10年口碑积累，成功培养50000多名研发工程师，铸就专业品牌形象

华清远见的企业理念是不仅要做良心教育、做专业教育，更要做受人尊敬的职业教育。

# 《从实践中学嵌入式 LINUX 操作系统》》

作者：华清远见



## 第 5 章　操作系统进程

本章简介

在传统的操作系统中，为了提高系统资源的利用率，通常采用多道程序技术，将多个程序同时载入内存，并使之并发执行。此时，作为资源分配和独立运行的基本单位都是进程。操作系统所具有的几大特征，也都是围绕进程形成的。在操作系统中，进程是最重要的概念。本章将讲解进程的相关内容。

# 5.1 进程的基本概念

在计算机使用过程中，我们经常谈及的概念是程序。作为最终用户，我们关心系统中哪些程序在运行，需要关闭哪个程序。但是从操作系统的范畴来说，我们使用更多的是进程。

进程和程序虽然有一定的联系，但是绝不能混为一谈。在传统的操作系统中，程序并不能独立运行，作为资源分配和独立运行的基本单元都是进程。程序是一个普通文件，是机器代码指令和数据的集合，这些指令和数据存储在磁盘上的一个可执行映像（Executable Image）中。进程是由正文段（Text）、用户数据段（User Segment）及系统数据段（System Segment）共同组成的一个执行环境，它是一个动态实体。程序是硬盘上存放的一个文件（代码）。当程序运行时，它也就成为了进程。进程的组成部分如下。

- 正文段：存放被执行的机器指令。这个段是只读的，它允许系统中正在运行的两个或多个进程之间能够共享这一代码，但是不能对其内容进行更改。
- 用户数据段：存放进程在执行时直接进行操作的所有数据，包括进程使用的全部变量在内。显然，这里包含的信息可以被改变。虽然进程之间可以共享正文段，但是每个进程需要有它自己的专用用户数据段。
- 系统数据段：该段有效地存放程序运行的环境。这也是进程和程序不同的一个原因之一。

如图 5.1 所示为程序和进程的区别。

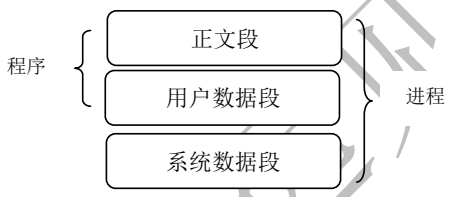


图 5.1　程序和进程的区别

Windows 和 Linux 操作系统都属于多任务系统，可以同时运行多个进程。我们经常使用的 Windows 任务管理器可以清楚地列出当前系统运行的进程，如图 5.2 所示，在图中可以看到，该系统此时正在运行的进程有 29 个。

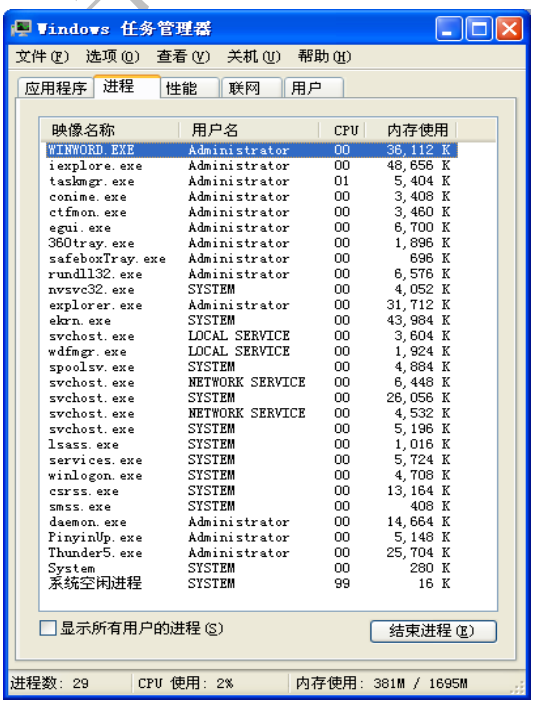


图 5.2　Windows 任务管理器

在谈到进程时，还要涉及线程的概念。线程是系统分配处理器时间资源的基本单元，或者说进程之内独立执行的一个单元。对于操作系统来说，其调度单元是线程。一个进程至少包括一个线程，通常将该线程称为主线程。一个进程从主线程的执行开始进而创建一个或多个附加线程，就是所谓基于多线程的多任务。

在 Linux 2.6 内核中，Linux 采用了更为先进的线程模型：NPTL（Native POSIX Thread Library）。与传统的 LinuxThreads 线程模型相比，NPTL 与 POSIX 标准兼容，并且性能提升明显，也具备更好的可伸缩性。对 Linux 内核而言，线程和进程没有本质的区别，它们都是调度的基本单位（实际上，Linux 内核基于进程机制实现线程），本书重点介绍进程的内容。

# 5.2 Linux 系统进程

## 5.2.1 Linux 进程基础

Linux 进程一般分为交互进程、批处理进程和守护进程 3 类。与 Windows 任务管理器一样，在 Linux 中可以通过 ps 命令查看系统当前的进程，例如，下面的命令将列出系统所有的进程：

```
[root@localhost ~]# ps -aux
```

如果进程太多，可以把 ps 命令的输出保存到一个文件中：

```
[root@localhost ~]# ps -aux  > mypsout
```

ps 是 Linux 进程管理中最重要的一个命令，它提供了很多选项参数，如表 5.1 所示。

**表 5.1 ps 命令的选项参数**

| 参　数 | 功　能 |
|---|---|
| l | 长格式输出 |
| u | 按用户名和启动时间的顺序来显示进程 |
| j | 用任务格式来显示进程 |
| f | 用树形格式来显示进程 |
| a | 显示所有用户的所有进程（包括其他用户） |
| x | 显示无控制终端的进程 |
| r | 显示运行中的进程 |
| ww | 避免详细参数被截断 |

常用的选项组合是 aux，使用不同的选项会产生不同的输出结果。表 5.2 列出了使用 ps 命令的输出列说明。

**表 5.2 ps 的输出列说明**

| 列 | 说　明 |
|---|---|

| | |
|---|---|
| USER | 进程的属主 |
| PID | 进程的 ID |
| PPID | 父进程 |
| %CPU | 进程占用的 CPU 百分比 |
| %MEM | 占用内存的百分比 |
| NI | 进程的 NICE 值，数值大，表示较少占用 CPU 时间 |
| VSZ | 进程虚拟大小 |
| RSS | 驻留中页的数量 |
| WCHAN | 正在等待的进程资源 |
| TTY | 终端 ID |
| STAT | 进程状态 |
| START | 启动进程的时间 |
| TIME | 进程消耗 CPU 的时间 |
| COMMAND | 命令的名称和参数 |

表 5.2 中的 STAT 列表示进程的状态。在 Linux 操作系统中，系统有几种不同的状态，在 CPU 的调度下，进行状态的切换。表 5.3 列出了 Linux 系统中的各种进程状态。

**表 5.3　Linux 进程状态**

| 值 | 说　明 |
|---|---|
| D | 该进程处于睡眠状态 |
| R | 该进程处于运行状态 |
| S | 该进程处于睡眠状态 |
| T | 该进程处于停止或被追踪状态 |
| W | 进入内存交换 |
| X | 死掉的进程 |
| Z | 僵尸进程 |
| < | 优先级高的进程 |
| N | 优先级较低的进程 |
| L | 有些页被锁进内存 |
| s | 进程的领导者 |
| l | 多线程进程 |
| + | 位于后台的进程组 |

### 5.2.2 进程描述符

Linux 系统的每一个可调度实体都有一个进程描述符。进程描述符可以表示进程的各种状态信息，是内核操作进程的手段。进程描述符用 task_struct 数据结构表示，该结构包含一个进程所拥有的各种信息，非常庞大，在内核文件的 sched.h 中定义。下面给出 task_struct 数据结构的片断。

```
struct task_struct {
    volatile long state;  /* -1 unrunnable, 0 runnable, >0 stopped */
    void *stack;
    atomic_t usage;
    unsigned int flags;   /* per process flags, defined below */
    unsigned int ptrace;
    int lock_depth;       /* BKL lock depth */
    int prio, static_prio, normal_prio;
    unsigned int rt_priority;
    const struct sched_class *sched_class;
    struct sched_entity se;
    struct sched_rt_entity rt;
    unsigned char fpu_counter;
    s8 oomkilladj; /* OOM kill score adjustment (bit shift). */
    unsigned int policy;
    cpumask_t cpus_allowed;
    struct list_head tasks;
    struct mm_struct *mm, *active_mm;
/* task state */
    struct linux_binfmt *binfmt;
    int exit_state;
    int exit_code, exit_signal;
    int pdeath_signal;  /*  The signal sent when the parent dies  */

    unsigned int personality;
    unsigned did_exec:1;
    pid_t pid;
    pid_t tgid;
    struct task_struct *real_parent; /* real parent process */
    struct task_struct *parent; /* recipient of SIGCHLD, wait4() reports */
    struct list_head children;/* list of my children */
    struct list_head sibling; /* linkage in my parent's children list */
    struct task_struct *group_leader;  /* threadgroup leader */
    struct list_head ptraced;
    struct list_head ptrace_entry;
    /* PID/PID hash table linkage. */
    struct pid_link pids[PIDTYPE_MAX];
    struct list_head thread_group;
    struct completion *vfork_done;     /* for vfork() */
    int __user *set_child_tid;     /* CLONE_CHILD_SETTID */
    int __user *clear_child_tid;       /* CLONE_CHILD_CLEARTID */
    cputime_t utime, stime, utimescaled, stimescaled;
    cputime_t gtime;
    cputime_t prev_utime, prev_stime;
    unsigned long nvcsw, nivcsw; /* context switch counts */
    struct timespec start_time;        /* monotonic time */
    struct timespec real_start_time;   /* boot based time */
    unsigned long min_flt, maj_flt;
    struct task_cputime cputime_expires;
    struct list_head cpu_timers[3];

/* process credentials */
    uid_t uid,euid,suid,fsuid;
    gid_t gid,egid,sgid,fsgid;
    struct group_info *group_info;
    kernel_cap_t   cap_effective, cap_inheritable, cap_permitted, cap_bset;
    struct user_struct *user;
    unsigned securebits;
/* file system info */
```

```
    int link_count, total_link_count;
#ifdef CONFIG_SYSVIPC
/* ipc stuff */
    struct sysv_sem sysvsem;
#endif
#ifdef CONFIG_DETECT_SOFTLOCKUP
/* hung task detection */
    unsigned long last_switch_timestamp;
    unsigned long last_switch_count;
#endif
/* CPU-specific state of this task */
    struct thread_struct thread;
/* filesystem information */
    struct fs_struct *fs;
/* open file information */
    struct files_struct *files;
/* namespaces */
    struct nsproxy *nsproxy;
/* signal handlers */
    struct signal_struct *signal;
    struct sighand_struct *sighand;

    sigset_t blocked, real_blocked;
    sigset_t saved_sigmask;   /* restored if set_restore_sigmask() was used */
    struct sigpending pending;
    unsigned long sas_ss_sp;
    size_t sas_ss_size;
    int (*notifier)(void *priv);
    void *notifier_data;
    sigset_t *notifier_mask;
    struct audit_context *audit_context;
    seccomp_t seccomp;
/* Thread group tracking */
    u32 parent_exec_id;
    u32 self_exec_id;
/* Protection of (de-)allocation: mm, files, fs, tty, keyrings */
    spinlock_t alloc_lock;
    /* Protection of the PI data structures: */
    spinlock_t pi_lock;
/* journalling filesystem info */
    void *journal_info;
/* stacked block device info */
    struct bio *bio_list, **bio_tail;
    atomic_t fs_excl; /* holding fs exclusive resources */
    struct rcu_head rcu;
    unsigned long timer_slack_ns;
    unsigned long default_timer_slack_ns;

    struct list_head  *scm_work_list;
};
```

在这个结构中，对一个进程进行了全面描述。例如，每一个进程都有自己的进程号，该数字在系统中是唯一的，它存放在进程描述符的成员变量 pid 中：

```
pid_t pid;
```

同样，线程组号存放在成员变量 tgid 中：

```
pid_t tpid;
```

这个结构大致可以分为以下几类：

- 进程状态。记录进程是在等待、运行还是死锁。
- 调度信息。由哪个调度函数调度，怎样调度等。
- 进程的通信状况。
- 因为要插入进程树，所以必须有联系父子兄弟的指针。
- 时间信息。如计算好执行的时间，以便 CPU 分配。

- 标号。决定改进程归属。
- 可以读/写打开的一些文件信息。
- 进程上下文和内核上下文。
- 处理器上下文。
- 内存信息。

### 5.2.3 进程的状态与转换

进程对系统资源的竞争和进程之间的并发执行，使得一个进程在从创建到销毁的这个生命周期内要经历各种不同状态的变化。一个进程的各种状态在 task_struct 结构中通过成员变量 state、exit_state 记录。那么内核究竟为进程定义了几种状态？我们可以通过查看 sched.h 文件中定义的宏查找到答案。

```
#define TASK_RUNNING      0
#define TASK_INTERRUPTIBLE1
#define TASK_UNINTERRUPTIBLE   2
#define __TASK_STOPPED    4
#define __TASK_TRACED 8
/* in tsk->exit_state */
#define EXIT_ZOMBIE       16
#define EXIT_DEAD     32
/* in tsk->state again */
#define TASK_DEAD     64
#define TASK_WAKEKILL 128
```

其中各个状态的含义如下。

- TASK_RUNNING：表示进程正在运行，或者是进程获得了除处理器以外的全部资源，一旦获得处理器，即可运行。
- TASK_INTERRUPTIBLE：表示进程处于阻塞态，处于该状态的进程睡眠在相应资源的等待队列上，当该类资源再次有效时，系统会唤醒进程，并转换到运行态（TASK_RUNNING）。
- TASK_UNINTERRUPTIBLE：与 TASK_INTERRUPTIBLE 态相似，不同的是，该状态下的进程一直等待，直到所需要的资源有效或等待超时从而由系统唤醒。
- TASK_STOPPED：表示进程处于暂停状态，通过任务控制信号 SIGSTOP 可以将 TASK_RUNNING 状态的进程转换到此状态；在此状态下的进程收到 SIGCONT 信号后会转换到 TASK_RUNNING 状态。
- TASK_TRACED：表示该进程处于监控状态，用于程序的 BUG 调试。
- EXIT_ZOMBIE：表示进程处于僵尸状态，在这个状态下的进程只能转换到 EXIT_DEAD 状态。僵尸状态的产生是由于父进程死亡而被终止的进程，但是在 task 数据中仍然保留 task_struct 结构。
- EXIT_DEAD：表示父进程已经获得了该进程的记账信息，该进程可以被销毁。
- TASK_WAKEKILL：表示该进程已经退出且不需要父进程来回收。

**提示**

阻塞操作是指在执行设备操作时，如果没有获得资源，则进程挂起，直到满足可操作的条件再进行操作。非阻塞操作的进程在不能进行设备操作时，并不挂起。

在 Linux 的 2.6.25 内核中，引入了一种新的进程状态：TASK_KILLABLE，它用于将进程置为睡眠状态，可以替代有效但可能无法终止的 TASK_UNINTERRUPTIBLE 进程状态，以及易于唤醒但更加安全的 TASK_INTERRUPTIBLE 进程状态。但是在更高版本的内核中，Linux 又引入了 TASK_WAKEKILL 状态，随着这个状态的出现，内核又定义了几个便于设置状态的宏：

```
/* Convenience macros for the sake of set_task_state */
#define TASK_KILLABLE (TASK_WAKEKILL | TASK_UNINTERRUPTIBLE)
#define TASK_STOPPED  (TASK_WAKEKILL | __TASK_STOPPED)
#define TASK_TRACED   (TASK_WAKEKILL | __TASK_TRACED)

/* Convenience macros for the sake of wake_up */
```

```
#define TASK_NORMAL   (TASK_INTERRUPTIBLE | TASK_UNINTERRUPTIBLE)
#define TASK_ALL      (TASK_NORMAL | __TASK_STOPPED | __TASK_TRACED)

/* get_task_state() */
#define TASK_REPORT   (TASK_RUNNING | TASK_INTERRUPTIBLE | \
                       TASK_UNINTERRUPTIBLE | __TASK_STOPPED | \
                       __TASK_TRACED)
```

通过上面的代码，读者应该会更清楚 TASK_STOPPED 和__TASK_STOPPED 的区别了。Linux 系统通过一系列机制，每个进程的状态不断转换，从而实现多任务同时进行，其状态转换图如图 5.3 所示。

获得 CPU 而正在运行的进程若申请不到某个资源，则调用 sleep_on()或 interruptible_sleep_on()睡眠，其 task_struct 挂到相应的等待队列。如果调用 sleep_on()睡眠，则其状态变为 TASK_UNINTERRUPTIBLE。如果调用 interruptible_sleep_on()睡眠，则其状态变为 TASK_INTERRUPTIBLE。sleep_on()或 interruptible_sleep_on()将调用 schedule()函数把睡眠进程释放的 CPU 分配给 run-queue 队列的某个就绪进程。

状态为 TASK_INTERRUPTIBLE 的睡眠进程当申请的资源有效时被唤醒（如 wake_up_interruptible()），也可以由信号（signal）或定时中断唤醒。而状态为 TASK_UNINTERRUPTIBLE 的睡眠进程只有当它申请的资源有效时才能被唤醒（如 wake_up()），不能被信号（Signal）、定时中断唤醒。唤醒后，进程状态改为 TASK_RUNNING，并进入运行队列。

进程执行系统调用 sys_exit()或收到 SIG_KILL 信号而调用 do_exit()时，进程状态变为 TASK_ZOMBIE，释放所申请的资源，同时启动 schedule()把 CPU 分配给运行队列中其他就绪进程。若进程通过系统调用设置 PF_SYSTRACE，则在系统调用返回前，进入 ptrace()，状态变为 TASK_STOPPED，CPU 分配给运行队列中其他就绪进程。只有通过其他进程发送 SIG_KILL 或 SIG_CONT，才能把 TASK_STOPPED 进程唤醒，重新进入运行队列。

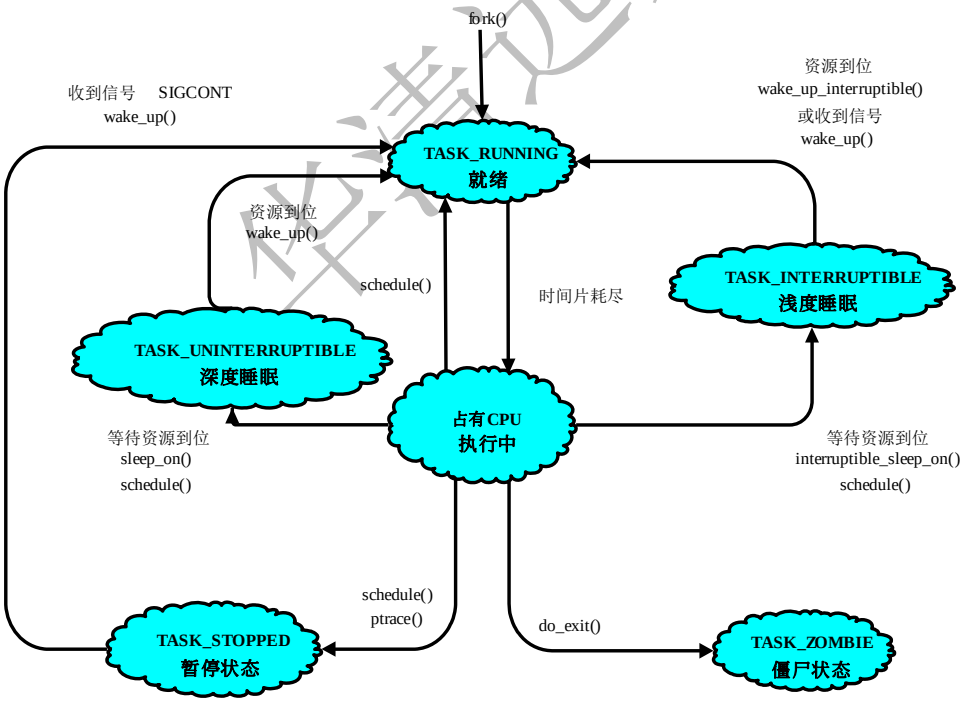


图 5.3 Linux 进程状态转换图

## 5.2.4 进程队列指针

为了有效地管理系统中的每个进程，需要采用一种高效的数据结构把系统内全部进程组织起来。当需要进程的信息时，可以从该结构中查找到相关进程。Linux 的所有进程组成一个双向链表，在内核中的类型是 struct list_head 的成员变量 tasks：

```
struct list_head tasks;
```

链表是一种常用的组织有序数据的数据结构，它通过指针将一系列数据节点连接成一条数据链，是线性表的一种重要实现方式。相对于数组，链表具有更好的动态性，建立链表时无须预先知道数据总量，可以随机分配空间，可以高效地在链表中的任意位置实时插入或删除数据。链表的开销主要是访问的顺序性和组织链的空间损失。

list_head 结构体在 include/linux/list.h 文件中定义，代码如下：

```
struct list_head {
    struct list_head *next, *prev;
};
```

include/linux/list.h 文件中提供了用来操作该链表的函数和宏，它们实现了对链表的插入、删除、转移、合并、遍历。这些函数和宏比较简单，这里不再详细分析。

```
static inline void list_add(struct list_head *new, struct list_head *head)
static inline void list_add_tail(struct list_head *new, struct list_head *head)
static inline void list_replace(struct list_head *old,struct list_head *new)
static inline void list_replace_init(struct list_head *old,struct list_head *new)
static inline void list_del_init(struct list_head *entry)
static inline void list_move(struct list_head *list, struct list_head *head)
static inline void list_move_tail(struct list_head *list,struct list_head *head)
static inline int list_is_last(const struct list_head *list,const struct list_head *head)
static inline int list_empty(const struct list_head *head)
static inline void list_splice(const struct list_head *list,struct list_head *head)
static inline void list_splice_tail(struct list_head *list,struct list_head *head)
static inline void list_splice_init(struct list_head *list,struct list_head *head)
static inline void list_splice_tail_init(struct list_head *list,struct list_head *head)

#define list_entry(ptr, type, member) \
    container_of(ptr, type, member)
#define list_first_entry(ptr, type, member) \
    list_entry((ptr)->next, type, member)
#define __list_for_each(pos, head) \
    for (pos = (head)->next; pos != (head); pos = pos->next)
#define list_for_each_prev(pos, head) \
    for (pos = (head)->prev; prefetch(pos->prev), pos != (head); \
         pos = pos->prev)
#define list_for_each_safe(pos, n, head) \
    for (pos = (head)->next, n = pos->next; pos != (head); \
         pos = n, n = pos->next)
#define list_for_each_prev_safe(pos, n, head) \
    for (pos = (head)->prev, n = pos->prev; \
         prefetch(pos->prev), pos != (head); \
         pos = n, n = pos->prev)
#define list_for_each_entry(pos, head, member)                \
    for (pos = list_entry((head)->next, typeof(*pos), member);    \
         prefetch(pos->member.next), &pos->member != (head);  \
         pos = list_entry(pos->member.next, typeof(*pos), member))
#define list_for_each_entry_reverse(pos, head, member)            \
    for (pos = list_entry((head)->prev, typeof(*pos), member);    \
         prefetch(pos->member.prev), &pos->member != (head);  \
         pos = list_entry(pos->member.prev, typeof(*pos), member))
#define list_prepare_entry(pos, head, member) \
    ((pos) ? : list_entry(head, typeof(*pos), member))
#define list_for_each_entry_continue(pos, head, member)       \
    for (pos = list_entry(pos->member.next, typeof(*pos), member);    \
         prefetch(pos->member.next), &pos->member != (head);  \
         pos = list_entry(pos->member.next, typeof(*pos), member))
#define list_for_each_entry_continue_reverse(pos, head, member)       \
    for (pos = list_entry(pos->member.prev, typeof(*pos), member);    \
         prefetch(pos->member.prev), &pos->member != (head);  \
         pos = list_entry(pos->member.prev, typeof(*pos), member))
#define list_for_each_entry_from(pos, head, member)           \
    for (; prefetch(pos->member.next), &pos->member != (head);    \
         pos = list_entry(pos->member.next, typeof(*pos), member))
#define list_for_each_entry_safe(pos, n, head, member)            \
```

```
        for (pos = list_entry((head)->next, typeof(*pos), member),    \
             n = list_entry(pos->member.next, typeof(*pos), member);  \
             &pos->member != (head);                       \
             pos = n, n = list_entry(n->member.next, typeof(*n), member))
    #define list_for_each_entry_safe_continue(pos, n, head, member)        \
        for (pos = list_entry(pos->member.next, typeof(*pos), member), \
             n = list_entry(pos->member.next, typeof(*pos), member);       \
             &pos->member != (head);                           \
             pos = n, n = list_entry(n->member.next, typeof(*n), member))
    #define list_for_each_entry_safe_from(pos, n, head, member)            \
        for (n = list_entry(pos->member.next, typeof(*pos), member);      \
             &pos->member != (head);                           \
             pos = n, n = list_entry(n->member.next, typeof(*n), member))
    #define list_for_each_entry_safe_reverse(pos, n, head, member)         \
        for (pos = list_entry((head)->prev, typeof(*pos), member),    \
             n = list_entry(pos->member.prev, typeof(*pos), member);  \
             &pos->member != (head);                       \
             pos = n, n = list_entry(n->member.prev, typeof(*n), member))
```

内核提供了宏 for_each_process，可以方便地搜索所有进程。当需要对系统中每一个进程进行操作时，该宏定义非常有用。它也在 sched.h 文件中定义，代码如下：

```
    #define next_task(p)  list_entry(rcu_dereference((p)->tasks.next), struct task_
struct, tasks)
    #define for_each_process(p) \
    for (p = &init_task ; (p = next_task(p)) != &init_task ; )
```

其中，init_task 是系统初始化过程中创建的第一个进程的进程描述符（PID），通过 list_entry()函数找到 task_struct 的地址，这样就可以打印进程的 PID 等域。

下面通过一个例子，进一步了解 list_head 结构的使用。下面的程序利用 list_head 结构遍历系统中的全部进程。

```
    static int hello_init(void)
    {
         struct task_struct *task,*p;
         struct list_head *pos;
         int count=0;
         printk("Hello World\n");
         task=&init_task;
         list_for_each(pos,&task->tasks)
                {
                p=list_entry(pos, struct task_struct, tasks);
                count++;
                printk("%d-->%s\n",p->pid,p->comm);
                }
         printk("Total process is:%d\n",count);
         return 0;
    }
    static void hello_exit(void)
    {
      printk( "Bye world!\n");
    }
    module_init(hello_init);
    module_exit(hello_exit);
```

### 5.2.5 进程队列的全局变量

在 Linux 中，内核使用 current 变量表示当前正在运行的进程。current 是一个 task_struct 类型的全局变量。下面的语句可以打印当前进程。

```
    printk( "Current task is %s [%d], current->comm, current->pid );
```

打开 arch/arm/include/asm/current.h 头文件，可以了解到 current 只是一个宏定义：

```
    static inline struct task_struct *get_current(void) __attribute_const__;
    static inline struct task_struct *get_current(void)
```

```
{
    return current_thread_info()->task;
}
#define current (get_current())
```

current_thread_info 函数用来获得当前进程的地址信息，代码及解释如下：

```
static inline struct thread_info *current_thread_info(void)
{
    register unsigned long sp asm ("sp");
    return (struct thread_info *)(sp & ~(THREAD_SIZE - 1));
}
```

在 ARM 平台中，内核栈是 8KB，在栈顶（最低地址）处存放了一个指向当前进程的 task_struct 的指针，而 sp 就是当前的内核态堆栈指针，把它和~8191 进行“与”操作（清空最后 13 位），即获得栈顶的地址。然后把里面的值赋给 current，这样 current 就得到了当前 task_struct 的指针。THREAD_SIZE 在 thread_info.h 文件中有定义如下：

```
#define THREAD_SIZE        8192
#define THREAD_START_SP    (THREAD_SIZE - 8)
```

thread_info.h 文件中定义的 thread_info 结构如下：

```
struct thread_info {
    struct task_struct    *task;        /* main task structure */
    struct exec_domain    *exec_domain;/* execution domain */
    unsigned long     flags;        /* low level flags */
    __u32         cpu;          /* current CPU */
    int           preempt_count;    /* 0 => preemptable,<0 => BUG */
    mm_segment_t      addr_limit;   /* thread address space:
                            0-0xBFFFFFFF for user
                            0-0xFFFFFFFF for kernel */
    struct restart_block    restart_block;
    struct thread_info    *real_thread;    /* Points to non-IRQ stack */
};
```

其实 thread_info 结构的第一个成员就是一个指向 task_struct 结构的指针，所以要用 current_thread_info()->task 表示 task_struct 的地址，但是整个过程对用户是完全透明的，我们还是可以用 current 表示当前进程。

## 5.3 Linux 进程的创建

一个进程不会平白无故地诞生，它总会有自己的“父母”。在 Linux 中，通过调用 fork 系统调用来创建一个新的进程。新创建的子进程同样也能执行 fork，所以，有可能形成一颗完整的进程树。注意，每个进程只有一个父进程，但可以有 0 个、1 个、2 个或多个子进程。图 5.4 描述了 Linux 进程层次结构。其中，Pa 和 Pb 是进程 P 的两个子进程，而 Pc 和 Pd 又是进程 Pa 的两个子进程。

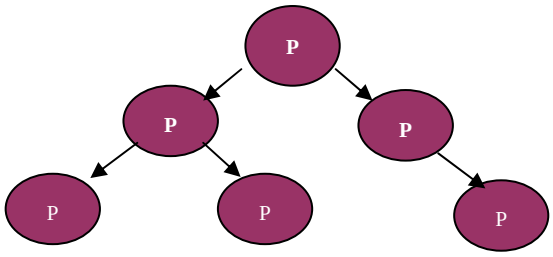


图 5.4　Linux 进程层次结构

Linux 在启动时就创建一个称为 init 的特殊进程，其进程标识符（PID）为 1，它是用户态下所有进程的祖先进程，以后诞生的所有进程都是它的子进程——或是它的儿子，或是它的孙子。1 号进程运行时查

询系统当前存在的终端数，然后为每个终端创建一个新的管理进程，这些进程在终端上等待着用户的登录。当用户正确登录后，系统再为每一个用户启动一个 Shell 进程，由 Shell 进程等待并接受用户输入的命令。

应用程序可以通过 fork、vfork 或 clone 函数建立新的用户线程，这些函数分别通过系统调用访问 sys_fork、sys_vfork，或 sys_clone 内核函数建立新线程，而 sys_fork、sys_vfork 与 sys_clone 共同的调用函数为 do_fork。sys_fork 和 sys_vfork 在 arch/um/kernel/syscall.c 文件中实现，sys_clone 函数在 arch/um/sys-i386/syscall.c 文件中实现，代码如下：

```
long sys_fork(void)
{
    long ret;

    current->thread.forking = 1;
    ret = do_fork(SIGCHLD, UPT_SP(&current->thread.regs.regs),
            &current->thread.regs, 0, NULL, NULL);
    current->thread.forking = 0;
    return ret;
}

long sys_vfork(void)
{
    long ret;

    current->thread.forking = 1;
    ret = do_fork(CLONE_VFORK | CLONE_VM | SIGCHLD,
            UPT_SP(&current->thread.regs.regs),
            &current->thread.regs, 0, NULL, NULL);
    current->thread.forking = 0;
    return ret;
}

long sys_clone(unsigned long clone_flags, unsigned long newsp,
          int __user *parent_tid, void *newtls, int __user *child_tid)
{
    long ret;

    if (!newsp)
        newsp = UPT_SP(&current->thread.regs.regs);

    current->thread.forking = 1;
    ret = do_fork(clone_flags, newsp, &current->thread.regs, 0, parent_tid,
            child_tid);
    current->thread.forking = 0;
    return ret;
}
```

Linux 创建进程的函数的层次结构如图 5.5 所示。

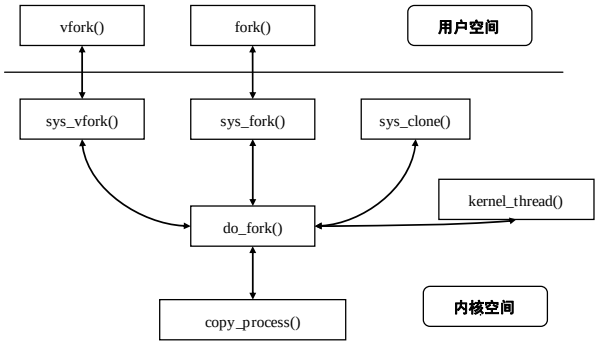


图 5.5　创建进程的函数的层次结构

从图 5.5 中可以看到，do_fork 函数是进程创建的基础，处理 clone、fork、vfork 系统调用。do_fork 使用辅助函数 copy_process 建立进程描述符及内核数据结构。可以在 kernel/fork.c 文件内找到 do_fork 函数原型（copy_process 函数也在该文件中）。

```
long do_fork(unsigned long clone_flags, //克隆标识
          unsigned long stack_start,          //栈起始位置
           //指向通用寄存器值的指针，通用寄存器的值是在用户态切换到内核态时保存到内核态堆栈
          struct pt_regs *regs,
          unsigned long stack_size,           //栈大小，一般设置为 0
          int __user *parent_tidptr,          //父进程用户态变量地址
          int __user *child_tidptr)           //子进程用户态变量地址
```

do_fork 函数中比较重要的有两个部分，首先是给子进程分配进程号，然后利用具体的进程号创建子进程。下面对这个函数进行简要分析。

```
{
    struct task_struct *p;          //进程描述符
    int trace = 0;
    long nr;
    if (unlikely(clone_flags & CLONE_STOPPED))
{
        static int __read_mostly count = 100;
        if (count > 0 && printk_ratelimit()) {
            char comm[TASK_COMM_LEN];
            count--;
            printk(KERN_INFO "fork(): process `%s' used deprecated "
                    "clone flags 0x%lx\n",
                get_task_comm(comm, current),
                clone_flags & CLONE_STOPPED);
        }
    }
```

在 2.6 内核中随处可以看到 likely()和 unlikely()宏，这两个宏在内核中定义如下：

```
#define likely(x)      __builtin_expect(!!(x), 1)
#define unlikely(x)        __builtin_expect(!!(x), 0)
```

__builtin_expect() 是 GCC (version>=2.96）提供给程序员使用的，目的是将“分支转移”的信息提供给编译器，这样，编译器就可以对代码进行优化，以减少指令跳转带来的性能下降。

__builtin_expect((x),1)表示 x 的值为真的可能性更大。

__builtin_expect((x),0)表示 x 的值为假的可能性更大。

也就是说，使用 likely()，执行 if 后面的语句的机会更大；使用 unlikely()，执行 else 后面的语句的机会更大。

回到 do_fork 函数中，CLONE_STOPPED 位表示设置进程为停止状态（类似的 clone 标志位在 sched.h 中定义），因此，为假的可能性更大。随后内核会通过 get_task_comm 函数获得当前进程（current）的命令名，显示给用户。

```
        if (likely(user_mode(regs)))
        trace = tracehook_prepare_clone(clone_flags);
```

上面的代码检查子进程是否为内核线程，如果不是，则将 clone_flags 设为 0。

```
p = copy_process(clone_flags, stack_start, regs, stack_size,
     child_tidptr, NULL, trace);
```

随后内核调用 copy_process 函数复制进程描述符。如果成功，该函数将返回刚创建的 task_struct 描述符的地址。

```
    if (!IS_ERR(p)) {
        struct completion vfork;   //completion 同步机制
        trace_sched_process_fork(current, p);
        nr = task_pid_vnr(p);      //获得 PID
        if (clone_flags & CLONE_PARENT_SETTID)
            put_user(nr, parent_tidptr);
        if (clone_flags & CLONE_VFORK) {
```

```
            p->vfork_done = &vfork;
            init_completion(&vfork);
        }
        audit_finish_fork(p);
        tracehook_report_clone(trace, regs, clone_flags, nr, p);
```

上面的语句判断是否设置 CLONE_VFORK 标志，如果设置，则初始化进程描述符的 vfork_done 标志，然后初始化 completion。

```
        p->flags &= ~PF_STARTING;
        if (unlikely(clone_flags & CLONE_STOPPED)) {
            /*
             * We'll start up with an immediate SIGSTOP.
             */
            sigaddset(&p->pending.signal, SIGSTOP);
            set_tsk_thread_flag(p, TIF_SIGPENDING);
            __set_task_state(p, TASK_STOPPED);
        } else {
            wake_up_new_task(p, clone_flags);
        }
        tracehook_report_clone_complete(trace, regs,clone_flags, nr, p);
```

如果子进程被跟踪或子进程初始化成 STOP 状态，则发送 SIGSTOP 信号。由于子进程现在还没有运行，信号不能被处理，所以设置 TIF_SIGPENDING 标志；如果设置了 CLONE_STOPPED 标志或必须跟踪子进程，则将子进程的状态设置为 TASK_STOPPED，并为子进程增加挂起的 SIGSTOP 信号；如果没有设置 CLONE_STOPPED 标志，则调用 wake_up_new_task 函数来处理父进程和子进程。

```
        if (clone_flags & CLONE_VFORK) {
            freezer_do_not_count();
            wait_for_completion(&vfork);
            freezer_count();
            tracehook_report_vfork_done(p, nr);
        }
        else {
        nr = PTR_ERR(p);
        }
        return nr;
```

如果设置了 CLONE_VFORK 标志，则挂起父进程直到子进程释放自己的内存地址空间，也就是说，直到子进程结束或执行新的程序。wait_for_completion(&vfork)用来等待子进程释放自己的内存地址空间。

## 5.4 Linux 进程相关的系统调用

由于 Linux 以分裂的方法来创建新进程，这样就使得系统中的所有进程都有亲属关系，这种亲属关系也会给进程的运行带来一定的影响。为了表示一个进程的亲属关系，每个进程控制块中都有记录这些亲属关系的成员，代码如下所示：

```
struct task_struct {
…
struct task_struct *real_parent;   /* 父进程 */
    struct task_struct *parent;    /* 养父进程*/
    /*
     * children/sibling forms the list of my natural children
     */
    struct list_head children;     /* 子进程列表 */
    struct list_head sibling;      /* linkage in my parent's children list */
…
};
```

从内核代码中可以看到，进程有父进程和养父进程。那么养父进程是什么呢？

当一个进程被创建出来以后，其控制块中的指针 real_parent 和 parent 都指向同一个进程→父进程。但是当其他某个进程通过系统调用 ptrace()跟踪这个进程时，被跟踪进程的 parent 会指向这个跟踪进程，而这个跟踪进程也有指针指向被跟踪进程，以便通过这个指针得到被跟踪进程的控制块以获得被跟踪进程的信息，所以这个跟踪进程有些类似“监管人”，因此这个进程也称为“养父”进程。

### 5.4.1 execve()系统调用

Linux 提供了 execl、execlp、execle、execv、execvp 和 execve 共 6 个用以执行一个可执行文件的函数（统称为 exec 函数）。这些函数的不同之处在于对命令行参数和环境变量参数的传递方式不同。每个函数的第一个参数都是要被执行的程序的路径，第二个参数则向程序传递了命令行参数，第三个参数则向程序传递环境变量。以上函数的本质都是调用 arch/x86/kernel/process.c 文件中实现的系统调用 sys_execve 来执行一个可执行文件，该函数代码如下：

```
    long sys_execve(char __user *name, char __user * __user *argv,char __user * __user *envp, struct
pt_regs *regs)
    {
        long error;
        char *filename;

        filename = getname(name);
        error = PTR_ERR(filename);
        if (IS_ERR(filename))
            return error;
        error = do_execve(filename, argv, envp, regs);

    #ifdef CONFIG_X86_32
        if (error == 0) {
            /* Make sure we don't return using sysenter.. */
                set_thread_flag(TIF_IRET);
          }
    #endif

        putname(filename);
        return error;
    }
```

从函数原型提供的参数可以看出，这个函数接收的参数都来自用户空间。在 sys_execve 中调用了 execve 函数，该函数也在 arch/um/kernel/exec.c 中实现。从下面的代码中可以了解到，execve 实质上调用了 do_execve 函数。

```
    int do_execve(char * filename,
        char __user *__user *argv,
        char __user *__user *envp,
        struct pt_regs * regs)
    {
        struct linux_binprm *bprm;
        struct file *file;
        struct files_struct *displaced;
        bool clear_in_exec;
        int retval;

        retval = unshare_files(&displaced);
        if (retval)
            goto out_ret;

        retval = -ENOMEM;
        bprm = kzalloc(sizeof(*bprm), GFP_KERNEL);
        if (!bprm)
            goto out_files;

        retval = prepare_bprm_creds(bprm);
        if (retval)
            goto out_free;
```

```
    retval = check_unsafe_exec(bprm);
    if (retval < 0)
        goto out_free;
    clear_in_exec = retval;
    current->in_execve = 1;

    file = open_exec(filename);
    retval = PTR_ERR(file);
    if (IS_ERR(file))
        goto out_unmark;

    sched_exec();

    bprm->file = file;
    bprm->filename = filename;
    bprm->interp = filename;

    retval = bprm_mm_init(bprm);
    if (retval)
        goto out_file;

    bprm->argc = count(argv, MAX_ARG_STRINGS);
    if ((retval = bprm->argc) < 0)
        goto out;

    bprm->envc = count(envp, MAX_ARG_STRINGS);
    if ((retval = bprm->envc) < 0)
        goto out;

    retval = prepare_binprm(bprm);
    if (retval < 0)
        goto out;

    retval = copy_strings_kernel(1, &bprm->filename, bprm);
    if (retval < 0)
        goto out;

    bprm->exec = bprm->p;
    retval = copy_strings(bprm->envc, envp, bprm);
    if (retval < 0)
        goto out;

    retval = copy_strings(bprm->argc, argv, bprm);
    if (retval < 0)
        goto out;

    current->flags &= ~PF_KTHREAD;
    retval = search_binary_handler(bprm,regs);
    if (retval < 0)
        goto out;

    /* execve succeeded */
    current->fs->in_exec = 0;
    current->in_execve = 0;
    acct_update_integrals(current);
    free_bprm(bprm);
    if (displaced)
        put_files_struct(displaced);
    return retval;

out:
    if (bprm->mm)
        mmput (bprm->mm);

out_file:
    if (bprm->file) {
```

```
            allow_write_access(bprm->file);
            fput(bprm->file);
        }

    out_unmark:
        if (clear_in_exec)
            current->fs->in_exec = 0;
        current->in_execve = 0;

    out_free:
        free_bprm(bprm);

    out_files:
        if (displaced)
            reset_files_struct(displaced);
    out_ret:
        return retval;
    }
```

上述代码中有两个函数：copy_strings_kernel 和 copy_strings。copy_strings 把参数及执行的环境从用户空间复制到内核空间的 bprm 变量中，而调用 copy_strings_kernel()从内核空间中复制文件名。

这里还提到了 linux_binfmt 数据结构（/include/linux/binfmt.h），它用来支持各种文件系统，所以 Linux 中的 exec()函数在执行时，使用已注册的 linux_binfmt 结构就可以支持不同的二进制格式，即多种文件系统（Ext3、FAT 等）。需要指出的是，binux_binfmt 结构中嵌入了两个指向函数的指针，一个指针指向可执行代码，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 结构描述如下：

```
    struct linux_binprm{
        char buf[BINPRM_BUF_SIZE];
    #ifdef CONFIG_MMU
        struct vm_area_struct *vma;
    #else
    # define MAX_ARG_PAGES    32
        struct page *page[MAX_ARG_PAGES];
    #endif
        struct mm_struct *mm;
        unsigned long p; /* current top of mem */
        unsigned int sh_bang:1,
                 misc_bang:1;
    #ifdef __alpha__
        unsigned int taso:1;
    #endif
        unsigned int recursion_depth;
        struct file * file;
        int e_uid, e_gid;
        kernel_cap_t cap_post_exec_permitted;
        bool cap_effective;
        void *security;
        int argc, envc;
        char * filename;      /* Name of binary as seen by procps */
        char * interp;        /* Name of the binary really executed. Most
                        of the time same as filename, but could be
                        different for binfmt_{misc,script} */
        unsigned interp_flags;
        unsigned interp_data;
        unsigned long loader, exec;
    };
```

在 do_execve 函数中有一个参数值得我们注意——*regs。它是 pt_regs 类型的数据结构，该参数描述了在执行该系统调用时，用户态下的 CPU 寄存器在核心态的栈中的保存情况。通过这个参数，sys_execve 能获得保存在用户空间的以下信息：可执行文件路径的指针（regs.ebx 中）、命令行参数的指针（regs.ecx 中）和环境变量的指针（regs.edx 中）。代码如下：

```
    struct pt_regs {
```

```
        long ebx;
        long ecx;
        long edx;
        long esi;
        long edi;
        long ebp;
        long eax;
        int  xds;
        int  xes;
        int  xfs;
        /* int  gs; */
        long orig_eax;
        long eip;
        int  xcs;
        long eflags;
        long esp;
        int  xss;
    };
```

回顾 do_execve()，它的工作流程如下：

（1）按参数 filename 找到相关文件的节点，保存与文件相关的数据，并检查它的存取权限。

（2）检查文件格式，并找到能打开这种格式的装载函数。

（3）确认文件的可执行权限后，首先放弃子进程继承自父进程的虚拟空间结构，调用 do_mmap()函数，根据文件中的虚拟地址信息建立自己的虚拟空间描述结构，建立空页表，并使其映射到子进程的进程虚拟地址空间。

（4）创建运行环境。

（5）在子进程运行过程中，由 Linux 的请页机制逐步为其分配所需要的物理内存空间。

下面看一个运行子进程的程序。

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main()
{
        char * argu_list[]={"ls","/","-l",NULL};
        int status=0;
        pid_t child_pid=fork();
        if(child_pid==0)
        {
                //子进程
                execvp("ls",argu_list);
                fprintf(stderr,"exec ls error\n");
                exit(1);
        }
        else
        {
                wait(&status);
                if(WIFEXITED(status))
                        printf("the child process exit normally.\n");
                else
                        printf("the child process exit abnormally.\n");
                exit(0);
        }
}
```

## 5.4.2 wait()系统调用

为了方便用户处理父进程与子进程之间的一些事物，Linux 允许父进程在创建了进程之后，通过调用 wait()先进入等待状态，以使子进程先运行，然后再决定自己的进一步行为，这种方式称为父进程的阻塞方式。那么，wait()在什么时候用呢？在一些情况下，父进程可能比子进程结束得要早。如果父进程提前结束

了，子进程就变成了僵尸进程。我们需要父进程来清理子进程结束后的一些环境。这时调用 wait，父进程将阻塞在 wait()处，等待子进程结束。

### 5.4.3 exit()系统调用

进程的结束可以用 exit()或_exit()系统调用。无论在程序中的什么位置，只要执行到 exit 系统调用，进程就会停止剩下的所有操作，清除包括 PCB 在内的各种数据结构，并终止本进程的运行。对系统调用而言，_exit 和 exit 是一对孪生兄弟，它们最大的区别就在于 exit()函数在调用之前要检查文件的打开情况，把文件缓冲区中的内容写回文件；而_exit()直接使进程停止运行，清除其使用的内存空间，并销毁其在内核中的各种数据结构，如图 5.6 所示。

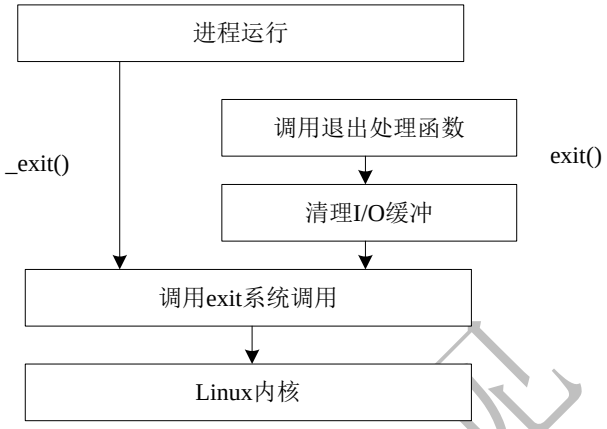


图 5.6 exit 和_exit 的区别

下面通过一个简单的程序了解一下 exit 系统调用的用法。

```
main()
{
printf("Process will be exit!\n");
exit(0);
printf("You cannot see this line!\n");
}
```

编译运行该程序可以看到，程序并没有打印后面的“You cannot see this line!”，因为在此之前，在执行到 exit(0)时，进程就已经终止了。

## 5.5 Linux 的进程调度

操作系统的职能之一是管理并调度系统中的进程，而衡量一个操作系统的关键之一就是调度算法。一个优秀的调度算法能够充分而高效地利用系统资源。存在于 Linux 中的进程通过 Linux 调度程序被调度。每个进程的调度策略在 task_struct 中规定（policy 属性），Linux 内核提供的调度策略类型如表 5.4 所示。其中，SCHED_FIFO 和 SCHED_RR 的优先级高于所有 SCHED_NORMAL 的进程。

> 提示 什么时候以何种方式选择一个进程运行的这组规则就是调度策略。

表 5.4 Linux 进程调度策略

| 调度策略 | 采用的调度算法 |
| --- | --- |
| SCHED_NORMAL | 非实时，基于优先级的轮转法 |

| | |
|---|---|
| SCHED_FIFO | 实时，先进先出算法 |
| SCHED_RR | 实时，基于优先级的轮转法 |
| SCHED_BATCH | 与 SCHED_NORMAL 类似，该调度算法将使得调度器假定进程对 CPU 时间要求较多 |
| SCHED_ISO | 目前还没有实现 |
| SCHED_IDLE | 进程优先级降为最低 |

为了实现进程的调度，每一个多任务操作系统都会为不同的任务分配一个任务优先级，进程之间是竞争资源（如 CPU 和内存的占用）关系，这个竞争优劣是通过一个数值来实现的，也就是谦让度。高谦让度表示进程优先级别最低，负值或 0 表示高优先级，对其他进程不谦让，也就是拥有优先占用系统资源的权力。谦让度的值从-20 到 19。

目前，硬件技术发展迅速，大多情况下不必设置进程的优先级，除非在进程失控而疯狂占用资源的情况下，才设置优先级。在迫不得已的情况下，可以杀掉失控进程。Linux 系统提供了 nice 命令来完成进程优先级调整。例如，下面的命令将运行 vim 程序，并为它指定谦让度增量为 6：

```
[root@localhost ~]# nice -n 6 vim
```

nice 的最常用的应用包括：

```
nice -n 谦让度的增量值 程序
renice 是通过进程 ID（PID）来改变谦让度，进而达到更改进程的优先级。
renice 谦让度 PID
renice 所设置的谦让度就是进程的绝对值。
```

2.6 版本的 Linux 内核提供了 140 个优先级，后 40 个和 nice 值一一对应，属于 SCHED_NORMAL 调度策略，前 100 个属于 SCHED_FIFO 和 SCHED_RR 策略。

在调度算法的实现上，Linux 中的每个任务有 4 个与调度相关的参数，它们是 rt_priority、policy、priority（nice）、counter。调度程序根据这 4 个参数进行进程调度。

在 SCHED_NORMAL 调度策略中，调度器总是选择那个 priority+counter 值最大的进程来调度执行。从逻辑上进行分析，SCHED_NORMAL 调度策略存在着调度周期（epoch），在每一个调度周期中，一个进程的 priority 和 counter 值的大小影响了当前时刻应该调度哪一个进程来执行，其中 priority 是一个固定不变的值，在进程创建时就已经确定了，它代表该进程的优先级，也代表该进程在每一个调度周期中能够得到的时间片的多少；counter 是一个动态变化的值，它反映了一个进程在当前的调度周期中还剩下的时间片。在每一个调度周期的开始，priority 的值被赋给 counter，然后每次该进程被调度执行时，counter 值都减少。当 counter 值为零时，该进程用完自己在本调度周期中的时间片，不再参与本调度周期的进程调度。当所有进程的时间片都用完时，一个调度周期结束，这样周而复始。另外，可以看出 Linux 系统中的调度周期不是静态的，它是一个动态变化的量，例如，处于可运行状态的进程的多少和它们 priority 的值都可以影响一个 epoch 的长短。在 2.4 以上的内核中，priority 被 nice 所取代，但二者作用类似。

在 SCHED_FIFO 调度策略中，不同的进程根据静态优先级进行排队；然后在同一优先级的队列中，谁先准备好运行就先调度谁，并且正在运行的进程不会被终止，直到以下情况发生：

- 被更高优先级的进程强占 CPU。
- 自己因为资源请求而阻塞。
- 自己主动放弃 CPU（调用 sched_yield）。

SCHED_RR 与上面的 SCHED_FIFO 类似，不同之处在于它给每个进程分配一个时间片，时间片到了正在执行的进程就放弃执行；时间片的长度可以通过 sched_rr_get_interval 调用得到。

## 5.6 实时 Linux

Linux 是一种非实时操作系统，在某些对实时性要求比较严格的嵌入式应用中，Linux 系统的实时性一直是最大的顽症。不过，也有人陆续推出 Linux 内核的实时补丁，在一定程度上解决了实时性的问题。其中，最有名的是 RT-Linux。

RT-Linux 是新墨西哥科技大学的研究成果，它的基本思想是，为了在 Linux 系统中提供对于硬实时的支持，实现了一个微内核的小的实时操作系统，而将普通 Linux 系统作为一个该操作系统中的一个低优先级的任务来运行。另外，普通 Linux 系统中的任务可以通过 FIFO 和实时任务进行通信。RT-Linux 的框架如图 5.7 所示。


图 5.7　RT-Linux 结构

RT-Linux 的关键技术是通过软件来模拟硬件的中断控制器。当 Linux 系统要封锁 CPU 的中断时，RT-Linux 中的实时子系统会截取到这个请求，把它记录下来，而实际上并不真正封锁硬件中断，这样就避免了由于封锁中断所造成的系统在一段时间没有响应的情况，从而提高了实时性。当有硬件中断到来时，RT-Linux 截取该中断，并判断是否有实时子系统中的中断例程来处理，还是传递给普通的 Linux 内核进行处理。另外，普通 Linux 系统中的最小定时精度由系统中的实时时钟的频率决定，一般 Linux 系统将该时钟设置为每秒 100 个时钟中断，所以 Linux 系统中一般的定时精度为 10ms，即时钟周期是 10ms，而 RT-Linux 通过将系统的实时时钟设置为单次触发状态，可以提供十几个微秒级的调度粒度。

RED-Linux 是另外一种实时 Linux 系统，由加州大学 Irvine 分校开发。它将对实时调度的支持和 Linux 很好地实现在同一个操作系统内核中。它同时支持 3 种类型的调度算法，即：Time-Driven、Priority-Driven、Share-Driven。RED-Linux 的设计目标就是提供一个可以支持各种调度算法的通用调度框架，该系统给每个任务增加了如下几项属性，并将它们作为进程调度的依据。

- Priority：作业的优先级。
- Start-Time：作业的开始时间。
- Finish-Time：作业的结束时间。
- Budget：作业在运行期间所要使用的资源数量。

通过调整这些属性的取值及调度程序按照什么样的优先顺序来使用这些属性值，几乎可以实现所有的调度算法。这样，可以将 3 种不同的调度算法无缝、统一地结合到一起。RED-Linux 调度程序的框架结构如图 5.8 所示。

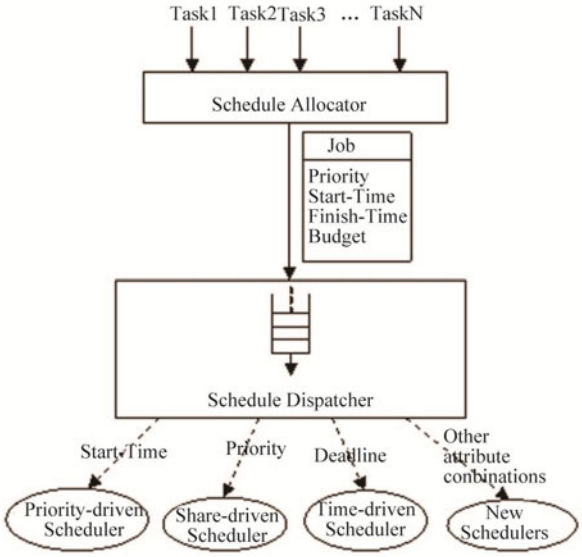


图 5.8　RED-Linux 调度框架

## 5.7 本章习题

1. 和进程管理相关的内核文件都有哪些？
2. 什么是进程和线程？
3. 什么是进程描述符？怎样得到当前进程的进程描述符？
4. 进程的内核栈有多大？
5. 进程的状态都有哪些？在什么情况下发生转化？
6. Linux 中所有进程之间的关系是怎么样的？



## 联系方式