LINUX 中的进程

本章以 i386 平台为例，从实现的角度介绍了进程控制块的结构，以及在进程生命周期中创建、调度切换与退出的过程。

内容组织如下：

一、 进程控制块 PCB 简介

进程即程序的一次执行。从组成上看，进程可划分为三个部分：PCB、CODE 与 DATA。从动态执行的角度来看，进程可视为程序的指令在 OS 根据 PCB 进行调度而分配的若干时间片内对程序中数据的操作过程。

PCB 是 OS 进程管理的依据和对象。为了调度，其中必须存有标识、状态、调度方法以及进程的上下文等信息；而每个进程运行在各自不同的虚拟地址空间，其中就必须有虚实地址映射机制；为了控制，其中便存有进程链信息以及时钟定时器等；为了通讯，其中又必然有信号、信号量等机制。OS 便是根据这些信息来控制的管理每个进程的创建、调度与切换以及消亡。

在 LINUX 中，结构 task_struct 即是 PCB。可能是由于 i386CPU 的原因，LINUX 并不区分进程与任务。

task_struct 结构的组成主要可分为如下几个部分：

- 进程运行状态信息。一个 Linux 进程的运行、等待、停止以及僵死状态。
- 用户标识信息。执行该进程的用户的信息。
- 标识号。pid 用以唯一标识一个进程，进程还有一个 id 用以标识它在进程数组中的索引。每个进程还有组号、会话号，这些标识用以判断一个进程是否有足够的优先权来访问外设等。
- 调度信息。调度策略、优先级等。
- 信号处理信息。信号挂起标志，信号阻塞掩码，信号处理例程等。
- 进程内部状态标志。调试跟踪标识，创建方法标识，用户 id 改变标识，最后一次系统调用时的错误码等等。这些标志一般与具体 CPU 有关。
- 进程链信息。Linux 中有一个 task 数组，其长度为允许进程的最大个数，一般为 512。

  ```
  struct task_struct * task[NR_TASKS] = {&init_task, };
  ```

  对于每一个进程，有两个指针用以形成一个全体进程的循环双向链表（进程 0 为根），还有两个指针用以形成一个可运行进程的循环双向链表。还有一些指针用以指向父进程、子进程、兄弟进程等。
- 等待队列。用于 wait4()函数等待子进程的返回。
- 时间与定时器。保存进程的建立时间，以及在其生命周期中所花费的 CPU 时间。应用程序还可以建立定时器，在定时器到期时，根据不同定时器类型发送相应的信号。
- 打开的文件以及文件系统信息。系统需要跟踪进程所打开的文件，以便在适当的时候关闭文件以及判断对文件操作的正确性。子进程从父进程处继承了父进程打开的所有文件的标识符。另外，进程还有指向 VFS 索引节点的指针，分别是进程的主目录以及当前目录。
- 内存管理信息。进程分别运行在各自的地址空间中，需要将虚实地址一一映射对应起来。
- 进程间通讯信息。信号量等。
- 上下文信息 tss。在进程的运行过程中，必然伴随着系统状态的改变，这些状态将影响进程的执行。当调度程序选择了一个新进程以运行时，旧进程从运行状态切换为暂停状态，它的运行环境如寄存器堆栈等，必须保存在上下文中，以便下次恢复运行。显然，上下文信息与 CPU 类型紧密相连。

以下对这些信息作详细介绍。

1．进程运行状态

volatile long state。Linux 进程共有如下 6 种状态：

- ✧ TASK_RUNNING，当前运行进程以及运行队列中的进程都处于该状态中。进程调度时，调度程序只在处于该状态的进程中选择最优进程来运行。
- ✧ TASK_INTERRUPTIBLE，在等待队列中，可被信号中断的等待状态。收到信号后，进程可能停止或者重新插入到运行队列中。
- ✧ TASK_UNINTERRUPTIBLE，直接等待硬件状态，不可被中断。
- ✧ TASK_ZOMBIE，僵死状态。进程已经消亡，但其 PCB 仍存在 task 数组中。在进程退出时，将状态设为 TASK_ZOMBIE，然后发送信号给父进程，由父进程在统计其中的一些数据后，释放它的 task_struct 结构。
- ✧ TASK_STOPPED，进程因接收到信号（如：SIGSTOP、SIGSTP、SIGTTIN、SIGTTOU）而停止或由于其他进程使用 ptrace 系统调用来跟踪而将控制权交回控制进程。
- ✧ TASK_SWAPPING，进程被交换到了交换区（2.0 版中未实现）。

2．用户标识信息

Linux 使用用户标识符与组标识符来判断用户进程对文件和目录的访问许可。每个进程的 task_struct 中均有 4 对标识符。

- uid、gid。执行该进程的用户的标识与组标识。在 fork 时，从父进程处继承。
- euid、egid。某些程序可以将 uid 和 gid 改变为自己私有的 uid 和 gid。系统在运行这样的程序时，会根据修改后的 uid 及 gid 判断程序的特权，例如，是否能够直接进行 I/O 输出等。通过 **setuid** 系统调用，可将程序的有效 uid 和 gid 设置为其他用户。在该程序映象文件的 VFS 索引节点中，有效 uid 和 gid 由索引节点的属性描述。
- suid、sgid。如果进程通过系统调用修改了进程的 uid 和 gid，这两个标识符则保存实际的 uid 和 gid。
- fsuid、fsgid。用于检查对文件系统的访问许可。处于用户模式的 NFS 服务器作为特殊进程访问文件时使用这两个标识符，用于限制用户的其他访问权限。一般而言，fsuid==euid && fsgid==egid。系统调用 setfsuid、setfsgid 可直接修改这两个 id，而不改动 euid 与 egid。

3．标识号

- pid。用以唯一标识一个进程。
- pgrp。进程所处的进程组的标识。pid 与 pgrp 用以判断进程是否有外设的访问权。
- session。进程所处的会话的标识。
- leader。会话的领头进程标志。
- groups[NGROUPS]。正如现代的许多 UNIX，一个进程可能同时属于许多个进程组，Linux 使用该数组来存储进程所在的各个进程组 id，这些进程组的最多个数为 NGROUPS，可以通过#define NGROUPS –1 来取消这项功能。

另外，进程还有一个隐含的 id，就是它在 task 数组中的索引。该索引可用以组成选择子来指向进程的 TSS 段描述符以及 LDT 段描述符在 GDT 中的位置。

4．调度信息

- unsigned long policy。Linux 有三种调度策略：

- ✧ SCHED_OTHER。适用于一般进程，一般进程的优先级较实时进程低。
- ✧ SCHED_FIFO。实时进程，先进先出（first in first out），如 kswapd 进程。
- ✧ SCHED_RR。实时进程，循环赛（round robin）。

- long counter 与 long priority。priority 是系统为进程给定的优先级，即从进程开始运行算起的，允许进程运行的时间。counter 在开始时设为 priority，每次时钟中断发生，该值减 1 直至为 0。counter 为 0 表示分配给该进程的执行时间已到，不可再被调度。若所有可运行进程的 counter 都为 0，则需要重新计算 counter。
- unsigned long rt_priority。实时进程的相对优先级。在计算权重时，实时进程的权重为 rt_priority+1000，因此总能获得比一般进程更高的优先级。
- unsigned long timeout。用于定时，当值为 0 时，将进程加入到运行队列中。

5．信号处理信息

在 Linux 中，信号种类的数目和具体的平台有关，因为内核用一个字代表所有的信号，因此字的位数就是信号种类的最多数目。对 32 位的 i386 平台而言，一个字为 32 位，因此信号有 32 种。

进程可以选择对某种信号所采取的特定操作，这些操作包括：

- ➢ 阻塞信号。进程可选择阻塞某些信号，SIGKILL 和 SIGSTOP 信号不能被阻塞。
- ➢ 由进程处理该信号。进程本身可在系统中注册处理信号的处理程序地址，当发出该信号时，由注册的处理程序处理信号。
- ➢ 由内核进行默认处理。信号由内核的默认处理程序处理。大多数情况下，信号由内核处理。

- unsigned long signal 与 unsigned long blocked。Signal 用以记录当前挂起的信号，blocked 用以记录当前阻塞的信号。挂起的信号指尚未进行处理的信号。阻塞的信号指进程当前不处理的信号，如果产生了某个当前被阻塞的信号，则该信号会一直保持挂起，直到该信号不再被阻塞为止。
- struct signal_struct *sig。

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

count 为该 signal_struct 的引用计数，当使用 CLONE_SIGHAND 标志来创建一个子进程时，父进程与子进程共享一个 signal_struct，count 计数加 1。在进程退出时，判断该 files_struct 是否还有其他进程引用。只有 count 为 0，方可删除该结构。action[32]中指定了进程处理所有这 32 个信号的方式，如果某个 sigaction 结构中包含有处理信号的例程地址，则由该处理例程处理该信号；反之，则根据结构中的一个标志或者由内核进行默认处理，或者只是忽略该信号。

进程不能向系统中所有的进程发送信号，一般而言，除系统和超级用户外，普通进程只能向具有相同 uid 和 gid 的进程，或者处于同一进程组的进程发送信号（一个例外：信号 SIGCONT 可以被发送到同一个会话的任意进程）。

产生信号时，内核将进程 task_struct 的 signal 字中的相应位设置为 1，从而表明产生了该信号。系统不对置位之前该位已经为 1 的情况进行处理，因而进程无法接收到前一次信号。如果进程当前没有阻塞该信号，并且进程正处于可中断的等待状态，则内核将该进程的状态改变为运行，并放置在运行队列中。这样，调度程序在进行调度时，就有可能选择该进程运行，从而可以让进程处理该信号。

发送给某个进程的信号并不会立即得到处理，相反，只有该进程再次运行时，才有机会

处理该信号。每次进程从系统调用中退出时，内核会检查它的 signal 和 block 字段，如果有任何一个未被阻塞的信号发出，内核就根据 sigaction 结构数组中的信息进行处理。处理过程如下：

1. 检查对应的 sigaction 结构，如果该信号不是 SIGKILL 或 SIGSTOP 信号，且被忽略，则不处理该信号。
2. 如果该信号利用默认的处理程序处理，则由内核处理该信号，否则转向第 3 步。
3. 该信号由进程自己的处理程序处理，内核将修改当前进程的调用堆栈帧，并将进程的程序计数寄存器修改为信号处理程序的入口地址。此后，指令将跳转到信号处理程序，当从信号处理程序中返回时，实际就返回了进程的用户模式部分。

Linux 是 POSIX 兼容的，因此，进程在处理某个信号时，还可以修改进程的 blocked 掩码。但是，当信号处理程序返回时，blocked 值必须恢复为原有的掩码值，这一任务由内核完成。Linux 在进程的调用堆栈帧中添加了对清理程序的调用，该清理程序可以恢复原有的 blocked 掩码值。当内核在处理信号时，可能同时有多个信号需要由用户处理程序处理，这时，Linux 内核可以将所有的信号处理程序地址推入堆栈帧，而当所有的信号处理完毕后，调用清理程序恢复原先的 blocked 值。

6．进程链信息与等待队列

- struct task_struct *next_task, *prev_task。所有进程的双向链表指针，其根为 task0 即 init_task。在拼装 init_task 的 task_struct 结构时，将这两个指针都指向它自身。
- struct task_struct *next_run, *prev_run。所有运行进程的双向链表指针，组成了运行队列，其根也为 task0。在拼装 init_task 的 task_struct 结构时，这两个指针也都指向它自身。若当前进程不处于运行队列中，这两个指针必为空，否则必为非空。这常用于进程入运行队与出运行队的判断。
- struct task_struct *p_opptr。初始的父进程指针，父进程可能由于退出或其他原因，而将其子进程转移到其他进程如 task1 名下。
- struct task_struct *p_pptr。父进程指针，init_task 的这个指针指向它自身。
- struct task_struct *p_cptr。子进程指针，init_task 的这个指针指向它自身。
- struct task_struct *p_ysptr。左兄弟进程指针。
- struct task_struct *p_osptr。右兄弟进程指针。
- struct wait_queue *wait_chldexit。等待队列，用于系统调用 wait4()，等待子进程的返回。在系统调用 wait()发生时，将自己插入到这个等待队列中，状态改为 TASK_INTERRUPTIBLE，等待子进程结束时的返回信号。参见“队列的操作”。

7．时间与定时器

- long utime, stime。进程在用户态、核心态下的运行时间。
- long cutime, cstime。分别为进程所有子孙进程 utime 与 stime 的总和。子进程退出时，将发送信号 SIGCHLD 给父进程，然后父进程更新这些数据。
- long start_time。进程创建的时间。

Linux 用指针 current 保存当前正在运行的进程之 task_struct。每当产生一次实时时钟中断（外部中断 #0，中断号为 0x20），Linux 就会更新 current 所指向的进程的时间信息，如果内核当前代表该进程执行任务（例如进程调用系统调用时），那么系统就将时间记录为进程在系统模式下花费的时间，否则记录为进程在用户模式下花费的时间。

除了为进程记录其消耗的 CPU 时间外，Linux 还支持和进程相关的间隔定时器。当定时器到期时，会向定时器的所属进程发送信号。进程可使用如下三种不同类型的定时器来给自己发送相应的信号。

Real	该定时器实时更新，到期时发送 SIGALRM 信号。
Virtual	该定时器只在进程运行时更新，到期时发送 SIGVTALRM 信号。
Profile	该定时器在进程运行时，以及内核代表进程运行时更新，到期时发送 SIGPROF 信号。

- unsigned long it_real_value, it_prof_value, it_virt_value。三种计时器各自的定时长度。
- unsigned long it_real_incr, it_prof_incr, it_virt_incr。三种计时器各自的到期时间。

Linux 对 Virtual 和 Profile 定时器的处理是相同的，在每个时钟中断，定时器的计数值减 1，直到计数值为 0 时发送信号。

- struct timer_list real_timer。

9．打开的文件以及文件系统信息

- struct fs_struct *fs。

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

count 为该 fs_struct 的引用计数，当使用 CLONE_FS 标志来创建一个子进程时，父进程与子进程共享一个 fs_struct，count 计数加 1。在进程退出时，判断该 fs_struct 是否还有其他进程引用。只有 count 为 0，方可删除该结构。

root 与 pwd 包含指向两个 VFS 索引节点的指针，这两个索引节点分别是进程的主目录以及进程的当前目录。索引节点中有一个引用计数器，当有新的进程指向某个索引节点时，该索引节点的引用计数器会增加计数。未被引用的索引节点的引用计数为 0，因此，当包含在某个目录中的文件正在运行时，就无法删除这一目录，因为这一目录的引用计数大于 0。

- struct files_struct *files。

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

同样，count 为该 files_struct 的引用计数，当使用 CLONE_FILES 标志来创建一个子进程时，父进程与子进程共享同一个 files_struct，count 计数加 1。在进程退出时，判断该 files_struct 是否还有其他进程引用。只有 count 为 0，方可删除该结构。

NR_OPEN 为 Linux 允许打开的文件的最大个数。fd[NR_OPEN]数组记录了进程打开的所有文件的描述符，某个元素非空表示该文件已经被该进程打开。这些描述符在子进程创建时，从父进程继承而来。在继承时，还要将这些文件的引用计数加 1，表明多了一个进程使用这些文件。

11．内存管理信息

- struct desc_struct *ldt。为 Wine 而保留。一般为 NULL，此时，将 default_ldt 作为进程的 ldt。Linux 中所有进程都使用相同的 ldt——default_ldt，这个 ldt 中只有一个表项，为指向 KERNEL_CS:lcall7 且 RPL 为 3 的调用门描述符。
- struct mm_struct *mm。记录进程分配的内存。mm_struct 主要包括进程代码段、数据段、BSS 段、调用参数区与环境区的起始结束地址，以及指向 struct vm_area_struct 结构的指针等。

13．上下文信息 tss

- struct thread_struct tss。保存进程运行的环境信息，如：通用寄存器，堆栈指针以及程序状态字等。

在“进程的切换”中详细介绍。

14．执行域

- struct exec_domain *exec_domain。
- unsigned long personality。

Linux 可执行遵循 iBCS2 基于 i386 结构的其他系统的程序，这些程序都有一些差别，在这个 exec_domain 结构中反映出要模拟运行的其他 UNIX 的信息。personality 用于记录这个程序对应的 UNIX 的版本，对于标准的 Linux 进程，personality 值为 PER_LINUX。

15．内核栈

- unsigned long kernel_stack_page。用户进程在系统调用时使用的内核栈。
- unsigned long saved_kernel_stack。在系统调用 vm86 中，用于保存旧的内核栈。

16．进程间通讯信息

- struct sem_undo *semundo。
- struct sem_queue *semsleeping。

17．其他信息

- struct rlimit rlim[RLIM_NLIMITS]。对进程使用资源的一些限制。
- unsigned short used_math。
- char comm[16]。本进程所执行的程序的名字，经常用于调试。
- int exit_code, exit_signal。exit_code 记录进程的返回码。在进程退出时，将 exit_signal 信号发送给父进程。
- int dumpable:1。用于标志接收到某个信号时，是否执行 memory dump。
- int did_exec:1。为了遵循 POSIX 标准，在调用 setpgid 时，用以判断进程是否通过系统调用 execve()而重新装载执行。
- struct linux_binfmt *binfmt。用于装载可执行程序的结构，其中的装载函数指针可以设置为不同的装载函数。

二、 进程的创建

1． 进程

新的进程通过复制旧进程（即父进程）而建立。为了创建新进程，首先在系统的物理内存中为新进程创建一个 task_struct 结构（使用 kmalloc 函数，以便得到一个连续的区域），将旧进程的 task_struct 结构内容复制到其中，再修改部分数据。接着，为新进程分配新的核心堆栈页，分配新的进程标识符 pid。然后，将这个新 task_struct 结构的地址填到 task 数组中，并调整进程链关系，插入运行队列中。于是，这个新进程便可以在下次调度时被选择执行。此时，由于父进程的进程上下文 TSS 结构复制到了子进程的 TSS 结构中，通过改变其中的部分数据，便可以使子进程的执行效果与父进程一致，都是从系统调用中退出，而且子进程将得到与父进程不同的返回值（返回父进程的是子进程的 pid，而返回子进程的是 0）。

在创建进程时，Linux 允许父子进程共享某些资源。可共享的资源包括文件、文件系统、

信号处理程序以及虚拟内存等。当某个资源被共享时，该资源的引用计数值加 1。在进程退出时，将所引用的资源的引用计数减 1。只有在引用计数为 0 时，才表明这个资源不再被使用，此时内核才会释放这些资源。

在进程创建时，Linux 内核并不为子进程分配所需的物理内存，而是让父进程与子进程以只读方式来共享原父进程所分配的内存。新的 vm_area_struct 结构、新进程自己的 mm_struct 结构以及新进程的页表在创建进程结构时便已准备好，但并不复制。如果旧进程的某些虚拟内存在物理内存中，而有些在交换文件中，那么虚拟内存的复制将会非常困难和费时。而且，为每个创建的进程都一一复制虚存，将导致极大的内存开销，而这些内存其实并不一定是必要的。于是，如同许多现代的 UNIX 系统，Linux 采用了 copy-on-write 技术，只有当两个进程中的任意一个向虚拟内存中写入数据时才复制相应的虚拟内存；而未执行过写操作的任何内存页均在两个进程之间以只读方式共享。代码页实际总是可以共享的。

若一个进程需要对某一页执行写操作，由于这一页已被写保护，此时将产生一个 page fault 异常。在这个异常的处理句柄中，为该页复制一个拷贝，并将其分配给执行写操作的进程，然后修改这两个进程的页表以及虚拟内存数据结构，以分别使用不同的页。对于进程而言，这一处理过程是透明的，它的操作可以成功的执行。如此处理，只在真正需要时为进程分配内存，好处在于减少了不必要的内存复制开销同时也减少了一些不必要的内存需求。这便是所谓的 copy-on-write “写时复制”技术。

2. 线程

和进程概念紧密相关的概念是线程。与进程相关的基本要素有：代码、数据、堆栈、文件、I/O 和虚拟内存信息等，而线程除了 CPU 资源，再没有其他资源可以分配。因此，线程也被称作“轻量级进程”。系统对进程的处理要花费更多的开支，尤其在进行进程调度时。利用线程则可以通过共享这些基本要素而减轻系统开支。

线程有“用户线程”和“内核线程”之分。用户线程指不需要内核支持而在用户程序中实现的线程，这种线程甚至在象 DOS 这样的操作系统中也可实现，但线程的调度需要用户程序完成，这有些类似 Windows 3.x 的协作式多任务。另外一种则需要内核的参与，由内核完成线程的调度。这两种模型各有其好处和缺点。用户线程不需要额外的内核开支，但是当一个线程因 I/O 而处于等待状态时，整个进程就会被调度程序切换为等待状态，其他线程得不到运行的机会；而内核线程则没有各个限制，但却占用了更多的系统开支。

Linux 并不确切区分进程与线程，而将线程定义为“执行上下文”，它实际只是同一个进程的另外一个执行上下文而已。对于调度，仍然可以使用进程的调度程序。Linux 的内核进程，使用 kernel_thread 创建，一般被称作线程。

有两个系统调用可用以建立新的进程：fork 与 clone。fork 一般用以创建普通进程，而 clone 可用以创建线程，kernel_thread 便是通过 sys_clone 来创建新的内核进程。fork 与 clone 都调用 do_fork 函数执行创建进程的操作。fork 并不指定克隆标志，而 clone 可由用户指定克隆标志。克隆标志有 CLONE_VM、CLONE_FS、CLONE_FILES、CLONE_SIGHAND 与 CLONE_PID 等，这些克隆标志分别对应相应的进程共享机制。而 fork 创建普通进程则使用 SIGCHLD 标志。这些克隆标志在处理后形成 exit_signal，将在进程退出时作为信号发送给父进程。

- CLONE_VM。父子进程共享同一个 mm_struct 结构，这个克隆标志用以创建一个线程。由于两个进程都使用同一个 mm_struct 结构，于是这两个进程的指令、数据都共享，也就是将线程视为同一个进程的不同执行上下文。
- CLONE_FS。父子进程共享同一个文件系统。
- CLONE_FILES。父子进程共享所打开的文件，如图 1 所示。
- CLONE_SIGHAND。父子进程共享信号处理句柄。

- CLONE_PID。父子进程共享 pid。

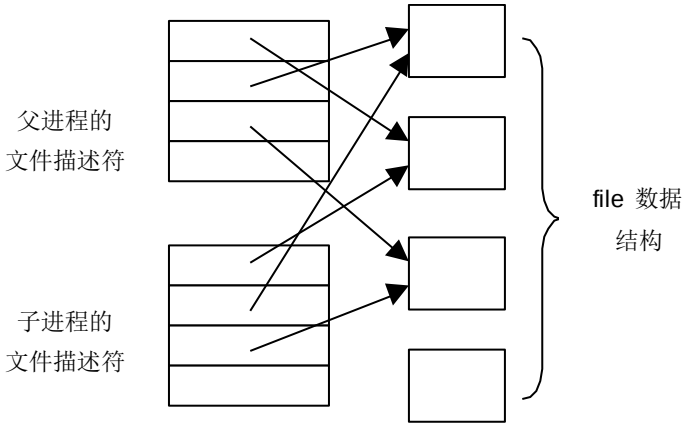

图 1 父进程与子进程共享打开的文件

系统调用 fork 使用 SIGCHLD 克隆标志来创建进程，而且新进程使用与父进程一致的用户堆栈。

```
asmlinkage int sys_fork(struct pt_regs regs)
{
    return do_fork(SIGCHLD, regs.esp, &regs);
}
```

系统调用 clone 可使用不同的克隆标志来创建进程，而且可以为子进程指定新的的用户堆栈。

```
asmlinkage int sys_clone(struct pt_regs regs)
{
    unsigned long clone_flags;
    unsigned long newsp;

    clone_flags = regs.ebx;
    newsp = regs.ecx;
    if (!newsp)
        newsp = regs.esp;
    return do_fork(clone_flags, newsp, &regs);
}
```

这些共享机制以及新进程用户堆栈的具体处理过程将在“进程的创建过程”中详细描述。

3. 进程的创建过程

只有 task0 的 task_struct 是在初始化过程中人工拼装的，为 INIT_TASK（include/kernel/sched.h），其余的进程的 task_struct 都在系统调用 fork 或 clone 中创建。

系统调用 fork 与 clone 用以创建进程，其底层处理过程都通过 do_fork 函数实现。在第 1 节 task_struct 结构的介绍中，可以看到，父子进程间可能共享的结构，都定义为指针类型，若不共享就需要为子进程重新生成一个这样的结构；若共享只需将原结构的引用计数加 1。而那些不需共享的结构则直接定义为结构类型，在复制 task_struct 时，便拷贝到了新 task_struct 中，就不需要重新创建。

do_fork 函数（kernel/fork.c）的具体处理过程如下：

```
int do_fork(unsigned long clone_flags, unsigned long usp, struct pt_regs *regs)
{
    int nr;
    int error = -ENOMEM;
    unsigned long new_stack;
    struct task_struct *p;
    p = (struct task_struct *) kmalloc(sizeof(*p), GFP_KERNEL);
    if (!p)
        goto bad_fork;
```

1. 使用 kmalloc（为了得到一块连续的区域）为新进程在内核空间创建一个 task_struct 结构，若出错则跳转到出错处理，此时的错误码为 -ENOMEM 表示内存不足。

```
    new_stack = alloc_kernel_stack();
    if (!new_stack)
        goto bad_fork_free_p;
    error = -EAGAIN;
```

2. 调用 alloc_kernel_stack()分配新进程的内核堆栈空间，若失败则跳转到出错处理。将设置错误码为-EAGAIN，表示可以再试一次。
   alloc_kernel_stack() 调用 __get_free_page(GFP_KERNEL)（include/asm-386/processor.h），而 __get_free_page(GFP_KERNEL)（include/linux/mm.h）调用了__get_free_pages((priority),0,0)分配了一个空白页。

```
    nr = find_empty_process();
    if (nr < 0)
        goto bad_fork_free_stack;
```

3. 为新进程的 task_struct 在 task 数组中找到一个空闲位置。若无空闲位置，则跳转到出错处理。find_empty_process（kernel/fork.c）的执行过程如下：

```
static inline int find_empty_process(void)
{
    int i;
    if (nr_tasks >= NR_TASKS - MIN_TASKS_LEFT_FOR_ROOT) {
        if (current->uid)
            return -EAGAIN;
    }
```

✧ 首先，判断系统的进程数是否已经饱和，需要为超级用户保留一些位置。

```
    if (current->uid) {
        long max_tasks = current->rlim[RLIMIT_NPROC].rlim_cur;
        max_tasks--;    /* count the new process.. */
        if (max_tasks < nr_tasks) {
            struct task_struct *p;
            for_each_task (p) {
                if (p->uid == current->uid)
```

```
                    if (--max_tasks < 0)
                        return -EAGAIN;
                }
            }
```

✧ 其次，判断该用户是否有创建进程的权限，即该用户执行的进程数是否超过最大限制。

```
        }
        for (i = 0 ; i < NR_TASKS ; i++) {
            if (!task[i])
                return i;
        }
```

✧ 然后，在 task 数组中查找一个空闲位置，若有，返回序号。

```
        return -EAGAIN;
}
```

✧ 否则返回错误码。

```
*p = *current;
```

4. 将当前进程的 task_struct 结构中的内容复制到新创建的 task_struct 结构中。

```
if (p->exec_domain && p->exec_domain->use_count)
    (*p->exec_domain->use_count)++;
if (p->binfmt && p->binfmt->use_count)
    (*p->binfmt->use_count)++;
```

5. 将其中 exec_domain 与 binfmt 的引用计数加 1，因为这些数据在父子进程间总是可以共享的。

```
p->did_exec = 0;
p->swappable = 0;
```

6. 子进程的 did_exec 标志置为 0，表示判断进程不是通过系统调用 execve()而重新装载执行的（为了遵循 POSIX 标准）。
   子进程的 swappable 可交换标志置为 0，因为尚未初始化完毕，不能被交换。

```
p->kernel_stack_page = new_stack;
*(unsigned long *) p->kernel_stack_page = STACK_MAGIC;
```

7. 将第 2 步时新创建的新内核页分配给新进程的内核堆栈，且将其最低 4 个字节（即堆栈栈底）置为 STACK_MAGIC （#define STACK_MAGIC 0xdeadbeef，include/linux/kernel.h）。这样，在进程退出时，便可根据这个值判断进程的数据结构是否被毁坏。

```
p->state = TASK_UNINTERRUPTIBLE;
p->flags &= ~(PF_PTRACED|PF_TRACESYS|PF_SUPERPRIV);
p->flags |= PF_FORKNOEXEC;
```

8. 将新进程状态置为 TASK_UNINTERRUPTIBLE，清除进程 flags 中的

PF_PTRACED、PF_TRACESYS、PF_SUPERPRIV 等位（因为，父进程的这些状态不可被子进程继承），将 PF_FORKNOEXEC（由 fork 创建而非 exec 执行标志）置位。

```
p->pid = get_pid(clone_flags);
```

9. 为新进程分配 pid，get_pid 函数处理过程如下：

```
int last_pid=0;

static int get_pid(unsigned long flags)
{
    struct task_struct *p;


    if (flags & CLONE_PID)
        return current->pid;
```

✧ 若 clone_flags 中 CLONE_PID 置位，表明父子进程共享同一个 pid，则将父进程的 pid 赋给子进程；

```
repeat:
    if ((++last_pid) & 0xffff8000)
        last_pid=1;
    for_each_task (p) {
        if (p->pid == last_pid || p->pgrp == last_pid || p->session == last_pid)
            goto repeat;
    }
    return last_pid;
}
```

✧ 否则，从上一次创建的进程的 pid 开始在 1~0x8000 中以 1 为增量查找一个未被所有进程的 pid，group，session 使用过的数，将该数作为进程的 pid。

```
p->next_run = NULL;
p->prev_run = NULL;
p->p_pptr = p->p_opptr = current;
p->p_cptr = NULL;
```

10. 初始化 next_run、prev_run、p_pptr、p_opptr 及 p_cptr 等进程链信息。

```
init_waitqueue(&p->wait_chldexit);
```

11. 初始化子进程的等待队列。参见“队列的操作”。

```
p->signal = 0;
```

12. 将 signal 清 0，未接收到信号。

```
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;
```

13. 三种定时器（Real、Virtual、Profile）计数值都清 0。然后调用 init_timer(&p->real_timer)（include/linux/timer.h）初始化子进程的实时钟，并将子进程 task_struct 结构的地址传给 real_timer，详见“时钟与定时器”。

```
p->leader = 0;          /* session leadership doesn't inherit */
p->tty_old_pgrp = 0;
```

14. 由于 session 的 leader 不能遗传，故将子进程的 leader 置为 0，tty_old_pgrp（？？）也置为 0。

```
p->utime = p->stime = 0;
p->cutime = p->cstime = 0;
p->start_time = jiffies;
```

15. 子进程尚未执行，将其 utime、stime、cutime、cstime 都初始化为 0。
start_time 置为 jiffies，jiffies 为从 1970.1.1 0:00:00 开始到现在发生的时钟滴答数。

```
task[nr] = p;
SET_LINKS(p);
nr_tasks++;
```

16. 将新的 task_struct 结构填入 task 数组中，调整进程链信息，修改系统的进程计数。

```
error = -ENOMEM;
/* copy all the process information */
if (copy_files(clone_flags, p))
    goto bad_fork_cleanup;
```

17. 修改子进程打开文件的信息，若发生错误，转相应的出错处理。
    - ✧ 若 clone_flags 中的 CLONE_FILES 置位，父子进程共享所打开的文件，则将 files 的引用计数加 1；
    - ✧ 否则，创建一个新的 files_struct 结构，赋予新 task_struct 的 files 指针，然后将父进程打开的所有文件的描述符一一拷贝到这个新结构中（也就是使这个结构与父进程的一致），同时要将这些文件的引用计数加 1。

    此时，错误码重新设置为-ENOMEM，因为只可能由于内存不足而产生错误。

```
if (copy_fs(clone_flags, p))
    goto bad_fork_cleanup_files;
```

18. 修改子进程文件系统的信息，发生错误则转相应的出错处理。
    - ✧ 若 clone_flags 中的 CLONE_FILES 置位，父子进程共享文件系统，则将 fs 的引用计数加 1；
    - ✧ 否则，创建一个新的 fs_struct 结构，复制其中的内容，并修改 root、pwd 这两个 inode 的引用计数。

```
if (copy_sighand(clone_flags, p))
    goto bad_fork_cleanup_fs;
```

19. 修改子进程的消息处理句柄，发生错误则转相应的出错处理。
    - ✧ 若 clone_flags 中的 CLONE_SIGHAND 置位，父子进程共享消息处理句柄，将 sig 的引用计数加 1；
    - ✧ 否则，创建一个新的 signal_struct 结构，复制父进程 signal_struct 内容到其中。

```
if (copy_mm(clone_flags, p))
    goto bad_fork_cleanup_sighand;
```

20. 修改子进程 mm_struct 的信息，发生错误则转相应的出错处理。
    - ✧ 若 clone_flags 中的 CLONE_VM 置位，父子进程共享虚存，表明创建的是一个线程。调用 SET_PAGE_DIR（include/asm/pgtable.h）将子进程 tss 的页目录寄存器 cr3 置为 mm_struct 中的页目录，mm_struct 的引用计数加 1；
    - ✧ 否则，创建一个新的 mm_struct 结构，复制父进程 mm_struct 内容到其中。

```
copy_thread(nr, clone_flags, usp, p, regs);
```

21. 设置子进程的执行上下文 TSS，详见“进程的切换”。参数说明：regs 为父进程堆栈中的 pt_regs 结构；nr 为子进程 task_struct 在 task[]数组中的位置，即该进程 LDT、TSS 在 GDT 中的入口位置；clone_flags 参数在 copy_thread 函数中并未使用；regs 为父进程堆栈中的 pt_regs 结构，将被复制到子进程的堆栈中；usp 将复制给子进程 pt_regs 结构的 esp；p 为子进程的 task_struct。

```
p->semundo = NULL;
```

22. 初始化进程的信号量 semundo。

```
/* ok, now we should be set up.. */
p->swappable = 1;
```

23. 此时，进程的 task_struct 结构已经设置完毕。将 swappable 标志置 1，允许占用的内存被置换到交换区中。

```
p->exit_signal = clone_flags & CSIGNAL;
```

24. exit_signal 在进程退出时，作为信号发送给父进程，在 include/linux/sched.h 中有如下定义：

```
#define CSIGNAL     0x000000ff      /* signal mask to be sent at exit */
#define CLONE_VM 0x00000100     /* set if VM shared between processes */
#define CLONE_FS  0x00000200     /* set if fs info shared between processes */
#define CLONE_FILES     0x00000400     /* set if open files shared between processes */
#define CLONE_SIGHAND       0x00000800     /* set if signal handlers shared */
#define CLONE_PID       0x00001000     /* set if pid shared */
```

在 include/asm-i386/signal.h 中有如下定义：

```
#define SIGCHLD         17
```

正如在“线程”一节中提到的，fork 使用 SIGCHLD 克隆标志来调用 do_fork，而 clone 可使用 CLONE_VM 、CLONE_FS 、CLONE_FILES、CLONE_SIGHAND 等 CLONE_PID 克隆标志来调用 do_fork 。通过这些克隆标志的值可以看出 clone_flags 与 CSIGNAL 运算后的结果为 0 或 SIGCHILD。于是，clone 创建的特殊进程（一般是内核进程即线程）在退出时并不发送信号给父进程，而 fork 创建的普通进程在退出时执行一般处理，发送 SIGCHLD 信号给父进程。参见“进程的退出”。

```
p->counter = (current->counter >>= 1);
```

25. 将计数器 counter 减半，并赋予子进程。参见“进程的调度”。

```
wake_up_process(p);                    /* do this last, just in case */
++total_forks;
```

26. 将子进程的状态改为 TASK_RUNNING，并插入到运行队列中。增加 fork 创建进程的计数 total_forks，用于一些系统数据的统计。

```
return p->pid;
```

27. 返回子进程的 pid。由于父进程与子进程得到的返回值是不同的，为这个不同的返回值还需要做一些额外的的处理，详见“进程的切换”。

```
bad_fork_cleanup_sighand:
        exit_sighand(p);
bad_fork_cleanup_fs:
        exit_fs(p);
bad_fork_cleanup_files:
        exit_files(p);
bad_fork_cleanup:
        if (p->exec_domain && p->exec_domain->use_count)
                (*p->exec_domain->use_count)--;
        if (p->binfmt && p->binfmt->use_count)
                (*p->binfmt->use_count)--;
        task[nr] = NULL;
        REMOVE_LINKS(p);
        nr_tasks--;
bad_fork_free_stack:
        free_kernel_stack(new_stack);
bad_fork_free_p:
        kfree(p);
bad_fork:
        return error;
}
```

28. 执行相应的出错处理，分别释放之前已经分配的内存，恢复已改动的数据等。

三、 进程的调度

1. 进程的调度策略

为了符合 POSIX 标准，Linux 中实现了三种进程调度策略：

- ✧ SCHED_OTHER。一般进程。
- ✧ SCHED_FIFO。先进先出（First In First Out）的实时进程。
- ✧ SCHED_RR。循环赛（Round Robin）方式执行的实时进程。

Linux 并不为这三种调度策略的进程分别设置一个运行队列，而是通过权重的不同计算以及其他的一些队列操作，在一个运行队列中实现这三种不同的调度。发生进程调度时，调度程序要在运行队列中选择一个最值得运行的进程来执行，这个进程便是通过在运行队列中一一比较各个可运行进程的权重来选择的。权重越大的进程越优，而对于相同权重的进程，

在运行队列中的位置越靠前越优。

如果当前进程的优先级和某个其他可运行进程一样，而当前进程至少已花费了一个时钟滴答，计算出来的权重必定小于这个进程，因此总处于劣势。如果当前进程的权重与某个其他可运行进程一样，则将当前进程的权重稍稍增大，以便在没有更高权重进程的情况下可选择当前进程继续执行，减少进程切换的开销。

相对于一般进程，实时进程总是会被认为是最值得运行的进程，只要队列中有一个实时进程就会选择该进程执行。与一般进程不同，实时进程权重的计算与进程的已执行时间无关，通过相对优先级反映，相对优先级越大则权重越大。

调度策略为 SCHED_RR 的实时进程，在分配的时间片到期后，插入到运行队列的队尾。对于相对优先级相同的其他 SCHED_RR 进程，此时它们的权重相同，但由于调度程序从运行队列的头部开始搜索，当前进程在队尾不会先被选择，其他进程便有了更大的机会执行。这个进程执行直至时间片到期，也插入队尾。同样，此时就会先选择上一次运行的那个进程。这便是“循环赛”策略名字的来由。

与 SCHED_RR 不同的是，调度策略为 SCHED_FIFO 的进程，在时间片到期后，调度程序并不改变该进程在运行队列中的位置。于是， 除非有一个更高相对优先级的实时进程入队，否则将一直执行该进程，直至该进程放弃执行或结束。这也是“先进先出”策略名字的来由。

进程的调度策略可以通过 setscheduler 函数（kernel/sched.c）改变，同时需要设置进程的相对优先级。超级用户可以改变所有进程的调度策略，而其他用户只能改变他自己执行的进程的调度策略。若设置调度策略为 SCHED_OTHER，相对优先级只能为 0；若为 SCHED_FIFO 与 SCHED_RR，其相对优先级可设置为 1 ~ 99。在改变了调度策略后，若进程在运行队列中，则将进程移到队尾，同时置调度标志 need_sched。

相关的系统调用：sys_sched_setscheduler，sys_sched_setparam，sys_sched_getscheduler，sys_sched_getparam，sys_sched_get_priority_max，sys_sched_get_priority_min。顾名思义，都可以了解这些系统调用的功能。

2．进程的权重

发生进程调度时，调度程序要在运行队列中选择一个最值得运行的进程来执行，这个进程便是通过在运行队列中一一比较各个可运行进程的权重来选择的。权重越大的进程越优，而对于相同权重的进程，在运行队列中的位置越靠前越优。进程权重的计算各不相同，与进程的调度策略、优先级以及已执行时间都有关系。

以下分别介绍与进程权重计算相关的各要素。

- 优先级 priority

进程的优先级反映了进程相对于其他进程的可选择度，其实就是系统每次允许进程运行的时间（时钟滴答数）。子进程继承了父进程的优先级。priority 也可以通过系统调用 sys_setpriority（sys_nice 已被 sys_setpriority 取代）设置。

系统为每个进程预定的 priority 为 DEF_PRIORITY（include/linux/sched.h），

```
#define DEF_PRIORITY   (20*HZ/100)
```

即每个进程每次最多只能执行 20 个时钟滴答的时间（200ms）。

根据 UNIX 的传统，进程的优先级从-20 到 20，-20 优先等级最高而 20 最低，缺省的优先级为 0。只有超级用户可以为进程设置数值小于 0 的优先级，而普通用户只能为他的进程设置数值大于 1 的优先级。为了使 UNIX 传统的-20..20 的优先级与 Linux 中的优先级代表进程可运行时间对应起来，在系统调用 sys_setpriority 与 sys_getpriority 中做了一些圆整以及符

号的处理，使优先级 0 对应的 priority 值为 20，优先级 20 对应的 priority 值为 1，优先级-20 对应的 priority 值为 40。

- 相对优先级 rt_priority

对于实时进程，除了用 priority 来反映其优先级（可执行时间）外，还有相对优先级用于同类进程之间的比较选择。实时进程的 rt_priority 取值 1 ~ 99，一般进程的 rt_priority 值只能取 0。进程的 rt_priority 可通过 setscheduler 函数而改变。

- 计数器 counter

counter 用以反映进程所剩余的可运行时间，在进程运行期间，每次发生时钟中断时，其值减 1，直至 0。由于时钟中断为快中断，在其底半处理过程中，才刷新当前进程的 counter 值。因此，也可能在发生了好几次时钟中断后才集中进行处理。

计数器 counter 是衡量一般进程权重的重要指标，主要因为如下几种事件而改变：

- ✧ task 0 的 counter 初值为 DEF_PRIORITY，在执行 sys_idle()时，将 counter 值置为 -100。
- ✧ 在创建子进程时，父进程的 counter 变为原值的一半，并将该值赋予子进程。
- ✧ 在进程运行期间，每次发生时钟中断时，counter 值减 1，直至为 0。
- ✧ 若所有的可运行进程的 counter 值都为 0，则需要为所有的进程都重新赋 counter 值，counter = counter/2 + priority。

- 权重通过调用函数 goodness（kernel/sched.c）来计算，过程如下：

```
static inline int goodness(struct task_struct * p, struct task_struct * prev, int this_cpu)
{
    int weight;
    if (p->policy != SCHED_OTHER)
        return 1000 + p->rt_priority;
```

✧ 对于实时进程，其权重为 1000+rt_priority；

```
    weight = p->counter;
```

✧ 否则，权重为 counter；

```
    if (weight) {
        /* .. and a slight advantage to the current process */
        if (p == prev)
            weight += 1;
```

✧ 对于当前进程，可以得到比其他进程稍高的权重，为 counter+1。这样处理是为了在某个进程与当前进程权重相同时可选择当前进程继续执行，以减少进程切换的开销。

```
    }
    return weight;
}
```

3．调度程序的执行过程

调度函数 schedule（kernel/sched.c）的执行过程如下：

```
asmlinkage void schedule(void)
{
    int c;
    struct task_struct * p;
    struct task_struct * prev, * next;
    unsigned long timeout = 0;
    if (intr_count)
        goto scheduling_in_interrupt;
```

1. 中断发生时，不可以调度程序。中断处理函数以及底半处理函数使用 intr_count 防止重入。

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

2. 首先，执行底半处理，将快中断遗留的任务完成。

```
run_task_queue(&tq_scheduler);
```

3. 处理其他进程遗留给调度程序执行的任务队列。

```
need_resched = 0;
prev = current;
cli();
```

4. 清 need_resched 调度标志，清中断。

```
/* move an exhausted RR process to be last.. */
if (!prev->counter && prev->policy == SCHED_RR) {
    prev->counter = prev->priority;
    move_last_runqueue(prev);
}
```

5. 处理当前进程：
   - ✧ 若当前进程的调度策略为 SCHED_RR，而且时间片已经用尽（counter=0），根据 priority 重置 counter，并将进程移到队尾。

```
switch (prev->state) {
    case TASK_INTERRUPTIBLE:
        if (prev->signal & ~prev->blocked)
            goto makerunnable;
```

   - ✧ 如果该进程是 TASK_INTERRUPTIBLE，并且自上次调度以来接收到未阻塞的信号，则任务状态设置为 TASK_RUNNING。

```
        timeout = prev->timeout;
        if (timeout && (timeout <= jiffies)) {
            prev->timeout = 0;
            timeout = 0;
```

```
        makerunnable:
            prev->state = TASK_RUNNING;
            break;
        }
```

- ✧ 如果当前进程设置了定时器 timeout，而且已经到了预定的时间（timeout<=jiffies），则定时器 timeout 清 0，进程状态变为 TASK_RUNNING。

```
        default:
            del_from_runqueue(prev);
        case TASK_RUNNING:
    }
```

- ✧ 如果当前进程状态为 TASK_RUNNING，则继续保持此状态。
- ✧ 否则，进程既不处于 TASK_RUNNING 状态，也不是 TASK_INTERRUPTIBLE，则调用 del_from_runqueue 函数将该进程移出运行队列。于是，在调度程序选择最优进程时，该进程不被考虑。

```
p = init_task.next_run;
sti();
```

6. 开中断。此时可能会有新的进程加入到运行队列中。

```
#define idle_task (&init_task)
    c = -1000;
    next = idle_task;
    while (p != &init_task) {
        int weight = goodness(p, prev, this_cpu);
        if (weight > c)
            c = weight, next = p;
        p = p->next_run;
    }
```

7. 调度程序在运行队列中一一比较权重，搜索一个最值得运行的进程。权重的计算如上所述。搜索的结果可能出现如下三种情况：

- ✧ 若 c 为负，则表明运行队列中没有可运行进程，此时 next 保持初值 idle_task，将选择 task 0 运行。
- ✧ 若 c 为 0，表明运行队列中，所有可运行进程的时间片都已用光，需要为所有的进程都重新计算 counter 值。此时，next 指向队头进程。

```
if (!c) {
    for_each_task(p)
        p->counter = (p->counter >> 1) + p->priority;
}
```

- ✧ 否则，c 为正数。此时，next 指向权重最大的那个进程。

```
if (prev != next) {
```

8. 若选择的进程为当前进程，则继续执行。否则，要暂停当前进程，切换到所选择的

进程重新执行：

```
struct timer_list timer;
kstat.context_swtch++;
```

✧ kstat（include/linux/kernel_stat.h）中存储了内核的一些数据，如 CPU 的使用率，进程切换的次数等。

```
if (timeout) {
    init_timer(&timer);
    timer.expires = timeout;
    timer.data = (unsigned long) prev;
    timer.function = process_timeout;
    add_timer(&timer);
}
get_mmu_context(next);
```

✧ 调用 get_mmu_context，设置预运行进程 mmu 的上下文。对于 i386 CPU，并不执行任何操作；对于 SPARC，这一操作在进程切换中完成。

```
switch_to(prev,next);
```

✧ 进行进程切换，详见“进程的切换”。

```
if (timeout)
    del_timer(&timer);
```

✧ 当前进程设置了 timeout 定时器，参见“时钟与定时器”。

```
}
return;
scheduling_in_interrupt:
    printk("Aiee: scheduling in interrupt %p\n",
        __builtin_return_address(0));
}
```

4．调度的发生条件

- 调度程序 schedule()的触发
  - ✧ 在系统调用 system_call 退出以及慢中断退出时，都会跳转到 ret_from_sys_call 去执行。在 ret_from_sys_call 中，先判断 need_resched 调度标志，若不为 0，调用 schedule()重新调度。
  - ✧ 某些系统调用的处理函数，如 sleep_on()，直接调用 schedule()重新调度。
  - ✧ task0 在其 idle 的过程中调用 schedule()重新调度。
  - ✧ 对文件的读写操作完成时，需要判断 need_resched 调度标志，若置位，则调用 schedule()重新调度。
- need_resched 调度标志的设置
  - ✧ 将进程插入运行队列时，若进程为实时进程，且该进程的 counter 大于当前进程的 counter+3，则置调度标志。

- ✧ 每一次时钟滴答，都要判断是否需要使用被置换到交换分区上的页。若需要，则将 kswapd 进程唤醒，插入到运行队列中，同时置 need_resched 标志。由于 kswapd 进程是一个 SCHED_RR 进程，优先级较高，所以调度时必先执行，将该页重新取回到内存中。
- ✧ 每一次时钟滴答，都需要更新当前进程的计数器 counter。若 counter 减为 0，表明该进程分配的时间片已到期，则置调度标志，以便选取其他进程执行。
- ✧ 在使用系统调用改变进程的调度策略时，若进程在运行队列中，则将进程移到队尾，同时置调度标志 need_sched。
- ✧ 进程可以通过系统调用 sys_sched_yield 放弃执行，此时将进程移到队尾，进程计数器清 0，同时置调度标志 need_sched。
- ✧ 进程调度的过程中，清调度标志。

5．队列的操作

Linux 中有一个 task 指针数组，所有进程的 task_struct 结构都通过 task 数组中的指针

```
struct task_struct * task[NR_TASKS] = {&init_task, };
```

索引。每个进程结构各有两个指针，将所有进程链接成一个双向链表。每个进程还有两个指针，将所有的可运行进程链接起来。这两个双向链表的根都是 init_task，每次对进程的搜索都通过 init_task 开始。

1. 运行队列

运行队列通过进程 task_struct 结构中的 prev_run 与 next_run 指针链接起来，利用根节点 init_task 在链表中查找某个运行进程十分方便。对运行队列的操作比较简单，主要可归结为以下几种：

- 入队操作与出队操作。这两个操作是对运行队列最基本的操作，分别通过函数 add_to_runqueue 与 del_from_runqueue 实现，主要是对这个双向链表指针的调整操作。在这两个函数中，要先对 prev_run 与 next_run 指针进行判断，不为 NULL，方可出队；若为 NULL，方可入队。另外，由于 task 0 不能被移出运行队列，在出队操作需要做相应的判断。
  由于对队列的操作必须是原子操作，而在这两个函数中并不屏蔽中断，中断的屏蔽需要在上层调用函数中实现。
- 唤醒操作。首先关中断，将进程的状态改为 TASK_RUNNING，插入到运行队列中。然后开中断。

2. 等待队列

等待队列是一个单向链表结构，每个节点为

```
struct wait_queue {
    struct task_struct * task;
    struct wait_queue * next;
};
```

结构，task 为等待某事件的进程的结构，next 指向下一个节点。在等待队列被唤醒时，将其中的进程依次插入到运行队列中。

Linux 关于等待队列的处理比较巧妙，等待队列是一个带头节点的循环链表。由于等待

队列的头节点其实并不使用，只是起索引作用，因此没有必要为头节点也分配空间。通过宏 WAIT_QUEUE_HEAD 可以得到一个虚拟的头节点。

```
#define WAIT_QUEUE_HEAD(x) ((struct wait_queue *)((x)-1))
```

将该等待队列的指针（4 个字节），以及该指针之前的 4 个字节，通过强制类型转化，可虚拟为一个 wait_queue 节点。由于头节点中的内容不使用，并不会破坏那 4 个字节的内容。如此处理可为每个等待队列都节省一个节点的空间。如图 2 所示。

```
static inline void init_waitqueue(struct wait_queue **q)
{
        *q = WAIT_QUEUE_HEAD(q);
}
```

等待队列的初始化，其实将等待队列的头指针指向其地址之前的 4 个字节处。如此便可组成一个循环链表。

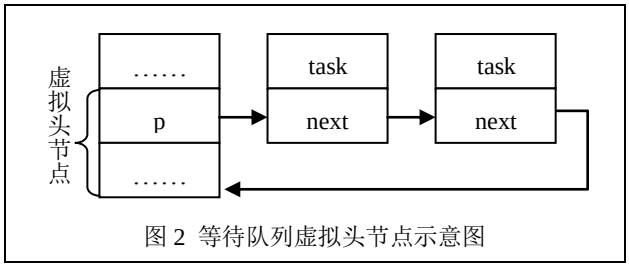

图 2 等待队列虚拟头节点示意图

```
extern inline void __add_wait_queue(struct wait_queue ** p, struct wait_queue * wait)
{
        struct wait_queue *head = *p;
        struct wait_queue *next = WAIT_QUEUE_HEAD(p);
        if (head)
                next = head;
        *p = wait;
        wait->next = next;
}
```

相应的入队操作。在该循环链表中，将虚拟的头节点作为队尾以及队头标志。

四、 进程的切换

以下以 i386CPU 为例介绍 Linux 中进程的切换操作。在 i386 平台上，使用任务这一术语，其作用相当于进程。

在 i386 平台上，有许多事件可能导致任务切换的发生，如长跳转/长调用指令所带的 16 位段地址选择的是一些特殊描述符，中断指令所带的中断号在 IDT 中索引的是一些特殊描述符等。Linux 使用 Linux 利用长跳转 ljmp 指令以发生一次任务切换。

1. TSS 结构

在进程的运行过程中，必然伴随着系统状态的改变，这些状态将影响进程的执行。当调度程序选择了一个新进程以运行时，此时将发生一次任务切换，旧进程从运行状态切换为暂停状态，而新进程从暂停状态切换为运行状态，开始继续执行。对旧进程而言，它的运行环境如寄存器、、机器状态字 MSW、指令计数器 PC 以及堆栈等，必须保存在上下文中，以便下次恢复运行；对新进程而言，它也需要从上下文中恢复上次保存的运行环境，然后继续运行。这个上下文便是 TSS 结构。

在 i386CPU 中定义了在其上运行的进程的基本 TSS 结构，如图 3 所示。

TSS 结构至少包含以上的 104 个字节（00h--67h），可分为以下三类：

- 保留域，不可使用，必须置为 0；

- 动态域，在任务开始执行或恢复执行时由处理器读出，在任务暂停时由处理器写入。如：通用寄存器（EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI），段寄存器（ES, CS, SS, DS, FS, GS），机器状态字 EFLAGS，指令计数器 EIP，以及 Link 域。
- 静态域，由处理器读出但不写入。如：LDT 域，CR3 域、012 权级堆栈指针域（SS0, SS1, SS2, ESP0, ESP1, ESP2），T 位，I/O 许可位图偏移地址域。

TSS 中 67h 以上的区域还可以包括三个部分，由操作系统指定是否使用。

- OS 指定数据，68h 开始的区域可由 OS 选用，其大小由 OS 指定。

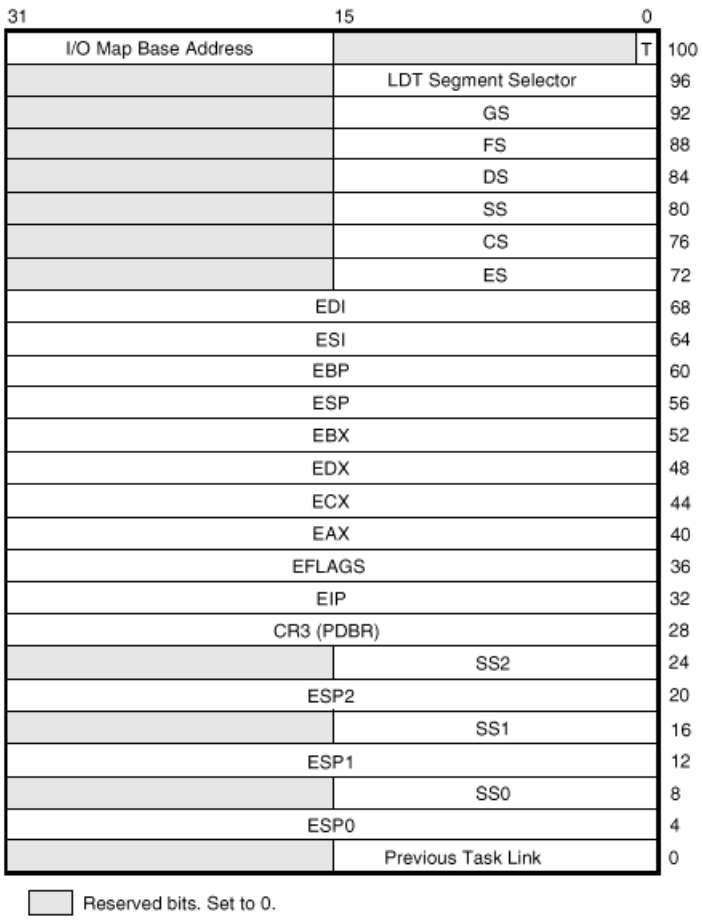

图 3. i386 的 TSS 结构

- 中断重定位位图，32 个字节，只在 OS 支持 VM86 模式时需要。
- I/O 允许位图，最大可到 8k 字节，当 OS 支持 IO 保护时使用。

TSS 结构由 TSS 段描述符表示，TSS 描述符段具有如图 4 的结构。TSS 描述符必须存放在 GDT 中。

当前任务的 TSS 描述符由 TR 寄存器索引，TR 寄存器具有段选择子的一般格式，共 16

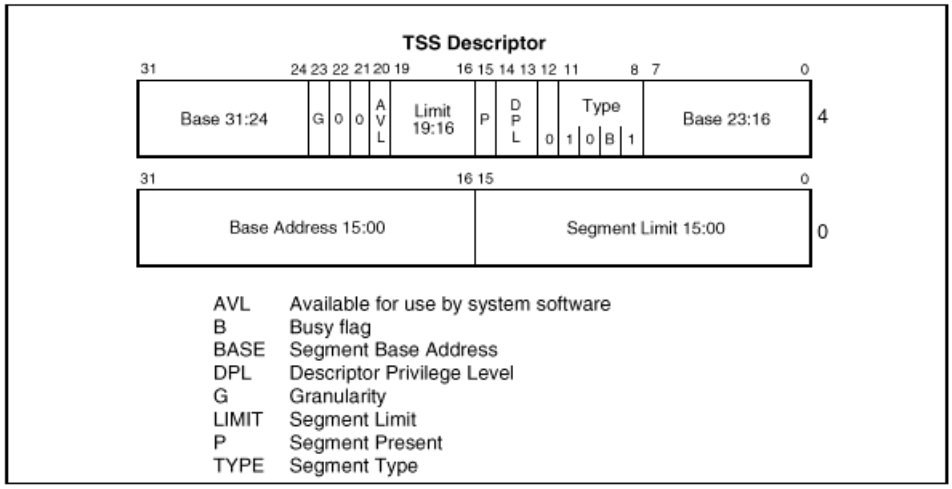

图 4 TSS 段描述符

位：第 0,1 位为 RPL，第 2 位（TI）必须为 0 表示 TSS 描述符存放在 GDT 中，高 13 位为 TSS 描述符在 GDT 中的索引。TR 寄存器可由 LDTR、SDTR 指令操作，在装入 TR 寄存器

的同时，还将其选择的 TSS 描述符装载到 TR 的投影寄存器中。TR 寄存器、TSS 描述符与 TSS 结构的关系如图 5 所示。

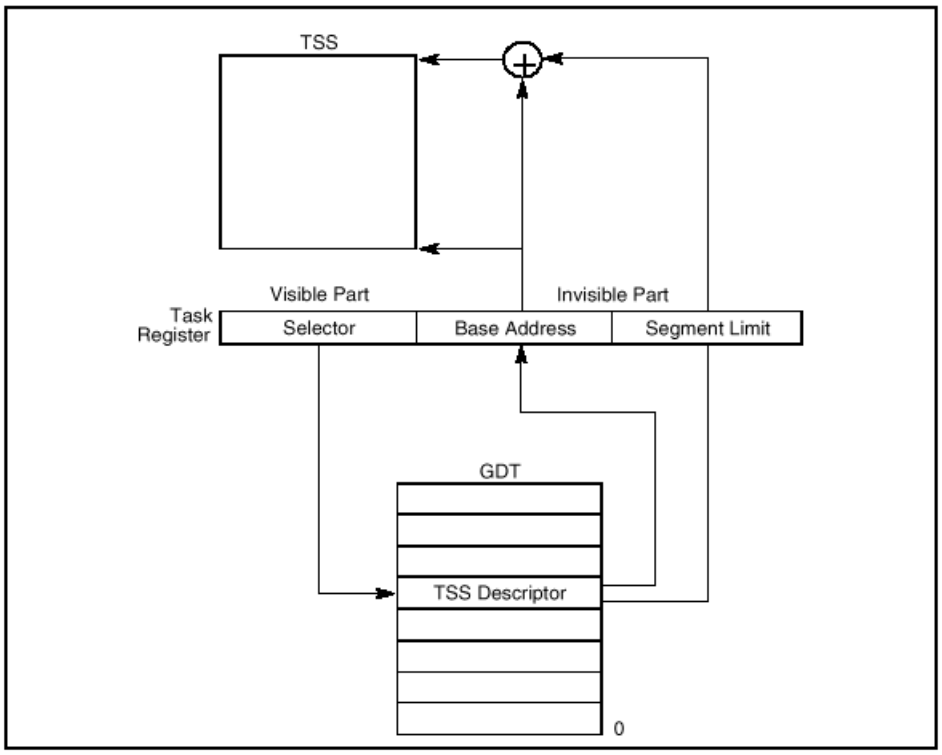

图 5 TR、TSS 描述符与 TSS 结构

Linux 用 thread_struct 结构来反映 TSS 结构，thread_struct 中除了包含以上的基本 TSS 结构（104 字节），还有定义了其他一些数据单元以方便任务的切换。

```
struct thread_struct {
    unsigned short  back_lin k,__blh;
```

1. 在发生任务嵌套时，link 指向上一个任务的 TSS 结构。

```
    unsigned long   esp0;
    unsigned short  ss0,__ss0h;
    unsigned long   esp1;
    unsigned short  ss1,__ss1h;
    unsigned long   esp2;
    unsigned short  ss2,__ss2h;
```

2. 伴随任务切换发生的还有任务权级的改变，不同权级的任务分别使用不同的堆栈。以上分别对应 0--2 权级的堆栈。这三种堆栈域为静态域，在任务切换时并不由处理器改变，而 SS:ESP 为任务自身的堆栈，在任务切换时由处理器将当前的 SS:ESP 填入。

```
    unsigned long   cr3;
```

3. 页目录寄存器，通过为不同任务指定各自的页目录，可以使任务具有不同的虚实地址映射函数，以达到任务虚地址空间的分离。Linux 中的所有进程都使用同一个页目录，即 swapper_pg_dir（arch/i386/kernel/head.S）。

```
    unsigned long   eip;
```

4. 指令计数器，cs:eip 指向了进程恢复执行后的第一条指令。在创建进程时，新进程 TSS 结构中的 eip 将被初始化为 ret_from_sys_call。于是，在调度程序选择了新进程时，新进程将从标号 ret_from_sys_call 处开始执行，看起来的效果，同父进程一样，也似乎是刚执行了 fork 系统调用，然后返回。而且另外做的一些处理将使子进程得到的返回值为 0，而父进程得到的返回值为子进程的 pid，以便区分。

```
unsigned long   eflags;
```

5. 机器状态字。

```
unsigned long   eax,ecx,edx,ebx;
unsigned long   esp;
unsigned long   ebp;
unsigned long   esi;
unsigned long   edi;
unsigned short  es, __esh;
unsigned short  cs, __csh;
unsigned short  ss, __ssh;
unsigned short  ds, __dsh;
unsigned short  fs, __fsh;
unsigned short  gs, __gsh;
```

6. i386CPU 中相应的寄存器。

```
unsigned short  ldt, __ldth;
```

7. 各个任务的 LDT。Linux 中所有进程都使用相同的 ldt——default_ldt，这个 ldt 中只有一个表项，为指向 KERNEL_CS:lcall7 且 RPL 为 3 的调用门描述符

```
unsigned short  trace, bitmap;
```

8. trace 对应着 TSS 结构 64h 的第 0 位即 Debug Trap 位（T），在切换到一个 T 位为 1 的任务时，将产生一个调试异常。该位可用以在任务切换时设置断点。
bitmap 对应着 TSS 结构的 66h、67h 两个字节，是一个 16 位的偏移量，保存 IO 许可位图的偏移地址。于是，TSS 结构 67h 以上的三个 OS 指定区域各自的偏移地址都可以得到。

```
unsigned long   io_bitmap[IO_BITMAP_SIZE+1];
```

9. IO 许可位图。Linux 只使用了 i386 提供的 TSS 结构 67h 以上三个 OS 指定区域中的 IO 许可位图，其它两个区域并不使用。
在进程执行一些与 I/O 相关的指令（IN, INS, OUT, OUTS, CLI, STI）时，首先要进行权限的判断：

   - 若 CPL 的数值小于或等于 IOPL，IO 指令正常执行。
   - 若 CPL 的数值大于 IOPL，而且执行的是 IN, INS, OUT 或 OUTS 指令时，处理器将在当前任务的 TSS 结构中检查 IO 许可位图，若 IO 指令访问的地址在 IO 许可位图中对应的位为 0，则允许执行；否则，产生 GP 异常。
   - 若 CPL 的数值大于 IOPL，而且执行的是 CLI 或 STI 指令时，将产生 GP 异常。

   I/O 地址空间最大可达 64kB，io_bitmap 以位来表示相应的 IO 地址的访问权限，其长度最大可达 8kB。通常，处理器每次读 IO 许可位图时的长度为两个字节，为了防止在 IO

许可位图边界处可能发生的异常，故在实际 IO 许可位图的后面添上两个全为 1 的字节，而且将这两个字节计入 TSS 结构的长度，正如以上 io_bitmap[IO_BITMAP_SIZE+1]所定义的。

以上为实际的 TSS 结构，由处理器自动处理，在描述符中定义了其长度为 235 字节。

```
    unsigned long   tr;
```

10. 本 TSS 结构对应的描述符的选择子，在任务切换时，将作为 ljmp 指令所带的 16 位段地址。

```
    unsigned long   cr2, trap_no, error_code;
/* floating point info */
    union i387_union i387;
/* virtual 86 mode info */
    struct vm86_struct * vm86_info;
    unsigned long screen_bitmap;
    unsigned long v86flags, v86mask, v86mode;
};
```

11. 相应的其它数据结构。10 与 11 均为 Linux 在 TSS 中扩充的数据，TSS 描述符中的段偏移并不包含这些数据的长度，处理器也不会自动处理。

2. TSS 相关数据的设置与初始化

任务切换发生时，根据当前 TR 寄存器选择的 TSS 描述符（已经预存到 TR 的投影寄存器中），将当前任务的运行环境存放到当前任务的 TSS 段中，然后根据导致任务切换的 TSS 段选择子，找出新任务的 TSS 段描述符，并将 TSS 段中的内容恢复为新进程的运行环境。于是，新进程便可以继续执行。

为了进行任务的切换，要设置好新旧进程的 TSS 段选择子、描述符以及 TSS 段本身。这些数据的设置和定义都在头文件 include/asm-i386/system.h 中。

TSS 描述符只能存放于 GDT 中。如 arch/kernel/head.S 所示，每个进程的 TSS 描述符、LDT 描述符从表项 FIRST_TSS_ENTRY 开始，依次存放，并利用宏_TSS、_LDT 作为选择子索引。各个进程 TSS、LDT 描述符的存放次序与该进程的进程结构在 task 数组中的次序相同。

```
#define FIRST_TSS_ENTRY 8
#define FIRST_LDT_ENTRY (FIRST_TSS_ENTRY+1)
#define _TSS(n) ((((unsigned long) n)<<4)+(FIRST_TSS_ENTRY<<3))
#define _LDT(n) ((((unsigned long) n)<<4)+(FIRST_LDT_ENTRY<<3))
```

由_TSS、_LDT 的定义可以看出，选择子的 TI 位置为 0，RPL 也置为 0。

```
#define _set_tssldt_desc(n,addr,limit,type) \
__asm__ __volatile__ ("movw $" #limit ",%1\n\t" \
    "movw %%ax,%2\n\t" \
    "rorl $16,%%eax\n\t" \
    "movb %%al,%3\n\t" \
    "movb $" type ",%4\n\t" \
    "movb $0x00,%5\n\t" \
    "movb %%ah,%6\n\t" \
```

```
        "rorl $16,%%eax" \
        : /* no output */ \
        :"a" (addr+0xc0000000), "m" (*(n)), "m" (*(n+2)), "m" (*(n+4)), \
         "m" (*(n+5)), "m" (*(n+6)), "m" (*(n+7)) \
        )
#define set_tss_desc(n,addr) _set_tssldt_desc(((char *) (n)),((int)(addr)),235,"0x89")
#define set_ldt_desc(n,addr,size) \
        _set_tssldt_desc(((char *) (n)),((int)(addr)),((size << 3) - 1),"0x82")
```

利用宏 set_tss_desc(n,addr)、set_ldt_desc(n,addr,size)可分别组合成进程 n 的 TSS、LDT 描述符，并通过给定相应的表项地址存放到 GDT 中。TSS 结构的长度为 235。如 task0 的 TSS 描述符、LDT 描述符的装载过程：

```
        p = (struct { unsigned long a[2]} *)gdt+FIRST_TSS_ENTRY;
        set_tss_desc(p, &init_task.tss);
        p++;
        set_ldt_desc(p, &default_ldt, 1);
```

task0 的 TSS 描述符、LDT 描述符在函数中 trap_init（arch/kernel/trap.c）中装载到 GDT 中，其它进程的 TSS 描述符、LDT 描述符在进程 TSS 创建的过程中装载到 GDT 中。

```
#define load_TR(n) __asm__("ltr %%ax": /* no output */ :"a" (_TSS(n)))
```

TSS 描述符的选择子可通过 LTR 指令装载到 TR 寄存器中。只有在系统初始化（trap_init）时需要显式使用 LTR 指令将当前 TR 寄存器初始化为 task0 的 TSS 选择子。

```
        load_TR(0);
```

在进程切换的过程中，旧进程的当前运行环境将通过 TR 寄存器存入该进程的 TSS 结构中，而新进程的 TSS 选择子将装载到 TR 寄存器中，这些操作都是由处理器自动完成的。

3. TSS 的创建以及进程堆栈的变换

只有 task0 的 TSS 结构在初始化由人工拼装，为 INIT_TSS（include/asm-i386/process.h）。其它进程的 TSS 结构均在进程的创建过程中生成，这一操作由 copy_thread 函数（arch/i386/kernel/processor.c）实现。

```
void copy_thread(int nr, unsigned long clone_flags, unsigned long esp,
        struct task_struct * p, struct pt_regs * regs)
{
        int i;
        struct pt_regs * childregs;
        p->tss.es = KERNEL_DS;
        p->tss.cs = KERNEL_CS;
        p->tss.ss = KERNEL_DS;
        p->tss.ds = KERNEL_DS;
        p->tss.fs = USER_DS;
        p->tss.gs = KERNEL_DS;
```

1. 相关段寄存器的初始化。

```
p->tss.ss0 = KERNEL_DS;
p->tss.esp0 = p->kernel_stack_page + PAGE_SIZE;
```

2. 0 权级堆栈设置为在创建进程 task_struct 结构时为其分配的内核堆栈页，而且 esp0 固定指向栈顶。其它权级的堆栈指针继承了为 task0 拼装的 INIT_TSS 中的 0 值（Linux 并不使用 1、2 权级）。由于 ss0,esp0 为静态域，处理器并不改变，在用户进程执行系统调用时，堆栈将发生切换，由处理器取出 TSS 结构中的 ss0,esp0 作为系统调用处理函数的堆栈。内核进程执行系统调用时，不会发生堆栈切换，于是 TSS 结构中的 ss0,esp0 并不使用。参见 4。

```
p->tss.tr = _TSS(nr);
```

3. 将本 TSS 结构对应的描述符的选择子存放在 tr 中。在任务切换时，tr 将作为 ljmp 指令所带的 16 位段选择子，以选择本进程开始执行。

```
childregs = ((struct pt_regs *) (p->kernel_stack_page + PAGE_SIZE)) - 1;
p->tss.esp = (unsigned long) childregs;
p->tss.eip = (unsigned long) ret_from_sys_call;
*childregs = *regs;
```

4. 子进程在恢复运行时，需要具有同父进程相似的效果，似乎都是从系统调用 fork/clone 中退出。于是，需要设置子进程 TSS 结构中的 esp、eip 指针，并复制父进程内核栈的内容到子进程内核栈中，使得子进程的堆栈情况为即将退出系统调用时的堆栈结构，而指令计数器指向的下一条指令恰好可以使子进程从系统调用中退出。因为子进程复制了父进程的堆栈结构，从系统调用退出时，将与父进程一致，从执行系统调用处的下一条指令开始执行。

   在新进程开始执行时，进程的执行环境根据 TSS 结构中保存的各域而恢复，于是 TSS 结构中的 cs:eip 指向进程将执行的第一条指令，ss:esp 为进程此时的堆栈。将子进程的 TSS 结构中的 eip 指针置值为标号 ret_from_sys_call，而且使子进程内核栈具有和父进程内核栈同样的结构，堆栈指针 esp 也指向相同的偏移。于是，在子进程开始执行时，看起来的效果，与父进程相似，好像也是刚执行了 fork/clone 系统调用，然后即将从系统调用退出。

   在进入系统调用时，当前的各个通用寄存器以及段寄存器都保存到 pt_regs 中（cs,eip,eflags 由于中断指令而自动保存）。从系统调用退出即从 ret_from_sys_call 开始执行时（参见“系统调用的分析”entry.S.doc），将根据内核堆栈的 pt_regs 结构，恢复各个寄存器的值，然后执行 iret 指令从系统调用中返回。此时，将根据堆栈中的 cs:eip、eflags 而继续执行，而 eax 中将存放返回值。改变 pt_regs 结构，便可以使进程从系统调用退出时，得到不同的用户堆栈以及不同的返回值。于是，语句

   ```
   childregs->eax = 0;
   ```

   便可以使进程得到的返回值为 0。而语句

   ```
   childregs->esp = esp;
   ```

   可设置用户进程在退出系统调用后的用户堆栈。系统调用 clone 可以为子进程设置新的用户堆栈，而系统调用 fork 创建的子进程其用户堆栈的虚地址与父进程用户堆栈的虚地址相同。clone 将 newsp（调用 clone 的另一个参数，由调用者指定）传递给 do_fork 函数的形参 esp，而 fork 将 regs.esp（父进程的用户堆栈指针）传递给 do_fork 函数的形

参 esp。

- 若父进程是用户进程（CPL=3），在进入系统调用处理函数时，CPL 将变为新 CS 选择子的 RPL 0，随着权级的改变，还将发生堆栈的切换。根据 TSS 结构保存的系统调用处理函数所在的代码段 DPL，堆栈也将切换成与其一致权级的内核堆栈（即 ss0,esp0），然后，将 old SS, old ESP, EFlags, CS, EIP 压入内核栈中，在内核栈

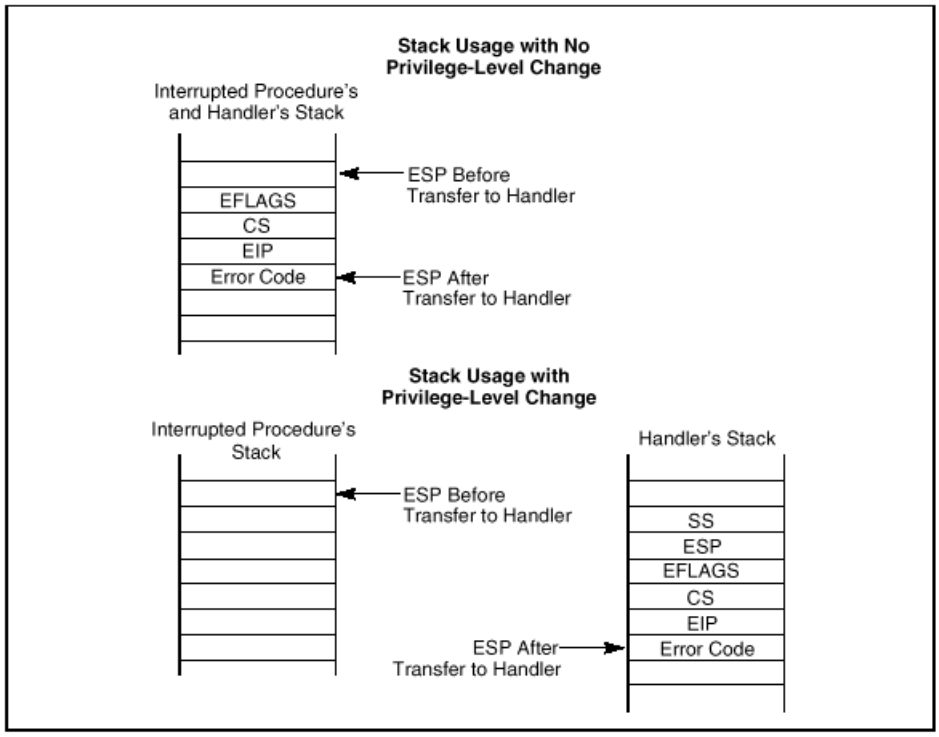

图 6 系统调用时权级的变化与堆栈的切换

中执行系统调用函数。从系统调用退出时，将根据内核栈中保存的 old SS, old ESP，恢复用户堆栈，重新执行。如图 6 所示。

- 若父进程是核心进程（CPL=0），进入系统调用处理函数时，并不发生堆栈的切换，系统调用将在当前栈中运行。退出系统调用时，还将使用同一个堆栈，也就是内核栈。

与进入系统调用相似，退出系统调用的 iret 指令执行时，根据当前 CS 选择子的 RPL（即 CPL）以及栈中保存的 CS 选择子的 RPL 进行权级的判断，若不一致，则将发生堆栈的切换。由于子进程复制了父进程的内核堆栈，虽然子进程并不是通过 int $0x80 而进入系统调用的，也将执行同父进程一样的退出操作。若父进程发生堆栈切换，子进程也将发生堆栈切换；若父进程不切换，子进程也不。

childregs->esp 中保存的是进程执行系统调用之前的用户堆栈，改变这个指针，便可以设置进程退出系统调用后的用户堆栈。而对于核心进程，由于堆栈并不切换，用户栈就是核心栈，并不改变。

```
p->tss.back_link = 0;
p->tss.eflags = regs->eflags & 0xffffcfff;      /* iopl is always 0 for a new process */
p->tss.ldt = _LDT(nr);
if (p->ldt) {
        p->ldt = (struct desc_struct*) vmalloc(LDT_ENTRIES*LDT_ENTRY_SIZE);
```

```
        if (p->ldt != NULL)
            memcpy(p->ldt, current->ldt, LDT_ENTRIES*LDT_ENTRY_SIZE);
    }
```

5. ldt 为某些应用程序（如：WINE）保留，一般进程的 ldt 均为 NULL。

```
    set_tss_desc(gdt+(nr<<1)+FIRST_TSS_ENTRY,&(p->  tss));
```

6. 组合 TSS 描述符，并装载到 GDT 的相应位置中。

```
    if (p->ldt)
        set_ldt_desc(gdt+(nr<<1)+FIRST_LDT_ENTRY,p->ldt, LDT_ENTRIES);
    else
        set_ldt_desc(gdt+(nr<<1)+FIRST_LDT_ENTRY,&default_ldt, 1);
```

7. 组合 LDT 描述符，并装载到 GDT 的相应位置中。

```
    p->tss.bitmap = offsetof(struct thread_struct,io_bitmap);
    for (i = 0; i < IO_BITMAP_SIZE+1 ; i++) /* IO bitmap is actually SIZE+1 */
        p->tss.io_bitmap[i] = ~0;
```

8. 新进程的 IO 许可位图都初始化为 1，禁止 CPL 的数值大于 IOPL 的进程访问 IO 端口。

```
    if (last_task_used_math == current)
        __asm__("clts ; fnsave %0 ; frstor %0":"=m" (p->tss.i387));
}
```

4．进程切换动作的实现

在调度程序选择了下一个执行进程后，若不是当前进程，则要发生进程切换。任务切换发生时，根据当前 TR 寄存器选择的 TSS 描述符（已经预存到 TR 的投影寄存器中），将当前任务的运行环境存放到当前任务的 TSS 段中，然后根据导致任务切换的 TSS 段选择子，找出新任务的 TSS 段描述符，并将 TSS 段中的内容恢复为新进程的运行环境。于是，新进程便可以继续执行。

进程切换的动作由 switch_to 宏实现（include/asm-i386/processor.h）。

```
#define switch_to(prev,next) do { \
__asm__("movl %2,"SYMBOL_NAME_STR(current_set)"\n\t" \
```

1. 将 current 指针更新为新进程。

```
    "ljmp %0\n\t" \
```

2. 通过 ljmp 指令产生一次任务切换。

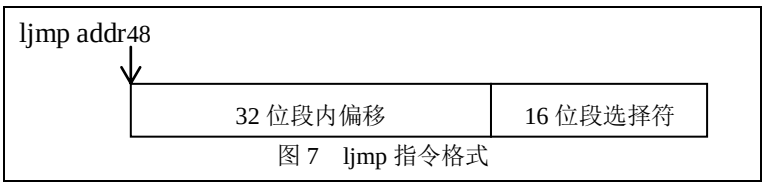

图 7　ljmp 指令格式

ljmp 指令的参数为 32 位地址，该地址处存放一个 48 位的全地址，格式如图 7 所示。

若 16 位段选择子索引的是 GDT 中的 TSS 描述符，此时将发生一次任务切换，32 位段内偏移地址并不使用。

- 首先进行权级的判断：段选择子中的 RPL 与当前任务的权级 CPL 中至少有一个与描述符权级 DPL 在同一权级上或较之为高。
- 在通过权级的判断后，由处理器自动进行任务切换操作。

- 否则，将产生 GP 异常。

%0 中的值为(*(((char *)&next->tss.tr)-4))，将从 tr 的前 4 个字节开始到 tr 的 2 个字节组合形成 ljmp 的 48 位全地址（由于 32 位段内偏移并不使用，所以可以不必理会 tr 之前 4 个字节的内容）。于是，tr 正好作为 16 位段选择子，索引的是新进程在 GDT 中的 TSS 描述符。任务切换发生。

```
        "cmpl %1,"SYMBOL_NAME_STR(last_task_used_math)"\n\t" \
        "jne 1f\n\t" \
        "clts\n" \
        "1:" \
        : /* no outputs */ \
        :"m" (*(((char *)&next->tss.tr)-4)), \
          "r" (prev), "r" (next)); \
        } \
    } while (0)
```

五、 进程的退出与消亡

通过系统调用 exit，便可以使进程终止执行。系统调用 exit 的执行函数 do_exit 的处理过程如下：

```
NORET_TYPE void do_exit(long code)
{
    if (intr_count) {
        printk("Aiee, killing interrupt handler\n");
        intr_count = 0;
    }
```

1. 在中断处理函数中，不可以使用 exit 退出。

```
    acct_process(code);
    current->flags |= PF_EXITING;
```

2. 若 acct 开关打开（可通过系统调用 acct 打开或关上），则将进程的一些数据保存到文件中。置当前进程的退出标志。

```
    del_timer(&current->real_timer);
    sem_exit();
    kerneld_exit();
    __exit_mm(current);
    __exit_files(current);
    __exit_fs(current);
    __exit_sighand(current);
```

3. 释放相应的结构，将该结构的引用计数减 1，若引用计数为 0，删除该结构。

```
    exit_thread();
```

4. 将当前的 FS、GS、LDTR 寄存器清 0。

```
    current->state = TASK_ZOMBIE;
```

5. 置当前进程状态为僵死状态，在父进程释放进程结构之前，本进程将一直保持僵死状态。调度程序不会选择本进程执行。

```
current->exit_code = code;
exit_notify();
```

6. 发送信号给父进程，将子进程退出的消息通知父进程，如果父进程由于执行了 wait4 系统调用处于等待状态，唤醒父进程。然后，将本进程所有的子进程都转移到 init 进程名下，如果本进程的某个子进程已经处于僵死状态，替该进程发退出信号给 init 进程，以便释放该进程的进程结构。

   在进程执行 wait4 系统调用时，

   - 首先，将自己插入到等待队列 wait_chldexit 中。
   - 然后，遍历所有的子进程，将处于僵死状态的子进程的进程结构释放。
   - 若要等待的进程尚未退出，将自身状态置为 TASK_INTERRUPTIBLE，调用调度程序。恢复运行时，跳转到第 2 步。
   - 否则，将自身从等待队列中移出，返回。

   只有在父进程执行 wait4 系统调用时，才会释放已经退出的子进程的进程结构，否则，子进程将一直保持 TASK_ZOMBIE 状态。

```
#ifdef DEBUG_PROC_TREE
    audit_ptree();
#endif
    if (current->exec_domain && current->exec_domain->use_count)
        (*current->exec_domain->use_count)--;
    if (current->binfmt && current->binfmt->use_count)
        (*current->binfmt->use_count)--;
```

7. 释放其它资源。

```
    schedule();
```

8. 重新调度进程。

```
}
```