809         struct fs_struct *fs;
810 /* open file information */
811         struct files_struct *files;
812 /* namespace */
813         struct namespace *namespace;
```

```
814 /* signal handlers */
815         struct signal_struct *signal;
816         struct sighand_struct *sighand;
817
818         sigset_t blocked, real_blocked;
819         sigset_t saved_sigmask;         /* To be restored with TIF_RESTORE_SIGMASK */
820         struct sigpending pending;
821
...
837 /* Thread group tracking */
838         u32 parent_exec_id;
839         u32 self_exec_id;
840 /* Protection of (de-)allocation: mm, files, fs, tty, keyrings */
841         spinlock_t alloc_lock;
842         /*         Protection         of         proc_dentry:         nesting         proc_lock,         dcache_lock,
write_lock_irq(&tasklist_lock); */
843         spinlock_t proc_lock;
844
...
850 /* journalling filesystem info */
851         void *journal_info;
852
853 /* VM state */
854         struct reclaim_state *reclaim_state;
855
856         struct dentry *proc_dentry;
857         struct backing_dev_info *backing_dev_info;
858
859         struct io_context *io_context;
860
863 /*
864  * current io wait handle: wait queue entry to use for io waits
865  * If this thread is processing aio, this points at the waitqueue
```

```
866  * inside the currently handled kiocb. It may be NULL (i.e. default
867  * to a stack based synchronous wait) if its doing sync IO.
868  */
869         wait_queue_t *io_wait;
870 /* i/o counters(bytes read/written, #syscalls */
871         u64 rchar, wchar, syscr, syscw;
...
888 };
```

702 进程的状态： -1 表示 unrunnable，0 表示 runnable，>0 表示 stopped；

703 指向 thread_info 的指针。关于 thread_info，后面会有说明的；

705 进程的一些标志位，等一会说明；

713-726 这一组基本都是调度器相关的一些变量；

713 进程的优先级；

714 优先级相同的进程组成的一个链表；

715 进程所在的优先级队列；

719 平均睡眠时间；

724 调度策略；

726 时间片相关变量；

732 用于链接系统中所有进程的链表；

740 指向内存管理数据结构的指针；

742-751 进程状态相关的一些信息；

743 二进制代码结构类型；

744　　退出状态；

745　　退出代码，退出信号；

750　　进程 id，每个进程都有唯一的 id；

752-765　进程家族关系的一些信息；

775　　进程在用户态执行的时间，和在内核态执行的时间；

777　　进程启动的时刻，使用 jiffies 标记；

781-783　定时器相关的几个变量；

785-791　进程授权，文件系统权限等相关的一些信息；

798　　该进程的名称，一般来说就是可执行程序名；

805　　进程间通信相关信息；

807　　保存 CPU 相关的该进程的信息，比如寄存器；

809　　该进程相关的文件系统的信息；

811　打开文件信息；

814-821　信号量相关信息；

851　　日志文件系统相关信息；

task_struct 中包含的信息非常多，这里只讨论了一部分变量（如果需要，请读者参看源代码中完整的 task_struct）。一方面，这是由于进程必须要知道/控制它所拥有的所有系统资源；另一方面，内核越来越复杂，加入功能模块也越来越多，大家都把和进程相

关的信息一股脑扔到 task_struct 里面，导致 task_struct 似乎有越来越臃肿的趋势。

接下来几个小节结合 task_struct 中的内容，分别对进程相关的概念做一些讨论。

2.1 task_struct 与内核栈

由于 2.4 版本及之前的 Linux 内核中，task_struct 和内核堆栈是放在同一个 4K 页面中的。如下：

```
include/linux/sched.h 2.4.18
511 union task_union {
512         struct task_struct task;
513         unsigned long stack[INIT_TASK_SIZE/sizeof(long)];
514 };
```

用图来表示的话，就是图 2-3。

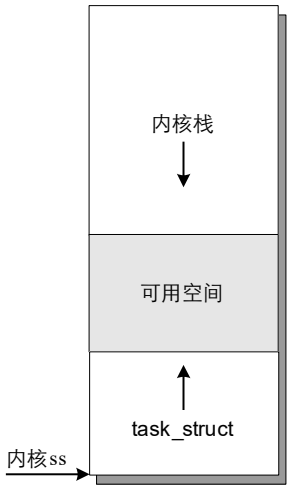

图 2-3 堆栈和 task_struct

这样的设计非常巧妙，因为在内核运行时，任何时候我们都可以通过栈指针得到当前运行进程的 task_struct，这给进程管理带来了非常大的方便。然而，这样实现的隐患也一直被很多内核黑客所讨论：如果 task_struct 越来越大怎么办？如果内核堆栈压得太多

（比如函数调用层次太深）怎么办?

2.6 版本的内核中采取两个办法（思路）来弥补这个缺陷。

1. 增大这部分空间：在 2.6 版的内核中，这部分空间的默认值从原先的 4K 增大到 8K:

```
include/asm-i386/thread_info.h, line 60
60 #define THREAD_SIZE             (8192)
...
111 #define alloc_thread_info(tsk) kmalloc(THREAD_SIZE, GFP_KERNEL)
```

2. 把 task_struct 从这部分空间中移走：在 2.6 版的内核中，抽象出一个 thread_info 的结构（把最经常被 entry.S 访问的变量抽出来）。

```
include/asm-i386/thread_info.h, line 28
28 struct thread_info {
29            struct task_struct        *task;              /* main task
structure */
30          struct exec_domain      *exec_domain;    /* execution domain
*/
31          unsigned long            flags;           /* low level flags
*/
32            unsigned long               status;             /* thread-
synchronous flags */
33          __u32                   cpu;             /* current CPU */
34          int                     preempt_count;   /* 0 => preemptable,
<0 => BUG */
35
36
37          mm_segment_t              addr_limit;      /* thread address
space:
```

```
38                                                      0-0xBFFFFFFF for
user-thead
39                                                      0-0xFFFFFFFF for
kernel-thread
40                                                          */
41          void                    *sysenter_return;
42          struct restart_block    restart_block;
43
44            unsigned long             previous_esp;    /* ESP of the
previous stack in case
45                                                       of nested (IRQ)
stacks
46                                                          */
47          __u8                    supervisor_stack[0];
48 };
```

thread_info 代替了原先 task_struct 的位置，跟内核堆栈放在一块，thread_info 中放置一个指向 task_struct 的指针，如图 2-4。

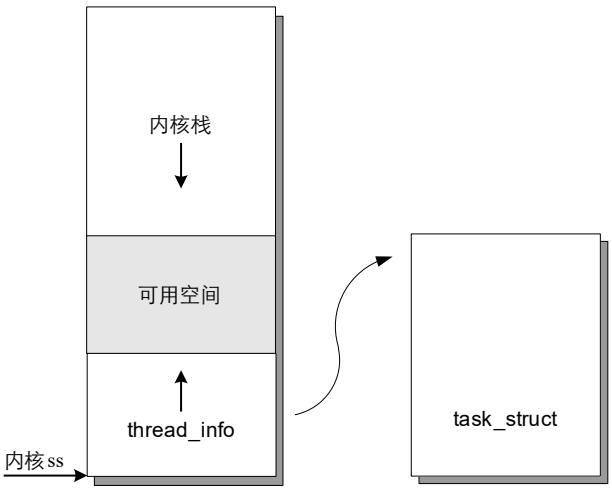

图 2-4 thread_info 和内核堆栈

相应的，大家熟悉的 current 宏，内部实现中，现在也做了相应的改变，先根据内核堆栈的位置找到 thread_info，然后在根据 thread_info 找到进程的 task_struct。

2.2 状态转换

```
volatile long state
long exit_state;
```

用于表示内核的状态，前者表示用来表征进程的可运行性，后者表征进程退出时候的状态。

```
include/linux/sched.h, line 114
114 /*
115  * Task state bitmask. NOTE! These bits are also
116  * encoded in fs/proc/array.c: get_task_state().
117  *
118  * We have two separate sets of flags: task->state
119  * is about runnability, while task->exit_state are
120  * about the task exiting. Confusing, but this way
121  * modifying one set can't modify the other one by
122  * mistake.
123  */
124 #define TASK_RUNNING            0
125 #define TASK_INTERRUPTIBLE      1
126 #define TASK_UNINTERRUPTIBLE    2
127 #define TASK_STOPPED            4
128 #define TASK_TRACED             8
129 /* in tsk->exit_state */
130 #define EXIT_ZOMBIE             16
131 #define EXIT_DEAD               32
132 /* in tsk->state again */
133 #define TASK_NONINTERACTIVE     64
```

它们的含义分别是:

TASK_RUNNING：正在运行的进程即系统的当前进程或准备运行的进程即在 Running 队列中的进程。只有处于该状态的进程才实际参与进程调度。

TASK_INTERRUPTIBLE：处于等待资源状态中的进程，当等待的资源有效时被唤醒，也可以被其他进程或内核用信号中断、唤醒后进入就绪状态。

TASK_UNINTERRUPTIBLE：处于等待资源状态中的进程，当等待的资源有效时被唤醒，不可以被其它进程或内核通过信号中断、唤醒。

TASK_STOPPED：进程被暂停，一般当进程收到下列信号之一时进入这个状态：SIGSTOP，SIGTSTP，SIGTTIN 或者 SIGTTOU。通过其它进程的信号才能唤醒。

TASK_TRACED：进程被跟踪，一般在调试的时候用到。

EXIT_ZOMBIE：正在终止的进程，等待父进程调用 wait4()或者 waitpid()回收信息。是进程结束运行前的一个过度状态（僵死状态）。虽然此时已经释放了内存、文件等资源，但是在内核中仍然保留一些这个进程的数据结构（比如 task_struct）等待父进程回收。

EXIT_DEAD：进程消亡前的最后一个状态，父进程已经调用了 wait4()或者 waitpid()。

TASK_NONINTERACTIVE：表明这个进程不是一个交互式进程，在调度器的设计中，对交互式进程的运行时间片会有一定的奖励或者惩罚。

状态转换图见图2-5。

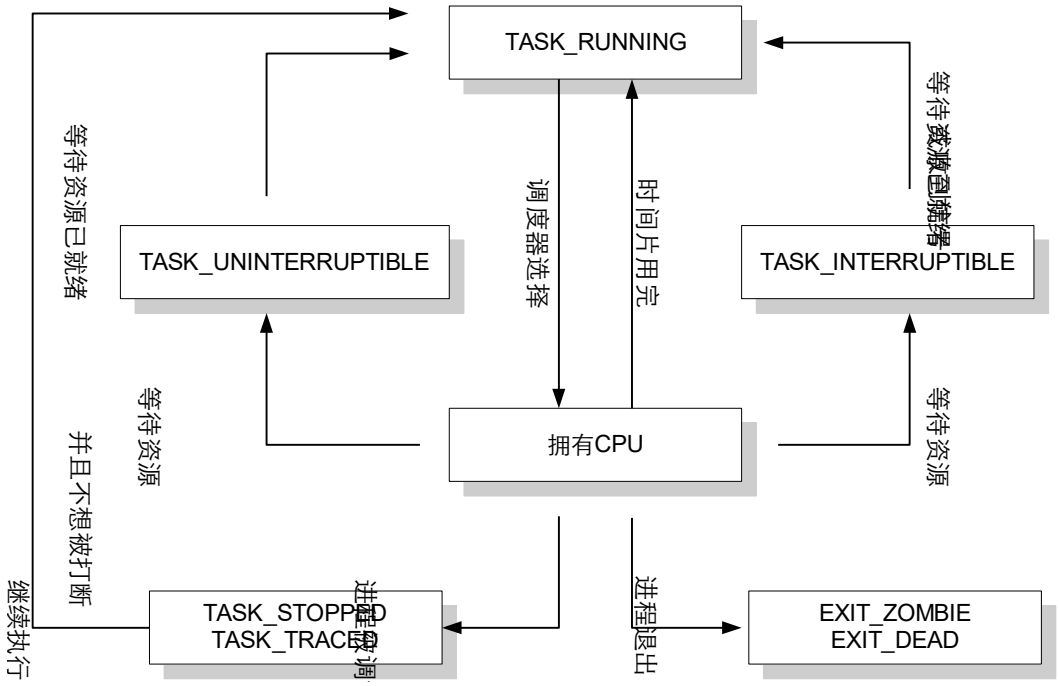

图 2-5 进程状态转换图

2.3 进程标志位

为了对每个进程运行进行更细粒度的控制，还有一些进程标志位。在 task_struct 中有

变量 flags：

```
unsigned long flags; /* per process flags, defined below */
```

这个 flags 可以是下面一些标志的组合：

```
include/linux/sched.h, line 920
919 /*
920  * Per process flags
921  */
922 #define PF_ALIGNWARN     0x00000001        /* Print alignment warning
msgs */
923                                          /* Not implemented yet, only
for 486*/
```

```
924 #define PF_STARTING     0x00000002      /* being created */
925 #define PF_EXITING      0x00000004      /* getting shut down */
926 #define PF_DEAD         0x00000008      /* Dead */
927 #define PF_FORKNOEXEC   0x00000040      /* forked but didn't exec */
928 #define PF_SUPERPRIV     0x00000100      /* used super-user privileges
*/
929 #define PF_DUMPCORE     0x00000200      /* dumped core */
930 #define PF_SIGNALED     0x00000400      /* killed by a signal */
931 #define PF_MEMALLOC     0x00000800      /* Allocating memory */
932 #define PF_FLUSHER        0x00001000        /* responsible for disk
writeback */
933 #define PF_USED_MATH     0x00002000       /* if unset the fpu must be
initialized before use */
934 #define PF_FREEZE        0x00004000      /* this task is being frozen
for suspend now */
935 #define PF_NOFREEZE      0x00008000      /* this thread should not be
frozen */
936 #define PF_FROZEN        0x00010000      /* frozen for system suspend
*/
937 #define PF_FSTRANS        0x00020000        /* inside a filesystem
transaction */
938 #define PF_KSWAPD       0x00040000      /* I am kswapd */
939 #define PF_SWAPOFF      0x00080000      /* I am in swapoff */
940 #define PF_LESS_THROTTLE 0x00100000      /* Throttle me less: I clean
memory */
941 #define PF_SYNCWRITE    0x00200000      /* I am doing a sync write */
942 #define PF_BORROWED_MM  0x00400000      /* I am a kthread doing use_mm
*/
943 #define PF_RANDOMIZE     0x00800000      /* randomize virtual address
space */
944 #define PF_SWAPWRITE    0x01000000      /* Allowed to write to swap */
```

这些标志的含义分别为：

PF_ALIGNWARN	标志打印“对齐”警告信息。
PF_STARTING	进程正被创建。
PF_EXITING	标志进程开始关闭。
PF_DEAD	标志进程已经完成退出。
PF_FORKNOEXEC	进程刚创建，但还没执行。
PF_SUPERPRIV	超级用户特权标志。
PF_DUMPCORE	标志进程是否清空 core 文件。
PF_SIGNALED	标志进程被信号杀出。
PF_MEMALLOC	进程分配内存标志。
PF_FLUSHER	负责磁盘写回。
PF_USED_MATH	如果没有置位，那么使用 fpu 之前必须初始化。
PF_FREEZE	由于系统要进入休眠，进程正在被停止。
PF_NOFREEZE	系统睡眠的时候，这个进程不能被停止。
PF_FROZEN	系统要进入睡眠，进程被停止。
PF_FSTRANS	在一个文件系统事务之中。
PF_KSWAPD	kswapd 内核线程。
PF_SWAPOFF	在换出页的过程中。
PF_LESS_THROTTLE	尽可能少把我换出。
PF_SYNCWRITE	负责把脏页写回。

PF_BORROWED_MM　　内核线程借用进程的 mm。

PF_RANDOMIZE　　随机虚拟地址空间。

PF_SWAPWRITE　　允许被写到 swap 中去。

这些标志对进程的运行产生各个方面的影响，但是脱离开具体的实例也不是很好分析，这里就不具体展开了。只举个例子，比如 PF_MEMALLOC 标志（正在分配内存）带有这个标志的进程，如果要分配内存的话，buddy system 即使在内存紧张的时候也要尽量满足这个进程的分配请求（可参考 kswapd 内核线程的代码 mm/vmscan.c line 1692）。

2.4 进程与调度

task_struct 中与进程调度相关的一些变量有：

`unsigned long policy:` 进程调度策略

Linux 中现在有四种类型的调度策略：

```
include/linux/sched.h, line 159
159 /*
160  * Scheduling policies
161  */
162 #define SCHED_NORMAL            0
163 #define SCHED_FIFO              1
164 #define SCHED_RR                2
165 #define SCHED_BATCH             3
```

每个进程都有自己的调度策略，系统中大部分进程的调度策略是

SCHED_NORMAL，有 root 权限的进程能改变自己和别的进程的调度策略。调度器根据每个进程的调度策略给予不同的优先级。

这四种调度策略之间差别很大，比如 SCHED_FIFO 和 SCHED_RR 属于实时进程调度策略，它们的优先级比 SCHED_NORMAL 和 SCHED_BATCH 都要高，如果一个实时进程准备运行，调度器总是试图先调度实时进程。SCHED_BATCH 是 2.6 版新加入的调度策略，这种类型的进程一般都是后台处理进程，总是倾向于跑完自己的时间片，没有交互性，调度器也不会对这类进程进行优先级奖惩。所以对于这种调度策略的进程，调度器一般给的优先级比较低，这样系统就能在没什么事情做的时候运行这些进程，而一旦有交互性的进程需要运行，则立刻切换到交互性的进程，从用户的角度来看，系统的响应性/交互性就很好。

进程的调度优先级。

```
int prio, static_prio;
unsigned long rt_priority;
```

prio 是进程的动态优先级，随着进程的运行而改变，调度器有时候还会根据进程的交互特性，平均睡眠时间等进行奖惩。系统默认的设置下，实时进程（SCHED_FIFO 和 SCHED_RR）的动态优先级范围为 0 到 99；非实时进程（SCHED_NORMAL 和 SCHED_BATCH）的动态优先级范围为 100 到 139。需要注意的是，优先级 0 为最高，

139 为最低的优先级。

static_prio 为普通进程的静态优先级，默认为 120。

rt_priority 为实时进程的静态优先级，

关于进程调度的详细信息，如果详细展开的话，也许需要另外独立的一整个章节。

2.5 进程 id，父进程 id，兄弟进程

每个进程都有自己独立的一个 id：

```
pid_t pid;
```

每个进程（init 进程除外）都是由父进程派生出来（关于这一点，我们在进程的产生中会详细讲述），并且也可能有自己的兄弟进程（指属于同一个父进程的进程）。所有这些进程组成一个类似于家族的关系：

```
/*
 * pointers to (original) parent process, youngest child, younger sibling,
 * older sibling, respectively.  (p->father can be replaced with
 * p->parent->pid)
 */
struct task_struct *real_parent; /* 当被调试的时候保存进程的真正的父进程 */

struct task_struct *parent;  /* 父进程 */

/*
 * children/sibling forms the list of my children plus the
 * tasks I'm ptracing.
 */
struct list_head children;/* list of my children */
struct list_head sibling; /* linkage in my parent's children list */
struct task_struct *group_leader;  /* threadgroup leader */
```

这些指针的集合为浏览进程家族提供很大方便，比如在寻找进程祖先，或者查找进程的某一个子孙的时候。

例如，系统调用中，用以得到进程的 pid 和它父进程的 pid 的接口是:

- pid_t getpid(void) : this function returns the PID of the process
- pid_t getppid(void) : this function returns the PID of the parent process

相关的例子详见本章实验 1.

2.6 用户 id，组 id

在 task_struct 里面维护了一些跟文件系统权限控制相关的一些变量。

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

uid: 是创建这个进程的用户的 id。在传统 Unix 系统的管理中，每个用户都有自己的访问系统的权限，Unix 管理每个用户，给每个用户分配一个 id 标志。比如:

Unix 根据这些 id（以及其他一些信息）控制每个用户的权限，比如一个普通用户不能创建用户，访问别的用户的家目录；而 root 用户（id 为 0）则几乎可以做任何事 。Linux 继承了 Unix 的这些行为。

uid 记录了创建这个进程的用户 id，相当于带着这个用户的授权，替这个用户去做一些事情。你可以认为系统通过一个进程的 uid，判断出哪个进程是代表着哪个用户来

执行命令。可以这样理解，然而事实并非如此。

euid: （effective uid, 即有效 uid。）事实上，系统是通过一个进程的 euid，来判断进程的权限的。为什么要这么做？在大多数的情况下，进程的 uid 和 euid 是相同的，但是在某些时候，进程需要以可执行文件的属主来运行那个程序，而不是以可执行程序的用户来运行。这个时候，euid 就是那个可执行文件的属主的用户 id。说起来很抽象，举个简单的例子：

你的系统中有一个改变用户密码的命令：passwd。由于这个程序需要修改/etc/passwd, /etc/shadow 等文件，所以需要是 root 权限：

```
[kai@localhost ~]$ ls -l  /usr/bin/passwd
-r-s--x--x 1 root root 21944 Feb 12  2006 /usr/bin/passwd
```

而且你可以看到这个命令的属性位中设置了 s 位，意思就是当普通用户执行这个命令的时候，具有该命令的属主 root 的权限，在你运行 passwd 命令的过程中，你的 euid 就是 root 的 id：0。

suid: (saved set-user-ID) 这是 POSIX 标准中要求的两个标识符。当有时候必须通过系统调用改变 uid 和 gid 的时候，需要用 suid 来保存真实的 uid。详细请见 getresuid, setresuid。

fsuid: Linux 内核检查进程对于文件系统的访问时所参考的位。一般来说等同于 euid，当 euid 改变的时候，fsuid 也会相应的被改变。这两个标识符最初是为了建立 NFS(Network File System， 网络文件系统)而使用的， 因为用户模式的 NFS 服务器需要像一个特别的进程一样来访问文件。 在这种情况下， 只有文件系统 uid 和 gid 被改变(有效的 uid 和 gid 不变)。这样可以防止恶意的用户向 NFS 服务器发送 kill 信号。 Kill 信号会被以一个特别的有效 uid 和 gid 发送到进程。（参考自 The Linux kernel）

详细请见 setfsuid 的 manpage。

对应的 gid, egid, sgid, fsgid 与上面讲到的类似，只不过对应的是用户组，不再累述。

使用下面的系统调用函数得到进程的 uid, gid 等。

```
int getresuid(uid_t *ruid, uid_t *euid, uid_t *suid);
int getresgid(gid_t *rgid, gid_t *egid, gid_t *sgid);
int setresuid(uid_t ruid, uid_t euid, uid_t suid);
int setresgid(gid_t rgid, gid_t egid, gid_t sgid);
int setfsuid(uid_t fsuid);
int setfsgid(uid_t fsgid);
```

2.7 进程自己的资源

从 task_struct 可以链接到很多属于该进程的资源，比如 mm_struct, vma_struct, fs 等等。

struct mm_struct *mm;

struct fs_struct *fs;

struct files_struct *files;

fs 和 files 结构主要用于管理进程当前的目录状况，和进程打开的所有文件。

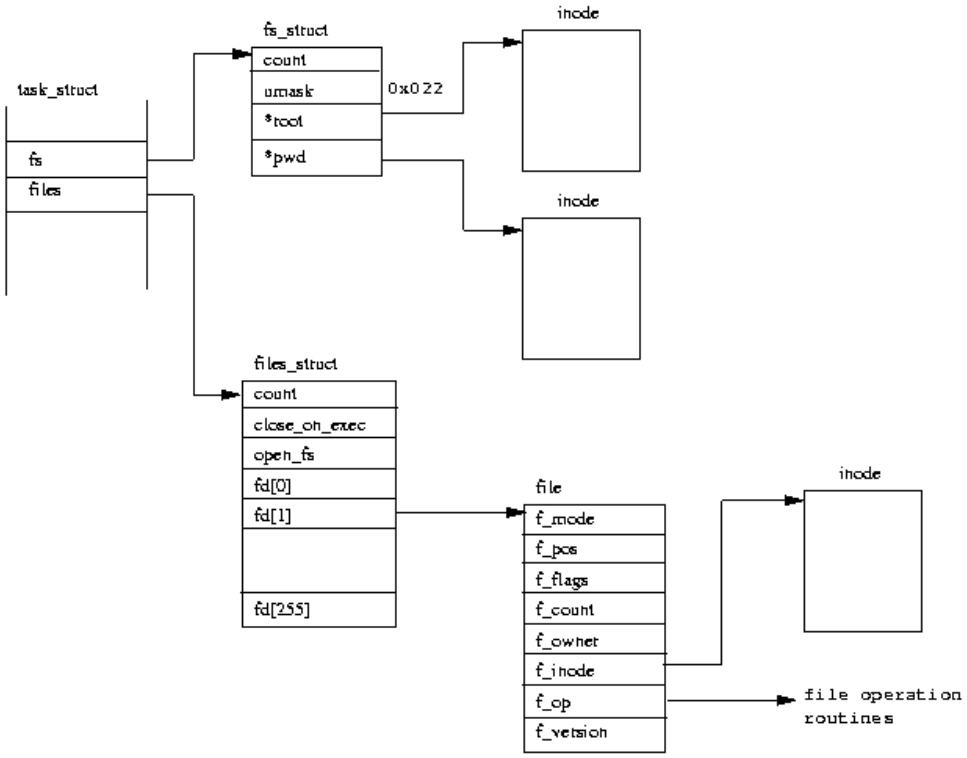

图 2-6 进程的 fs 和 files 结构

图 2-6 表明系统中的每个进程有 2 个数据结构描述文件系统相关的信息。

如图 2-6。第一：fs_struct，包含指针指向进程的 fs_struct，fs_struct 用来描述进程工作的文件系统的信息，包括根目录和当前工作目录的 dentry，它们 mount 的文件系统的

信息，以及在 umask 中保存的初始的打开文件的权限。

第二：files_struct，包含进程当前正在使用的所有文件的信息。比如进程从标准输入读并且写到标准输出；任何错误消息输出到标准出错。这三个设备可以是文件，终端输入/输出或一台真实的设备，但是在 Unix 中，程序都把它们当作文件。每个文件有它的自己的描述符， files_struct 中就包含可以指向这些文件数据结构的指针，每个可以描述进程打开的一个文件。f_mode 描述文件是以什么模式被创建的：只读，读写或者只写 。f_pos 记录下一个读或写操作的位置。f_inode 指向描述该文件的 VFS 索引节点，而 f_ops 指向操作这个文件的函数的函数集。

每打开一个文件，在 files_struct 的一个空闲的文件指针被用来指向新文件结构。每个 Linux 进程启动的时候，默认会有 3 个文件描述符被打开，它们是标准输入，标准输出和标准错误，这些通常都是从父进程中继承来的。 所有的文件访问都要使用系统调用，它们使用或者返回 file descriptor (文件描述符)。文件描述符是到进程的 fd 向量的索引，所以标准输入，标准输出和标准错误的文件描述符是 0，1 和 2。文件的每次访问基本都会使用文件数据结构的文件操作函数集。

而 mm_struct 主要是管理进程的整个内存空间。由 mm_struct 包含已装载的可执行的

映像的信息，还有到进程的页表的指针。进程的页表包含一些指针，指到 vm_area_struct 数据结构的一个表。每个 vm_area_struct 描述进程的一个内存区域，这个区域有较为独立的属性，比如这个区域映射的是某一个动态链接库，可读，不可写，不可执行，可以跟别的进程共享；而另一个区域则属于进程的堆，可读，可写，不可执行，进程私有不能共享。Linux 把这样的一个内存区域单独出来，便于对每个区域属性的管理，同时也便于在不同的进程间进行共享。