新创建的进程如何被调度的

1 概述

从kernel/fork.c里面,我们能够看到,无论是userspace还是kernel space在创建进程的时候最后的调用路径都是相同的,最后都走到_do_fork函数,我们看看源码:

```
/* For compatibility with architectures that call do_fork directly rather than
 * using the syscall entry points below. */
long do_fork(unsigned long clone_flags,
          unsigned long stack_start,
          unsigned long stack_size,
          int __user *parent_tidptr,
          int __user *child_tidptr)
{
    return _do_fork(clone_flags, stack_start, stack_size,
            parent_tidptr, child_tidptr, 0);
}
#endif

/*
 * Create a kernel thread.
 *//*创建内核进程,比如在start_kernel-->rest_init里面创建了2号进kthreadd,*/
pid_t kernel_thread(int (*fn)(void *), void *arg, unsigned long flags)
{
    return _do_fork(flags|CLONE_VM|CLONE_UNTRACED, (unsigned long)fn,
        (unsigned long)arg, NULL, NULL, 0);
}
/*下面是提供userspace调用的.fork/vfork*/
#ifdef __ARCH_WANT_SYS_FORK
SYSCALL_DEFINE0(fork)
{
#ifdef CONFIG_MMU
    return _do_fork(SIGCHLD, 0, 0, NULL, NULL, 0);
```

```
#else
    /* can not support in nommu mode */
    return -EINVAL;
#endif
}
#endif

#ifdef __ARCH_WANT_SYS_VFORK
SYSCALL_DEFINE0(vfork)
{
    return _do_fork(CLONE_VFORK | CLONE_VM | SIGCHLD, 0,
            0, NULL, NULL, 0);
}
#endif
/*下面是clone相关的系统调用.*/
#ifdef __ARCH_WANT_SYS_CLONE
#ifdef CONFIG_CLONE_BACKWARDS
SYSCALL_DEFINE5(clone, unsigned long, clone_flags, unsigned long, newsp,
         int __user *, parent_tidptr,
         unsigned long, tls,
         int __user *, child_tidptr)
#elif defined(CONFIG_CLONE_BACKWARDS2)
SYSCALL_DEFINE5(clone, unsigned long, newsp, unsigned long, clone_flags,
         int __user *, parent_tidptr,
         int __user *, child_tidptr,
         unsigned long, tls)
#elif defined(CONFIG_CLONE_BACKWARDS3)
SYSCALL_DEFINE6(clone, unsigned long, clone_flags, unsigned long, newsp,
        int, stack_size,
        int __user *, parent_tidptr,
        int __user *, child_tidptr,
        unsigned long, tls)
#else
SYSCALL_DEFINE5(clone, unsigned long, clone_flags, unsigned long, newsp,
         int __user *, parent_tidptr,
         int __user *, child_tidptr,
         unsigned long, tls)
#endif
{
    return _do_fork(clone_flags, newsp, 0, parent_tidptr, child_tidptr,
tls);
}
#endif
```

他们最终的调用函数都是_do_fork函数,至于userspace通过何种方式陷入内核创建进程的,以后会详细讲解,仅仅看调度相关的.

我们看下_do_fork函数源码:

```
/*
 *  Ok, this is the main fork-routine.
 *
 * It copies the process, and if successful kick-starts
 * it and waits for it to finish using the VM if required.
 */
long _do_fork(unsigned long clone_flags,
          unsigned long stack_start,
```

```
            unsigned long stack_size,
            int __user *parent_tidptr,
            int __user *child_tidptr,
            unsigned long tls)
{
    struct task_struct *p;
    int trace = 0;
    long nr;

    /*
     * Determine whether and which event to report to ptracer.  When
     * called from kernel_thread or CLONE_UNTRACED is explicitly
     * requested, no event is reported; otherwise, report if the event
     * for the type of forking is enabled.
     */
    if (!(clone_flags & CLONE_UNTRACED)) {
        if (clone_flags & CLONE_VFORK)
            trace = PTRACE_EVENT_VFORK;
        else if ((clone_flags & CSIGNAL) != SIGCHLD)
            trace = PTRACE_EVENT_CLONE;
        else
            trace = PTRACE_EVENT_FORK;

        if (likely(!ptrace_event_enabled(current, trace)))
            trace = 0;
    }
    /*创建进程的关键性函数,里面设置填充了若干新创建的进程task_struct结构体,同时调用了sched_fork函数,设置新创建进程相关的调度信息,比权重和vruntime等信息*/
    p = copy_process(clone_flags, stack_start, stack_size,
             child_tidptr, NULL, trace, tls, NUMA_NO_NODE);
    /*
     * Do this prior waking up the new thread - the thread pointer
     * might get invalid after that point, if the thread exits quickly.
     */
    if (!IS_ERR(p)) {
        struct completion vfork;
        struct pid *pid;

        trace_sched_process_fork(current, p);

        pid = get_task_pid(p, PIDTYPE_PID);
        nr = pid_vnr(pid);

        if (clone_flags & CLONE_PARENT_SETTID)
            put_user(nr, parent_tidptr);

        if (clone_flags & CLONE_VFORK) {
            p->vfork_done = &vfork;
            init_completion(&vfork);
            get_task_struct(p);
        }
        /*将创建的进程加入到对应的rq中,并进程调度处理.*/
        wake_up_new_task(p);

```

```
            /* forking complete and child started to run, tell ptracer */
            if (unlikely(trace))
                ptrace_event_pid(trace, pid);

            if (clone_flags & CLONE_VFORK) {
                if (!wait_for_vfork_done(p, &vfork))
                    ptrace_event_pid(PTRACE_EVENT_VFORK_DONE, pid);
            }

            put_pid(pid);
        } else {
            nr = PTR_ERR(p);
        }
        return nr;
    }
```

下面对两个核心函数的分析:

2 对核心函数sched_fork分析

```
/*
 * This creates a new process as a copy of the old one,
 * but does not actually start it yet.
 *
 * It copies the registers, and all the appropriate
 * parts of the process environment (as per the clone
 * flags). The actual kick-off is left to the caller.
 */
static struct task_struct *copy_process(unsigned long clone_flags,
                    unsigned long stack_start,
                    unsigned long stack_size,
                    int __user *child_tidptr,
                    struct pid *pid,
                    int trace,
                    unsigned long tls,
                    int node)
{
............
        /* Perform scheduler related setup. Assign this task to a CPU. */
    retval = sched_fork(clone_flags, p);
............
}
/*
 * fork()/clone()-time setup:
 */ /*sched_fork函数的具体实现*/
int sched_fork(unsigned long clone_flags, struct task_struct *p)
{
    unsigned long flags;
    /*禁止抢占并获得当前运行此函数的cpu id*/
    int cpu = get_cpu();

    __sched_fork(clone_flags, p);
```

```
        /*
         * We mark the process as NEW here. This guarantees that
         * nobody will actually run it, and a signal or other external
         * event cannot wake it up and insert it on the runqueue either.
         */
        /*设置task的状态为TASK_NEW,随着task的不断变化,其state会不断的变化,并且
        调度器会根据这些不同的状态做出不同的行为*/
        p->state = TASK_NEW;

        /*
         * Make sure we do not leak PI boosting priority to the child.
         */
        /*子进程继承父进程的优先级*/
        p->prio = current->normal_prio;

        /*
         * Revert to default priority/policy on fork if requested.
         */   /*如果需要,重置这个进程的优先级/权重和policy*/
        if (unlikely(p->sched_reset_on_fork)) {
            if (task_has_dl_policy(p) || task_has_rt_policy(p)) {
                p->policy = SCHED_NORMAL;
                p->static_prio = NICE_TO_PRIO(0);
                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);

            /*
             * We don't need the reset flag anymore after the fork. It has
             * fulfilled its duty:
             */
            p->sched_reset_on_fork = 0;
        }
        /*根据进程的优先级,选择调度类.*/
        if (dl_prio(p->prio)) {
            put_cpu();
            return -EAGAIN;
        } else if (rt_prio(p->prio)) {
            p->sched_class = &rt_sched_class;
        } else {
            p->sched_class = &fair_sched_class;
        }
        /*初始化这个task作为一个调度实体的 struct sched_entity 里面struct sched_avg
        结构体函数,比如设置初始化的load的更新时间,load_sum,util_sum,util_avg,
        load_avg,他们的数值会在PELT算法里面即update_load_avg函数里面进行更新.*/
        init_entity_runnable_average(&p->se);

        /*
         * The child is not yet in the pid-hash so no cgroup attach races,
         * and the cgroup is pinned to this child due to cgroup_fork()
         * is ran before sched_fork().
         *
```

```
	 * Silence PROVE_RCU.
	 */
	raw_spin_lock_irqsave(&p->pi_lock, flags);
	/*
	 * We're setting the cpu for the first time, we don't migrate,
	 * so use __set_task_cpu().
	 *//*设置进程的cpu,以及对应的的cfs_rq,task_group等信息*/
	__set_task_cpu(p, cpu);
	/*调用对应的调度类的task_fork函数*/
	if (p->sched_class->task_fork)
		p->sched_class->task_fork(p);
	raw_spin_unlock_irqrestore(&p->pi_lock, flags);

#ifdef CONFIG_SCHED_INFO
	if (likely(sched_info_on()))
		memset(&p->sched_info, 0, sizeof(p->sched_info));
#endif
#if defined(CONFIG_SMP)
	p->on_cpu = 0;
#endif
	/*初始化task_struct结构体的抢占计数器的初始值*/
	init_task_preempt_count(p);
#ifdef CONFIG_SMP
	plist_node_init(&p->pushable_tasks, MAX_PRIO);
	RB_CLEAR_NODE(&p->pushable_dl_tasks);
#endif
	/*enable 抢占*/
	put_cpu();
	return 0;
}
```

对于sched_fork里面几个关键函数的分析如下

2.1 __sched_fork函数分析;

```
/*
 * Perform scheduler related setup for a newly forked process p.
 * p is forked by current.
 *
 * __sched_fork() is basic setup used by init_idle() too:
 */
static void __sched_fork(unsigned long clone_flags, struct task_struct *p)
{  /*初始化on_rq,即是否在rq里面*/
	p->on_rq			= 0;
	/*初始化新创建的进程作为调度实体的数据结构*/
	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;
/*调度实体的vruntime,根据这个数值cfs调度算法将进程组成rb tree*/
	p->se.vruntime			= 0;
   /*WALT算法标记task休眠的时间点*/
#ifdef CONFIG_SCHED_WALT
```

```
    p->last_sleep_ts      = 0;
#endif
    /*初始化se的group节点*/
    INIT_LIST_HEAD(&p->se.group_node);
    /*根据WALT算法,即通过若干个窗口类的进程的runnable时间,来调节cpu的频率.下面这个
    函数是初始化一个新创建的进程的struct task_struct--->struct ravg结构体里面
    demand和sum_history[8]数值,demand是初始化当前task的runnable时间,即task load
     sum_history[8]是作为若干个窗口保存的数值,并且每个窗口都会进行update.具体怎么
    update详细查看:https://blog.csdn.net/wukongmingjing/article/details/81633225*/
    walt_init_new_task_load(p);

#ifdef CONFIG_FAIR_GROUP_SCHED
    p->se.cfs_rq          = NULL;
#endif

#ifdef CONFIG_SCHEDSTATS
    memset(&p->se.statistics, 0, sizeof(p->se.statistics));
#endif

    RB_CLEAR_NODE(&p->dl.rb_node);
    init_dl_task_timer(&p->dl);
    __dl_clear_params(p);

    INIT_LIST_HEAD(&p->rt.run_list);
    /*初始化抢占通知*/
#ifdef CONFIG_PREEMPT_NOTIFIERS
    INIT_HLIST_HEAD(&p->preempt_notifiers);
#endif

#ifdef CONFIG_NUMA_BALANCING
    if (p->mm && atomic_read(&p->mm->mm_users) == 1) {
        p->mm->numa_next_scan = jiffies + msecs_to_jiffies(sysctl_numa_balancing_scan_delay);
        p->mm->numa_scan_seq = 0;
    }

    if (clone_flags & CLONE_VM)
        p->numa_preferred_nid = current->numa_preferred_nid;
    else
        p->numa_preferred_nid = -1;

    p->node_stamp = 0ULL;
    p->numa_scan_seq = p->mm ? p->mm->numa_scan_seq : 0;
    p->numa_scan_period = sysctl_numa_balancing_scan_delay;
    p->numa_work.next = &p->numa_work;
    p->numa_faults = NULL;
    p->last_task_numa_placement = 0;
    p->last_sum_exec_runtime = 0;

    p->numa_group = NULL;
#endif /* CONFIG_NUMA_BALANCING */
}

```

```
void walt_init_new_task_load(struct task_struct *p)
{
    int i;
    u32 init_load_windows =
            div64_u64((u64)sysctl_sched_walt_init_task_load_pct *
                          (u64)walt_ravg_window, 100);
    u32 init_load_pct = current->init_load_pct;

    p->init_load_pct = 0;
    memset(&p->ravg, 0, sizeof(struct ravg));

    if (init_load_pct) {
        init_load_windows = div64_u64((u64)init_load_pct *
              (u64)walt_ravg_window, 100);
    }

    p->ravg.demand = init_load_windows;
    for (i = 0; i < RAVG_HIST_SIZE_MAX; ++i)
        p->ravg.sum_history[i] = init_load_windows;
}

```

2.2 set_load_weight，进程权重函数分析

```
static void set_load_weight(struct task_struct *p)
{    /*获取task的优先级*/
     int prio = p->static_prio - MAX_RT_PRIO;
     struct load_weight *load = &p->se.load;

     /*
      * SCHED_IDLE tasks get minimal weight:
      *//*设置idle thread的优先级权重*/
     if (idle_policy(p->policy)) {
         load->weight = scale_load(WEIGHT_IDLEPRIO);
         load->inv_weight = WMULT_IDLEPRIO;
         return;
     }
     /*设置正常进程优先级的权重,*/
     load->weight = scale_load(prio_to_weight[prio]);
     /*进程权重的倒数,数值为2^32/weight*/
     load->inv_weight = prio_to_wmult[prio];
     /*上面两个数值都可以通过查表获取的*/
}
/*
 * Nice levels are multiplicative, with a gentle 10% change for every
 * nice level changed. I.e. when a CPU-bound task goes from nice 0 to
 * nice 1, it will get ~10% less CPU time than another CPU-bound task
 * that remained on nice 0.
 *
 * The "10% effect" is relative and cumulative: from _any_ nice level,
 * if you go up 1 level, it's -10% CPU usage, if you go down 1 level
```

```
 * it's +10% CPU usage. (to achieve that we use a multiplier of 1.25.
 * If a task goes up by ~10% and another task goes down by ~10% then
 * the relative distance between them is ~25%.)
 */
static const int 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,
};

/*
 * Inverse (2^32/x) values of the prio_to_weight[] array, precalculated.
 *
 * In cases where the weight does not change often, we can use the
 * precalculated inverse to speed up arithmetics by turning divisions
 * into multiplications:
 *//*2^32/weight*  : 2^32=4294967296 ,2^32/NICE_0_LOAD=2^32/1024=4194304
   符合预期*/
static const u32 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,
};

```

2.3 初始化调度实体负载追踪结构体(struct sched_avg)

```
/* Give new sched_entity start runnable values to heavy its load in infant time */
void init_entity_runnable_average(struct sched_entity *se)
{
     /*获取新进程调度实体的由于计算se util和load的结构体，用来做初始化动作*/
     struct sched_avg *sa = &se->avg;
     /*初始化load的更新时间*/
     sa->last_update_time = 0;
     /*
      * sched_avg's period_contrib should be strictly less then 1024, so
      * we give it 1023 to make sure it is almost a period (1024us), and
      * will definitely be update (after enqueue).
```

```
     */
    sa->period_contrib = 1023;
    /*
     * Tasks are intialized with full load to be seen as heavy tasks until
     * they get a chance to stabilize to their real load level.
     * Group entities are intialized with zero load to reflect the fact that
     * nothing has been attached to the task group yet.
     */
    if (entity_is_task(se))
        sa->load_avg = scale_load_down(se->load.weight);
    sa->load_sum = sa->load_avg * LOAD_AVG_MAX;
    /*
     * In previous Android versions, we used to have:
     *  sa->util_avg = scale_load_down(SCHED_LOAD_SCALE);
     *  sa->util_sum = sa->util_avg * LOAD_AVG_MAX;
     * However, that functionality has been moved to enqueue.
     * It is unclear if we should restore this in enqueue.
     */
    /*
     * At this point, util_avg won't be used in select_task_rq_fair anyway
     */
    sa->util_avg = 0;
    sa->util_sum = 0;
    /* when this task enqueue'ed, it will contribute to its cfs_rq's load_avg */
}
```

2.4 __set_task_cpu(p, cpu)

```
/* Change a task's cfs_rq and parent entity if it moves across CPUs/groups
*/
static inline void set_task_rq(struct task_struct *p, unsigned int cpu)
{
#if defined(CONFIG_FAIR_GROUP_SCHED) || defined(CONFIG_RT_GROUP_SCHED)
    struct task_group *tg = task_group(p);
#endif

#ifdef CONFIG_FAIR_GROUP_SCHED
    set_task_rq_fair(&p->se, p->se.cfs_rq, tg->cfs_rq[cpu]);
    p->se.cfs_rq = tg->cfs_rq[cpu];
    p->se.parent = tg->se[cpu];
#endif

#ifdef CONFIG_RT_GROUP_SCHED
    p->rt.rt_rq  = tg->rt_rq[cpu];
    p->rt.parent = tg->rt_se[cpu];
#endif
}

#else /* CONFIG_CGROUP_SCHED */

static inline void set_task_rq(struct task_struct *p, unsigned int cpu) { }
static inline struct task_group *task_group(struct task_struct *p)
{
    return NULL;
}

#endif /* CONFIG_CGROUP_SCHED */

static inline void __set_task_cpu(struct task_struct *p, unsigned int cpu)
{   /*设置进程所属的进程组的cpu上,即在进程组里面**cfs_rq,**se所属的cpu上*/
    set_task_rq(p, cpu);
#ifdef CONFIG_SMP
    /*
     * After ->cpu is set up to a new value, task_rq_lock(p, ...) can be
     * successfuly executed on another CPU. We must ensure that updates of
     * per-task data have been completed by this moment.
     */
    smp_wmb();
  /*设置进程所属cpu为当前cpu*/
#ifdef CONFIG_THREAD_INFO_IN_TASK
    p->cpu = cpu;
#else
    task_thread_info(p)->cpu = cpu;
#endif
    p->wake_cpu = cpu;
#endif
}
```

2.5 task_fork_fair函数分析

```
/*
 * called on fork with the child task as argument from the parent's context
 *  - child not yet on the tasklist
 *  - preemption disabled
 */
static 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();

    raw_spin_lock(&rq->lock);
    /*更新rq的clock*/
    update_rq_clock(rq);
    /*获取当前进程的cfs_rq*/
    cfs_rq = task_cfs_rq(current);
    /*获取当前进程的调度实体*/
    curr = cfs_rq->curr;
    /*如果当前进程的调度实体存在,则设置新进程的调度实体的vruntime为
   父进程的vruntime*/
    if (curr) {
        /*更加权重重新调整当前进程的vruntime*/
        update_curr(cfs_rq);
        se->vruntime = curr->vruntime;
    }
    /*调整新进程的vruntime*/
    place_entity(cfs_rq, se, 1);
    /*如果当前进程vruntime比新进程的vruntime要小,则设置当前进程
    调度标志,在中断退出或者异常退出的时候会检查这个标记*/
    if (sysctl_sched_child_runs_first && curr && entity_before(curr, se)) {
        /*
         * Upon rescheduling, sched_class::put_prev_task() will place
         * 'current' within the tree based on its new key value.
         */
        swap(curr->vruntime, se->vruntime);
        resched_curr(rq);
    }
    /*新进程的vruntime减去当前cpu的cfs_rq的最小vruntime,目的是你
   不知道这个新进程最后会在哪个cpu上执行,如果确定了,则会重新加上对应
    cpu cfs_rq的最小vruntime很巧妙.任何进程的vruntime时间都是
   所在cfs_rq最小vruntime基础上累加的数值*/
    se->vruntime -= cfs_rq->min_vruntime;
    raw_spin_unlock(&rq->lock);
}
```

至此shced_fork全部分析完毕.

3 对核心函数wake_up_new_task分析

```
/*
 * wake_up_new_task - wake up a newly created task for the first time.
 *
 * This function will do some initial scheduler statistics housekeeping
 * that must be done for every newly created context, then puts the task
 * on the runqueue and wakes it.
 */
void wake_up_new_task(struct task_struct *p)
{
    unsigned long flags;
    struct rq *rq;

    raw_spin_lock_irqsave(&p->pi_lock, flags);
    /*OK ,新进程的状态标记为running了,即可以被调度器调度了*/
    p->state = TASK_RUNNING;
    /*再次初始化struct task_struct---> struct ravg里面的成员变量*/
    walt_init_new_task_load(p);
    /*再次初始化新进程调度实体的load/util*/
    /* Initialize new task's runnable average */
    init_entity_runnable_average(&p->se);
#ifdef CONFIG_SMP
    /*
     * Fork balancing, do it here and not earlier because:
     *  - cpus_allowed can change in the fork path
     *  - any previously selected cpu might disappear through hotplug
     *
     * Use __set_task_cpu() to avoid calling sched_class::migrate_task_rq,
     * as we're not fully set-up yet.
     *//*选择一个合适的cpu,并设置此进程balance标记SD_BALANCE_FORK,即在fork/clone
     时候,根据当前系统状态,将创建的进程balance到合适的cpu上,核心函数*/
    __set_task_cpu(p, select_task_rq(p, task_cpu(p), SD_BALANCE_FORK, 0,
1));
#endif
    /*获取当前进程的rq*/
    rq = __task_rq_lock(p);
    /*更新rq的时间*/
    update_rq_clock(rq);
    /*调整新进程的调度实体的util数值,否则为0 的话会导致整个rq的util变的很小,需要调整*/
    post_init_entity_util_avg(&p->se);
    /*更新新进行在WALT窗口里面的运行时间,即更新struct task_struct --->
    struct ravg 成员变量 mark_start数值为当前时间.在WLAT文章中有详细讲解*/
    walt_mark_task_starting(p);
    /*新进程入队,核心函数*/
    activate_task(rq, p, ENQUEUE_WAKEUP_NEW);
    /*新进程已经在rq里面,可以运行*/
    p->on_rq = TASK_ON_RQ_QUEUED;
    trace_sched_wakeup_new(p);
```

```
    /*抢占check*/
    check_preempt_curr(rq, p, WF_FORK);
#ifdef CONFIG_SMP
    if (p->sched_class->task_woken) {
        /*
         * Nothing relies on rq->lock after this, so its fine to
         * drop it.
         */
        lockdep_unpin_lock(&rq->lock);
        p->sched_class->task_woken(rq, p);
        lockdep_pin_lock(&rq->lock);
    }
#endif
    task_rq_unlock(rq, p, &flags);
}
```

3.1 对核心函数activate_task分析

```
void activate_task(struct rq *rq, struct task_struct *p, int flags)
{   /*check 进程的状态,并对rq里面处于uninterruptible的进程数量
    nr_uninterruptible--*/
    if (task_contributes_to_load(p))
        rq->nr_uninterruptible--;
    /*入队的核心函数*/
    enqueue_task(rq, p, flags);
}

#define task_contributes_to_load(task)  \
                ((task->state & TASK_UNINTERRUPTIBLE) != 0 && \
                 (task->flags & PF_FROZEN) == 0 && \
                 (task->state & TASK_NOLOAD) == 0)

static inline void enqueue_task(struct rq *rq, struct task_struct *p, int
flags)
{
    update_rq_clock(rq);
    if (!(flags & ENQUEUE_RESTORE))
        sched_info_queued(rq, p);
#ifdef CONFIG_INTEL_DWS
    if (sched_feat(INTEL_DWS))
        update_rq_runnable_task_avg(rq);
#endif
    /*调用对应调度类的入队函数*/
    p->sched_class->enqueue_task(rq, p, flags);
}

/*
 * The enqueue_task method is called before nr_running is
 * increased. Here we update the fair scheduling stats and
 * then put the task into the rbtree:
 */  /*CFS调度算法入队函数*/
static void
enqueue_task_fair(struct rq *rq, struct task_struct *p, int flags)
{
    struct cfs_rq *cfs_rq;
    struct sched_entity *se = &p->se;
```

```
#ifdef CONFIG_SMP
    int task_new = flags & ENQUEUE_WAKEUP_NEW;
#endif
    /*增加rq的runnable time,即当前的rq的runnable time+新进程的p->ravg.demand
    数值*/
    walt_inc_cumulative_runnable_avg(rq, p);

    /*
     * Update SchedTune accounting.
     *
     * We do it before updating the CPU capacity to ensure the
     * boost value of the current task is accounted for in the
     * selection of the OPP.
     *
     * We do it also in the case where we enqueue a throttled task;
     * we could argue that a throttled task should not boost a CPU,
     * however:
     * a) properly implementing CPU boosting considering throttled
     *    tasks will increase a lot the complexity of the solution
     * b) it's not easy to quantify the benefits introduced by
     *    such a more complex solution.
     * Thus, for the time being we go for the simple solution and boost
     * also for throttled RQs.
     */
    schedtune_enqueue_task(p, cpu_of(rq));

    /*
     * If in_iowait is set, the code below may not trigger any cpufreq
     * utilization updates, so do it here explicitly with the IOWAIT flag
     * passed.
     *//*如果新进程是一个iowait的进程,则进行频率调整,根据iowait boost freq*/
    if (p->in_iowait)
        cpufreq_update_util(rq, SCHED_CPUFREQ_IOWAIT);
  /* 这里是一个迭代, 我们知道, 进程有可能是处于一个进程组中的, 所以当这个处于进程
     组中的进程加入到该进程组的队列中时, 要对此队列向上迭代 */
    for_each_sched_entity(se) {
        /*新创建的进程on_rq为0,只有入队之后,其数值才会被赋值为TASK_ON_RQ_QUEUED*/
        if (se->on_rq)
            break;
       /* 如果不是CONFIG_FAIR_GROUP_SCHED, 获取其所在CPU的rq运行队列的cfs_rq
       运行队列如果是CONFIG_FAIR_GROUP_SCHED, 获取其所在的cfs_rq运行队列*/
        cfs_rq = cfs_rq_of(se);
        walt_inc_cfs_cumulative_runnable_avg(cfs_rq, p);
        /*入队的核心函数*/
        enqueue_entity(cfs_rq, se, flags);

        /*
         * end evaluation on encountering a throttled cfs_rq
         *
         * note: in the case of encountering a throttled cfs_rq we will
         * post the final h_nr_running increment below.
         *//*已经throttle,则退出迭代*/
        if (cfs_rq_throttled(cfs_rq))
            break;
```

```
            cfs_rq->h_nr_running++;
            /*将新创建的进程状态修改为ENQUEUE_WAKEUP状态*/
            flags = ENQUEUE_WAKEUP;
        }
        /* 只有se不处于队列中或者cfs_rq_throttled(cfs_rq)返回真才会运行这个循环 */
        for_each_sched_entity(se) {
            cfs_rq = cfs_rq_of(se);
            cfs_rq->h_nr_running++;
            walt_inc_cfs_cumulative_runnable_avg(cfs_rq, p);

            if (cfs_rq_throttled(cfs_rq))
                break;

            update_load_avg(se, UPDATE_TG);
            update_cfs_shares(se);
        }
        /*增加rq的nr_running的数值*/
        if (!se)
            add_nr_running(rq, 1);

    #ifdef CONFIG_SMP
        if (!se) {
            struct sched_domain *sd;

            rcu_read_lock();
            sd = rcu_dereference(rq->sd);
            if (!task_new && sd) {
                if (cpu_overutilized(rq->cpu))
                    set_sd_overutilized(sd);
                if (rq->misfit_task && sd->parent)
                    set_sd_overutilized(sd->parent);
            }
            rcu_read_unlock();
        }

    #endif /* CONFIG_SMP */
        hrtick_update(rq);
    }
```

3.1.1 入队函数enqueue_entity分析

```
    static void
    enqueue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se, int flags)
    {
        /*
         * Update the normalized vruntime before updating min_vruntime
         * through calling update_curr().
         */
        /*在task_fork_fair函数里面,对新进程的vruntime减去了对应cpu的cfs rq的最小
        vruntime,我们看到新创建进程的flags为ENQUEUE_WAKEUP_NEW=0x20
        ENQUEUE_WAKEUP=0x01,ENQUEUE_WAKING=0x04,
        所以!(0x20 & 0x01) || (0x20 & 0x04) 为true.*/
        if (!(flags & ENQUEUE_WAKEUP) || (flags & ENQUEUE_WAKING))
            se->vruntime += cfs_rq->min_vruntime;

```

```
        /*
         * Update run-time statistics of the 'current'.
         */
        /*更新cfs_rq调度实体的vruntime和相关调度的统计信息*/
        update_curr(cfs_rq);
        /*对新进程的调度实体进行util/load进行衰减,根据PELT算法*/
        update_load_avg(se, UPDATE_TG);
        /*更新cfs_rqrunnable_load_sum/avg负载信息已经struct sched_entity →
        struct sched_avg成员变量数值累加到整个struct cfs_rq-->struct sched_avg上去并
        触发频率的调整.*/
        enqueue_entity_load_avg(cfs_rq, se);
        update_cfs_shares(se);
        account_entity_enqueue(cfs_rq, se);
        /*新创建进程flags为ENQUEUE_WAKEUP_NEW*/
        if (flags & ENQUEUE_WAKEUP) {
            place_entity(cfs_rq, se, 0);
            enqueue_sleeper(cfs_rq, se);
        }
        /*更新调度相关状态和统计信息*/
        update_stats_enqueue(cfs_rq, se);
        check_spread(cfs_rq, se);
        /*如果当前调度实体不是cfs_rq当前的调度实体,则将新进程的调度实体插入rb tree中,根据
        vruntime的大小加入rb tree*/
        if (se != cfs_rq->curr)
            __enqueue_entity(cfs_rq, se);
        /*新进程在rq中*/
        se->on_rq = 1;

        if (cfs_rq->nr_running == 1) {
            list_add_leaf_cfs_rq(cfs_rq);
            check_enqueue_throttle(cfs_rq);
        }
    }
```
至此新进程如何被调度的讲解完毕,下一章节将讲解,idle进程被wakeup之后是怎样被调度的.
怎么选择cpu在第八章单独分析（select_task_rq_fair函数）