第 1 章 引言

Linux 是一种能运行于多种平台、源代码公开、免费、功能强大、遵守 POSIX 标准、与 UNIX 兼容的操作系统。

Linux 最初版本是由 Linus Benedict Torvalds 编写的，为了能够使 Linux 更加完善,Torvalds 在网络上公开了 Linux 的源码,邀请全世界的志愿者来参与 Linux 的开发。在这些无私的人们的帮助下，Linux 得到不断的完善，并在短时期内迅速崛起。现在，Linux 内核已经发展到了 2.5.X 版，并还在以相当快的速度不断地发展着。据报道,它是一个很有发展前途的操作系统,也是为数不多可以与 Microsoft 旗下操作系统相竞争的操作系统。

我国的 IT 产业起步较晚，技术落后于西方经济发达国家。在我国，由于受知识产权的限制，无论是使用 PC 平台上的 Windows，还是使用应用于大中型机的 UNIX，都无法窥视到其内部结构。这些系统很可能存在不为我们所知的漏洞，如果这些漏洞为别有用心者所利用，将会严重危及我国的经济安全和国家安全。操作系统不同于其它软件产品，它是其它应用程序得以运行的平台，应用软件的开发必须基于对相应平台(操作系统)的技术的理解和掌握。由于我们的软件企业无法获知这些系统的细节，根本无法与拥有这些关键技术的国外先进企业相抗衡，长此以往，将会对我国软件产业产生深远的负面影响。

为了打破这种受制于人的局面,我国迫切的需要开发一个具有自主知识产权的操作系统。由于 Linux 性能优越，属于自由软件，公开源代码且完全免费，所以拿来作发展自主知识产权的操作系统的底版是非常合适的选择。只有完全了解当前 Linux 的实现细节，才能在它的基础上作进一步的开发和完善。因此，分析 Linux 源代码对于在 Linux 现有的基础上开发我们自己的 Linux 就具有非常现实和重要的意义。

本论文主要在源码水平上讨论 Linux 内核进程调度与控制的实现,其目的是通过对源码的分析与研究,找出 Linux 在本论文研究方向上的优缺点,作为今后 Linux 系统开发的参考。

第 2 章 Linux 内核的整体结构

Linux 内核由 5 个主要的子系统组成。这 5 个子系统分别是进程调度(SCHED)、内存管理(MM)、虚拟文件系统(Virtual File System,VFS)、网络接口(NET)和进程间通信(IPC)。

进程调度控制着进程对 CPU 的访问。当需要选择下一个进程运行时，由调度程序选择最值得运行的进程。可运行进程实际是仅等待 CPU 资源的进程，如果某个进程在等待其它资源，则该进程是不可运行进程。Linux 使用了比较简单的基于优先级的进程调度算法选择新的进程。

内存管理允许多个进程安全地共享主内存区域。Linux 的内存管理支持虚拟内存，即在计算机中运行的程序，其代码、数据和堆栈的总量可以超过实际内存的大小,操作系统只将当前使用的程序块保留在内存中,其余的程序块则保留在磁盘上。必要时，操作系统负责在磁盘和内存之间交换程序块。

内存管理从逻辑上可以分为硬件无关的部分和硬件相关的部分。硬件无关的部分提供了进程的映射和虚拟内存的对换;硬件相关的部分为内存管理硬件提供了虚拟接口。

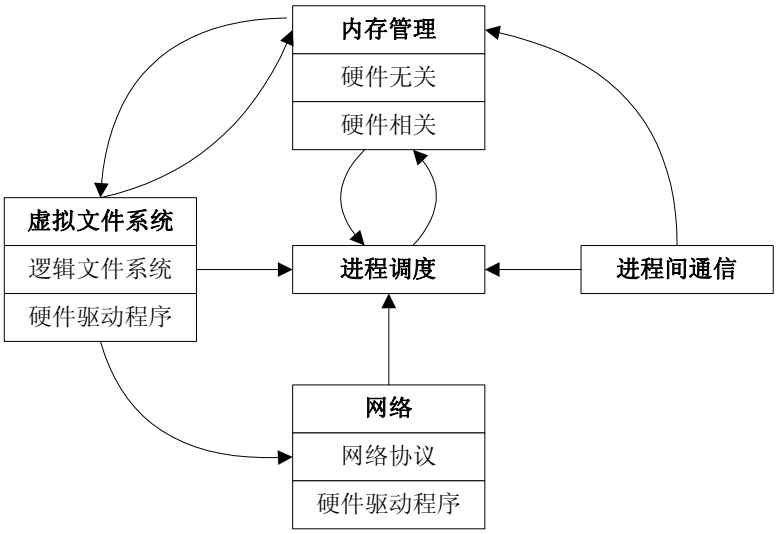
虚拟文件系统隐藏了各种不同硬件的具体细节，为所有设备提供了统一的接口，虚拟文件系统还支持多达数十种不同的文件系统，这也是 Linux 较有特色的部分。

虚拟文件系统可分为逻辑文件系统和设备驱动程序。逻辑文件系统指 Linux 所支持的文件系统，如 ext2、fat 等，设备驱动程序指为每一种硬件控制器所编写的设备驱动程序模块。

网络接口提供了对各种网络标准的存取和各种网络硬件的支持。网络接口可分为网络协议和网络驱动程序两部分。网络协议部分负责实现每一种可能的网络传输协议，网络设备驱动程序负责与硬件设备进行通信，每一种可能的硬件设备都有相应的设备驱动程序。

进程间通信支持进程间各种通信机制。

下图显示了上述五个子系统之间的关系：



各个子系统之间的依赖关系如下：

1. 进程调度与内存管理之间的关系：

这两个子系统互相依赖。在多道程序环境下，程序要运行必须为之创建进程，而创建进程的第一件事，就是要将程序和数据装入内存。

2. 进程间通信与内存管理之间的关系：

进程间通信子系统要依赖内存管理支持共享内存通信机制,这种机制允许两个进程除了拥有自己的私有内存，还可存取共同的内存区域。

3. 虚拟文件系统与网络接口之间的关系：

虚拟文件系统利用网络接口支持网络文件系统(NFS)，也利用内存管理支持 RAMDISK 设备。

4. 内存管理与虚拟文件系统之间的关系：

内存管理利用虚拟文件系统支持交换，交换进程定期地由调度程序调度，这也是内存管理依赖于进程调度的唯一原因。当一个进程存取的内存映射被换出时，内存管理向文件系统发出请求，同时，挂起当前当前正在运行的进程。

在这些子系统中，进程调度子系统是其他子系统得以顺利工作的关键。无论是文件系统的系统进程还是网络子系统的服务进程都需要通过进程调度来获得相应的 CPU 时间以正常运行。

第 3 章 Linux 进程调度

3.1 相关概念简述

3.1.1Linux 进程的四个要素

一般来说 Linux 系统的进程都具备下列诸要素:

(1) 有一段程序供其执行。这段程序不一定是某个进程所专有，可以与其他进程共用。
(2) 有进程专用的内核空间堆栈。
(3) 在内核中有一个 task_struct 数据结构，即通常所说的“进程控制块”。有了这个数据结构，进程才能成为内核调度的一个基本单位接受内核的调度。同时，这个结构还记录着进程所占用的各项资源。
(4) 有独立的存储空间，这意味着拥有专有的用户空间；进一步，还意味着除前述的内核空间堆栈外还有其专用的用户空间堆栈。有一点必须指出，内核空间是不能独立的，任何进程都不可能直接（不通过系统调用）改变内核空间的内容（除其本身的内核空间堆栈以外）。

这四条都是必要条件，缺了任何一条都不能成为“进程”。如果只具备了前三条而缺第四条，就称为“线程”。如果完全没有用户空间，就称为“内核线程”（**kernel thread**）；而如果共享用户空间则称为“用户线程”。二者往往都简称“线程”。事实上，在 Linux 系统中，进程和线程的区分并不十分严格，许多进程在“诞生”之初都与其父进程共用同一个存储空间，所以严格说来还是线程；但是子进程可以建立其自己的存储空间，并与父进程分离，成为真正意义上的进程。

3.1.2 task_struct 结构描述

task_struct 结构是向系统表明进程存在的唯一凭证，它包含了进程的全部信息。同时，也是进程为实现操作而取得必要资源的唯一途径。下面列出了 task_struct 结构的全部源码，在源码后面有对 task_struct 结构（参见 include\linux\sched.h）各数据项的分类解释：

```
struct task_struct {
/* these are hardcoded - don't touch */
    volatile long state;    /* -1 unrunnable, 0 runnable, >0 stopped */
    unsigned long flags;  /* per process flags, defined below */
    int sigpending;
    mm_segment_t addr_limit;    /* thread address space:
                                   0-0xBFFFFFFF for user-thead
                                   0-0xFFFFFFFF for kernel-thread
                               */
    struct exec_domain *exec_domain;
    long need_resched;
```

```
    /* various fields */
        long counter;
        long priority;
        cycles_t avg_slice;
    /* SMP and runqueue state */
        int has_cpu;
        int processor;
        int last_processor;
        int lock_depth;        /* Lock depth. We can context switch in and out
of holding a syscall kernel lock... */
        struct task_struct *next_task, *prev_task;
        struct list_head run_list;

    /* task state */
        struct linux_binfmt *binfmt;
        int exit_code, exit_signal;
        int pdeath_signal;   /*    The signal sent when the parent dies    */
        /* ??? */
        unsigned long personality;
        int dumpable:1;
        int did_exec:1;
        pid_t pid;
        pid_t pgrp;
        pid_t tty_old_pgrp;
        pid_t session;
        /* boolean value for session group leader */
        int leader;
        /*
          * pointers to (original) parent process, youngest child, younger sibling,
          * older sibling, respectively.    (p->father can be replaced with
          * p->p_pptr->pid)
          */
        struct task_struct *p_opptr, *p_pptr, *p_cptr, *p_ysptr, *p_osptr;

        /* PID hash table linkage. */
        struct task_struct *pidhash_next;
        struct task_struct **pidhash_pprev;

        wait_queue_head_t wait_chldexit; /* for wait4() */
        struct semaphore *vfork_sem;        /* for vfork() */
        unsigned long policy, rt_priority;
        unsigned long it_real_value, it_prof_value, it_virt_value;
        unsigned long it_real_incr, it_prof_incr, it_virt_incr;
```

```
        struct timer_list real_timer;
        struct tms times;
        unsigned long start_time;
        long per_cpu_utime[NR_CPUS], per_cpu_stime[NR_CPUS];
    /* mm fault and swap info: this can arguably be seen as either mm-specific or thread-specific */
        unsigned long min_flt, maj_flt, nswap, cmin_flt, cmaj_flt, cnswap;
        int swappable:1;
    /* process credentials */
        uid_t uid,euid,suid,fsuid;
        gid_t gid,egid,sgid,fsgid;
        int ngroups;
        gid_t      groups[NGROUPS];
        kernel_cap_t      cap_effective, cap_inheritable, cap_permitted;
        struct user_struct *user;
    /* limits */
        struct rlimit rlim[RLIM_NLIMITS];
        unsigned short used_math;
        char comm[16];
    /* file system info */
        int link_count;
        struct tty_struct *tty; /* NULL if no tty */
    /* ipc stuff */
        struct sem_undo *semundo;
        struct sem_queue *semsleeping;
    /* CPU-specific state of this task */
        struct thread_struct thread;
    /* filesystem information */
        struct fs_struct *fs;
    /* open file information */
        struct files_struct *files;

    /* memory management info */
        struct mm_struct *mm, *active_mm;

    /* signal handlers */
        spinlock_t sigmask_lock; /* Protects signal and blocked */
        struct signal_struct *sig;
        sigset_t signal, blocked;
        struct signal_queue *sigqueue, **sigqueue_tail;
        unsigned long sas_ss_sp;
        size_t sas_ss_size;
    };
```

1. 调度数据成员

(1) volatile long states;
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，也可由其它进程通过信号(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，不可由其它进程通过信号(signal)或定时中断唤醒。

• 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，通过其它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，或者是受其它进程的 ptrace 系统调用的控制而暂时将 CPU 交给控制进程。

• TASK_SWAPPING: 进程页面被交换出内存的进程。

(2) unsigned long flags;
进程标志:

• PF_ALIGNWARN	打印“对齐”警告信息。
• PF_PTRACED	被 ptrace 系统调用监控。
• PF_TRACESYS	正在跟踪。
• PF_FORKNOEXEC	进程刚创建，但还没执行。
• PF_SUPERPRIV	超级用户特权。
• PF_DUMPCORE	dumped core。
• PF_SIGNALED	进程被信号(signal)杀出。
• PF_STARTING	进程正被创建。
• PF_EXITING	进程开始关闭。
• PF_USEDFPU	该进程使用 FPU(SMP only)。
• PF_DTRACE	delayed trace (used on m68k)。

(3) 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 jiffies 计)。优先级可通过系统调用 sys_setpriorty 改变(在 kernel/sys.c 中)。

(4) 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 jiffies 计)。实时进程的优先级可通过系统调用 sys_sched_setscheduler()改变(见 kernel/sched.c)。

(5) long counter;
在轮转法调度时表示进程当前还可运行多久。在进程开始运行是被赋为 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 run_queue 队列选出 counter 值最大的就绪进程并给予 CPU 使用权，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)。

(6) 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(见 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。

2. 信号处理

(1) unsigned long signal;

进程接收到的信号。每位表示一种信号，共 32 种。置位有效。

(2) unsigned long blocked;

进程所能接受信号的位掩码。置位表示屏蔽，复位表示不屏蔽。

(3) 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“慢速型”中断服务程序结束后(IRQ#_interrupt()，参见 9。5 节“启动内核”)。

3. 进程队列指针

(1) struct task_struct *next_task，*prev_task;

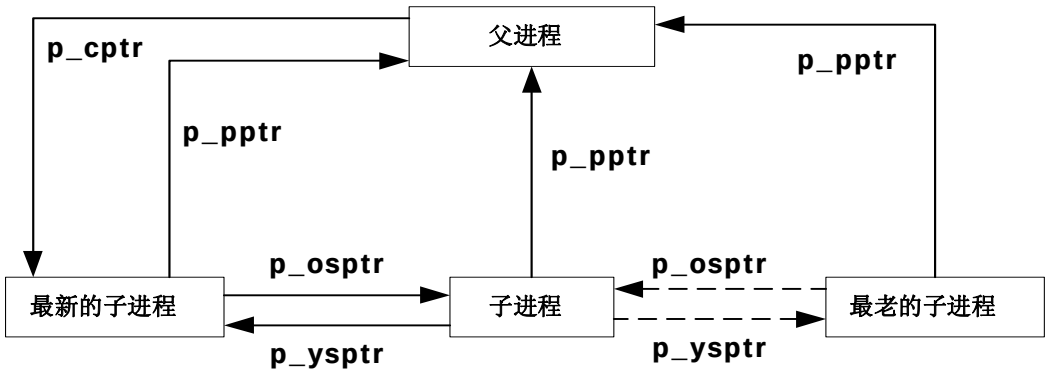
所有进程(以 PCB 的形式)组成一个双向链表。next_task 和就是链表的前后指针。链表的头和尾都是 init_task(即 0 号进程)。

(2) struct task_struct *next_run，*prev_run;

由正在运行或是可以运行的，其进程状态均为 TASK_RUNNING 的进程所组成的一个双向循环链表，即 run_queue 就绪队列。该链表的前后向指针用 next_run 和 prev_run，链表的头和尾都是 init_task(即 0 号进程)。

(3) struct task_struct *p_opptr,*p_pptr;和 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 kenerl/linux/sched.h。它们之间的关系见下图:



父子进程间的关系

4. 进程标识

(1) unsigned short uid，gid;

uid 和 gid 是运行进程的用户标识和用户组标识。

(2) int 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) unsigned short euid，egid;

euid 和 egid 又称为有效的 uid 和 gid。出于系统安全的权限的考虑，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。

(4) unsigned short fsuid，fsgid;

fsuid 和 fsgid 称为文件系统的 uid 和 gid，用于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。

(5) unsigned short suid，sgid;

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。

(6) int pid，pgrp，session;

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

(7) int leader;

是否是 session 的主管，布尔量。

5. 时间数据成员

(1) unsigned long timeout;

用于软件定时，指出进程间隔多久被重新唤醒。采用 tick 为单位。

(2) unsigned long it_real_value，it_real_iner;

用于 itimer(interval timer)软件定时。采用 jiffies 为单位，每个 tick 使 it_real_value 减到 0 时向进程发信号 SIGALRM，并重新置初值。初值由 it_real_incr 保存。具体代码见 kernel/itimer.c 中的函数 it_real_fn()。

(3) struct timer_list real_timer;

一种定时器结构(Linux 共有两种定时器结构，另一种称作 old_timer)。数据结构的定义在 include/linux/timer.h 中，相关操作函数见 kernel/sched.c 中 add_timer()和 del_timer()等。

(4) unsigned long it_virt_value，it_virt_incr;

关于进程用户态执行时间的 itimer 软件定时。采用 jiffies 为单位。进程在用户态运行时，每个 tick 使 it_virt_value 减 1，减到 0 时向进程发信号 SIGVTALRM，并重新置初值。初值由 it_virt_incr 保存。具体代码见 kernel/sched.c 中的函数 do_it_virt()。

(5) unsigned long it_prof_value，it_prof_incr;

同样是 itimer 软件定时。采用 jiffies 为单位。不管进程在用户态或内核态运行，每个 tick 使 it_prof_value 减 1，减到 0 时向进程发信号 SIGPROF，并重新置初值。初值由 it_prof_incr 保存。 具体代码见 kernel/sched.c 中的函数 do_it_prof。

(6) long utime，stime，cutime，cstime，start_time;

以上分别为进程在用户态的运行时间、进程在内核态的运行时间、所有层次子进程在用户态的运行时间总和、所有层次子进程在核心态的运行时间总

和，以及创建该进程的时间。

6. 信号量数据成员

(1) struct sem_undo *semundo;

进程每操作一次信号量，都生成一个对此次操作的 undo 操作，它由 sem_undo 结构描述。这些属于同一进程的 undo 操作组成的链表就由 semundo 属性指示。当进程异常终止时，系统会调用 undo 操作。sem_undo 的成员 semadj 指向一个数据数组，表示各次 undo 的量。结构定义在 include/linux/sem.h。

(2) struct sem_queue *semsleeping;

每一信号量集合对应一个 sem_queue 等待队列(见 include/linux/sem.h)。进程因操作该信号量集合而阻塞时，它被挂到 semsleeping 指示的关于该信号量集合的 sem_queue 队列。反过来，semsleeping。sleeper 指向该进程的 PCB。

7. 进程上下文环境

(1) struct desc_struct *ldt;

进程关于 CPU 段式存储管理的局部描述符表的指针，用于仿真 WINE Windows 的程序。其他情况下取值 NULL，进程的 ldt 就是 arch/i386/traps.c 定义的 default_ldt。

(2) struct thread_struct tss;

任务状态段，其内容与 INTEL CPU 的 TSS 对应，如各种通用寄存器.CPU 调度时，当前运行进程的 TSS 保存到 PCB 的 tss，新选中进程的 tss 内容复制到 CPU 的 TSS。结构定义在 include/linux/tasks.h 中。

(3) unsigned long saved_kernel_stack;

为 MS-DOS 的仿真程序(或叫系统调用 vm86)保存的堆栈指针。

(4) unsigned long kernel_stack_page;

在内核态运行时，每个进程都有一个内核堆栈，其基地址就保存在 kernel_stack_page 中。

8. 文件系统数据成员

(1) struct fs_struct *fs;

fs 保存了进程本身与 VFS 的关系消息，其中 root 指向根目录结点，pwd 指向当前目录结点，umask 给出新建文件的访问模式(可由系统调用 umask 更改)，count 是 Linux 保留的属性，如下页图所示。结构定义在 include/linux/sched.h 中。

(2) struct files_struct *files;

files 包含了进程当前所打开的文件(struct file *fd[NR_OPEN])。在 Linux 中，一个进程最多只能同时打开 NR_OPEN 个文件。而且，前三项分别预先设置为标准输入、标准输出和出错消息输出文件。

(3) int link_count;

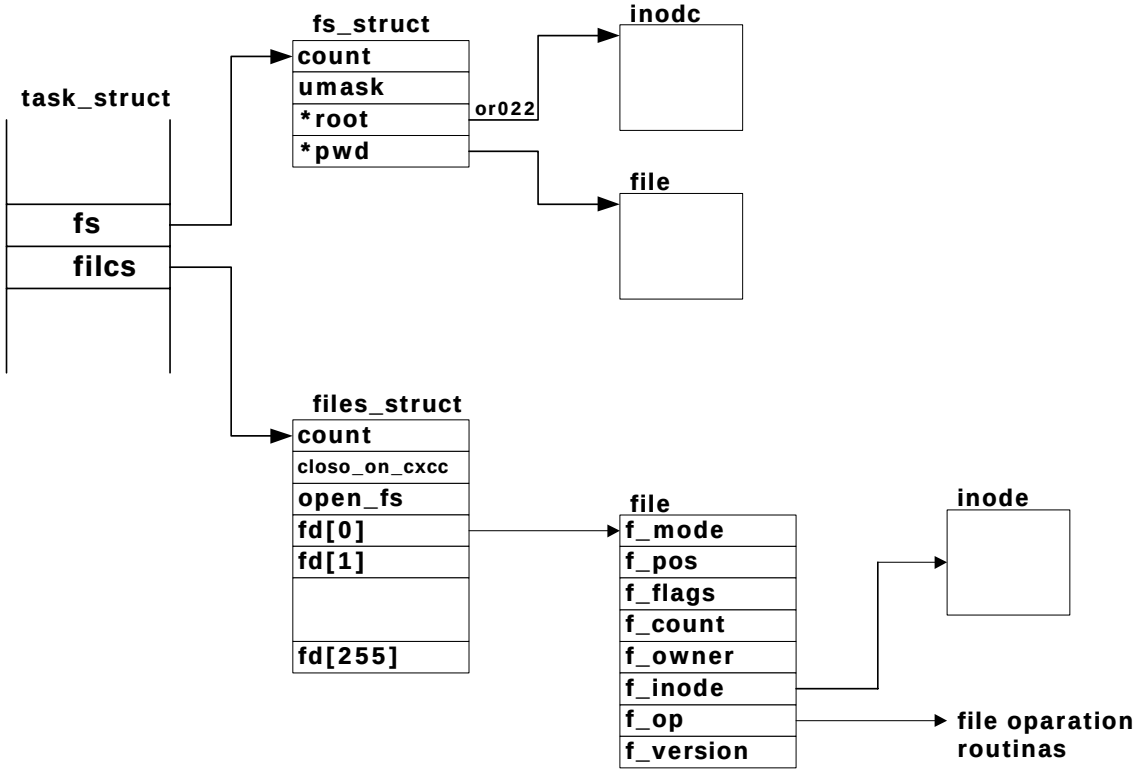
文件链(link)的数目。

9. 内存数据成员

(1) struct mm_struct *mm;

在 linux 中，采用按需分页的策略解决进程的内存需求。task_struct 的数据成员 mm 指向关于存储管理的 mm_struct 结构。其中包含了一个虚存队列 mmap，指向由若干 vm_area_struct 描述的虚存块。同时，为了加快访问速度，mm 中的 mmap_avl 维护了一个 AVL 树。在树中，所有的 vm_area_struct 虚存块均由左指针指向相邻的低虚存块，右指针指向相邻的高虚存块，见下图。 结

构定义在 include/linux/sched.h 中。



文件系统的数据成员

10. 页面管理

(1) int swappable:1;

进程占用的内存页面是否可换出。swappable 为 1 表示可换出。对该标志的复位和置位均在 do_fork()函数中执行(见 kerenl/fork.c)。

(2) unsigned long swap_address;

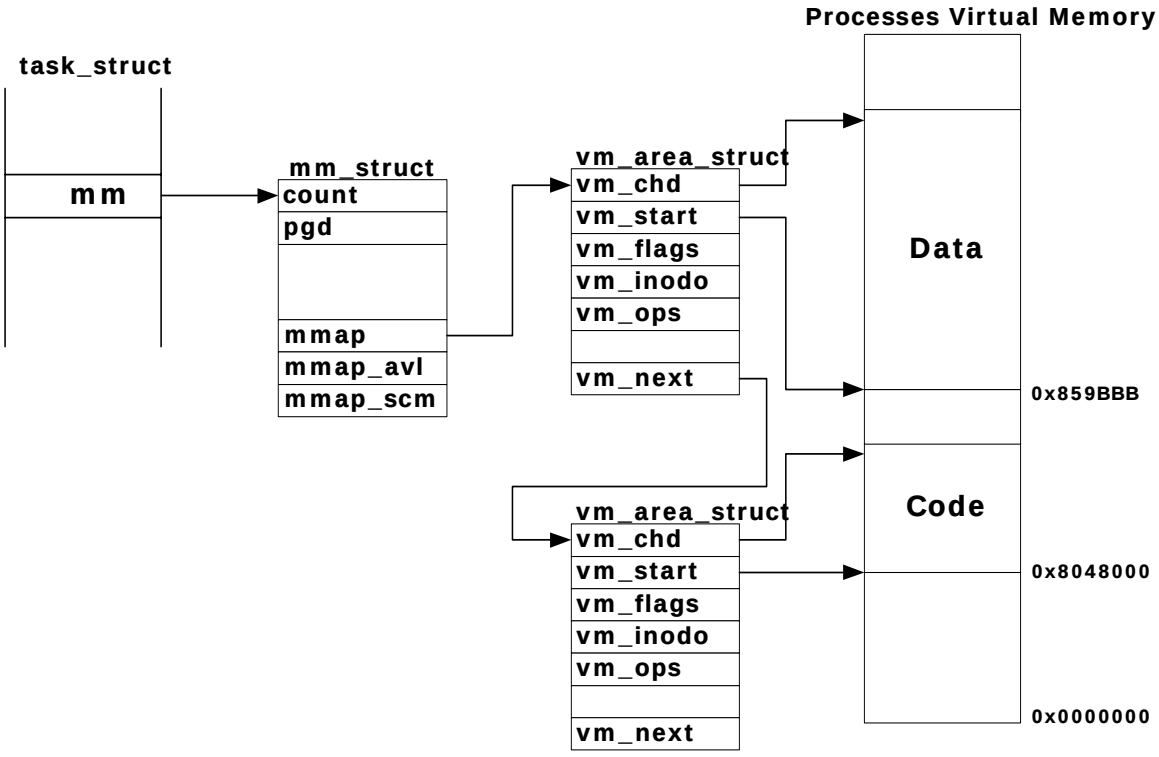
虚存地址比 swap_address 低的进程页面，以前已经换出或已换出过，进程下一次可换出的页面自 swap_address 开始。参见 swap_out_process()和 swap_out_pmd()(见 mm/vmscan.c)。

(3) unsigned long min_flt, maj_flt;

该进程累计的 minor 缺页次数和 major 缺页次数。maj_flt 基本与 min_flt 相同，但计数的范围比后者广(参见 fs/buffer.c 和 mm/page_alloc.c)。min_flt 只在 do_no_page()、do_wp_page()里(见 mm/memory.c)计数新增的可以写操作的页面。

(4) unsigned long nswap;

该进程累计换出的页面数。



虚存数据成员

(5) unsigned long cmin_flt，cmaj_flt，cnswap;

以本进程作为祖先的所有层次子进程的累计换入页面、换出页面计数。

(6) unsigned long old_maj_flt，dec_flt;

(7) unsigned long swap_cnt;

下一次信号最多可换出的页数。

11. 支持对称多处理器方式(SMP)时的数据成员

(1) int processor;

进程正在使用的 CPU。

(2) int last_processor;

进程最后一次使用的 CPU。

(3) int lock_depth;

上下文切换时系统内核锁的深度。

12. 其它数据成员

(1) unsigned short used_math;

是否使用 FPU。

(2) char comm[16];

进程正在运行的可执行文件的文件名。

(3) struct rlimit rlim[RLIM_NLIMITS];

结构 rlimit 用于资源管理，定义在 linux/include/linux/resource.h 中，成员共有两项:rlim_cur 是资源的当前最大数目;rlim_max 是资源可有的最大数目。在 i386 环境中，受控资源共有 RLIM_NLIMITS 项，即 10 项，定义在 linux/include/asm/resource.h 中，见下表:

(4) int errno;

最后一次出错的系统调用的错误号，0 表示无错误。系统调用返回时，全程量也拥有该错误号。

(5) long debugreg[8];

保存 INTEL CPU 调试寄存器的值，在 ptrace 系统调用中使用。

(6) struct exec_domain *exec_domain;

Linux 可以运行由 80386 平台其它 UNIX 操作系统生成的符合 iBCS2 标准的程序。关于此类程序与 Linux 程序差异的消息就由 exec_domain 结构保存。

(7) unsigned long personality;

Linux 可以运行由 80386 平台其它 UNIX 操作系统生成的符合 iBCS2 标准的程序。Personality 进一步描述进程执行的程序属于何种 UNIX 平台的“个性”信息。通常有 PER_Linux、PER_Linux_32BIT、PER_Linux_EM86、PER_SVR3、PER_SCOSVR3、PER_WYSEV386、PER_ISCR4、PER_BSD、PER_XENIX 和 PER_MASK 等，参见 include/linux/personality.h。

(8) struct linux_binfmt *binfmt;

指向进程所属的全局执行文件格式结构，共有 a。out、script、elf 和 java 等四种。结构定义在 include/linux/binfmts.h 中(core_dump、load_shlib(fd)、load_binary、use_count)。

(9) int exit_code，exit_signal;

引起进程退出的返回代码 exit_code，引起错误的信号名 exit_signal。

(10) int dumpable:1;

布尔量，表示出错时是否可以进行 memory dump。

(11) int did_exec:1;

按 POSIX 要求设计的布尔量，区分进程是正在执行老程序代码，还是在执行 execve 装入的新代码。

(12) int tty_old_pgrp;

进程显示终端所在的组标识。

(13) struct tty_struct *tty;

指向进程所在的显示终端的信息。如果进程不需要显示终端，如 0 号进程，则该指针为空。结构定义在 include/linux/tty.h 中。

(14) struct wait_queue *wait_chldexit;

在进程结束时，或发出系统调用 wait4 后，为了等待子进程的结束，而将自己(父进程)睡眠在该队列上。结构定义在 include/linux/wait.h 中。

13. 进程队列的全局变量

(1) current;

当前正在运行的进程的指针，在 SMP 中则指向 CPU 组中正被调度的 CPU 的当前进程:

```
#define current(0+current_set[smp_processor_id()])/*sched.h*/
struct task_struct *current_set[NR_CPUS];
```

(2) struct task_struct init_task;

即 0 号进程的 PCB，是进程的“根”，始终保持初值 INIT_TASK。

(3) struct task_struct *task[NR_TASKS];

进程队列数组，规定系统可同时运行的最大进程数(见 kernel/sched.c)。NR_TASKS 定义在 include/linux/tasks.h 中，值为 512。每个进程占一个数组元素(元素的下标不一定就是进程的 pid)，task[0]必须指向 init_task(0 号进程)。可

以通过 task[]数组遍历所有进程的 PCB。但 Linux 也提供一个宏定义 for_each_task()(见 include/linux/sched.h),它通过 next_task 遍历所有进程的 PCB:

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

(4) unsigned long volatile jiffies;

Linux 的基准时间(见 kernal/sched.c)。系统初始化时清 0，以后每隔 10ms 由时钟中断服务程序 do_timer()增 1。

(5) int need_resched;

重新调度标志位(见 kernal/sched.c)。当需要 Linux 调度时置位。在系统调用返回前(或者其它情形下)，判断该标志是否置位。置位的话，马上调用 schedule 进行 CPU 调度。

(6) unsigned long intr_count;

记录中断服务程序的嵌套层数(见 kernal/softirq.c)。正常运行时，intr_count 为 0。当处理硬件中断、执行任务队列中的任务或者执行 bottom half 队列中的任务时，intr_count 非 0。这时，内核禁止某些操作，例如不允许重新调度。

3.1.3 调度和时间片

对 CPU 访问的裁决过程被称为调度（Scheduling）。良好的调度决策要尊重用户赋予的优先级，这可以建立一种所有进程都在同时运行的十分逼真的假象。糟糕的调度决策会使操作系统变得沉闷缓慢。这是 Linux 调度程序必须经过高度优化的一个原因。

从概念上来说，调度程序把时间分为小片断，并根据一定的原则把这些片断分配给进程。时间的这些小片断称为时间片。

3.1.4 实时进程与非实时进程

Linux 把执行的任务比较紧迫的进程定义为实时进程，这些进程通常会比一般进程先得到机会运行。根据它们的调度策略（后面还会讲到）实时进程分为两种：一种执行的任务比较短小，因此采用先进先出的策略，即当前进程执行完才给其它进程运行的机会；另一种则采用时间片轮转法，让这样的实时进程周期性地交替运行。

非实时进程是相对实时进程而言的,Linux 系统把执行的任务不怎么紧迫的进程统称为非实时进程。Linux 通常对这类进程采用优先权算法进行调度,即定期检查进程的优先级,优先级高的进程具有较高的优先权,将会优先获得 CPU 时间而运行。

3.1.5 Linux 进程优先级

优先级是一些简单的整数，它代表了为决定应该允许哪一个进程使用 CPU 的资源时判断方便而赋予进程的权值——优先级越高，它得到 CPU 时间的机会也就越大。

在 Linux 中，非实时进程有两种优先级，一种是静态优先级，另一种是动态优先级。实时进程又增加了第三种优先级，实时优先级。

1. **静态优先级**（priority）——被称为“静态”是因为它不随时间而改变，

只能由用户进行修改。它指明了在被迫和其它进程竞争 CPU 之前该进程所应该被允许的时间片的最大值（20）。

2. **动态优先级（counter）**——counter 即系统为每个进程运行而分配的时间片，Linux 兼用它来表示进程的动态优先级。只要进程拥有 CPU，它就随着时间不断减小；当它为 0 时，标记进程重新调度。它指明了在当前时间片中所剩余的时间量（最初为 20）。
3. **实时优先级(rt_priority)**——值为 1000。Linux 把实时优先级与 counter 值相加作为实时进程的优先权值。较高权值的进程总是优先于较低权值的进程，如果一个进程不是实时进程，其优先权就远小于 1000，所以实时进程总是优先。

3.1.6 Linux 进程系统的特点

Linux 是一个多进程系统，它具有以下特点：

1. 并行化

一件复杂的事件是可以分解成若干个简单事件来解决的，这在程序员的大脑中早就形成了这种概念，首先将问题分解成一个个小问题，将小问题再细分，最后在一个合适的规模上做成一个函数。 在软件工程中也是这么说的。如果我们以图的方式来思考，一些小问题的计算是可以互不干扰的，可以同时处理，而在关键点则需要统一在一个地方来处理，这样程序的运行就是并行的，至少从人的时间观念上来说是这样的。 而每个小问题的计算又是较简单的。

2. 简单有序

程序员为每个进程设计好相应的功能，并通过一定的通讯机制将它们有机地结合在一起，对每个进程的设计是简单的，只在总控部分小心应付，就可完成整个程序的施工。

3. 互不干扰

这个特点是操作系统的特点，各个进程是独立的，不会串位。

4. 事务化

比如在一个数据电话查询系统中，将程序设计成一个进程只处理一次查询即可，即完成一个事务。当电话查询开始时，产生这样一个进程对付这次查询；另一个电话进来时，主控程序又产生一个这样的进程对付，每个进程完成查询任务后消失. 这样的编程多简单，只要做一次查询的程序就可以了。

一个多进程的操作系统，进程是分离的任务，拥有各自的权利和责任。如果一个进程崩溃，它不应该让系统的另一个进程崩溃。每一个独立的进程运行在自己的虚拟地址空间，除了通过安全的核心管理的机制之外无法影响其他的进程。

在一个进程的生命周期中，进程会使用许多系统资源。比如利用系统的 CPU 执行它的指令，用系统的物理内存来存储它和它的数据。它会打开和使用文件系统中的文件，会直接或者间接使用系统的物理设备。如果一个进程独占了系统的大部分物理内存和 CPU，对于其他进程就是不公平的。所以 Linux 必须跟踪进程本身和它使用的系统资源以便公平地管理系统中的进程。

系统最宝贵的资源就是 CPU。通常系统只有一个 CPU。Linux 作为一个多进程的操作系统，它的目标就是让进程在系统的 CPU 上运行，充分利用 CPU。如果进程数多于 CPU（一般情况都是这样），其他的进程就必须等到 CPU 被释放才能运行。多进程的思想就是：一个进程一直运行，直到它必须等待，通常是等待一些系统资源(也包括时间片)，等拥有了资源，它才可以继续运行。在一个单进程的系统中，比如 DOS，CPU 被简单地设为空闲，这样等待资源的时间就会被浪费。而在一个多进程的系统中，同一时刻许多进程在内存中，当一个进程必须等待时，操作系统将 CPU 从这个进程切换到另一个更需要的进程。

在 Linux 中，每个进程用一个 task_struct 的结构来表示，用来管理系统中的进程。Task 向量表是指向系统中每一个 task_struct 结构的指针的数组。这意味着系统中的最大进程数受到 Task 向量表的限制，缺省是 512。这个表让 Linux 可以查到系统中的所有的进程。操作系统初始化后，建立了第一个 task_struct 结构 INIT_TASK。当新的进程创建时，从系统内存中分配一个新的 task_struct，并增加到 Task 向量表中。为了更容易查找，用 current 指针指向当前运行的进程。

task_struct 结构中有关于进程调度的两个重要的数据项：

```
struct task_struct {
    .............
    volatile  long  state; /* -1 unrunnable , 0 runnable , >0 stopped
*/
    unsigned  long  flags; /* per process flags, defined below */
    .............
 };
```

每个在 Task 向量表中登记的进程都有相应的进程状态和进程标志，它们是进行进程调度的重要依据。进程在执行了相应的进程调度操作后，会由于某些原因改变自身的状态和标志，也就是改变 state 和 flags 这两个数据项。进程的状态不同、标志位不同对应了进程可以执行不同操作。

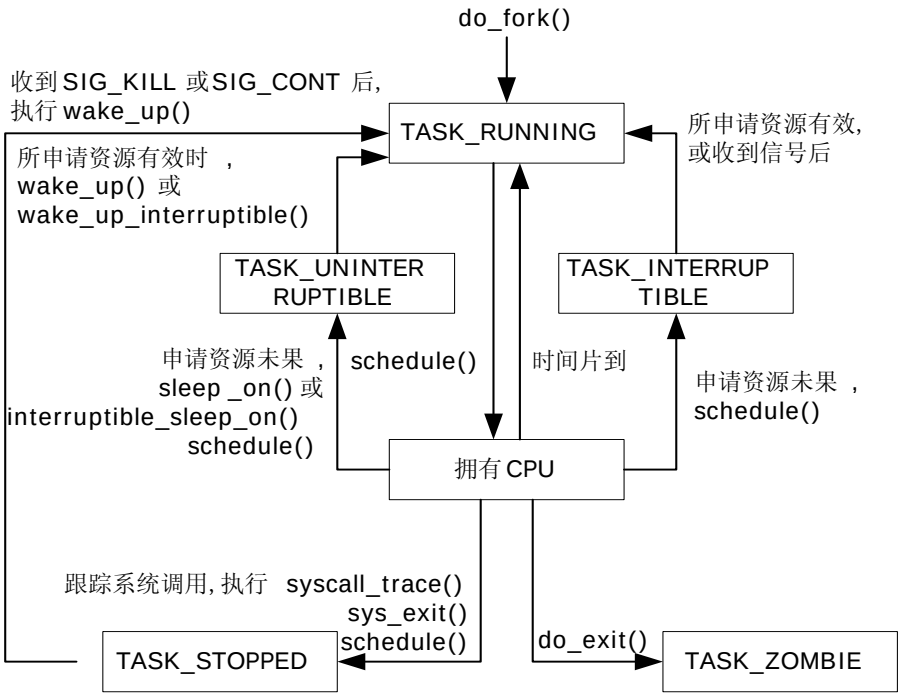
Linux 在 sched.h 中定义了六种状态，十一种标志（参见 3.1.2task_struct 结构描述）：

```
#define TASK_RUNNING            0
#define TASK_INTERRUPTIBLE      1
#define TASK_UNINTERRUPTIBLE    2
#define TASK_ZOMBIE             4
#define TASK_STOPPED            8
#define TASK_SWAPPING           16
```

它们的含义分别是：

1. TASK_RUNNING：正在运行的进程（是系统的当前进程）或准备运行的进程（在Running队列中，等待被安排到系统的CPU）。处于该状态的进程实际参与了进程调度。
2. TASK_INTERRUPTIBLE：处于等待队列中的进程，待资源有效时唤醒，也可由其它进程被信号中断、唤醒后进入就绪状态。
3. TASK_UNINTERRUPTIBLE：处于等待队列中的进程，直接等待硬件条件，待资源有效时唤醒，不可由其它进程通过信号中断、唤醒。
4. TASK_ZOMBIE：终止的进程，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Task向量表中仍有一个task_struct数据结构项。它不进行任何调度或状态转换，等待父进程将它彻底释放。
5. TASK_STOPPED：进程被暂停，通过其它进程的信号才能唤醒。正在调试的进程可以在该停止状态。
6. TASK_SWAPPING：进程页面被兑换出内存的进程。这个状态基本上没有用到，只有在sched.c的count_active_tasks（）函数中判断处于该种状态的进程也属于active的进程，但没有对该状态的赋值。

进程的状态随着进程的调度发生改变，下图显示了一个进程的状态转换关系：



Linux 进程的状态转换

3.2 进程的调度

作为多进程的系统，Linux 系统必须担负起调度进程的责任，不断地切换进程，以使 CPU 得到最大化的利用，提高系统的效率。

3.2.1 Linux 进程调度的策略

进程调度的策略主要考虑以下几个原则：

（1） 高效　使处理器的利用率最高，空闲最小；

（2） 公平　使每一个申请处理器的进程都得到合理的处理器时间；

（3） 周转时间短　使用户提交任务后得到结果的时间尽可能短；

（4） 吞吐量大　使单位时间内处理的任务数量尽可能多；

（5） 响应时间短　使对每个用户响应的时间尽可能短。

在 3.1 节中我们已经提到过，Linux 有两种类型的进程：实时和非实时（普通）。实时进程比所有其它进程的优先级高。如果有一个实时的进程准备运行，那么它总是先被运行。实时进程有两种策略：时间片轮转或先进先出（round robin and first in first out）。在时间片轮转的调度策略下，每一个实时进程依次运行，而在先进先出的策略下，每一个可以运行的进程按照它在调度队列中的顺序运行，这个顺序不会改变。对于非实时进程，Linux 永远选择优先级最高的进程来运行。

3.2.2 Linux 进程的调度算法

1. **时间片轮转调度算法**

   用于实时进程。系统使每个进程依次地按时间片轮流执行的方式。

2. **优先权调度算法**

   用于非实时进程。系统选择运行队列中优先级最高的进程运行。Linux 采用抢占式的优级算法，即系统中当前运行的进程永远是可运行进程中优先权最高的那个。

3. **FIFO(先进先出) 调度算法**

   前面已经提到过，实时进程按调度策略分为两种。采用 FIFO 的实时进程必须是运行时间较短的进程，因为这种进程一旦获得 CPU 就只有等到它运行完或因等待资源主动放弃 CPU 时其它进程才能获得运行机会。

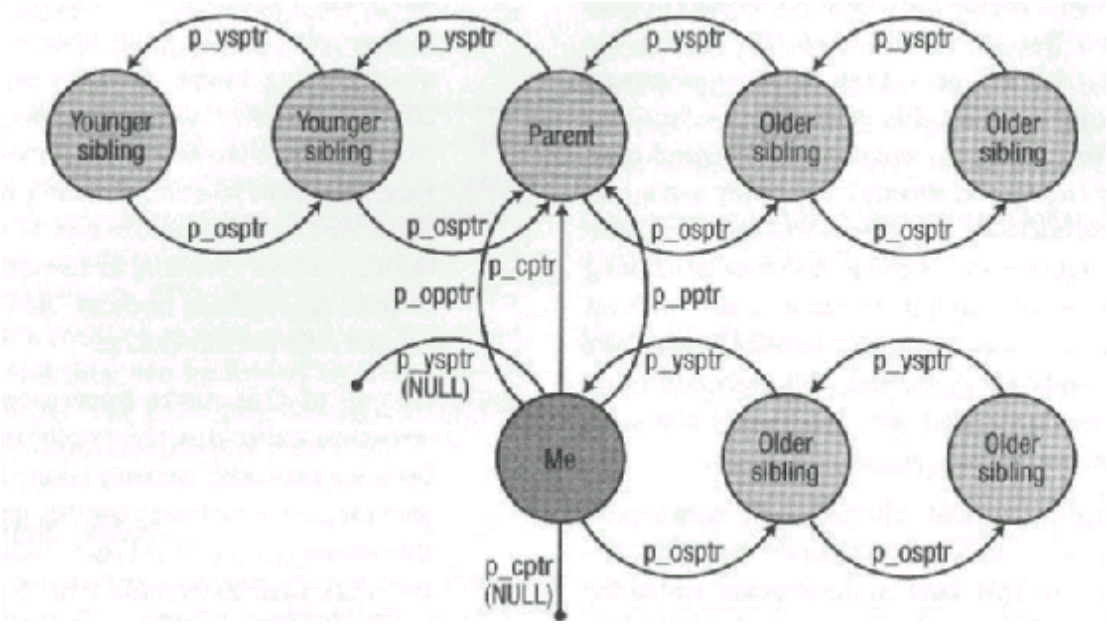
3.2.3 Linux 进程的调度时机

时机如下：

1. 进程状态转换时： 如进程终止，睡眠等；
2. 可运行队列中增加新的进程时；
3. 当前进程的时间片耗尽时；
4. 进程从系统调用返回到用户态时（见第 6 章系统调用返回部分的代码注释）；
5. 内核处理完中断后，进程返回到用户态。

3.2.4 Linux 进程的队列

存在于系统中的进程并不是孤立的，它们是有联系的。依据进程的创建关系，系统中的进程组成了一个庞大的网络（类似于家谱）。它们的关系可以用下图表示：



通过这种亲缘关系网，我们可以从一个进程遍历到任何一个进程。

运行队列：

Linux 系统为处于就绪态的进程单独组建了一个队列，只有在这个队列中的进程才有机会获得 CPU。

对运行队列的操作（注释见第 6 章）：

由于 linux 的 task_struct 结构有自己的一套结构指针，所以 Linux 为进程运行队列单独设计了一套用于队列操作的函数。

（1） 使用函数 add_to_runqueue（）向运行队列插入一个新进程 task_struct 结构。

（2） 使用函数 del_from_runqueue（）从运行队列摘除一个进程 task_struct 结构。

（3） 使用函数 move_last_runqueue（）将当前 task_struct 结构移到运行队列队尾。

（4） 使用函数 move_first_runqueue（）将当前 task_struct 结构移到运行队列队头。

等待队列：

与运行队列相对应，Linux 系统也为处于睡眠态的进程组建了一个队列。

对队列的基本操作：

对队列的操作无外乎初始化、添加、删除等。为了实现方便，Linux 内核中采用了一套抽象的、通用的、一般的、可以用到各种不同数据结构的队列操作函数。上面提到的进程运行队列和等待队列实际上也是应用这套抽象函数来完成具体队列操作的。由于这些函数(位置：include/linux/list.h)都很简单，我们直接在这里讨论，而不再在第 5、6 章另付流程和注释。

（1）Linux 单独设置一个 list_head 结构，并把它放在各种不同的结构（称之为“宿主”）中作为将那个结构队列连接指针。list_head 结构定义如下：

```
struct list_head { struct list_head *next,*prev; }; /* prev 指向队列的前一个元素，* next 指向
                                                    队列的后一个元素*/
```

（2）Linux 通过一个宏对宿主数据结构内部的每个 list_head 数据结构进行初始化：

```
#define INIT_LIST_HEAD(ptr) do { \
    (ptr)->next=(ptr);(ptr)->prev=(ptr);\
}while(0)
```

参数 ptr 为需要初始化的 list_head 结构。初始化以后两个指针都指向该 list_head 结构自身。

（3）Linux 使用 list_add( )将一个结构通过其“队列头”list 链入一个队列，该函数定义如下：

```
static __inline__ void list_add(struct list_head *new,struct list_head *head)
    {
     __list_add(new,head,head->next);   /*调用__list_add( )来完成操作*/
   }
static __inline__ void __list_add(struct list_head * new,   /* new 指向欲链入队列的宿主数
                                                              * 结构内部的 list_head 数据结构*/
    struct list_head * prev,        /* prev 指向链入点，它可以是个独立的、真正意义上的队
                                     * 列头，也可以在另一个宿主数据结构内部*/
    struct list_head * next)
{
    next->prev = new;
    new->next = next;
    new->prev = prev;
    prev->next = new;
}
```

(5) Linux 使用 list_del( )完成脱链操作，该函数定义如下：

```
static __inline__ void list_del(struct list_head *entry)
{
    __list_del(entry->prev, entry->next);                    /*调用__list_del( )来完成操作*/
}
static __inline__ void __list_del(struct list_head * prev,
                      struct list_head * next)
{   /* 把被删除元素的前后元素直接钩链，以达到将当前元素从队列摘除的目的*/
    next->prev = prev;
    prev->next = next;
}
```

(6) Linux 通过宏 list_entry 来获取通过 list_head 结构找到的宿主结构指针：

```
#define list_entry(ptr, type, member) \
 ((type *)((char *)(ptr)-(unsigned long)(&((type *)0)->member)))
```

3.2.5 Linux 进程调度的全过程

1. 转入 schedule 处理函数前的过程

根据 3.2.3 Linux 进程调度时机，Linux 可以以五种方式转入 schedule 处理函数。

（1） 进程状态转换时

当进程要调用 sleep（）或 exit（）等函数使进程状态发生改变时，这些函数会主动调用 schedule（）转入进程调度。

（2） 通过时钟中断

Linux 初始化时，设定系统定时器的周期为 10 毫秒。当时钟中断发生时，时钟中断服务程序 timer_interrupt 立即调用时钟处理函数 do_timer( )。该函数会调用 mark_bh(TIMER_BH),将 bh_active 标志的 TIMER_BH 位置 1。接着，Linux 会在时钟中断服务程序中通过代码片段

```
if (bh_active& bh_mask)
   { intr_count=1;
     do_bottom_half();
     intr_count=0;
    }
```

来判断此时是否有 bottom half 服务要处理，若有则执行 do_bottom_half()。该函数会调用时钟响应函数 timer_bh()，分别由 update_times()、run_old_timers()和 run_timer_list()检查、执行定时服务。update_times()又调用 update_process_times()函数调整进程的时间片，当时间片小于 0 时，need_resched（需要重调度）标志会被置位。

当时钟中断处理完毕后，系统会返回到入口 ret_from_intr,再跳转到 ret_with_reschedule，判断 need_resched 标志是否置位，若是则转入执行 schedule()。

（3） 内核处理完中断服务，返回到用户态时

同（2）中时钟中断处理完毕后的返回处理一致，系统会返回到入口 ret_from_intr,再跳转到 ret_with_reschedule,判断 need_resched 标志是否置位，若是则转入执行 schedule()。

（4） 进程从系统调用返回到用户态时

系统调用的结束后，系统要通过入口 ret_from_sys_call 从核心态转回用户态。在 ret_from_sys_call 中，系统会先判断是否有 half bottom 服务请求，若没有，则跳转到 ret_with_reschedule，判断 need_resched 标志是否置位，若是则转入执行 schedule()；若有，则执行 do_bottom_half()，然后在转回 ret_with_reschedule，判断 need_resched 标志是否置位，若是则转入执行 schedule()。

（5） 运行队列增加进程时

当需要向运行队列增加一个进程是，系统会调用 add_to_runqueue(),在此过程中，比较要加入的进程和正在运行的进程的 counter 值，若要加入进程的 counter 值大于正在运行的进程的 counter 值加 3,将 need_resched 标志置一。当处理时钟中断服务后返回时，转入 schedule()。

2. 执行 schedule( ),完成进程调度，切换进程

在 schedule 中，先检查是否是中断服务程序调用了 schedule（这是不允许的），如果是则退出 schedule。若不是，则检查是否有 bottom half 服务请

求，若有则执行 do_half_bottom。然后在判断当前进程是否是采用时间片轮转调度法的实时进程，若是则重新给它分配时间片并把它移到运行队列尾部。接着根据当前进程的状态对当前进程作相关处理。接下来，便是调度正文。通过函数 goodness( )遍历运行队列中所有的进程，选择权值最大的进程作为下一个运行的进程。若运行队列中所有的进程的时间片都耗尽了，则要给系统中所有的进程重新分配时间片。最后通过宏 switch_to()切换堆栈，从而达到从当前进程切换到选中的进程的目的。调度结束。

3.2.6 schedule( )及其调用函数

1. schedule( )

schedule( ) 是 Linux 系统实现进程调度的函数，有些文章也把它称为调度器。schedule 按照不同的调度策略对实时进程和非实时进程进行不同的调度处理。

它的简单流程如下（详细见第 5、6 章）：

（1） 检查是否是中断服务程序调用了 schedule( )(这是不允许的)。若是，则退出 shcedule( )。
（2） 若有 half bottom 服务请求则处理 half bottom。
（3） 保存当前 CPU 调度进程的数据区;对运行队列加锁。
（4） 如果采用轮转法进行调度,则要重新检查一下当前进程的 counter 是否为零。若是则重新分配时间片(counter),并将其挂到队列尾部。
（5） 检测进程状态:对 need_resched 重新置 0。
（6） 搜索运行队列,计算出每一个进程的 goodness 并与当前的 goodness 相比较,goodness 值最高的进程将获取 CPU。
（7） 若整个运动队列中的进程的时间片耗尽，重新分配时间片。

对于非实时进程:

运行队列中的进程由于时间片已耗尽,重新赋值后 p->counter=p->pripority=20; 不在运行队列的进程通过 p->counter>>1 将原有时间片减半再加 p->pripority,以此来提高其优先级,使之能够在转入就绪态时有较高的竞争力。但这种竞争力的提高不是无限制的,通过 p->counter>>1 使 p->counter 永远不会超过两倍 p->pripority 的值,这样就避免了由于 p->counter 无限增长导致其优先级高于实时进程的情况。

对于实时进程:

由于实时进程不采用 counter 值作为优先级,所有对实时进程的 counter 的赋值操作无关紧要.

时间片分配完成后转向（5）。

（8） 切换至新的进程。其中要调用 switch_mm 重载 LDT（详见第 5、6 章）。

2. goodness( )

goodness( ) 用来计算进程的权值。
流程如下（详细见第 5、6 章）：

（1） 根据 policy 区分实时进程和普通进程。
（2） 若为实时进程返回权值 1000 + p->rt_priority。
（3） 若非实时进程，将进程剩余时间片加 priority 作为权值返回。如果进程是内核线程或用户空间与当前进程相同，因而不必切换用户空间，则给予权值额外加一的"优惠"。

3.3 时钟中断

进程时间片是在时钟中断时更新的。每次通过时钟函数调用 bottom half 处理程序，递减进程的时间片，当时间片小于 0 时，置 need_resched 标志从而引起进程调度。

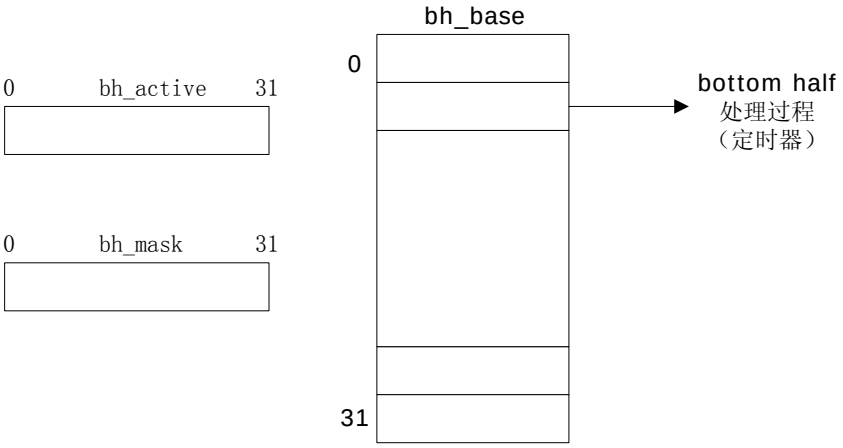
3.3.1 时钟

所有的操作系统都必须准确的得到当前时间，所以现代 PC 包含一个特殊的外设称为实时时钟(RTC)。它提供了两种服务：可靠的日期和时间以及精确的时间间隔。RTC 有其自身的电池这样即使 PC 掉电时它照样可以工作，这就是 PC 总是"知道"正确时间和日期的原因。而时间间隔定时器使得操作系统能进行准确的调度工作。

3.3.2 实时时钟(RTC)中断

实时时钟(RTC)每隔千分之一秒(10ms)就产生一次中断。这个周期性中断被称为系统时钟滴答，它象节拍器一样来组织系统任务。

实时时钟的周期性定时器被固定连接到中断控制器 PIC1 的引脚 0 上，它的中断处理例程是 timer_interrupt(){arch/i386/kernel/time.c}。时钟中断发生时，服务例程 timer_interrupt 马上调用 do_timer（sched.c），后者把核心的 bottom half 中的 TIMER 处理过程标记为活动，使得核心的定时器队列机制能够在核心从时钟中断处理返回前（调用 bottom half 例程时）被驱动。



bottom half处理机制数据结构

3.3.3 bottom half 处理机制

当中断发生时处理器将停止当前的工作，操作系统将中断发送到相应的设备驱动上去。由于此时系统中其他程序都不能运行，所以设备驱动中的中断处理过程不宜过长。有些任务最好稍后执行。Linux 的 bottom half 处理机制可以让设备驱动和 Linux 核心其他部分将这些工作进行排序以延迟执行。上图给出了一个与 bottom half 的处理相关的核心数据结构。

系统中最多可以有 32 个不同的 bottom half 处理过程；bh_base 是指向这些过程入口的指针数组。而 bh_active 和 bh_mask 用来表示那些处理过程已经安装以及那些处于活动状态。如果 bh_mask 的第 N 位置位则表示 bh_base 的 第 N 个元素包含 bottom half 处理例程。如果 bh_active 的第 N 位置位则表示第 N 个 bottom half 处理例程可在调度器认为合适的时刻调用。这些索引被定义成静态的；定时器的 bottom half 处理例程具有最高优先级（索引值为 0），控制台 bottom half 处理例程其次（索引值为 1）。典型的 bottom half 处理例程包含与之相连的任务链表。

有些核心 bottom half 处理过程是设备相关的，但有些更加具有通用性：

(1) TIMER　每次系统的周期性时钟中断发生时此过程被标记为活动，它被用来驱动核心的定时器队列机制。

(2) CONSOLE　此过程被用来处理进程控制台消息。

(3) TQUEUE　此过程被用来处理进程 tty 消息。

(4) NET　此过程被用来做通用网络处理。

(5) IMMEDIATE 这是被几个设备驱动用来将任务排队成稍后执行的通用过程。

当设备驱动或者核心中其他部分需要调度某些工作延迟完成时，它们将把这些任务加入到相应的系统任务队列中去，然后设置 bh_active 中的某些位，也就使核心获知它需要调用某个 bottom half 处理过程。如果设备驱动将某个任务加入到了 immediate 队列并希望 bottom half 处理过程运行和处理它,可将第 8 位置 1。每次系统调用结束返回前都要检查 bh_active。如果有位被置 1 则调用处于活动状态的 bottom half 处理过程。检查的顺序是从 0 位开始直到第 31 位。

每次调用底层处理过程时 bh_active 中的对应位将被清除。bh_active 是一个瞬态变量,在相应的 bottom half 处理过程无须工作的状态下避免了对此处理过程的调

用。

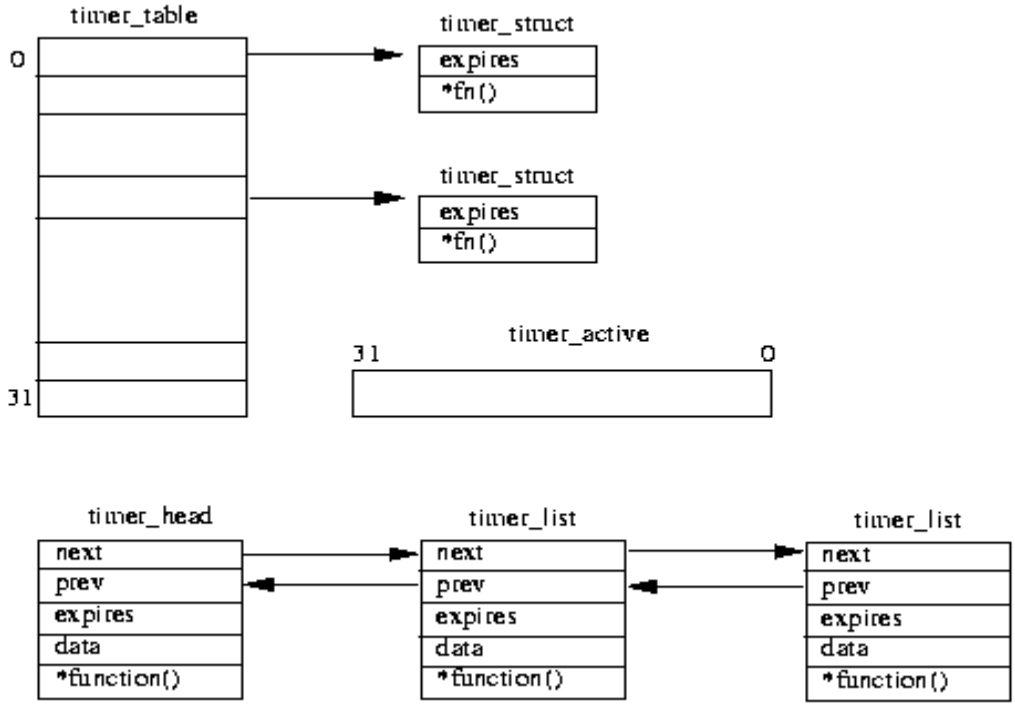
3.3.4 系统定时器

1. 定时器（TIMER）的原理

操作系统应该能够在将来某个时刻准时调度某个任务。所以需要一种能保证任务较准时被调度运行的机制。希望支持某种操作系统的微处理器必须具有一个可周期性中断它的可编程间隔定时器。这个周期性中断被称为系统时钟滴答，它象节拍器一样来组织系统任务。

Linux 的时钟观念很简单：它表示系统启动后的以时钟滴答记数的时间。所有的系统时间都基于这种量度，它和系统中的一个全局变量 jiffies 的名称相同。

Linux 包含两种类型的系统定时器，它们都可以把将在某个系统时间上被调用的例程进行排列，但是它们的实现稍有区别。这两种机制如下图所示。



第一种是老的定时器机制，它包含指向 timer_struct 结构的 32 位指针的静态数组以及当前活动定时器的屏蔽码 ：time_active。 此定时器表中的位置是静态定义的（类似 bottom half 的处理表 bh_base），其入口在系统初始化时被加入到表中。 第二种是相对较新的定时器，它使用一个以升序的到期时间排列的 timer_list 结构链表。

这两种方法都使用 jiffies 作为终结时间，这样，希望运行 5 秒的定时器将 5 秒时间转换成 jiffies 的单位并且将它和以 jiffies 记数的当前系统时间相加从而得到定时器的终结时间。在每个系统时钟滴答时,定时器的 bottom half 处理过程被标记成活动状态以便调度程序在下次运行时能进行系统定时器队列的处理。定时器的 bottom half 处理过程会处理两种类型的系统定时器。老的系统定时器将检查

timer_active 位是否置位。

如果活动定时器已经到期则其定时器例程将被调用，同时 bottom half 中相应的活动位也被清除。新定时器位于 timer_list 结构链表中的入口也将受到检查。每个过期定时器将从链表中清除，同时它的例程将被调用。新定时器机制的优点之一是能传递一个参数给定时器例程。

2．系统定时器队列

系统定时器的终结时间指的是定时器到期时以 jiffies 记数的系统时间。运行时间指的是定时器的终结时间减去以 jiffies 记数的当前系统时间。

周期性时钟中断发生时，其服务例程 timer_interrupt 马上调用 do_timer（sched.c），后者把核心的 bottom half 中的 TIMER 处理过程标记为活动，使得核心在从时钟中断处理返回前能够调用定时器 bottom half 处理例程。

3. timer_list 结构定时器结构

本节仅对 timer_list 作结构上的分析，为了说明方便，具体的定时器机制实现和控制管理相关的函数将在后面介绍系统调用 nanosleep( )中一并解释。

（1） 主要数据结构的定义(位置在/linux/kernelsched.c)

```
unsigned long volatile jiffies=0;
static unsigned long timer_jiffies = 0;

#define TVN_BITS 6
#define TVR_BITS 8
#define TVN_SIZE (1 << TVN_BITS)
#define TVR_SIZE (1 << TVR_BITS)
#define TVN_MASK (TVN_SIZE - 1)
#define TVR_MASK (TVR_SIZE - 1)

struct timer_vec {
        int index;
        struct timer_list *vec[TVN_SIZE];
};
struct timer_vec_root {
        int index;
        struct timer_list *vec[TVR_SIZE];
};

static struct timer_vec tv5 = { 0 };
static struct timer_vec tv4 = { 0 };
static struct timer_vec tv3 = { 0 };
static struct timer_vec tv2 = { 0 };
static struct timer_vec_root tv1 = { 0 };
```

```
static struct timer_vec * const tvecs[] = {
    (struct timer_vec *)&tv1, &tv2, &tv3, &tv4, &tv5
};
```

Linux 借助对以 jiffies 记数的系统时间在 0~7、0~13、0~19、0~25 位片段上的进位判断来操纵间隔定时器在系统的定时器队列上的位置。我们可以把 jiffies 值和定时器的终结时间(expires)按位描述为：

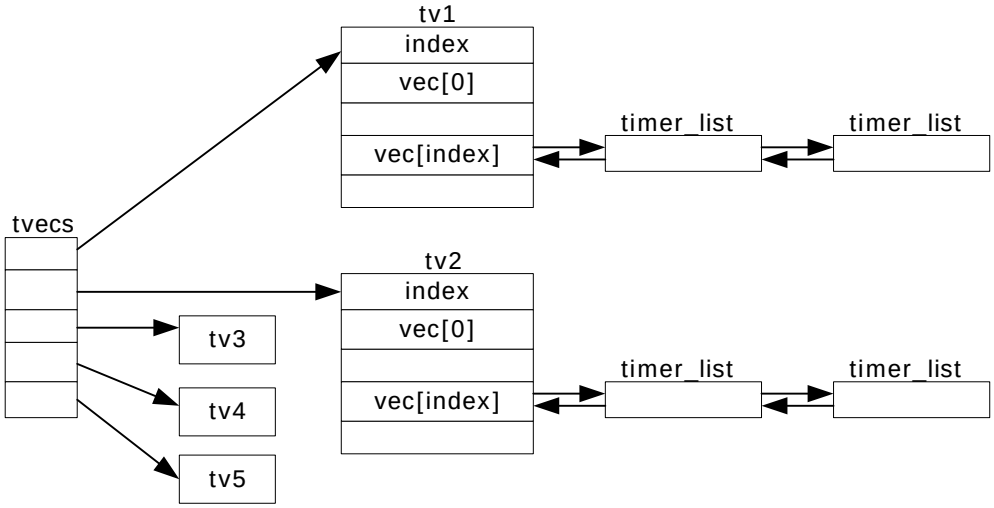
jiffies（32 位）: 0 ..... 7**8**.... 13**14** .... 19**20** .... 25**26**.... **31**

expires（32 位）: 0 ..... 7**8**.... 13**14**.... 19**20**.... 25**26**.... **31**

（2） tvecs：系统定时器队列

Linux 系统在调用定时器 bottom half 处理过程时，在其中的 run_timer_list()（kernel/sched.c）中处理 timer_list 结构的系统定时器队列。

tvecs 结构如下:



time_list型定时器服务体系结构

3.4 系统调用 nanosleep( ) 和 pause( )

出于种种原因，运行中的进程常常需要主动进入睡眠状态，并发起一次调度让出 CPU。这必须通过系统调用 nanosleep( ) 和 pause( )来实现。

3.4.1 nanosleep（）及部分子函数

1．nanosleep 有两个入口参数 rqtp,rmtp，都是 timespec 结构指针。rqtp 指向给定所需睡眠时间的数据结构；rmtp 指向返回剩余睡眠时间的数据结构。

数据结构 timespec 的定义在 include/linux/time.h 中

```
struct timespect {
        time_t    tv_sec;    /*秒*/
        long    tv_nsec;    /*纳秒*/
    }
```

nanosleep 的工作流程是这样的：

（1） 检查入口参数是否有错，有错则退出。

（2） 若要求睡眠的进程是实时进程,且睡眠时间小于 2 毫秒，延时等待，并不真正睡眠。由于时钟中断的频率为 100，即时钟中断的周期为 10 毫秒，所以进程进入睡眠并按正常途径由时钟中断服务程序唤醒只能达到 10 毫秒的精度。所以睡眠时间小于 2 毫秒就必须采用延时的方式以保证精度。

（3） 若非（2）中的情况，将当前进程置可中断睡眠态并调用 schedule_timeout（）进入睡眠。

（4） 进程被唤醒后，若还要剩余的睡眠时间，则将其传给 rmtp，并返回错误码；若睡眠时间无剩余则正常返回。

2. schedule_timeout( )

schedule_timeout( )通过调用 add_timer( )负责安排定时器，使系统将来能够按时唤醒进程。在安排定时器后，schedule_timeout( )主动调用 schedule（）转入进程调度。

具体的流程和解释参见第 5、6 章。

3. add_timer( )

add_timer( )调用 internal_add_timer( )将定时器挂入定时器队列。

4. internal_add_timer( )

internal_add_timer( )是将定时器挂入定时器队列的底层函数。在 4.3 节中我们已经介绍了定时器队列的结构。

internel_add_timer(struct timer_list *timer)根据系统定时器的终结时间与以 jiffies 记数的当前系统时间之差而把定时器添加到系统定时器队列中适当的定时器链上。

internel_add_timer(struct timer_list *timer)的流程如下：

(1)计算 timer_list 结构指针 timer 指向的时间间隔定时器的终结时间与 timer_jiffies 的差（在级联处理系统定时器队列时，timer_jiffies == jiffies）：

unsigned long idx = timer->expires - timer_jiffies;

(2)根据 idx 值（体现出系统定时器的生存时间）的大小，以及定时器终结时间 timer->expires 的 0~7、8~13、14~19、20~25 或 26~31 位片段上的数值，将 timer 所指示的定时器对应地添加到 tvecs[0]、tvecs[1]、tvecs[2]、tvecs[3]或 tvecs[4]上的定时器链上。(注：TVR_SIZE= = 256，TVN_SIZE= = 64)

其中，timer->expires < timer_jiffies 的情况在系统定时器队列的级联处理中不可能发生。它在理论上只可能发生在用户添加了终止时间等于以 jiffies 记数的系统时间的定时器或者定时器被设置成是在小于当前系统时间的某个时间到期。这样的定时器会被添加在由 tv1.index 指向的定时器链上，并在核心下一次调用定时器 bottom half 处理过程时得到及时的处理。

因此，系统在添加时间间隔定时器时，基于判断（idx = timer->expires — timer_jiffies）的大小，把idx小于 256 的定时器加在tv1 上由终止时间的低 8 位($expires_{0\sim7}$)指向的定时器链上；把idx位于[256，2^{14})的定时器加在tv2 上由终止时间的次低 6 位($expires_{8\sim13}$)指向的定时器链上；把idx位于[2^{14}，2^{20})的定时器加在tv3 上由终止时间的较低 6 位($expires_{14\sim19}$)指向的定时器链上；把idx位于[2^{20}，2^{26})的定时器加在tv3 上由终止时间的 6 位长的片段($expires_{20\sim26}$)指向的定时器链上；另外，把idx位于[2^{26}，2^{32}]的定时器加在tv3 上由终止时间的高 6 位($expires_{27\sim32}$)指向的定时器链上。

系统的 timer_jiffies 值在进程添加时间间隔定时器时等于 jiffies+1，而在核心对系统定时器队列进行级联处理时等于 jiffies。

5. 中断唤醒进程

时钟中断属于硬件中断，会定期发生。在从时钟中断返回之前要执行于时钟有关的 bottom half 函数 timer_bh( )。timer_bh( )执行两个操作，一是更新系统时间，二是调用
run_timer_list（）执行定时器队列中到点的任务。如果进程的睡眠在此时到点，进程就被唤醒了。

Run_timer_list（）流程如下：

（1） 定时器队列的级联处理

若定时器队列 tv1 中已没有待处理的任务，设置循环,从 tv[n]搬运一组链表,分散插入到 tv[n-1]的各个链表中。

（2） 循环,激活所有在此时应唤醒的定时器

（3） 到下次时钟中断服务时 timer_jiffies 增加一个单位,以保持插入定时器操作 internel_add_timer 中计算 idx=expires-timer_jiffies 的正确性下次时钟中断时,应对第 tv1.index + 1 条链表操作。

3.4.2 sys_pause( )

sys_pause( )的代码非常简单，仅仅是调用 schedle( )，也就是说让进程

一直睡眠下去，并不指定睡眠多长时间。

sys_pause( )代码如下：

```
/* 位置 arch/i386/kernel/sys_i386.c */
asmlinkage int sys_pause(void)
{
current->state = TASK_INTERRUPTIBLE;  /* 设当前进程状态为可中断睡眠态*/
schedule();                                          /* 转入进程调度*/
return -ERESTARTNOHAND;                    /* 返回错误码-ERESTARTNOHAND*/
}
```

3.5 对于 Linux 进程调度的研究总结

3.5.1 Linux 对调度算法的应用

Linux 系统把进程分为实时进程和非实时进程。对于实时进程，Linux 又把它们分成两种：

一种是具有某种紧迫性的实时进程，Linux 对它们采用 FIFO 的调度方式，即排在运行队列前面的进程先运行，直到它主动放弃 CPU 后，才能轮到下一个进程运行。由于采取这种算法，若是进程执行的任务比较短小，还能满足实时的需要，若是任务比较大，就会出现一个紧急的任务因为当前进程没有执行完而无法获得 CPU，导致任务被迫等待而无法满足实时要求。对于这种实时进程，采用抢占式的调度算法更为合适，因此 Linux 处理实时任务的能力并不令人满意。

另一种实时进程是用于保证人－机及时性而设置的,称为"分时"进程也许更为贴切。Linux 对这种进程采用时间片轮转调度法，使每个进程都能及时地得到 CPU，完全可以满足人－机交互的需要。

对于非实时进程，Linux 采用优先权调度算法。采用这种算法，在很大程度上是在模仿 UNIX。UNIX 系统应用的事实证明，这种算法是很有效率的，它可以充分利用 CPU 时间，并使每一个进程都能够较公平地获得 CPU 时间。

3.5.2 Linux 进程优先级的代码实现

Linux 很巧妙的把分配给进程的时间片和进程的优先级结合起来。Linux 直接用进程的时间片作为进程优先级的量度,进程拥有的剩余时间片数目即进程的优先级，时间片越多，优先级越高。进程在运行中时间片不断消耗减少，时间片越小，进程的优先级就越小，从而使其它进程有机会取代当前进程获得 CPU。

3.5.3 Linux 对 CPU 的充分利用

在整个计算机系统中 CPU 是最宝贵的资源，所有的任务必须由 CPU 来执行。因此只有充分利用 CPU，才能最大程度的提高系统的效率。Linux 是一个多进程系统，它通过进程调度解决了充分利用 CPU 的问题。当一个进程因为得不到某些资源而等待时，Linux 在此时不是让 CPU 空转等待进程获得足够资源后继续运行，而是采取一定的调度策略选择另外一个进程交给 CPU 去执行。虽然某个进程会因为各种原因停止运行，CPU 却总是有任务在执行，这样，CPU 就得到了充分的利用，从而大大提高了系统的效率和吞吐量。

第 4 章 Linux 进程的控制

4.1 进程的创建、执行、消亡

每个进程都有被创建、执行某段程序以及最后消亡的过程。在 Linux 系统中，第一个进程是系统固有的或者说是由内核的设计者安排好了的。内核在引导并完成了基本的初始化以后，就有了系统的第一进程（实际上是内核线程，它没有用户空间）。除此以外，所有的其它进程都是由这个原始进程或其子孙进程创建的，都是这个原始进程的后代。在 Linux 系统中，一个新的进程一定要由一个已经存在的进程“复制”出来，而不是“创造”出来，所以这里所说的“创建”实际就是复制。

Linux 将进程的创建与目标程序的执行分成两步。第一步是从已经存在的父进程中复制出一个子进程。复制出来的子进程有自己的 task_struct 结构和内核空间堆栈，并与父进程共享其他所有的资源。Linux 为此提供了三个系统调用，一个是 fork( )，一个是 clone( )，还有一个是 vfork( )。

第二步是目标程序的执行。一般来说，创建一个新的进程是因为有不同的目标程序要让新的进程去执行，所以，复制完成用户，子进程通常要与父进程分离，走自己的路线。Linux 为此提供了一个系统调用 execve( )，让一个进程执行以文件形式存在的一个可执行程序的映象。

创建了子进程以后，父进程有三个选择：

第一：继续走自己的路，与子进程分离。如果子进程先于父进程消亡，则由内核给父进程发一个子进程消亡的信号。

第二：进入睡眠状态，等待子进程完成任务而最终消亡，然后父进程再继续运行。Linux 为此提供了两个系统调用，wait4( )和 wait3( )。wait4( )等待特定的子进程消亡，而 wait3( )则等待任何一个子进程消亡。

第三：结束自己的生命。Linux 为此提供了一个系统调用 exit( )。

这里的第三种选择其实是第一种选择的特例，所以从本质上说是两种选择：一种是父进程不受阻（non_blocking）的方式，也称为“异步”方式；另一种是父进程受阻（blocking）的方式，也称为“同步”方式。

下面是一个演示程序：

```
01  #include <stdio.h>
02  #include <unistd.h>
03  int main( )
04  {
05        int child;
06        char *arg[ ] = { “/bin/echo”,”Hello”,”World!”,NULL};
07
08        if(!(child = fork( )))
09          {
10                /* child */
11                printf(“pid %d: %d is my father\n”,getpid( ),getppid( ));
12                execve(“/bin/echo”,args,NULL);
13                printf(“pid %d: I am back,something is wrong!\n”,getpid());
```

```
14            }
15          else
16            {
17                  int myself = getpid( );
18                  printf("pid %d: %d is my son\n",myself,child));
19                  wait4(child,NULL,0,NULL);
20                  printf("pid %d: done\n",myself);
21             }
22          return 0;
23    }
```

这里，进入 main()的父进程，它在第 8 行执行了系统调用 fork( )创建一个子进程，也就是复制了一个子进程。子进程复制出来以后，就象其父进程一样地接受内核的调度，而且具有相同的返回地址。所以，当父进程和子进程受调度继续运行而从内核空间返回时都返回到同一点上。以前的代码只有一个进程执行，而从这一点开始却有两个进程在执行了。复制出来的子进程与父进程是有区别的。首先，子进程有与父进程的 pid，且子进程的 task_struct 结构中有几个数据成员说明了谁是它的“父亲”。其次，子进程和父进程从 fork( )返回时的所具有的返回值不同。子进程从 fork( )返回时，其返回值是 0；而父进程的返回值是子进程的 pid，这是不可能为零的。这样，第 8 行的 if 语句就可以根据这个特征把二者区分开。第 10～12 行属于子进程，16～19 行属于父进程。在这个程序中，父进程执行 wait4( )停下来等待子进程的消亡或让出 CPU；而子进程则通过 execve( )执行“/bin/echo”。子进程在执行 echo 以后不会回到这里的第 13 行，这是因为在/bin/echo 中必定有一个 exit( )调用，使子进程自行消亡。对 exit( )的调用是每一个可执行映象必有的，虽然这个程序中并没有调用它，而是以 return 语句从 main( )返回，但 gcc 在编译和连接时会自动加上。

由于子进程与父进程一样接受内核调度，而每次系统调用都有可能引起调度，所以二者返回的先后次序是不确定的。

4.1.1 系统调用的实现

无论是 fork()，还是 execve( )，都可以在应用程序中直接调用。所谓系统调用，就是调用操作系统直接提供的内核例程来完成需要的服务。用户所写的应用程序是在用户态下运行的，在用户态下的程序需要以中断的方式透过操作系统进入核心空间执行内核例程。

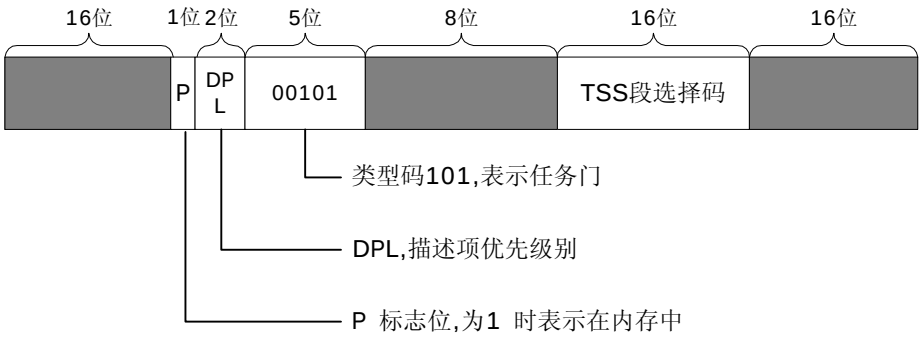
1. 中断的硬件支持

Intel X86 CPU 有两种工作模式,实地址模式和保护模式。 Linux 工作在保护模式下。在保护模式下，Linux 的内核空间得到有效保护，任何在用户态下运行的程序都无法直接访问内核空间，执行内核态例程（即处于内核空间属于 Linux 内核组成部分的函数）。用户程序只能使用语句 INT 0X80 产生专门用于系统调用的中断，从而进入内核空间执行相应的内核服务例程。

Intel 为满足 CPU 在保护模式下的这种特性将中断向量表（参见后面的中断机制示意图）中的表项设置为类似于 PSW（程序状态字）加入口地址并且更为复杂

的描述项，称为门（gate），意思是当中断发生时必须先通过这些门，才能进入相应的服务程序。其实，这样的门并不光是为中断而设的，只要想切换 CPU 的运行状态，即其优先级别，例如从用户的 3 级进入系统的 0 级，就都要通过一道门。按不同的用途和目的，CPU 中一共有四种门，即任务门（task gate）、中断门（interrupt gate）、陷阱门（trap gate）以及调用门（call gate）。其中除任务门外其它三种门的结构基本相同，不过调用门并不是与中断向量表相联系的。

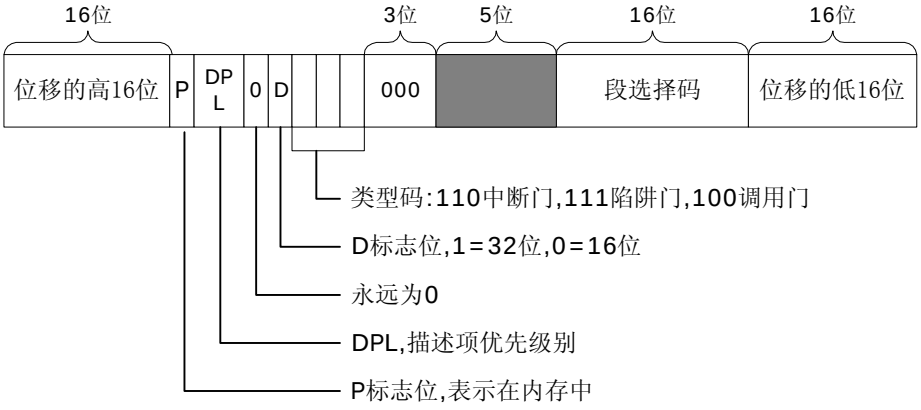
任务门大小为 64 位，结构如下图所示。



任务门结构图

TSS 段选择码的作用和段寄存器 CS、DS 等相似，通过 GDT 或 LDT 指向特殊的“系统段”中的一种，称为“任务状态段”（task_state segment）TSS。TSS 实际上是一个用来保存任务运行“现场”的数据结构，其中包括 CPU 总所有于具体进程有关的寄存器的内容，还包括了三个堆栈指针。中断发生时，CPU 在中断向量表中找到相应的表项。如果此表项是一个任务门，并且通过了优先级别检查，CPU 就会将当前任务的运行现场保存在相应的 TSS 中，并将任务门所指向的 TSS 作为当前任务，将其内容装入 CPU 中的各个寄存器，从而完成一次任务的切换。为此目的，CPU 中又增设了一个“任务寄存器”TR，用来指向当前任务的 TSS。为了对 TR 寄存器进行装入操作还增设了一条指令 LTR。按照 Intel 的本意是希望操作系统为每一个任务分配一个 TSS，当任务切换时通过 TR 指向的变化来指导 CPU 将当前任务的信息载入各个寄存器以完成任务切换。前面已经说过，在 Linux 中任务即进程。进程的 task_struct 结构需要存放更多的信息，所以 Linux 的进程并不完全是 Intel 设计意图中的任务。因此 Linux 不采用任务门作为进程切换的手段。但 Linux 要使用 TSS 保存进程的现场(保存进程在不同空间的堆栈指针 SS0 和 ESP0)。Intel 是想让 TR 的内容，随着不同的 TSS，随着任务的切换而轮转。但由于 Linux 只需要 TSS 中的 SS0 和 ESP0，而改变 SS0 和 ESP0 所花的开销比通过装入 TR 以更换一个 TSS 要小得多。因此，在 Linux 中只有一个 TSS，这个 TSS 并不属于某个进程的资源，而是全局性的公共资源，它里面的内容（当前任务的系统堆栈指针）随着进程的调度切换而变动。

下面再来看一看其余三种门的结构。这三种门基本相同，大小都是 64 位，如下图所示。

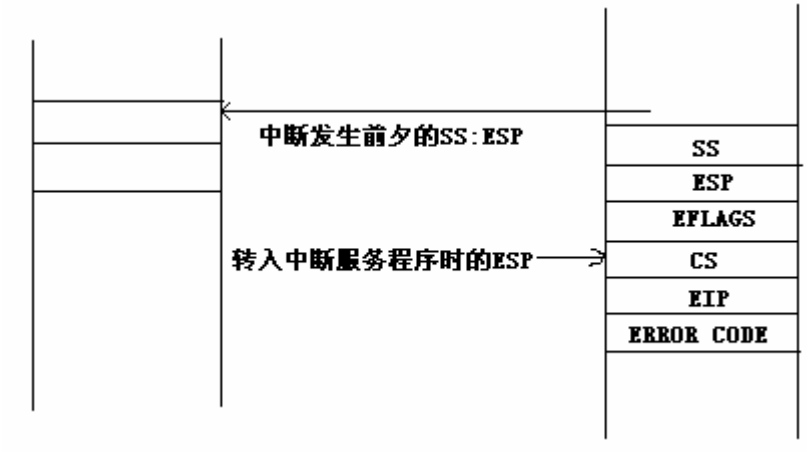


中断门、陷阱门和调用门结构图

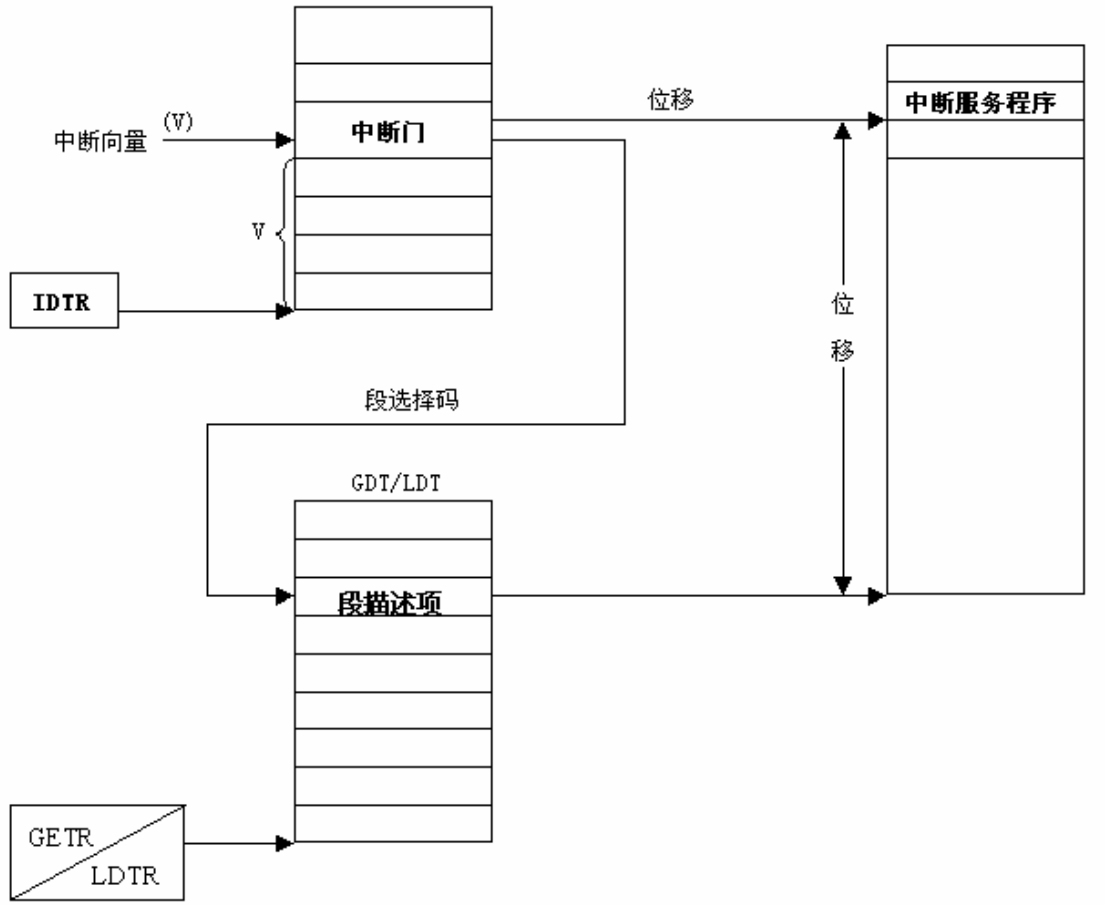
三种门之间的不同之处在于 3 位类型码：110 代表中断门、111 代表陷阱门、100 代表调用门。中断门和陷阱门在使用上的区别是，通过中断门进入中断服务程序时 CPU 会最大将中断关闭，以防止嵌套中断的发生，而陷阱门则不然。中断门都一个 DPL（描述项优先级别）位段。当 CPU 通过中断门找到一个代码段描述项，并进而转入相应的服务程序时，就把这个代码段描述项装入 CPU 中，而描述项的 DPL 就变成 CPU 的当前运行级别，称为 CPL。

当通过一条 INT 指令进入一个中断服务程序时，在指令中给出一个中断向量。CPU 先根据该中断在中断向量表中找出一扇门，然后将这个门的 DPL 与 CPU 的 CPL 相比，CPL 必须小于或等于 DPL，即优先级不低于 DPL，才能穿过这道门。穿过门之后还要将目标代码段描述项中的 DPL 与 CPL 比较，目标段的 DPL 必须小于或等于 CPL。即通过中断门时只允许保持或提升 CPU 的运行级别。

进入中断服务程序时，CPU 要将 EFLAGS 寄存器的内容和返回地址压栈。如果 DPL 与中断发生时的 CPL 不同（在 Linux 中是从 3 级的用户态进入 0 级的核心态，3 级<0 级），就需要更换堆栈，而新堆栈（核心空间堆栈）的指针是从 TSS 结构中取得的。在这种情况下，CPU 除了要将 EFLAGS 寄存器的内容和返回地址压入核心空间堆栈，还要将原来的用户空间堆栈指针压栈，以备返回时使用。下图形象地说明了这种情况。



下图说明了中断的机制：



中断机制示意图

2. 中断的软件实现

Linux 为了实现中断机制，需要在系统启动之初初始化中断向量表 IDT 和中断请求队列。

Linux 内核在初始化阶段完成了对页式虚存管理的初始化以后，会调用 trap_init( )和 init_IRQ( )两个函数进行中断机制的初始化。

trap_init( )主要对一些系统保留的中断向量（异常处理）的初始化，init_IRQ 则设置用于外设的通用中断（门）向量，对这些门所设置的服务程序都来自函数指针数组 interrupt[ ]。

IDT 共有 256 个表项，分为两种。一种为保留专用于 CPU 本身的中断门（异常处理）和用户程序通过 INT 指令产生的中断（trap，主要是系统调用）。另一种是用于外设的通用中断门，从 0x20 开始除 0x80（用于系统调用）以外的 224 项都是。

中断请求队列用于通用中断门，它不在本论文专题的讨论范围内，因此略去。

3. 系统调用的实现

本节介绍进程在系统调用中进入内核空间，以及在完成了所需的服务后从内核空间返回的过程。这个过程是每一个系统调用都要经历的过程。

系统调用是 CPU 主动地、同步地进入内核空间的手段。Linux 的系统调用是

通过中断指令“INT 0x80”实现的。

仍以 4.1 中列出的程序为例，看一个简单的 fork( )语句是如何转化为 INT 0x80 指令，从而进入内核空间的。

进程的系统调用命令进行格式转换和参数传递要利用一个宏定义 syscallN()(见 include/asm/unistd.h).请看下例：

```
#define _syscall5(type,name,type1,arg1,type2,arg2,type3,arg3,\
                  type4,arg4,type5,arg5)\
    type neme(type1 arg1,type2 arg2,type3 arg3,type4 arg4,type5 arg5)\
     {\
      long __res;\
      __asm__volatile("int $0x80" \
          :"=a"(__res)\
          :"0"(__NR_##name),"b"((long)(arg1)),"c"((long)(arg2)),\
           "d"((long)(arg3)),"S"((long)(arg4)),"D"((long)(arg5)));\
      if(__res>=0)\
         return(type)__res;\
         errno=-__res;\
      return -1;\
     }
```

通过这样的宏,进程的系统调用命令转换为 INT 0x80 中断请求.

N 取 0 与 5 之间任意整数.参数个数为 N 的系统调用由 syscallN 负责格式转换和参数传递.例如,fork()无参数,它对应的格式转换宏就是 syscall0();open()有 3 个参数,它对应的格式转换宏就是 syscall3().

syscallN()的第一个参数说明响应函数返回值的类型，第二个参数为系统调用的名称(即 name)，其余的参数依次为系统调用参数的类型和名称.例如:

_syscall3(int,open,const char *,file,int,flag,int,mode)

说明了系统调用命令:

int open(const char *file,int flag,int mode)

宏定义的余下部分描述了参数描述,启动 INT 0x80 以及接受,判断返回值的过程.也就是说,以系统调用号对 EAX 寄存器赋值,启动 INT 0x80.规定返回值送 EAX 寄存器。函数的参数进行压栈，压栈顺序见下表：

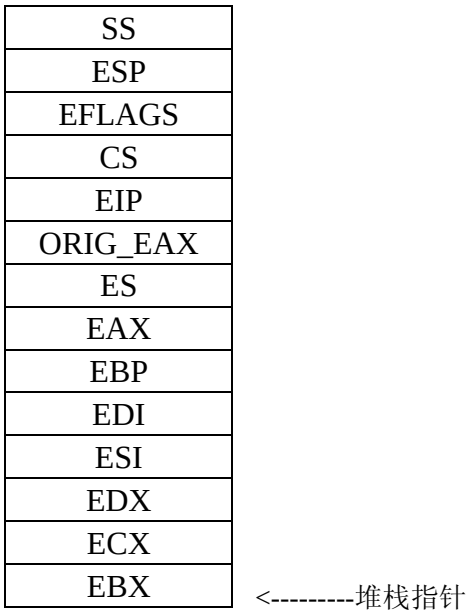
参数	参数在堆栈的位置	传递参数的寄存器
arg1	00(%esp)	ebx
arg2	04(%esp)	ecx
arg3	08(%esp)	edx
arg4	0c(%esp)	esi
arg5	10(%esp)	edi

当执行 INT 0x80 时,以内核态进入入口地址 system_call.

ENTRY(system_call)

```
pushl %eax
SAVE_ALL
    :
    :
```

SAVE_ALL 是一个宏,目的是把 CPU 当前的寄存器的值压入堆栈(系统栈)
SAVE_ALL 执行后系统栈的内容如图：



而内核中每一个处理系统调用的函数都以 asmlinkage 标志开头。这是为一些 gcc 功能定义的一个宏,它告诉编译器该函数不希望从寄存器中取得任何参数，而希望仅仅从 CPU 堆栈中取得参数。也就是说，像上面我们看到的，一个系统调用命令所含有的参数在进入 system_call 后通过 SAVE ALL 被压入系统栈，内核处理函数直接从堆栈中取得这些参数.值得注意的是,Linux 设置了一个称为 pt_regs 的结构与系统栈中的内容相对应,内核处理函数通过该结构取得系统栈内相应参数的值.pt_regs 定义如下：

```
11544 /* this struct defines the way the registers are stored
11545  * on the stack during a system call. */
11546 struct pt_regs {
11547   long ebx;
11548   long ecx;
11549   long edx;
11550   long esi;
11551   long edi;
11552   long ebp;
11553   long eax;
11554   int  xds;
11555   int  xes;
11556   long orig_eax;
11557   long eip;
11558   int  xcs;
11559   long eflags;
11560   long esp;
11561   int  xss;
11562 };
```

正是因为要使用 syscallN()来将用户程序中简单的系统调用语句转换成 INT 0x80 指令，并传递参数，所以要在程序的开头写上

#include <unistd.h>

syscallN()宏保存在这个头文件里。

并不是所有的系统调用都用走 syscallN()这条路线。例如 sys_clone( ),这是一个用于创建进程的内核服务程序。函数源码如下：

```
2698 asmlinkage int sys_clone(struct pt_regs regs)
2699 {
2700   unsigned long clone_flags;
2701   unsigned long newsp;
2702
2703   clone_flags = regs.ebx;/*?????*/
2704   newsp = regs.ecx;
2705   if (!newsp)
2706     newsp = regs.esp;
2707   return do_fork(clone_flags, newsp, &regs);
2708 }
```

该系统调用的程序设计接口是这样的：

int __clone(int(*fn)(void *arg),void *child_stack,int flags,void *arg);

当执行这条命令时，如果要调用 syscallN()宏的话，其参数将通过 syscall4(...)被读入 CPU 的寄存器,然后执行 INT 0x80 进入 system_call,再通过 SAVE ALL 将寄存器的值统统压入系统栈.sys_clone 通过 regs 取得系统栈中保存的参数值，

按上述路线执行后，sys_clone 通过 regs 取得系统栈中保存的参数值，其对应值应该如下(请参照上面所画的系统栈)：

```
regs.ebx<===>int(*fn)(void *arg)
regs.ecx<===>void *child_stack
regs.edx<===>int flags
regs.esi<===>void *arg
```

请看 sys_clone 的代码，其中 2703 行

```
2703    clone_flags = regs.ebx;
```

clone_flags 应该对应 regs.edx，而在这里却对应 regs.ebx。

通过阅读接口函数

int __clone(int(*fn)(void *arg),void *child_stack,int flags,void *arg);

的具体代码（该函数位置在 glibc/sysdeps/unix/sysv/linux/i386/clone.S）可知，程序在执行上述接口语句时，并没有调用 syscallN()宏进入 system_call，而是直接通过接口函数进入 system_call 的。在接口函数中，fn 和 arg 被压到 child_stack 中，传给 kernel 的只有两个参数，ebx=flags，ecx=child_stack-8（因为 fn 和 arg 被压到这里）函数代码如下：

```
/* int clone(int (*fn)(void *arg), void *child_stack, int flags, void *arg); */

#define PARMS LINKAGE
#define FUNC PARMS          /* FUNC 为 fn 在堆栈中相对于（esp 指向的）栈顶的偏移量*/
#define STACK FUNC+4        /* STACK 为 child_stack 在堆栈中相对于（esp 指向的）栈顶的偏移量*/
#define FLAGS STACK+PTR_SIZE  /* FLAGS 为 flag 在堆栈中相对于（esp 指向的）栈顶的偏移量*/
```

```
                                        /* PTR_SIZE 为指针 child_stack 的 size*/
#define ARG FLAGS+4          /* ARG 为 arg 在堆栈中相对于（esp 指向的）栈顶的偏移量*/

.text
ENTRY (BP_SYM (__clone))
/* Sanity check arguments. */
movl $-EINVAL,%eax
movl FUNC(%esp),%ecx                    /*将用户设定的新栈基址(child_stack)赋给 ecx */
#ifdef PIC
jecxz SYSCALL_ERROR_LABEL
#else
testl %ecx,%ecx
jz SYSCALL_ERROR_LABEL
#endif
movl STACK(%esp),%ecx /* no NULL stack pointers */
#ifdef PIC
jecxz SYSCALL_ERROR_LABEL
#else
testl %ecx,%ecx
jz SYSCALL_ERROR_LABEL
#endif

/* 将变量 arg 和 fn 压入新栈 */
subl $8,%ecx
movl ARG(%esp),%eax /* no negative argument counts */
movl %eax,4(%ecx)

/* Save the function pointer as the zeroth argument.
It will be popped off in the child in the ebx frobbing below. */
movl FUNC(%esp),%eax
movl %eax,0(%ecx)

/* 执行系统调用*/
pushl %ebx
movl FLAGS+4(%esp),%ebx                 /* 将父进程堆栈中的 flag 值读入 ebx*/
movl $SYS_ify(clone),%eax               /* 将中断序列号读入 eax*/
int $0x80                          /* 中断指令，跳转到 ENTRY（system_call）*/
popl %ebx

test %eax,%eax
jl SYSCALL_ERROR_LABEL
jz thread_start

L(pseudo_end):
```

```
ret
thread_start:
subl %ebp,%ebp /* terminate the stack frame */
call *%ebx
#ifdef PIC
call L(here)
L(here):
popl %ebx
addl $_GLOBAL_OFFSET_TABLE_+[.-L(here)], %ebx
#endif
pushl %eax
call JUMPTARGET (_exit)

PSEUDO_END (BP_SYM (__clone))

weak_alias (BP_SYM (__clone), BP_SYM (clone))
```

由此可见，只要愿意，编写系统调用服务程序的程序员可以自由决定将系统调用命令转换为 INT 0x80 中断请求的方式。

不论是通过 syscallN（）还是别的函数将系统调用命令转换为 INT 0x80 中断请求，这些函数都会将系统调用名换算成系统调用号赋给寄存器 eax，在转向 ENTRY(system_call)（这部分的详细解释请见第 6 章：内核源码注释）后通过 eax 中的系统调用号查询系统调用跳转表，找到对应的内核服务程序，并执行。

每一个系统调用都有一个唯一的系统调用号。这些系统调用号定义在 include/asm/unistd.h。部分编号如下：

```
#define __NR_exit          1
#define __NR_fork          2
#define __NR_read          3
#define __NR_write         4
#define __NR_open          5
```

系统调用跳转表是一个函数指针数组，跳转时以系统调用号为下标在数组中找到相应的函数指针。该数组是在 arch/i386/kernel/entry.S 中定义的。数组的大小由常数 NR_syscalls（见 include/linux/sys.h）决定，该常数定义为 256。在 Linux 2.4.0 版本下，共定义了 221 个系统调用，其余的 35 项定义为 sys_ni_syscall( )，该函数是一个空函数，只返回错误码－ENOSYS。

部分代码如下

```
ENTRY(sys_call_table)
        .long SYMBOL_NAME(sys_ni_syscall)       /* 0  -  old "setup()" system call*/
        .long SYMBOL_NAME(sys_exit)
        .long SYMBOL_NAME(sys_fork)
        .long SYMBOL_NAME(sys_read)
        .long SYMBOL_NAME(sys_write)
```

```
.rept NR_syscalls-(.-sys_call_table)/4
    .long SYMBOL_NAME(sys_ni_syscall)
```

当CPU执行完具体系统调用的服务程序后，就要从内核空间返回到用户空间。在返回之前服务程序会把准备好的返回值存在寄存器%eax中，当CPU回到进入内核空间前的那一处代码后执行的第一条语句

```
movl %eax,EAX(%esp)        # save the return value
```

就是将返回值写入到堆栈中与%eax对应的地方，这样在RESTORE_ALL（功能是从堆栈恢复各个寄存器的值，与SAVE_ALL相对）以后，这个返回值仍通过%eax传回用户空间。

执行完上条语句后，CPU就到达了ENTRY(ret_from_sys_call)（具体的解释参见第6章）,从这里向下顺序执行，CPU就可以安全返回用户空间了。

4.1.2 子进程的创建

1. sys_fork( )、sys_clone( )和sys_vfork( )

Linux系统提供fork( )、clone( )和vfork( )这三个系统调用用于进程的创建。这里主要讨论这三个系统调用内核服务程序，即本小节标题列出的三个函数（详细解释参见第6章）。

这三个函数都是通过调用 do_fork( )来实现具体创建工作的，但它们传递给do_fork( )的参数不同，因此创建出的进程是不同的。

sys_clone( )用于创建一个线程，这个线程可以是内核线程，也可以是用户线程。

sys_fork( )用于全面地复制，创建出的新进程与父进程几乎相同。

sys_vfork( )也用于创建线程。但主要只是作为创建进程的中间步骤，目的在于提高创建时的效率，减少系统开销。

2. do_fork( )

do_fork()函数流程

do_fork()大致做了四块内容：

（1） 新建子进程分配task_struct 空间

（2） 子进程task_struct的初始化

（3） 子进程file,fs,sighand,mm,thread五方面信息的复制

（4） 出错滚回处理

具体解释和流程参见第5、6章。

简要说明：

（1）为新建子进程分配task_struct 空间

调用 kmalloc(sizeof(*p), GFP_KERNEL);给新建子进程分配一个task_struct结构。

（2）子进程task_struct的初始化

初始化指除了后面要分析的五方面进程信息之外的那些虽然琐屑无比却不可或缺的信息。

（3）子进程file,fs,sighand,mm,thread五方面信息的复制

这些信息的拷贝在 do_fork()函数中是通过调用子函数实现的。代码如下：

```
if (copy_files(clone_flags, p))
    goto bad_fork_cleanup;
if (copy_fs(clone_flags, p))
    goto bad_fork_cleanup_files;
if (copy_sighand(clone_flags, p))
    goto bad_fork_cleanup_fs;
if (copy_mm(clone_flags, p))
    goto bad_fork_cleanup_sighand;
copy_thread(nr, clone_flags, usp, p, regs);
```

3. 文件的复制 copy_files()

在解释 copy_files()的流程之前，有必要先看一下相关的数据结构：

（1）task_struct 中与文件相关的数据项是 files

```
struct  files_struct  *files;
```

它指向一个 files_struct 结构变量。

（2）files_struct

它描述的是一个进程已经打开的文件：

```
struct files_struct {
  int count;
  fd_set close_on_exec;
  fd_set open_fds;
  struct file * fd[NR_OPEN];
};
```

可以看出，Linux 中一个进程最多只能同时打开 NR_OPEN 个文件，而且前三项分别设为标准输入、标准输出和出错信息输出文件。

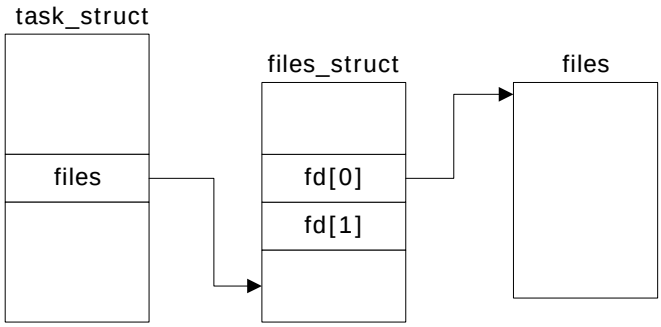
（3）file 结构变量

每个打开的文件及插口附加项(socket etcetera)都由 file 结构表示：

```
struct file {
  mode_t f_mode;
  loff_t f_pos;
  unsigned short f_flags;
  unsigned short f_count;
  unsigned long f_reada, f_ramax, f_raend, f_ralen, f_rawin;
  struct file *f_next, *f_prev;
  int f_owner;    /* pid or -pgrp where SIGIO should be sent
*/
  struct inode * f_inode;  struct file_operations * f_op;
  unsigned long f_version;
  void *private_data;
  /* needed for tty driver, and maybe others */
```

};

三个数据的关系如下图所示：



copy_files 函数主要做了以下工作：

（1） 义了指向 files_struct 结构的指针 oldf 和 newf，其中 oldf 指向父进程的文件结构：

```
oldf=current->files;
```

（2）然后作一下判断：如果子进程与父进程共享一个 files_struct，则把父进程的 files_struct 中的 count 增 1：

```
if(clone_flags&CLONE_FILES) {
    oldf->count++;
    return 0;
}
```

注：在文件\Include\Linux\Sched.h 中可以看到：

```
#define  CLONE_FILES      0x00000400
```

否则，就要给子进程另外分配一块放 files_struct 的空间：

```
newf = kmalloc(sizeof(*newf), GFP_KERNEL);
tsk->files = newf;
if (!newf)
    return -1;
```

（3）这只是分配了空间，还要初始化：

```
newf->count = 1;
newf->close_on_exec = oldf->close_on_exec;
newf->open_fds = oldf->open_fds;
```

（4）最后要做的是子进程逐项拷贝父进程的指向文件的指针。

4. 文件系统的复制 copy_fs()

其中重要的数据结构是 struct fs_struct *fs;

fs 指出了进程本身在 VFS(Virtual File System)中的位置，其中 root 指向根目录，pwd 指向当前目录结点，umask 给出了文件的缺省创建模式，count 是文件的引用次数，在文件\Include\Linux\Sched.h 中有：

```
struct fs_struct {
int count;
unsigned short umask;
struct inode * root, * pwd;
```

};

观察函数 copy_fs()可以发现，其操作流程与 copy_file()完全类似，这里略去。

5. 信号处理信息的复制 copy_sighand()

在 Intel 机器上最多只能接受 32 种信号，其实信号数目决定于信号变量的长度，32 位信号变量自然可以表示 32 种信号。对每种信号，各种进程可以选择使用自定义的处理函数或是系统的缺省处理函数。task_struct 中由一个指针指向处理信息：

```
struct signal_struct *sig;
```

signal_struct 的结构(/include/linux/sched.h)如下：

```
struct signal_struct{
       int    count;
       struct sigaction action[32];
};
```

其中 action 数组的每一项为信号处理函数的入口地址，至于哪个信号对应哪个函数，则由信号变量的值映射到 action 数组的偏移量来对应。

函数 copy_sighand()的流程与前两个基本类似：

(1) 判断是否共享(clone)

```
if (clone_flags & CLONE_SIGHAND) {
  current->sig->count++;
  return 0;
```

(2) 若不是，则分配空间：

```
tsk->sig = kmalloc(sizeof(*tsk->sig), GFP_KERNEL);
if (!tsk->sig)
 return -1;
```

(3) 再做初始化：

```
tsk->sig->count = 1;
memcpy(tsk->sig->action,current->sig->action, sizeof(tsk->sig->action));
```

这里是把父进程对信号的处理函数的入口地址逐项拷贝到子进程的 signal_struct 结构中。

其实，task_struct中除了sig指针和前面提到的signal长型变量(那是用来记录进程收到的信号)之外，还有blocked变量:

```
unsigned long 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。

除SIGSTOP和SIGKILL之外，所有的信号都可以被阻塞(blocked)。只有当信号被解除阻塞时，才会有效，否则一直被挂起。

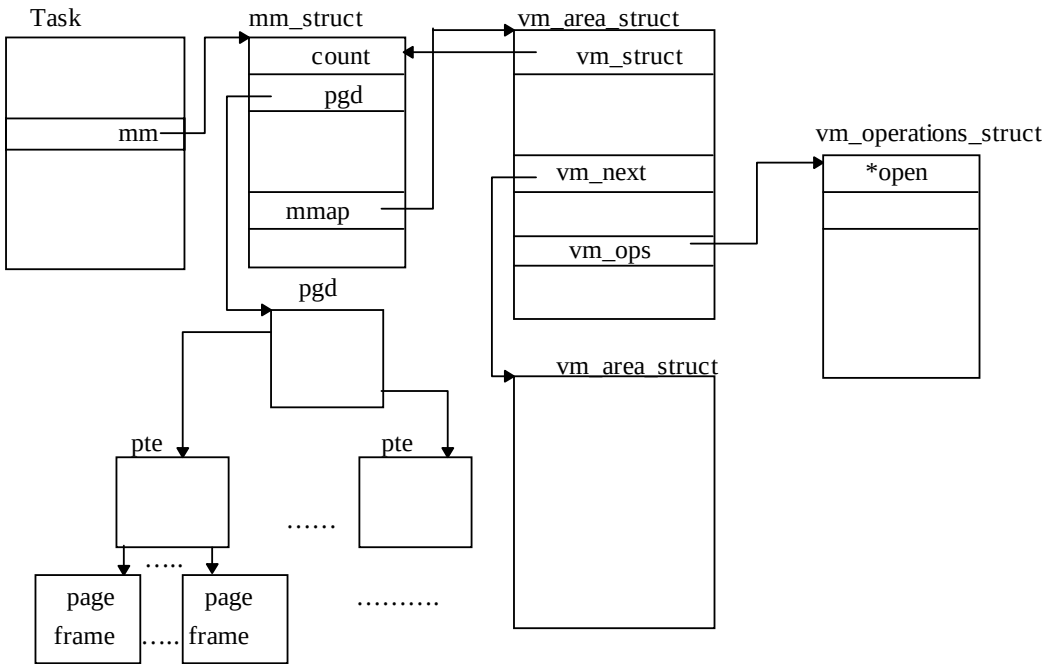
6. 内存管理信息的复制 copy_mm()

与前面三种信息的复制类似的，Linux主张内存也是父子进程共享而不是从新分配空间。只要子进程的task_struct中的mm指针指向父进程的mm_struct

结构变量，而该mm_struct中的count加1便可。

但这是其中最简单的情况，如果子进程需要另有一个内存信息空间，工作将是艰巨的：要为子进程产生新的mm_struct、vm_area_struct和页表等。如果拷贝父进程的虚存那更是不可想象，进程的虚存可能在物理内存，可能在兑换区，也可能在执行状态。所以，Linux采用了所谓“写时拷贝”(Copy On Write)的方法,即父子进程一开始总是共用同一份虚存空间，只有当其中一个准备往里写东西的时候系统才复制一份，这时将有缺页中断发生。

相关的数据结构的关系可参考下图：



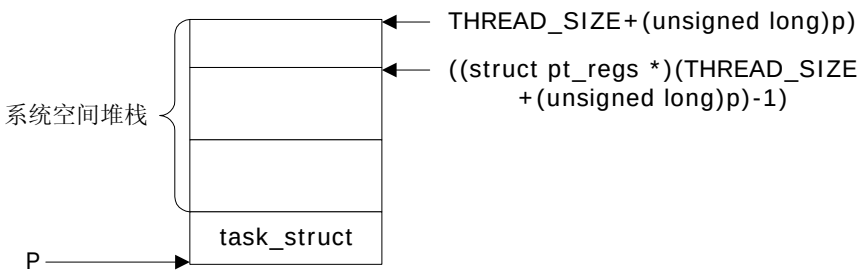
函数copy_mm()中先用以下语句给子进程一个mm_struct的框架：

```
struct mm_struct * mm = kmalloc(sizeof(*tsk->mm), GFP_KERNEL);
    if (!mm)
        return -ENOMEM;
    *mm = current->mm;
     /*这是从父进程把mm_struct结构拷贝到mm指向的结构*/
```

尔后要做的是给子进程的mm_struct装配一切必需的空间，其中主要工作是由另一个子函数dup_mmap()做的。

7. copy_thread()

名为copy_thread()，实际只是复制父进程的内核空间堆栈。具体解释见第6章。



子进程系统空间堆栈示意图

8. 出错滚回处理

fork系统调用也有可能失败，这时就要作滚回似乎没调用过一样，关键是把已分配给子进程的空间全部释放，这也意味着子进程不再存在。如果改变了父进程的task_struct数据的，也要改回来。

观察这一块源代码，可以发现次序与上面出错的次序正好相反，这种技巧正好保证作过的工作越多，清理工作也越多。

只用看一下其中exit_sighand()是怎么做的就大致有数了：

```
void exit_sighand( struct task_struct *tsk)
{
    _exit_sighand(tsk);
}
```

以及：

```
static inline void _exit_sighand( struct task_struct *tsk)
{
    struct signal_struct *sig=tsk->sig;
    if(sig) {
        tsk->sig=NULL;
        if(!--sig->count){
            kfree(sig);
        }
}
```

不难看出，exit_sighand 把tsk中的sig指针置为NULL，并且如果这时该signal_struct结构变量已没有进程指着了，那就把它释放掉。

4.1.3 子进程的执行

通过 fork 可以创建一个新的子进程，但到现在为止，这个新进程还只是其创建者的“影子”，不能执行任何不同于父进程的任务。之所以要创建新进程，是因为有大量的事情需要处理，而新的独立的进程正是处理这些事情的最佳选择。

要让若干新进程按照需要处理不同的事情，就必须通过系统调用 exec（这实际上不止是一个名为 **exec** 的函数；而是 **exec** 通常用作一个引用一系列函数的通用术语，所有这些函数基本上都处理相同的事情，但是使用的参数稍微有些不同。）

来实现。

1. sys_execve( )

系统调用 exec 的内核服务程序是 sys_execve( )。该函数主要做两件事：

1．通过子函数 getname( )将可执行文件的名字从用户空间取入核心空间。详细流程和解释，参见第 5、6 章。
2．调用 do_execve( )执行具体的操作。

2. do_execve( )

do_execve( ) 是实现所有 exec 家族函数的底层内核函数。按照该函数的流程分析，它的执行过程如下：

（1） 打开可执行文件,获取该文件的 file 结构。
（2） 获取参数区长度,调用 memset()将存放参数的页面清零。memset()属于存储管理部分的函数。
（3） 对 linux_binprm 结构的其它项作初始化。

linux_binprm 结构用来读取并存储运行可执行文件的必要信息
该结构定义如下:

```
struct linux_binprm{
  char  buf[BINPRM_BUF_SIZE];  /*文件头缓冲区,BINPRM_BUF_SIZE=128*/
  unsigned long page[MAX_ARG_PAGES];/*存放参数页面的页表*/
  unsigned long p;                    /*参数区长度*/
  int sh_bang;
  struct file * file;
  int e_uid, e_gid;
  kernel_cap_t cap_inheritable, cap_permitted, cap_effective;
  int argc, envc;                     /*参数个数,环境个数*/
  char * filename;                 /*可执行文件路径名*/
  unsigned long loader, exec;
};
```

（4） 通过对参数和环境个数的计算来检查是否在这方面有错误，有错，则直接返回。
（5） 调用 prepare_binprm() 对数据结构 bprm 作进一步准备,并进行访问权限等内容的检测,从可执行文件中读开头的 128 个字节到 bprm 中的缓冲区。
（6） 把一些参数(文件名、环境变量、文件参数)从用户空间复制到核心空间。
（7） 调用 search_binary_handler()，搜寻目标文件的处理模块并执行,将返回值赋给 retval。

3.search_binary_handler( )

search_binary_handler()的功能是搜寻目标文件的处理模块（即二进制处理程序）并执行之。通过该函数新进程得以真正处理不同于父进程的事情。这个过程相当烦琐，具体的流程和解释请参见第 5、6 章。

二进制处理程序是 Linux 内核统一处理各种二进制格式的机制，这是我们需要的，因为不是所有的文件都是以相同的文件格式存储的。一个很合适的例子是 Java 的.class 文件。Java 定义了一种平台无关的二进制可执行格式——无论它们是在什么平台上运行，它们的文件本身都是相同的——因此这些文件显然应该和 Linux 特有的可执行格式一样构建。通过使用适当的二进制处理程序，Linux 可以把它们仿佛当作是自己特有的可执行文件一样处理。

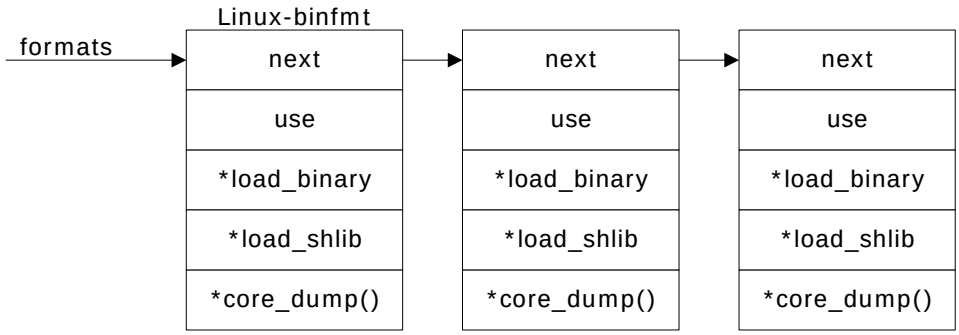
search_binary_handler()通过一个大循环，遍历 Linux_binfmt 结构链表来找出合适的二进制处理程序。

Linux_binfmt 结构定义如下：

```
struct linux_binfmt{
    struct linux_binfmt *next;
    long use_count;                          /*  使用计数*/
    int (*load_binary)( struct linux_binfmt,struct pt_regs regs);
                                             /*  指向二进制代码*/
    int (*load_shlib)(int fd);               /*  指向库函数*/
    int (*core_dump)long signr,struct pt_regs *regs);
  };
```

Linux_binfmt 结构中嵌入了两个指向函数的指针，一个指向可执行代码，另一个指向库函数，使用这两个指针是为了装入可执行代码和要使用的库。

Linux_binfmt 结构链表如下图所示：



Linux-binfmt链表

4.1.4 父进程的等待

当通过 do_fork()创建了一个子进程后，系统会执行调度操作，父进程和子进程都有机会获得 CPU。有时，需要让子进程先在 CPU 上运行，而让父进程转入睡眠

等待，直到子进程放弃 CPU。有两种方式可以让父进程变为睡眠态等待子进程。

第一种情况是：在执行 do_fork 时，若函数的参数中的 CLONE_VFORK 位置一，即父子进程共享用户空间。这时，必须先运行子进程。在 do_fork 中系统会对父进程执行一个 down( )操作，使父进程进入临界区并因为得不到临界资源而转入睡眠，从而达到等待子进程的目的。

第二种情况是在应用程序中直接利用调用系统调用 wait4( )来达到让父进程转入睡眠而等待子进程的目的。

1. sys_wait4( )

sys_wait4( )是系统调用 wait4( )的内核服务例程。父进程在执行该例程时，会在程序的 repeat 处睡眠，等待子进程唤醒。当子进程消亡时会唤醒父进程，父进程从 repeat 处继续执行为子进程处理善后工作，彻底消灭子进程。

sys_wait4( )的基本流程（具体的流程和解释参见第 5、6 章）：

（1） 在当前进程的系统堆栈上分配一个 wait_queue_t 结构

（2） 调用 add_wait_queue()把分配的 wait_queue_t 结构链入当前进程的 wait_chldexit 队列中。

（3） 由标号 repeat 进入循环，这个由 goto 实现的循环要到当前进程被调度运行，并且下列条件之一满足时才结束：

1. 所等待的子进程的状态变成暂停态或僵尸态；
2. 所等待的子进程存在，可不是上两个状态，而调用参数 options 中的 WNOHANG 标志置位，或当前进程收到了其它信号；
3. 进程号为 pid 的那个进程根本不存在，或不是当前进程的子进程。

否则，当前进程将其自身的状态设为可中断睡眠态，并调用 schedule( )进入睡眠而让其它进程先运行。当该进程因收到信号被唤醒，并且受到调度从 schedule( )返回时，再次循环扫描其子进程队列，察看所等待的子进程的状态是否满足条件。若满足条件，则消灭僵死的子进程的剩余资源。

4.1.5 子进程的消亡

进程的有两种消亡方式：

（1） 主动方式，自行消亡。

在前面举的应用程序的例子中，我们曾经提到 gcc 在编连程序的时候，会自动加入 exit 系统调用。这样，任何一个用户进程都不可避免的在运行结束时通过 exit 系统调用杀死自己。

（2） 被动方式，被强行消灭。

用户可以通过给进程发送信号量 9 强行杀掉进程。

自行消亡的子进程在调用 exit 后并不是彻底地在系统中消失了，而只是把自己的状态变为了僵尸态，其剩余资源（task_struct 结构等）还存在于系统中。真正要

清除子进程所有的资源，还需要父进程来处理。父进程先前在创建了子进程后，会通过系统调用 wait 睡眠。实际上它是在 sys_wait4（wait 调用的内核服务例程）中睡眠。当他被子（自行消亡）进程唤醒时，就在 sys_wait4 中清除子进程在系统中留下的所有资源。

1. sys_exit( )

sys_exit( )是系统调用 exit 对应的内核服务程序。它的函数体内仅有一条语句，即调用 do_exit( )完成具体操作。

2. do_exit( )及其调用的函数

首先，do_exit 判断表示是否有正在处理的中断服务全局变量 intr_count 是否为 1，如果为 1，表明当前还有中断正在处理，执行 intr_count=0，停止处理中断。

接着，do_exit 要为关闭系统，逐步退出一些运行操作系统所必须的模块。

(1) 把当前任务的标记记为退出：

current->flags |= PF_EXITING;

向所有的进程宣布，现在系统要退出了，以便一些调度处理函数得知这一消息（通过检测该标记）。

(2) 删除当前的实定时器：

del_timer(&current->real_timer);

(3) 删除信号队列（destroye semaphore arrays），释放信号撤消结构（free semaphores undo structures）

Sem_exit（）;（定义在/linux/ipc/sem.c 文件中）

在该函数中，增加调整值（semval）给信号，再释放撤消结构（free undo structures）。由于某些信号（semaphore）可能已经过时或无效了，直到信号数组(semaphore array)被删除了以后，撤消结构（undo structures）才被释放。具体做法如下：

a. 如果当前进程正在睡眠状况（需要一信号(semaphore)来唤醒），将进程当前指向所需信号（semaphore）的指针置空。

b. 在当前的信号撤消链表(struct sem_undo)里查找 a 中提到的那信号（semaphore），找到以后，调整该信号（已在信号撤消链表中注册过的）的内容。

c. 由于有可能有一个队列的进程在等该信号，故须更新整个操作系统的数组。

(4) 退出内存管理系统：

__exit_mm(current);（函数定义在/linux/kernel/exit.c 文件中）

具体做法如下：

a. 将 cache，tlb，page 里的内容全部回写。

b. 退出内存影射。

c. 释放页表（page table）。

(5) 把当前任务所打开的文件都关闭，释放文件指针。

__exit_files(current);（函数定义在/linux/kernel/exit.c 文件中）

(6) 退出文件系统：

__exit_fs(current);（函数定义在/linux/kernel/exit.c 文件中）

(7) 释放当前任务的所有信号（signal）：

__exit_sighand(current);（函数定义在/linux/kernel/exit.c 文件中）

(8) 释放当前线程数据：

__exit_thread();

(9) 调用 exit_notify()

通知当前进程的父进程和子进程,它要消亡了。并将子进程托付给其它进程。具体的流程和解释参见第 5、6 章。

(10) 将当前任务的用户数目减一：

(*current->exec_domain->use_count)--;

(*current->binfmt->use_count)--;

(11) 继续调度：

schedule();

至此全部完成。

3. exit_notify( )

自行消亡的子进程在 do_exit ( )中调用 exit_notify( )告知其父子进程它要消亡的消息，并把它的子进程托付给其它进程。

简要流程：

（1） 将所有原始进程为 current 的进程变成 init 进程的孙子。

（2） 如果父进程和 current 进程不在同一组，但在同一 session 内并且 current 进程组内所有进程的父进程和它在同一组，也就是说，current 进程所在组会因 current 的退出而悬挂，同时 current 进程所在组内有 stopped 进程，就向整个组发 SIGHUP 和 SIGCONT 信号。

（3） 通知父进程子进程消亡了。

（4） 调整所有 current 进程的子进程的父进程指针，将它们挂到它们的原始进程下，将以往的跟踪被跟踪历史清除，调整它和新的兄弟的关系；检查每一个 current 进程的子进程所在的组是否会悬挂，如果子进程和 current 进程不在同一组，并且这个组已悬挂，组内有 stopped 的进程，就向组员发 SIGHUP 和 SIGCONT 信号。

（5） 如果 current 进程是 session 的主管， 就和它所控制的 tty 脱离，向 current 进程显示终端所在的组发 SIGHUP 和 SIGCONT 信号。

4.1.6 进程控制的全过程

通过 4.1 的各小节，可以把通过系统调用创建、运行、消灭子进程的整个过程描述如下（以 4.1 节的演示程序为例）：

当一个父进程要创建子进程的时候，他调用系统调用函数 fork( ),从系统调用入口 ENTRY(system_call)执行系统调用总控部分的代码进入内核，然后调用 sys_fork( ),进而调用 do_fork( )创建一个子进程。子进程创建完成后，转入调度。无论是父进程获得 CPU 还是子进程获得 CPU，都会回到系统调用总控部分的代码的

ret_from_sys_call，从这里进程返回到用户态继续执行用户程序。若从子进程返回，则返回值为 0，继续执行系统调用 execve( )按前述方式进入内核执行 sys_execve( )，进而调用 do_execve( )装入可执行文件并执行之。若从父进程返回，则返回值为子进程 pid，继续执行系统调用 wait( ) 按前述方式进入内核执行 sys_wait( )转入睡眠，等待子进程消亡时唤醒。子进程完成任务后通过系统调用 exit( )进入内核执行 do_exit( )自我消亡，并唤醒等待它的父进程。父进程唤醒后，在 sys_wait( )中清除子进程的剩余资源，把子进程从系统中完全清除。

4.2 对于 Linux 进程控制的研究总结

4.2.1 Linux 系统的内核保护与系统调用

Linux 系统把代码的运行级别分为 0 级和 3 级。0 级权限高于 3 级。处在内核空间的 Linux 内核代码为 0 级，处在用户空间的应用程序代码为 3 级。

属于 3 级的用户程序只有通过系统调用这一条路才能进入内核执行相应的服务例程。这样 Linux 可以被有效的保护起来，而不会发生因为用户进程非法进入系统内核而造成系统崩溃的情况。

4.2.2 Linux 进程的创建、执行、等待和消亡

这部分的设计思想源于 UNIX，方法与 UNIX 基本相同。系统中的进程形成一张按家族体系联系在一起的网，通过这张网，可以从一个访问到任何一个其它进程。

新创建的进程最初只是父进程的副本，共享或复制父进程的资源，这与 UNIX 及其类似。

直得一提的是，Linux 允许新创建的子进程与父进程共享虚存资源。进程的虚存可能在物理内存，可能在兑换区，也可能在执行状态。所以，Linux 采用了“写时拷贝”(Copy On Write)的方法,即父子进程一开始总是共用同一份虚存空间，只有当其中一个准备往里写东西的时候系统才复制一份。

4.2.3 Linux 进程的并发特性

在 Linux 中，当一个父进程通过系统调用创建了一个子进程后，父子进程将同时参与进程调度，即父子进程都有有获得 CPU 的机会。Linux 会根据父子进程（还有其他处于就绪态的进程）的优先权交替的执行他们。虽然每一时刻只有一个进程在运行，而从宏观上看就好像是同时执行多个进程一样（即并发执行进程）。Linux 通过这种并发执行进程的特性，可以使各个任务得到合理地、及时地处理，提高系统的整体工作效能。

第 5 章　Linux 内核源码模块功能

内核源码以 2.4.0 版本为准

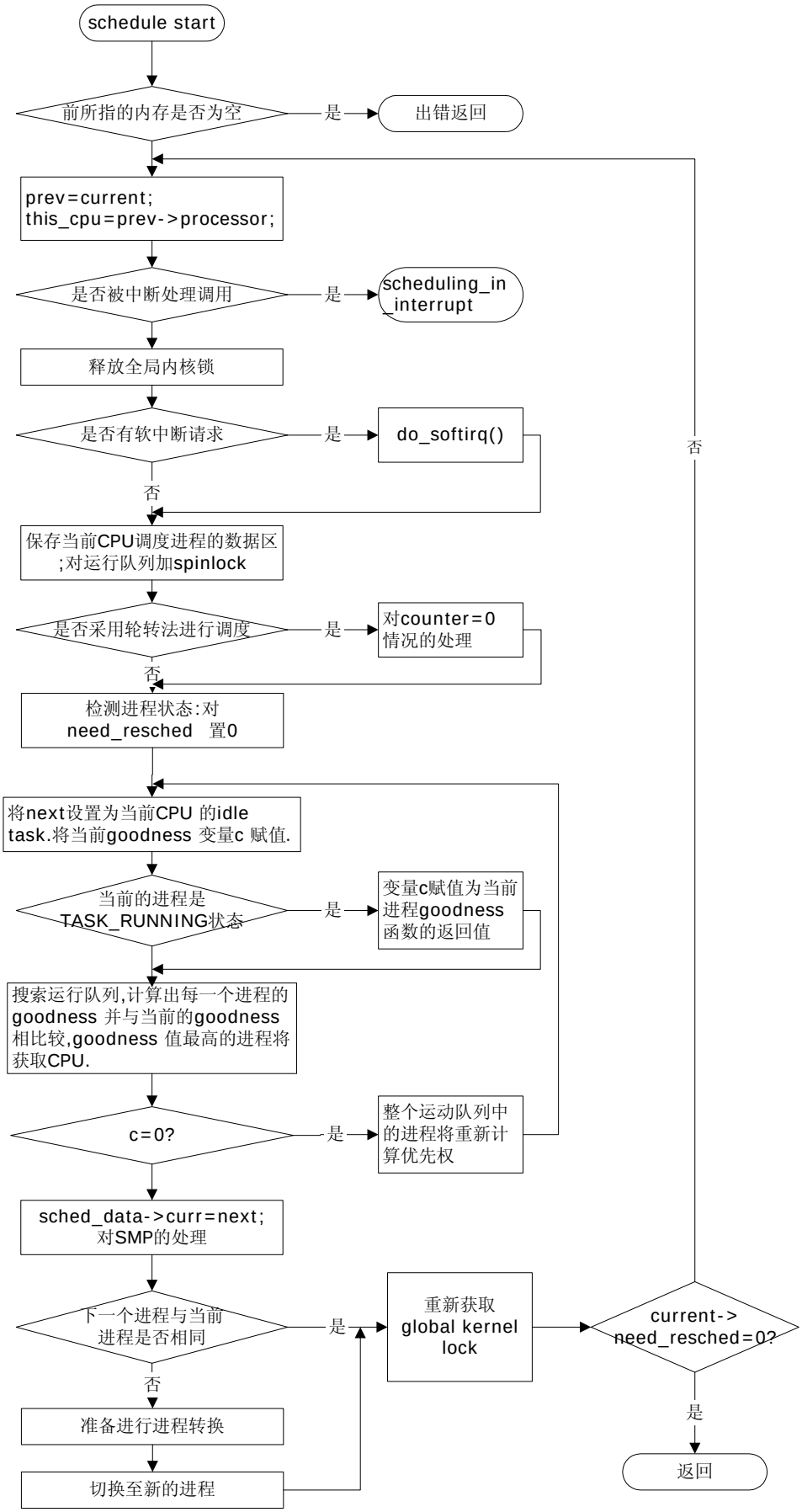
5.1 进程调度

5.1.1 schedule ( )

位置:kernel/sched.c
形式:asmlinkage void schedule(void)
参数:无
功能: 调度进程
流程:

1. 当前进程所指的内存若为空，则出错返回。
2. 令 prev=current;this_cpu=prev->processor;
3. 若是中断服务程序调用 schedule(),跳转至 scheduling_in_interrupt。
4. 释放全局内核锁。
5. 若有 bottom half 服务请求，调用 do_bottom_half()。
6. 保存当前 CPU 调度进程的数据区，对运行队列加 spinlock。
7. 若采用轮转法进行调度，对 counter=0 的情况的进行处理。
8. 检测进程状态:对 need_resched　置 0。
9. 将 next 设置为当前 CPU 的 idle task。将当前 goodness 变量 c 赋值。
10. 若当前的进程是 TASK_RUNNING 状态,则变量 c 赋值为当前进程 goodness 函数的返回值。
11. 搜索运行队列,计算出每一个进程的 goodness 并与当前的 goodness 相比较,goodness 值最高的进程将获取 CPU。
12. 若运行队列中所有进程的时间片都耗尽，则对系统中的所有进程重新分配时间片。转回步骤 9。
13. 令 sched_data->curr=next。
14. 若下一个进程与当前进程不同，则执行下一步骤；否则，跳转至步骤 17。
15. 准备进行进程转换。
16. 切换至新的进程。
17. 重新获取 global kernel lock。
18. 若当前进程的重调度标志为零，则返回，退出；否则，跳转至步骤 2。

流程图见下页。

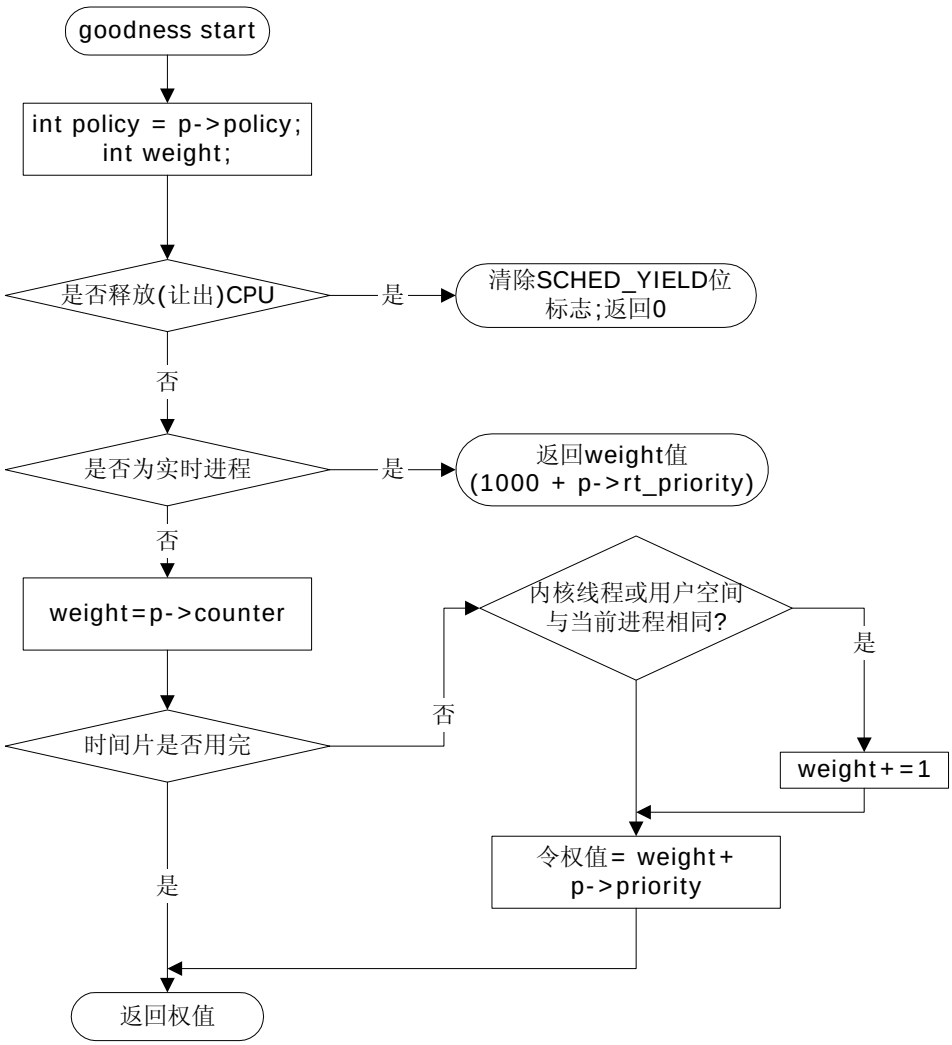


schedule()函数的流程图

5.1.2 goodness （ ）

位置:kernel/sched.c
形式: static inline int goodness(struct task_struct * p, struct task_struct * prev, int this_cpu)
参数: p,prev 两个进程的 task_struct 结构指针, this_cpu 当前 CPU
功能: 调度进程
流程:

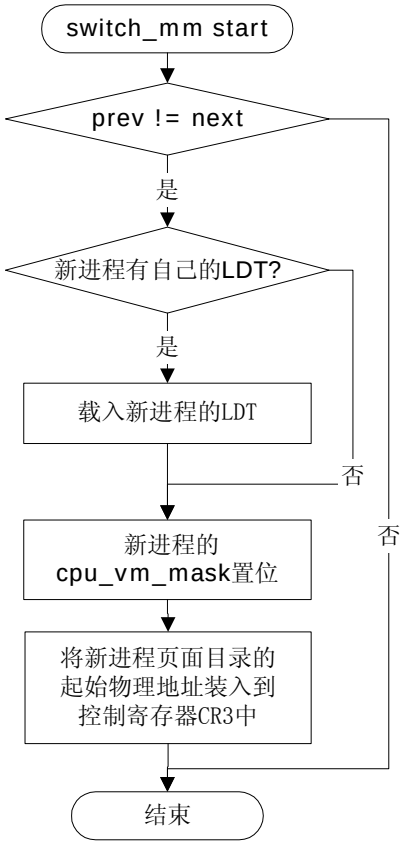
1. 若当前进程释放(让出)CPU，则清除 SCHED_YIELD 位标志，返回 0，退出。
2. 若为实时进程，则返回 weight 值(1000 + p->rt_priority)。
3. 令 weight=p->counter。
4. 若时间片用完，则返回权值，退出。否则执行下一步骤。
5. 若内核线程或用户空间与当前进程相同，则令 weight+=1；否则令权值等于 weight+p->priority。
6. 返回权值，退出。



5.1.3 switch_mm ( )

位置:include/asm_i386/mmu_context.h*/
形式:static inline void switch_mm(struct mm_struct *prev, struct mm_struct *next,
unsigned cpu)
参数: p,prev 两个进程的 task_struct 结构指针, cpu 当前 CPU
功能: 载入新进程的 LDT
流程:

1. 若 prev != next，执行下一步骤，否则结束，退出。
2. 若新进程有自己的 LDT，则载入新进程的 LDT。
3. 将新进程的 cpu_vm_mask 置位。
4. 将新进程页面目录的起始物理地址装入到控制寄存器 CR3 中。
5. 结束，退出。



5.1.4 add_to_runqueue ( )

位置:kernel/sched.c
形式:static inline void add_to_runqueue(struct task_struct * p)
参数: p 进程的 task_struct 结构指针
功能:将一个进程的 task_struct 结构链入运行队列。

流程：

1. 令 next 指向 init_task 进程的下一个进程。
2. 令 p 进程的向前指针指向 init_task 进程。
3. 令 init_task 进程的向后指针指向 p 进程。
4. 令 p 进程的向后指针指向 init_task 进程先前的下一个进程。
5. 令 next 指向的进程的向前指针指向 p 进程。
6. 运行队列进程计数增一。

5.1.5 del_from_runqueue ( )

位置:kernel/sched.c
形式: static inline void del_from_runqueue(struct task_struct * p)
参数: p 进程的 task_struct 结构指针
功能:将一个进程的 task_struct 结构从运行队列摘除
流程：

1. 令 next 指向 p 进程的下一个进程。
2. 令 prev 指向 p 进程的上一个进程。
3. 运行队列进程计数减一。
4. 令 next 进程的向前指针指向 prev 进程。
5. 令 prev 进程的向后指针指向 next 进程。
6. 令 p 进程的向后指针为空。
7. 令 p 进程的向前指针为空。

5.1.6 move_last_runqueue ( )

位置:kernel/sched.c
形式: static inline void move_last_runqueue(struct task_struct * p)
参数: p 进程的 task_struct 结构指针
功能:将 p 进程的 task_struct 移到运行队列尾部
流程：

1. 令 next 指向 p 进程的下一个进程。
2. 令 prev 指向 p 进程的上一个进程。
3. 先将 p 进程从队列中摘除。
4. 将 p 进程加到队尾。

5.1.7 move_first_runqueue ( )

位置:kernel/sched.c
形式: static inline void move_first_runqueue(struct task_struct * p)
参数: p 进程的 task_struct 结构指针
功能:将 p 进程移到运行队列首部
流程：

1. 令 next 指向 p 进程的下一个进程。
2. 令 prev 指向 p 进程的上一个进程。
3. 先将 p 进程从队列中摘除。
4. 将 p 进程加到队首。

5.2 nanosleep,pause 及时钟函数

5.2.1 sys_nanosleep （ ）

位置:kernel/sched.c
形式:asmlinkage int sys_nanosleep(struct timespec *rqtp, struct timespec *rmtp)
参数: rqtp timespec 结构指针,
指向所需睡眠时间的 timespec 结构 rmtp timespec 结构指针,
指向返回剩余睡眠时间的 timespec 结构
功能: 使进程睡眠一段时间
流程:

1. 调用 copy_from_user()将 rgtp 指向的 timespec 结构复制给 t。若复制失败，则 return －EFAULT，退出。
2. 若睡眠时间设置不正确，则 return -EINVAL，退出。
3. 若实时进程且睡眠时间小于 2 毫秒，则作延时操作,并不真正睡眠，返回 0，退出。
4. 把睡眠时间换算成时钟中断的次数设当前进程为可中断睡眠态。
5. 调用 schedule_timeout()进入睡眠,醒来时将剩余的睡眠时间赋 给 expire。
6. 若 expire=0，返回 0，退出。否则，判断 rmtp 是否为空，若为空则 return －EINTR，退出；若非空，则执行下一步骤。
7. 调用 jiffies_to_timespec()将剩余时间赋给 t 所指向的 timespec 结构。
8. 调用 copy_from_user()将 t 的值复制到 rmtp。若复制成功，则 return －EINTR，退出；若复制失败，则 return －EFAULT，退出。

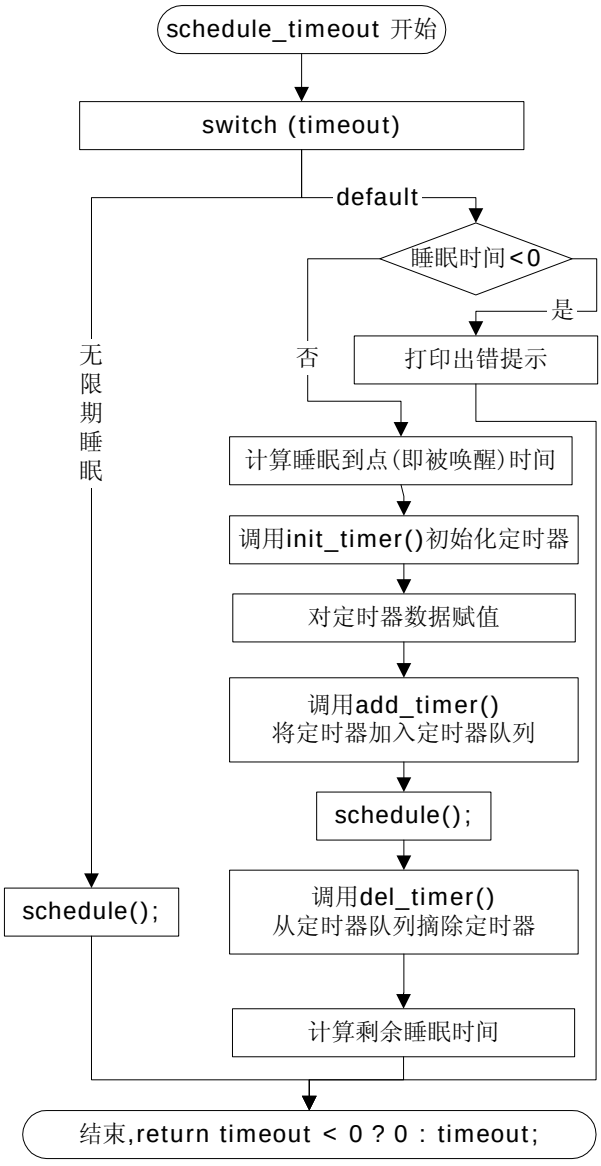
流程图见后面。

5.2.2 schedule_timeout （ ）

位置:kernel/sched.c
形式: signed long schedule_timeout(signed long timeout)
参数: timeout 睡眠时间
功能: 使进程睡眠一段时间,并转入调度
流程:

1. 执行开关语句 switch (timeout)。若为无限期睡眠，跳转至步骤 11；若为 default，执行下一步骤。
2. 若睡眠时间<0，则打印出错提示，转至步骤 12。
3. 计算睡眠到点(即被唤醒)时间。
4. 调用 init_timer()初始化定时器。
5. 对定时器数据赋值。
6. 调用 add_timer()将定时器加入定时器队列。
7. 执行 schedule()。
8. 调用 del_timer()

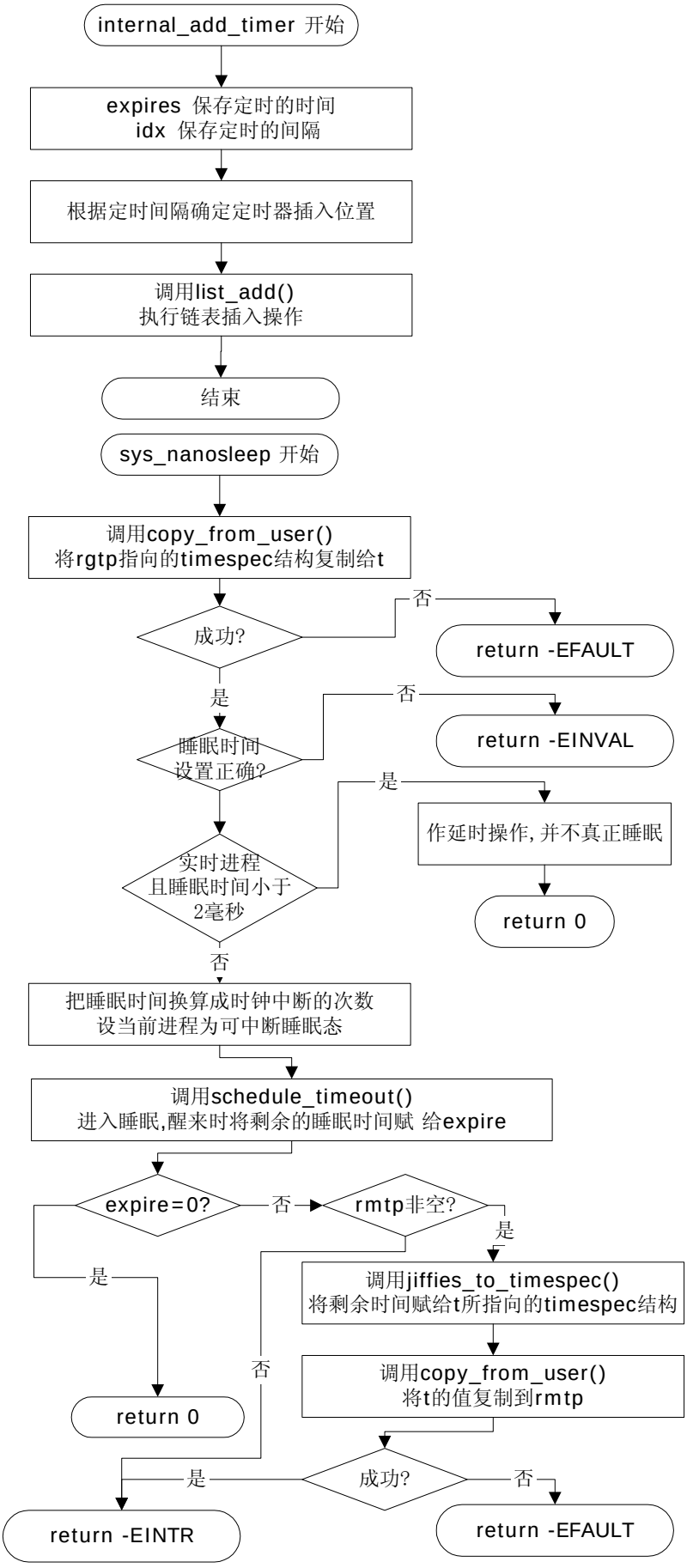
9. 从定时器队列摘除定时器。
10. 计算剩余睡眠时间，转至步骤 12。
11. 执行 schedule()。
12. 结束,return timeout < 0 ? 0 : timeout，结束。



5.2.3 internal_add_timer ( )

位置:kernel/timer.c
形式:static inline void internal_add_timer(struct timer_list *timer)
参数: timer 定时器指针
功能: 将定时器链入队列
流程:

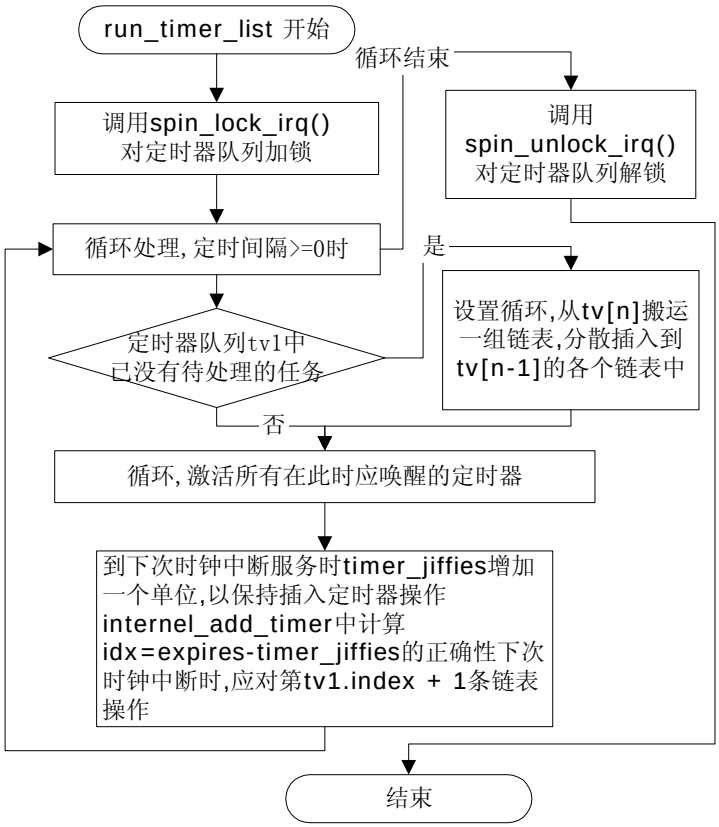
1. 根据定时间隔确定定时器插入位置。
2. 调用 list_add()执行链表插入操作。
3. 结束，退出。


internal_add_timer 开始
expires 保存定时的时间
idx 保存定时的间隔
根据定时间隔确定定时器插入位置
调用list_add()
执行链表插入操作
结束
sys_nanosleep 开始
调用copy_from_user()
将rgtp指向的timespec结构复制给t
成功?
否
return -EFAULT
是
睡眠时间
设置正确?
否
return -EINVAL
是
作延时操作, 并不真正睡眠
return 0
实时进程
且睡眠时间小于
2毫秒
是
否
把睡眠时间换算成时钟中断的次数
设当前进程为可中断睡眠态
调用schedule_timeout()
进入睡眠,醒来时将剩余的睡眠时间赋 给expire
expire=0?
否
rmtp非空?
是
是
调用jiffies_to_timespec()
将剩余时间赋给t所指向的timespec结构
否
调用copy_from_user()
将t的值复制到rmtp
return 0
是
成功?
否
return -EINTR
return -EFAULT

5.2.4 run_timer_list ( )

位置:kernel/timer.c
形式: static inline void run_timer_list(void)
参数:无
功能:激活到点定时器
流程:

1. 调用 spin_lock_irq()对定时器队列加锁。
2. 循环处理,条件为定时间隔>=0。循环结束后，跳转步骤 6。
3. 若定时器队列 tv1 中已没有待处理的任务，则设置循环,从 tv[n]搬运一组链表,分散插入到 tv[n-1]的各个链表中。
4. 循环,激活所有在此时应唤醒的定时器。
5. 到下次时钟中断服务时 timer_jiffies 增加一个单位,以保持插入定时器操作 internel_add_timer 中计算 idx=expires-timer_jiffies 的正确性下次时钟中断时,应对第 tv1.index + 1 条链表操作。转回步骤 2。
6. 调用 spin_unlock_irq()对定时器队列解锁。
7. 结束，退出。

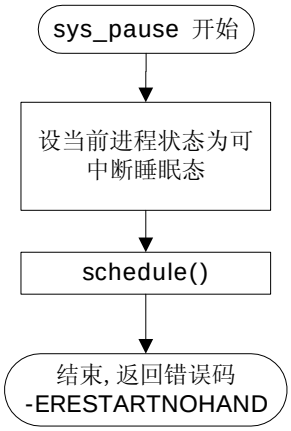


5.2.5 sys_pause()

位置 arch/i386/kernel/sys_i386.c
形式:asmlinkage int sys_pause(void)

参数:无
功能:中止进程
流程:

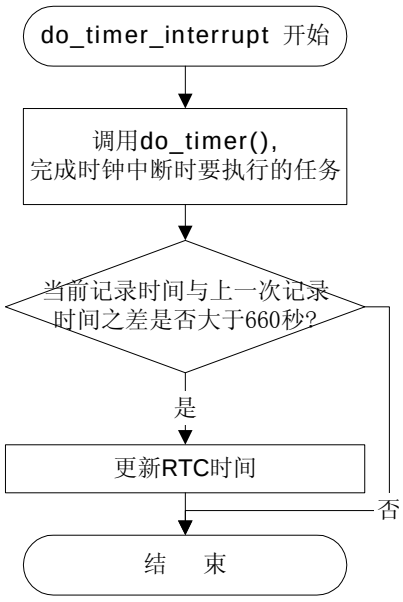
1. 设当前进程状态为可中断睡眠态。
2. 调用 schedule( )转入进程调度。
3. 进程重新获得 CPU 后，退出 sys_pause（）。



5.2.6 do_timer_interrupt ()

位置:arch/i386/kernel/time.c
形式:static inline void do_timer_interrupt(int irq, void *dev_id, struct pt_regs *regs)
参数: irq , dev_id 设备号,regs pt_regs 结构指针
功能:调用 do_timer() 执行时钟中断
流程:

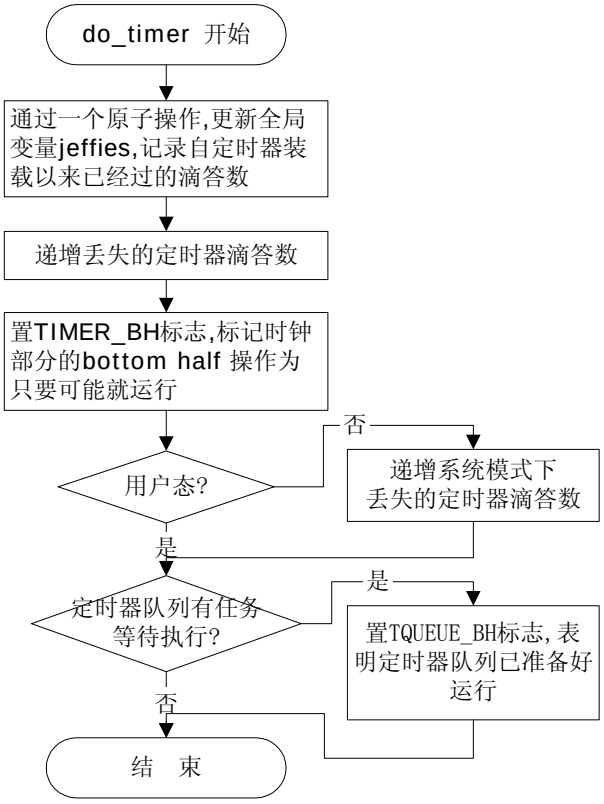
1. 用 do_timer( ),完成发生时钟中断时要执行的任务。
2. 断当前记录时间与上一次记录时间之差是否大于 660 秒，若是则更新 RTC 时间后结束，若不是则直接退出函数。



5.2.7 do_timer ()

位置:kernel/sched.c
形式:void do_timer(struct pt_regs * regs)
参数: regs pt_regs 结构指针
功能: 执行时钟中断
流程:

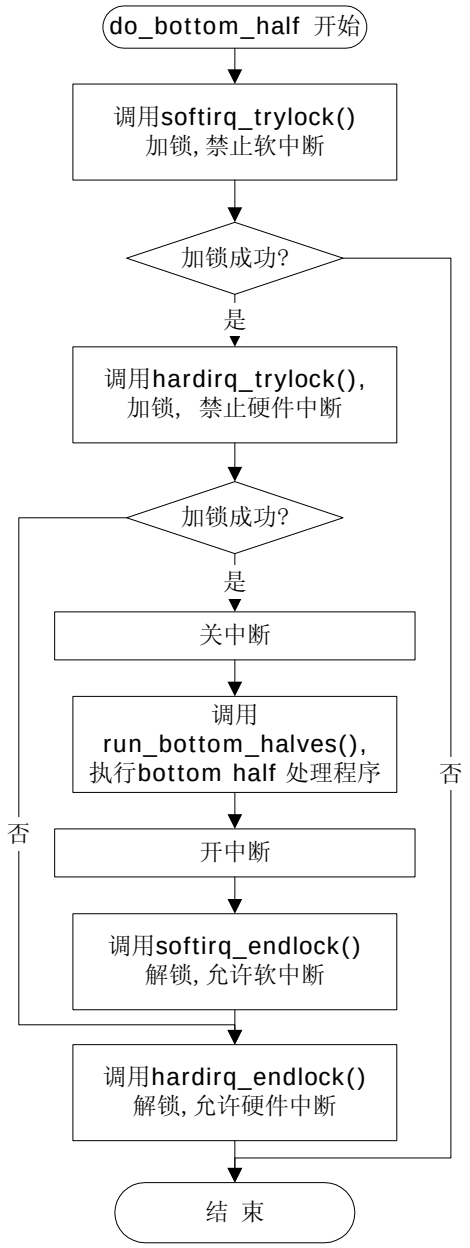
1. 通过一个原子操作,更新全局变量 jeffies,记录自定时器装载以来已经过的滴答数。
2. 递增丢失的定时器滴答数。
3. 置 TIMER_BH 标志,标记时钟部分的 bottom half 操作为只要可能就运行。
4. 若不是用户态，则递增系统模式下丢失的定时器滴答数。
5. 若定时器队列有任务等待执行，则置 TQUEUE_BH 标志,表明定时器队列已准备好运行。
6. 结束，退出。



5.2.8 do_bottom_half ()

位置:kernel/softirq.c*/
形式:asmlinkage void do_bottom_half(void)
参数: 无
功能: 执行 bottom half 处理程序
流程:

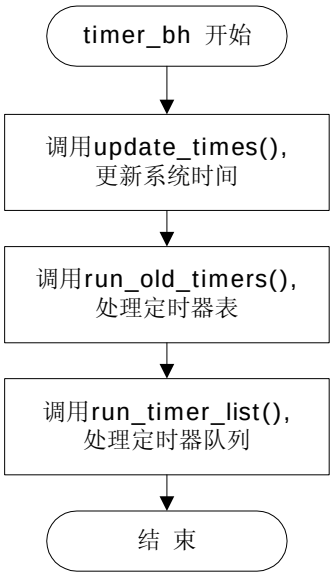
1. 调用 softirq_trylock()加锁,禁止软中断。若加锁失败，则结束，退出。
2. 调用 hardirq_trylock()，加锁， 禁止硬件中断。若加锁失败，则跳转至步骤 7。
3. 关中断。
4. 调用 run_bottom_halves(),执行 bottom half 处理程序。
5. 开中断。
6. 调用 softirq_endlock()解锁,允许软中断。
7. 调用 hardirq_endlock()解锁,允许硬件中断。
8. 结束，退出。



5.2.9 timer_bh ()

位置:kernel/sched.c
形式:static void timer_bh(void)
参数:无
功能: 执行定时器 bottom half 处理
流程:

1. 调用 update_times(),更新系统时间。
2. 调用 run_old_timers(),处理定时器表。
3. 调用 run_timer_list(),处理定时器队列。
4. 结束，退出。



5.3 系统调用总控入口

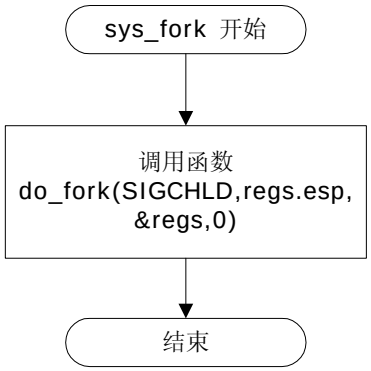
ENTRY(system_call)
位置: arch/i386/kernel/Entry.S
形式: AT&T 汇编代码,以 ENTRY(system_call)为入口
功能:提供系统调用从用户态进入核心态服务,同时系统调用执行结束后也由此段代码返回.
流程：这是一段 AT&T 汇编代码，请见第 6 章源码注释。

5.4 子进程的创建

5.4.1 sys_fork( )

位置:arch/i386/kernel/process.c
形式:asmlinkage int sys_fork(struct pt_regs regs)
参数:regs,pt_regs 结构,可用此结构访问堆栈中的相应内容
功能:调用 do_fork()完全复制出一个子进程
流程:

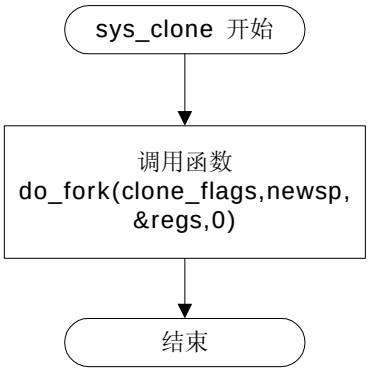
1. 调用 do_fork( )函数执行创建进程的具体操作。
2. 结束。



5.4.2 sys_clone( )

位置:arch/i386/kernel/process.c
形式:asmlinkage int sys_clone(struct pt_regs regs)
参数:regs,pt_regs 结构,可用此结构访问堆栈中的相应内容
功能:调用 do_fork()复制出一个子进程
流程:

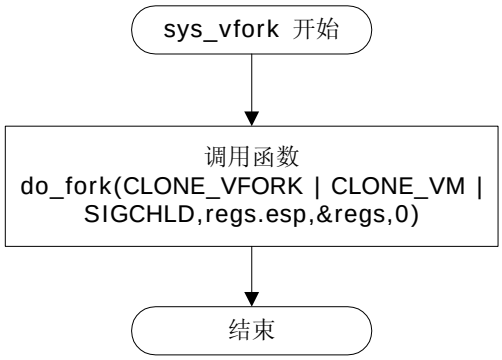
1. 用 do_fork( )函数执行创建进程的具体操作。
2. 结束。



5.4.3 sys_vfork( )

位置:arch/i386/kernel/process.c
形式:asmlinkage int sys_vfork(struct pt_regs regs)
参数:regs,pt_regs 结构,可用此结构访问堆栈中的相应内容
功能:调用 do_fork()复制出一个子进程,主要用于进程创建的中间步骤
流程:

1. 用 do_fork( )函数执行创建进程的具体操作。
2. 结束。

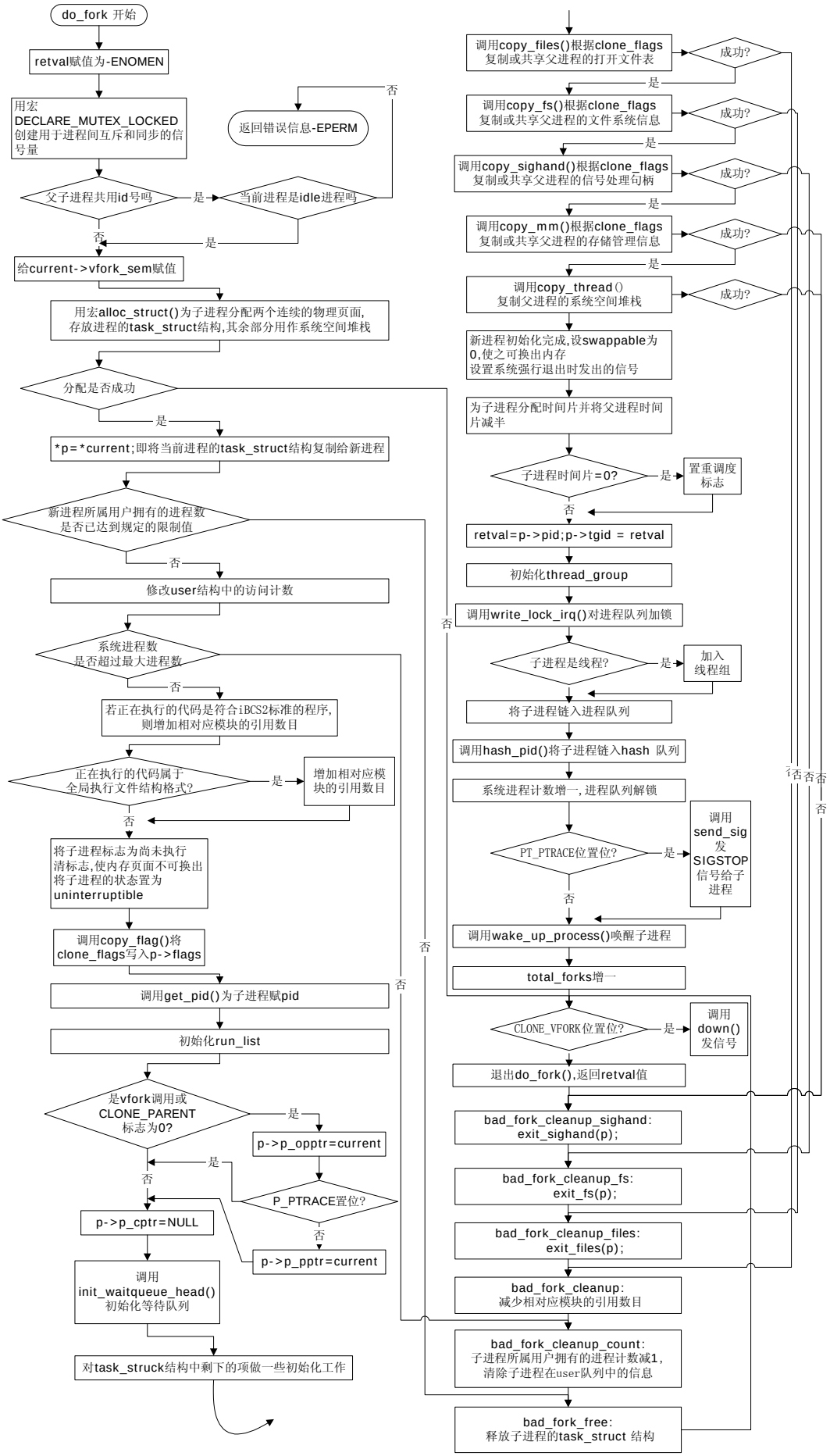


5.4.4 do_fork( )

位置:kernel/fork.c
形式:int do_fork(unsigned long clone_flags, unsigned long stack_start,
struct pt_regs *regs, unsigned long stack_size)
参数: clone_flags 标志,用于标明新创建进程与父进程共享资源的方式,
stack_start 新进程堆栈的开始地址,regs pt_regs 结构, stack_size 堆栈大小.
功能: 创建一个进程
流程:

1. retval 赋值为-ENOMEN。
2. 用宏 DECLARE_MUTEX_LOCKED 创建用于进程间互斥和同步的信号量。
3. 判断父子进程是否共用 id 号,若共用且当前进程是 idle 进程则出错返回;若不共用 id 号则继续向下执行。
4. 给 current->vfork_sem 赋值。
5. 用宏 alloc_struct()为子进程分配两个连续的物理页面,存放进程的 task_struct 结构,其余部分用作系统空间堆栈。若分配失败则跳转至 fork_out 退出 do_fork(),返回 retval 值。
6. *p=*current;即将当前进程的 task_struct 结构复制给新进程。
7. 若系统进程数超过最大进程数，则跳转至 bad_fork_cleanup_count，对子进程所属用户拥有的进程计数减 1,清除子进程在 user 队列中的信息。释放子进程的 task_struct 结构，退出 do_fork(),返回 retval 值。
8. 若系统进程数未超过最大进程数，则增加相对应模块的引用数目。
9. 将子进程标志为尚未执行清标志,使内存页面不可换出将子进程的状态置为 uninterruptible。

10. 调用 copy_flag()将 clone_flags 写入 p->flags。
11. 调用 get_pid()为子进程赋 pid。
12. 初始化 run_list。
13. 若是 vfork 调用 do_fork( )或 CLONE_PARENT 标志为 0,则 p->p_opptr=current；继续判断 P_PTRACE 是否置位，若未置位，则 p->p_pptr=current。
14. 令 p->p_cptr=NULL。
15. 调用 init_waitqueue_head()初始化等待队列。
16. 对 task_struck 结构中剩下的项做一些初始化工作。
17. 调用 copy_files()根据 clone_flags 复制或共享父进程的打开文件表。若不成功则跳转至 bad_fork_cleanup 减少相对应模块的引用数。
18. 调用 copy_fs()根据 clone_flags 复制或共享父进程的文件系统信息。若不成功则跳转至 bad_fork_cleanup_files 调用 exit_files()释放文件系统信息。
19. 调用 copy_sighand()根据 clone_flags 复制或共享父进程的信号处理句柄。若不成功则跳转至 bad_fork_cleanup_fs。
20. 调用 copy_mm()根据 clone_flags 复制或共享父进程的存储管理信息。若不成功则跳转至 bad_fork_cleanup_sighand。
21. 调用 copy_thread()复制父进程的系统空间堆栈。若不成功则跳转至 bad_fork_cleanup_mm。
22. 新进程初始化完成,设 swappable 为 0,使之可换出内存设置系统强行退出时发出的信号。
23. 为子进程分配时间片并将父进程时间片减半。
24. 判断子进程时间片是否为零，若为零则置重调度标志。
25. 令 retval=p->pid;p->tgid = retval。
26. 初始化 thread_group。
27. 调用 write_lock_irq()对进程队列加锁。
28. 若子进程是线程，则将其加入线程组。
29. 将子进程链入进程队列。
30. 调用 hash_pid()将子进程链入 hash 队列。
31. 系统进程计数增一,进程队列解锁。
32. 若 PT_PTRACE 位置位，则调用 send_sig( )发 SIGSTOP 信号给子进程。
33. 调用 wake_up_process()唤醒子进程。
34. total_forks 增一。
35. 若 CLONE_VFORK 位置位，则调用 down()发信号给父进程。
36. 退出 do_fork(),返回 retval 值。


do_fork 开始
retval赋值为-ENOMEN
用宏DECLARE_MUTEX_LOCKED创建用于进程间互斥和同步的信号量
父子进程共用id号吗
是
当前进程是idle进程吗
否
返回错误信息-EPERM
否
是
给current->vfork_sem赋值
用宏alloc_struct()为子进程分配两个连续的物理页面,存放进程的task_struct结构,其余部分用作系统空间堆栈
分配是否成功
是
*p=*current;即将当前进程的task_struct结构复制给新进程
新进程所属用户拥有的进程数是否已达到规定的限制值
否
修改user结构中的访问计数
系统进程数是否超过最大进程数
否
若正在执行的代码是符合iBCS2标准的程序,则增加相对应模块的引用数目
正在执行的代码属于全局执行文件结构格式?
是
增加相对应模块的引用数目
否
将子进程标志为尚未执行清标志,使内存页面不可换出将子进程的状态置为uninterruptible
调用copy_flag()将clone_flags写入p->flags
调用get_pid()为子进程赋pid
初始化run_list
是vfork调用或CLONE_PARENT标志为0?
是
p->p_opptr=current
是
否
P_PTRACE置位?
否
p->p_pptr=current
p->p_cptr=NULL
调用init_waitqueue_head()初始化等待队列
对task_struck结构中剩下的项做一些初始化工作
调用copy_files()根据clone_flags复制或共享父进程的打开文件表
成功?
是
调用copy_fs()根据clone_flags复制或共享父进程的文件系统信息
成功?
是
调用copy_sighand()根据clone_flags复制或共享父进程的信号处理句柄
成功?
是
调用copy_mm()根据clone_flags复制或共享父进程的存储管理信息
成功?
是
调用copy_thread()复制父进程的系统空间堆栈
成功?
新进程初始化完成,设swappable为0,使之可换出内存设置系统强行退出时发出的信号
为子进程分配时间片并将父进程时间片减半
子进程时间片=0?
是
置重调度标志
否
retval=p->pid;p->tgid = retval
初始化thread_group
调用write_lock_irq()对进程队列加锁
子进程是线程?
是
加入线程组
将子进程链入进程队列
调用hash_pid()将子进程链入hash 队列
系统进程计数增一,进程队列解锁
PT_PTRACE位置位?
是
调用send_sig发SIGSTOP信号给子进程
否
调用wake_up_process()唤醒子进程
total_forks增一
CLONE_VFORK位置位?
是
调用down()发信号
退出do_fork(),返回retval值
bad_fork_cleanup_sighand:
exit_sighand(p);
bad_fork_cleanup_fs:
exit_fs(p);
bad_fork_cleanup_files:
exit_files(p);
bad_fork_cleanup:
减少相对应模块的引用数目
bad_fork_cleanup_count:
子进程所属用户拥有的进程计数减1,清除子进程在user队列中的信息
bad_fork_free:
释放子进程的task_struct 结构
否
否
否
否
否
否
否

5.4.5 get_pid( )

位置:kernel/fork.c
形式:static int get_pid(unsigned long)
参数: flags 判断标志
功能: 获取进程 pid
流程:

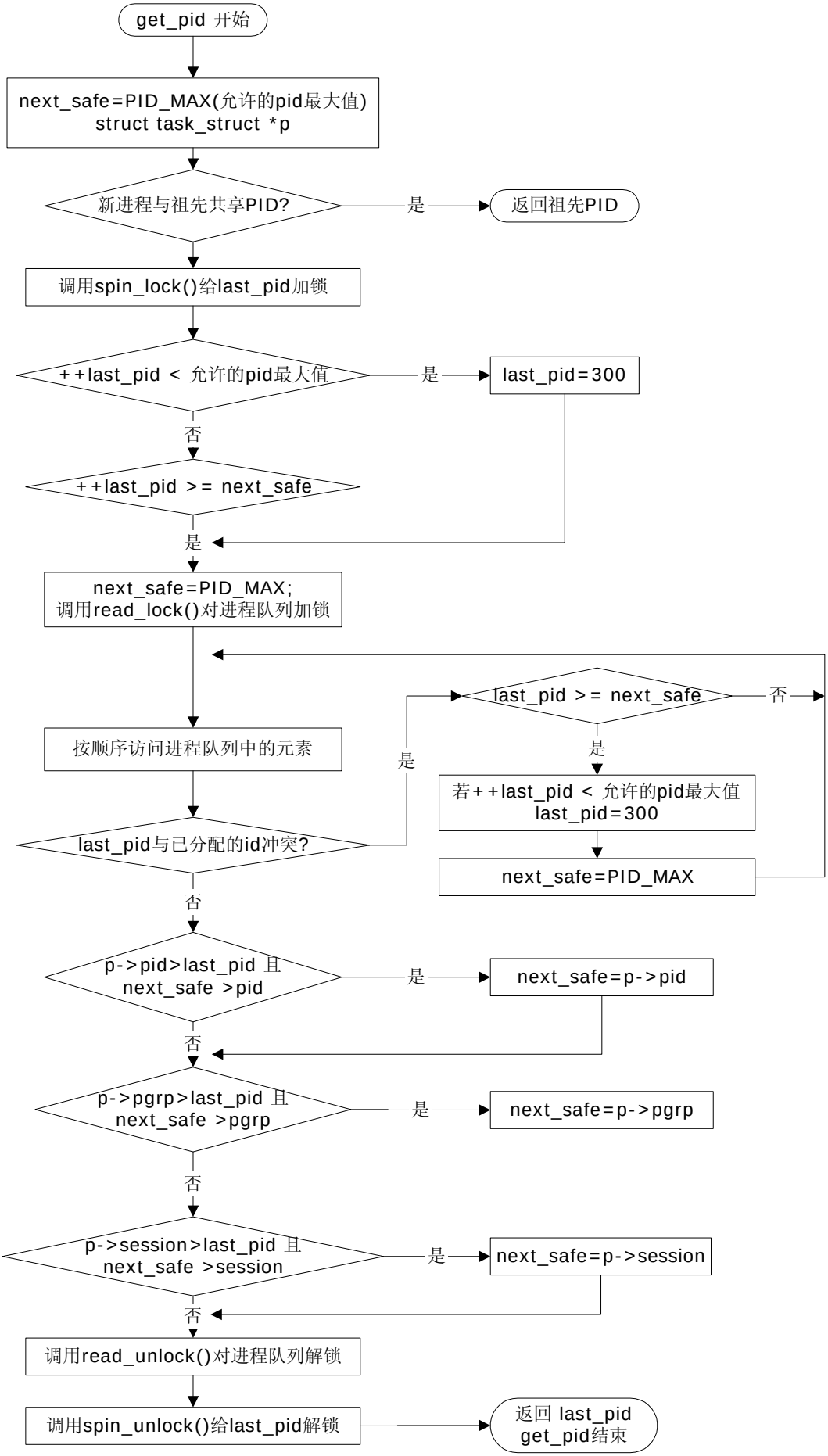
1. 令 next_safe=PID_MAX(允许的 pid 最大值)。
2. 若新进程与祖先共享 PID，则返回祖先 PID,退出函数。
3. 调用 spin_lock()给 last_pid 加锁。
4. ++last_pid 小于允许的 pid 最大值，则令 last_pid=300。
5. 令 next_safe=PID_MAX;调用 read_lock()对进程队列加锁。
6. 按顺序访问进程队列中的元素。
7. 若 last_pid 与已分配的 id 冲突且 last_pid >= next_safe，则判断 ++last_pid 是否小于允许的 pid 最大值，若是，令 last_pid=300; next_safe=PID_MAX; 然后重复第 6 步。
8. 若 p->pid>last_pid 且 next_safe >pid，令 next_safe=p->pid。
9. 若 p->pgrp>last_pid 且 next_safe >pgrp，令 next_safe=p->pgrp。
10. 若 p->session>last_pid 且 next_safe >session ，令 next_safe=p->session。
11. 调用 read_unlock()对进程队列解锁。
12. 调用 spin_unlock()给 last_pid 解锁。
13. 返回 last_pidget_pid 结束。

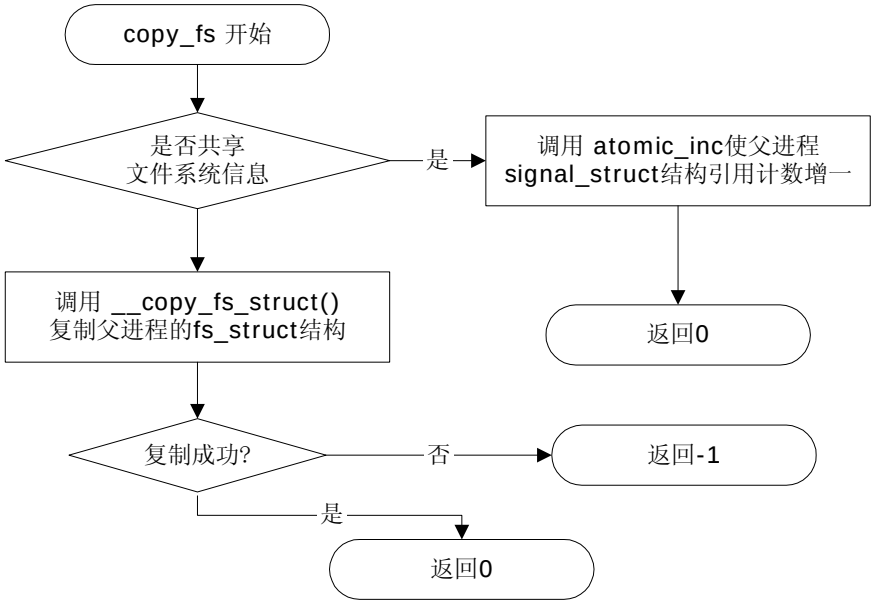
图见后面。

5.4.6 copy_fs( )

位置:kernel/fork.c
形式:static inline int copy_fs(unsigned long clone_flags, struct task_struct * tsk)
参数: clone_flags 标志,用于标明新创建进程与父进程共享资源的方式,
tsk 进程的 task_struct 结构指针.
功能: 根据 clone_flags 复制或共享父进程的文件系统信息
流程:

1. 若父子进程共享文件系统信息，则调用 atomic_inc 使父进程 signal_struct 结构引用计数增一。返回 0，退出。
2. 调用 __copy_fs_struct()复制父进程的 fs_struct 结构。若复制成功，则返回 0，退出；若复制失败，则返回－1，退出。


get_pid 开始
next_safe=PID_MAX(允许的pid最大值)
struct task_struct *p
新进程与祖先共享PID?
是
返回祖先PID
调用spin_lock()给last_pid加锁
++last_pid < 允许的pid最大值
是
last_pid=300
否
++last_pid >= next_safe
是
next_safe=PID_MAX;
调用read_lock()对进程队列加锁
last_pid >= next_safe
否
是
按顺序访问进程队列中的元素
若++last_pid < 允许的pid最大值
last_pid=300
last_pid与已分配的id冲突?
是
next_safe=PID_MAX
否
p->pid>last_pid 且
next_safe >pid
是
next_safe=p->pid
否
p->pgrp>last_pid 且
next_safe >pgrp
是
next_safe=p->pgrp
否
p->session>last_pid 且
next_safe >session
是
next_safe=p->session
否
调用read_unlock()对进程队列解锁
调用spin_unlock()给last_pid解锁
返回 last_pid
get_pid结束



5.4.7 copy_mm( )

位置:kernel/fork.c

形式:static int copy_mm(unsigned long clone_flags, struct task_struct * tsk)

参数: clone_flags 标志,用于标明新创建进程与父进程共享资源的方式,
tsk 进程的 task_struct 结构指针.

功能: 根据 clone_flags 复制或共享父进程的存储管理信息

流程:

1. 初始化子进程 task_struct 结构与内存存储信息相关的数据项。
2. 若父进程的指向 mm_struct 结构的指针为空，则返回 0，退出。
3. 若父子进程共享存储管理信息,则调用 atomic_inc 使父进程 mm_struct 结构引用计数增一 。然后跳转至 good_mm，令 tsk->mm = mm; tsk->active_mm = mm;返回 0，退出。
4. 调用 allocate_mm 申请分配一个 mm_struct 结构，若申请失败，则跳转至 fail_nomem，失败返回。
5. 调用 memcpy 将父进程的 mm_struct 结构复制给子进程。
6. 调用 mm_init 对子进程 mm_struct 结构作初始化处理，若初始化失败，则跳转至 fail_nomem，失败返回。
7. 调用 down_write 对父进程相应页面发写操作睡眠信号。
8. 调用 dup_mmap 复制 vm_area_struct 结构和页面映射表。
9. 调用 up_write 对父进程相应页面发写操作唤醒信号，若复制失败，则跳转至 free_pt。
10. 调用 copy_segments 复制 LDT。
11. 调用 init_new_context 初始化上下文，若初始化失败，则跳转至 free_pt。
12. good_mm 标号处理，正常返回。
13. free_pt 标号处理。
14. fail_nomem:失败返回。

5.4.8 copy_files( )

位置:kernel/fork.c
形式:static int copy_files(unsigned long clone_flags, struct task_struct * tsk)
参数: clone_flags 标志,用于标明新创建进程与父进程共享资源的方式,
tsk 进程的 task_struct 结构指针.
功能: 根据 clone_flags 复制或共享父进程的打开文件表
流程:

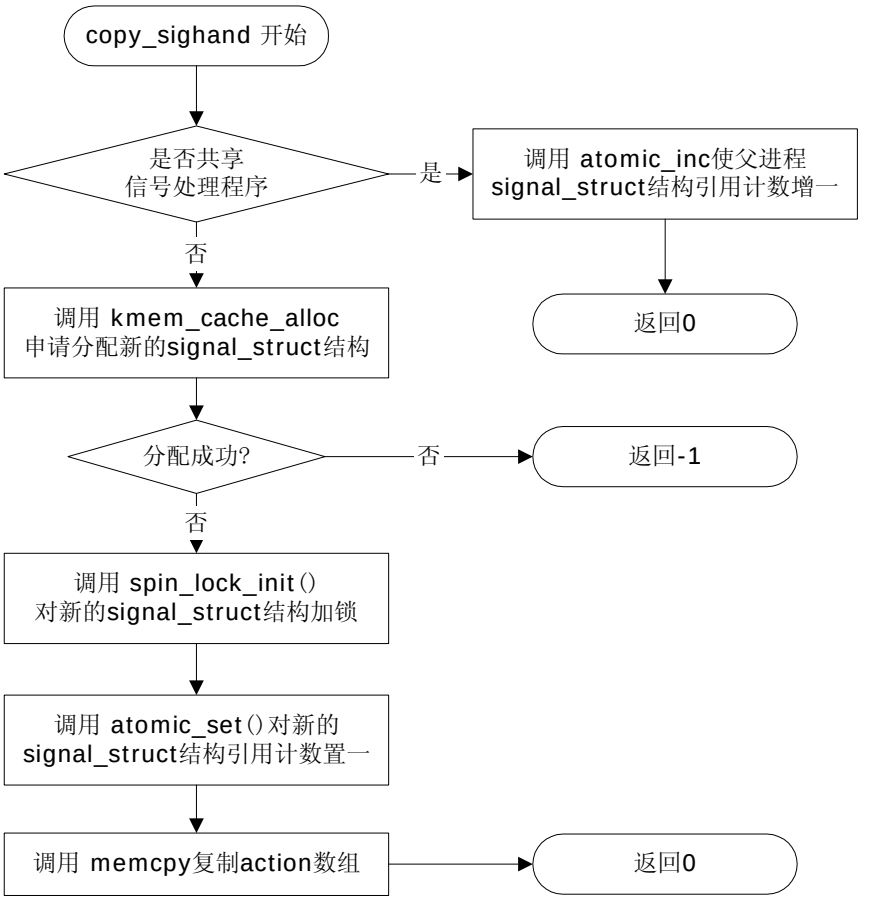
1. 若父进程的 files_struct 结构指针为空，则返回 error。
2. 若 CLONE_FILES 标志置位，则调用 atomic_inc 使父进程 files_struct 结构引用计数增一，返回 error。
3. 为子进程申请新的 files_struct 结构，若申请失败，则返回 error。
4. 调用 atomic_set 将新分配的 files_struct 结构引用计数置 1。
5. 设置为子进程申请新的 files_struct 结构。
6. 若父进程的 max_fdset 大于 __FD_SETSIZE,则调用 expand_fdset 扩展子进程的 fd;若扩展失败,则释放新申请的文件结构的相关域,返回 error。
7. 父进程的文件描述符数组复制给子进程。
8. 新进程的 task_struct 结构的 files 指向新申请的文件结构，令 error = 0。
9. 返回 error，退出。

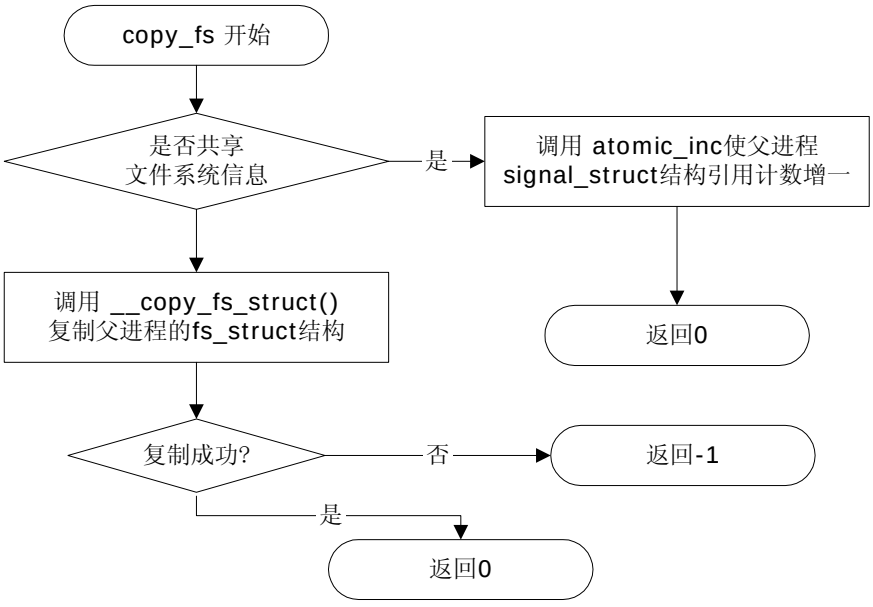
流程图见后面。

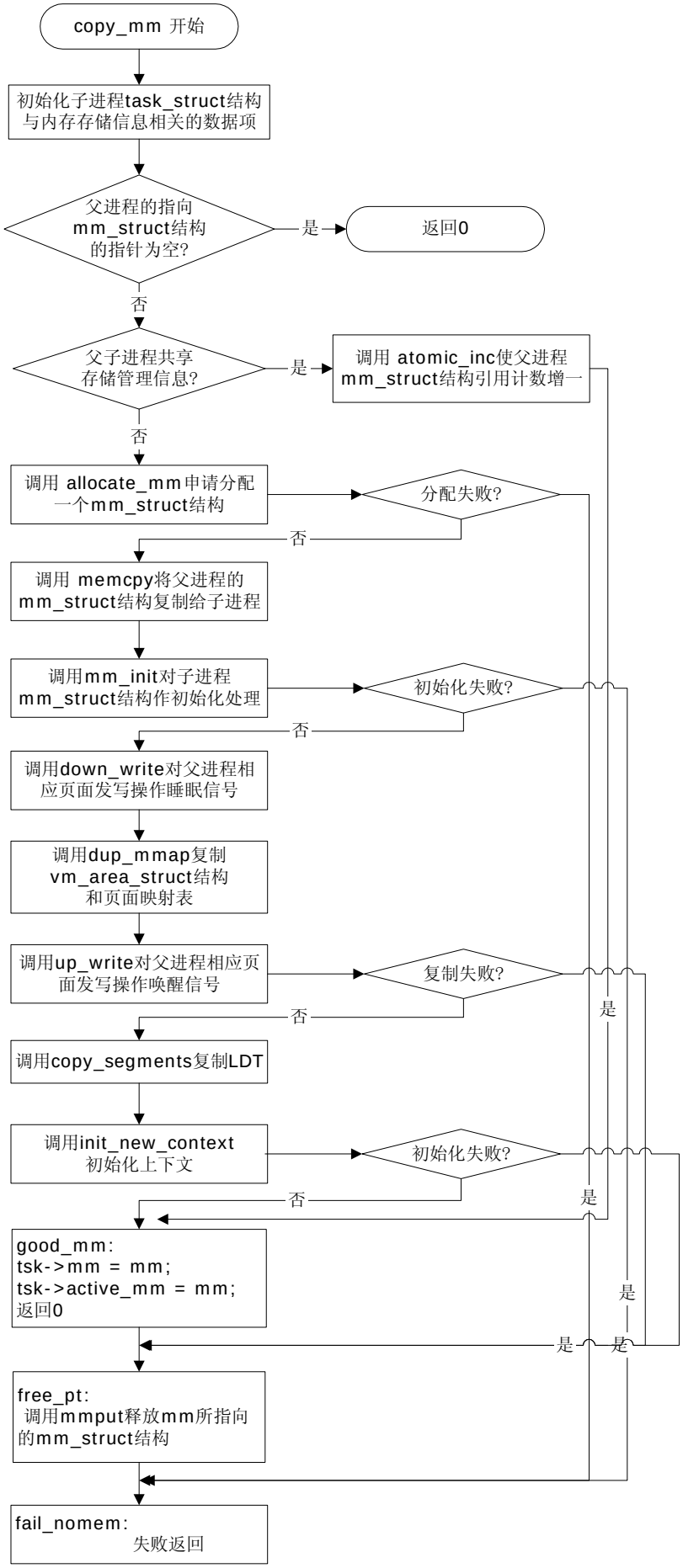
5.4.9 copy_sighand

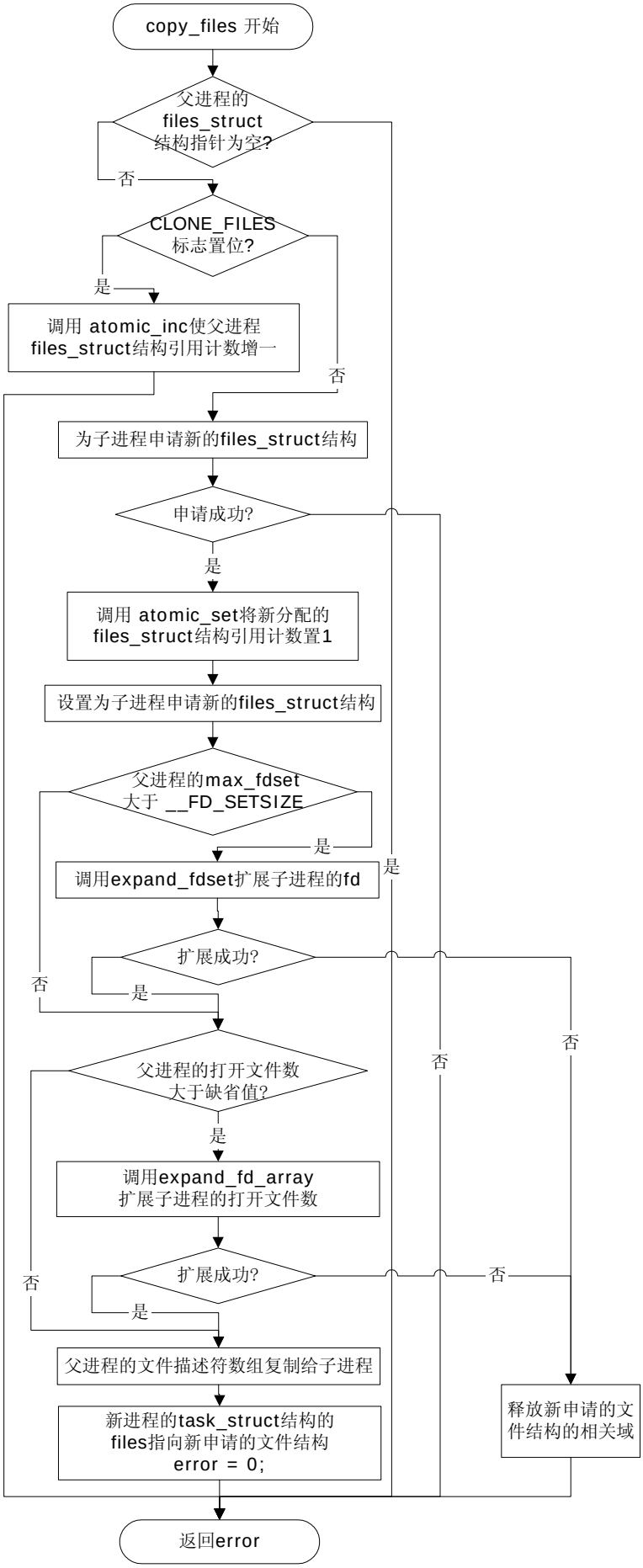
位置:kernel/fork.c
形式:static inline int copy_sighand(unsigned long clone_flags, struct task_struct * tsk)
参数: clone_flags 标志,用于标明新创建进程与父进程共享资源的方式,
tsk 进程的 task_struct 结构指针.
功能: 根据 clone_flags 复制或共享父进程的信号处理句柄
流程:

1. 若父子进程共享信号处理程序，则调用 atomic_inc 使父进程 signal_struct 结构引用计数增一，返回 0，退出。
2. 调用 kmem_cache_alloc 申请分配新的 signal_struct 结构。若分配失败，则返回-1，退出。
3. 调用 spin_lock_init()对新的 signal_struct 结构加锁。
4. 调用 atomic_set()对新的 signal_struct 结构引用计数置一。
5. 调用 memcpy 复制 action 数组。
6. 返回 0，退出。


copy_sighand 开始
是否共享
信号处理程序
是
调用 atomic_inc使父进程
signal_struct结构引用计数增一
返回0
否
调用 kmem_cache_alloc
申请分配新的signal_struct结构
分配成功?
否
返回-1
否
调用 spin_lock_init()
对新的signal_struct结构加锁
调用 atomic_set()对新的
signal_struct结构引用计数置一
调用 memcpy复制action数组
返回0


copy_fs 开始
是否共享
文件系统信息
是
调用 atomic_inc使父进程
signal_struct结构引用计数增一
返回0
调用 __copy_fs_struct()
复制父进程的fs_struct结构
复制成功?
否
返回-1
是
返回0


copy_mm 开始
初始化子进程task_struct结构
与内存存储信息相关的数据项
父进程的指向
mm_struct结构
的指针为空?
是
返回0
否
父子进程共享
存储管理信息?
是
调用 atomic_inc使父进程
mm_struct结构引用计数增一
否
调用 allocate_mm申请分配
一个mm_struct结构
分配失败?
否
调用 memcpy将父进程的
mm_struct结构复制给子进程
调用mm_init对子进程
mm_struct结构作初始化处理
初始化失败?
否
调用down_write对父进程相
应页面发写操作睡眠信号
调用dup_mmap复制
vm_area_struct结构
和页面映射表
调用up_write对父进程相应页
面发写操作唤醒信号
复制失败?
否
调用copy_segments复制LDT
调用init_new_context
初始化上下文
初始化失败?
否
good_mm:
tsk->mm = mm;
tsk->active_mm = mm;
返回0
free_pt:
调用mmput释放mm所指向
的mm_struct结构
fail_nomem:
失败返回
是
是
是
是
是


copy_files 开始
父进程的
files_struct
结构指针为空?
否
CLONE_FILES
标志置位?
是
调用 atomic_inc使父进程
files_struct结构引用计数增一
否
为子进程申请新的files_struct结构
申请成功?
是
调用 atomic_set将新分配的
files_struct结构引用计数置1
设置为子进程申请新的files_struct结构
父进程的max_fdset
大于 __FD_SETSIZE
是
调用expand_fdset扩展子进程的fd
是
扩展成功?
否
是
否
父进程的打开文件数
大于缺省值?
是
否
调用expand_fd_array
扩展子进程的打开文件数
扩展成功?
否
是
否
父进程的文件描述符数组复制给子进程
新进程的task_struct结构的
files指向新申请的文件结构
error = 0;
释放新申请的文
件结构的相关域
返回error

5.4.10 copy_thread( )

位置:arch/i386/kernel/process.c*/

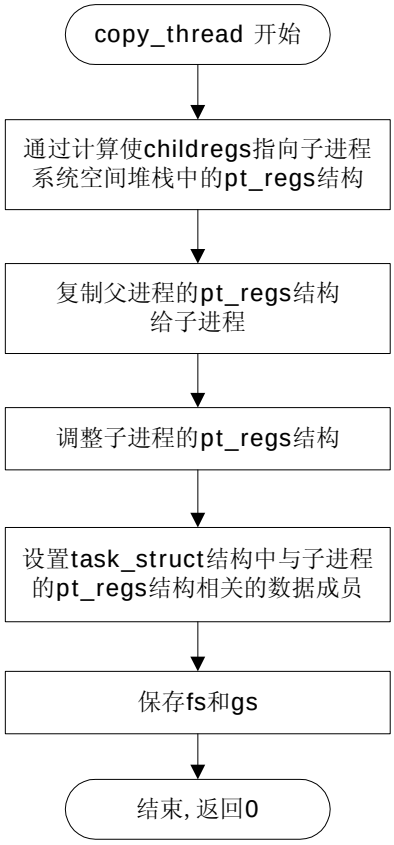
形式:int copy_thread(int nr, unsigned long clone_flags, unsigned long esp,
struct task_struct * p, struct pt_regs * regs)

参数: nr 未使用, clone_flags 标志,用于标明新创建进程与父进程共享资源的方式, esp 堆栈指针,p 进程的 task_struct 结构指针,regs pt_regs 结构.

功能: 复制父进程的系统空间堆栈

流程:

1. 通过计算使 childregs 指向子进程系统空间堆栈中的 pt_regs 结构。
2. 复制父进程的 pt_regs 结构给子进程。
3. 调整子进程的 pt_regs 结构。
4. 设置 task_struct 结构中与子进程的 pt_regs 结构相关的数据成员。
5. 保存 fs 和 gs。
6. 结束,返回 0。

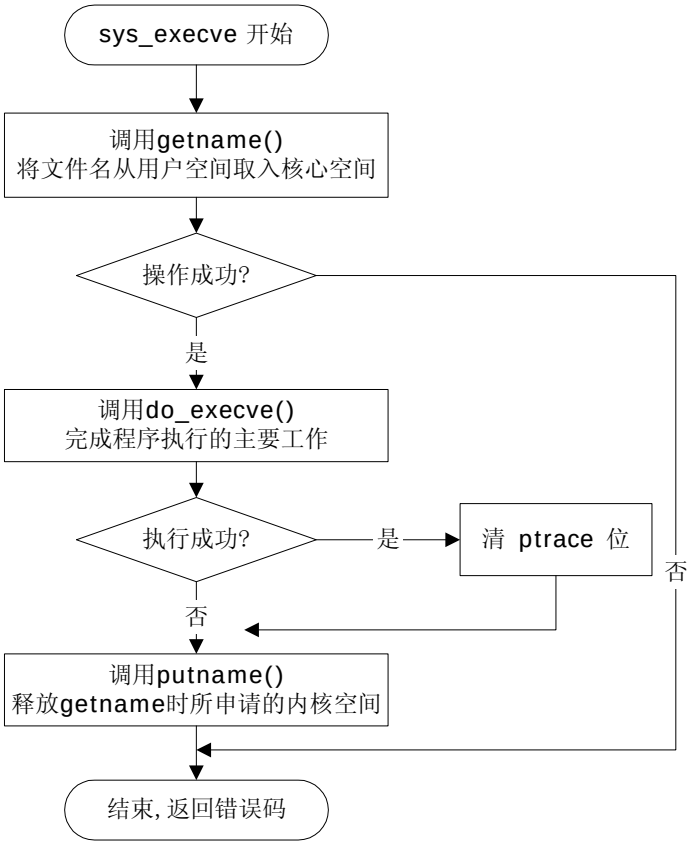


5.5 子进程的装入和执行

5.5.1 sys_execve （ ）

位置:fs/exec.c
形式:asmlinkage int sys_execve(struct pt_regs regs)
参数:regs, pt_regs 结构,可用此结构访问堆栈中的相应内容
功能:调用 do_execve 执行一个可执行二进制文件
流程:

1. 调用 getname()将文件名从用户空间取入核心空间。若操作失败，则转向步骤 4。
2. 调用 do_execve()完成程序执行的主要工作。若执行成功，则清 ptrace 位。
3. 调用 putname()释放 getname 时所申请的内核空间。
4. 结束,返回错误码。



5.5.2 getname ( )

位置:fs/namei.c
形式:char * getname(const char * filename)
参数:filename 目标文件路径名字符串指针
功能: 将文件名从用户空间取入核心空间
流程:

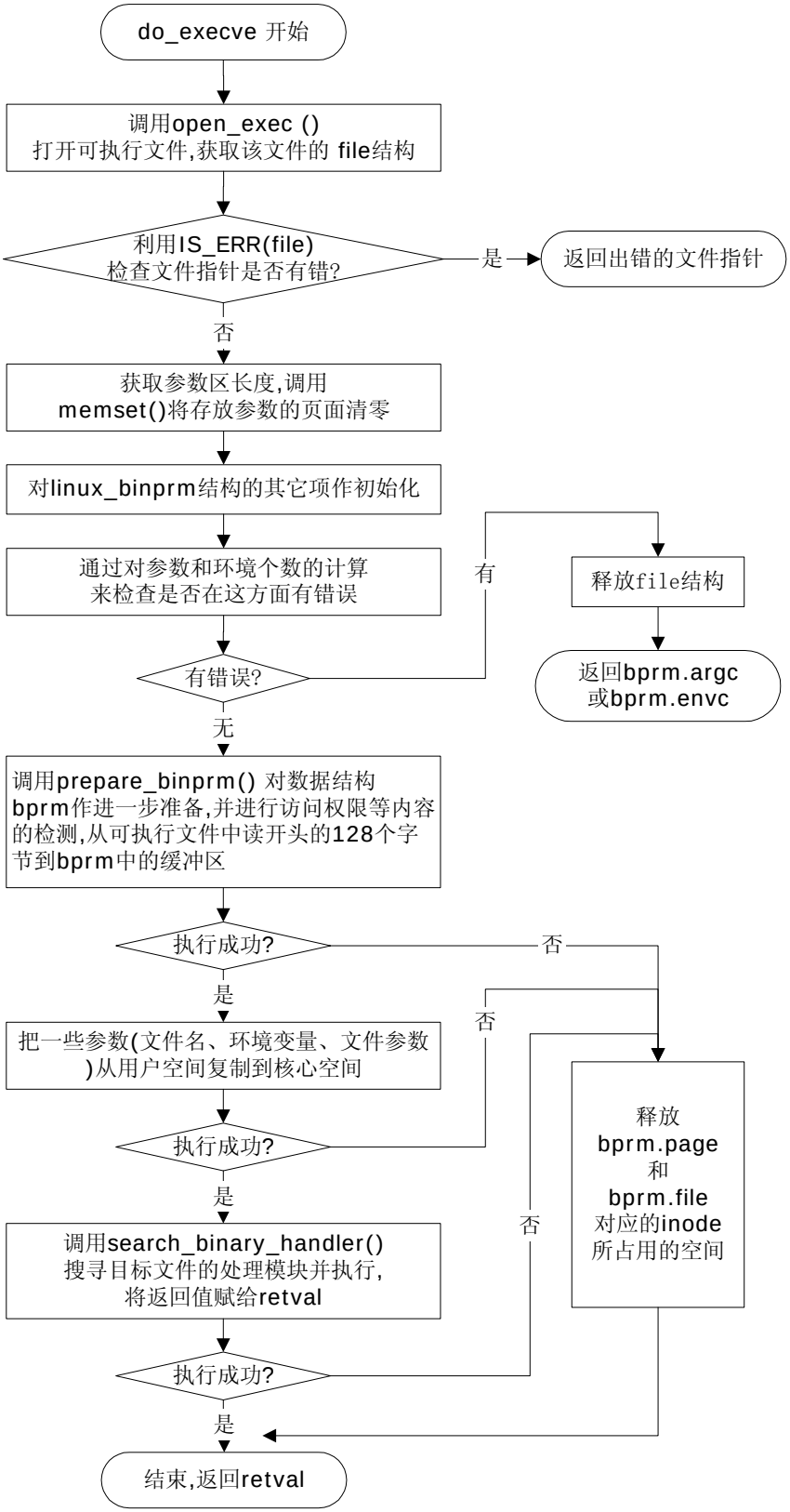
1. 通过 result=ERR_PTR(-ENOMEM);将错误号赋给返回值 result 。
2. 调用__getname()分配一个物理页面作为缓冲区,将返回值赋给 tmp。若页面分配失败，则结束,返回 result。
3. 调用 do_getname()从用户空间拷贝字符串到缓冲区。若拷贝失败，则结束,返回 result。
4. 调用 putname()释放为文件名分配的缓冲区并将错误码赋给 result。
5. 结束,返回 result。

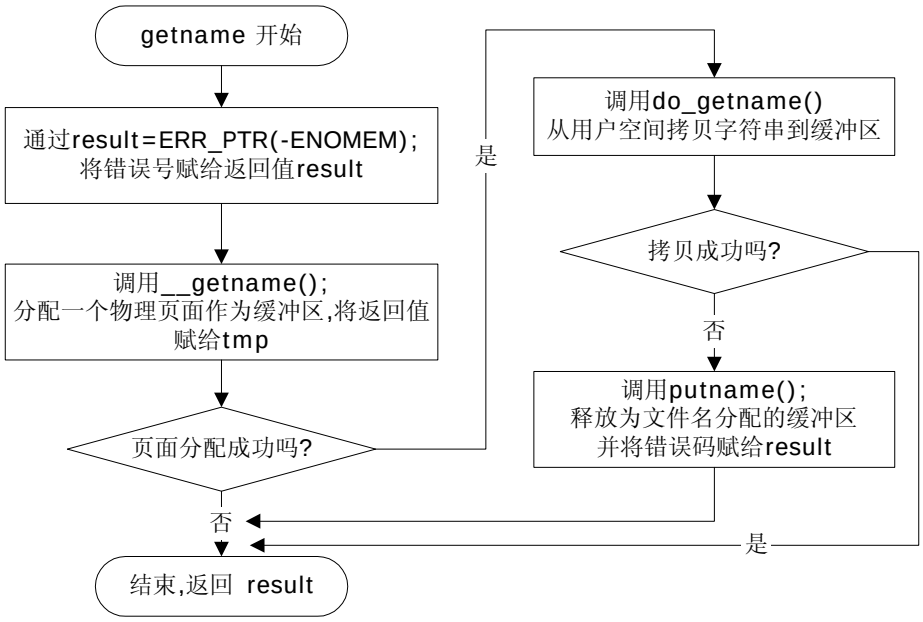
流程图见后面。

5.5.3 do_execve ( )

位置:fs/exec.c
形式: int do_execve(char * filename, char ** argv, char ** envp, struct pt_regs * regs)
参数: filename 目标文件路径名字符串指针,argv 参数指针,envp 环境指针,
regs, pt_regs 结构,可用此结构访问堆栈中的相应内容
功能:执行一个可执行二进制文件
流程:

1. 调用 open_exec ()打开可执行文件,获取该文件的 file 结构。
2. 利用 IS_ERR(file)检查文件指针是否有错，若有错则返回出错的文件指针，退出。
3. 获取参数区长度,调用 memset()将存放参数的页面清零。
4. 对 linux_binprm 结构的其它项作初始化。
5. 通过对参数和环境个数的计算来检查是否在这方面有错误，若有错误，则释放 file 结构，返回 bprm.argc 或 bprm.envc，退出。
6. 调用 prepare_binprm() 对数据结构 bprm 作进一步准备,并进行访问权限等内容的检测,从可执行文件中读开头的 128 个字节到 bprm 中的缓冲区。若执行失败,则释放 bprm.page 和 bprm.file 对应的 inode 所占用的空间,结束,返回 retval。
7. 调用 search_binary_handler()搜寻目标文件的处理模块并执行,将返回值赋给 retval。若失败，则则释放 bprm.page 和 bprm.file 对应的 inode 所占用的空间。
8. 结束,返回 retval。


do_execve 开始
调用open_exec ()
打开可执行文件,获取该文件的 file结构
利用IS_ERR(file)
检查文件指针是否有错?
是
返回出错的文件指针
否
获取参数区长度,调用
memset()将存放参数的页面清零
对linux_binprm结构的其它项作初始化
通过对参数和环境个数的计算
来检查是否在这方面有错误
有
释放file结构
返回bprm.argc
或bprm.envc
有错误?
无
调用prepare_binprm() 对数据结构
bprm作进一步准备,并进行访问权限等内容
的检测,从可执行文件中读开头的128个字
节到bprm中的缓冲区
执行成功?
否
是
否
把一些参数(文件名、环境变量、文件参数
)从用户空间复制到核心空间
执行成功?
否
释放
bprm.page
和
bprm.file
对应的inode
所占用的空间
是
调用search_binary_handler()
搜寻目标文件的处理模块并执行,
将返回值赋给retval
执行成功?
否
是
结束,返回retval



5.5.4 search_binary_handler （ ）

位置:fs/exec.c

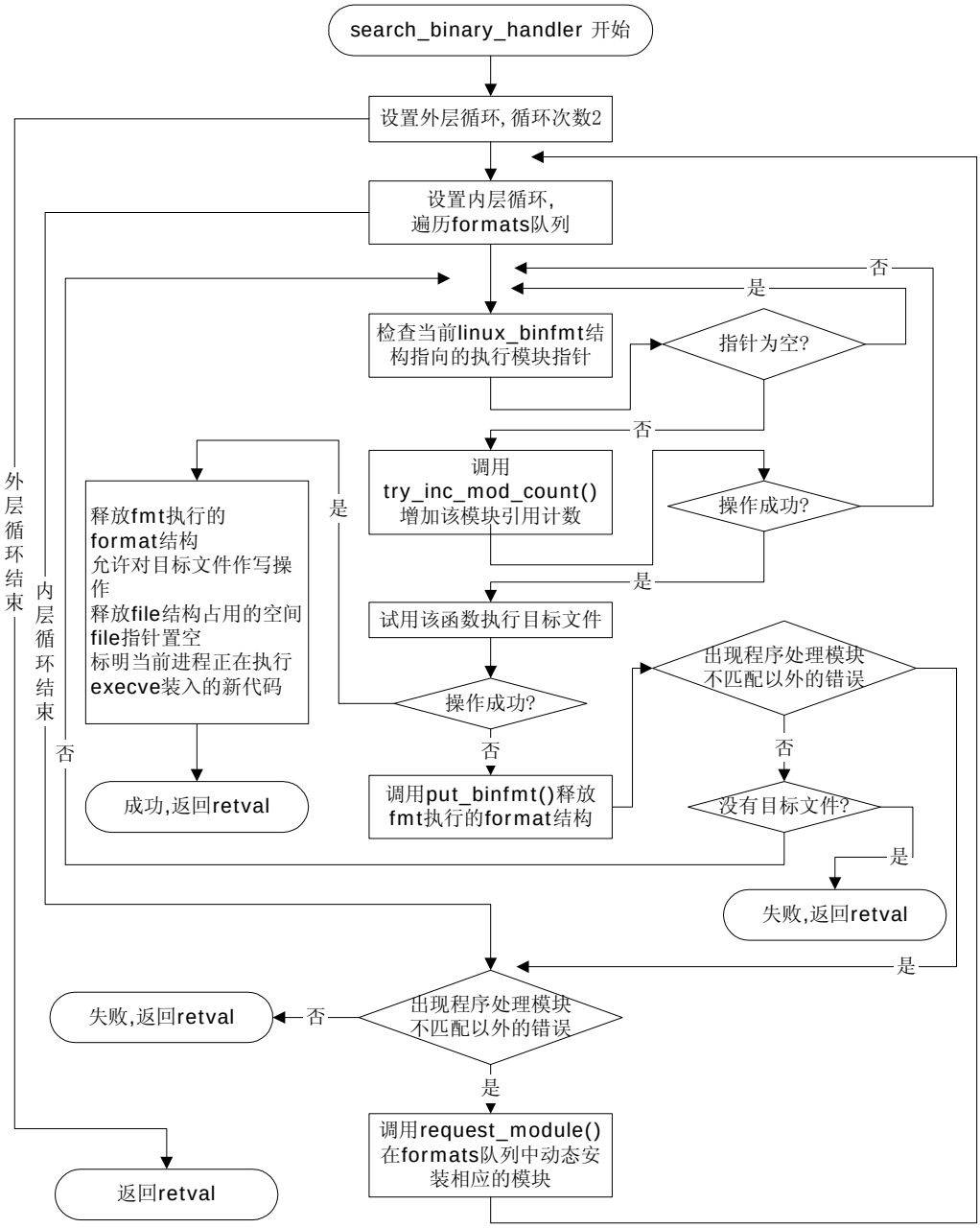
形式:int search_binary_handler(struct linux_binprm *bprm,struct pt_regs *regs)

参数:bprm linux_binprm 结构,该结构拥有执行可执行文件的服务程序指针,
regs, pt_regs 结构,可用此结构访问堆栈中的相应内容

功能:搜寻目标文件的处理模块并执行

流程:

1. 设置外层循环,循环次数 2。
2. 设置内层循环,遍历 formats 队列。
3. 检查当前 linux_binfmt 结构指向的执行模块指针，若指针为空，则重复步骤 3 的操作。
4. 调用 try_inc_mod_count()增加该模块引用计数。若操作失败则转回步骤 3。
5. 试用该函数执行目标文件。若操作成功，则释放 fmt 执行的 format 结构允许对目标文件作写操作释放 file 结构占用的空间 file 指针置空标明当前进程正在执行 execve 装入的新代码，返回 retval，退出。
6. 调用 put_binfmt()释放 fmt 执行的 format 结构。若出现程序处理模块不匹配以外的错误，则执行步骤 7；否则进一步判断没有目标文件,若有则返回 retval，退出，若没有，则转向步骤 3。
7. 内层循环结束。若出现程序处理模块不匹配以外的错误，则调用 request_module()在 formats 队列中动态安装相应的模块，转向步骤 2；否则返回 retval，退出。
8. 外层循环结束。返回 retval，退出。



5.6 父进程的等待

5.6.1 sys_wait4 ( )

位置:kernel/exit.c

形式:asmlinkage long sys_wait4(pid_t pid,unsigned int * stat_addr, int options,struct rusage * ru)

参数: pid 子进程进程号, stat_addr 指向堆栈始址, options 选项, tsk 进程的 task_struct 结构指针

功能: 最终消灭处于僵尸态的进程的剩余资源

流程:

1. 在当前进程的系统堆栈上分配一个 wait_queue_t 结构。

2. 若 options 有错，则返回错误码-EINVAL。
3. 调用 add_wait_queue()把分配的 wait_queue_t 结构链入当前进程的 wait_chldexit 队列中。
4. 令 flag = 0; tsk = current;current->state =TASK_INTERRUPTIBLE。
5. 设置循环遍历当前进程的子进程。
6. 若当前子进程(p)pid 和 pgrp 与搜寻的 pid 不匹配，则执行下一个循环过程。
7. 若 p 进程不是"克隆"子进程，。
8. flag=0;执行开关语句 switch (p->state)。进程若是停止态，从步骤 9 开始执行；若是僵尸态，从步骤 16 开始执行；若是其它状态，则执行下一个循环过程。
9. 若 p->exit_code=0，则执行下一个循环过程。
10. 若!(options & WUNTRACED) &&!(p->ptrace & PT_PTRACED)为真，则执行下一个循环过程。
11. 若 ru 有效且 stat_addr 非空，则将进程状态复制到 stat_addr。若复制失败则转向步骤 22。
12. 若__WNOTHREAD 未置位，则令 tsk 指向线程组下一个线程，设置循环 while (tsk != current)，转回步骤 5。
13. 若 flag=1，则令 retval=0；否则令 retval=-ECHILD，跳转至步骤 22。
14. 若 WNOHANG 未置位，则令 retval=-ERESTARTSYS；否则跳转至步骤 22。
15. 若当前进程未挂起，则调用 schedule()，转入调度，调度后若能获得 CPU，则跳转至步骤 4；若当前进程挂起，跳转至步骤 22。
16. 将 p 进程在用户态和核心态运行的时间加在当前进程上。
17. 若 ru 有效且 stat_addr 非空，则将进程状态复制到 stat_addr。若复制失败则转向步骤 22。
18. 令 retval = p->pid。
19. 若 p 进程的祖先与父进程不同，则执行步骤 20；否则执行步骤 21。
20. 将 p 进程从进程队列中摘除令 p 的父进程指向祖先进程在祖先进程的控制下重新链入队列，并通知祖先进程 p 进程已死亡，转向步骤 22。
21. 调用 release_task()释放死亡进程的剩余资源。转向步骤 22。
22. 设当前进程为就绪态将当前进程链入运行队列。
23. 结束,返回 retval。

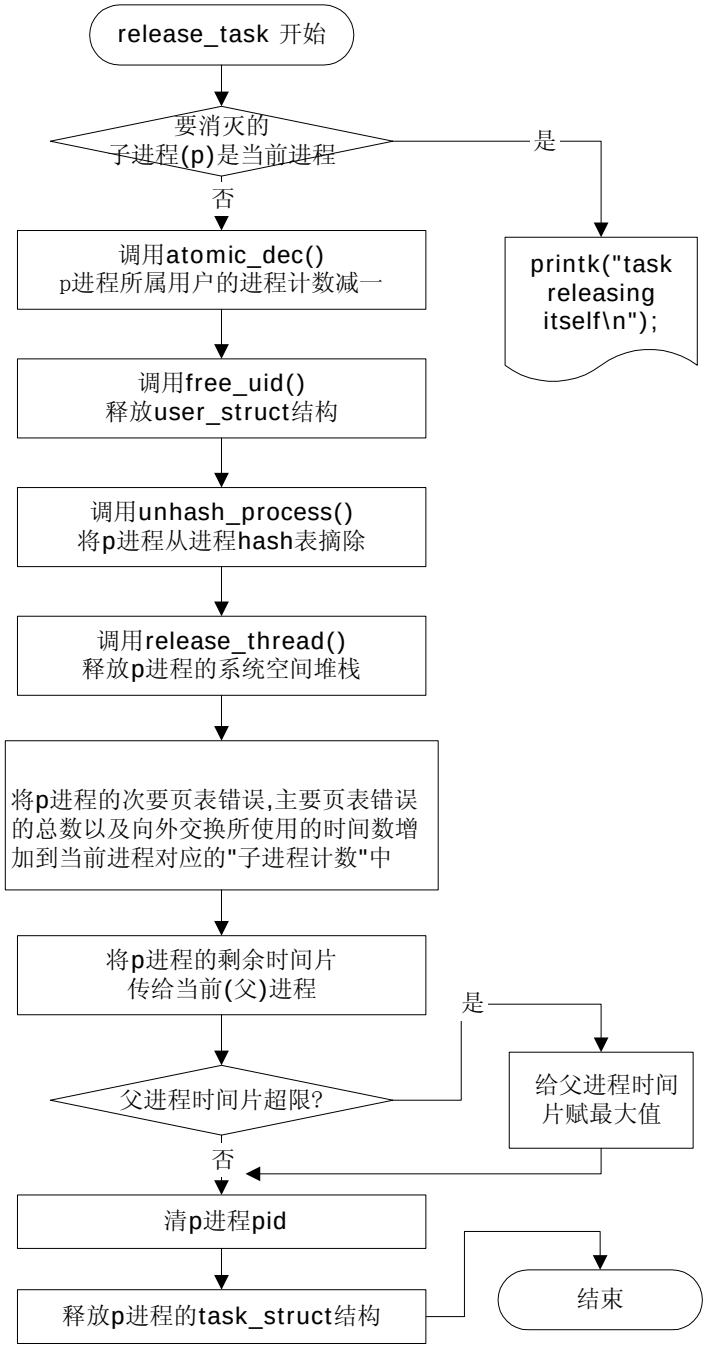
流程图见后面。

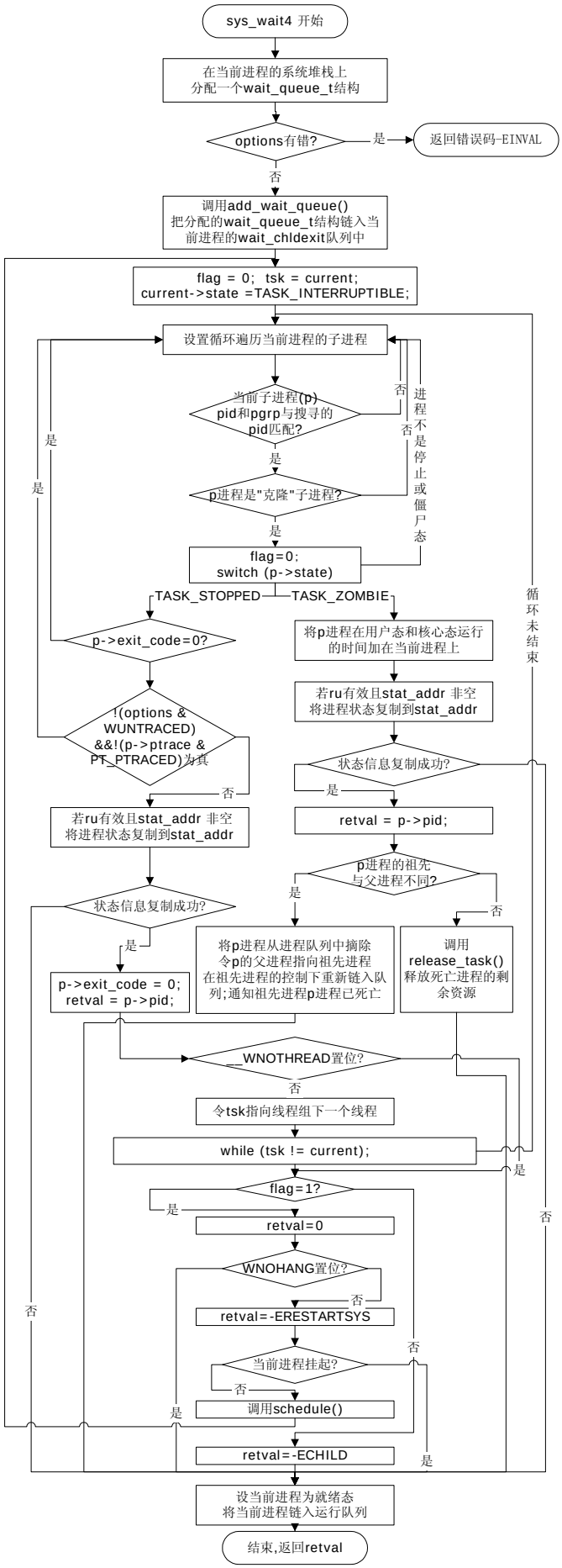
5.6.2 release_task ( )

位置:kernel/exit.c
形式:static void release_task(struct task_struct * p)
参数: tsk 进程的 task_struct 结构指针
功能: 释放进程控制块
流程:

1. 要消灭的子进程(p)是当前进程，则打印提示，退出。
2. 调用 atomic_dec()p 进程所属用户的进程计数减一。
3. 调用 free_uid()释放 user_struct 结构。

4．调用 unhash_process()将 p 进程从进程 hash 表摘除。
5．调用 release_thread()释放 p 进程的系统空间堆栈。
6．将 p 进程的次要页表错误,主要页表错误的总数以及向外交换所使用的时间数增加到当前进程对应的"子进程计数"中见后面。
7．将 p 进程的剩余时间片传给当前(父)进程。
8．若父进程时间片超限，则给父进程时间片赋最大值。
9．清 p 进程 pid。
10．释放 p 进程的 task_struct 结构。
11．结束，退出。



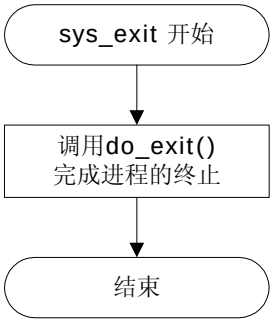

sys_wait4 开始
在当前进程的系统堆栈上
分配一个wait_queue_t结构
options有错?
是
返回错误码-EINVAL
否
调用add_wait_queue()
把分配的wait_queue_t结构链入当
前进程的wait_chldexit队列中
flag = 0; tsk = current;
current->state =TASK_INTERRUPTIBLE;
设置循环遍历当前进程的子进程
当前子进程(p)
pid和pgrp与搜寻的
pid匹配?
否
是
进程不是停止或僵尸态
p进程是"克隆"子进程?
是
否
是
flag=0;
switch (p->state)
TASK_STOPPED
TASK_ZOMBIE
p->exit_code=0?
!(options &
WUNTRACED)
&&!(p->ptrace &
PT_PTRACED)为真
否
若ru有效且stat_addr 非空
将进程状态复制到stat_addr
状态信息复制成功?
是
p->exit_code = 0;
retval = p->pid;
将p进程在用户态和核心态运行
的时间加在当前进程上
若ru有效且stat_addr 非空
将进程状态复制到stat_addr
状态信息复制成功?
是
retval = p->pid;
p进程的祖先
与父进程不同?
是
否
将p进程从进程队列中摘除
令p的父进程指向祖先进程
在祖先进程的控制下重新链入队
列;通知祖先进程p进程已死亡
调用
release_task()
释放死亡进程的剩
余资源
__WNOTHREAD置位?
否
令tsk指向线程组下一个线程
while (tsk != current);
是
循环未结束
否
flag=1?
是
retval=0
WNOHANG置位?
否
retval=-ERESTARTSYS
当前进程挂起?
否
调用schedule()
是
否
否
retval=-ECHILD
是
设当前进程为就绪态
将当前进程链入运行队列
结束,返回retval

5.7 子进程的消亡

5.7.1 sys_exit( )

位置:kernel/exit.c
形式:asmlinkage long sys_exit(int error_code)
参数:error_code 错误码
功能:调用 do_exit( )消灭进程
流程:

1. 调用 do_exit()完成进程的消亡。
2. 结束，退出。



5.7.2 do_exit( )

位置:kernel/exit.c
形式: NORET_TYPE void do_exit(long code)
参数: code 错误码
功能:消灭进程
流程:

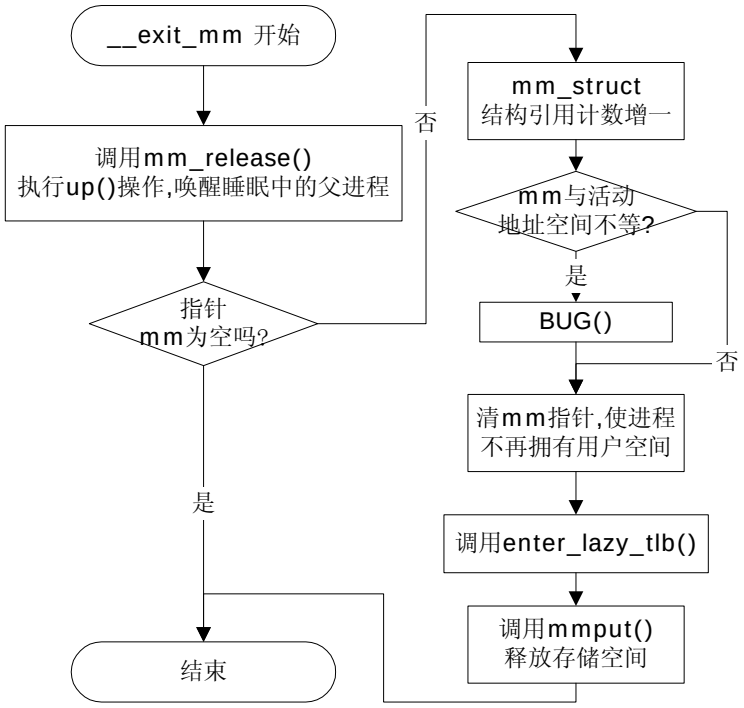
1. 若是中断控制程序或 0 号进程(idle)以及 init 进程，则调用 panic()挂起系统。
2. 调用 __exit_mm()释放进程的存储管理信息。
3. 调用 sem_exit()删除信号队列,释放信号结构。
4. 调用 __exit_files()释放进程的已打开文件的信息。
5. 调用 __exit_fs()释放进程的文件系统。
6. 调用 exit_sighand()释放进程的 signal 管理信息。
7. 调用 exit_thread()释放进程的 LDT。
8. 若是 session 的主进程，则调用 disassociate_ctty()释放 tty。
9. exec_domain 和 binfmt 结构共享计数减一， tsk->exit_code = code。
10. 调用 exit_notify()通知当前进程的父进程和子进程,它要消亡了。
11. 调用 schedule() 转入调度。
12. 若经过调度后重新获得 CPU，转向步骤 2。

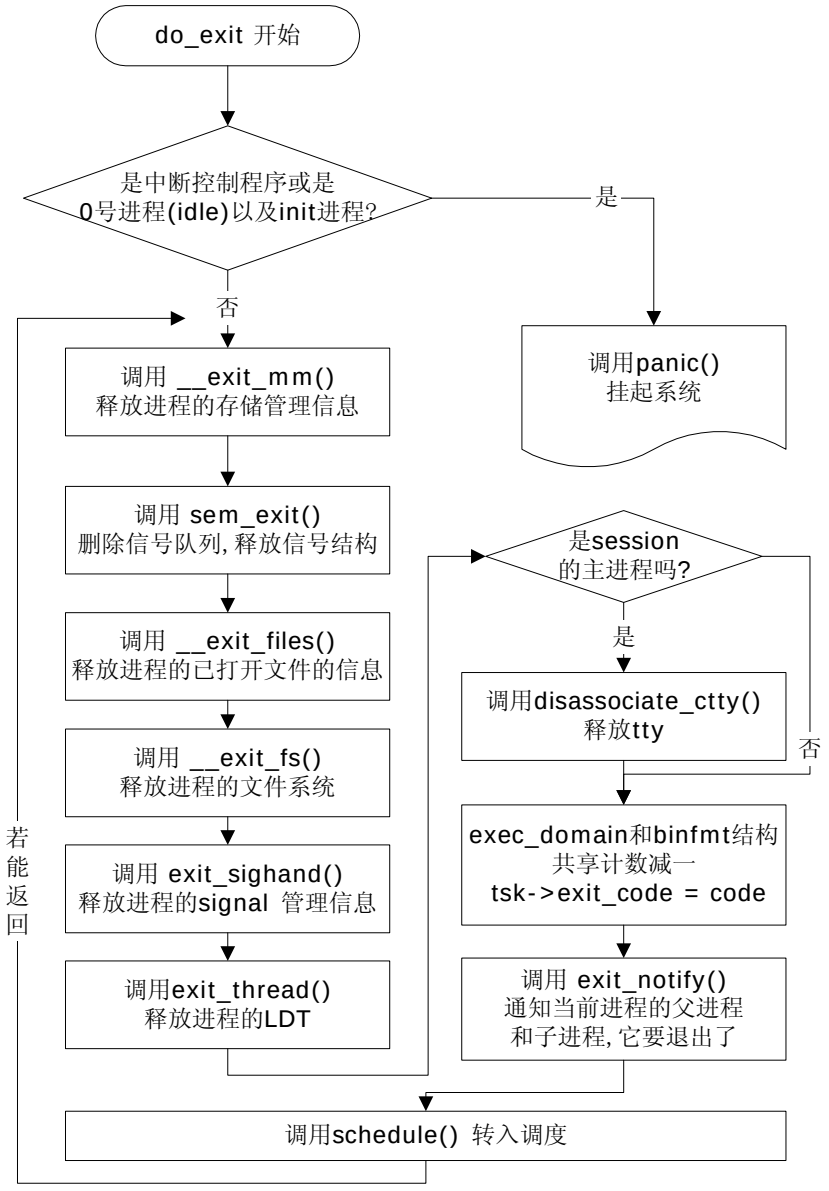
流程图见后面。

5.7.3 __exit_mm( )

位置:kernel/exit.c
形式: static inline void __exit_mm(struct task_struct * tsk)
参数: tsk 进程的 task_struct 结构指针
功能: 释放进程的存储管理信息
流程:

1. 调用 mm_release()执行 up()操作,唤醒睡眠中的父进程。
2. 指针 mm 为空，则结束，退出。若不为空，则继续执行第 3 步。
3. mm_struct 结构引用计数增一。
4. 若 mm 与活动地址空间不等，则调用 BUG()。
5. 清 mm 指针,使进程不再拥有用户空间。
6. 调用 enter_lazy_tlb()。
7. 调用 mmput()释放存储空间。
8. 结束，退出。





5.7.4 __exit_files( )

位置:kernel/exit.c
形式: static inline void __exit_files(struct task_struct * tsk)
参数: tsk 进程的 task_struct 结构指针
功能: 释放进程的已打开文件的信息
流程:

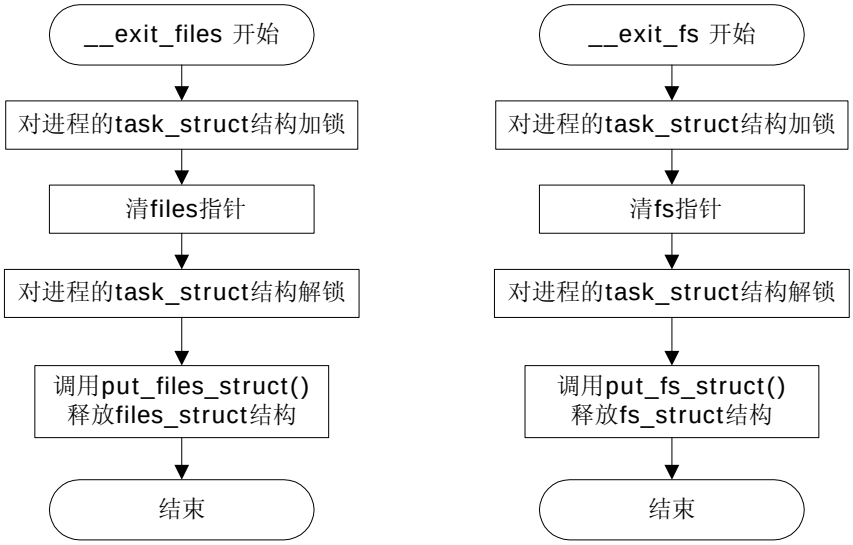
1. 对进程的 task_struct 结构加锁。
2. 清 files 指针。
3. 对进程的 task_struct 结构解锁。
4. 调用 put_files_struct()释放 files_struct 结构。
5. 结束，退出。

流程图见后面。

5.7.5 __exit_fs( )

位置:kernel/exit.c
形式: static inline void __exit_fs(struct task_struct * tsk)
参数: tsk 进程的 task_struct 结构指针
功能: 释放进程的文件系统
流程:

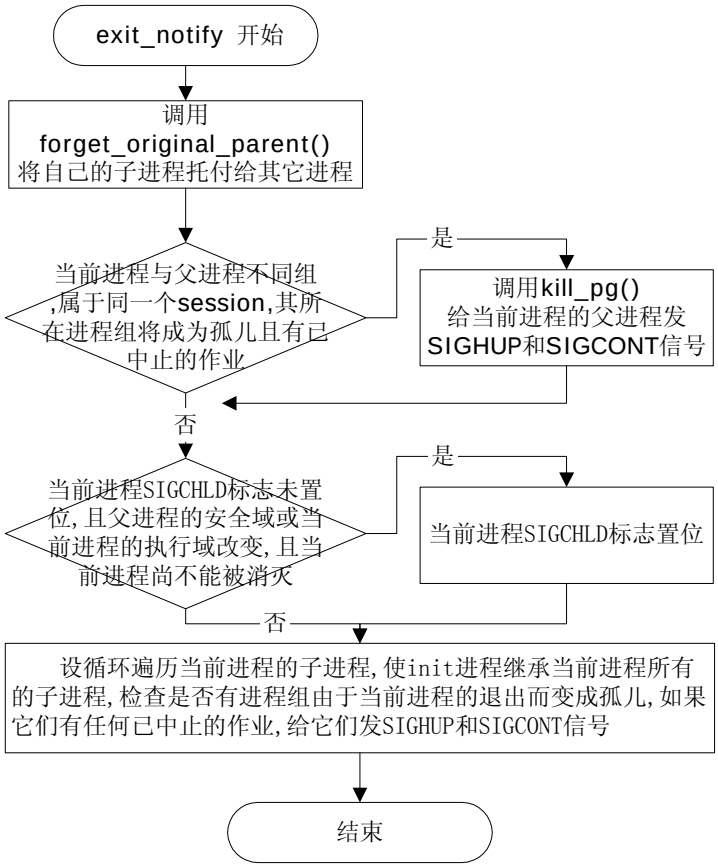
1. 对进程的 task_struct 结构加锁。
2. 清 fs 指针。
3. 对进程的 task_struct 结构解锁。
4. 调用 put_fs_struct()释放 fs_struct 结构。
5. 结束，退出。



5.7.6 exit_notify ( )

位置:kernel/exit.c
形式: static void exit_notify(void)
参数: 无
功能: 告知父子进程自己要消亡了
流程:

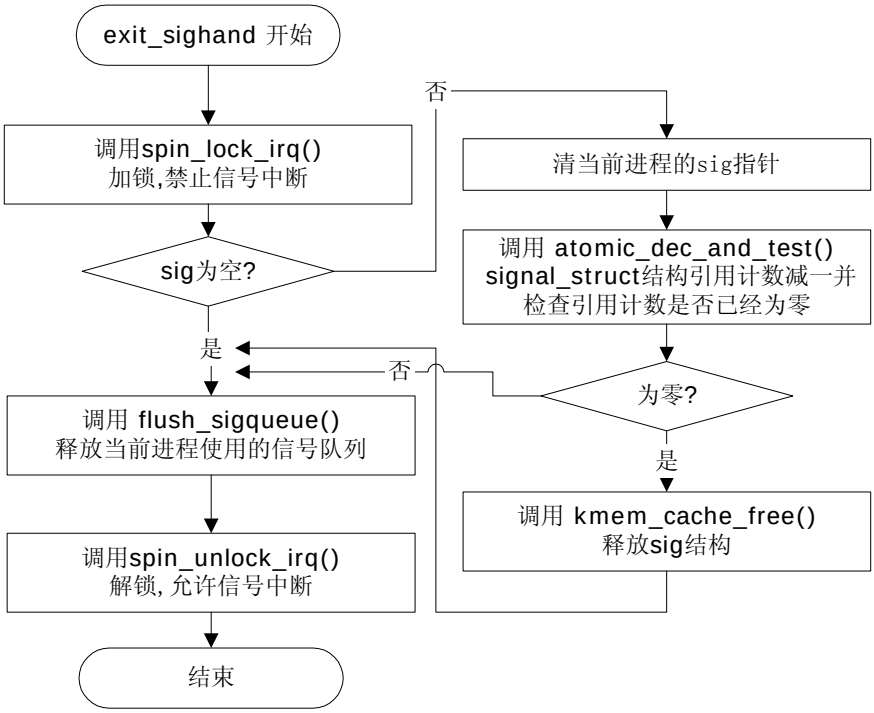
1. 调用 forget_original_parent()将自己的子进程托付给其它进程。
2. 若当前进程与父进程不同组,属于同一个 session,其所在进程组将成为孤儿且有已中止的作业，则调用 kill_pg()给当前进程的父进程发 SIGHUP 和 SIGCONT 信号。
3. 若当前进程 SIGCHLD 标志未置位,且父进程的安全域或当前进程的执行域改变,且当前进程尚不能被消灭，则当前进程 SIGCHLD 标志置位。
4. 设循环遍历当前进程的子进程,使 init 进程继承当前进程所有的子进程,检查是否有进程组由于当前进程的退出而变成孤儿,如果它们有任何已中止的作业,给它们发 SIGHUP 和 SIGCONT 信号。
5. 结束，退出。



5.7.7 exit_sighand ( )

位置:kernel/signal.c
形式:void exit_sighand(struct task_struct *tsk)
参数: tsk 进程的 task_struct 结构指针
功能: 释放进程的 signal 管理信息
流程:

1. 调用 spin_lock_irq()加锁,禁止信号中断。
2. 若 sig 为空，则清当前进程的 sig 指针；若 sig 不空，则跳转至步骤 4 继续执行。
3. 调用 atomic_dec_and_test()，使 signal_struct 结构引用计数减一并检查引用计数是否已经为零。若引用计数为零，则调用 kmem_cache_free()释放 sig 结构，执行步骤 4；若不为零，则直接转向步骤 4。
4. 调用 flush_sigqueue()释放当前进程使用的信号队列。
5. 调用 spin_unlock_irq()解锁,允许信号中断。
6. 结束，退出。


exit_sighand 开始
调用spin_lock_irq()
加锁,禁止信号中断
sig为空?
否
清当前进程的sig指针
调用 atomic_dec_and_test()
signal_struct结构引用计数减一并
检查引用计数是否已经为零
为零?
否
是
是
调用 flush_sigqueue()
释放当前进程使用的信号队列
调用 kmem_cache_free()
释放sig结构
调用spin_unlock_irq()
解锁, 允许信号中断
结束

第 6 章 Linux 内核源代码分析

6.1 进程调度

```
/*****************************************************************************/
/* 位置 kernel/sched.c*/
asmlinkage void schedule(void)
{
    struct schedule_data * sched_data;
    struct task_struct *prev, *next, *p;
    struct list_head *tmp;
    int this_cpu, c;

    if (!current->active_mm) BUG();                    /*若当前进程所指内存为空,则出错返回*/
need_resched_back:
    prev = current;                                    /*将局部变量 prev 初始化为当前进程*/
    this_cpu = prev->processor;                        /*将局部变量 this_cpu 初始化为当前 cpu*/

    if (in_interrupt())
        /*
         * 检查 schedule()是否正被中断服务程序(中断服务程序不允许调用 schedule()),
         * 若是则转向 scheduling_in_interrupt,打印提示信息后,直接返回.
         */
        goto scheduling_in_interrupt;

    release_kernel_lock(prev, this_cpu); /*释放全局内核锁,对于 i386 单处理器系统为空语句*/

    /* Do "administrative" work here while we don't hold any locks */
    /* 趁我们不加任何锁的时候,在这作一些"管理"操作"*/
    if (bh_mask & bh_active)                           /* 如果有 bottom half 服务请求则执行
                                                        * do_bottom_half()*/
        goto handle_bh;
handle_bh_back:

    /*
     * 由于每个 CPU 在每一时刻只能运行一个进程,
     * 所以要保护'sched_data'
     * 'sched_data' is protected by the fact that we can run
     * only one process per CPU.
     */
    /*sched_data 指向一个 sched_data 数据结构,用来保存下一次调度时使用的信息*/
    sched_data = & aligned_data[this_cpu].schedule_data;
```

```
    spin_lock_irq(&runqueue_lock);                          /*对运行队列加锁,禁止中断*/

    /* move an exhausted RR process to be last.. */
    if (prev->policy == SCHED_RR)     /*如果采用轮转法进行调度,跳转至 move_rr_last*/
        goto move_rr_last;
move_rr_back:

    switch (prev->state) {                          /* 检测当前进程状态*/
        case TASK_INTERRUPTIBLE:                    /* 当前进程为 TASK_INTERRUPTIBLE 态*/
            if (signal_pending(prev)) {             /* 如果能够唤醒他的信号已经迫近*/
                prev->state = TASK_RUNNING; /* 置状态为 TASK_RUNNING*/
                break;                              /* 跳出开关语句*/
            }
        default:                                    /* 若当前进程处于其它状态*/
            del_from_runqueue(prev);                /* 则将当前进程从运行队列中删除*/
        case TASK_RUNNING:                  /*当前进程为 TASK_RUNNING 态,则什么也不做*/
    }
    prev->need_resched = 0;                         /* 当前进程重调度标志置 0*/

    /*
     * this is the scheduler proper: 以下是调度正文,选择下一个可运行进程
     */

repeat_schedule:
    /*
     * Default process to select..
     */
    /*
     * 令 next 指向 idle 进程的 task_struct 结构,表示缺省选中的进程是 idle.
     * 由于已经设定 c=-1000,所以仅在没有其它进程可以运行时
     * 才会让 idle 运行.
     */

    next = idle_task(this_cpu);
    c = -1000;                                      /* 设权值 c=-1000,这是可能的最低值*/
    if (prev->state == TASK_RUNNING)            /* 若当前进程是 TASK_RUNNING 态,
        goto still_running;                          * 跳转至 still_running */

still_running_back:
    list_for_each (tmp , &runqueue_head) {   /* 令 tmp 指向运行队列中的第一个进程*/

        p = list_entry(tmp, struct task_struct, run_list);
        if (can_schedule(p)) {    /* 在单 CPU 系统中,can_schedule()* 的返回值永远为 1*/
```

```
                int weight = goodness(p, prev, this_cpu);          /* 计算进程 p 的权值*/
            if (weight > c)                                        /* 若进程 p 的权值大于 c*/
                c = weight, next = p;                  /*令 c=进程 p 的权值,将 p 赋给 next*/
        }
        tmp = tmp->next;            /* 令 tmp 指向运行队列中下一个进程的 task_struct 结构*/

    }

    /* Do we need to re-calculate counters? */
    /* 我们需要重新计算时间片吗?*/
    if (!c)
            /*若 c=0,则说明运行队列中的所有进程的时间片都已用完,需要重新分配
             *同时,Linux 将非实时进程拥有的时间片数量作为其优先级
             */

        goto recalculate;                                          /*跳转至 recalculate*/
    /*
     * from this point on nothing can prevent us from
     * switching to the next task, save this fact in
     * sched_data.
     * 从这里开始,任何事都无法阻止我们切换到下一个进程了
     */
    sched_data->curr = next;
#ifdef __SMP__
    next->has_cpu = 1;
    next->processor = this_cpu;
#endif
    spin_unlock_irq(&runqueue_lock);                       /*对运行队列解锁,允许中断*/

    if (prev == next)                                  /*若当前进程与选中的进程是同一进程*/
        goto same_process;                                     /*跳转至 same_process*/

#ifdef __SMP__
……
#endif /* __SMP__ */

    kstat.context_swtch++;
    /*
     * This just switches the register state and the
     * stack.切换寄存器的值和堆栈
     */
    switch_to(prev, next, prev);
    __schedule_tail(prev);
```

```
same_process:
    reacquire_kernel_lock(current);                          /*加内核锁*/
    return;                                                  /*返回*/

recalculate:
    {
        struct task_struct *p;
        spin_unlock_irq(&runqueue_lock);                     /* 对运行队列解锁,允许中断*/
        read_lock(&tasklist_lock);                           /* 对进程队列加锁*/
        /*
         * for_each_task()是一个宏,其定义(见 sched.h)为:
         *              #define for_each_task(p) \
         *                      for(p=&init_task;(p=p->next_task)!=&init_task;)
        */
        for_each_task(p)                                     /*为系统中的每个进程分配时间片 */
            /*
             * 对于非实时进程:
             * 运行队列中的进程由于时间片已耗尽,重新赋值后 p->counter=p->pripority=20;
             * 不在运行队列的进程通过 p->counter>>1 将原有时间片减半再加 p->pripority,
             * 以此来提高其优先级,使之能够在转入就绪态时有较高的竞争力.但这种竞争
             * 力的提高不是无限制的,通过 p->counter>>1 使 p->counter 永远不会超过两倍
             *  p->pripority 的值,这样就避免了由于 p->counter 无限增长导致其优先级高于
             * 实时进程的情况.
             * 对于实时进程:
             * 由于实时进程不采用 counter 值作为优先级,所有对实时进程的 counter 的赋值
             * 操作无关紧要.
             */
            p->counter = (p->counter >> 1) + p->priority;    /*重新分配时间片*/
        read_unlock(&tasklist_lock);                         /*对进程队列解锁*/
        spin_lock_irq(&runqueue_lock);                       /* 对运行队列加锁,禁止中断*/
    }
    goto repeat_schedule;                                    /* 跳转至 repeat_schedule*/

still_running:
    c = goodness(prev, prev, this_cpu);      /* 计算当前进程权值,将计算结果赋给 c 结果赋给 c*/
    next = prev;                             /* 将当前进程 task_struct 指针赋给 next*/
    goto still_running_back;                 /* 跳转至 still_running_back */

handle_bh:
    do_bottom_half();                        /* 执行 bottom half 服务*/
    goto handle_bh_back;                     /*跳转回 handle_bh_back */

move_rr_last:
     if (!prev->counter) {                   /*若当前进程的 counter 不为零,则重新
```

```
                                        * 分配时间片(counter),并将其挂到队列尾部.*/
            prev->counter = prev->priority;
            move_last_runqueue(prev);
        }
        goto move_rr_back;                                  /* 跳转回 move_rr_back*/

scheduling_in_interrupt:
        printk("Scheduling in interrupt\n");
        *(int *)0 = 0;
        return;
}

/*************************************************************************/
/* 位置 kernel/sched.c*/
static inline int goodness(struct task_struct * p, struct task_struct * prev, int this_cpu)
{
        int policy = p->policy;
        int weight;

        if (policy & SCHED_YIELD) {                 /* 如果已经释放了 CPU,那么返回 0*/
            p->policy = policy & ~SCHED_YIELD;      /* 清除 SCHED_YIELD 位标志*/
            return 0;                               /* 返回 0*/
        }

        /*
         * Realtime process, select the first one on the
         * runqueue (taking priorities within processes
         * into account).
         */
        /* 若为实时进程,选择运行队列的第一个实时进程*/
        if (policy != SCHED_OTHER)                          /* 如果是实时进程*/
            return 1000 + p->rt_priority;                   /* 计算权值,并以该值返回*/

        /*
         * Give the process a first-approximation goodness value
         * according to the number of clock-ticks it has left.
         *
         * Don't do any other calculations if the time slice is
         * over..
         */
        /* 根据进程剩余时间片数赋给进程一个最接近的 goodness(权)值
         *
         * 如果时间片用完了,不要再作其它计算...
         */
```

```
    weight = p->counter;                          /* 非实时进程,时间片剩余值即权值(优先级)*/
    if (weight) {                                 /* 如果进程时间片未用完,执行条件语句
                                                   * 否则,直接返回 weight=0*/

#ifdef __SMP__              /*SMP 处理*/
        /* Give a largish advantage to the same processor...   */
        /* (this is equivalent to penalizing other processors) */
        if (p->processor == this_cpu)
            weight += PROC_CHANGE_PENALTY;
#endif

        /* .. and a slight advantage to the current thread */
        /* 给当前线程一点优惠*/
        if (p->mm == prev->mm)
                                                  /* 如果是内核线程或用户空间与当前进程相同,
                                                   * 因而不必切换用户空间
                                                   * 给予权值额外加一的"优惠" */
            weight += 1;
        weight += p->priority;                    /* 令权值= weight+p->priority*/
    }

    return weight;                                /* 返回权值*/
}

/*********************************************************************/
/* 位置 include/asm_i386/mmu_context.h*/
static inline void switch_mm(struct mm_struct *prev, struct mm_struct *next, unsigned cpu)
{

    if (prev != next) {                           /* 若当前进程与将要切换到的新进程不是同
                                                   * 一进程 */
        /*
         * Re-load LDT if necessary 如果需要,重载 LDT
         */
        if (prev->segments != next->segments)     /* 若新进程有自己的 LDT*/
            load_LDT(next);                       /* 则载入新进程的 LDT*/
  #ifdef CONFIG_SMP                               /* SMP 操作,略去*/
    ......................
  #endif
        set_bit(cpu, &next->cpu_vm_mask);         /* 新进程的 cpu_vm_mask 置位*/
        /* Re-load page tables 重载页表*/
        asm volatile("movl %0,%%cr3": :"r" (__pa(next->pgd)));
                                                  /* 将新进程页面目录的起始物理地址装入到
                                                   *  控制寄存器 CR3 中*/
```

```
  #ifdef CONFIG_SMP                                    /* SMP 操作,略去*/
    ……
  #endif
    }
    }
/*************************************************************************/
/* 位置 kernel/sched.c*/
static inline void add_to_runqueue(struct task_struct * p)
{                                                      /*把 p 进程插入到 init_task 进程的后面*/
    struct task_struct *next = init_task.next_run;/*令 next 指向 init_task 进程的下一个进程*/

    p->prev_run = &init_task;                          /*令 p 进程的向前指针指向 init_task 进程*/
    init_task.next_run = p;                            /*令 init_task 进程的向后指针指向 p 进程*/
    p->next_run = next;              /*令 p 进程的向后指针指向 init_task 进程先前的下一个进程*/
    next->prev_run = p;                              /*令 next 指向的进程的向前指针指向 p 进程*/
    nr_running++;                                                  /*运行队列进程计数增一*/
}
/*************************************************************************/
/* 位置 kernel/sched.c*/
static inline void del_from_runqueue(struct task_struct * p)
{                                                            /*将 p 进程从进程队列中摘除*/
    struct task_struct *next = p->next_run;              /*令 next 指向 p 进程的下一个进程*/
    struct task_struct *prev = p->prev_run;              /*令 prev 指向 p 进程的上一个进程*/

    nr_running--;                                                  /*运行队列进程计数减一*/
    next->prev_run = prev;                             /*令 next 进程的向前指针指向 prev 进程*/
    prev->next_run = next;                             /*令 prev 进程的向后指针指向 next 进程*/
    p->next_run = NULL;                                          /*令 p 进程的向后指针为空*/
    p->prev_run = NULL;                                          /*令 p 进程的向前指针为空*/
}
/*************************************************************************/
/* 位置 kernel/sched.c*/
static inline void move_last_runqueue(struct task_struct * p)
{                                                            /*把 p 进程移到运行队列尾部*/
    struct task_struct *next = p->next_run;              /*令 next 指向 p 进程的下一个进程*/
    struct task_struct *prev = p->prev_run;              /*令 prev 指向 p 进程的上一个进程*/

                                                         /* 先将 p 进程从队列中摘除*/
    next->prev_run = prev;
    prev->next_run = next;
                                                                 /* 将 p 进程加到队尾 */
    p->next_run = &init_task;                          /*令 p 进程的向后指针指向 init_task 进程*/
    prev = init_task.prev_run;                         /*令 prev 指向 init_task 进程的前一个进程*/
    init_task.prev_run = p;                            /*令 init_task 进程的向前指针指向 p 进程*/
```

```
    p->prev_run = prev;                                  /*令 p 进程的向后指针指向 prev 进程*/
    prev->next_run = p;                                  /*令 prev 进程的向后指针指向 p 进程*/
}
/*************************************************************************/
/* 位置 kernel/sched.c*/
static inline void move_first_runqueue(struct task_struct * p)
{                                                            /*把 p 进程移到运行队列首部*/
    struct task_struct *next = p->next_run;              /*令 next 指向 p 进程的下一个进程*/
    struct task_struct *prev = p->prev_run;              /*令 prev 指向 p 进程的上一个进程*/

                                                          /* 先将 p 进程从队列中摘除*/
    next->prev_run = prev;
    prev->next_run = next;
                                                                /* 将 p 进程加到队首*/
    p->prev_run = &init_task;                          /*令 p 进程的向前指针指向 init_task 进程*/
    next = init_task.next_run;                        /*令 next 指向 init_task 进程的后一个进程*/
    init_task.next_run = p;                            /*令 init_task 进程的向后指针指向 p 进程*/
    p->next_run = next;                                  /*令 p 进程的向后指针指向 next 进程*/
    next->prev_run = p;                                  /*令 next 进程的向前指针指向 p 进程*/
                                          }
```

6.2 nanosleep, pause 及时钟函数

```
/*************************************************************************/
/* 位置 kernel/sched.c*/
asmlinkage int sys_nanosleep(struct timespec *rqtp, struct timespec *rmtp)
{
    struct t;
    unsigned long expire;                              /* 保存剩余睡眠时间*/

    if(copy_from_user(&t, rqtp, sizeof(struct timespec)))
    return -EFAULT;                                    /* 将 rgtp 指向的 timespec 结构复制给 t,
                                                        *  若复制出错返回错误码*/

    if (t.tv_nsec >= 1000000000L || t.tv_nsec < 0 || t.tv_sec < 0)
        return -EINVAL;                                /* 若睡眠时间设置有误,返回错误码*/


    if (t.tv_sec == 0 && t.tv_nsec <= 2000000L &&
        current->policy != SCHED_OTHER)
     {                                                 /* 若要求睡眠的进程是实时进程,且睡眠时间
                                                        * 小于 2 毫秒*/
        udelay((t.tv_nsec + 999) / 1000);             /* 那么,仅作延时操作,并不真正睡眠*/
        return 0;                                      /* 返回*/
    }
```

```
    expire = timespec_to_jiffies(&t) +(t.tv_sec || t.tv_nsec);
                                                    /* 把睡眠时间换算成时钟中断的次数*/
                                                    /*(t.tv_sec || t.tv_nsec)值为 0 或 1*/
    current->state = TASK_INTERRUPTIBLE;            /* 设当前进程为可中断睡眠态*/
    expire = schedule_timeout(expire);              /* 进入睡眠,醒来时将剩余的睡眠时间赋
                                                    * 给 expire*/
    if (expire) {                                   /* 若进程唤醒后睡眠时间有剩余*/
        if (rmtp) {
            jiffies_to_timespec(expire, &t);      /* 将剩余时间赋给 t 所指向的 timespec 结构*/
            if (copy_to_user(rmtp, &t, sizeof(struct timespec)))/*将 t 的值复制到 rmtp*/
                return -EFAULT;                     /* 复制出错,返回错误码*/
        }
        return -EINTR;                              /* 若 rmtp 指针为空,返回错误码*/
    }
    return 0;                                       /* 正常返回*/
}
/*****************************************************************************/
/* 位置 kernel/sched.c*/
signed long schedule_timeout(signed long timeout)
{
    struct timer_list timer;                        /* 定时器*/
    unsigned long expire;                           /* 表示睡眠的到点时间*/

    switch (timeout)
    {
    case MAX_SCHEDULE_TIMEOUT:                      /* 若睡眠时间太长,长到无法用带符号整数
                                                    * 表达,则在这里视为"无限期睡眠"*/
        schedule();                                 /* 直接转入进程调度*/
        goto out;                                   /* 跳转至 out*/
    default:
        if (timeout < 0)                            /* 若睡眠时间小于 0,打印出错提示*/
        {
            printk(KERN_ERR "schedule_timeout: wrong timeout "
                "value %lx from %p\n", timeout,
                __builtin_return_address(0));
            goto out;                               /* 跳转至 out*/
        }
    }

    expire = timeout + jiffies;                     /* 计算睡眠到点(即被唤醒)时间*/

    init_timer(&timer);                             /* 初始化定时器*/
    timer.expires = expire;                         /* 对定时器数据赋值*/
```

```
    timer.data = (unsigned long) current;
    timer.function = process_timeout;

    add_timer(&timer);                          /* 将定时器加入定时器队列*/
    schedule();                                 /* 转入进程调度*/
    del_timer(&timer);                          /* 进程唤醒后从定时器队列摘除定时器*/

    timeout = expire - jiffies;                 /* 计算剩余睡眠时间*/

 out:
    return timeout < 0 ? 0 : timeout;           /* 返回剩余睡眠时间*/
}
/*****************************************************************/
/* 位置 kernel/timer.c*/
static inline void internal_add_timer(struct timer_list *timer)
{
        unsigned long expires = timer->expires;         /* 保存定时的时间*/
        unsigned long idx = expires - timer_jiffies;    /* 保存定时的间隔*/
        struct list_head * vec;

        if (idx < TVR_SIZE) {                           /* 若定时间隔小于 2^8,则
                int i = expires & TVR_MASK;              * 插入 tv1 的第 i 个链表*/
                vec = tv1.vec + i;
        } else if (idx < 1 << (TVR_BITS + TVN_BITS)) { /* 若定时间隔在 2^8~2^14 之间,
                int i = (expires >> TVR_BITS) & TVN_MASK;* 则插入 tv2 的第 i 个链表*/
                vec = tv2.vec + i;
        } else if (idx < 1 << (TVR_BITS + 2 * TVN_BITS)) {/* 若定时间隔在 2^14~2^20 之
                int i = (expires >> (TVR_BITS + TVN_BITS)) & TVN_MASK;* 间, 则插入 tv3
                vec =  tv3.vec + i;                      * 的第 i 个链表*/
        } else if (idx < 1 << (TVR_BITS + 3 * TVN_BITS)) {/* 若定时间隔在 2^20~2^26 之
                int i = (expires >> (TVR_BITS + 2 * TVN_BITS)) & TVN_MASK; * 间, 则插
                vec = tv4.vec + i;                       * 入 tv4 的第 i 个链表*/
        } else if ((signed long) idx < 0) {
                /* can happen if you add a timer with expires == jiffies,
                 * or you set a timer to go off in the past
                 */
                vec = tv1.vec + tv1.index;
        } else if (idx <= 0xffffffffUL) {               /* 若定时间隔小于 2^32,则插入
                int i = (expires >> (TVR_BITS + 3 * TVN_BITS)) & TVN_MASK;
                vec = tv5.vec + i;                       * tv5 的第 i 个链表*/
        } else {
                /* Can only get here on architectures with 64-bit jiffies */
                INIT_LIST_HEAD(&timer->list);           /* 初始化定时器队列表头*/
                return;                                  /* 返回*/
```

```
        }
        /*
         * Timers are FIFO!      定时器是先进先出的!
         */
        list_add(&timer->list, vec->prev);                  /* 执行链表插入操作*/
}

/***********************************************************************/
/* 位置 kernel/timer.c*/
static inline void run_timer_list(void)
{
        spin_lock_irq(&timerlist_lock);                     /* 对定时器队列加锁*/
        while ((long)(jiffies - timer_jiffies) >= 0) {      /* 循环处理,定时间隔>=0 时*/
                struct list_head *head, *curr;
                if (!tv1.index) {                   /* 若定时器队列 tv1 中已没有待处理的任务*/
                        int n = 1;
                        do {                        /* 设置循环,从 tv[n]搬运一组链表,分散插入到
                                cascade_timers(tvecs[n]);       * tv[n-1]的各个链表中*/
                        } while (tvecs[n]->index == 1 && ++n < NOOF_TVECS);
                }
repeat:                                             /* 循环,激活所有在此时应唤醒的定时器*/
                head = tv1.vec + tv1.index;         /* head 指向应唤醒定时器队列的链表头*/
                curr = head->next;                  /* curr 依次指向队列中的定时器*/
                if (curr != head) {
                        struct timer_list *timer;
                        void (*fn)(unsigned long);
                        unsigned long data;

                        timer = list_entry(curr, struct timer_list, list);/* 获取定时器指针*/
                        fn = timer->function;               /*定时器指定的处理函数指针*/
                        data= timer->data;                  /* 处理函数的参数*/

                        detach_timer(timer);                /* 从队列摘除定时器*/
                        timer->list.next = timer->list.prev = NULL;
                        timer_enter(timer);
                        spin_unlock_irq(&timerlist_lock);   /* 对定时器队列解锁*/
                        fn(data);                           /* 执行定时器指定的处理函数*/
                        spin_lock_irq(&timerlist_lock);     /* 对定时器队列加锁,禁止中断*/
                        timer_exit();                       /* 清除当前定时器*/
                        goto repeat;                        /* 转向下一轮循环*/
                }
                ++timer_jiffies;  /* 到下次时钟中断服务时 timer_jiffies 增加一个单位,以保持插入*
                                定时器操作 internel_add_timer 中计算
                                * idx=expires-timer_jiffies 的正确性*/
```

```
                tv1.index = (tv1.index + 1) & TVR_MASK;
                                /* 下次时钟中断时,应对第 tv1.index + 1 条链表操作*/
        }
        spin_unlock_irq(&timerlist_lock);           /* 对定时器队列解锁,允许中断*/
}
/*****************************************************************************/
/* 位置 arch/i386/kernel/sys_i386.c*/
asmlinkage int sys_pause(void)
{
    current->state = TASK_INTERRUPTIBLE;            /* 设当前进程状态为可中断睡眠态*/
    schedule();                                     /* 转入进程调度*/
    return -ERESTARTNOHAND;                         /* 返回错误码-ERESTARTNOHAND*/
}

/*****************************************************************************/
/* 位置 arch/i386/kernel/time.c*/
static inline void do_timer_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
#ifdef CONFIG_VISWS
    /* Clear the interrupt */
    co_cpu_write(CO_CPU_STAT,co_cpu_read(CO_CPU_STAT) & ~CO_STAT_TIMEINTR);
#endif
    do_timer(regs);                             /* 调用时钟函数,完成时钟中断时要执行的任务*/
/*
 * In the SMP case we use the local APIC timer interrupt to do the
 * profiling, except when we simulate SMP mode on a uniprocessor
 * system, in that case we have to call the local interrupt handler.
 */
#ifndef __SMP__
    if (!user_mode(regs))
        x86_do_profile(regs->eip);
#else
    if (!smp_found_config)
        smp_local_timer_interrupt(regs);
#endif

    /*
     * If we have an externally synchronized Linux clock, then update
     * CMOS clock accordingly every ~11 minutes. Set_rtc_mmss() has to be
     * called as close as possible to 500 ms before the new second starts.
     */
    /*  如果使用了外部同步 Linux 时钟,则每隔 11 分钟对 CMOS 时钟更新一次. Set_rtc_mmss()
     * 应尽量在新的调度开始前 500ms 调用 */
    if ((time_status & STA_UNSYNC) == 0 &&
```

```
        xtime.tv_sec > last_rtc_update + 660 &&
        xtime.tv_usec >= 500000 - ((unsigned) tick) / 2 &&
        xtime.tv_usec <= 500000 + ((unsigned) tick) / 2) {
         if (set_rtc_mmss(xtime.tv_sec) == 0)
              last_rtc_update = xtime.tv_sec;                    /* 更新 RTC 时间*/
         else
              last_rtc_update = xtime.tv_sec - 600;              /* 每隔 60 s 更新一次 */
    }

#ifdef CONFIG_MCA                                  /* 对应 IBM PS/2,不做解释*/
    if( MCA_bus ) {
         /* The PS/2 uses level-triggered interrupts.  You can't
         turn them off, nor would you want to (any attempt to
         enable edge-triggered interrupts usually gets intercepted by a
         special hardware circuit).  Hence we have to acknowledge
         the timer interrupt.  Through some incredibly stupid
         design idea, the reset for IRQ 0 is done by setting the
         high bit of the PPI port B (0x61).  Note that some PS/2s,
         notably the 55SX, work fine if this is removed.  */

         irq = inb_p( 0x61 );    /* read the current state */
         outb_p( irq|0x80, 0x61 );  /* reset the IRQ */
    }
#endif
}

/************************************************************************/
/* 位置 kernel/sched.c*/
void do_timer(struct pt_regs * regs)
{
    (*(unsigned long *)&jiffies)++;          /* 通过一个原子操作,更新全局变量 jeffies,记录自定
                                              * 时器装载以来已经过的滴答数*/
    lost_ticks++;                            /* 递增丢失的定时器滴答数*/
    mark_bh(TIMER_BH);                       /* 置 TIMER_BH 标志,标记时钟部分的 bottom half
                                              *操作为只要可能就运行 */
    if (!user_mode(regs))                    /* 若当前不是用户态
         lost_ticks_system++;                 * 递增系统模式下丢失的定时器滴答数*/
    if (tq_timer)                            /* 若定时器队列有任务等待执行*/
         mark_bh(TQUEUE_BH);                 /* 置 TQUEUE_BH 标志,表明定时器队列已准备好运行*/
}

/************************************************************************/
/* 位置 kernel/softirq.c*/
asmlinkage void do_bottom_half(void)
```

```
{
    int cpu = smp_processor_id();       /* 获取当前运行的 cpu 的 id 号*/

    if (softirq_trylock(cpu)) {          /* 加锁,禁止软中断*/
        if (hardirq_trylock(cpu)) {      /* 加锁, 禁止硬件中断*/
            __sti();                     /* 关中断*/
            run_bottom_halves();         /* 执行 bottom half 处理程序*/
            __cli();                     /* 开中断*/
            hardirq_endlock(cpu);        /* 解锁,允许硬件中断*/
        }
        softirq_endlock(cpu);            /* 加锁,允许软中断*/
    }
}

/*****************************************************************************/
/* 位置 kernel/sched.c*/
static void timer_bh(void)
{
    update_times();                      /* 更新系统时间*/
    run_old_timers();                    /* 处理定时器表*/
    run_timer_list();                    /* 处理定时器队列*/
}
```

6.3 系统调用总控入口

```
/*****************************************************************************/
/*位置 arch/i386/kernel/Entry.S*/

ENTRY(system_call)                                          #系统调用入口
    pushl %eax                                              # 保存 orig_eax
    SAVE_ALL                                                # 保存现场
    GET_CURRENT(%ebx)                  #利用 GET_CURRENT 宏从 ebx 中取得当前任务指针
    cmpl $(NR_syscalls),%eax                                #检查系统调用号是否超过最大值
    jae badsys                                              #超过,转 badsys
    testb $0x20,flags(%ebx)                                 # PF_TRACESYS 是否置位(被跟踪)
    jne tracesys                                            # 置位转 tracesys
    call *SYMBOL_NAME(sys_call_table)(,%eax,4) #通过系统调用跳转表进入相应的内核服务程
                                               #序,%eax 存放着系统调#用号(跳转表的相
                                               #对偏#移量),表中每一表项占 4 字节
    movl %eax,EAX(%esp)                                     # 保存返回值
    ALIGN
    .globl ret_from_sys_call                                #定义从系统调用返回过程名
    .globl ret_from_intr                                    #定义从中断返回过程名
ret_from_sys_call:
    movl SYMBOL_NAME(bh_mask),%eax                          #检查是否要执行
```

```
        andl SYMBOL_NAME(bh_active),%eax                    #bottom half 例程
        jne handle_bottom_half                              #需要转 bottom half 处理
ret_with_reschedule:
        cmpl $0,need_resched(%ebx)                          #检查进程返回用户空间后是否需要重调度
        jne reschedule                                      # 需要则转向重调度处理
        ……..
ret_from_intr:
        GET_CURRENT(%ebx)
        movl EFLAGS(%esp),%eax                              # 把堆栈中 EFLAGS 的值赋给寄存器 eax
        movb CS(%esp),%al                                   # 把堆栈中 CS 的值赋给 al
        testl $(VM_MASK | 3),%eax                          # 是 VM86 模式 或 non-supervisor 吗?
        jne ret_with_reschedule                             # 是则跳转至 ret_with_reschedule
        jmp restore_all                                     # 否则跳转至 restore_all
        ALIGN
handle_bottom_half:
        call SYMBOL_NAME(do_bottom_half)                    # 执行例程 do_bottom_half
        jmp ret_from_intr                                   # 跳转 ret_from_intr

        ALIGN
reschedule:
        call SYMBOL_NAME(schedule)                          # 执行 schedule( )重调度
        jmp ret_from_sys_call                               # 跳转 ret_from_sys_call
```

6.4 子进程的创建

```
/*********************************************************************/
/*位置 kernel/fork.c*/
int do_fork(unsigned long clone_flags, unsigned long stack_start,
            struct pt_regs *regs, unsigned long stack_size)
{
        int retval = -ENOMEM;                   /* 将 retval 赋值-ENOMEM,作为 task_struct
                                                * 结构申请失败时的返回值*/
        struct task_struct *p;
        DECLARE_MUTEX_LOCKED(sem);              /* 创建用于进程间互斥或同步的信号量*/


        if (clone_flags & CLONE_PID) {          /* 若 clone_flags 的位是置位的*/
                if (current->pid)               /* 若调用 do_fork 的当前(父)进程不是 idle
                                                 * 进程(其 pid=0)*/
                        return -EPERM;          /* 返回错误信息-EPERM*/
        }

        current->vfork_sem = &sem;              /* 将 current->vfork_sem 赋值*/
```

```
p = alloc_task_struct();                    /* 申请一个新的 task_struct 结构*/
if (!p)                                     /* 若申请失败*/
        goto fork_out;                      /* 跳转至 fork_out*/

*p = *current;                              /* 将当前(父)进程 task_struct 结构的值赋给新
                                             * 创建的(子)进程*/

retval = -EAGAIN;                           /* 将错误信息-EAGAIN 作为返回值*/
if (atomic_read(&p->user->processes) >= p->rlim[RLIMIT_NPROC].rlim_cur)
        goto bad_fork_free;                 /* 若子(新)进程所属的用户拥有的进程数已达到
                                             * 规定的限制值,则跳转至 bad_fork_free*/
                                            /* user 结构的数据成员的详细描述,请参见前面
                                             * 对 user 结构的介绍*/
atomic_inc(&p->user->__count);              /* user->__count 增一*/
atomic_inc(&p->user->processes);            /* user->processes(用户拥有的进程数)增一*/

if (nr_threads >= max_threads)              /* 若系统进程数超过最大进程数*/
        goto bad_fork_cleanup_count;        /* 则跳转至 bad_fork_cleanup_count*/

get_exec_domain(p->exec_domain);            /* 若正在执行的代码是符合 iBCS2 标准的程序,
                                             * 则增加相对应模块的引用数目   */

if (p->binfmt && p->binfmt->module)         /* 若正在执行的代码属于全局执行文件结构格式
                                            * 则增加相对应模块的引用数目   */
        __MOD_INC_USE_COUNT(p->binfmt->module);

p->did_exec = 0;                            /* 将子进程标志为尚未执行*/
p->swappable = 0;                           /* 清标志,使内存页面不可换出*/
p->state = TASK_UNINTERRUPTIBLE;            /* 将子进程的状态置为 uninterruptible*/

copy_flags(clone_flags, p);                 /* 将 clone_flags 略加修改写入 p->flags*/
p->pid = get_pid(clone_flags);              /* 调用 kernel/fork.c:get_pid()为子进程分配
                                             *  一个 pid. 若是 clone 系统调用且
                                            *  clone_flags 中 CLONE_PID 位为 1,那么父子
                                            *  进程共享一个 pid 号;否则要分配给子进程一
                                            *  个从未用过的 pid*/
p->run_list.next = NULL;                    /* 对运行队列接口 run_list 初始化*/
p->run_list.prev = NULL;

if (clone_flags & CLONE_VFORK || !(clone_flags & CLONE_PARENT)) {
                                           /* 若是 vfork 调用或 CLONE_PARENT 标志为 0*/
        p->p_opptr = current;              /* 令子进程的 p_opptr 指向父进程*/
       if   ( ! (p->ptrace & PT_PTRACED))          /* 若 PT_PTRACED 未置位*/
              p->p_pptr = current;                 /* 令子进程的 p_pptr 指向父进程*/
```

```
        }
         p->p_cptr = NULL;
         init_waitqueue_head(&p-> wait_chldexit);  /* 初始化 wait_chldexit 等待队列.
                                                   *  wait_chldexit 用于在进程结束时,或发
                                                   *  出系统调用 wait4 后,为了等待子进程结
                                                   *  束,而将自己(父进程)睡眠在该队列上*/
         p->vfork_sem = NULL;                 /* 从这里开始对 task_struck 结构中剩下的项做
                                               * 一些初始化工作,task_struck 结构中的数据成
                                               * 员的具体描述,请参考前面对 task_struck 结构
                                               * 的介绍文字*/
        spin_lock_init(&p->alloc_lock);       /* 加锁*/

        p->sigpending = 0;
        init_sigpending(&p->pending);

        p->it_real_value = p->it_virt_value = p->it_prof_value = 0;
        p->it_real_incr = p->it_virt_incr = p->it_prof_incr = 0;
        init_timer(&p->real_timer);
        p->real_timer.data = (unsigned long) p;

        p->leader = 0;
        p->tty_old_pgrp = 0;
        p->times.tms_utime = p->times.tms_stime = 0;
        p->times.tms_cutime = p->times.tms_cstime = 0;
#ifdef CONFIG_SMP
        {
                int i;
                p->cpus_runnable = ~0UL;
                p->processor = current->processor;
                /* ?? should we just memset this ?? */
                for(i = 0; i < smp_num_cpus; i++)
                        p->per_cpu_utime[i] = p->per_cpu_stime[i] = 0;
                spin_lock_init(&p->sigmask_lock);
        }
#endif
        p->lock_depth = -1;                   /* -1 = 没有锁 */
        p->start_time = jiffies;              /* 将当前的 jiffies 值作为子进程的创建时间*/
                                              /*  task_struct 结构初始化完毕*/
        retval = -ENOMEM;                     /* 将错误标志-ENOMEM 作为返回值*/
        /* copy all the process information */  /* 复制所有的进程信息*/
        if (copy_files(clone_flags, p))       /* 根据 clone_flags 复制或共享父进程的打开
                                               * 文件表*/
                goto bad_fork_cleanup;        /* 失败则跳转至 bad_fork_cleanup*/
        if (copy_fs(clone_flags, p))          /* 根据 clone_flags 复制或共享父进程的文件
```

```
                                        * 系统信息*/
        goto bad_fork_cleanup_files;    /* 失败则跳转至 bad_fork_cleanup_files*/
if (copy_sighand(clone_flags, p))       /* 根据 clone_flags 复制或共享父进程的信号
                                        * 处理句柄*/
        goto bad_fork_cleanup_fs;       /* 失败则跳转至 bad_fork_cleanup_fs*/
if (copy_mm(clone_flags, p))            /* 根据 clone_flags 复制或共享父进程的存储
                                        * 管理信息*/
        goto bad_fork_cleanup_sighand;/* 失败则跳转至
                                       * bad_fork_cleanup_sighand*/
retval = copy_thread(0, clone_flags, stack_start, stack_size, p, regs);
                                      /* 为子进程复制父进程系统空间堆栈*/
if (retval)                           /* 若系统空间堆栈复制失败*/
        goto bad_fork_cleanup_mm;     /* 跳转至 bad_fork_cleanup_mm*/
p->semundo = NULL;


p->parent_exec_id = p->self_exec_id; /* 将子进程 task_struct 结构的 self_exec_id
                                     *  赋给 parent_exec_id*/

p->swappable = 1;                    /* 新进程已经完成初始化,可以换出内存,所以
                                      * 将 p->swappable 赋 1*/
p->exit_signal = clone_flags & CSIGNAL;/* 设置系统强行退出时发出的信号*/
p->pdeath_signal = 0;                 /* 设置 p->pdeath_signal*/

p->counter = (current->counter + 1) >> 1;/* 将父进程的时间片加一再除以二后赋给子
                                         *  进程*/
current->counter >>= 1;                /* 将父进程的时间片减半*/
if (!current->counter)                 /* 若父进程的时间片耗尽*/
        current->need_resched = 1;     /* 置重调度标志*/

/*
 * Ok, add it to the run-queues and make it
 * visible to the rest of the system.
 * 把子进程加入运行队列让系统的其余部分可看到它.
 * Let it rip!
 */
retval = p->pid;                       /* 如果一切顺利,将子进程的 pid 作为返回值*/
p->tgid = retval;                      /* 将 retval 赋给 p->tgid*/
INIT_LIST_HEAD(&p->thread_group);      /* 初始化 thread_group */
write_lock_irq(&tasklist_lock);        /* 给进程队列加锁*/
if (clone_flags & CLONE_THREAD) {      /* 若 CLONE_THREAD 置位,即新创建的
                                       * 进程是线程*/
        p->tgid = current->tgid;       /* 将父进程 tgid 的值赋给子进程的 tgid*/
        list_add(&p->thread_group, &current->thread_group);
```

```
                                        /* 将子进程加入"线程组"*/
        }
        SET_LINKS(p);                   /* 将子进程的 task_struct 结构链入进程队列*/
        hash_pid(p);                    /* 将子进程的 task_struct 结构链入进程 hash 表*/
        nr_threads++;                   /* 系统进程计数递增一*/
        write_unlock_irq(&tasklist_lock);   /* 解除对进程队列的封锁*/

        if (p->ptrace & PT_PTRACED)     /* 若 PT_PTRACED 标志位置位*/
                send_sig(SIGSTOP, p, 1);    /* 发 SIGSTOP 信号给子进程*/

        wake_up_process(p);             /* 最后做这件事,唤醒子进程 */
        ++total_forks;                  /* total_forks 增一*/

fork_out:
        if ((clone_flags & CLONE_VFORK) && (retval > 0))
                down(&sem);             /* 若是 vfork()调用 do_fork,发 down 信号*/
        return retval;                  /* 退出 do_fork(),返回 retval 值*/

bad_fork_cleanup_sighand:
        exit_sighand(p);                /* 处理子进程 task_struct 结构与信号处理相关
                                         * 的数据成员, 并删除信号队列中与子进程相
                                         * 关的信号量*/
bad_fork_cleanup_fs:
        exit_fs(p); /* blocking */      /* 处理子进程 task_struct 结构与文件系统信息相
                                        * 关的数据成员*/
bad_fork_cleanup_files:
        exit_files(p); /* blocking */   /* 处理子进程 task_struct 结构与打开文件表相关
                                         * 的数据成员, 并释放子进程的 files_struct 结
                                         * 构*/
bad_fork_cleanup:
        put_exec_domain(p->exec_domain); /* 若正在执行的代码是符合 iBCS2 标准的程序,则
                                        * 减少相对应模块的引用数目 */
        if (p->binfmt && p->binfmt->module) /* 若正在执行的代码属于全局执行文件结构格式
                                        * 则减少相对应模块的引用数目 */
                __MOD_DEC_USE_COUNT(p->binfmt->module);
bad_fork_cleanup_count:
        atomic_dec(&p->user->processes);  /* 子进程所属用户拥有的进程计数减 1*/
        free_uid(p->user);              /* 清除子进程在 user 队列中的信息*/
bad_fork_free:
        free_task_struct(p);            /* 释放子进程的 task_struct 结构*/
        goto fork_out;                  /* 跳转至标号 fork_out 处*/
}

/*********************************************************************/
```

```
/*位置 kernel/fork.c*/
static int get_pid(unsigned long flags)
{
        static int = PID_MAX;                   /*将可分配给进程的最的 id 值赋给 next_safe*/
        struct task_struct *p;
        int pid;

        if (flags & CLONE_PID)                  /*若父子进程共享 id 号,返回当前进程的 id 号*/
                return current->pid;

        spin_lock(&_lock);                                          /*给 last_pid 加锁*/
        if((++last_pid) & 0xffff8000) {   /* 上次调用 get_pid 子进程分配的 id 号在允许范围内*/
                                          /* 将 last_pid 赋值为 300,原因是 Linux 系统有一些内核
                                           * 线程常驻内存,这些内核线程的 id 号一般在 300 以
                                           * 内,last_pid 直接赋值为 300,可以避免这一段无谓的检
                                           * 索,提高分配 id 号的效率*/
                                          /*
                last_pid = 300;           /* Skip daemons etc. */
                goto inside;
        }
        if(last_pid >= next_safe) {                                 /*检查 last_pid 是否超限*/
inside:
                next_safe = PID_MAX;            /*将可分配给进程的最的 id 值赋给 next_safe*/
                read_lock(&tasklist_lock);      /*对进程队列加锁,防止两个进程分配同一 id 号*/
        repeat:
                for_each_task(p) {              /*自增 last_pid,并检查是否与已有进程的 id 冲
                                                *突,直到找到一个未分配给已有进程的 id 为止*/
                        if(p->pid == last_pid   ||
                           p->pgrp == last_pid  ||
                           p->tgid == last_pid  ||
                           p->session == last_pid) {
                                if(++last_pid >= next_safe) {
                                        if(last_pid & 0xffff8000)
                                                last_pid = 300;
                                        next_safe = PID_MAX;
                                }
                                goto repeat;
                        }
                                                        /* 再次检查新分配的 pid,并重新确定
                                                        * next_safe 的值为下次运行作准备*/
                        if(p->pid > last_pid && next_safe > p->pid)
                                next_safe = p->pid;
                        if(p->pgrp > last_pid && next_safe > p->pgrp)
                                next_safe = p->pgrp;
```

```
                        if(p->session > last_pid && next_safe > p->session)
                                next_safe = p->session;
                }
                read_unlock(&tasklist_lock);                    /* 对进程队列解锁*/
        }
        pid = last_pid;                                         /*将得到的 id 值赋给 pid*/
        spin_unlock(&lastpid_lock);                             /*对 last_pid 解锁*/

        return pid;                                             /*返回得到的新进程 id*/
}
/*********************************************************************/
/*位置 kernel/fork.c*/
static inline int copy_fs(unsigned long clone_flags, struct task_struct * tsk)
{
        if (clone_flags & CLONE_FS) {                   /* 若 CLONE_FS 置位,表明子进程共享父进
                                                         * 程的文件系统信息(fs_struct 结构)*/
                atomic_inc(&current->fs->count);        /* 父进程文件系统信息引用计数增一*/
                return 0;                               /* 成功返回*/
        }
        tsk->fs = __copy_fs_struct(current->fs);        /* 若不共享,则复制父进程的 fs_struct 结构
                                                         * 给子进程*/
        if (!tsk->fs)                                   /* 复制失败*/
                return -1;                              /* 返回-1*/
        return 0;                                       /* 成功返回 0*/
}

/*********************************************************************/
/*位置 kernel/fork.c*/
static int copy_mm(unsigned long clone_flags, struct task_struct * tsk)
{
        struct mm_struct * mm, *oldmm;
        int retval;
                                                /* 初始化子进程 task_struct 结构与内存存储
        tsk->min_flt = tsk->maj_flt = 0;         * 信息相关的数据项*/
        tsk->cmin_flt = tsk->cmaj_flt = 0;      /* task_struct 结构数据成员的详细描述,请
        tsk->nswap = tsk->cnswap = 0;            * 参见前面对 task_struct 结构的介绍*/

        tsk->mm = NULL;                         /* 初始化子进程的 mm 指针*/
        tsk->active_mm = NULL;                  /* 初始化子进程的 active_mm 指针*/

        /*
         * Are we cloning a kernel thread?      我们是否克隆了一个内核线程?
         *
         * We need to steal a active VM for that.. 我们需要借用一个 active VM 给它
```

```
         */
        oldmm = current->mm;                    /* 将父进程的指向 mm_struct 结构的指针
                                                 * 赋给 oldmm*/
        if (!oldmm)                             /* 若 oldmm 为空*/
                return 0;                       /* 返回 0*/

        if (clone_flags & CLONE_VM) {           /* CLONE_VM 置位,即父子进程共享存储
                                                 * 管理信息*/
                atomic_inc(&oldmm->mm_users);/* 父进程 mm_struct 结构引用计数增一*/
                mm = oldmm;                     /* 将父进程的指向 mm_struct 结构的指针
                                                 * 赋给 mm*/
                goto good_mm;                   /* 跳转至 good_mm*/
        }

        retval = -ENOMEM;                       /* 将错误信息-ENOMEM 赋给返回值*/
        mm = allocate_mm();                     /* 若不共享,申请分配一个 mm_struct 结构*/
        if (!mm)                                /* 若分配失败*/
                goto fail_nomem;                /* 跳转至 fail_nomem */

        /* Copy the current MM stuff.. */       /* 复制父进程的 MM 结构的信息*/
        memcpy(mm, oldmm, sizeof(*mm));     /* 将父进程的 mm_struct 结构复制给子进程*/
        if (!mm_init(mm))                       /* 对子进程 mm_struct 结构作初始化处理*/
                goto fail_nomem;                /* 若失败,跳转至 fail_nomem*/

        down_write(&oldmm->mmap_sem);           /* 对父进程相应页面发写操作睡眠信号(禁止
                                                /* 写)*/
        retval = dup_mmap(mm);                  /* 复制 vm_area_struct 结构和页面映射表*/
        up_write(&oldmm->mmap_sem);             /* 对父进程相应页面发写操作唤醒信号*/

        if (retval)                             /* 若复制失败 */
                goto free_pt;                   /* 跳转至 free_pt */

        /*
         * child gets a private LDT (if there was an LDT in the parent)
         * 子进程获得一个私有的 LDT(如果父进程拥有 LDT 的话)
         */
        copy_segments(tsk, mm);                 /* 复制 LDT*/

        if (init_new_context(tsk,mm))           /* 初始化上下文*/
                goto free_pt;                   /* 若初始化失败,跳转至 free_pt */

good_mm:
        tsk->mm = mm;                           /* 将 mm 赋给子进程 task_struct 结构的 mm*/
        tsk->active_mm = mm;            /* 将 mm 赋给子进程 task_struct 结构的 active_mm*/
```

```
        return 0;                               /* 成功返回*/

free_pt:
        mmput(mm);                              /* 释放 mm 所指向的 mm_struct 结构*/
fail_nomem:
        return retval;                          /* 失败返回*/
}

/*****************************************************************************/
/*位置 kernel/fork.c*/
static int copy_files(unsigned long clone_flags, struct task_struct * tsk)
{
        struct files_struct *oldf, *newf;
        struct file **old_fds, **new_fds;       /*父子进程的文件描述符数组*/
        int open_files, nfds, size, i, error = 0;

        /*
         * A background process may not have any files ...
         * 后台进程可能没有文件资源
        */
        oldf = current->files;                  /* 将父进程的 files_struct 结构指针赋给 oldf*/
        if (!oldf)                              /* 若父进程的 files_struct 结构指针为空*/
                goto out;                       /* 跳转至 out*/

        if (clone_flags & CLONE_FILES) {        /* 若 CLONE_FILES 置位,即父子进程共享打开
                                                 * 文件表*/
                atomic_inc(&oldf->count);       /* 父进程 files_struct 结构引用计数增一*/
                goto out;                       /* 跳转至 out*/
        }

        tsk->files = NULL;                      /* 初始化子进程 files_struct 结构指针*/
        error = -ENOMEM;                        /* 将错误码-ENOMEM 赋给变量 error*/
        newf = kmem_cache_alloc(files_cachep, SLAB_KERNEL); /* 不共享,申请新的
                                                             * files_struct 结构*/
        if (!newf)                              /* 若申请失败*/
                goto out;                       /* 跳转至 out*/

        atomic_set(&newf->count, 1);            /* 将新分配的 files_struct 结构引用计数置 1*/

        newf->file_lock     = RW_LOCK_UNLOCKED; /* 以下是对 files_struct 结构的
        newf->next_fd       = 0;                /* 设置, files_struct 结构的详
        newf->max_fds       = NR_OPEN_DEFAULT;  /* 细描述,可参考"文件系统"*/
        newf->max_fdset     = __FD_SETSIZE;
        newf->close_on_exec = &newf->close_on_exec_init;
```

```
newf->open_fds       = &newf->open_fds_init;
newf->fd             = &newf->fd_array[0];


size = oldf->max_fdset;                         /* 将父进程的 max_fdset 赋给变量 size*/
if (size > __FD_SETSIZE) {                      /* size 大于设定值*/
        newf->max_fdset = 0;                    /*设 max_fdset 为 0*/
        write_lock(&newf->file_lock);           /* 对子进程打开文件表加锁,禁止写操作*/
        error = expand_fdset(newf, size-1); /* 按 size 值扩大空间*/
        write_unlock(&newf->file_lock);         /* 对子进程打开文件表解锁*/
        if (error)                              /* 若出错*/
                goto out_release;               /* 跳转至 out_release*/
}
read_lock(&oldf->file_lock);                    /* 对父进程打开文件表加锁*/

open_files = count_open_files(oldf, size);   /* 计算父进程打开文件数并将其赋给
                                                * open_files*/

/*
 * Check whether we need to allocate a larger fd array.
 * Note: we're not a clone task, so the open count won't
 * change.
 */
/*
 * 检查是否需要申请分配一个更大的 fd 数组.
 * 注意: 这不是一个克隆任务,所以打开计数不会改变.
 */

nfds = NR_OPEN_DEFAULT;
if (open_files > nfds) {                                /*若打开文件数大于缺省最大值,
                                                         *则扩展 fd 数组*/
        read_unlock(&oldf->file_lock);
        newf->max_fds = 0;                              /*初始化子进程的 max_fds*/
        write_lock(&newf->file_lock);                   /*对 file 结构加锁,禁止写操作*/
        error = expand_fd_array(newf, open_files-1);/*扩展操作*/
        write_unlock(&newf->file_lock);                 /*对 file 结构解锁,允许写操作*/
        if (error)                                      /* 扩展失败,转向 out_release*/
                goto out_release;
        nfds = newf->max_fds;
        read_lock(&oldf->file_lock);            /*对 file 结构加锁,禁止读操作*/
}
                                                /*子进程拷贝父进程的文件描述符数组*/
old_fds = oldf->fd;
new_fds = newf->fd;
```

```
        memcpy(newf->open_fds->fds_bits, oldf->open_fds->fds_bits, open_files/8);
        memcpy(newf->close_on_exec->fds_bits,        oldf->close_on_exec->fds_bits,
open_files/8);

        for (i = open_files; i != 0; i--) {          /*通过循环将父进程的 fd 数组元素拷贝给子
                                                     *进程*/
                struct file *f = *old_fds++;
                if (f)
                        get_file(f);
                *new_fds++ = f;
        }
        read_unlock(&oldf->file_lock);                  /*对 file 结构解锁,允许读操作*/

        size = (newf->max_fds - open_files) * sizeof(struct file *);/*计算扩展空间大小*/

        /* This is long word aligned thus could use a optimized version */
        memset(new_fds, 0, size);

        if (newf->max_fdset > open_files) {
                int left = (newf->max_fdset-open_files)/8;
                int start = open_files / (8 * sizeof(unsigned long));

                memset(&newf->open_fds->fds_bits[start], 0, left);
                memset(&newf->close_on_exec->fds_bits[start], 0, left);
        }

        tsk->files = newf;                              /*令新进程的 task_struct 结构的 files
                                                          *指向新申请的文件结构*/
        error = 0;
out:
        return error;                                   /*错误返回*/

out_release:                                            /*释放新申请的文件结构的相关域*/
        free_fdset (newf->close_on_exec, newf->max_fdset);
        free_fdset (newf->open_fds, newf->max_fdset);
        kmem_cache_free(files_cachep, newf);
        goto out;
}

/*********************************************************************/
/*位置 kernel/fork.c*/
static inline int copy_sighand(unsigned long clone_flags, struct task_struct * tsk)
{
```

```
        struct signal_struct *sig;

        if (clone_flags & CLONE_SIGHAND) {      /* 若 CLONE_SIGHAND 置位,即父子进程共
                                                 * 享信号处理程序*/
                atomic_inc(&current->sig->count);/* 父进程 signal_struct 结构引用计数增一*/
                return 0;                       /* 返回 0*/
        }
        sig = kmem_cache_alloc(sigact_cachep, GFP_KERNEL);/* 不共享,申请分配新的
                                                           *signal_struct 结构*/
        tsk->sig = sig;                         /* 将新分配的 signal_struct 结构指针赋给子
                                                 * 进程的 sig */
        if (!sig)                               /* 若分配失败*/
                return -1;                      /* 返回-1*/
        spin_lock_init(&sig->siglock);          /* 加锁*/
        atomic_set(&sig->count, 1);             /* 给新 signal_struct 结构引用计数置一*/
        memcpy(tsk->sig->action, current->sig->action, sizeof(tsk->sig->action));
                                                /* 复制 action 数组*/
        return 0;                               /* 返回 0*/
}

/*********************************************************************/
/*位置 arch/i386/kernel/process.c*/
int copy_thread(int nr, unsigned long clone_flags, unsigned long esp,
    struct task_struct * p, struct pt_regs * regs)
{
    struct pt_regs * childregs;                 /* pt_regs 指针,指向新的 pt_regs 结构*/

    childregs = ((struct pt_regs *) (THREAD_SIZE + (unsigned long) p)) - 1;
                                                /* 通过计算使 childregs 指向子进程系统空间
                                                 * 堆栈中的 pt_regs 结构 */
    *childregs = *regs;                         /* 复制父进程的 pt_regs 结构给子进程*/
    childregs->eax = 0;                         /* 将结构中的 eax 置 0,当子进程受调度"恢复"
                                                 * 运行,从系统调用"返回"时,这就是返回值*/
    childregs->esp = esp;                       /* 设置子进程在用户空间的堆栈的位置*/

    p->thread.esp = (unsigned long) childregs;  /* 子进程 pt_regs 结构始址赋给子进程
                                                 * task_struct 结构的 thread.esp*/
    p->thread.esp0 = (unsigned long) (childregs+1);/*使 p->thread.esp0 指向子进程系统空
                                                    * 间堆栈的顶端*/

    p->thread.eip = (unsigned long) ret_from_fork; /* 设置子进程下一次被切换运行时的切入点
                                                    *  这里切入点设置为 ret_from_fork */

    savesegment(fs,p->thread.fs);               /* 将段寄存器 fs 的值保存在 p->thread.fs 中*/
```

```
    savesegment(gs,p->thread.gs);            /* 将段寄存器 gs 的值保存在 p->thread.gs 中*/

    unlazy_fpu(current);                     /* 为 i387 浮点处理器而作的设置,与 i386
    p->thread.i387 = current->thread.i387;    * 无关*/

    return 0;                                /* 返回 0 */
}

/*********************************************************************/
/*位置 arch/i386/kernel/process.c*/
asmlinkage int sys_fork(struct pt_regs regs)
{
    return do_fork(SIGCHLD, regs.esp, &regs);      /* 调用 do_fork 完成进程创建工作*/
}

/*********************************************************************/
/*位置 arch/i386/kernel/process.c*/
asmlinkage int sys_clone(struct pt_regs regs)
{
    unsigned long clone_flags;
    unsigned long newsp;

    clone_flags = regs.ebx;
    newsp = regs.ecx;                        /* 将为新进程分配的用户堆栈首址赋给 newsp*/
    if (!newsp)                              /* 若 newsp 为空,则将父进程的栈址赋给 newsp*/
        newsp = regs.esp;
    return do_fork(clone_flags, newsp, &regs);          /* 调用 do_fork 完成进程创建工作*/
}

/*********************************************************************/
/*位置 arch/i386/kernel/process.c*/
asmlinkage int sys_vfork(struct pt_regs regs)
{
    return do_fork(CLONE_VFORK | CLONE_VM | SIGCHLD, regs.esp, &regs);
/* 调用 do_fork 完成进程创建工作, CLONE_VFORK 说明子进程创建后,一定要先执行,而让父进程等待*/
}
```

6.5 子进程的装入和执行

```
/*********************************************************************/
/*位置 fs/exec.c*/
asmlinkage int sys_execve(struct pt_regs regs)
{
    int error;
    char * filename;                         /* 用于指向一个文件名字符串*/
```

```
    filename = getname((char *) regs.ebx);          /* 将文件名从用户空间取入核心空间*/
    error = PTR_ERR(filename);                      /* 将错误码赋给变量 error*/
    if (IS_ERR(filename))                           /* 若取文件名操作失败*/
        goto out;                                   /*  跳转至 out*/
    error = do_execve(filename, (char **) regs.ecx, (char **) regs.edx, &regs);
                                                    /* 调用 do_execve 完成主要工作*/
    if (error == 0)                                 /* 若程序执行成功*/
        current->flags &= ~PF_DTRACE;               /* 清 ptrace 位*/
    putname(filename);                              /* 释放 getname 时所申请的内核空间*/
out:
    return error;                                   /* 返回错误码*/
}

/*********************************************************************/
/*位置 fs/namei.c*/
char * getname(const char * filename)
{
    char *tmp, *result;

    result = ERR_PTR(-ENOMEM);                      /* 将错误码-ENOMEM 赋给局部变量 result*/
    tmp = __getname();                              /* 分配一个物理页面作为缓冲区*/
    if (tmp)  {                                     /* 若分配成功*/
        int retval = do_getname(filename, tmp);/* 从用户空间拷贝字符串到缓冲区*/

        result = tmp;                               /* 将__getname()返回值赋给 result*/
        if (retval < 0) {                           /* 拷贝字符串操作失败*/
            putname(tmp);                           /* 释放为文件名分配的缓冲区*/
            result = ERR_PTR(retval);               /*  将错误码赋给 result*/
        }
    }
    return result;                                  /* 返回 result*/
}
/*********************************************************************/
/*位置 fs/exec.c*/
int do_execve(char * filename, char ** argv, char ** envp, struct pt_regs * regs)
{
        struct linux_binprm bprm;
        struct file *file;
        int retval;
        int i;

        file = open_exec(filename);                         /* 打开可执行文件,获取该文件的
                                                             * file 结构*/
```

```
        retval = PTR_ERR(file);                         /*将文件指针以错误码形式赋给 retval*/
        if (IS_ERR(file))                                /* 若文件指针有错*/
                return retval;                          /* 返回出错的文件指针*/

        bprm.p = PAGE_SIZE*MAX_ARG_PAGES-sizeof(void *); /* 获取参数区长度*/
        memset(bprm.page, 0, MAX_ARG_PAGES*sizeof(bprm.page[0]));
                                                        /* 将存放参数的页面清零*/
                                                        /* 以下对 linux_binprm 结构的
        bprm.file = file;                                * 其它项作初始化*/
        bprm.filename = filename;
        bprm.sh_bang = 0;
        bprm.loader = 0;
        bprm.exec = 0;
        if ((bprm.argc = count(argv, bprm.p / sizeof(void *))) < 0) {/* 若计算参数个数出错*/
                allow_write_access(file);                /* 允许对文件作写操作*/
                fput(file);                              /* 释放 file 结构*/
                return bprm.argc;                        /* 将 bprm.argc 作为返
                                                          * 回值返回*/
        }

        if ((bprm.envc = count(envp, bprm.p / sizeof(void *))) < 0) {/*若计算环境个数出错*/
                allow_write_access(file);                /* 允许对文件作写操作*/
                fput(file);                              /* 释放 file 结构*/
                return bprm.envc;                        /* 将 bprm.envc 作为返
                                                          * 回值返回*/
        }

retval = prepare_binprm(&bprm);                         /* 对数据结构 bprm 作进一步准备,
                                                         * 并进行访问权限等内容的检测,从
                                                         * 可执行文件中读开头的 128 个字
                                                         * 节到 bprm 中的缓冲区*/
        if (retval < 0)                                 /* 若出错*/
                goto out;                               /* 跳转至 out */

        retval = copy_strings_kernel(1, &bprm.filename, &bprm);/* 把可执行文件路径名从用
                                                          * 户空间复制到核心空间*/
        if (retval < 0)                                 /* 若出错*/
                goto out;                               /* 跳转至 out */

        bprm.exec = bprm.p;                             /* 将参数区长度赋给 bprm.exec*/
        retval = copy_strings(bprm.envc, envp, &bprm); /*  将环境变量从用户空间复制到核心
                                                          * 空间*/
        if (retval < 0)                                 /* 若出错*/
                goto out;                               /* 跳转至 out */
```

```
        retval = copy_strings(bprm.argc, argv, &bprm); /*   将文件参数从用户空间复制到核心
                                                        * 空间*/
        if (retval < 0)                                 /* 若出错*/
                goto out;                               /* 跳转至 out */

        retval = search_binary_handler(&bprm,regs);     /*   搜寻该类可执行文件的处理函数并
                                                        *   执行,将返回值赋给 retval */
        if (retval >= 0)
                /* execve success */                    /* 若执行成功*/
                return retval;                          /* 返回 retval */

out:
        /* Something went wrong, return the inode and free the argument pages*/
        /* 如果出现某种错误,则返回 inode 并释放所占用的页面*/
        allow_write_access(bprm.file);                  /* 允许对文件执行写操作*/
        if (bprm.file)                                  /* 若 file 指针不空*/
                fput(bprm.file);                        /* 释放 file 结构占用的页面*/

        for (i = 0 ; i < MAX_ARG_PAGES ; i++) {         /* 设置循环释放 bprm.page 对应
                struct page * page = bprm.page[i];      * 的 inode 所占用的空间*/
                if (page)
                        __free_page(page);
        }

        return retval;                                  /* 返回 retval*/
}

/*********************************************************************/
/*位置 fs/exec.c*/
    int search_binary_handler(struct linux_binprm *bprm,struct pt_regs *regs)
{
        int try,retval=0;
        struct linux_binfmt *fmt;
#ifdef __alpha__                                        /* #ifdef 与#endif 之间的代码是专门针对
                                                        * alpha 处理器的条件编译,与 i386 无关
                                                        * 故略去*/
#endif
        for (try=0; try<2; try++) {
/* 作两次尝试,第一次是搜索 formats 队列,寻找匹配的处理模块;若队列中没有合适的匹配且内核支持动态
安装模块,则根据目标文件的第 2 和第 3 个字节生成一个 binfmt 模块名,通过 request_module()将相应模
块装入,然后对 formats 队列进行第二次搜索*/
                read_lock(&binfmt_lock);                /* 对 linux_binfmt 结构加锁*/
                for (fmt = formats ; fmt ; fmt = fmt->next) {/* 遍历 formats 队列*/
```

```
                int (*fn)(struct linux_binprm *, struct pt_regs *) = fmt->load_binary;
                                        /* 检查当前 linux_binfmt 结构引用的
                                        * 程序处理函数*/
                    if (!fn)                        /* 若程序处理函数不存在*/
                            continue;               /* 跳出,执行下一轮循环*/
                    if (!try_inc_mod_count(fmt->module))  /*增加该函数模块引用计数*/
                            continue;               /* 若失败,跳出,执行下一轮循环*/
                    read_unlock(&binfmt_lock);    /* 对 linux_binfmt 结构解锁*/
                    retval = fn(bprm, regs);/* 若处理模块存在,则试用该函数执行目标文件*/
                    if (retval >= 0) {                  /* 若执行成功*/
                            put_binfmt(fmt);            /* 释放 fmt 执行的 format 结构*/
                            allow_write_access(bprm->file);/* 允许对目标文件作写操作*/
                            if (bprm->file)
                                    fput(bprm->file);   /* 释放 file 结构占用的空间*/
                            bprm->file = NULL;          /* file 指针置空*/
                            current->did_exec = 1;      /* 标明当前进程正在执行 execve
                                                        * 装入的新代码*/
                            return retval;              /* 返回 retval*/
                    }
                    read_lock(&binfmt_lock);            /* 对 linux_binfmt 结构加锁*/
                    put_binfmt(fmt);                    /* 释放 fmt 执行的 format 结构*/
                    if (retval != -ENOEXEC)     /* 若出现程序处理模块不匹配以外的错误*/
                            break;                      /* 跳出内层循环*/
                    if (!bprm->file) {                  /* 若没有目标文件*/
                            read_unlock(&binfmt_lock);/* 对 linux_binfmt 结构解锁*/
                            return retval;              /* 返回 retval 值*/
                    }
            }
            read_unlock(&binfmt_lock);                  /* 对 linux_binfmt 结构解锁*/
            if (retval != -ENOEXEC) {           /* 若出现程序处理模块不匹配以外的错误*/
                    break;                              /* 跳出外层循环*/
#ifdef CONFIG_KMOD                                  /* 对支持动态安装模块的内核的条件编译*/
            }else{                                  /* 若 formats 队列中没有匹配的处理模块*/
#define printable(c) (((c)=='\t') || ((c)=='\n') || (0x20<=(c) && (c)<=0x7e))
                    char modname[20];
                    if (printable(bprm->buf[0]) &&
                        printable(bprm->buf[1]) &&
                        printable(bprm->buf[2]) &&
                        printable(bprm->buf[3]))
                            break; /* -ENOEXEC */
                    sprintf(modname,        "binfmt-%04x",        *(unsigned        short
*)(&bprm->buf[2]));
                    request_module(modname);            /* 动态安装相应的模块*/
#endif
```

```
                }
        }
        return retval;                                  /* 返回 retval*/
}
```

6.6 父进程的等待

```
/*****************************************************************************/
/*位置 kernel/exit.c*/
asmlinkage long sys_wait4(pid_t pid,unsigned int * stat_addr, int options, struct rusage * ru)
{
        int flag, retval;
        DECLARE_WAITQUEUE(wait, current);       /* 在当前进程的系统堆栈上分配一个
                                                 *  wait_queue_t 结构*/
        struct task_struct *tsk;

        if (options & ~(WNOHANG|WUNTRACED|__WNOTHREAD|__WCLONE|__WALL))
                return -EINVAL;                  /* 若 options 有错,返回错误码-EINVAL */

        add_wait_queue(&current->wait_chldexit,&wait);/* 把分配的 wait_queue_t 结构链入
                                                 * 当前进程的 wait_chldexit 队列中*/
repeat:
        flag = 0;                                /* 标志置零*/
        current->state = TASK_INTERRUPTIBLE;     /* 设当前进程为可中断等待态*/
        read_lock(&tasklist_lock);               /* 对进程队列加锁*/
        tsk = current;
        do {
                struct task_struct *p;
                for (p = tsk->p_cptr ; p ; p = p->p_osptr) {/* 从当前进程最新的子进程开始遍
                                                 * 历所有进程,寻找 pid 号子进程*/
                        if (pid>0) {             /* 若要找的进程不是 0 号(idle)进程*/
                                if (p->pid != pid)       /* 若 p 进程不是要找的进程*/
                                        continue;        /* 转入下一轮循环*/
                        } else if (!pid) {       /* 若要找的进程是 0 号(idle)进程*/
                                if (p->pgrp != current->pgrp)/*若 p 进程与当前进程不同组*/
                                        continue;        /* 转入下一轮循环*/
                        } else if (pid != -1) {          /* 若 pid 不超过进程 id 的临界值*/
                                if (p->pgrp != -pid)     /* 若 p 进程的组号不等于-pid*/
                                        continue;        /* 转入下一轮循环*/
                        }
                        /* Wait for all children (clone and not) if __WALL is set;
                         * otherwise, wait for clone children *only* if is __WCLONE
                         * set; otherwise, wait for non-clone children *only*.  (Note:
                         * A "clone" child here is one that reports to its parent
                         * using a signal other than SIGCHLD.) */
```

```
                    /* 若__WALL 置位,等待所有子进程(无论是否为克隆的) ;若__WCLONE
                     * 置位,仅等待克隆得来的子进程;否则,仅等待非克隆的子进程.(注意:这里所
                     * 说的"克隆"子进程是使用非 SIGCHLD 信号向其父进程报告的克隆子进程
                     */
                    if (((p->exit_signal != SIGCHLD) ^ ((options & __WCLONE) != 0))
                        && !(options & __WALL))  /* 若 p 进程是克隆子进程*/
                            continue;                /* 转入下一轮循环*/
                    flag = 1;                        /* 标志置一*/
                    switch (p->state) {              /* 开关语句,根据 p 进程的不同状态执行
                                                     *  不同操作 */
                    case TASK_STOPPED:               /* p 进程处于暂停态*/
                            if (!p->exit_code)       /* 若退出码为 0,即该进程不终止*/
                                    continue;        /* 转入下一轮循环*/
                            if (!(options & WUNTRACED) && !(p->ptrace & PT_PTRACED))
/* 若 WUNTRACED 和 PT_PTRACED 置位*/
                                    continue;            /* 转入下一轮循环*/
                            read_unlock(&tasklist_lock);/* 对进程队列加锁*/
                            retval = ru ? getrusage(p, RUSAGE_BOTH, ru) : 0;
                                                         /*确定复制子孙进程资源信息的地址
                                                         * ru 是否有效*/
                            if (!retval && stat_addr)    /* 若 ru 有效且 stat_addr 非空*/
                                    retval = put_user((p->exit_code << 8) | 0x7f,
stat_addr);                                              /* 将进程状态复制到 stat_addr*/
                            if (!retval) {               /* 若状态信息复制成功*/
                                    p->exit_code = 0;    /* 退出码置 0*/
                                    retval = p->pid;     /* 将 p(目标)进程的 id 赋给 retval*/
                            }
                            goto end_wait4;              /* 跳转至 end_wait4*/
                    case TASK_ZOMBIE:                    /* 僵尸态*/
                            current->times.tms_cutime += p->times.tms_utime +
p->times.tms_cutime;                                     /* 将 p 进程在用户态运行的时间加
                                                          * 在当前进程上*/
                            current->times.tms_cstime += p->times.tms_stime +
p->times.tms_cstime;                                     /* 将 p 进程在核心态运行的时间加
                                                         * 在当前进程上*/
                            read_unlock(&tasklist_lock);/* 对进程队列解锁*/
                            retval = ru ? getrusage(p, RUSAGE_BOTH, ru) : 0;
                                                         /*确定复制子孙进程资源信息的地址
                                                         * ru 是否有效*/
                            if (!retval && stat_addr)    /* 若 ru 有效且 stat_addr 非空*/
                                    retval = put_user(p->exit_code, stat_addr);
                                                         /* 将进程状态复制到 stat_addr*/
                            if (retval)                  /* 若复制失败*/
                                    goto end_wait4;      /* 跳转至 end_wait4*/
```

```
                        retval = p->pid;            /* 将 p(目标)进程的 id 赋给 retval*/
                        if(p->p_opptr != p->p_pptr){/*若 p 进程的祖先与父进程不同*/
                                write_lock_irq(&tasklist_lock);/* 对进程队列加锁*/
                                REMOVE_LINKS(p);  /* 将 p 进程从进程队列中摘除*/
                                p->p_pptr = p->p_opptr;/*  令 p 的父进程指向祖先
                                                        * 进程*/
                                SET_LINKS(p); /* 在祖先进程的控制下重新链入队列*/
                                do_notify_parent(p, SIGCHLD);/* 通知祖先进程,
                                                              * p 进程已死亡*/
                                write_unlock_irq(&tasklist_lock);/*对进程队列解锁*/
                        } else                      /*若 p 进程的祖先与父进程相同*/
                                release_task(p);     /* 释放死亡进程的剩余资源*/
                        goto end_wait4;             /* 跳转至 end_wait4*/
                default:                            /* 若进程不是停止或僵尸态*/
                        continue;                   /* 转入下一轮循环*/
                }
        }
        if (options & __WNOTHREAD)                  /* 若__WNOTHREAD 置位*/
                break;                              /* 跳出 do{}while 循环*/
        tsk = next_thread(tsk);                     /* 令 tsk 指向线程组下一个线程*/
    } while (tsk != current);                       /* do{}while 循环,遍历线程组*/
    read_unlock(&tasklist_lock);
    if (flag) {                                 /* 若 flag 置位,即找不到与 pid 匹配的子进程*/
        retval = 0;                             /* retval 清零*/
        if (options & WNOHANG)                  /* 若 WNOHANG 置位 */
                goto end_wait4;                 /* 跳转至 end_wait4*/
        retval = -ERESTARTSYS;                  /*将错误码-ERESTARTSYS 赋给 retval*/
        if (signal_pending(current))            /* 若当前进程挂起*/
                goto end_wait4;                 /* 跳转至 end_wait4*/
        schedule();                             /* 转入调度*/
        goto repeat;                            /* 继续循环,查找要退出的进程*/
    }
    retval = -ECHILD;                           /* 就错误码-ECHILD 赋给 retval*/
end_wait4:
    current->state = TASK_RUNNING;              /* 设当前进程为就绪态*/
    remove_wait_queue(&current->wait_chldexit,&wait);/* 将当前进程链入运行队列*/
    return retval;                              /* 返回 retval*/
}

/*****************************************************************************/
/*位置 kernel/exit.c*/
static void release_task(struct task_struct * p)
{
    if (p != current) {                         /* 确定要消灭的子进程不是当前进程*/
```

```
#ifdef CONFIG_SMP                                   /* SMP 处理,不做解释*/
                /*
                 * Wait to make sure the process isn't on the
                 * runqueue (active on some other CPU still)
                 */
                for (;;) {
                        task_lock(p);
                        if (!task_has_cpu(p))
                                break;
                        task_unlock(p);
                        do {
                                cpu_relax();
                                barrier();
                        } while (task_has_cpu(p));
                }
                task_unlock(p);
#endif
                atomic_dec(&p->user->processes);        /* p 进程所属用户的进程计数减一*/
                free_uid(p->user);                      /* 释放 user_struct 结构*/
                unhash_process(p);                      /* 将 p 进程从进程 hash 表摘除*/

                release_thread(p);                      /* 释放 p 进程的系统空间堆栈*/
                                                        /* 将 p 进程的次要页表错误,主要页
                                                         * 表错误的总数以及向外交换所使用
                                                         * 的时间数增加到当前进程对应的
                                                         * "子进程计数"中*/
                current->cmin_flt += p->min_flt + p->cmin_flt;
                current->cmaj_flt += p->maj_flt + p->cmaj_flt;
                current->cnswap += p->nswap + p->cnswap;

                current->counter += p->counter;             /* 将 p 进程的剩余时间片
                                                             * 传给当前(父)进程*/
                if (current->counter >= MAX_COUNTER)        /* 若父进程时间片超限*/
                        current->counter = MAX_COUNTER;     /* 给父进程时间片赋最大值*/
                p->pid = 0;                                 /* 清 p 进程 pid */
                free_task_struct(p);                    /* 释放 p 进程的 task_struct 结构*/
        } else {
                printk("task releasing itself\n");          /* 出错打印*/
        }
}
```

6.7 子进程的消亡

```
/*****************************************************************************/
/*位置 kernel/exit.c*/
```

```
asmlinkage long sys_exit(int error_code)
{
        do_exit((error_code&0xff)<<8);      /* 调用 do_exit 作主要工作,取 error_code 的低
                                             * 八位作为退出码*/
}

/*****************************************************************************/
/*位置 kernel/exit.c*/

NORET_TYPE void do_exit(long code)
{
        struct task_struct *tsk = current;

        if (in_interrupt())                          /* 若在中断处理例程中调用了 do_exit*/
                panic("Aiee, killing interrupt handler!");/* 将系统挂起 */
        if (!tsk->pid)                               /* 若是 0 号进程调用 do_exit*/
                panic("Attempted to kill the idle task!"); /* 将系统挂起 */
        if (tsk->pid == 1)                           /* 若是 init 号进程调用 do_exit*/
                panic("Attempted to kill init!");    /* 将系统挂起 */
        tsk->flags |= PF_EXITING;                /* 设置进程关闭状态,当前进程的标记记为退出*/
        del_timer_sync(&tsk->real_timer);            /* 删除当前的实时定时器*/

fake_volatile:
#ifdef CONFIG_BSD_PROCESS_ACCT
        acct_process(code);
#endif
        __exit_mm(tsk);                          /* 释放进程的存储管理信息.将 cache,tlb,pag 里
                                                  * 的内容全部回写,退出内存映射,释放页表*/
        lock_kernel();                           /* 加内核锁*/
        sem_exit();                              /* 删除信号队列,释放信号结构*/
        __exit_files(tsk);                       /* 释放进程的已打开文件的信息*/
        __exit_fs(tsk);                          /* 释放进程的文件系统*/
        exit_sighand(tsk);                       /* 释放进程的 signal 管理信息*/
        exit_thread();                           /* 释放进程的 LDT*/

        if (current->leader)                     /* 若当前进程是一个 session(人机交互分组)的主
                disassociate_ctty(1);            * 进程,则切断与其主控终端的联系,释放该 tty*/

        put_exec_domain(tsk->exec_domain);/* exec_domain 结构共享计数减一*/
        if (tsk->binfmt && tsk->binfmt->module) /* binfmt 结构共享计数减一*/
                __MOD_DEC_USE_COUNT(tsk->binfmt->module);

        tsk->exit_code = code;                   /* 填写退出码*/
        exit_notify();                           /* 通知当前进程的父进程和子进程,它要消亡了*/
```

```
        schedule();                             /* 转入调度*/

        goto fake_volatile;                     /* 跳转至 fake_volatile*/
}

/*****************************************************************************/
/*位置 kernel/exit.c*/
static inline void __exit_mm(struct task_struct * tsk)
{
        struct mm_struct * mm = tsk->mm;

        mm_release();                                   /* 执行 up()操作,唤醒睡眠中的父进程*/
        if (mm) {
                atomic_inc(&mm->mm_count);              /* mm_struct 结构引用计数增一*/
                if (mm != tsk->active_mm) BUG();        /* mm 与活动地址空间不等,出错处理*/

                task_lock(tsk);                         /* 对当前进程的 task_struct 结构加锁*/
                tsk->mm = NULL;                         /* 清 mm 指针,使进程不再拥有用户空间*/
                task_unlock(tsk);                       /* 对当前进程的 task_struct 结构解锁*/
                enter_lazy_tlb(mm, current, smp_processor_id());
                mmput(mm);                              /* 释放存储空间*/
        }
}

/*****************************************************************************/
/*位置 kernel/exit.c*/
static inline void __exit_files(struct task_struct *tsk)
{
        struct files_struct * files = tsk->files;

        if (files) {
                task_lock(tsk);                         /* 对进程的 task_struct 结构加锁*/
                tsk->files = NULL;                      /*  清 files 指针*/
                task_unlock(tsk);                       /* 对进程的 task_struct 结构解锁*/
                put_files_struct(files);                /* 释放 files_struct 结构*/
        }
}

/*****************************************************************************/
/*位置 kernel/exit.c*/
static inline void __exit_fs(struct task_struct *tsk)
{
        struct fs_struct * fs = tsk->fs;
```

```
        if (fs) {
                task_lock(tsk);                         /* 对进程的 task_struct 结构加锁*/
                tsk->fs = NULL;                         /* 清 fs 指针*/
                task_unlock(tsk);                       /* 对进程的 task_struct 结构解锁*/
                __put_fs_struct(fs);                    /* 释放 fs_struct 结构*/
        }
}

/*********************************************************************/
/*位置 kernel/exit.c*/
static void exit_notify(void)
{
        struct task_struct * p, *t;

        forget_original_parent(current);            /* 当前进程将自己的子进程托付给其它进程
                                                     * 若当前进程是线程,就托付给同一线程组的
                                                     * 下一个线程;否则,托付给 init 进程*/
        /*
         * Check to see if any process groups have become orphaned
         * as a result of our exiting, and if they have any stopped
         * jobs, send them a SIGHUP and then a SIGCONT.  (POSIX 3.2.2.2)
         *
         * Case i: Our father is in a different pgrp than we are
         * and we were the only connection outside, so our pgrp
         * is about to become orphaned.
         */
        /* 检查是否有进程组由于当前进程的退出而变成孤儿,如果它们有任何已中止的作业,给它们发
         *  SIGHUP 和 SIGCONT 信号. (POSIX 3.2.2.2)
         *
         * 问题 i:若当前进程的父进程与子进程们不再同一个进程组,且当前进程是与外界联系的唯一
          * 途径,那么当前进程退出后子进程们所在的进程组就成了孤儿.
         */
        t = current->p_pptr;                        /* 令 t 指向当前进程的父进程*/

        if ((t->pgrp != current->pgrp) &&           /* 当前进程与父进程不同组*/
            (t->session == current->session) &&     /* 当前进程与父进程属于同一个 session*/
            will_become_orphaned_pgrp(current->pgrp, current) &&/*当前进程所在进程组将
                                                     * 会成为孤儿*/
            has_stopped_jobs(current->pgrp)) {      /* 当前进程的子进程们有已中止的作业*/
                kill_pg(current->pgrp,SIGHUP,1);    /* 给当前进程的父进程发一个 SIGHUP 信号
                kill_pg(current->pgrp,SIGCONT,1); /*给当前进程的父进程发一个 SIGCONT 信号
        }

        /* Let father know we died
```

```
 * 让父进程知道当前进程死了
 * Thread signals are configurable, but you aren't going to use
 * that to send signals to arbitary processes.
 * That stops right now.
 *
 * If the parent exec id doesn't match the exec id we saved
 * when we started then we know the parent has changed security
 * domain.
 * 如果父进程的 exec id 与我们当前进程启动时保存的不同,那么当前进程将得知父进程改变了
  * 安全域
 * If our self_exec id doesn't match our parent_exec_id then
 * we have changed execution domain as these two values started
 * the same after a fork.
 * 如果当前进程的 self_exec id 与 parent_exec_id 不匹配,那么在一次 fork 调用后这两个
  * 值同时产生时,当前进程改变了执行域.
 */

if(current->exit_signal != SIGCHLD &&              /*当前进程 SIGCHLD 标志未置位*/
    ( current->parent_exec_id != t->self_exec_id  ||     /*父进程的安全域发生改变*/
      current->self_exec_id != current->parent_exec_id) /*当前进程的执行域改变*/
    && !capable(CAP_KILL))                         /* 当前进程尚不能被消灭*/
        current->exit_signal = SIGCHLD;            /* 当前进程 SIGCHLD 标志置位*/


/*
 * This loop does two things:
 * 该循环做两件事:
 * A.  Make init inherit all the child processes 使 init 进程继承当前进程所有的子进程
 * B.  Check to see if any process groups have become orphaned
 *       as a result of our exiting, and if they have any stopped
 *       jobs, send them a SIGHUP and then a SIGCONT.  (POSIX 3.2.2.2)
 *  检查是否有进程组由于当前进程的退出而变成孤儿,如果它们有任何已中止的作业,给它们发
 *  SIGHUP 和 SIGCONT 信号. (POSIX 3.2.2.2)
*/

write_lock_irq(&tasklist_lock);                    /* 对进程队列加锁,禁止写操作*/
current->state = TASK_ZOMBIE;                      /* 设当前进程为僵尸态*/
do_notify_parent(current, current->exit_signal);/* 通知父进程当前进程已死亡*/
while (current->p_cptr != NULL) {                  /* 遍历当前进程的子进程,将其托付给当*
                                          前进程的祖先进程,并检查变成孤儿的进程组*/
        p = current->p_cptr;                       /* 令 p 指向当前进程最新的子进程*/
        current->p_cptr = p->p_osptr;              /* 当前进程的 p_cptr 指向 p 的兄长*/
        p->p_ysptr = NULL;                         /* p 的新兄弟进程为空*/
        p->ptrace = 0;
```

```
                p->p_pptr = p->p_opptr;              /* 令 p 的父进程指针指向 p 的祖先进程*/
                p->p_osptr = p->p_pptr->p_cptr;    /* 令 p 的祖先进程指向 p 进程*/
                if (p->p_osptr)                       /* 若 p 有老兄弟进程*/
                        p->p_osptr->p_ysptr = p;    /* 令老兄弟进程的新兄弟进程指针指向 p*/
                p->p_pptr->p_cptr = p;               /* 令 p 的父进程的最新子进程指针指向 p*/
                if (p->state == TASK_ZOMBIE)        /* 若 p 进程的状态为僵尸态*/
                        do_notify_parent(p, p->exit_signal);/* 通知 p 的父进程 p 以死亡*/
                /*
                 * process group orphan check 进程组孤儿检查
                 * Case ii: Our child is in a different pgrp
                 * than we are, and it was the only connection
                 * outside, so the child pgrp is now orphaned.
                 * 问题 ii: 当前进程的子进程与当前进程不同组,且该子进程是与外界联系的唯一纽带,
                 * 于是子进程组现在变成了孤儿.
                */
                if ((p->pgrp != current->pgrp) &&          /* p 进程与当前进程不同组*/
                   (p->session == current->session)){ /* p 进程与当前进程属于同一 session*/
                        int pgrp = p->pgrp;                /* 令 pgrp 指向 p 进程的进程组*/

                        write_unlock_irq(&tasklist_lock); /* 对进程队列解锁,写允许*/
                        if (is_orphaned_pgrp(pgrp) && has_stopped_jobs(pgrp)) {
                                                /* pgrp 所指进程组是孤儿且有已中止的作业*/
                            kill_pg(pgrp,SIGHUP,1); /*给 p 的父进程发一个 SIGHUP 信号*/
                            kill_pg(pgrp,SIGCONT,1);/* 给 p 的父进程发一个 SIGCONT 信号*/
                        }
                        write_lock_irq(&tasklist_lock);     /*对进程队列加锁*/
                }
        }
        write_unlock_irq(&tasklist_lock);                    /*对进程队列解锁*/
}

/*********************************************************************/
/*位置 kernel/signal.c*/
void exit_sighand(struct task_struct *tsk)
{
        struct signal_struct * sig = tsk->sig;

        spin_lock_irq(&tsk->sigmask_lock);          /* 加锁,禁止信号中断*/
        if (sig) {                                  /* 若指向 signal_struct 结构的指针不空*/
                tsk->sig = NULL;                    /* 清当前进程的 sig 指针*/
                if (atomic_dec_and_test(&sig->count))/* signal_struct 结构引用计数减一,若此
                                                      * 时计数值为 0*/
                        kmem_cache_free(sigact_cachep, sig);  /* 释放 sig 结构*/
```

```
        }
        tsk->sigpending = 0;
        flush_sigqueue(&tsk->pending);                  /* 释放当前进程使用的信号队列*/
        spin_unlock_irq(&tsk->sigmask_lock);            /* 解锁,允许信号中断*/
}
```

结束语

作者用了一个学期的时间详细的阅读并注释 Linux 内核中进程调度和进程控制部分的源代码，并对其作了较深入的分析。

通过对 Linux 代码的深入学习，作者加深了对操作系统原理的理性认识，并在较深层次上理解了操作系统的代码实现过程。通过对内核代码的分析，作者对 Linux 进程调度和进程控制部分的实现过程有了一个清晰的思路。

所有这些都必将对作者日后的工作起到积极的作用。同时，作者在阅读内核代码的过程中深为这些代码精妙的实现方法而折服，即使作者以后不从事系统软件方面的工作，这些代码实现的技巧也会对作者作其他软件开发工作带来极大的助益。同时，对阅读内核代码进行阅读、总结的过程也提高了作者分析问题、解决问题的能力，而这种能力是无论做什么工作都需要的。

由于作者的知识有限，所以对 Linux 内核的分析总是有这样或那样的不足。同时，受时间和精力的限制，并对代码的分析也没有达到毫无疏漏的地步。通过对 Linux 内核的分析，作者深感自身操作系统理论的匮乏，今后还应不断加深自身的理论水平，同时注重理论与实践的结合。

致谢

感谢指导老师范廉明先生对本文作者无微不至的帮助。作者在分析 Linux 源码的过程中得到了范先生耐心、细致的指导，事实证明这些指导工作对作者深入理解 Linux 内核起了关键性的作用。范先生还认真审阅了作者的论文，并提出了十分中肯的修改意见，使作者能够较顺利地完成毕业论文。作者对范廉明先生严谨的治学态度深感钦佩。

感谢赵毅、张永峰、陈秋朗和杨少鹏四位同学。这四位同学在百忙之中为作者提供了相关资料和许多有建设性的意见。

感谢章俊玲同学对作者在精神上的支持和帮助。

感谢作者的家人。他们对作者生活上无微不至的照顾和精神上坚定的支持使作者能够充满信心地完成毕业论文。

参考文献

[1] 陈莉君编著. Linux 操作系统内核分析.北京: 人民邮电出版社,2000

[2] 冯锐等译. Linux 内核源代码分析.北京: 机械工业出版社,2000

[3] 李善平等编著.Linux 操作系统及实验教程.北京: 机械工业出版社,1999

[4] 李善平等编著.Linux 内核 2.4 版源代码分析大全.北京: 机械工业出版社,2002

[5] 毛德操等著.Linux 内核源代码情景分析.杭州: 浙江大学出版社,2001

[6] 杨晓云等译.Linux 程序设计.北京: 机械工业出版社,2002

[7] 金惠华等译.80×86、奔腾机汇编语言程序设计.北京: 电子工业出版社,1998

[8] 孙钟秀等编著.操作系统教程.北京: 高等教育出版社,2000

[9] 汤子瀛等编著.计算机操作系统.西安: 西安电子科技大学出版社,2000

[10] Andrew S. Tanenbaum. Modern Operating System.北京: 机械工业出版社,2002