进程管理

Task_struct详解

进程状态

```
/*
 * Task state bitmask. NOTE! These bits are also
 * encoded in fs/proc/array.c: get_task_state().
 *
 * We have two separate sets of flags: task->state
 * is about runnability, while task->exit_state are
 * about the task exiting. Confusing, but this way
 * modifying one set can't modify the other one by
 * mistake.
 */
#define TASK_RUNNING            0
#define TASK_INTERRUPTIBLE      1
#define TASK_UNINTERRUPTIBLE    2
#define __TASK_STOPPED          4
#define __TASK_TRACED           8

/* in tsk->exit_state */
#define EXIT_DEAD               16
#define EXIT_ZOMBIE             32
#define EXIT_TRACE              (EXIT_ZOMBIE | EXIT_DEAD)

/* in tsk->state again */
#define TASK_DEAD               64
#define TASK_WAKEKILL           128     /** wake on signals that are deadly **/
#define TASK_WAKING             256
#define TASK_PARKED             512
#define TASK_NOLOAD             1024
#define TASK_STATE_MAX          2048

/* 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)
```

5个互斥状态

	A	B
1	状态	描述
2	TASK_RUNNING	表示进程要么正在执行，要么正要准备执行（已经就绪），正在等待cpu时间片的调度
3	TASK_INTERRUPTIBLE	进程因为等待一些条件而被挂起（阻塞）而所处的状态。这些条件主要包括：硬中断、资源、一些信号……，一旦等待的条件成立，进程就会从该状态（阻塞）迅速转化成为就绪状态TASK_RUNNING
4	TASK_UNINTERRUPTIBLE	意义与TASK_INTERRUPTIBLE类似，除了不能通过接受一个信号来唤醒以外，对于处于TASK_UNINTERRUPIBLE状态的进程，哪怕我们传递一个信号或者有一个外部中断都不能唤醒他们。只有它所等待的资源可用的时候，他才会被唤醒。这个标志很少用，但是并不代表没有任何用处，其实他的作用非常大，特别是对于驱动刺探相关的硬件过程很重要，这个刺探过程不能被一些其他的东西给中断，否则就会让进城进入不可预测的状态
5	TASK_STOPPED	进程被停止执行，当进程接收到SIGSTOP、SIGTTIN、SIGTSTP或者SIGTTOU信号之后就会进入该状态
6	TASK_TRACED	表示进程被debugger等进程监视，进程执行被调试程序所停止，当一个进程被另外的进程所监视，每一个信号都会让进城进入该状态

2个终止状态

其实还有两个附加的进程状态既可以被添加到state域中，又可以被添加到exit_state域中。只有当进程终止的时候，才会达到这两种状态.‘

```
1 /* task state */
2 int exit_state;
3 int exit_code, exit_signal;
```

	A	B
1	状态	描述
2	EXIT_ZOMBIE	进程的执行被终止，但是其父进程还没有使用wait()等系统调用来获知它的终止信息，此时进程成为僵尸进程
3	EXIT_DEAD	进程的最终状态

而`int exit_code`，`exit_signal`;我们会在后面进程介绍

新增睡眠状态

如前所述，进程状态 TASK_UNINTERRUPTIBLE 和 TASK_INTERRUPTIBLE 都是睡眠状态。现在，我们来看看内核如何将进程置为睡眠状态。

内核如何将进程置为睡眠状态

Linux 内核提供了两种方法将进程置为睡眠状态。

将进程置为睡眠状态的普通方法是将进程状态设置为 TASK_INTERRUPTIBLE 或 TASK_UNINTERRUPTIBLE 并调用调度程序的 schedule() 函数。这样会将进程从 CPU 运行队列中移除。

- 如果进程处于可中断模式的睡眠状态（通过将其状态设置为 TASK_INTERRUPTIBLE），那么可以通过显式的唤醒呼叫（wakeup_process()）或需要处理的信号来唤醒它。
- 但是，如果进程处于非可中断模式的睡眠状态（通过将其状态设置为 TASK_UNINTERRUPTIBLE），那么只能通过显式的唤醒呼叫将其唤醒。除非万不得已，否则我们建议您将进程置为可中断睡眠模式，而不是不可中断睡眠模式（比如说在设备 I/O 期间，处理信号非常困难时）。

当处于可中断睡眠模式的任务接收到信号时，它需要处理该信号（除非它已被屏蔽），离开之前正在处理的任务（此处需要清除代码），并将 -EINTR 返回给用户空间。再一次，检查这些返回代码和采取适当操作的工作将由程序员完成。

因此，懒惰的程序员可能比较喜欢将进程置为不可中断模式的睡眠状态，因为信号不会唤醒这类任务。

但需要注意的一种情况是，对不可中断睡眠模式的进程的唤醒呼叫可能会由于某些原因不会发生，这会使进程无法被终止，从而最终引发问题，因为惟一的解决方法就是重启系统。一方面，您需要考虑一些细节，因为不这样做会在内核端和用户端引入 bug。另一方面，您可能会生成永远不会停止的进程（被阻塞且无法终止的进程）。

现在，我们在内核中实现了一种新的睡眠方法

Linux Kernel 2.6.25 引入了一种新的进程睡眠状态，

	A	B
1	状态	描述
2	TASK_KILLABLE	当进程处于这种可以终止的新睡眠状态中，它的运行原理类似于 TASK_UNINTERRUPTIBLE，只不过可以响应致命信号

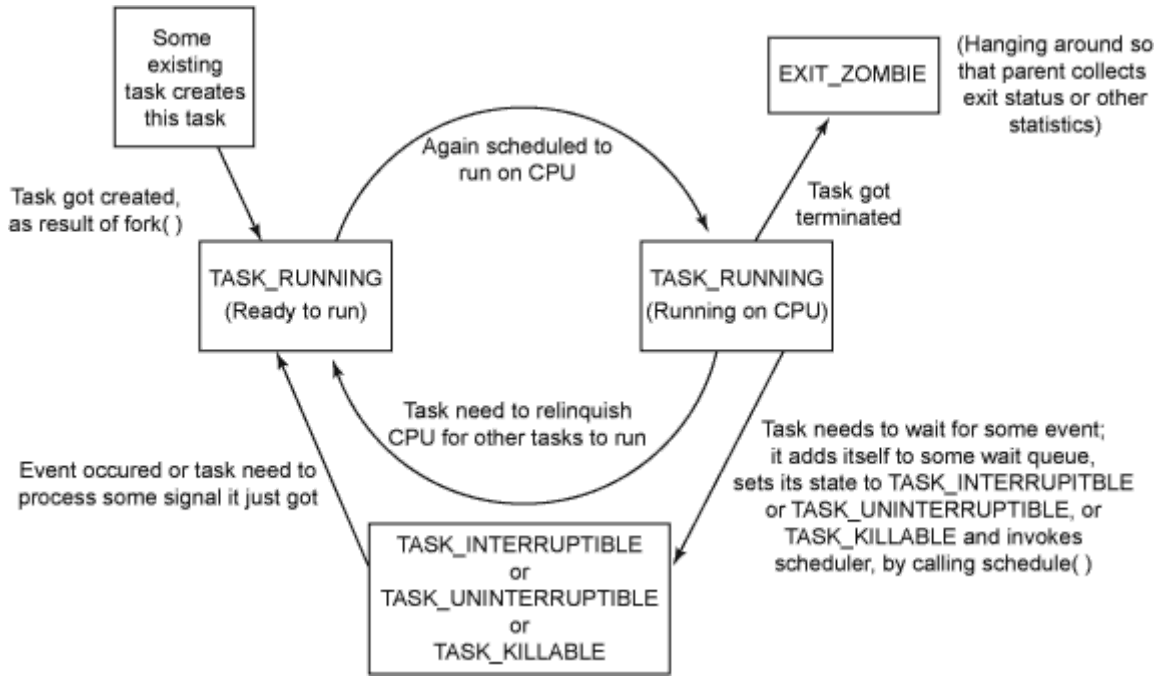
```
#define TASK_WAKEKILL           128 /** wake on signals that are deadly **/

/* 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)
```

换句话说，TASK_UNINTERRUPTIBLE + TASK_WAKEKILL = TASK_KILLABLE。

而TASK_WAKEKILL 用于在接收到致命信号时唤醒进程

新的睡眠状态允许 TASK_UNINTERRUPTIBLE 响应致命信号

进程状态的切换过程和原因大致如下图



进程标识符（PID）

pid_t pid;
pid_t tgid;

Unix系统通过pid来标识进程，linux把不同的pid与系统中每个进程或轻量级线程关联，而unix程序员希望同一组线程具有共同的pid，遵照这个标准linux引入线程组的概念。一个线程组所有线程与领头线程具有相同的pid，存入tgid字段，getpid()返回当前进程的tgid值而不是pid的值。

在Linux系统中，一个线程组中的所有线程使用和该线程组的领头线程（该组中的第一个轻量级进程）相同的PID，并被存放在tgid成员中。只有线程组的领头线程的pid成员才会被设置为与tgid相同的值。注意，getpid()系统调用返回的是当前进程的tgid值而不是pid值。

进程内核栈

```
1  void *stack;
```

ThreadInfo结构和内核栈的两种关系

ThreadInfo结构在内核栈中

Threadinfo结构存储在内核栈中，这种方式是最经典的。因为task_struct结构从1.0到现在5.0内核此结构一直在增大。如果将此结构放在内核栈中则很浪费内核栈的空间，则在threadinfo结构中有一个task_struct的指针就可以避免。

可以看到thread_info结构中存在一个struct task_struct的指针。

```
1  struct thread_info {
2      unsigned long        flags;        /* low level flags */
```

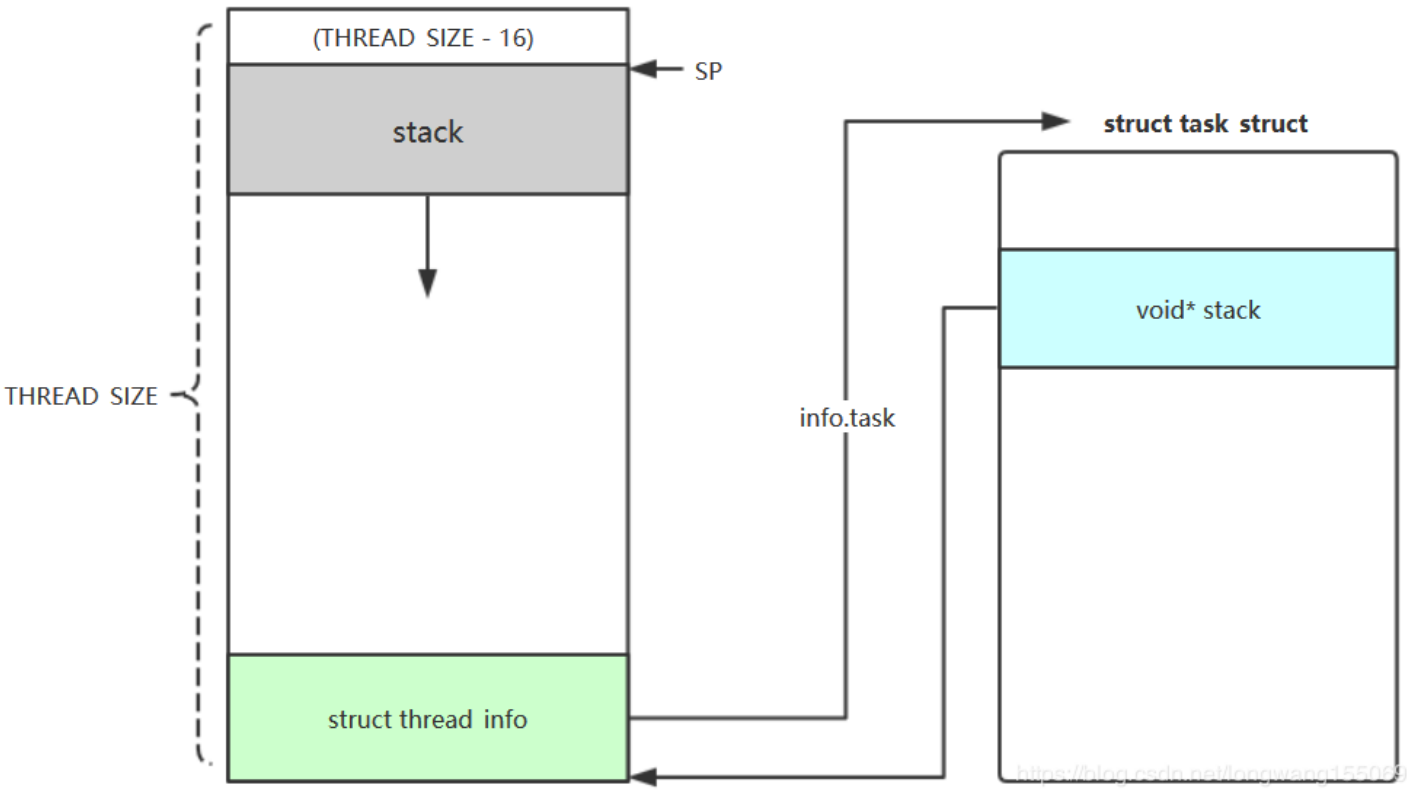
```
    mm_segment_t        addr_limit;     /* address limit */
    struct task_struct    *task;         /* main task structure */
    int             preempt_count;    /* 0 => preemptable, <0 => bug */
    int             cpu;          /* cpu */
};
```

struct task_struct结构体重有一个stack的结构，此stack指针就是内核栈的指针。

接下来再看看内核stack和thread_info结构的关系

内核定义了一个thread_union的联合体，联合体的作用就是thread_info和stack共用一块内存区域。而THREAD_SIZE就是内核栈的大小，ARM64定义THREAD_SIZE的大小为16K

现在我们已经理清了这三个结构体的关系，下面通过一张图来说明下这三者的关系。



那如何获取一个进程的task_struct结构呢？ 我们获得当前内核栈的sp指针的地址，然后根据THREAD_SIZE对齐就可以获取thread_info结构的基地址，然后从thread_info.task就可以获取当前task_struct结构的地址了。

ThreadInfo在task_struct结构中

上面的一种方式是thread_info结构和内核栈共用一块存储区域，而另一种方式是thread_info结构存储在task_struct结构中。

```
struct task_struct {
```

```
#ifdef CONFIG_THREAD_INFO_IN_TASK
    /*
     * For reasons of header soup (see current_thread_info()), this
     * must be the first element of task_struct.
     */
    struct thread_info        thread_info;
#endif
    ...
```

可以看到必须打开CONFIG_THREAD_INFO_IN_TASK这个配置，这时候thread_info就会在task_struct的第一个成员。而task_struct中依然存在void* stack结构

接着看下thread_info结构，如下是ARM64架构定义的thread_info结构

```
struct thread_info {
    unsigned long        flags;        /* low level flags */
    mm_segment_t        addr_limit;    /* address limit */
#ifdef CONFIG_ARM64_SW_TTBR0_PAN
    u64            ttbr0;        /* saved TTBR0_EL1 */
#endif
    union {
        u64        preempt_count;    /* 0 => preemptible, <0 => bug */
        struct {
#ifdef CONFIG_CPU_BIG_ENDIAN
            u32    need_resched;
            u32    count;
#else
            u32    count;
            u32    need_resched;
#endif
        } preempt;
    };
};
```

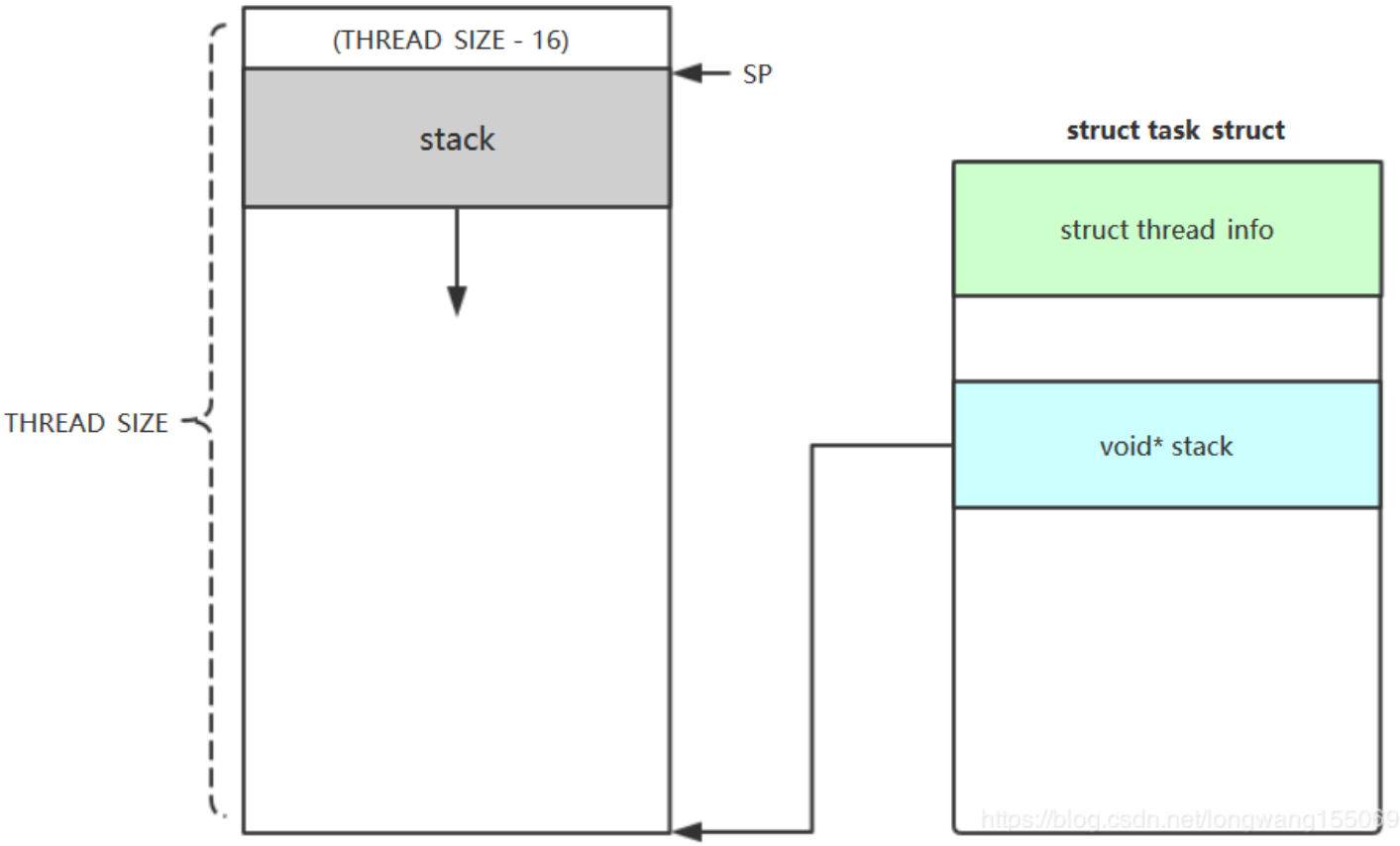
从此结构中则没有struct task_struct的指针了。

接着再来看下内核栈的定义：

```
union thread_union {
#ifndef CONFIG_THREAD_INFO_IN_TASK
    struct thread_info thread_info;
#endif
    unsigned long stack[THREAD_SIZE/sizeof(long)];
};
```

当CONFIG_THREAD_INFO_IN_TASK这个配置打开的时候，则thread_union结构中只存在stack成员了。

这时候我们再来看下当thread_info在task_struct结构中时，用一张图描述下。



当thread_info和内核栈是这种关系的时候，内核如何获取当前进程的task_struct结构呢?

通过分析current这个宏

```
static __always_inline struct task_struct *get_current(void)
{
    unsigned long sp_el0;

    asm ("mrs %0, sp_el0" : "=r" (sp_el0));

    return (struct task_struct *)sp_el0;
}

#define current get_current()
```

可以看到内核通过读取sp_el0的值，然后将此值强转成task_struct结构就可以获得。那sp_el0是什么东西?

sp：堆栈指针寄存器

el0: ARM64架构分为EL0,EL1,EL2,EL3。EL0则就是用户空间，EL1则是内核空间，EL2则是虚拟化，EL3则是secure层。

sp_el0: 则就是用户空间栈指针寄存器。

进程的标志

进程的标志由task_struct的flags字段表示，常见的取值如表14-3所示。

表14-3 进程标志表

标　志	描　述
PF_IDLE	idle 进程
PF_EXITING	进程正在退出
PF_EXITPIDONE	pi state 清理完毕，Priority Inheritance state
PF_VCPU	进程运行在虚拟 CPU 上
PF_WQ_WORKER	进程是一个 workqueue 的 worker
PF_SUPERPRIV	进程拥有超级用户权限
PF_SIGNALED	进程被某个信号杀掉
PF_NOFREEZE	进程不能被 freeze
PF_FROZEN	进程处于 frozen 状态
PF_KTHREAD	进程是一个内核线程

与信号相关的信息

```
/* Signal handlers: */
struct signal_struct    *signal;
struct sighand_struct    *sighand;
sigset_t      blocked;
sigset_t      real_blocked;
sigset_t      saved_sigmask;
struct sigpending    pending;
unsigned long      sas_ss_sp;
size_t        sas_ss_size;
unsigned int      sas_ss_flags;
```

- blocked：sigset_t 是一个位图，每个位都表示一个信号。blocked 表示的是该进程的哪些信号被阻塞暂不处理
- pending：表示进程接收到了哪些信号，需要被处理
- sighand：用户可以定义相应信号的处理方法，其信息保存在这里
- sas_ss_xxx：信号的处理默认使用的是进程用户空间的函数栈，也可以开辟新的栈专门用于信号处理，这三个变量就是用户维护这个栈信息

在 signal 中，定义了 struct sigpending shared_pending，这个 shared_pending 和 pending 的区别是，pending 表示该 task_struct 收到的信号，而 shared_pending 是整个线程组共享的。也就是说，对于 pending 中接收到的信号，只能由这个 task_struct 来处理，而 shared_pending 中收到的信号，可以由线程组中的任意一个线程处理

进程调度相关信息

```
1
2  //是否在运行队列上
3  int         on_rq;
4  //优先级
5  int         prio;//动态优先级
6  int         static_prio;//静态优先级
7  int         normal_prio;//常规优先级
8  unsigned int       rt_priority;
9  //调度器类
10 const struct sched_class  *sched_class;
11 //调度实体
12 struct sched_entity    se;
13 struct sched_rt_entity    rt;
14 struct sched_dl_entity    dl;
15 //调度策略
16 unsigned int       policy;
17 //可以使用哪些CPU
18 int         nr_cpus_allowed;
19 cpumask_t       cpus_allowed;
20 struct sched_info    sched_info;
```

进程间的关系

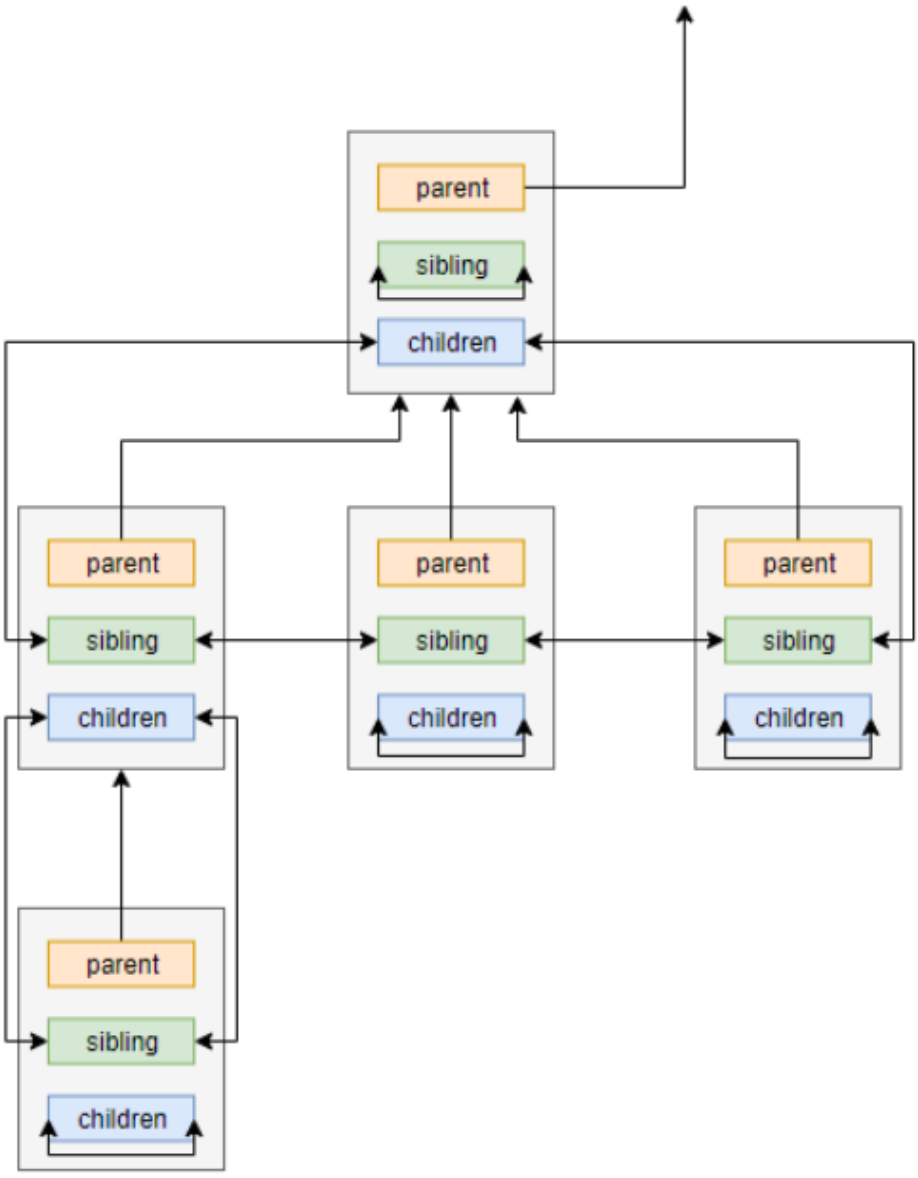
亲属关系

```
1 struct task_struct __rcu *real_parent; /* real parent process */
2 struct task_struct __rcu *parent; /* recipient of SIGCHLD, wait4() reports */
3 struct list_head children;      /* list of my children */
4 struct list_head sibling;       /* linkage in my parent's children list */
5 struct task_struct *group_leader;       /* threadgroup leader */
```

进程之间有亲缘关系，所以所有的进程实际上是一棵树，上面说过，进程组成一个双向循环链表，这并不冲突，因为既是双向循环链表，又是一棵树:

- parent：指向父进程
- children：所有子进程的组成的链表的链表头
- sibling：兄弟链表，又相当于父进程的 children 链表中的一个节点
- group_leader：指向其所在线程组的领头进程

所有进程组成的关系如下:



线程组

同一个线程组的线程，它们的task_struct都通过thread_group字段链接到同一个链表中，链表的头为线程组领导进程的task_struct的thread_group字段，可以据此来遍历线程组，如图14-1所示。

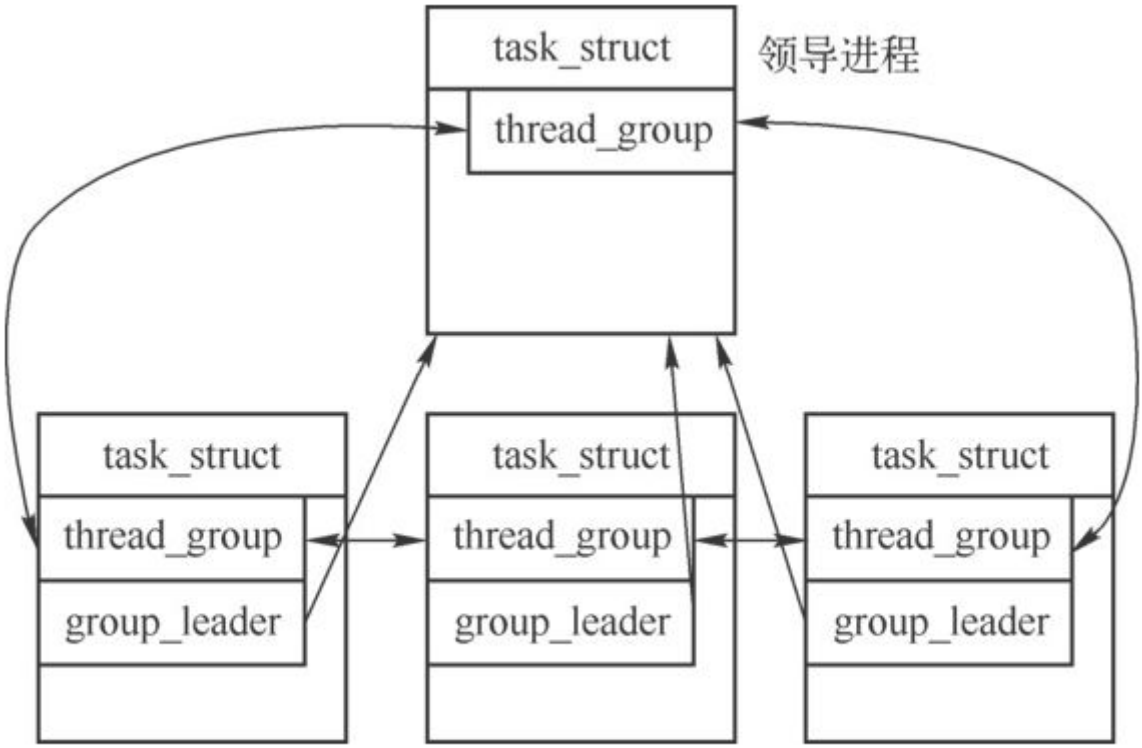


图14-1 线程组结构图

进程组

文件系统信息

```
1 struct fs_struct *fs;
2 struct files_struct *files;
```

每个进程都有自己的根目录和当前工作目录，内核使用 `struct fs_struct` 来记录这些信息，进程描述符的 `fs` 字段便是指向该进程的 `fs_struct` 结构。

除了根目录和当前工作目录，进程还需要记录自己打开的文件。进程已经打开的所有文件使用 `struct files_struct` 来记录，进程描述符的 `files` 字段便指向该进程的 `files_struct` 结构。

虚拟内存信息

task_struct进程描述符中包含两个跟进程地址空间相关的字段mm, active_mm

```
1 struct task_struct
2 {
3     // ...
```

```
    struct mm_struct *mm;
    struct mm_struct *avtive_mm;
    //...
};
```

对于普通用户进程来说，mm指向虚拟地址空间的用户空间部分，而对于内核线程，mm为NULL。

这位优化提供了一些余地, 可遵循所谓的惰性TLB处理(lazy TLB handing)。active_mm主要用于优化，由于内核线程不与任何特定的用户层进程相关，内核并不需要倒换虚拟地址空间的用户层部分，保留旧设置即可。由于内核线程之前可能是任何用户层进程在执行，故用户空间部分的内容本质上是随机的，内核线程决不能修改其内容，故将mm设置为NULL，同时如果切换出去的是用户进程，内核将原来进程的mm存放在新内核线程的active_mm中，因为某些时候内核必须知道用户空间当前包含了什么。

懒惰TLB

惰性TLB的目的就是为了避免多处理器系统上无用的TLB刷新，保证懒惰TLB模式下CPU可以专心执行内核进程，不会接受其他与TLB刷新相关的 * 处理器间中断 * 请求

当内核为某个用户态进程分配页框并把页框的物理地址存入页表项时（缺页中断需要完成的事情），它必须刷新本地CPU相应页的TLB区域，在多处理系统中如果有多个CPU正在使用相同的页表集，那么内核还必须刷新这些CPU上使用相同页表集的TLB区域。惰性是指，当CPU收到刷新TLB的指令时，如果此时CPU整处于内核态运行一个内核线程时，TLB的刷新将推迟到切换成用户态后再执行，因为内核态线程几乎不会访问用户进程地址空间。如果CPU处于用户态将即刻刷新对应页的TLB区域。

例子：进程A，B拥有相同的页表集，CPU1正在执行进程A但是遇到了硬件中断切换到内核态运行一个内核线程（硬件中断程序），此时CPU2正在执行B并触发了缺页中断，为进程B分配了新的页框，CPU2的TLB对应区域刷新。

1. 此时当CPU1收到刷新TLB请求时如果 不考虑惰性刷新时 TLB可能刷新1-2次
    a. 1. 立即刷新TLB
    b. 2. 硬件中断处理完后还是切换成进程B时TLB不刷新，硬件中断处理完后切换成进程C时TLB刷新
2. 此时当CPU1收到刷新TLB请求时如果 考虑惰性刷新时 TLB可能刷新1次
    a. 硬件中断处理完后还是切换成进程B时延时刷新TLB，硬件中断处理完后切换成进程C时TLB刷新

页面管理信息

对称多处理机（SMP）信息

和处理器相关的环境（上下文）信息

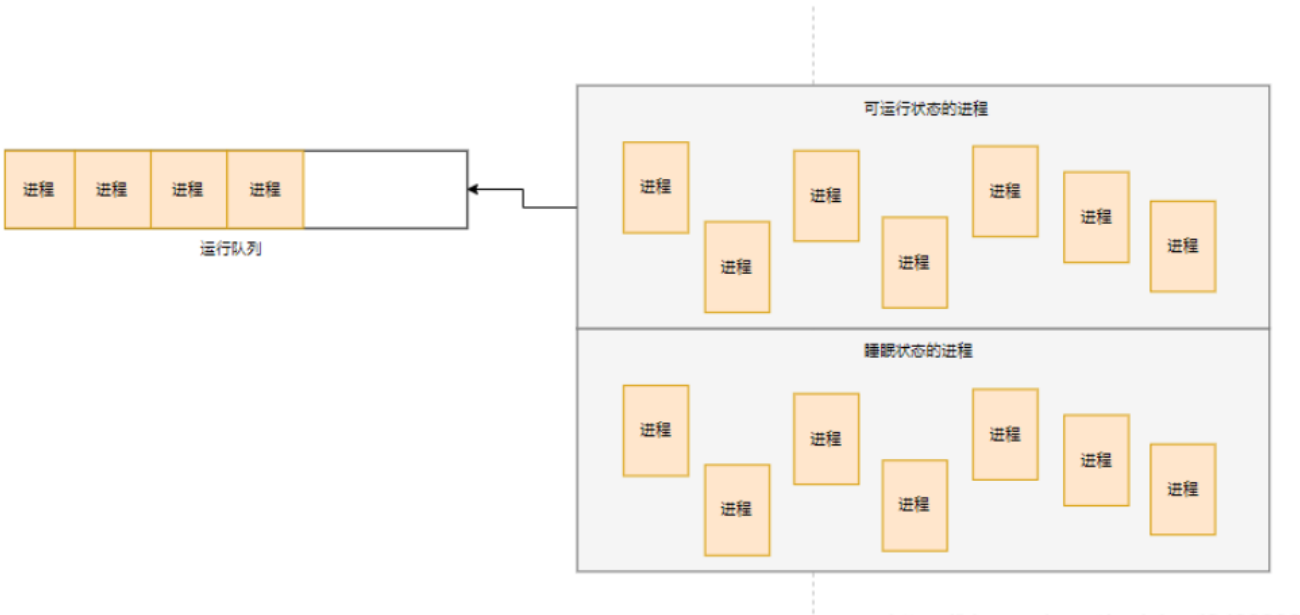
时间和定时器信息

```
u64        utime;//用户态消耗的CPU时间
u64        stime;//内核态消耗的CPU时间
unsigned long       nvcsw;//自愿(voluntary)上下文切换计数
unsigned long       nivcsw;//非自愿(involuntary)上下文切换计数
u64        start_time;//进程启动时间，不包含睡眠时间
u64        real_start_time;//进程启动时间，包含睡眠时间
```

进程调度

调度过程

操作系统管理非常多的进程，这些进程当前可能处于可运行状态或者睡眠状态。进程调度解决的是当前应该运行哪一个进程，它关心的对象是当前可运行状态的进程，内核为了管理这些可运行的进程，准备了一个运行队列，如下图所示



对于多CPU处理器，每一个CPU都有属于它的运行队列

我们将CPU当前正在运行的进程称为 current 进程，current 进程是不在运行队列中的，如下图所示：

进程切换一般分为两步：

- 第一步对current进程设置需要重新调度标志**TIF_NEED_RESCHED**
- 第二步在系统调用返回或中断返回时等**时机**检查current进程是否设置了需要重新调度标志，如果需要，则调用schedule发生进程切换

设置调度标志的时机

首先我们来解决第一步，设置current进程需要重新调度的标志

我们通过什么机制来设置current进程需要重新调度的标志呢?

内核在两种情况下会设置该标志，一个是在时钟中断进行周期性的检查时，另一个是在被唤醒进程的优先级比正在运行的进程的优先级高时。

时钟中断的周期性检查

硬件电路中有一个硬件定时器，它负责周期性的产生时钟中断（一般为10ms），我们称它为滴答定时器，可以认为，它就是操作系统的心脏。每当产生定时器中断的时候，CPU就会执行中断处理程序：

定时中断处理函数中会调用**schedule_tick()**用于处理关于调度的周期性检查和处理，其调用路径是和时钟处理有关的tick_periodic()->update_process_times()->scheduler_tick()或者tick_sched_handle()->update_process_times()->scheduler_tick()，主要用于更新就绪队列的时钟、CPU负载和当前任务的运行时间统计等，如下所示：

在滴答定时器的中断处理中，我们会判断current进程是否需要被抢占，怎么判断?

很明显，这一部分需要具体的调度算法来实现，Linux将调度算法的实现抽象成调度类

在滴答定时器的中断处理中，通过调度类去实现相应的计算，然后判断current进程是否需要被抢占，如果需要被抢占，那么就在current进程设置需要重新调度的标志。

```
1 void scheduler_tick(void)
2 {
3         int cpu = smp_processor_id();
4         struct rq *rq = cpu_rq(cpu);
5         struct task_struct *curr = rq->curr;
6         struct rq_flags rf;
7
8         sched_clock_tick();
```

```
 9
10          rq_lock(rq, &rf);
11
12          update_rq_clock(rq);
13          curr->sched_class->task_tick(rq, curr, 0);
14          cpu_load_update_active(rq);
15          calc_global_load_tick(rq);
16          psi_task_tick(rq);
17
18          rq_unlock(rq, &rf);
19
20          perf_event_task_tick();
21
22  #ifdef CONFIG_SMP
23          rq->idle_balance = idle_cpu(cpu);
24          trigger_load_balance(rq);
25  #endif
26  }
```

update_rq_clock(rq);会更新当前CPU就绪队列（rq）中的时钟计数clock和clock_task成员。

task_tick(rq, curr, 0);是调度类中实现的方法，其最重要的是检查是否需要调度。

cpu_load_update_active(rq);更新运行队列中的cpu_load[]数组

trigger_load_balance(rq);触发SMP负载均衡机制

睡眠的任务被唤醒时：

当睡眠任务所等待的事件到达时，内核（例如驱动程序的中断处理函数）将会调用wake_up()唤醒相关的任务，并最终调用try_to_wake_up()。它完成三件事：将任务重新添加到就绪队列，将运行标志设置为TASK_RUNNING，如果被唤醒的任务可以抢占当前运行任务则设置当前任务的TIF_NEED_RESCHED标志。

实际调度过程schedule 函数

```
1  asmlinkage __visible void __sched schedule(void)
2  {
3       ...
4       __schedule(false);
5       ...
6  }
```

schedule 函数最主要的是调用 __schedule 函数，下面看一下 __schedule 函数的定义:

```
static void __sched notrace __schedule(bool preempt)
{
    struct task_struct *prev, *next;
    struct rq *rq;
    int cpu;

    /* 获取运行队列 */
    cpu = smp_processor_id();
    rq = cpu_rq(cpu);
    prev = rq->curr;

    ...
 /* 从运行队列中挑选下一个运行的进程 */
    next = pick_next_task(rq, prev, &rf);
    ...

    ...
 /* 发生进程切换 */
    context_switch(rq, prev, next, &rf);
 ...
}
```

- 首先获取CPU对应的运行队列，前面我们说过，每个CPU都有其自己对应的运行队列
- 然后通过 pick_next_task 来获取下一个运行的进程
- 最后通过 context_switch 来实现进程切换

所以 schedule 函数可以总结成两件事，第一件事就是从运行队列中挑选下一个运行的进程，第二件事就是实现进程切换

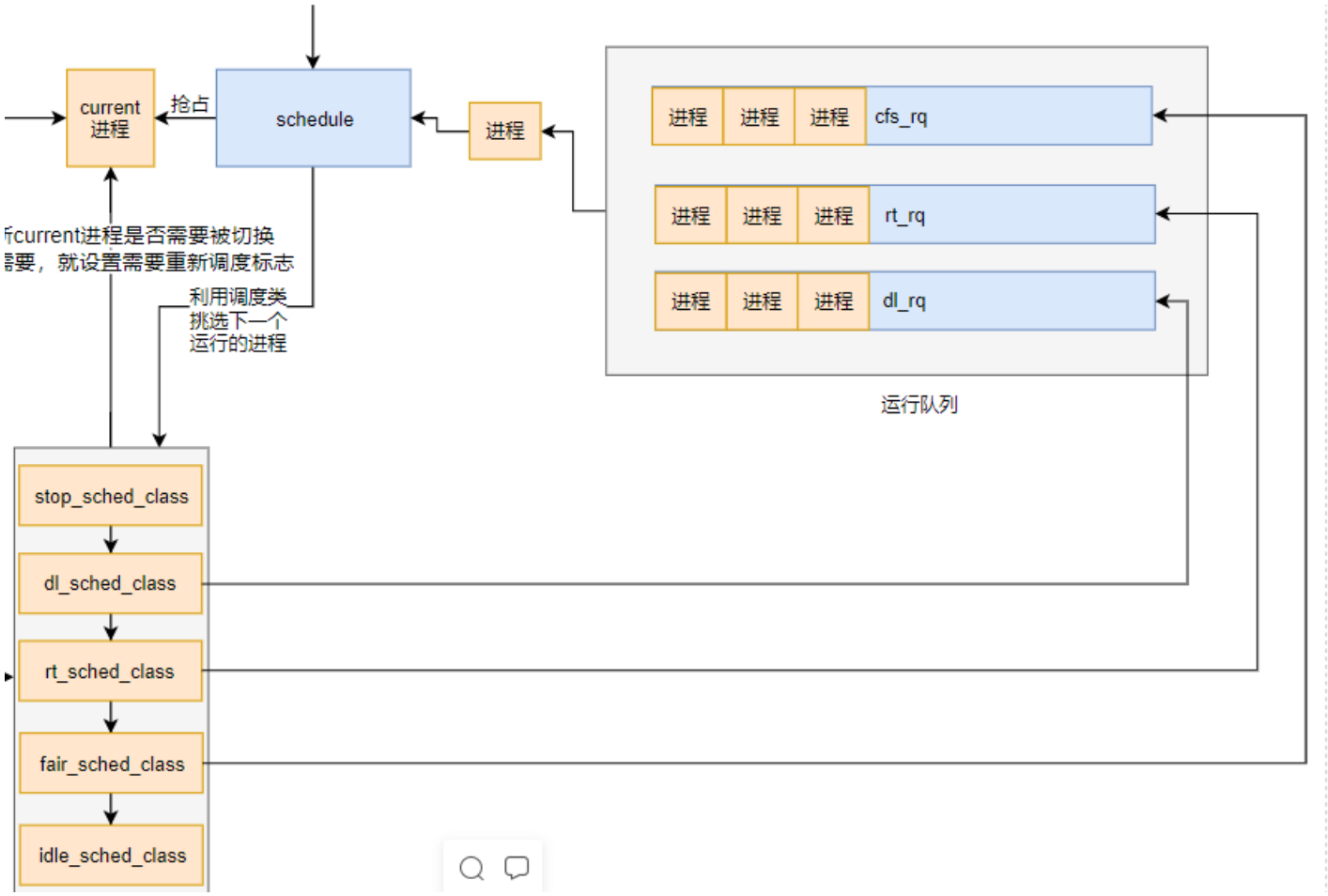
挑选下一个运行的进程

首先我们来看如何通过 pick_next_task 来获取下一个运行的进程，其定义如下:

```
static inline struct task_struct *
pick_next_task(struct rq *rq, struct task_struct *prev, struct rq_flags *rf)
{
        const struct sched_class *class;
        struct task_struct *p;

         /*此处有个优化，如果当前进程prev的调度类是CFS的调度类，并且该CPU整个就绪队列
          *中的进程数量等于CFS就绪队列中的进程数量，那么说明该CPU就绪队列中只有普通进
          *程而没有其它调度类进程，否则遍历所有调度类
          */
        if (likely((prev->sched_class == &idle_sched_class ||
                    prev->sched_class == &fair_sched_class) &&
                   rq->nr_running == rq->cfs.h_nr_running)) {
```

```
14
15                  p = fair_sched_class.pick_next_task(rq, prev, rf);
16                  if (unlikely(p == RETRY_TASK))
17                          goto again;
18
19                  /* Assumes fair_sched_class->next == idle_sched_class */
20                  if (unlikely(!p))
21                          p = idle_sched_class.pick_next_task(rq, prev, rf);
22
23                  return p;
24          }
25
26  again:
27          for_each_class(class) {
28                  p = class->pick_next_task(rq, prev, rf);
29                  if (p) {
30                          if (unlikely(p == RETRY_TASK))
31                                  goto again;
32                          return p;
33                  }
34          }
35
36          /* The idle class should always have a runnable task: */
37          BUG();
38  }
```

该CPU就绪队列中有除完全公平调度之外的其它调度类进程按优先级遍历所有的调度类，从对应的运行队列中找到下一个运行的任务。



进程调度中的结构体

进程执行在CPU上，所以CPU需要记录正在和将要运行在它上的进程的情况，内核以rq（runqueue）结构体描述它，主要字段如表所示。

表15-1 rq字段表

字　段	类　型	描　述
nr_running	unsigned int	TASK_RUNNING 状态的进程数
cfs	cfs_rq	完全公平调度 rq
rt	rt_rq	实时调度 rq
dl	dl_rq	最终期限 rq
curr	task_struct *	当前执行的进程
idle	task_struct *	idle 进程
stop	task_struct *	stop 进程
clock	u64	CPU 累计运行的时间，单位为纳秒
clock_task	u64	进程累计占用 CPU 的时间
cpu	int	对应的 CPU

每颗cpu都有一个运行队列rq，这个队列中又存在多个**子队列**例如rt_rq(实时运行队列),cfs_rq。当cpu需要一个任务执行时，其会先按照优先级选择不同的调度类，不同的调度类操作不同的队列，例如

rt_sched_class先被调用，其会在rt_rq中找下一个任务，只有找不到时才轮到fair_sched_class被调用，它会在cfs_rq中寻找下一个任务。clock_task字段表示进程累计运行的时间，有可能不包括中断处理等时间（与系统的配置有关），clock可能比它大。

rq结构体并没有直接与task_struct关联的字段，所以进程也并不由它直接管理，实际的管理者是cfs、rt和dl字段。

cfs_rq结构体定义了与完全公平调度（Completely Fair Scheduler，cfs）相关的字段，如表所示。

表15-3 cfs_rq字段表

字　段	类　型	描　述
nr_running	unsigned int	TASK_RUNNING 状态的进程数
min_vruntime	u64	见下文
tasks_timeline.rb_root	rb_root	sched_entity 对象组成的红黑树的根
tasks_timeline.rb_leftmost	rb_node *	红黑树最左边的叶子
curr next	sched_entity *	当前/下一个 sched_entity

cfs_rq维护了一个由sched_entity对象组成的红黑树（下文称之为进程时间轴红黑树），task_struct的se字段就是sched_entity类型，正是它关联了cfs_rq和task_struct。

rt_rq结构体定义了与实时调度（rt, Real Time）相关的字段，如表所示。

表15-4 rt_rq字段表

字　段	类　型	描　述
active	rt_prio_array	priority array
highest_prio.curr	int	rt 进程的最高优先级
highest_prio.next	int	rt 进程的次高优先级
rt_throttled	int	rt 进程被禁止
rt_time	u64	实时进程累计执行时间
rt_runtime	u64	实时进程最大执行时间

active字段是rt_prio_array类型，定义如下。

```
1 struct rt_prio_array {
2         DECLARE_BITMAP(bitmap, MAX_RT_PRIO+1); /* include 1 bit for delimiter */
3         struct list_head queue[MAX_RT_PRIO];
4 };
```

MAX_RT_PRIO等于100，实时进程的优先级是0～99，queue字段定义了100个链表，实时进程根据优先级链接到链表上，优先级与queue数组的下标相等。queue数组的链表链接的是sched_rt_entity结构体，也就是task_struct的rt字段。

dl_rq定义了与最后期限（dl, Deadline）调度相关的字段，如表所示

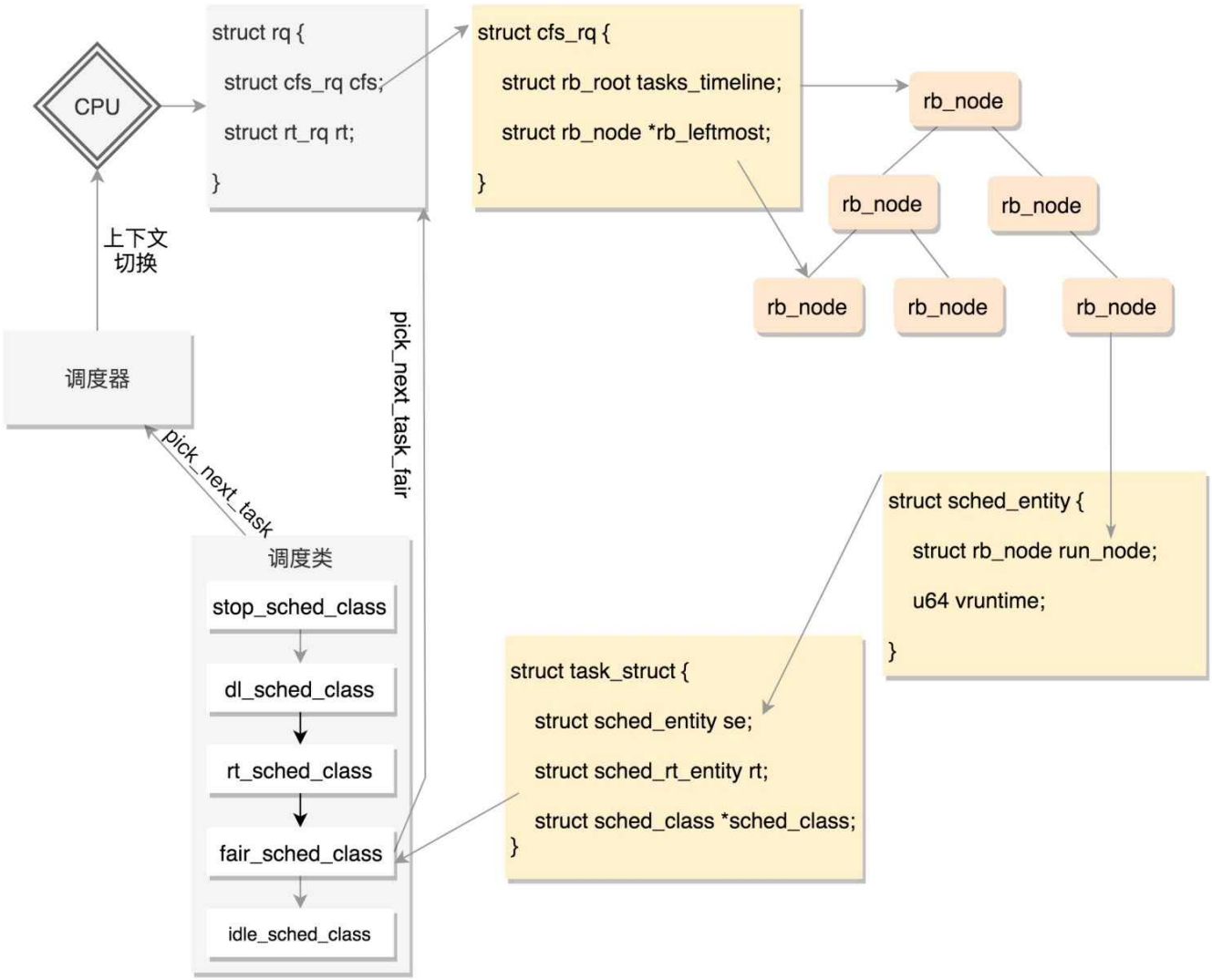
表15-5 dl_rq字段表

字　段	类　型	描　述
root.rb_root	rb_root	sched_dl_entity 对象组成的红黑树的根
root.rb_leftmost	rb_node *	红黑树最左边的叶子
dl_nr_running	unsigned long	TASK_RUNNING 状态的进程数
earliest_dl.curr	u64	当前/下一个 sched_dl_entity
earliest_dl.next	u64	

与cfs_rq和task_struct的关系类似，dl_rq维护了一个由sched_dl_entity对象（task_struct的dl字段）组成的红黑树。

rq的cfs管理完全公平调度的进程，rt管理实时进程，dl管理最后期限调度的进程。

进程调度还有另外一种以组来管理进程的方式，也就是占用CPU时间不以单个进程计算，以一组进程来衡量，以上三个结构体都有额外的字段来实现该功能。



进程优先级

	A	B
1	字段	描述
2	static_prio	用于保存静态优先级, 是进程启动时分配的优先级, , 可以通过nice和sched_setscheduler系统调用来进行修改, 否则在进程运行期间会一直保持恒定
3	rt_priority	用于保存实时优先级
4	normal_prio	表示基于进程的静态优先级static_prio和调度策略计算出的优先级. 因此即使普通进程和实时进程具有相同的静态优先级, 其普通优先级也是不同的, 进程分叉(fork)时, 子进程会继承父进程的普通优先级
5	prio	保存进程的动态优先级

静态优先级，static_prio = 100 + nice + 20 (nice值为-20~19,所以static_prio值为100~139)

为什么表示动态优先级需要两个值prio和normal_prio

调度器会考虑的优先级则保存在prio. 由于在某些情况下内核需要暂时提高进程的优先级, 因此需要用prio表示. 由于这些改变不是持久的, 因此静态优先级static_prio和普通优先级normal_prio不受影响.

5种调度类

数据结构

linux 在 2.6 版本中引入了“调度类（Sched Class）”的概念，意在将调度逻辑模块化，内核通过 `struct sched_class` 抽象出了调度类的通用行为：

```
struct sched_class {
        const struct sched_class *next;
        //将进程插入/移除相关优先级队列或红黑树
        void (*enqueue_task) (struct rq *rq, struct task_struct *p, int flags);
        void (*dequeue_task) (struct rq *rq, struct task_struct *p, int flags);

        //任务主动让出 CPU，  但是其状态依然是 runnable
        void (*yield_task)   (struct rq *rq);
        bool (*yield_to_task)(struct rq *rq, struct task_struct *p, bool preempt)

        /* 检查 p 是否会抢占 rq 中当前正在运行的 task. 通常情况下是在 p 进入 runnable
        * 状态时，检查 p 是否会抢占当前正在运行的 task */
        void (*check_preempt_curr)(struct rq *rq, struct task_struct *p, int flag

        /* 从 rq 中选择下一个任务来运行 */
        struct task_struct * (*pick_next_task)(struct rq *rq,
                                               struct task_struct *prev,
                                               struct rq_flags *rf);
        //将当前进程放入运行队列的合适位置
        void (*put_prev_task)(struct rq *rq, struct task_struct *p);

#ifdef CONFIG_SMP
        /* 当系统调用 fork，exec 创建一个新的 task 时，在 SMP 系统中需要选择一个合理的
         * runqueue 来将该 task 入队。Scheduler 此时需要考虑负载均衡问题 */
        int  (*select_task_rq)(struct task_struct *p, int task_cpu, int sd_flag,

        /* 将任务迁移到目标 CPU 上 */
        void (*migrate_task_rq)(struct task_struct *p, int new_cpu);

        /* 当任务被唤醒（wake up）之后调用 */
        void (*task_woken)(struct rq *this_rq, struct task_struct *task);

        /* 修改任务的 CPU 偏好，即其可以被调度到哪些 CPU 上运行 */
        void (*set_cpus_allowed)(struct task_struct *p,
                                 const struct cpumask *newmask);

        //调用该函数 ，可以启动运行队列 ;
```

```
38          void (*rq_online)(struct rq *rq);
39          void (*rq_offline)(struct rq *rq);
40  #endif
41          // 当进程改变它的调度类或进程组时被调用
42          void (*set_curr_task)(struct rq *rq);
43
44          //其最重要的是检查是否需要调度。
45          void (*task_tick)(struct rq *rq, struct task_struct *p, int queued);
46
47          // 当进程创建的时候调用，不同的调度策略的进程初始化也是不一样的
48          void (*task_fork)(struct task_struct *p);
49
50          // 进程退出时会使用
51          void (*task_dead)(struct task_struct *p);
52
53          //进行实际的上下文的保存与切换
54          void (*switched_from)(struct rq *this_rq, struct task_struct *task);
55          void (*switched_to)  (struct rq *this_rq, struct task_struct *task);
56
57          // 改变进程的优先级
58          void (*prio_changed) (struct rq *this_rq, struct task_struct *task,
59                                int oldprio);
60
61          unsigned int (*get_rr_interval)(struct rq *rq,
62                                          struct task_struct *task);
63          //更新当前正在运行的调度实体的运行时间信息。
64          void (*update_curr)(struct rq *rq);
65
66  #define TASK_SET_GROUP                0
67  #define TASK_MOVE_GROUP                1
68
69  #ifdef CONFIG_FAIR_GROUP_SCHED
70          void (*task_change_group)(struct task_struct *p, int type);
71  #endif
72  };
```

stop调度类

整个进程调度的过程中，涉及了sched_class的很多操作，它们在调度过程中扮演着重要的角色。每个sched_class都会根据它们代表的进程的需求，针对性地定制这些操作，接下来按照优先级逐个分析它们。需要说明的是，这里只是列举了几个常用的事件，不要把思路局限于此，这些操作可能还用于其他使用场景，比如15.8小节中的set_user_nice也会调用dequeue和enqueue等操作。

它们定义操作时，使用的函数名基本按照“操作名_class名”形式，比如stop_sched_class的pick_next_task对应的函数为pick_next_task_stop。

stop_sched_class（以下简称stop类）代表优先级最高的进程，它实际上只定义了pick_next_task_stop和put_prev_task_stop，enqueue_task_stop和dequeue_task_stop也只是操作了rq的nr_running字段而已，其余的操作要么没有定义，要么为空。

task_tick_stop为空，说明时钟中断对stop类代表的进程没有影响，理论上占用CPU无时间限制。check_preempt_curr_stop也为空，说明没有进程可以抢占它。

enqueue_task_stop和dequeue_task_stop的策略说明进程不可能通过wakeup类函数变成stop类管理的进程。实际上，一般只能通过调用sched_set_stop_task函数指定进程的sched_class为stop类，该函数会将进程的task_struct赋值给rq的stop字段，所以对一个rq而言，同一时刻最多只能有一个stop进程。

pick_next_task_stop判断rq->stop是否存在，如果存在且stop->on_rq等于TASK_ON_RQ_QUEUED，返回stop，否则返回NULL。

总结，只要stop进程存在，且没有睡眠，它就是最高优先级，不受运行时间限制，也不会被抢占。鉴于它如此强大的“背景”，一般进程是不可能享受类似待遇的，少数场景如hotplug才有可能使用它。

最后期限调度类

dl_rq也使用红黑树管理进程，树的根为它的root.rb_root字段。

关联task_struct和dl_rt的是task_struct的dl字段，类型为sched_dl_entity，主要字段如表15-12所示。

表15-12 sched_dl_entity字段表

字　段	类　型	描　述
rb_node	rb_node	将 sched_dl_entity 链接到红黑树
dl_runtime	u64	最大运行时间
dl_deadline	u64	相对截止时间
runtime	s64	剩余运行时间
deadline	u64	绝对截止时间
flags	unsigned int	标志

deadline是由dl_deadline加上当前时间得到的，runtime随着task_tick减少。

cfs_rq的红黑树比较的是sched_entity.vruntim，dl_rq的红黑树比较的则是sched_dl_entity.deadline，deadline小的优先执行。

最后期限调度类有以下两点特别之处。

首先，task_fork_dl定义为空，意味着不能创建一个由最后期限调度类管理的进程。那么deadline进程从何而来？一般由已存在的进程调用sched_setattr变成deadline进程。

```
int sched_setattr(pid_t pid, struct sched_attr *attr, unsigned int
flags);
```

其次，我们在创建进程过程中调用的sched_fork函数的第3步，if (dl_prio(p->prio)) return -EAGAIN，意味着最后期限调度类进程不能创建进程，除非它的task_struct ->sched_reset_on_fork字段置1。

实时调度类

rt_sched_class（以下简称rt类）比stop_sched_class丰满许多，它管理实时进程。

我们在数据结构一节介绍过，sched_rt_entity结构体充当rq和task_struct的媒介，如图所示。

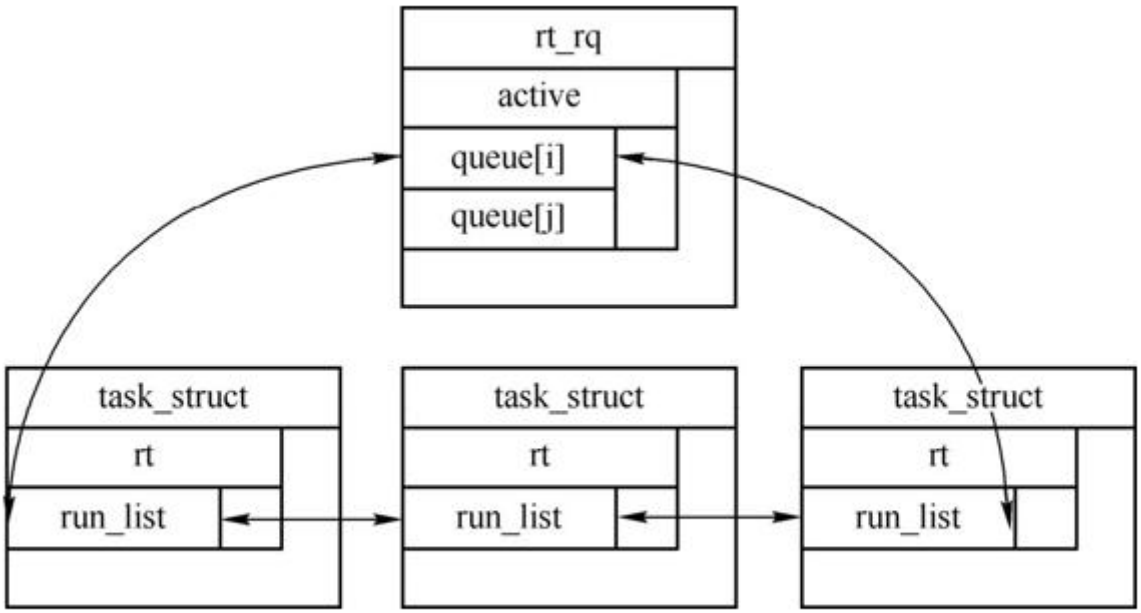


图15-1 rt_rq和task_struct关系图

很显然，enqueue_task_rt的主要任务就是要建立图15-1中的关系，具体代码如下。

```
void enqueue_task_rt(struct rq *rq, struct task_struct *p, int flags)
{
    struct sched_rt_entity *rt_se = &p->rt;
    enqueue_rt_entity(rt_se, flags);
    if (!task_current(rq, p) && p->nr_cpus_allowed > 1)
        enqueue_pushable_task(rq, p);
}
```

enqueue_rt_entity的任务是将rt_se插入到进程优先级（p->prio）对应的链表中，如果进程不是rq当前执行的进程，且它可以在其他rq上执行，enqueue_pushable_task将进程（p->pushable_tasks）按照优先级顺序插入rq->rt.pushable_tasks链表中（优先级高的进程在前），并根据情况更新rq->rt.highest_prio.next字段。

check_preempt_curr_rt判断进程p是否应该抢占目标rq当前正在执行的进程，如果p的优先级更高，则调用resched_curr(rq)请求抢占rq。如果二者优先级相等（意味着都是实时进程），且当前进程的TIF_NEED_RESCHED标记没有置位，调用check_preempt_equal_prio。如果rq->curr可以在其他

CPU上执行（可迁移），而p不可以（不可迁移），check_preempt_equal_prio也会调用resched_curr(rq)，这样p可以在目标CPU上执行，rq->curr换一个CPU继续执行。

check_preempt_curr_rt可以抢占普通进程，也可以抢占实时进程。

task_woken_rt在满足一系列条件的情况下，调用push_rt_tasks，代码如下。

```
void task_woken_rt(struct rq *rq, struct task_struct *p)
{
    if (!task_running(rq, p) &&
        !test_tsk_need_resched(rq->curr) &&

        p->nr_cpus_allowed > 1 &&
        (dl_task(rq->curr) || rt_task(rq->curr)) &&
        (rq->curr->nr_cpus_allowed < 2 ||
         rq->curr->prio <= p->prio))
        push_rt_tasks(rq);
}
```

注意，此时的rq指的并不是当前进程的rq，而是被唤醒的进程所属的rq，二者可能不同。满足以上条件的情况下，push_rt_tasks循环执行push_rt_task(rq)直到它返回0。其中一个条件是dl_task(rq->curr)|| rt_task(rq->curr)，也就是说task_woken_rt只会在rq正在执行实时进程或者最后期限进程的时候才会考虑push_rt_tasks。

push_rt_task选择rq->rt.pushable_tasks链表上的进程，判断它的优先级是否比rq当前执行的进程优先级高。如果是，则调用resched_curr请求进程切换，返回0；否则调用find_lock_lowest_rq尝试查找另一个合适的rq，记为lowest_rq，找到则切换被唤醒进程的rq至lowest_rq，然后调用resched_curr(lowest_rq)请求抢占lowest_rq。

所以，task_woken_rt可能抢占进程所属的rq，有可能抢占其他rq。当然了，抢占其他rq并不是没有要求的，find_lock_lowest_rq查找lowest_rq的最基本要求就是lowest_rq->rt.highest_prio.curr>p->prio，也就是进程的优先级比rq上的进程的优先级都高。

不难发现，内核将可以在多个CPU上执行的（p->nr_cpus_allowed>1）实时进程划为一类，称作pushable tasks，并给它们定制了特殊的机制。它们比较灵活，如果抢占不了目标rq，则尝试抢占其他可接受的rq，也可以在一定的条件下让出CPU，选择其他CPU继续执行，这样的策略无疑可以减少实时进程的等待时间。

task_tick_rt函数

task_tick_rt在时钟中断时被调用，主要逻辑如下。

```
void task_tick_rt(struct rq *rq, struct task_struct *p, int queued)
{
    struct sched_rt_entity *rt_se = &p->rt;

    update_curr_rt(rq);    //1
    if (p->policy != SCHED_RR)    //2
        return;
    if (--p->rt.time_slice)    //3
        return;

    p->rt.time_slice = sched_rr_timeslice;    //4
    if (rt_se->run_list.prev != rt_se->run_list.next) {
        requeue_task_rt(rq, p, 0);
        resched_curr(rq);


        return;
    }
}
```

第1步，update_curr_rt更新时间，主要逻辑如下。

```
void update_curr_rt(struct rq *rq)
{
    //省略变量定义、同步等
    now = rq->clock_task;
    delta_exec = now - curr->se.exec_start;
    curr->se.sum_exec_runtime += delta_exec;
    curr->se.exec_start = now;

    if (!rt_bandwidth_enabled())
        return;

    if (sched_rt_runtime(rt_rq) != RUNTIME_INF) {
        rt_rq->rt_time += delta_exec;
        if (sched_rt_runtime_exceeded(rt_rq))
            resched_curr(rq);
    }
}
```

update_curr_rt引入了实时进程带宽（rt bandwidth）的概念，为了不让普通进程“饿死”，默认情况下1秒（sysctl_sched_rt_period）时间内，实时进程总共可占用CPU 0.95秒（sysctl_sched_rt_runtime）。

rt_rq->rt_time表示一个时间间隔（1s）内，实时进程累计执行的时间，sched_rt_runtime_exceeded判断该时间是否已经超过0.95s（rt_rq->rt_runtime），如果是，将rt_rq->rt_throttled置1，并返回1；这种情况下update_curr_rt会调用resched_curr申请切换进程。

update_curr_rt只是增加了rt_rq->rt_time，那么1s时间间隔的逻辑是如何实现的呢？答案是有一个hrtimer定时器（rt_bandwidth的rt_period_timer字段），每隔1s执行一次sched_rt_period_timer函数，由它来配合调整rt_rq->rt_time，并在一定的条件下清除rt_rq->rt_throttled，让实时进程可以继续执行。

此处需要插入说明，函数中使用的curr->se是sched_entity类型的，也就是说sched_entity结构体的部分成员不仅用于CFS调度，也可以在其他调度中用来统计时间。

task_tick_rt的第2步，如果进程的调度策略不是SCHED_RR，函数直接返回。实时进程共有SCHED_FIFO（先到先服务）和SCHED_RR（时间片轮转）两种调度策略，这说明采用SCHED_FIFO策略的进程除了受带宽影响之外，不受执行时间限制。

一个SCHED_RR策略的实时进程的时间片由task_struct的rt.time_slice字段表示，初始值为sched_rr_timeslice，等于100 × HZ/1000，也就是100ms。每一次tick，rt.time_slice就会减1（第3步），如果减1后它仍大于0，说明进程的时间片还没有用完，继续执行；如果等于0，说明分配给进程的时间片已经用完了。第4步，将rt.time_slice恢复至sched_rr_timeslice，将进程移动到优先级链表（rt_rq->active.queue[prio]）的尾部，这样下次调度优先执行链表中其他进程，然后调用resched_curr申请切换进程。

选择下一个进程

pick_next_task_rt尝试选择一个实时进程，展开如下。

```
        struct task_struct *
        pick_next_task_rt(struct rq *rq, struct task_struct *prev, struct
rq_flags *rf)
        {
            struct rt_prio_array *array = &rt_rq->active;
            struct sched_rt_entity * rt_se = NULL;
            struct list_head *queue;
            idx = sched_find_first_bit(array->bitmap);
            queue = array->queue + idx;
            rt_se = list_entry(queue->next, struct sched_rt_entity, run_list);
            p = rt_task_of(rt_se);
            p->se.exec_start = rq->clock_task;
            return p;
        }
```

查找优先级链表中优先级最高的非空链表，获取链表上第一个进程，更新它的执行时间，然后将它返回。同一个链表上的进程，按照先后顺序依次执行，呼应了SCHED_RR策略（用光了时间片的进程插入到链表尾部）。

完全公平调度

fair_sched_class管理普通进程的调度，它的调度方式又被称为完全公平调度（Completely Fair Scheduler，CFS）。完全公平是理论上的，所有的进程同时执行，每个占用等份的一小部分CPU时间。实际的硬件环境中这种假设是不可能满足的，而且进程间由于优先级不同，本身就不应该公平对待，所以CFS引入了虚拟运行时间（virtual runtime）的概念。

虚拟运行时间是CFS挑选进程的标准，由实际时间结合进程优先级转换而来，由task_struct的se.vruntime字段表示，vruntime小的进程优先执行。

该转换与task_struct的se.load字段有关，类型为load_weight，它有weight和inv_weight两个字段，由set_load_weight函数设置，精简后如下。

```
void set_load_weight(struct task_struct *p, bool update_load)
{
    int prio = p->static_prio - MAX_RT_PRIO;
    struct load_weight *load = &p->se.load;
    load->weight = scale_load(sched_prio_to_weight[prio]);
    load->inv_weight = sched_prio_to_wmult[prio];
    p->se.runnable_weight = load->weight;
}
```

sched_prio_to_weight和sched_prio_to_wmult两个数组分别如下。

```
static const int sched_prio_to_weight[40] = {
 /* -20 */     88761,     71755,     56483,     46273,     36291,
 /* -15 */     29154,     23254,     18705,     14949,     11916,
 /* -10 */      9548,      7620,      6100,      4904,      3906,
 /*  -5 */      3121,      2501,      1991,      1586,      1277,
 /*   0 */      1024,       820,       655,       526,       423,
 /*   5 */       335,       272,       215,       172,       137,
 /*  10 */       110,        87,        70,        56,        45,
 /*  15 */        36,        29,        23,        18,        15,
};
static const u32 sched_prio_to_wmult[40] = {
 /* -20 */     48388,     59856,     76040,     92818,    118348,
 /* -15 */    147320,    184698,    229616,    287308,    360437,
 /* -10 */    449829,    563644,    704093,    875809,   1099582,
 /*  -5 */   1376151,   1717300,   2157191,   2708050,   3363326,
 /*   0 */   4194304,   5237765,   6557202,   8165337,  10153587,
 /*   5 */  12820798,  15790321,  19976592,  24970740,  31350126,
 /*  10 */  39045157,  49367440,  61356676,  76695844,  95443717,
 /*  15 */ 119304647, 148102320, 186737708, 238609294, 286331153,
};
```

数组中的每一个元素对应一个进程优先级，数组的下标与p->static_prio - MAX_RT_PRIO相等。普通进程的优先级（static_prio）范围为[100, 140)，优先级越高，weight越大，inv_weight越小，weight和inv_weight的乘积约等于2^32。

以优先级等于120的进程作为标准，它的实际时间和虚拟时间相等。其他进程需要在此基础上换算，由calc_delta_fair函数完成，片段如下。

```
if (unlikely(se->load.weight != NICE_0_LOAD))
    delta = __calc_delta(delta, NICE_0_LOAD, &se->load);
return delta;
```

NICE_0_LOAD等于prio_to_weight[120-100]，也就是1024，不考虑数据溢出等因素的情况下，__calc_delta可以粗略简化为delta*=NICE_0_LOAD/(se->load.weight)。同样的delta，weight越大的进程，转换后得到的时间越小，也就是说进程优先级越高，它的虚拟时间增加得越慢，而虚拟时间越小的进程越先得到执行。由此可见，CFS确实优待了优先级高的进程。

我们在数据结构一节说过，task_struct是通过sched_entity结构体与cfs_rq建立联系的。事实上，sched_entity的功能不局限于此，主要字段如表15-10所示。

表15-10 sched_entity字段表

字　段	类　型	描　述
load	load_weight	load weight，已介绍
run_node	rb_node	将它链接到 cfs_rq 的红黑树中
on_rq	unsigned int	是否在 rq 上
exec_start	u64	上次更新时的时间，实际时间
sum_exec_runtime	u64	总共执行的时间，实际时间
vruntime	u64	virtual runtime，虚拟时间
prev_sum_exec_runtime	u64	被调度执行时的总执行时间，实际时间

有了上面的理论基础，下面分析fair_sched_class的操作。

task_fork_fair在普通进程被创建时调用，主要逻辑如下。

```
void task_fork_fair(struct task_struct *p)
{
    struct cfs_rq *cfs_rq;
    struct sched_entity *se = &p->se, *curr;
    struct rq *rq = this_rq();
    //省略同步等
    update_rq_clock(rq);    //1

    cfs_rq = task_cfs_rq(current);
    curr = cfs_rq->curr;
    if (curr) {
        update_curr(cfs_rq);    //2
        se->vruntime = curr->vruntime;    //3
    }

    place_entity(cfs_rq, se, 1);    //4

    if (sysctl_sched_child_runs_first && curr && entity_before(curr,

se)) {
            swap(curr->vruntime, se->vruntime);
            resched_curr(rq);
        }

        se->vruntime -= cfs_rq->min_vruntime;    //5
    }
```

task_fork_fair本身并不复杂，但它涉及了理解CFS的两个重点。

第2步，update_curr更新cfs_rq和它当前执行进程的信息，是CFS中比较重要的函数，展开如下。

```
void update_curr(struct cfs_rq *cfs_rq)
{
    struct sched_entity *curr = cfs_rq->curr;
    u64 now = rq_of(cfs_rq)->clock_task;

    delta_exec = now - curr->exec_start;
    curr->exec_start = now;
    curr->sum_exec_runtime += delta_exec;
    curr->vruntime += calc_delta_fair(delta_exec, curr);
    update_min_vruntime(cfs_rq);
}
```

clock_task是在第1步中update_rq_clock函数更新的，curr->exec_start表示进程上次update_curr的时间，二者相减得到时间间隔delta_exec。delta_exec是实际时间，可以直接加到curr->sum_exec_runtime上，增加进程的累计运行时间；但它必须通过calc_delta_fair函数转换为虚拟时间，才可以加到curr->vruntime上。

update_min_vruntime会更新cfs_rq->min_vruntime，又是一个重点，主要逻辑如下。

```
        void update_min_vruntime(struct cfs_rq *cfs_rq)
        {
            struct sched_entity *curr = cfs_rq->curr;
            struct rb_node *leftmost = rb_first_cached(&cfs_rq->tasks_
timeline);
            u64 vruntime = cfs_rq->min_vruntime;

            if (curr) {
                if (curr->on_rq)
                    vruntime = curr->vruntime;
                else
                    curr = NULL;
            }


            if (leftmost) {
                struct sched_entity *se;
                se = rb_entry(leftmost, struct sched_entity, run_node);

                if (!curr)
                    vruntime = se->vruntime;
                else
                    vruntime = min_vruntime(vruntime, se->vruntime);
            }
            cfs_rq->min_vruntime = max_vruntime(cfs_rq->min_vruntime,
vruntime);
        }
```

cfs_rq->min_vruntime是单调递增的，可以理解为cfs_rq上可运行状态的进程中，最小的vruntime。可运行状态的进程包含两部分，一部分是进程时间轴红黑树上的进程，还有一个就是cfs_rq正在执行的进程（它并不在红黑树上）。update_min_vruntime就是比较当前进程和红黑树最左边的进程的vruntime，然后取小。

你应该猜到了进程时间轴红黑树是按照进程的vruntime来排序的，较小者居左。

第4步，place_entity计算se->vruntime，如下。

```
        void place_entity(struct cfs_rq *cfs_rq, struct sched_entity *se, int
initial)
        {
            u64 vruntime = cfs_rq->min_vruntime;
            if (initial && sched_feat(START_DEBIT))    //1
                vruntime += sched_vslice(cfs_rq, se);
            if (!initial) {
                unsigned long thresh = sysctl_sched_latency;
                if (sched_feat(GENTLE_FAIR_SLEEPERS))    //2
                    thresh >>= 1;
                vruntime -= thresh;
            }
            se->vruntime = max_vruntime(se->vruntime, vruntime);
        }
```

进程被创建的时候，initial等于1，se->vruntime等于cfs_rq->min_vruntime加上sched_vslice(cfs_rq, se)，sched_vslice根据进程优先级和当前cfs_rq的情况计算得到进程期望得到的虚拟运行时间。

如果进程因为之前睡眠被唤醒，place_entity也会为它计算vruntime，initial等于0，se->vruntime等于cfs_rq->min_vruntime减去thresh，算是对睡眠进程的奖励。所以，是时候优化下那些while(true)的程序了，不需要执行的时候适当睡眠，无论对程序本身还是对系统都有好处。

第5步，se->vruntime减去cfs_rq->min_vruntime，等于sched_vslice(cfs_rq, se)。task_fork_fair执行时，新进程还不是“完全体”，不能插入cfs_rq的进程时间轴红黑树中。

enqueue_task和check_preempt

enqueue_task_fair调用enqueue_entity将进程插入进程优先级红黑树，主要逻辑如下。

```
        void enqueue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se, int
flags)
        {
            bool renorm = !(flags & ENQUEUE_WAKEUP) || (flags & ENQUEUE_MIGRATED);
            bool curr = cfs_rq->curr == se;
            if (renorm && curr)    //1
                se->vruntime += cfs_rq->min_vruntime;

            update_curr(cfs_rq);    //2

            if (renorm && !curr)
                se->vruntime += cfs_rq->min_vruntime;
            if (flags & ENQUEUE_WAKEUP)    //3
                place_entity(cfs_rq, se, 0);
            if (!curr)    //4
                __enqueue_entity(cfs_rq, se);
            se->on_rq = 1;
        }
```

如果并不是之前睡眠被唤醒，执行enqueue_entity第1步，结果是vruntime += cfs_rq->min_vruntime；如果进程之前睡眠，执行enqueue_entity第3步，调用place_entity，结果是se->vruntime=cfs_rq->min_vruntime-thresh。

第4步，__enqueue_entity根据se->vruntime将se插入进程优先级红黑树，vrumtime值越小，越靠左。如果新插入的进程的vruntime最小，更新cfs_rq->tasks_timeline.rb_leftmost使其指向se。

check_preempt_wakeup（fair_sched_class的check_preempt_curr操作，“操作名_class名”的特例）调用update_curr并在以下两种情况下会调用resched_curr申请切换进程。

第一种，rq->curr->policy == SCHED_IDLE且p->policy != SCHED_IDLE，采用SCHED_IDLE调度策略，就类似于告诉CFS：“我并不着急”。

第二种，进程p的vruntime比rq->curr进程的vruntime小的“明显”，是否“明显”，由wakeup_preempt_entity函数决定，满足条件函数返回1，一般是判断差值是否小于

sysctl_sched_wakeup_granularity转换为虚拟时间之后的值。

task_tick_fair函数

task_tick_fair在时钟中断时被调用，调用entity_tick实现。entity_tick先调用update_curr，如果cfs_rq->nr_running>1，调用check_preempt_tick检查是否应该抢占进程，主要逻辑如下。

```
        void check_preempt_tick(struct cfs_rq *cfs_rq, struct sched_entity
*curr)
        {
            unsigned long ideal_runtime, delta_exec;
            struct sched_entity *se;
            s64 delta;

            ideal_runtime = sched_slice(cfs_rq, curr);    //1
            delta_exec = curr->sum_exec_runtime - curr->prev_sum_exec_runtime;
    //2
            if (delta_exec > ideal_runtime) {
                resched_curr(rq_of(cfs_rq));
                return;
            }
            if (delta_exec < sysctl_sched_min_granularity)
                return;

            se = __pick_first_entity(cfs_rq);    //3
            delta = curr->vruntime - se->vruntime;

            if (delta < 0)
                return;

            if (delta > ideal_runtime)
                resched_curr(rq_of(cfs_rq));
        }
```

第1步，调用sched_slice计算进程被分配的时间ideal_runtime。

sched_slice先调用__sched_period，得到的进程可以执行的时间（记为slice），该时间与进程优先级无关，只与当前的进程数有关。得到了时间后，sched_slice再调用__calc_delta将进程优先级考虑进去，进程最终获得的时间等于slice * se->load.weight / cfs_rq->load.weight。

进程的优先级越高，weight越大，获得的执行时间越多，这是CFS给高优先级进程的又一个优待。总之，优先级越高的进程，执行的机会越多，可执行的时间越久。

需要说明的是，此处se->on_rq等于1，sched_slice在task_fork_fair的时候也会被调用，计算sched_vslice，那里se->on_rq等于0。另外，sched_slice尽管考虑了进程优先级，但它仍然是实际时间。

第2步，计算进程获得了执行时间后

第2步，计算进程获得了执行时间后累计执行的时间delta_exec，其中curr->sum_exec_runtime是在update_curr时更新的，curr->prev_sum_exec_runtime是在进程被挑选执行的时候更新的（见15.5.4.3小节）。

如果进程已经用完了分配给它的时间，接下来调用resched_curr申请切换进程。

如果进程并没有用完它的时间，第3步，找到进程优先级红黑树上最左的进程，计算当前进程的vruntime和它的vruntime的差值delta。如果delta小于0，当前进程的vruntime更小，函数返回，进程继续执行。如果delta大于0，说明当前进程已经不是CFS的第一选择，但是它并没有用完自己的时间，这种情况下只要delta不大于ideal_runtime，当前进程继续执行，否则调用resched_curr申请切换进程。这么做可以防止进程频繁切换，比如在前面两个进程的vruntime接近的情况下，如果不这么做会出现二者快速交替执行的情况。

进程切换

切换的流程由__schedule函数定义，具体的调度类仅决定了回调函数部分，下面我们来分别分析dequeue_task_fair、put_prev_task_fair和pick_next_task_fair。

1.dequeue_task

dequeue_task_fair调用dequeue_entity实现，它完成如下几步。

（1）调用update_curr。

（2）如果进程当前没有占用CPU（se != cfs_rq->curr），调用__dequeue_entity从进程优先级红黑树将其删除（如果进程正在执行，它不在红黑树中，见pick_next）。

（3）更改se->on_rq为0。

（4）如果进程接下来不会睡眠（!(flags&DEQUEUE_SLEEP)），更新进程的vruntime，se->vruntime-=cfs_rq->min_vruntime。

一个从不睡眠的普通进程，被enqueue的时候，se->vruntime会加上cfs_rq->min_vruntime，被dequeue的时候，会减去cfs_rq->min_vruntime。

一加一减，看似回到原状实则不然，enqueue和dequeue的时候cfs_rq->min_vruntime可能不一样。试想一种情况，一个进程被enqueue之后始终没有得到执行，它的vruntime没有变化。过了一段时间，cfs_rq->min_vruntime变大了，再被dequeue的时候，它的vruntime就变小了，等的越久vruntime就越小。不同的cfs_rq的min_vruntime可能是不等的，如果进程从一个CPU迁移到另一个CPU上，采用这种方式可以保证一定程度的公平，因为它可以衡量进程在前一个CPU上等待的时间的多少。

2.put_prev

put_prev_task_fair调用put_prev_entity实现，它的行为取决于进程的状态，主要逻辑如下。

```
void put_prev_entity(struct cfs_rq *cfs_rq, struct sched_entity *prev)
{
    if (prev->on_rq){
        update_curr(cfs_rq);
        __enqueue_entity(cfs_rq, prev);
    }
    cfs_rq->curr = NULL;
}
```

prev->on_rq等于1还是0，取决于prev进程接下来希望继续执行还是更改状态并睡眠，如果进程是因为执行时间到了被迫让出CPU，prev->on_rq等于1，此时并不会将进程从进程优先级红黑树删除，而是调用__enqueue_entity根据它的vruntime重新插入红黑树。

进程的状态、是否on_rq和是否在红黑树上，读者需要理清这三者的关系，以免迷惑，如表15-11所示。

	正在执行	可执行（就绪）	睡　眠
on_rq	1	1	0
on Tree	0	1	0

prev之前执行过一段时间，所以它的vruntime有所增加，重新插入红黑树中时位置会发生变化，其他进程可能会得到执行机会。

3.pick_next

CFS调用pick_next_task_fair选择下一个进程，前面的讨论中已经提到了它的一部分内容，总结如下。

调用pick_next_entity选择进程优先级红黑树上最左的（cfs_rq->tasks_timeline. rb_leftmost）进程，实际的过程要复杂一些，这里跳过了cfs_rq->last、cfs_rq->next和cfs_rq->skip等的讨论。

调用set_next_entity更新cfs_rq当前执行的进程，首先如果进程在红黑树中（se->on_rq），调用__dequeue_entity将进程从红黑树中删除（正在执行的进程不在红黑树中），然后更新以下几个字段。

```
se->exec_start = rq_clock_task(rq_of(cfs_rq));
cfs_rq->curr = se;

se->prev_sum_exec_runtime = se->sum_exec_runtime;
```

idle调度类

idle_sched_class作为优先级最低的sched_class，它的使命就是没有其他进程需要执行的时候占用CPU，有进程需要执行的时候让出CPU。它不接受dequeue和enqueue操作，只负责idle进程，由rq->idle指定。pick_next_task_idle返回rq->idle，check_preempt_curr_idle，则直接调用resched_curr(rq)。

调度策略

总体来说，调度类实际上是根据优先级对任务的一个粗略划分，调度器总是从高优先级的调度类开始寻找可执行的任务。但对于同一个调度类中的多个任务，如果他们的优先级相同的话，调度器如何决定该选哪一个呢？

这个问题通过调度策略（Sched Policy）来解决，不同调度类的调度策略实现如下：

- Stop 调度类
  Stop 调度类中只有一个任务可供执行，不需要定义任何调度策略。
- DL （Deadline）调度类
  DL 只实现了一种调度策略： `SCHED_DEADLINE` , 用来调度优先级最高的用户任务。
- RT （Real-Time）
  RT 提供了两种调度策略： `SCHED_FIFO` 与 `SCHED_RR`, 对于使用 `SCHED_FIFO` 的任务，以先进先出的队列方式进行调度，在优先级一样的情况下，谁先执行的就先调度谁，除非它退出或者主动释放CPU。而对于 `SCHED_RR` 的任务，以时间片轮转的方式对相同优先级的多个进程进行处理。时间片长度为100ms。，即使一个任务一直处于可运行状态，在使用完自己的时间切片之后也会被抢占，然后被放入队列的尾巴等待下次机会。
- Fair CFS 实现了三种调度策略：
  a. 1. `SCHED_NORMAL` : 被用于绝大多数用户进程
  b. 2. `SCHED_BATCH` : 适用于没有用户交互行为的后台进程，用户对该类进程的响应时间要求不高，但对吞吐量要求较高，因此调度器会在完成所有 `SCHED_NORMAL` 的任务之后让该类任务不受打扰地跑上一段时间，这样能够最大限度地利用缓存。
  c. 3. `SCHED_IDLE` : 这类调度策略被用于系统中优先级最低的任务，只有在没有任何其他任务可运行时，调度器才会将运行该类任务。
- Idle 同 Stop 一样，Idle 调度类也没有实现调度策略，注意不要将这类调度类与 CFS 中的 `SCHED_IDLE` 混淆。

调度策略的定义如下：

```
1 #define SCHED_NORMAL 0
2 #define SCHED_FIFO 1
3 #define SCHED_RR 2
4 #define SCHED_BATCH 3
5 /* SCHED_ISO: reserved but not implemented yet */
6 #define SCHED_IDLE 5
7 #define SCHED_DEADLINE 6
```

每个进程在创建时都会指定一个调度策略，从而自动归结到某个调度类下。

我们可以通过 `/proc/<pid>/sched` 中的内容来查看进程的调度策略

进程创建

内核提供了不同的函数来满足创建进程、线程和内核线程的需求。它们都调用_do_fork函数实现，区别在于传递给函数的参数不同。

_do_fork的主要逻辑如下。

copy_process

我们跳过了参数检查、错误处理等，可以看到do_fork的逻辑并不复杂，真正完成复杂任务的是copy_process。把这段代码单独拿出来是为了强调它的返回值，这与后面讨论的一个话题有关（见14.4.1），在此加深印象。

copy_process的实现比较复杂，逻辑上可以分成多个部分，每一部分都可能是一个复杂的话题，接下来将它拆分成几节分开讨论。

copy_process的第一个参数clone_flags可以是多种标志的组合，它们在很大程度上决定了函数的行为，常用的标志如表14-7所示。

标　志	描　述
CLONE_VM	与当前进程共享 VM
CLONE_FS	共享文件系统信息
CLONE_FILES	共享打开的文件
CLONE_PARENT	与当前进程共有同样的父进程
CLONE_THREAD	与当前进程同属一个线程组，也意味着创建的是线程

(续)

标　志	描　述
CLONE_SYSVSEM	共享 sem_undo_list
CLONE_VFORK	新进程会在当前进程之前“运行”，见 do_fork 函数
CLONE_SETTLS	设置新进程的 TLS（Thread Local Storage，线程局部存储）
CLONE_PARENT_SETTID	创建新进程成功，则存储它的 id 到 parent_tidptr
CLONE_CHILD_CLEARTID	新进程退出时，将 child_tidptr 指定的内存清零
CLONE_NEWUTS	新进程拥有新的 UTS、IPC、USER、PID 和 NET 空间
CLONE_NEWIPC	
CLONE_NEWUSER	
CLONE_NEWPID	
CLONE_NEWNET	

抛开CLONE_VFORK等几个与资源管理没有直接关系的标志，其他的标志从名字上把它们分为两类：CLONE_XXX和CLONE_NEWXXX。一般在CLONE_XXX标志置1的情况下，新进程才会与当前进程共享相应的资源，CLONE_NEWXXX则相反。

dup_task_struct函数

新进程由当前进程创建，当前被作为参考模板。既然要创建进程，必然需要创建新的task_struct与之对应。copy_process在参数和权限等检查后，调用dup_task_struct创建新进程的task_struct。

复制creds

接下来copy_process的主要任务就是为tsk的字段赋值了，首先调用copy_creds，它涉及cred结构体（cred，credentials的缩写，翻译为凭证，下文以cred表示结构体和对象，其他情况使用“凭证”）。

cred结构体表示进程安全相关的上下文（context），记录着进程的uid（user）、gid（group）和capability等信息。

设置时间

task_struct有几个与时间相关的字段

表14-8 task_struct与时间有关的字段表

字　段	类　型	描　述
utime	u64	进程在用户态下经历的节拍数
utimescaled	u64	进程在用户态下经历的节拍数，以处理器的频率为刻度

字　段	类　型	描　述
stime	u64	进程在内核态下经历的节拍数
stimescaled	u64	进程在内核态下经历的节拍数，以处理器的频率为刻度
gtime	u64	以节拍数计算的虚拟 CPU 运行时间
start_time	u64	起始时间
real_start_time	u64	起始时间，将系统的睡眠时间计算在内

copy_process将新进程CPU时间清零，并设置正确的启动时间

sched_fork函数

copy_process接下来调用sched_fork函数，从函数名就可以看出来，sched_fork和进程调度有关（sched，schedule的缩写），事实也确实如此，如下。

sched_fork先调用__sched_fork初始化task_struct的se、dl和rt字段，它们都与进程调度有关，基本上都是一些清零操作

```
sched_fork
->__sched_fork
            p->on_rq                        = 0;
            p->se.on_rq                     = 0;
            p->se.exec_start                = 0;
            p->se.sum_exec_runtime          = 0;
            p->se.prev_sum_exec_runtime     = 0;
            p->se.nr_migrations             = 0;
            p->se.vruntime                  = 0;
            INIT_LIST_HEAD(&p->se.group_node);
            ...
```

然后设置了一些比较重要的一些属性：

```
sched_fork
-> p->state = TASK_NEW;   //设置进程初始化状态
  p->prio = current->normal_prio;  //进程的动态优先级设置
   /*
  !* Revert to default priority/policy on fork if requested.
  !*/
  if (unlikely(p->sched_reset_on_fork)) {
          if (task_has_dl_policy(p) || task_has_rt_policy(p)) {
                  p->policy = SCHED_NORMAL;                //调度策略
                  p->static_prio = NICE_TO_PRIO(0);                  //静态优先级设
                  p->rt_priority = 0;
          } else if (PRIO_TO_NICE(p->static_prio) < 0)
                  p->static_prio = NICE_TO_PRIO(0);

          p->prio = p->normal_prio = __normal_prio(p);
          set_load_weight(p, false);    //设置进程权重
            ...
  }
  if (dl_prio(p->prio))
          return -EAGAIN;
  else if (rt_prio(p->prio))
          p->sched_class = &rt_sched_class;
  else
          p->sched_class = &fair_sched_class;     //设置调度类为cfs
  __set_task_cpu(p, smp_processor_id());       //设置 进程运行的cpu为当前cpu
  if (p->sched_class->task_fork)
          p->sched_class->task_fork(p);        //执行调度类的task_fork方法即是task_f

#if defined(CONFIG_SMP)
         p->on_cpu = 0;
#endif
         init_task_preempt_count(p);     //初始化抢占计数器
```

将新进程的状态置为TASK_NEW。

第2步，如果task_struct的sched_reset_on_fork字段为1，需要重置（reset）新进程的优先级和调度策略等字段为默认值。sched_reset_on_fork的值是从当前进程复制来的，也就是说如果一个进程的sched_reset_on_fork为1，由它创建的新进程都会经历重置操作。p->sched_reset_on_fork = 0表示新进程不会继续重置由它创建的进程。

第3步，选择新进程的sched_class，并调用它的task_fork回调函数。设置虚拟运行时间等（注意在task_fork_fair中会将设置的vruntime减去当前cpu运行cfs队列的最小min_vruntime，唤醒的时候会加上所在cpu运行队列的min_vruntime）

第4步，__set_task_cpu建立新进程和CPU之间的关系，设置task_struct的cpu字段。

copy_semundo

semundo与进程通信有关，在此不必深究，我们会在进程通信的章节详细讨论，此处只关心copy相关的逻辑，具体如下。

```
int copy_semundo(unsigned long clone_flags, struct task_struct *tsk)
{
    struct sem_undo_list *undo_list;
    int error;

    if (clone_flags & CLONE_SYSVSEM) {


        error = get_undo_list(&undo_list);
        refcount_inc(&undo_list->refcnt);
        tsk->sysvsem.undo_list = undo_list;
    } else
        tsk->sysvsem.undo_list = NULL;

    return 0;
}
```

果clone_flags的CLONE_SYSVSEM标志被置位，新进程与当前进程共享sem_undo_list，先调用get_undo_list获取undo_list，get_undo_list会先判断当前进程的undo_list是否为空，为空则申请一个新的undo_list赋值给当前进程并返回，否则直接返回。得到了undo_list后，赋值给新进程，达到共享的目的。

如果CLONE_SYSVSEM标志没有置位，直接将新进程的undo_list置为NULL。

最后，线程并不具备独立的sem_undo_list，所以创建线程的时候CLONE_SYSVSEM是被置位的。

copy_files

copy_files涉及进程的文件管理，行为同样与clone_flags的值有关。

如果clone_flags的CLONE_FILES标志被置位，新进程与当前进程共享files_struct，增加引用计数后直接返回。

CLONE_FILES没有被置位的情况下，copy_files调用dup_fd为新进程创建files_struct并复制当前值为其赋值。

一句话总结就是，新线程共享当前进程的文件信息，新进程复制当前进程的文件信息。共享和复制类似于传址与传值，共享意味着不独立，修改对彼此可见，类似函数传址；复制表示此刻相同，但此后彼此独立，互不干涉，类似函数传值。

copy_fs

copy_files复制的是文件信息，copy_fs则复制文件系统的信息。

与copy_files的逻辑类似，如果clone_flags的CLONE_FS标志被置位，新进程与当前进程共享fs_struct，增加引用计数后直接返回。

fs_struct表示进程与文件系统相关的信息，由task_struct的fs字段表示，它的主要字段如表14-11所示。

表14-11 fs_struct字段表

字　段	类　型	描　述
users	int	引用计数
in_exec	int	进程是否在 load 可执行文件
root	path	进程的 root 路径

(续)

字　段	类　型	描　述
pwd	pwd	进程的当前工作目录路径

如果CLONE_FS标志没有被置位，则调用copy_fs_struct函数创建新的fs_struct并复制old_fs的值给它，最后把它赋值给当前进程task_struct的fs字段。创建进程需要复制多种资源，为了节省篇幅，从本节开始，不再罗列类似逻辑的简单代码。

复制sighand和signal

copy_sighand和copy_signal，看名字就可以知道它们和信号处理有关。前者涉及sighand_struct结构体，由tast_struct的sighand字段指向，sighand可以理解为signal handler。copy_signal涉及signal_struct结构体，由tast_struct的signal字段指向，表示进程当前的信号信息。

我们会在信号处理一章会专门讨论这两个结构体，此处不扩展讨论，本节只需要掌握copy的逻辑。

前面三个copy，主要分为两种方式。如果CLONE_XXX标志被置位，资源会被共享，复制操作并不会发生。如果CLONE_XXX标志没有被置位，copy_semundo会给新进程的相关字段赋初值，copy_files和copy_fs则会复制当前进程的值给新进程的相关字段，也就是说第一种方式是重置，第二种方式是复制。

采取哪种方式，取决于逻辑需要。

copy_sighand和copy_signal，前者采用的是复制，后者采用的是重置。从逻辑上是可以讲通的，sighand表示进程处理它的信号的手段，新进程复制当前进程的手段符合大多数需求；而signal是当前进程的信号信息，这些信号并不是发送给新进程的，新进程立起炉灶为妙。

最后，copy_sighand和copy_signal检查的标志分别为CLONE_SIGHAND和CLONE_THREAD，标志置1的情况下，不会复制或重置；标志没有置位的情况下，前者复制，后者重置。

copy_mm

进程与内存管理之间的故事，由copy_mm开启，又一个有趣的话题，代码如下。

```
1 static int copy_mm(unsigned long clone_flags, struct task_struct *tsk)
2 {
3         struct mm_struct *mm, *oldmm;
```

```
4         int retval;
5
6         tsk->min_flt = tsk->maj_flt = 0;
7         tsk->nvcsw = tsk->nivcsw = 0;
8 #ifdef CONFIG_DETECT_HUNG_TASK
9         tsk->last_switch_count = tsk->nvcsw + tsk->nivcsw;
10        tsk->last_switch_time = 0;
11 #endif
12
13        tsk->mm = NULL;
14        tsk->active_mm = NULL;
15
16        /*
17         * Are we cloning a kernel thread?
18         *
19         * We need to steal a active VM for that..
20         */
21        oldmm = current->mm;
22        if (!oldmm)
23                return 0;
24
25        /* initialize the new vmacache entries */
26        vmacache_flush(tsk);
27
28        if (clone_flags & CLONE_VM) {
29                mmget(oldmm);
30                mm = oldmm;
31                goto good_mm;
32        }
33
34        retval = -ENOMEM;
35        mm = dup_mm(tsk);
36        if (!mm)
37                goto fail_nomem;
38
39 good_mm:
40        tsk->mm = mm;
41        tsk->active_mm = mm;
42        return 0;
43
44 fail_nomem:
45        return retval;
46 }
```

task_struct的mm和active_mm两个字段与内存管理有关，它们都是指向mm_struct结构体的指针，前者表示进程所管理的内存的信息，后者表示进程当前使用的内存的信息。何解？所属不同，mm管理

的内存至少有一部分是属于进程本身的；active_mm，进程使用的内存，可能不属于进程。二者有可能不一致。