Linux的调度分析

一、就绪队列

内核为每个CPU创建一个进程就绪队列，该队列上的进程均有该CPU执行。

per-cpu变量在每个CPU上都有一个副本，对它的访问几乎不需要锁，因为每个CPU都在自己的副本上工作。

```
/*在 kernel/sched/core.c中定义*/
DEFINE_PER_CPU_SHARED_ALIGNED(struct rq, runqueues);

/* 宏定义展开在include/linux/percpu-defs.h
 * 静态分配per_cpu数组，数组名为name，结构类型为type
 */
#define DEFINE_PER_CPU_SHARED_ALIGNED(type, name)             \
    DEFINE_PER_CPU_SECTION(type, name, PER_CPU_SHARED_ALIGNED_SECTION) \
    ____cacheline_aligned_in_smp
```

每个数组元素就是一个就绪队列，对应一个cpu。

```
/*
 * This is the main, per-CPU runqueue data structure.
 *
 * Locking rule: those places that want to lock multiple runqueues
 * (such as the load balancing or the thread migration code), lock
 * acquire operations must be ordered by ascending &runqueue.
 */
struct rq {
    /* runqueue lock: */
    raw_spinlock_t      lock;

    /*
     * nr_running and cpu_load should be in the same cacheline because
     * remote CPUs use both these fields when doing load calculation.
     */
    unsigned int        nr_running;
#ifdef CONFIG_NUMA_BALANCING
    unsigned int        nr_numa_running;
    unsigned int        nr_preferred_running;
    unsigned int        numa_migrate_on;
#endif
#ifdef CONFIG_NO_HZ_COMMON
#ifdef CONFIG_SMP
    unsigned long       last_blocked_load_update_tick;
    unsigned int        has_blocked_load;
    call_single_data_t  nohz_csd;
#endif /* CONFIG_SMP */
    unsigned int        nohz_tick_stopped;
    atomic_t        nohz_flags;
#endif /* CONFIG_NO_HZ_COMMON */

```

```c
#ifdef CONFIG_SMP
    unsigned int        ttwu_pending;
#endif
    u64         nr_switches;

#ifdef CONFIG_UCLAMP_TASK
    /* Utilization clamp values based on CPU's RUNNABLE tasks */
    struct uclamp_rq    uclamp[UCLAMP_CNT] ____cacheline_aligned;
    unsigned int        uclamp_flags;
#define UCLAMP_FLAG_IDLE 0x01
#endif

    struct cfs_rq       cfs;                    /* 嵌入普通进程的cfs调度队列 */
    struct rt_rq        rt;                     /* 实时进程的实时调度策略调度队列 */
    struct dl_rq        dl;                     /* 空闲进程的调度队列 */

#ifdef CONFIG_FAIR_GROUP_SCHED
    /* list of leaf cfs_rq on this CPU: */
    struct list_head    leaf_cfs_rq_list;
    struct list_head    *tmp_alone_branch;
#endif /* CONFIG_FAIR_GROUP_SCHED */

    /*
     * This is part of a global counter where only the total sum
     * over all CPUs matters. A task can increase this counter on
     * one CPU and if it got migrated afterwards it may decrease
     * it on another CPU. Always updated under the runqueue lock:
     */
    unsigned long       nr_uninterruptible;

    struct task_struct __rcu    *curr;
    struct task_struct  *idle;
    struct task_struct  *stop;
    unsigned long       next_balance;
    struct mm_struct    *prev_mm;

    unsigned int        clock_update_flags;
    u64         clock;
    /* Ensure that all clocks are in the same cache line */
    u64         clock_task ____cacheline_aligned;
    u64         clock_pelt;
    unsigned long       lost_idle_time;

    atomic_t        nr_iowait;

#ifdef CONFIG_MEMBARRIER
    int membarrier_state;
#endif

#ifdef CONFIG_SMP
    struct root_domain      *rd;
    struct sched_domain __rcu   *sd;

    unsigned long       cpu_capacity;
    unsigned long       cpu_capacity_orig;

    struct callback_head    *balance_callback;

```

```
	unsigned char		nohz_idle_balance;
	unsigned char		idle_balance;

	unsigned long		misfit_task_load;

	/* For active balancing */
	int			active_balance;
	int			push_cpu;
	struct cpu_stop_work	active_balance_work;

	/* CPU of this runqueue: */
	int			cpu;
	int			online;

	struct list_head cfs_tasks;

	struct sched_avg	avg_rt;
	struct sched_avg	avg_dl;
#ifdef CONFIG_HAVE_SCHED_AVG_IRQ
	struct sched_avg	avg_irq;
#endif
#ifdef CONFIG_SCHED_THERMAL_PRESSURE
	struct sched_avg	avg_thermal;
#endif
	u64			idle_stamp;
	u64			avg_idle;

	/* This is used to determine avg_idle's max value */
	u64			max_idle_balance_cost;
#endif /* CONFIG_SMP */

#ifdef CONFIG_IRQ_TIME_ACCOUNTING
	u64			prev_irq_time;
#endif
#ifdef CONFIG_PARAVIRT
	u64			prev_steal_time;
#endif
#ifdef CONFIG_PARAVIRT_TIME_ACCOUNTING
	u64			prev_steal_time_rq;
#endif

	/* calc_load related fields */
	unsigned long		calc_load_update;
	long			calc_load_active;

#ifdef CONFIG_SCHED_HRTICK
#ifdef CONFIG_SMP
	call_single_data_t	hrtick_csd;
#endif
	struct hrtimer		hrtick_timer;
#endif

#ifdef CONFIG_SCHEDSTATS
	/* latency stats */
	struct sched_info	rq_sched_info;
	unsigned long long	rq_cpu_time;
	/* could above be rq->cfs_rq.exec_clock + rq->rt_rq.rt_runtime ? */

```

```c
    /* sys_sched_yield() stats */
    unsigned int        yld_count;

    /* schedule() stats */
    unsigned int        sched_count;
    unsigned int        sched_goidle;

    /* try_to_wake_up() stats */
    unsigned int        ttwu_count;
    unsigned int        ttwu_local;
#endif

#ifdef CONFIG_CPU_IDLE
    /* Must be inspected within a rcu lock section */
    struct cpuidle_state    *idle_state;
#endif
}
```

1.1 普通进程cfs就绪队列

cfs_rq和rt_rq以及dl_rq都定义在`kernel/sched/sched.h`中。

```c
/* CFS-related fields in a runqueue */
struct cfs_rq {
    struct load_weight  load;                   /* load维护了所有这些进程的累积负荷值 */
    unsigned int        nr_running;             /* nr_running计算了队列上可运行进程的数目 */
    unsigned int        h_nr_running;           /* SCHED_{NORMAL,BATCH,IDLE} */
    unsigned int        idle_h_nr_running;      /* SCHED_IDLE */

    u64         exec_clock;
    u64         min_vruntime;                   /* 跟踪记录队列上所有进程的最小虚拟运行时间 */
#ifndef CONFIG_64BIT
    u64         min_vruntime_copy;
#endif

    struct rb_root_cached   tasks_timeline;     /* 红黑树根以及待被调用的进程所在的树节点 */

    /*
     * 'curr' points to currently running entity on this cfs_rq.
     * It is set to NULL otherwise (i.e when none are currently running).
     */
    struct sched_entity *curr;                  /* curr指向当前正运行的实体 */
    struct sched_entity *next;                  /* next指向将被唤醒的进程 */
    struct sched_entity *last;                  /* last指向唤醒next进程的进程 */
    struct sched_entity *skip;

#ifdef  CONFIG_SCHED_DEBUG
```

```c
	unsigned int		nr_spread_over;
#endif

#ifdef CONFIG_SMP
	/*
	 * CFS load tracking
	 */
	struct sched_avg	avg;
#ifndef CONFIG_64BIT
	u64		load_last_update_time_copy;
#endif
	struct {
		raw_spinlock_t	lock ____cacheline_aligned;
		int		nr;
		unsigned long	load_avg;
		unsigned long	util_avg;
		unsigned long	runnable_avg;
	} removed;

#ifdef CONFIG_FAIR_GROUP_SCHED
	unsigned long		tg_load_avg_contrib;
	long			propagate;
	long			prop_runnable_sum;

	/*
	 *   h_load = weight * f(tg)
	 *
	 * Where f(tg) is the recursive weight fraction assigned to
	 * this group.
	 */
	unsigned long		h_load;
	u64			last_h_load_update;
	struct sched_entity	*h_load_next;
#endif /* CONFIG_FAIR_GROUP_SCHED */
#endif /* CONFIG_SMP */

#ifdef CONFIG_FAIR_GROUP_SCHED
	struct rq		*rq;	/* CPU runqueue to which this cfs_rq is attached */

	/*
	 * leaf cfs_rqs are those that hold tasks (lowest schedulable entity in
	 * a hierarchy). Non-leaf lrqs hold other higher schedulable entities
	 * (like users, containers etc.)
	 *
	 * leaf_cfs_rq_list ties together list of leaf cfs_rq's in a CPU.
	 * This list is used during load balance.
	 */
	int			on_list;
	struct list_head	leaf_cfs_rq_list;
	struct task_group	*tg;	/* group that "owns" this runqueue */

#ifdef CONFIG_CFS_BANDWIDTH
	int			runtime_enabled;
	s64			runtime_remaining;

	u64			throttled_clock;
	u64			throttled_clock_task;
```

```
    u64         throttled_clock_task_time;
    int         throttled;
    int         throttle_count;
    struct list_head    throttled_list;
#endif /* CONFIG_CFS_BANDWIDTH */
#endif /* CONFIG_FAIR_GROUP_SCHED */
};
```

其中`rb_root_cached`包含了两个成员：

一个是红黑树根`rb_root`；

另一个是`rb_node`，其总是设置为指向树最左边的结点，即最需要被调度的进程；

```
/*
 * Leftmost-cached rbtrees.
 *
 * We do not cache the rightmost node based on footprint
 * size vs number of potential users that could benefit
 * from O(1) rb_last(). Just not worth it, users that want
 * this feature can always implement the logic explicitly.
 * Furthermore, users that want to cache both pointers may
 * find it a bit asymmetric, but that's ok.
 */
struct rb_root_cached {
    struct rb_root rb_root;
    struct rb_node *rb_leftmost;
};
```

1.2 实时进程rt就绪队列

```
/* Real-Time classes' related field in a runqueue: */
struct rt_rq {
    struct rt_prio_array    active;
    unsigned int        rt_nr_running;
    unsigned int        rr_nr_running;
#if defined CONFIG_SMP || defined CONFIG_RT_GROUP_SCHED
    struct {
        int     curr; /* highest queued rt task prio */
#ifdef CONFIG_SMP
        int     next; /* next highest */
#endif
    } highest_prio;
#endif
#ifdef CONFIG_SMP
    unsigned long       rt_nr_migratory;
    unsigned long       rt_nr_total;
    int         overloaded;
    struct plist_head   pushable_tasks;

```

```c
#endif /* CONFIG_SMP */
	int		rt_queued;

	int		rt_throttled;
	u64		rt_time;
	u64		rt_runtime;
	/* Nests inside the rq lock: */
	raw_spinlock_t	rt_runtime_lock;

#ifdef CONFIG_RT_GROUP_SCHED
	unsigned long	rt_nr_boosted;

	struct rq	*rq;
	struct task_group	*tg;
#endif
};
```

二、调度实体

2.1 普通进程调度实体

由于调度器可以操作比进程更一般的实体，因此需要一个适当的数据结构来描述此类实体

```c
struct sched_entity {
	/* For load-balancing: */
	struct load_weight	load;			/* 用于负载均衡 */
	struct rb_node		run_node;		/* run_node是标准的树结点，使得实体可以在红黑树上排序 */
	struct list_head	group_node;
	unsigned int		on_rq;			/* on_rq表示该实体当前是否在就绪队列上接受调度 */

	u64			exec_start;
	u64			sum_exec_runtime;
	u64			vruntime;		/* 在进程执行期间虚拟时钟上流逝的时间数量由vruntime统计 */
	u64			prev_sum_exec_runtime;

	u64			nr_migrations;

	struct sched_statistics		statistics;

#ifdef CONFIG_FAIR_GROUP_SCHED
	int			depth;
	struct sched_entity	*parent;
	/* rq on which this entity is (to be) queued: */
	struct cfs_rq		*cfs_rq;
	/* rq "owned" by this entity/group: */
	struct cfs_rq		*my_q;
	/* cached value of my_q->h_nr_running */
	unsigned long		runnable_weight;
#endif

```

```
#ifdef CONFIG_SMP
    /*
     * Per entity load average tracking.
     *
     * Put into separate cache line so it does not
     * collide with read-mostly values above.
     */
    struct sched_avg        avg;
#endif
};
```

`struct sched_entity` 该结构体有两个作用：

（1） 包含有进程调度的信息（比如进程的运行时间，睡眠时间等等，调度程序参考这些信息决定是否调度进程）

（2） 使用该结构体来组织进程

- `struct load_weight  load` 指定了权重，决定了各个实体占队列总负荷的比例。计算负荷权重是调度器的一项重任，因为CFS所需的虚拟时钟的速度最终依赖于负荷。
- `run_node` 是红黑树节点，因此 `struct sched_entity` 调度实体将被组织成红黑树的形式，同时意味着普通进程也被组织成红黑树的形式。
- 在进程运行时，我们需要记录消耗的CPU时间，以用于完全公平调度器。 `sum_exec_runtime` 即用于该目的。跟踪运行时间是由 `update_curr` 不断累积完成的。调度器中许多地方都会调用该函数，例如，新进程加入就绪队列时，或者周期性调度器中。每次调用时，会计算当前时间和 `exec_start` 之间的差值， `exec_start` 则更新到当前时间。差值则被加到 `sum_exec_runtime` 。
- 在进程被撤销CPU时，其当前 `sum_exec_runtime` 值保存到 `prev_exec_runtime` 。此后，在进程抢占时又需要该数据。但请注意，在 `prev_exec_runtime` 中保存 `sum_exec_runtime` 的值，并不意味着重置 `sum_exec_runtime` ！ 原值保存下来，而 `um_exec_runtime` 则持续单调增长

2.2 实时进程rt调度实体

用于组织实时进程的调度。

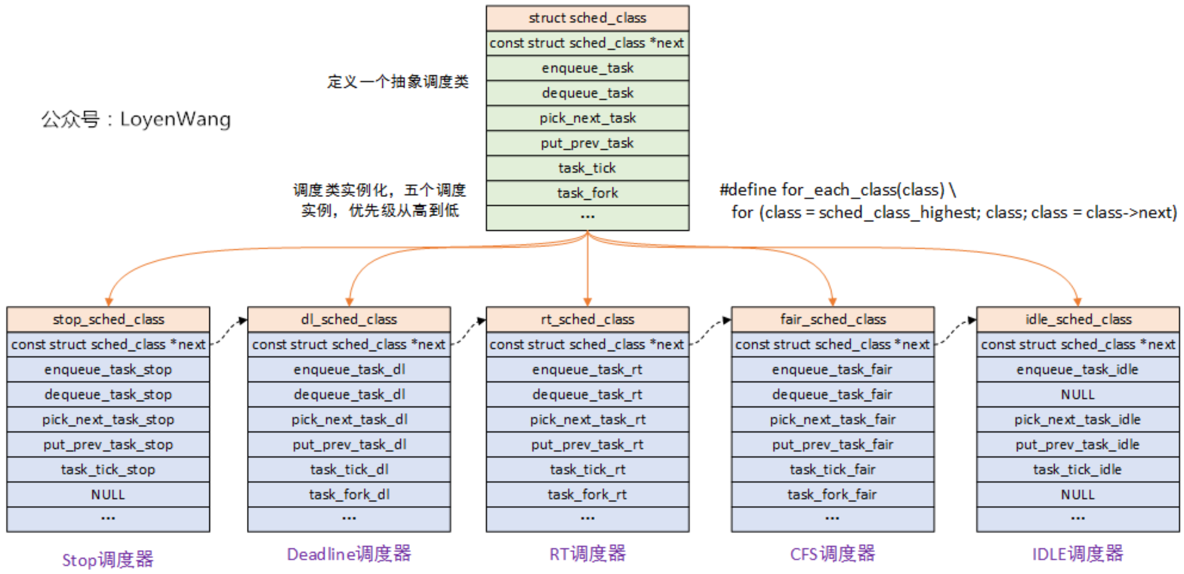
```
struct sched_rt_entity {
    struct list_head        run_list;
    unsigned long           timeout;
    unsigned long           watchdog_stamp;
    unsigned int            time_slice;
    unsigned short          on_rq;
    unsigned short          on_list;

    struct sched_rt_entity      *back;
#ifdef CONFIG_RT_GROUP_SCHED
    struct sched_rt_entity      *parent;
    /* rq on which this entity is (to be) queued: */
    struct rt_rq            *rt_rq;
    /* rq "owned" by this entity/group: */
    struct rt_rq            *my_q;
#endif
} __randomize_layout;
```

`struct list_head run_list` 表明 `rt_entity` 是由双链表管理，而不是红黑树。

三、调度类

3.1 调度类sched_class

调度类整体关系：



`kernel/sched/sched.h`中声明了调度类。

`sched_class`中定义了一堆函数指针，指针指向的函数就是调度策略的具体实现，所有和进程调度有关的函数都直接或者间接调用了这些成员函数，来实现进程调度。此外，每个进程描述符中都包含一个指向该结构体类型的指针sched_class，指向了所采用的调度类。

```
struct sched_class {

#ifdef CONFIG_UCLAMP_TASK
    int uclamp_enabled;
#endif

    void (*enqueue_task) (struct rq *rq, struct task_struct *p, int flags);
    void (*dequeue_task) (struct rq *rq, struct task_struct *p, int flags);
    void (*yield_task)   (struct rq *rq);
    bool (*yield_to_task)(struct rq *rq, struct task_struct *p);

    void (*check_preempt_curr)(struct rq *rq, struct task_struct *p, int flags);

    struct task_struct *(*pick_next_task)(struct rq *rq);

    void (*put_prev_task)(struct rq *rq, struct task_struct *p);
    void (*set_next_task)(struct rq *rq, struct task_struct *p, bool first);

#ifdef CONFIG_SMP
    int (*balance)(struct rq *rq, struct task_struct *prev, struct rq_flags *rf);
    int  (*select_task_rq)(struct task_struct *p, int task_cpu, int sd_flag, int flags);
    void (*migrate_task_rq)(struct task_struct *p, int new_cpu);

    void (*task_woken)(struct rq *this_rq, struct task_struct *task);
```

```

    void (*set_cpus_allowed)(struct task_struct *p,
                 const struct cpumask *newmask);

    void (*rq_online)(struct rq *rq);
    void (*rq_offline)(struct rq *rq);
#endif

    void (*task_tick)(struct rq *rq, struct task_struct *p, int queued);
    void (*task_fork)(struct task_struct *p);
    void (*task_dead)(struct task_struct *p);

    /*
     * The switched_from() call is allowed to drop rq->lock, therefore we
     * cannot assume the switched_from/switched_to pair is serliazed by
     * rq->lock. They are however serialized by p->pi_lock.
     */
    void (*switched_from)(struct rq *this_rq, struct task_struct *task);
    void (*switched_to)  (struct rq *this_rq, struct task_struct *task);
    void (*prio_changed) (struct rq *this_rq, struct task_struct *task,
                 int oldprio);

    unsigned int (*get_rr_interval)(struct rq *rq,
                     struct task_struct *task);

    void (*update_curr)(struct rq *rq);

#define TASK_SET_GROUP      0
#define TASK_MOVE_GROUP     1

#ifdef CONFIG_FAIR_GROUP_SCHED
    void (*task_change_group)(struct task_struct *p, int type);
#endif
} __aligned(STRUCT_ALIGNMENT); /* STRUCT_ALIGN(), vmlinux.lds.h */
```

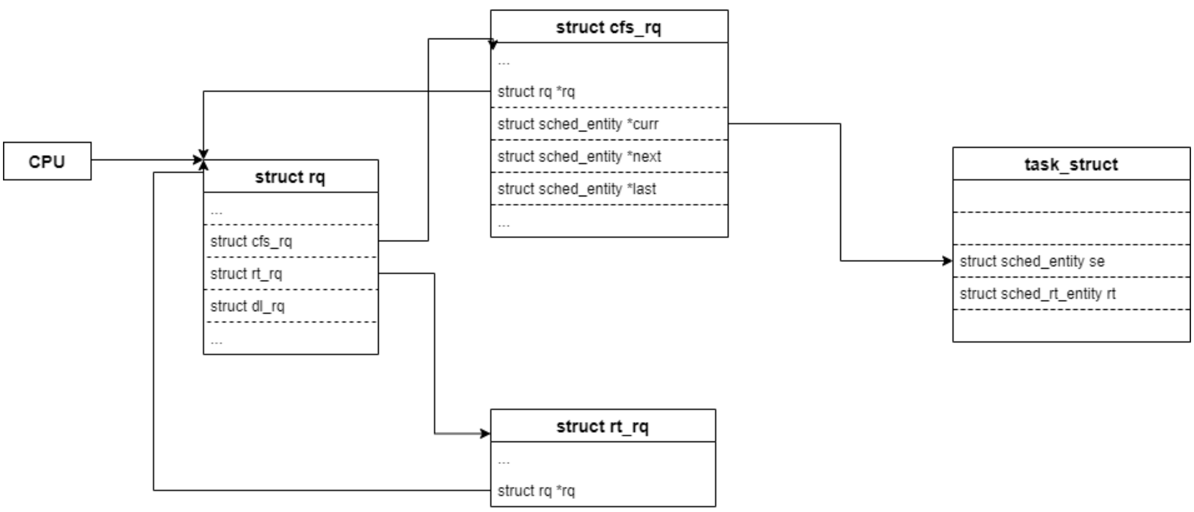
3.2 CFS调度策略类

`kernel/sched/fair.c`中定义并初始化了完全公平调度策略的调度类`fair_sched_class`

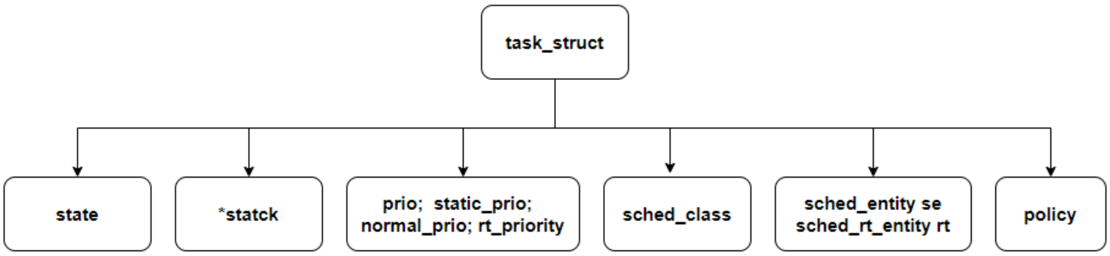
```
/*
 * All the scheduling class methods:
 */
const struct sched_class fair_sched_class
    __attribute__((section("__fair_sched_class"))) = {
    .enqueue_task       = enqueue_task_fair,
    .dequeue_task       = dequeue_task_fair,
    .yield_task     = yield_task_fair,
    .yield_to_task      = yield_to_task_fair,

    .check_preempt_curr = check_preempt_wakeup,

    .pick_next_task     = __pick_next_task_fair,
    .put_prev_task      = put_prev_task_fair,
    .set_next_task          = set_next_task_fair,

```

```
#ifdef CONFIG_SMP
    .balance        = balance_fair,
    .select_task_rq     = select_task_rq_fair,
    .migrate_task_rq    = migrate_task_rq_fair,

    .rq_online      = rq_online_fair,
    .rq_offline     = rq_offline_fair,

    .task_dead      = task_dead_fair,
    .set_cpus_allowed   = set_cpus_allowed_common,
#endif

    .task_tick      = task_tick_fair,
    .task_fork      = task_fork_fair,

    .prio_changed       = prio_changed_fair,
    .switched_from      = switched_from_fair,
    .switched_to        = switched_to_fair,

    .get_rr_interval    = get_rr_interval_fair,

    .update_curr        = update_curr_fair,

#ifdef CONFIG_FAIR_GROUP_SCHED
    .task_change_group  = task_change_group_fair,
#endif

#ifdef CONFIG_UCLAMP_TASK
    .uclamp_enabled     = 1,
#endif
};
```

四、进程描述符



task_struct中包含了很多重要的元素，如进程状态、栈内存指针、进程优先级（动态、静态、普通、实时）、进程所属调度类、进程调度实体（普通和实时）、调度策略等等。



`task_struct`在`include/linux/sched.h`中声明，值得好好分析一下。

```
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
	/* -1 unrunnable, 0 runnable, >0 stopped: */
	volatile long			state;					/* -1表示不可运行，0表示可运行，>0表示停止 */

	/*
	 * This begins the randomizable portion of task_struct. Only
	 * scheduling-critical items should be added above here.
	 */
	randomized_struct_fields_start

	void				*stack;
	refcount_t			usage;
	/* Per task flags (PF_*), defined further below: */
	unsigned int			flags;					/* 每进程标志，下文定义 */
	unsigned int			ptrace;

#ifdef CONFIG_SMP
	int				on_cpu;
	struct __call_single_node	wake_entry;
#ifdef CONFIG_THREAD_INFO_IN_TASK
	/* Current CPU: */
	unsigned int			cpu;
#endif
	unsigned int			wakee_flips;
	unsigned long			wakee_flip_decay_ts;
	struct task_struct		*last_wakee;

	/*
	 * recent_used_cpu is initially set as the last CPU used by a task
	 * that wakes affine another task. Waker/wakee relationships can
	 * push tasks around a CPU where each wakeup moves to the next one.
	 * Tracking a recently used CPU allows a quick search for a recently
	 * used CPU that may be idle.
	 */
```

```
    int             recent_used_cpu;
    int             wake_cpu;
#endif
    int             on_rq;

    int             prio;                                               /* 动态优先级 */
    int             static_prio;                                        /* 静态优先级 */
    int             normal_prio;                                        /* 普通优先级 */
    unsigned int            rt_priority;                                /* 实时优先级 */

    const struct sched_class    *sched_class;                           /* 每个PCB都会指向一个调度类，表示该进程所属的调度器类*/
    struct sched_entity     se;                                         /* 普通调度实体 */
    struct sched_rt_entity      rt;                                     /* 实时调度实体 */
#ifdef CONFIG_CGROUP_SCHED
    struct task_group       *sched_task_group;
#endif
    struct sched_dl_entity      dl;                                     /* 空闲调度实体 */

#ifdef CONFIG_UCLAMP_TASK
    /*
     * Clamp values requested for a scheduling entity.
     * Must be updated with task_rq_lock() held.
     */
    struct uclamp_se        uclamp_req[UCLAMP_CNT];
    /*
     * Effective clamp values used for a scheduling entity.
     * Must be updated with task_rq_lock() held.
     */
    struct uclamp_se        uclamp[UCLAMP_CNT];
#endif

#ifdef CONFIG_PREEMPT_NOTIFIERS
    /* List of struct preempt_notifier: */
    struct hlist_head       preempt_notifiers;
#endif

#ifdef CONFIG_BLK_DEV_IO_TRACE
    unsigned int            btrace_seq;
#endif

    unsigned int            policy;
    int             nr_cpus_allowed;
    const cpumask_t         *cpus_ptr;
    cpumask_t           cpus_mask;

#ifdef CONFIG_PREEMPT_RCU
    int             rcu_read_lock_nesting;
    union rcu_special       rcu_read_unlock_special;
    struct list_head        rcu_node_entry;
    struct rcu_node         *rcu_blocked_node;
```

```
#endif /* #ifdef CONFIG_PREEMPT_RCU */

#ifdef CONFIG_TASKS_RCU
	unsigned long			rcu_tasks_nvcsw;
	u8				rcu_tasks_holdout;
	u8				rcu_tasks_idx;
	int				rcu_tasks_idle_cpu;
	struct list_head		rcu_tasks_holdout_list;
#endif /* #ifdef CONFIG_TASKS_RCU */

#ifdef CONFIG_TASKS_TRACE_RCU
	int				trc_reader_nesting;
	int				trc_ipi_to_cpu;
	union rcu_special		trc_reader_special;
	bool				trc_reader_checked;
	struct list_head		trc_holdout_list;
#endif /* #ifdef CONFIG_TASKS_TRACE_RCU */

	struct sched_info		sched_info;

	struct list_head		tasks;
#ifdef CONFIG_SMP
	struct plist_node		pushable_tasks;
	struct rb_node			pushable_dl_tasks;
#endif

	struct mm_struct		*mm;
	struct mm_struct		*active_mm;

	/* Per-thread vma caching: */
	struct vmacache			vmacache;

#ifdef SPLIT_RSS_COUNTING
	struct task_rss_stat		rss_stat;
#endif
	int				exit_state;
	int				exit_code;
	int				exit_signal;
	/* The signal sent when the parent dies: */
	int				pdeath_signal;
	/* JOBCTL_*, siglock protected: */
	unsigned long			jobctl;

	/* Used for emulating ABI behavior of previous Linux versions: */
	unsigned int			personality;

	/* Scheduler bits, serialized by scheduler locks: */
	unsigned			sched_reset_on_fork:1;
	unsigned			sched_contributes_to_load:1;
	unsigned			sched_migrated:1;
	unsigned			sched_remote_wakeup:1;
#ifdef CONFIG_PSI
	unsigned			sched_psi_wake_requeue:1;
#endif

	/* Force alignment to the next boundary: */
	unsigned			:0;

```

```
	/* Unserialized, strictly 'current' */

	/* Bit to tell LSMs we're in execve(): */
	unsigned			in_execve:1;
	unsigned			in_iowait:1;
#ifndef TIF_RESTORE_SIGMASK
	unsigned			restore_sigmask:1;
#endif
#ifdef CONFIG_MEMCG
	unsigned			in_user_fault:1;
#endif
#ifdef CONFIG_COMPAT_BRK
	unsigned			brk_randomized:1;
#endif
#ifdef CONFIG_CGROUPS
	/* disallow userland-initiated cgroup migration */
	unsigned			no_cgroup_migration:1;
	/* task is frozen/stopped (used by the cgroup freezer) */
	unsigned			frozen:1;
#endif
#ifdef CONFIG_BLK_CGROUP
	unsigned			use_memdelay:1;
#endif
#ifdef CONFIG_PSI
	/* Stalled due to lack of memory */
	unsigned			in_memstall:1;
#endif

	unsigned long			atomic_flags; /* Flags requiring atomic access. */

	struct restart_block		restart_block;

	pid_t				pid;
	pid_t				tgid;

#ifdef CONFIG_STACKPROTECTOR
	/* Canary value for the -fstack-protector GCC feature: */
	unsigned long			stack_canary;
#endif
	/*
	 * Pointers to the (original) parent process, youngest child, younger sibling,
	 * older sibling, respectively.  (p->father can be replaced with
	 * p->real_parent->pid)
	 */

	/* Real parent process: */
	struct task_struct __rcu	*real_parent;

	/* Recipient of SIGCHLD, wait4() reports: */
	struct task_struct __rcu	*parent;

	/*
	 * Children/sibling form the list of natural children:
	 */
	struct list_head		children;
	struct list_head		sibling;
```

```
	struct task_struct		*group_leader;

	/*
	 * 'ptraced' is the list of tasks this task is using ptrace() on.
	 *
	 * This includes both natural children and PTRACE_ATTACH targets.
	 * 'ptrace_entry' is this task's link on the p->parent->ptraced list.
	 */
	struct list_head		ptraced;
	struct list_head		ptrace_entry;

	/* PID/PID hash table linkage. */
	struct pid			*thread_pid;
	struct hlist_node		pid_links[PIDTYPE_MAX];
	struct list_head		thread_group;
	struct list_head		thread_node;

	struct completion		*vfork_done;

	/* CLONE_CHILD_SETTID: */
	int __user			*set_child_tid;

	/* CLONE_CHILD_CLEARTID: */
	int __user			*clear_child_tid;

	u64				utime;
	u64				stime;
#ifdef CONFIG_ARCH_HAS_SCALED_CPUTIME
	u64				utimescaled;
	u64				stimescaled;
#endif
	u64				gtime;
	struct prev_cputime		prev_cputime;
#ifdef CONFIG_VIRT_CPU_ACCOUNTING_GEN
	struct vtime			vtime;
#endif

#ifdef CONFIG_NO_HZ_FULL
	atomic_t			tick_dep_mask;
#endif
	/* Context switch counts: */
	unsigned long			nvcsw;
	unsigned long			nivcsw;

	/* Monotonic time in nsecs: */
	u64				start_time;

	/* Boot based time in nsecs: */
	u64				start_boottime;

	/* MM fault and swap info: this can arguably be seen as either mm-specific or thread-specific: */
	unsigned long			min_flt;
	unsigned long			maj_flt;

	/* Empty if CONFIG_POSIX_CPUTIMERS=n */
	struct posix_cputimers		posix_cputimers;

```

```c
#ifdef CONFIG_POSIX_CPU_TIMERS_TASK_WORK
	struct posix_cputimers_work	posix_cputimers_work;
#endif

	/* Process credentials: */

	/* Tracer's credentials at attach: */
	const struct cred __rcu		*ptracer_cred;

	/* Objective and real subjective task credentials (COW): */
	const struct cred __rcu		*real_cred;

	/* Effective (overridable) subjective task credentials (COW): */
	const struct cred __rcu		*cred;

#ifdef CONFIG_KEYS
	/* Cached requested key. */
	struct key			*cached_requested_key;
#endif

	/*
	 * executable name, excluding path.
	 *
	 * - normally initialized setup_new_exec()
	 * - access it with [gs]et_task_comm()
	 * - lock it with task_lock()
	 */
	char				comm[TASK_COMM_LEN];

	struct nameidata		*nameidata;

#ifdef CONFIG_SYSVIPC
	struct sysv_sem			sysvsem;
	struct sysv_shm			sysvshm;
#endif
#ifdef CONFIG_DETECT_HUNG_TASK
	unsigned long			last_switch_count;
	unsigned long			last_switch_time;
#endif
	/* Filesystem information: */
	struct fs_struct		*fs;

	/* Open file information: */
	struct files_struct		*files;

#ifdef CONFIG_IO_URING
	struct io_uring_task		*io_uring;
#endif

	/* Namespaces: */
	struct nsproxy			*nsproxy;

	/* Signal handlers: */
	struct signal_struct		*signal;
	struct sighand_struct __rcu		*sighand;
	sigset_t			blocked;
	sigset_t			real_blocked;
	/* Restored if set_restore_sigmask() was used: */
```

```
	sigset_t			saved_sigmask;
	struct sigpending		pending;
	unsigned long			sas_ss_sp;
	size_t				sas_ss_size;
	unsigned int			sas_ss_flags;

	struct callback_head		*task_works;

#ifdef CONFIG_AUDIT
#ifdef CONFIG_AUDITSYSCALL
	struct audit_context		*audit_context;
#endif
	kuid_t				loginuid;
	unsigned int			sessionid;
#endif
	struct seccomp			seccomp;

	/* Thread group tracking: */
	u64				parent_exec_id;
	u64				self_exec_id;

	/* Protection against (de-)allocation: mm, files, fs, tty, keyrings, mems_allowed, mempolicy: */
	spinlock_t			alloc_lock;

	/* Protection of the PI data structures: */
	raw_spinlock_t			pi_lock;

	struct wake_q_node		wake_q;

#ifdef CONFIG_RT_MUTEXES
	/* PI waiters blocked on a rt_mutex held by this task: */
	struct rb_root_cached		pi_waiters;
	/* Updated under owner's pi_lock and rq lock */
	struct task_struct		*pi_top_task;
	/* Deadlock detection and priority inheritance handling: */
	struct rt_mutex_waiter		*pi_blocked_on;
#endif

#ifdef CONFIG_DEBUG_MUTEXES
	/* Mutex deadlock detection: */
	struct mutex_waiter		*blocked_on;
#endif

#ifdef CONFIG_DEBUG_ATOMIC_SLEEP
	int				non_block_count;
#endif

#ifdef CONFIG_TRACE_IRQFLAGS
	struct irqtrace_events		irqtrace;
	unsigned int			hardirq_threaded;
	u64				hardirq_chain_key;
	int				softirqs_enabled;
	int				softirq_context;
	int				irq_config;
#endif

#ifdef CONFIG_LOCKDEP
```

```
# define MAX_LOCK_DEPTH          48UL
    u64                 curr_chain_key;
    int                 lockdep_depth;
    unsigned int                lockdep_recursion;
    struct held_lock        held_locks[MAX_LOCK_DEPTH];
#endif

#ifdef CONFIG_UBSAN
    unsigned int                in_ubsan;
#endif

    /* Journalling filesystem info: */
    void                    *journal_info;

    /* Stacked block device info: */
    struct bio_list         *bio_list;

#ifdef CONFIG_BLOCK
    /* Stack plugging: */
    struct blk_plug         *plug;
#endif

    /* VM state: */
    struct reclaim_state        *reclaim_state;

    struct backing_dev_info     *backing_dev_info;

    struct io_context       *io_context;

#ifdef CONFIG_COMPACTION
    struct capture_control      *capture_control;
#endif
    /* Ptrace state: */
    unsigned long           ptrace_message;
    kernel_siginfo_t        *last_siginfo;

    struct task_io_accounting   ioac;
#ifdef CONFIG_PSI
    /* Pressure stall state */
    unsigned int                psi_flags;
#endif
#ifdef CONFIG_TASK_XACCT
    /* Accumulated RSS usage: */
    u64                 acct_rss_mem1;
    /* Accumulated virtual memory usage: */
    u64                 acct_vm_mem1;
    /* stime + utime since last update: */
    u64                 acct_timexpd;
#endif
#ifdef CONFIG_CPUSETS
    /* Protected by ->alloc_lock: */
    nodemask_t          mems_allowed;
    /* Seqence number to catch updates: */
    seqcount_spinlock_t     mems_allowed_seq;
    int                 cpuset_mem_spread_rotor;
    int                 cpuset_slab_spread_rotor;
#endif
#ifdef CONFIG_CGROUPS
```

```
	/* Control Group info protected by css_set_lock: */
	struct css_set __rcu		*cgroups;
	/* cg_list protected by css_set_lock and tsk->alloc_lock: */
	struct list_head		cg_list;
#endif
#ifdef CONFIG_X86_CPU_RESCTRL
	u32				closid;
	u32				rmid;
#endif
#ifdef CONFIG_FUTEX
	struct robust_list_head __user	*robust_list;
#ifdef CONFIG_COMPAT
	struct compat_robust_list_head __user *compat_robust_list;
#endif
	struct list_head		pi_state_list;
	struct futex_pi_state		*pi_state_cache;
	struct mutex			futex_exit_mutex;
	unsigned int			futex_state;
#endif
#ifdef CONFIG_PERF_EVENTS
	struct perf_event_context	*perf_event_ctxp[perf_nr_task_contexts];
	struct mutex			perf_event_mutex;
	struct list_head		perf_event_list;
#endif
#ifdef CONFIG_DEBUG_PREEMPT
	unsigned long			preempt_disable_ip;
#endif
#ifdef CONFIG_NUMA
	/* Protected by alloc_lock: */
	struct mempolicy		*mempolicy;
	short				il_prev;
	short				pref_node_fork;
#endif
#ifdef CONFIG_NUMA_BALANCING
	int				numa_scan_seq;
	unsigned int			numa_scan_period;
	unsigned int			numa_scan_period_max;
	int				numa_preferred_nid;
	unsigned long			numa_migrate_retry;
	/* Migration stamp: */
	u64				node_stamp;
	u64				last_task_numa_placement;
	u64				last_sum_exec_runtime;
	struct callback_head		numa_work;

	/*
	 * This pointer is only modified for current in syscall and
	 * pagefault context (and for tasks being destroyed), so it can be read
	 * from any of the following contexts:
	 *  - RCU read-side critical section
	 *  - current->numa_group from everywhere
	 *  - task's runqueue locked, task not running
	 */
	struct numa_group __rcu		*numa_group;

	/*
	 * numa_faults is an array split into four regions:
	 * faults_memory, faults_cpu, faults_memory_buffer, faults_cpu_buffer
```

```
	 * in this precise order.
	 *
	 * faults_memory: Exponential decaying average of faults on a per-node
	 * basis. Scheduling placement decisions are made based on these
	 * counts. The values remain static for the duration of a PTE scan.
	 * faults_cpu: Track the nodes the process was running on when a NUMA
	 * hinting fault was incurred.
	 * faults_memory_buffer and faults_cpu_buffer: Record faults per node
	 * during the current scan window. When the scan completes, the counts
	 * in faults_memory and faults_cpu decay and these values are copied.
	 */
	unsigned long			*numa_faults;
	unsigned long			total_numa_faults;

	/*
	 * numa_faults_locality tracks if faults recorded during the last
	 * scan window were remote/local or failed to migrate. The task scan
	 * period is adapted based on the locality of the faults with different
	 * weights depending on whether they were shared or private faults
	 */
	unsigned long			numa_faults_locality[3];

	unsigned long			numa_pages_migrated;
#endif /* CONFIG_NUMA_BALANCING */

#ifdef CONFIG_RSEQ
	struct rseq __user *rseq;
	u32 rseq_sig;
	/*
	 * RmW on rseq_event_mask must be performed atomically
	 * with respect to preemption.
	 */
	unsigned long rseq_event_mask;
#endif

	struct tlbflush_unmap_batch	tlb_ubc;

	union {
		refcount_t		rcu_users;
		struct rcu_head		rcu;
	};

	/* Cache last used pipe for splice(): */
	struct pipe_inode_info		*splice_pipe;

	struct page_frag		task_frag;

#ifdef CONFIG_TASK_DELAY_ACCT
	struct task_delay_info		*delays;
#endif

#ifdef CONFIG_FAULT_INJECTION
	int				make_it_fail;
	unsigned int			fail_nth;
#endif
	/*
	 * When (nr_dirtied >= nr_dirtied_pause), it's time to call
	 * balance_dirty_pages() for a dirty throttling pause:
```

```
	 */
	int				nr_dirtied;
	int				nr_dirtied_pause;
	/* Start of a write-and-pause period: */
	unsigned long			dirty_paused_when;

#ifdef CONFIG_LATENCYTOP
	int				latency_record_count;
	struct latency_record		latency_record[LT_SAVECOUNT];
#endif
	/*
	 * Time slack values; these are used to round up poll() and
	 * select() etc timeout values. These are in nanoseconds.
	 */
	u64				timer_slack_ns;
	u64				default_timer_slack_ns;

#ifdef CONFIG_KASAN
	unsigned int			kasan_depth;
#endif

#ifdef CONFIG_KCSAN
	struct kcsan_ctx		kcsan_ctx;
#ifdef CONFIG_TRACE_IRQFLAGS
	struct irqtrace_events		kcsan_save_irqtrace;
#endif
#endif

#ifdef CONFIG_FUNCTION_GRAPH_TRACER
	/* Index of current stored address in ret_stack: */
	int				curr_ret_stack;
	int				curr_ret_depth;

	/* Stack of return addresses for return function tracing: */
	struct ftrace_ret_stack		*ret_stack;

	/* Timestamp for last schedule: */
	unsigned long long		ftrace_timestamp;

	/*
	 * Number of functions that haven't been traced
	 * because of depth overrun:
	 */
	atomic_t			trace_overrun;

	/* Pause tracing: */
	atomic_t			tracing_graph_pause;
#endif

#ifdef CONFIG_TRACING
	/* State flags for use by tracers: */
	unsigned long			trace;

	/* Bitmask and counter of trace recursion: */
	unsigned long			trace_recursion;
#endif /* CONFIG_TRACING */

#ifdef CONFIG_KCOV
```

```
	/* See kernel/kcov.c for more details. */

	/* Coverage collection mode enabled for this task (0 if disabled): */
	unsigned int			kcov_mode;

	/* Size of the kcov_area: */
	unsigned int			kcov_size;

	/* Buffer for coverage collection: */
	void				*kcov_area;

	/* KCOV descriptor wired with this task or NULL: */
	struct kcov			*kcov;

	/* KCOV common handle for remote coverage collection: */
	u64				kcov_handle;

	/* KCOV sequence number: */
	int				kcov_sequence;

	/* Collect coverage from softirq context: */
	unsigned int			kcov_softirq;
#endif

#ifdef CONFIG_MEMCG
	struct mem_cgroup		*memcg_in_oom;
	gfp_t				memcg_oom_gfp_mask;
	int				memcg_oom_order;

	/* Number of pages to reclaim on returning to userland: */
	unsigned int			memcg_nr_pages_over_high;

	/* Used by memcontrol for targeted memcg charge: */
	struct mem_cgroup		*active_memcg;
#endif

#ifdef CONFIG_BLK_CGROUP
	struct request_queue		*throttle_queue;
#endif

#ifdef CONFIG_UPROBES
	struct uprobe_task		*utask;
#endif
#if defined(CONFIG_BCACHE) || defined(CONFIG_BCACHE_MODULE)
	unsigned int			sequential_io;
	unsigned int			sequential_io_avg;
#endif
#ifdef CONFIG_DEBUG_ATOMIC_SLEEP
	unsigned long			task_state_change;
#endif
	int				pagefault_disabled;
#ifdef CONFIG_MMU
	struct task_struct		*oom_reaper_list;
#endif
#ifdef CONFIG_VMAP_STACK
	struct vm_struct		*stack_vm_area;
#endif
#ifdef CONFIG_THREAD_INFO_IN_TASK
```

```
    /* A live task holds one reference: */
    refcount_t          stack_refcount;
#endif
#ifdef CONFIG_LIVEPATCH
    int patch_state;
#endif
#ifdef CONFIG_SECURITY
    /* Used by LSM modules for access restriction: */
    void                *security;
#endif

#ifdef CONFIG_GCC_PLUGIN_STACKLEAK
    unsigned long           lowest_stack;
    unsigned long           prev_lowest_stack;
#endif

#ifdef CONFIG_X86_MCE
    u64             mce_addr;
    __u64               mce_ripv : 1,
                    mce_whole_page : 1,
                    __mce_reserved : 62;
    struct callback_head        mce_kill_me;
#endif

    /*
     * New fields for task_struct should be added above here, so that
     * they are included in the randomized portion of task_struct.
     */
    randomized_struct_fields_end

    /* CPU-specific state of this task: */
    struct thread_struct        thread;

    /*
     * WARNING: on x86, 'thread_struct' contains a variable-sized
     * structure.  It *MUST* be at the end of 'task_struct'.
     *
     * Do not put anything below here!
     */
}
```

五、调度器

5.1 周期性调度器

周期性调度器在scheduler_tick中实现。如果系统正在活动中，内核会按照频率HZ自动调用该函数。如果没有进程在等待调度，那么在计算机电力供应不足的情况下，也可以关闭该调度器以减少电能消耗。

`kernel/sched/core.c`中定义

```
/*
 * This function gets called by the timer code, with HZ frequency.
 * We call it with interrupts disabled.
 */
void scheduler_tick(void)
```

```c
{
    int cpu = smp_processor_id();                          /* 获取当前cpu号 */
    struct rq *rq = cpu_rq(cpu);                           /* 获取cpu就绪队列rq（每个cpu都有一个就绪队列） */
    struct task_struct *curr = rq->curr;                   /* 从rq中获取当前运行进程的描述符 */
    struct rq_flags rf;
    unsigned long thermal_pressure;                        /* 5.7内核后的新特性，CPU热压过高后会限频，但调度器并不知道，所以需要让调度器感知CPU频率被限制住，这样更好的调度任务*/

    arch_scale_freq_tick();
    sched_clock_tick();

    rq_lock(rq, &rf);

    update_rq_clock(rq);                     /* 更新就绪队列中的clock和clock_task成员值，代表当前的时间，一般我们会用到clock_task*/
    thermal_pressure = arch_scale_thermal_pressure(cpu_of(rq));
    update_thermal_load_avg(rq_clock_thermal(rq), rq, thermal_pressure);
    curr->sched_class->task_tick(rq, curr, 0);   /*进入当前进程的调度类的task_tick函数中，更新当前进程的时间片，不同调度类的该函数实现不同*/
    calc_global_load_tick(rq);
    psi_task_tick(rq);

    rq_unlock(rq, &rf);

    perf_event_task_tick();

#ifdef CONFIG_SMP
    rq->idle_balance = idle_cpu(cpu);            /* 判断cpu是否空闲 */
    trigger_load_balance(rq);                    /* 挂起SCHED_SOFTIRQ软中断函数，去做周期性的负载平衡操作 */
#endif
}
```

该函数主要作用：

(1) 管理内核中与整个系统和各个进程的调度相关的统计量。其间执行的主要操作是对各种计数器加1，我们对此没什么兴趣。
(2) 激活负责当前进程的调度类的周期性调度方法。

5.2 主调度器

在内核中的许多地方，如果要将CPU分配给与当前活动进程不同的另一个进程，都会直接调用主调度器函数（schedule）。在从系统调用返回之后，内核也会检查当前进程是否设置了重调度标志TIF_NEED_RESCHED，例如，前述的scheduler_tick就会设置该标志。如果是这样，则内核会调用schedule。该函数假定当前活动进程一定会被另一个进程取代。

5.2.1 __sched前缀

```
/* Attach to any functions which should be ignored in wchan output. */
#define __sched     __attribute__((__section__(".sched.text")))
```

将相关函数的代码编译之后，放到目标文件的一个特定的段中，即`.sched.text`中。该信息使得内核在显示栈转储或类似信息时，忽略所有与调度有关的调用。

5.2.2 schedule函数

```
#define tif_need_resched() test_thread_flag(TIF_NEED_RESCHED)

static __always_inline bool need_resched(void)      /* 该函数用于判断TIF_NEED_RESCHED标志位看是否需要重新调度 */
{
    return unlikely(tif_need_resched());
}
```

`schedule()` 函数

```
asmlinkage __visible void __sched schedule(void)
{
    struct task_struct *tsk = current;          /* 获取当前进程的结构体*/

    sched_submit_work(tsk);                     /* 防止死锁问题 */
    do {
        preempt_disable();                      /* 关闭抢占 */
        __schedule(false);
        sched_preempt_enable_no_resched();      /* 开启抢占 */
    } while (need_resched());                   /* 如果需要重新调度，则循环？ */
    sched_update_worker(tsk);
}
EXPORT_SYMBOL(schedule);
```

`__schedule()` 函数

```
/*
 * __schedule()是主要的调度函数.
 * 所谓的主要函数是指推动调度，因此进入该函数的原因有：
 * 1. 明显的阻塞：锁、信号、等待队列等等；
 * 2. TIF_NEED_RESCHED标志被中断和用户空间返回路劲检测到；
 * 3. 唤醒wakeup并没有真正的进入schedule()，唤醒只是将进程加入run-queue
 * The main means of driving the scheduler and thus entering this function are:
 *
 *   1. Explicit blocking: mutex, semaphore, waitqueue, etc.
 *
 *   2. TIF_NEED_RESCHED flag is checked on interrupt and userspace return
 *      paths. For example, see arch/x86/entry_64.S.
 *
 *      To drive preemption between tasks, the scheduler sets the flag in timer
 *      interrupt handler scheduler_tick().
 *
```

```
 *   3. Wakeups don't really cause entry into schedule(). They add a
 *      task to the run-queue and that's it.
 *
 *      Now, if the new task added to the run-queue preempts the current
 *      task, then the wakeup sets TIF_NEED_RESCHED and schedule() gets
 *      called on the nearest possible occasion:
 *
 *       - If the kernel is preemptible (CONFIG_PREEMPTION=y):
 *
 *         - in syscall or exception context, at the next outmost
 *           preempt_enable(). (this might be as soon as the wake_up()'s
 *           spin_unlock()!)
 *
 *         - in IRQ context, return from interrupt-handler to
 *           preemptible context
 *
 *       - If the kernel is not preemptible (CONFIG_PREEMPTION is not set)
 *         then at the next:
 *
 *          - cond_resched() call
 *          - explicit schedule() call
 *          - return from syscall or exception to user-space
 *          - return from interrupt-handler to user-space
 *
 * WARNING: must be called with preemption disabled!  调用时必须禁用抢占
 */
static void __sched notrace __schedule(bool preempt)
{
    struct task_struct *prev, *next;
    unsigned long *switch_count;
    unsigned long prev_state;
    struct rq_flags rf;
    struct rq *rq;
    int cpu;

    cpu = smp_processor_id();                           /* 获取当前cpu号 */
    rq = cpu_rq(cpu);                                   /* 获取当前cpu的runqueue */
    prev = rq->curr;                                    /* 将当前进程的描述符指针保存在prev变量中 */

    schedule_debug(prev, preempt);

    if (sched_feat(HRTICK))
        hrtick_clear(rq);

    local_irq_disable();
    rcu_note_context_switch(preempt);

    /*
     * Make sure that signal_pending_state()->signal_pending() below
     * can't be reordered with __set_current_state(TASK_INTERRUPTIBLE)
     * done by the caller to avoid the race with signal_wake_up():
     *
     * __set_current_state(@state)      signal_wake_up()
     * schedule()                     set_tsk_thread_flag(p, TIF_SIGPENDING)
     *                        wake_up_state(p, state)
     *   LOCK rq->lock              LOCK p->pi_state
```

```
     *   smp_mb__after_spinlock()           smp_mb__after_spinlock()
     *     if (signal_pending_state())      if (p->state & @state)
     *
     * Also, the membarrier system call requires a full memory barrier
     * after coming from user-space, before storing to rq->curr.
     */
    rq_lock(rq, &rf);
    smp_mb__after_spinlock();

    /* Promote REQ to ACT */
    rq->clock_update_flags <<= 1;
    update_rq_clock(rq);

    switch_count = &prev->nivcsw;

    /*
     * We must load prev->state once (task_struct::state is volatile), such
     * that:
     *
     *  - we form a control dependency vs deactivate_task() below.
     *  - ptrace_{,un}freeze_traced() can change ->state underneath us.
     */
    prev_state = prev->state;
    if (!preempt && prev_state) {
        if (signal_pending_state(prev_state, prev)) {
            prev->state = TASK_RUNNING;
        } else {
            prev->sched_contributes_to_load =
                (prev_state & TASK_UNINTERRUPTIBLE) &&
                !(prev_state & TASK_NOLOAD) &&
                !(prev->flags & PF_FROZEN);

            if (prev->sched_contributes_to_load)
                rq->nr_uninterruptible++;

            /*
             * __schedule()         ttwu()
             *   prev_state = prev->state;    if (p->on_rq && ...)
             *   if (prev_state)            goto out;
             *     p->on_rq = 0;          smp_acquire__after_ctrl_dep();
             *                p->state = TASK_WAKING
             *
             * Where __schedule() and ttwu() have matching control dependencies.
             *
             * After this, schedule() must not care about p->state any more.
             */
            deactivate_task(rq, prev, DEQUEUE_SLEEP | DEQUEUE_NOCLOCK);

            if (prev->in_iowait) {
                atomic_inc(&rq->nr_iowait);
                delayacct_blkio_start();
            }
        }
        switch_count = &prev->nvcsw;
    }

```

```
    next = pick_next_task(rq, prev, &rf);           /* 将下一个被调度的进程描述符指针存放在next变量中 */
    clear_tsk_need_resched(prev);                   /* 清除当前进程的TIF_NEED_RESCHED标志位 */
    clear_preempt_need_resched();                   /* 清除PREEMPT_NEED_RESCHED */

    if (likely(prev != next)) {
        rq->nr_switches++;
        /*
         * RCU users of rcu_dereference(rq->curr) may not see
         * changes to task_struct made by pick_next_task().
         */
        RCU_INIT_POINTER(rq->curr, next);
        /*
         * The membarrier system call requires each architecture
         * to have a full memory barrier after updating
         * rq->curr, before returning to user-space.
         *
         * Here are the schemes providing that barrier on the
         * various architectures:
         * - mm ? switch_mm() : mmdrop() for x86, s390, sparc, PowerPC.
         *   switch_mm() rely on membarrier_arch_switch_mm() on PowerPC.
         * - finish_lock_switch() for weakly-ordered
         *   architectures where spin_unlock is a full barrier,
         * - switch_to() for arm64 (weakly-ordered, spin_unlock
         *   is a RELEASE barrier),
         */
        ++*switch_count;

        psi_sched_switch(prev, next, !task_on_rq_queued(prev));

        trace_sched_switch(preempt, prev, next);                /* event事件 sched_switch */

        /* Also unlocks the rq: */
        rq = context_switch(rq, prev, next, &rf);           /* 当前进程和下一个进程的上下文进行切换*/
    } else {
        rq->clock_update_flags &= ~(RQCF_ACT_SKIP|RQCF_REQ_SKIP);
        rq_unlock_irq(rq, &rf);
    }

    balance_callback(rq);
}
```

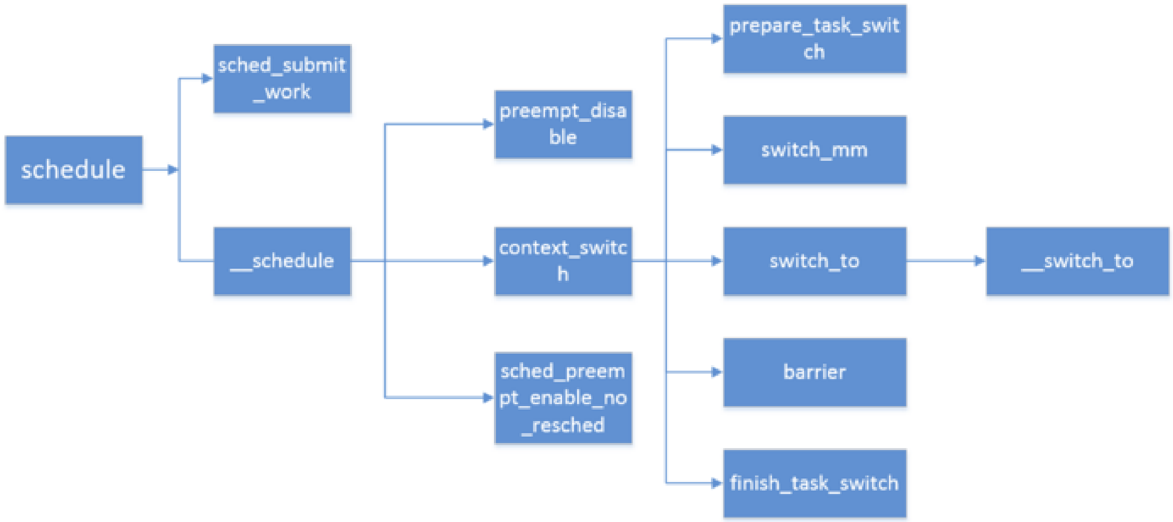
上下文切换context_switch()

上下文切换一般分为两个，一个是硬件上下文切换（指的是cpu寄存器，要把当前进程使用的寄存器内容保存下来，再把下一个程序的寄存器内容恢复），另一个是切换进程的地址空间（说白了就是程序代码）。进程的地址空间（程序代码）主要保存在进程描述符中struct mm_struct结构体中，因此该函数主要是操作这个结构体。

```
/*
 * context_switch - switch to the new MM and the new thread's register state.
 */
```

```c
static __always_inline struct rq *
context_switch(struct rq *rq, struct task_struct *prev,
	       struct task_struct *next, struct rq_flags *rf)
{
	prepare_task_switch(rq, prev, next);

	/*
	 * For paravirt, this is coupled with an exit in switch_to to
	 * combine the page table reload and the switch backend into
	 * one hypercall.
	 */
	arch_start_context_switch(prev);

	/*
	 * kernel -> kernel   lazy + transfer active
	 *   user -> kernel   lazy + mmgrab() active
	 *
	 * kernel ->   user   switch + mmdrop() active
	 *   user ->   user   switch
	 */
	if (!next->mm) {                                // to kernel
		enter_lazy_tlb(prev->active_mm, next);

		next->active_mm = prev->active_mm;
		if (prev->mm)                           // from user
			mmgrab(prev->active_mm);
		else
			prev->active_mm = NULL;
	} else {                                        // to user
		membarrier_switch_mm(rq, prev->active_mm, next->mm);
		/*
		 * sys_membarrier() requires an smp_mb() between setting
		 * rq->curr / membarrier_switch_mm() and returning to userspace.
		 *
		 * The below provides this either through switch_mm(), or in
		 * case 'prev->active_mm == next->mm' through
		 * finish_task_switch()'s mmdrop().
		 */
		switch_mm_irqs_off(prev->active_mm, next->mm, next);

		if (!prev->mm) {                        // from kernel
			/* will mmdrop() in finish_task_switch(). */
			rq->prev_mm = prev->active_mm;
			prev->active_mm = NULL;
		}
	}

	rq->clock_update_flags &= ~(RQCF_ACT_SKIP|RQCF_REQ_SKIP);

	prepare_lock_switch(rq, next, rf);

	/* Here we just switch the register state and the stack. */
	switch_to(prev, next, prev);
	barrier();

	return finish_task_switch(prev);
}
```

网上有个4.x的内核代码，这个和5.9稍微有点不一样，大体流程还是差不多



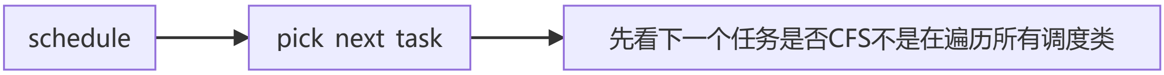
5.2.3 linux-5.9.10调度器框架

调度类的顺序

```
/* vmlinux.lds.h文件中定义
 * The order of the sched class addresses are important, as they are
 * used to determine the order of the priority of each sched class in
 * relation to each other.这里固定死了调度类的优先级顺序，stop > deadline > rt > fair > idle
 */
#define SCHED_DATA              \
    STRUCT_ALIGN();             \
    __begin_sched_classes = .;      \
    *(__idle_sched_class)           \
    *(__fair_sched_class)           \
    *(__rt_sched_class)         \
    *(__dl_sched_class)         \
    *(__stop_sched_class)           \
    __end_sched_classes = .;
```

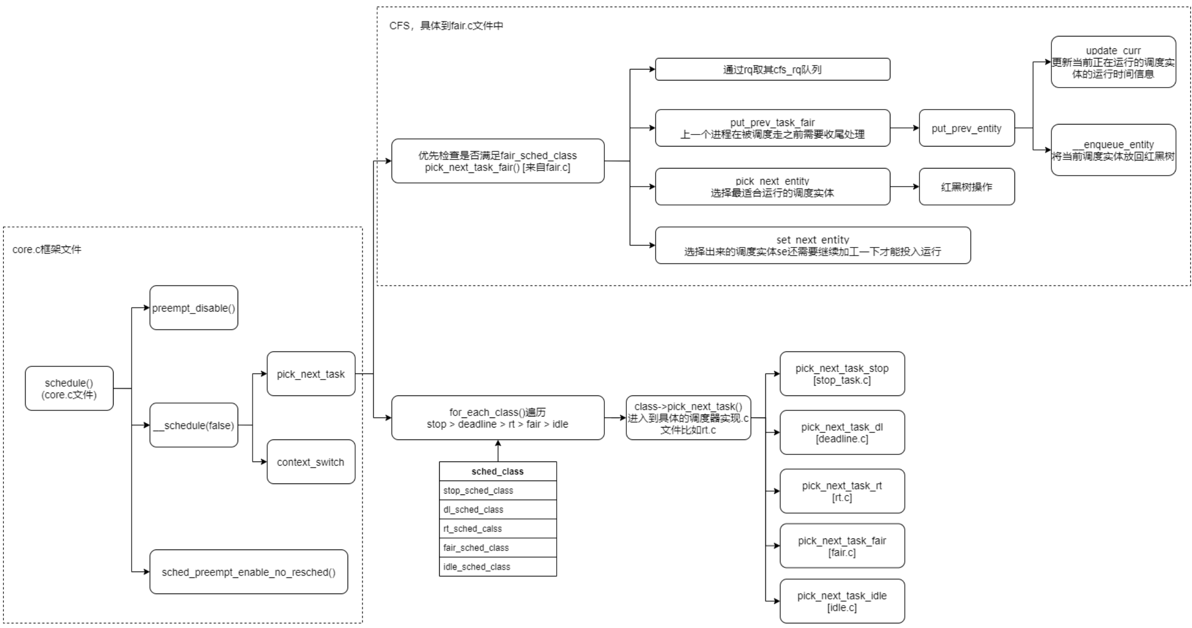
```
/*kernel/sched/sched.h中定义*/
/* Defined in include/asm-generic/vmlinux.lds.h */
extern struct sched_class __begin_sched_classes[];
extern struct sched_class __end_sched_classes[];

#define sched_class_highest (__end_sched_classes - 1)
#define sched_class_lowest  (__begin_sched_classes - 1)   /*此处为何是begin -1 ？ 难道不是 +1 ?*/

#define for_class_range(class, _from, _to) \
```

```
    for (class = (_from); class != (_to); class--)

#define for_each_class(class) \
    for_class_range(class, sched_class_highest, sched_class_lowest) /*从stop遍历至idle*/

extern const struct sched_class stop_sched_class;
extern const struct sched_class dl_sched_class;
extern const struct sched_class rt_sched_class;
extern const struct sched_class fair_sched_class;
extern const struct sched_class idle_sched_class;
```

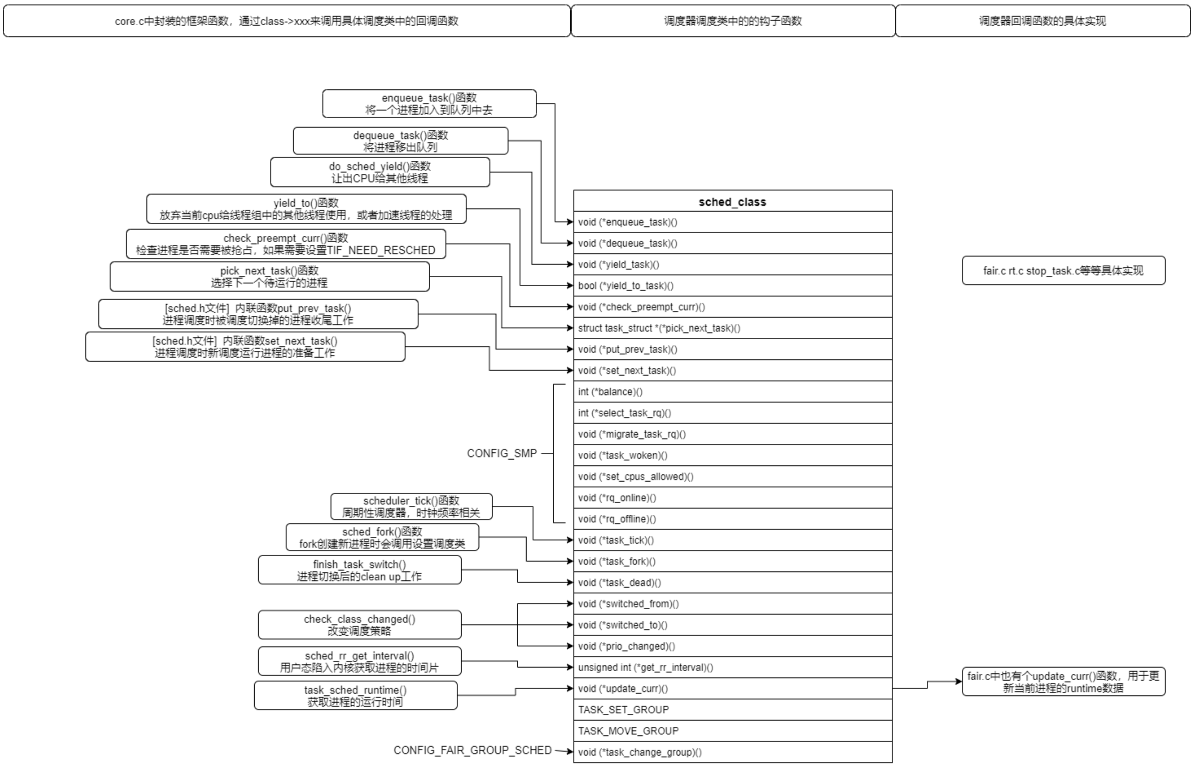


linux5.9.10中调度器的主框架core.c是如何与CFS、实时rt、deadline、idle等具体的调度器实现整合起来的？

```
/*
 * Scheduling policies
 */
#define SCHED_NORMAL        0
#define SCHED_FIFO      1
#define SCHED_RR        2
#define SCHED_BATCH     3
/* SCHED_ISO: reserved but not implemented yet */                  /*MuQSS使用的是该policy*/
#define SCHED_IDLE      5
#define SCHED_DEADLINE      6
```



5.2.4 sched_class连接调度器框架和调度器



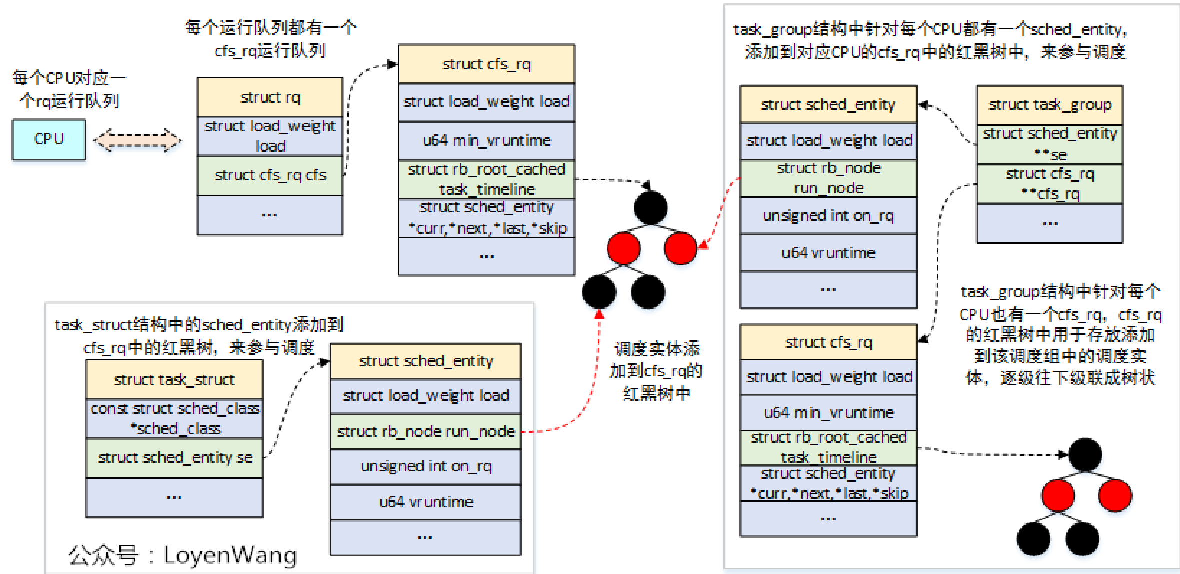
六、CFS

```
struct cfs_rq {
    struct load_weight  load;
    unsigned int        nr_running;
    unsigned int        h_nr_running;      /* SCHED_{NORMAL,BATCH,IDLE} */
    unsigned int        idle_h_nr_running; /* SCHED_IDLE */

    u64         exec_clock;
    u64         min_vruntime;

    struct rb_root_cached   tasks_timeline; /**/

    /*
     * 'curr' points to currently running entity on this cfs_rq.
     * It is set to NULL otherwise (i.e when none are currently running).
     */
    struct sched_entity *curr;
    struct sched_entity *next;
    struct sched_entity *last;
    struct sched_entity *skip;



#ifdef CONFIG_SMP
    /*
```

```
         * CFS load tracking
         */
        struct sched_avg    avg;

        struct {
            raw_spinlock_t  lock ____cacheline_aligned;
            int      nr;
            unsigned long   load_avg;
            unsigned long   util_avg;
            unsigned long   runnable_avg;
        } removed;
    #endif /* CONFIG_SMP */

    }
```

进程调度过程分为两部分，一是对进程信息进行修改，主要是修改和调度相关的信息，比如进程的运行时间，睡眠时间，进程的状态，cpu的负荷等等，二是进程的切换。和进程调度相关的所有函数中，只有schedule函数是用来进行进程切换的，其他函数都是用来修改进程的调度信息。

rq、cfs_rq、task_group、task_struct之间的关系：

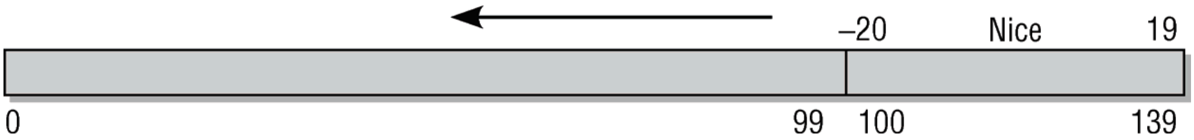


6.1 进程优先级

程的优先级和调度关系密切，计算进程的虚拟运行时间要用到优先级，优先级决定进程权重，权重决定进程虚拟时间的增加速度，最终决定进程可运行时间的长短。权重越大的进程可以执行的时间越长。

6.1.1 优先级

在用户空间可以通过nice命令设置进程的静态优先级，这在内部会调用nice系统调用。 进程的nice值在□20和+19之间（包含）。值越低，表明优先级越高。



内核使用一个简单些的数值范围，从0到139（包含），用来表示内部优先级。同样是值越低，优先级越高。从0到99的范围专供实时进程使用。nice值[-20, +19]映射到范围100到139。**实时进程的优先级总是比普通进程更高。**

```
#define MAX_NICE    19
#define MIN_NICE    -20
#define NICE_WIDTH  (MAX_NICE - MIN_NICE + 1)

#define MAX_USER_RT_PRIO    100
#define MAX_RT_PRIO     MAX_USER_RT_PRIO

#define MAX_PRIO        (MAX_RT_PRIO + NICE_WIDTH)
#define DEFAULT_PRIO        (MAX_RT_PRIO + NICE_WIDTH / 2)
```

```
static int effective_prio(struct task_struct *p)
{
    p->normal_prio = normal_prio(p);
    /*如果是实时进程或已经提高到实时优先级，则保持优先级不变。否则，返回普通优先级：*/
    if (!rt_prio(p->prio))
        return p->normal_prio;
    return p->prio;
}
```

该函数用于设置进程的优先级，该函数设计的有一定技巧性，函数的返回值是用来设置进程的活动优先级，但是在函数体中也把进程的普通优先级设置了。

假定我们在处理普通进程，不涉及实时调度。在这种情况下，normal_prio只是返回静态优先级。结果很简单：所有3个优先级都是同一个值，即静态优先级！

```
static inline int normal_prio(struct task_struct *p)  /* 获取普通优先级 */
{
    int prio;

    if (task_has_dl_policy(p))  /* 判断当前进程是否空闲进程，是则设置进程的普通优先级-1*/
        prio = MAX_DL_PRIO-1;
    else if (task_has_rt_policy(p))  /* 判断是否实时进程，是则设置实时进程普通优先级0-99（越小优先级越高）*/
        prio = MAX_RT_PRIO-1 - p->rt_priority;
    else
        prio = __normal_prio(p);        /* 普通进程的普通优先级等于其静态优先级 */
    return prio;
}
```

其中，第8行，看到这块减去了p->rt_priority，比较奇怪，这是因为实时进程描述符的rt_priority成员中事先存放了它自己的优先级（数字也是0-99，但在这里数字越大，优先级越高），因此往p->prio中倒换的时候，需要处理一下，MAX_RT_PRIO值为100，因此MAX_RT_PRIO-1-（0，99）就倒换成了（99，0），这仅仅是个小技巧。

6.1.2 权重

进程的重要性不仅是由优先级指定的，而且还需要考虑保存在task_struct->se.load的负荷权重。

`include/linux/sched.h`中定义了权重的结构体：

```
struct load_weight {
    unsigned long           weight;
    u32             inv_weight;
};
```

```
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,
};
```

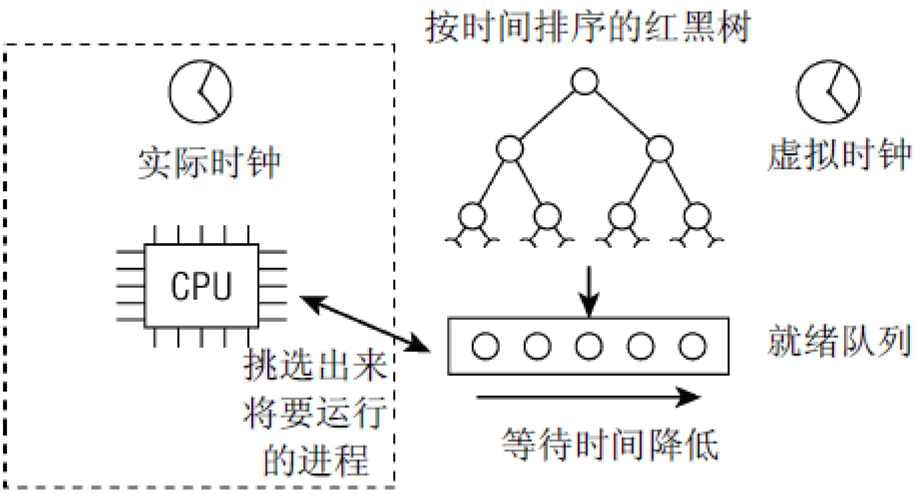
进程每降低一个nice值，则多获得10%的CPU时间，每升高一个nice值，则放弃10%的CPU时间。对内核使用的范围[0, 39]中的每个nice级别，该数组中都有一个对应项。各数组之间的乘数因子是1.25。

设置权重

```
static 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;   /* 权重保存在task_struct的se.load中 */

    /*SCHED_IDLE进程得到的权重最小：*/
    if (task_has_idle_policy(p)) {
        load->weight = scale_load(WEIGHT_IDLEPRIO);
        load->inv_weight = WMULT_IDLEPRIO;
        return;
    }

    /*
     * SCHED_OTHER tasks have to update their load when changing their
     * weight
     */
    if (update_load && p->sched_class == &fair_sched_class) {
        reweight_task(p, prio);
    } else {
        load->weight = scale_load(sched_prio_to_weight[prio]);
```

```
            load->inv_weight = sched_prio_to_wmult[prio];
        }
}

```

虚拟时间



所有的可运行进程都按时间在一个红黑树中排序，所谓时间即其等待时间。等待CPU时间最长的进程是最左侧的项，调度器下一次会考虑该进程。等待时间稍短的进程在该树上从左至右排序。

完全公平调度算法依赖于虚拟时钟，用以度量等待进程在完全公平系统中所能得到的CPU时间。

所有与虚拟时钟有关的计算都在update_curr中执行，该函数在系统中各个不同地方调用，包括周期性调度器之内

```
/*
 * Update the current task's runtime statistics.
 */
static void update_curr(struct cfs_rq *cfs_rq)
{
    struct sched_entity *curr = cfs_rq->curr;
    u64 now = rq_clock_task(rq_of(cfs_rq));     /* 从就绪队列rq的clock_task成员中获取当前时间 */
    u64 delta_exec;

    if (unlikely(!curr))
        return;

    delta_exec = now - curr->exec_start;     /*当前时间减去进程上次时钟中断tick中开始时间得到进程运行的时间间隔*/
    if (unlikely((s64)delta_exec <= 0))
        return;

    curr->exec_start = now;                  /* 当前时间赋值给进程新的开始时间 */

    schedstat_set(curr->statistics.exec_max,
              max(delta_exec, curr->statistics.exec_max));

```

```
    /*将进程运行的时间间隔delta_exec累加到调度实体的sum_exec_runtime成员中，该成员代表进程到目前为止运行了多长时间*/
    curr->sum_exec_runtime += delta_exec;
    schedstat_add(cfs_rq->exec_clock, delta_exec);  /*将进程运行的时间间隔delta_exec也累加到公平调度就绪队列cfs_rq的exec_clock成员中*/

    /*calc_delta_fair函数很关键，它将进程执行的真实运行时间转换成虚拟运行时间，然后累加到调度实体的vruntime域中*/
    curr->vruntime += calc_delta_fair(delta_exec, curr);
    update_min_vruntime(cfs_rq); /*更新cfs_rq队列中的最小虚拟运行时间min_vruntime，该时间是就绪队列中所有进程包括当前进程的已运行的最小虚拟时间，只能单调递增*/

    if (entity_is_task(curr)) {
        struct task_struct *curtask = task_of(curr);

        trace_sched_stat_runtime(curtask, delta_exec, curr->vruntime);
        cgroup_account_cputime(curtask, delta_exec);
        account_group_exec_runtime(curtask, delta_exec);
    }

    account_cfs_rq_runtime(cfs_rq, delta_exec);
}

```

每个cfs_rq队列均有一个min_vruntime成员，装的是就绪队列中所有进程包括当前进程已运行的虚拟时间中最小的那个时间。`update_min_vruntime`用于更新该时间。

队列中的min_vruntime成员非常重要，用于在睡眠进程被唤醒后以及新进程被创建好时，进行虚拟时间补偿或者惩罚

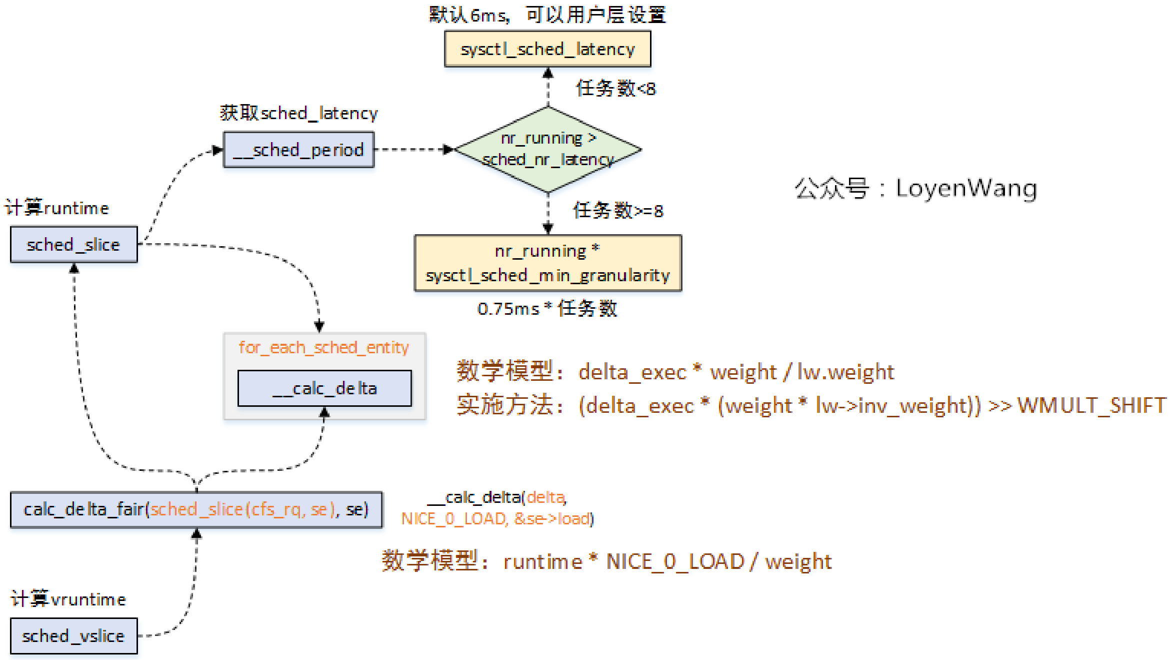
```
static void update_min_vruntime(struct cfs_rq *cfs_rq)
{
    struct sched_entity *curr = cfs_rq->curr;
    struct rb_node *leftmost = rb_first_cached(&cfs_rq->tasks_timeline);

    u64 vruntime = cfs_rq->min_vruntime;

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

    if (leftmost) { /* non-empty tree */  /*就绪队列中有下一个要被调度的进程，则进入下一个调度实体*/
        struct sched_entity *se;
        se = rb_entry(leftmost, struct sched_entity, run_node);

        /*从当前进程和下个被调度进程中，选择最小的已运行虚拟时间，保存到vruntime中*/
        if (!curr)
            vruntime = se->vruntime;
        else
            vruntime = min_vruntime(vruntime, se->vruntime);
```

```
    }

    /*从当前队列的min_vruntime域和vruntime变量中，选最大的保存到队列的min_vruntime域中，完成更新*/
    /* ensure we never gain time by being placed backwards. */
    cfs_rq->min_vruntime = max_vruntime(cfs_rq->min_vruntime, vruntime);
#ifndef CONFIG_64BIT
    smp_wmb();
    cfs_rq->min_vruntime_copy = cfs_rq->min_vruntime;
#endif
}
```

```
static inline u64 calc_delta_fair(u64 delta, struct sched_entity *se)
{
    if (unlikely(se->load.weight != NICE_0_LOAD))
        delta = __calc_delta(delta, NICE_0_LOAD, &se->load);

    return delta;
}

static u64 __calc_delta(u64 delta_exec, unsigned long weight, struct load_weight *lw)
{
    u64 fact = scale_load_down(weight);
    int shift = WMULT_SHIFT;

    __update_inv_weight(lw);

    if (unlikely(fact >> 32)) {
        while (fact >> 32) {
            fact >>= 1;
            shift--;
        }
    }

    fact = mul_u32_u32(fact, lw->inv_weight);

    while (fact >> 32) {
        fact >>= 1;
        shift--;
    }

    return mul_u64_u32_shr(delta_exec, fact, shift);
}
```

虚拟时间到底怎么一回事?

`sched_vslice`函数计算虚拟时间



负载

内核中计算CPU负载的方法是PELT(Per-Entity Load Tracing)，不仅考虑进程权重，而且跟踪每个调度实体的负载情况。

sched_entity结构中有一个struct sched_avg用于描述进程的负载

```
struct sched_avg {
	u64				last_update_time;
	u64				load_sum;
	u64				runnable_sum;
	u32				util_sum;
	u32				period_contrib;
	unsigned long			load_avg;
	unsigned long			runnable_avg;
	unsigned long			util_avg;
	struct util_est			util_est;
} ____cacheline_aligned;
```

6.1.3 选择下一个进程

```
/*
 * Pick up the highest-prio task:
 */
static inline struct task_struct *
pick_next_task(struct rq *rq, struct task_struct *prev, struct rq_flags *rf)
{
	const struct sched_class *class;
```

```c
    struct task_struct *p;

    if (likely(prev->sched_class <= &fair_sched_class &&
           rq->nr_running == rq->cfs.h_nr_running)) {

        p = pick_next_task_fair(rq, prev, rf);
        if (unlikely(p == RETRY_TASK))
            goto restart;

        /* Assumes fair_sched_class->next == idle_sched_class */
        if (!p) {
            put_prev_task(rq, prev);
            p = pick_next_task_idle(rq);
        }

        return p;
    }

restart:
    put_prev_task_balance(rq, prev, rf);

    for_each_class(class) {
        p = class->pick_next_task(rq);
        if (p)
            return p;
    }

    /* The idle class should always have a runnable task: */
    BUG();
}





/**主要是pick_next_task_fair函数**/
struct task_struct *
pick_next_task_fair(struct rq *rq, struct task_struct *prev, struct rq_flags *rf)
{
    struct cfs_rq *cfs_rq = &rq->cfs;
    struct sched_entity *se;
    struct task_struct *p;
    int new_tasks;

again:
    if (!sched_fair_runnable(rq))
        goto idle;

#ifdef CONFIG_FAIR_GROUP_SCHED
    if (!prev || prev->sched_class != &fair_sched_class)
        goto simple;

    do {
        struct sched_entity *curr = cfs_rq->curr;

        if (curr) {
            if (curr->on_rq)
```

```
                update_curr(cfs_rq);
            else
                curr = NULL;

            if (unlikely(check_cfs_rq_runtime(cfs_rq))) {
                cfs_rq = &rq->cfs;

                if (!cfs_rq->nr_running)
                    goto idle;

                goto simple;
            }
        }

        se = pick_next_entity(cfs_rq, curr);
        cfs_rq = group_cfs_rq(se);
    } while (cfs_rq);  /*对所有的调度组进行遍历，从中选择下一个可调度的进程，而不只局限在当前队列的当前组*/

    p = task_of(se);

    if (prev != p) {
        struct sched_entity *pse = &prev->se;

        while (!(cfs_rq = is_same_group(se, pse))) {
            int se_depth = se->depth;
            int pse_depth = pse->depth;

            if (se_depth <= pse_depth) {
                put_prev_entity(cfs_rq_of(pse), pse);
                pse = parent_entity(pse);
            }
            if (se_depth >= pse_depth) {
                set_next_entity(cfs_rq_of(se), se);
                se = parent_entity(se);
            }
        }

        put_prev_entity(cfs_rq, pse);
        set_next_entity(cfs_rq, se);
    }

    goto done;
simple:
#endif
    if (prev)
        put_prev_task(rq, prev);

    do {
        se = pick_next_entity(cfs_rq, NULL);
        set_next_entity(cfs_rq, se);
        cfs_rq = group_cfs_rq(se);
    } while (cfs_rq);

    p = task_of(se);

done: __maybe_unused;
#ifdef CONFIG_SMP
```

```
    /*
     * Move the next running task to the front of
     * the list, so our cfs_tasks list becomes MRU
     * one.
     */
    list_move(&p->se.group_node, &rq->cfs_tasks);
#endif

    if (hrtick_enabled(rq))
        hrtick_start_fair(rq, p);

    update_misfit_status(p, rq);

    return p;

idle:
    if (!rf)
        return NULL;

    new_tasks = newidle_balance(rq, rf);

    /*
     * Because newidle_balance() releases (and re-acquires) rq->lock, it is
     * possible for any higher priority task to appear. In that case we
     * must re-start the pick_next_entity() loop.
     */
    if (new_tasks < 0)
        return RETRY_TASK;

    if (new_tasks > 0)
        goto again;

    /*
     * rq is about to be idle, check if we need to update the
     * lost_idle_time of clock_pelt
     */
    update_idle_rq_clock_pelt(rq);

    return NULL;
}
```

6.1.4 就绪队列的入队和出队

enqueue_task_fair()函数

CFS的enqueue_task钩子函数是`enqueue_task_fair()`函数:

```
/*
 * 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:
 */
/**nr_running是cfs_rq结构体中的成员，计数所有就绪的进程数包括cfs_rq中以及正在运行的进程**/
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;       /*获取进程p的调度实体*/
    int idle_h_nr_running = task_has_idle_policy(p);    /*判断进程 p->policy == SCHED_IDLE，有何作用？/

    /*
     * The code below (indirectly) updates schedutil which looks at
     * the cfs_rq utilization to select a frequency.
     * Let's add the task's estimated utilization to the cfs_rq's
     * estimated utilization, before we update schedutil.
     */
    util_est_enqueue(&rq->cfs, p);  /*cfs的负载估算，占用cpu的算力负载估算*/

    /*
     * 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.
     */
    if (p->in_iowait)
        cpufreq_update_util(rq, SCHED_CPUFREQ_IOWAIT);

    /*这段代码在做什么？*/
    for_each_sched_entity(se) {
        if (se->on_rq) /*判断进程是否已经在队列里，on_rq为1则不需要再加入队列了，已经存在队列里*/
            break;
        cfs_rq = cfs_rq_of(se);
        enqueue_entity(cfs_rq, se, flags);   /*将调度实体加入队列*/

        cfs_rq->h_nr_running++;             /*计数增加*/
        cfs_rq->idle_h_nr_running += idle_h_nr_running;

        /* end evaluation on encountering a throttled cfs_rq */
        if (cfs_rq_throttled(cfs_rq))
            goto enqueue_throttle;

        flags = ENQUEUE_WAKEUP;
    }

    /*为什么需要两次循环，操作不同在什么地方呢？*/
    for_each_sched_entity(se) {
        cfs_rq = cfs_rq_of(se);

        update_load_avg(cfs_rq, se, UPDATE_TG);
        se_update_runnable(se);
        update_cfs_group(se);

        cfs_rq->h_nr_running++;
        cfs_rq->idle_h_nr_running += idle_h_nr_running;

        /* end evaluation on encountering a throttled cfs_rq */
        if (cfs_rq_throttled(cfs_rq))
            goto enqueue_throttle;

               /*
                * One parent has been throttled and cfs_rq removed from the
                * list. Add it back to not break the leaf list.
```

```
                */
               if (throttled_hierarchy(cfs_rq))
                       list_add_leaf_cfs_rq(cfs_rq);
    }

    /* At this point se is NULL and we are at root level*/
    add_nr_running(rq, 1);

    /*
     * Since new tasks are assigned an initial util_avg equal to
     * half of the spare capacity of their CPU, tiny tasks have the
     * ability to cross the overutilized threshold, which will
     * result in the load balancer ruining all the task placement
     * done by EAS. As a way to mitigate that effect, do not account
     * for the first enqueue operation of new tasks during the
     * overutilized flag detection.
     *
     * A better way of solving this problem would be to wait for
     * the PELT signals of tasks to converge before taking them
     * into account, but that is not straightforward to implement,
     * and the following generally works well enough in practice.
     */
    if (flags & ENQUEUE_WAKEUP)
        update_overutilized_status(rq);

enqueue_throttle:
    if (cfs_bandwidth_used()) {
        /*
         * When bandwidth control is enabled; the cfs_rq_throttled()
         * breaks in the above iteration can result in incomplete
         * leaf list maintenance, resulting in triggering the assertion
         * below.
         */
        for_each_sched_entity(se) {
            cfs_rq = cfs_rq_of(se);

            if (list_add_leaf_cfs_rq(cfs_rq))
                break;
        }
    }

    assert_list_leaf_cfs_rq(rq);

    hrtick_update(rq);
}
```

enqueue_task_fair主要职责:

1）更新运行时的数据，比如负载、权重、组调度的占比等等;

2）将sched_entity插入红黑树;

将调度实体入队红黑树。

```
/*
 * Enqueue an entity into the rb-tree:
 */
static void __enqueue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se)
{
    struct rb_node **link = &cfs_rq->tasks_timeline.rb_root.rb_node;  /*获取就绪队列中红黑树的根节点*/
    struct rb_node *parent = NULL;      /* 用于指向树根*/
    struct sched_entity *entry;
    bool leftmost = true;

    /*
     * Find the right place in the rbtree:
     */
    while (*link) {
        parent = *link;
        entry = rb_entry(parent, struct sched_entity, run_node); /*获得树根节点的调度实体*/
```

```

        /*比较要入队的实体中的已运行虚拟时间和树根实体中的该信息，如果前者小的话，就要插入到树的左子树上（link指向树根的左孩子，再次进入循环，类似于递归），否则就要插入到树的右子树上（同上）。这块就将进程的调度策略展现的淋漓尽致：根据进程已运行的虚拟时间来决定进程的调度，红黑树的左子树比右子树要先被调度，已运行的虚拟时间越小的进程越在树的左侧*/
        if (entity_before(se, entry)) {
            link = &parent->rb_left;
        } else {
            link = &parent->rb_right;
            leftmost = false;
        }
    }

    rb_link_node(&se->run_node, parent, link); /*红黑树重新着色*/
    rb_insert_color_cached(&se->run_node,
                   &cfs_rq->tasks_timeline, leftmost);
}
```

dequeue_task_fair()函数

```
/*
 * The dequeue_task method is called before nr_running is
 * decreased. We remove the task from the rbtree and
 * update the fair scheduling stats:
 */
static void dequeue_task_fair(struct rq *rq, struct task_struct *p, int flags)
{
    struct cfs_rq *cfs_rq;
    struct sched_entity *se = &p->se;
    int task_sleep = flags & DEQUEUE_SLEEP;
    int idle_h_nr_running = task_has_idle_policy(p);
    bool was_sched_idle = sched_idle_rq(rq);

    for_each_sched_entity(se) {
        cfs_rq = cfs_rq_of(se);
        dequeue_entity(cfs_rq, se, flags);

        cfs_rq->h_nr_running--;
        cfs_rq->idle_h_nr_running -= idle_h_nr_running;

        /* end evaluation on encountering a throttled cfs_rq */
        if (cfs_rq_throttled(cfs_rq))
            goto dequeue_throttle;

        /* Don't dequeue parent if it has other entities besides us */
        if (cfs_rq->load.weight) {
            /* Avoid re-evaluating load for this entity: */
            se = parent_entity(se);
            /*
             * Bias pick_next to pick a task from this cfs_rq, as
             * p is sleeping when it is within its sched_slice.
             */
            if (task_sleep && se && !throttled_hierarchy(cfs_rq))
                set_next_buddy(se);
```

```
            break;
        }
        flags |= DEQUEUE_SLEEP;
    }

    for_each_sched_entity(se) {
        cfs_rq = cfs_rq_of(se);

        update_load_avg(cfs_rq, se, UPDATE_TG);
        se_update_runnable(se);
        update_cfs_group(se);

        cfs_rq->h_nr_running--;
        cfs_rq->idle_h_nr_running -= idle_h_nr_running;

        /* end evaluation on encountering a throttled cfs_rq */
        if (cfs_rq_throttled(cfs_rq))
            goto dequeue_throttle;

    }

    /* At this point se is NULL and we are at root level*/
    sub_nr_running(rq, 1);

    /* balance early to pull high priority tasks */
    if (unlikely(!was_sched_idle && sched_idle_rq(rq)))
        rq->next_balance = jiffies;

dequeue_throttle:
    util_est_dequeue(&rq->cfs, p, task_sleep);
    hrtick_update(rq);
}
```

dequeue_task_fair的主要工作内容和enqueue其实类型:

1）更新运行时间、负载等;

2）将实体移出红黑树队列;

出队列

```
static void __dequeue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se)
{
    rb_erase_cached(&se->run_node, &cfs_rq->tasks_timeline);
}

static inline void rb_erase_cached(struct rb_node *node,
                   struct rb_root_cached *root)
{
    if (root->rb_leftmost == node)    /*判断要出队的实体是不是红黑树最左侧的孩子（rb_leftmost所指向的）*/
        root->rb_leftmost = rb_next(node); /*是最左子树的话需要找出下一个*/
    rb_erase(node, &root->rb_root);
}
```

6.1.5睡眠进程被唤醒后抢占当前进程

当某个资源空出来后，等待该资源的进程就会被唤醒，唤醒后也许就要抢占当前进程。

该函数会唤醒睡眠中的指定p的进程。

```
static int
try_to_wake_up(struct task_struct *p, unsigned int state, int wake_flags)
{
    [...]
}
```

唤醒一个刚被创建的进程

```
/*
 * 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)
{
    [...]
}
```

检查唤醒进程是否能抢占当前进程.

```
/*
 * Preempt the current task with a newly woken task if needed:
 */
static void check_preempt_wakeup(struct rq *rq, struct task_struct *p, int wake_flags)
{
    [...]
}
```

6.1.6 fork的处理

该函数在do_fork--->copy_process函数中调用，用来设置新创建进程的虚拟时间信息。

```
/*
 * 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();
    struct rq_flags rf;

    rq_lock(rq, &rf);
    update_rq_clock(rq);

    cfs_rq = task_cfs_rq(current);
    curr = cfs_rq->curr;
    if (curr) {
        update_curr(cfs_rq);
        se->vruntime = curr->vruntime;      /*当前进程（父进程）的虚拟运行时间拷贝给新进程（子进程)*/
    }
    place_entity(cfs_rq, se, 1);            /*完成新进程的“时间片”计算以及虚拟时间惩罚，之后将新进程加入红黑树中*/

    /*如果设置了子进程先于父进程运行的标志并且当前进程不为空且当前进程已运行的虚拟时间比新进程小，则执行if体*/
    if (sysctl_sched_child_runs_first && curr && entity_before(curr, se)) {
        /*交换当前进程和新进程的虚拟时间（新进程的虚拟时间变小，就排在了红黑树的左侧，当前进程之前，下次就能被调度）*/
        swap(curr->vruntime, se->vruntime);
        resched_curr(rq); /*设置重新调度标志*/
    }

    se->vruntime -= cfs_rq->min_vruntime; /*给新进程的虚拟运行时间减去队列的最小虚拟时间来做一点补偿（因为在上边的place_entity函数中给新进程的虚拟时间加了一次min_vruntime，所以在这里要减去）*/
    rq_unlock(rq, &rf);
}
```

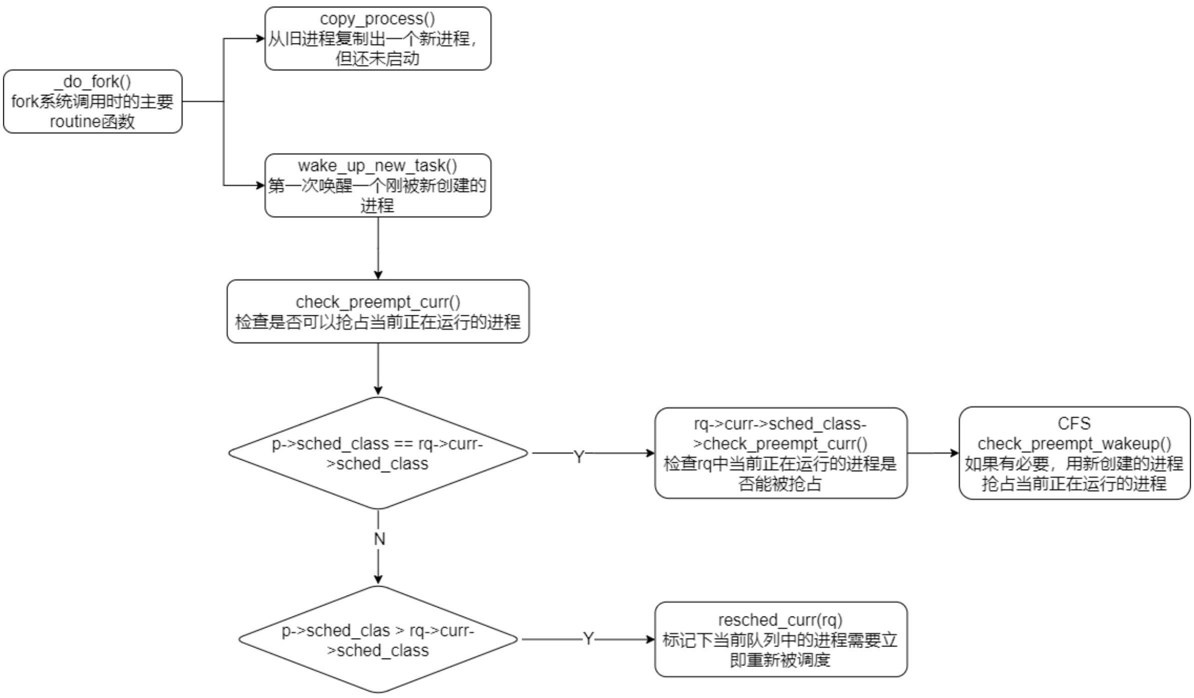
看下place_entity函数，该函数完成新进程的“时间片”计算和虚拟时间惩罚，并且将新进程加入就绪队列。

```
static void
place_entity(struct cfs_rq *cfs_rq, struct sched_entity *se, int initial)
{
    u64 vruntime = cfs_rq->min_vruntime;

    /*如果initial标志为1的话（说明当前计算的是新进程的时间），将计算出的新进程的虚拟时间片累加到vruntime中，累加到原因是调度系统要保证先把就绪队列中的所有的进程执行一遍之后才能执行新进程*/
    if (initial && sched_feat(START_DEBIT))
        vruntime += sched_vslice(cfs_rq, se);

    /* sleeps up to a single latency don't count. */
    if (!initial) {             /*如果当前计算的不是新进程（睡眠的进程），把一个延迟周期的长度sysctl_sched_latency（6ms）赋给thresh*/
        unsigned long thresh = sysctl_sched_latency;

        /*
         * Halve their sleep time's effect, to allow
```

```
         * for a gentler effect of sleepers:
         */
        if (sched_feat(GENTLE_FAIR_SLEEPERS))
            thresh >>= 1;           /*thresh减半*/

        vruntime -= thresh; /*睡眠进程的虚拟运行时间减去减半后的thresh，因为睡眠进程好长时间未运行，因此要进行虚拟时间补偿，把它已运行的虚拟时间减小一点，使得它能多运行一会*/
    }

    /* ensure we never gain time by being placed backwards. */
    se->vruntime = max_vruntime(se->vruntime, vruntime);          /*将设置好的虚拟时  间保存到进程调度实体的vruntime域*/
}
```

为什么要对新进程进行虚拟时间惩罚，其实原因只有一个，就是调度系统要保证将就绪队列中现有的进程执行一遍之后再执行新进程，那么就必须使新进程的 vruntime=cfs_rq->min_vruntime+新进程的虚拟时间片，才能使得新进程插入到红黑树的右边，最后参与调度，不然无法保证所有进程在新进程之前执行。

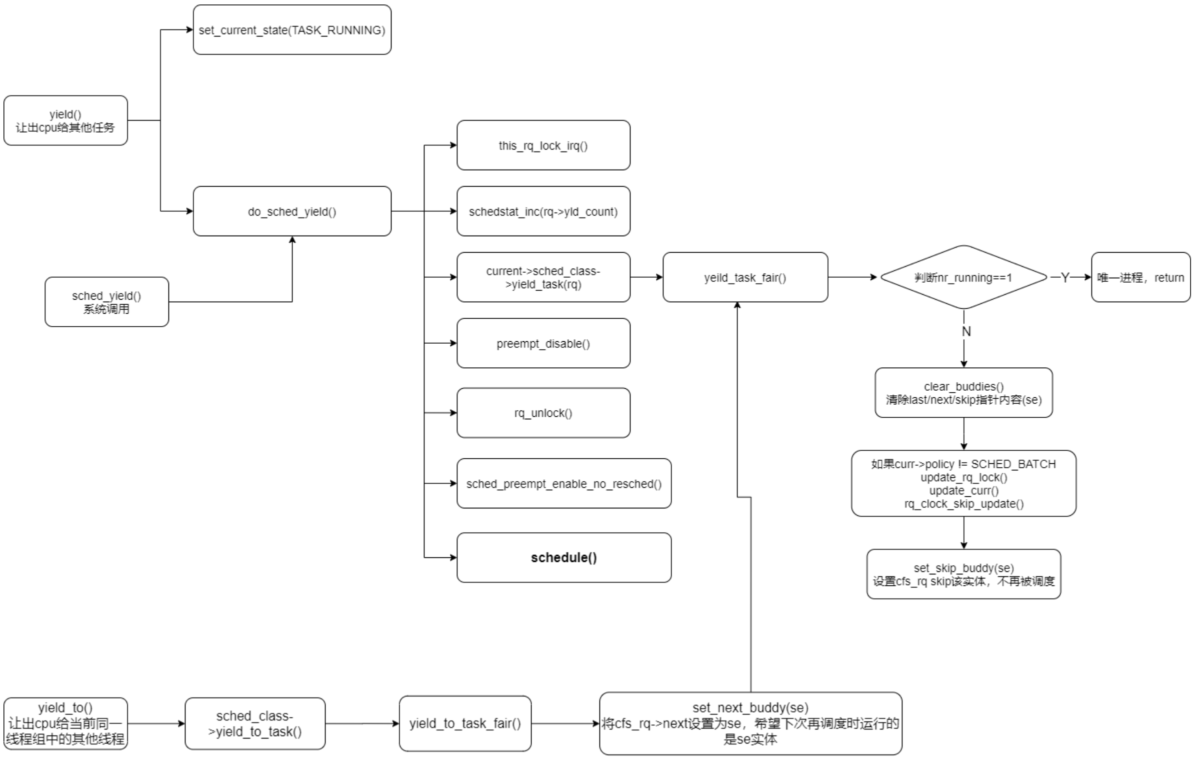
check_preemp_curr()函数



6.1.7 让出cpu

两个yield_task相关函数:

- yield()让出cpu给其他线程
- yield_to()让出cpu给同一线程去的其他线程（是否是同一队列？）



七、MuQSS调度器

查看MuQSS调度器是否启用。

```
/ # dmesg | grep schedule
[    0.000000] rcu: RCU calculated value of scheduler-enlistment delay is 25 jiffies.
[    1.637563] io scheduler mq-deadline registered
[    2.446012] MuQSS CPU scheduler v0.204 by Con Kolivas.
```

全称：The Multiple Queue Skiplist Scheduler

疑问1：这里的skiplist是什么？

7.1 skiplist跳表

skiplist是一种数据结构，类似CFS的红黑树，但比红黑树简单，比双链表更高效。

跳跃表使用概率均衡技术而不是使用强制性均衡，因此，对于插入和删除结点比传统上的平衡树算法更为简洁高效。

跳表参考这篇博客：https://blog.csdn.net/ict2014/article/details/17394259

7.2 MuQSS简介

参考这篇博客https://cloud.tencent.com/developer/article/1517909

BFS虽然简单，但是两个问题却非常明显：

1. 遍历查找的O(n)问题。链表为什么不基于Virtual Deadline进行预排序呢？

2. 多CPU操作全局链表的锁问题。

我们看看BFS的算法简单到何种程度：

- task插入：直接将task插入链表末尾。
- task选择：冒泡选择Virtual Deadline最小的task。【在遍历过程中会有trick，发现当前jiffies大于task的VD，就退出，这像极了Linux内核的timer处理】

最终，Con Kolivas认为：

1. 在task数量并不太大的情况下，O(n)算法没有任何问题。
2. 在CPU数量保持在16个以内时，争锁的开销可以忽略。

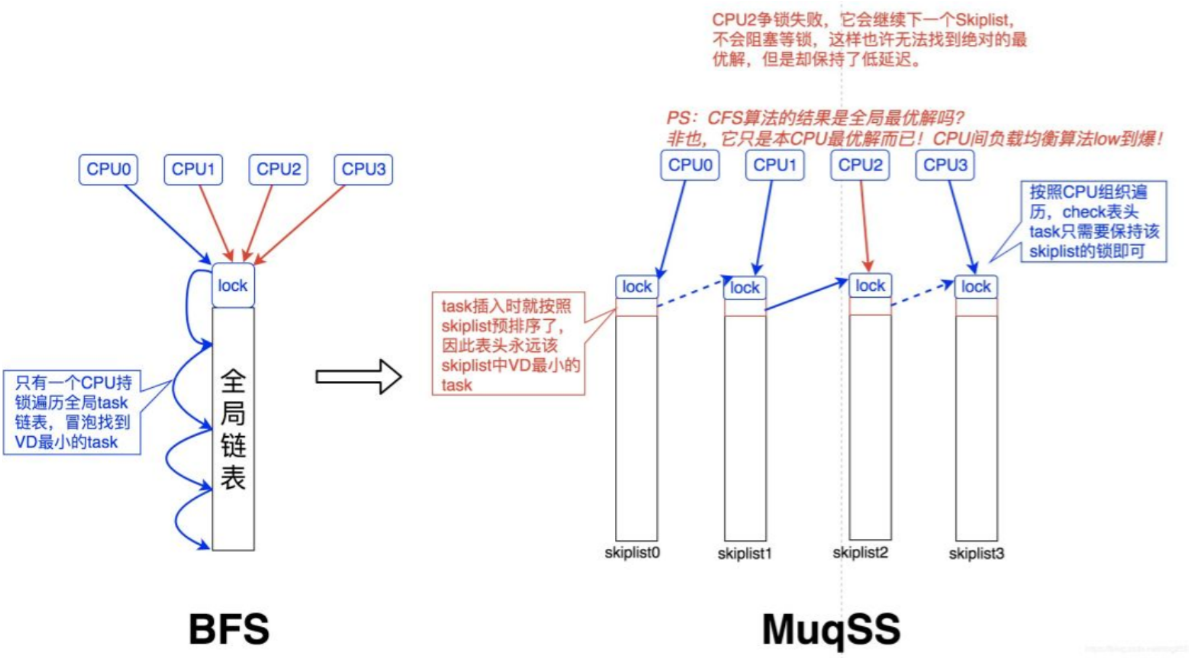
MuqSS零代价解决了BFS存在的两个问题：

1. 遍历查找的O(n)问题。 引入Skiplist数据结构替换双向链表，在*O*(log*n*)的插入代价下将查找的时间复杂度降为O(1)。【关于Skiplist，可以参考我的另一篇文章： https://blog.csdn.net/dog250/article/details/46997155】
2. 多CPU操作全局链表的锁问题。引入每CPU链表，避免全局争锁。同时以trylock代替lock，以损失准确性为代价实现无锁操作。

Con Kolivas在 *保持简单* 这个约束下设计了MuqSS，其要点是：

- Skiplist的作用类似主线Linux内核CFS中的红黑树，但比红黑树简单得多。
- 选择task的算法遍历所有CPU的Skiplist表头，选择当前全局最优task。
- 锁粒度细化到每个CPU的Skiplist。
- 遍历过程针对每CPU锁采用trylock，失败则继续下一个CPU，实现无锁化。

时间复杂度同样都是O(n)，但MuqSS的n指的是CPU数量而非task数量。



7.3 MuQSS关键因子

proc参数	默认值	含义	
iso_cpu	70	该值设置了无特权的SCHED_ISO进程可以以实施优先级运行的cpu百分占比，即在整个系统(即所有cpu)上滚动5秒的平均cpu百分比。 SCHED_ISO在linux-5.9.10中保留了，并未实现。	MuQSS独有
kexec_load_disabled	0	ROM/Flash boot loader	
rr_interval	6	MuQSS独有；该值是任何cpu调度单元可以运行的最小时间长度。增加该值可以提高计算密集型任务的吞吐量，但会增加延迟；同样，减少该值，牺牲吞吐量，降低了平均和最大延迟。 该值是ms级别，可设置范围为1-1000，一般默认值是根据调度器初始化时可用的cpu数量来决定，一般最小为6；	MuQSS独有
sched_energy_aware	1	softlockup threshold，是看门狗threshold的两倍大。如果将该值设置为0，则会关闭lockup探测。	
yield_type	1	该值决定了sched_yield函数调用时会怎么表现 0：不放弃cpu 1：只放弃cpu给更高优先级的进程 2：耗尽时间片并重新计算deadline	MuQSS独有

7.4 MuQSS的源码实现

7.4.1 skip_list分析

MuQSS的skip_list主要增加了两个文件`include/linux/skip_list.h 以及 kernel/skip_list.c`

skip_list.h中：

```
typedef u64 keyType;
typedef void *valueType;
typedef struct nodeStructure skiplist_node;

struct nodeStructure {
    int level;  /* Levels in this structure */
    keyType key;
    valueType value;
    skiplist_node *next[8];   /*这里一共是8个，和后面的MaxNumberOfLevels对应么?*/
    skiplist_node *prev[8];
};  //定义跳表节点

typedef struct listStructure {
    int entries;   /*记录元素个数，每次插入加1，每次删除减1*/
    int level;  /* Maximum level of the list
            (1 more than the number of levels in the list) */
    skiplist_node *header; /* pointer to header */
```

```
} skiplist;
```

skiplist_node节点结构体中有个`level`， skiplist表结构体中也有个`level`，这两个level有何区别：

skip_list.c中：

初始化一个slnode节点：

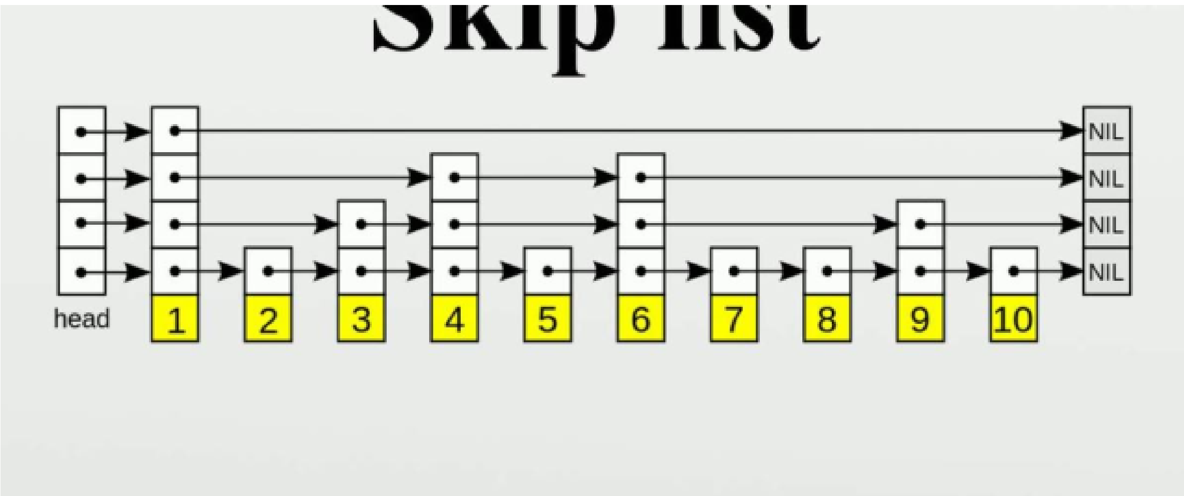
```c
void skiplist_node_init(skiplist_node *node)
{
    memset(node, 0, sizeof(skiplist_node)); /*内存置零，就是准备留给新对象使用*/
}
```

初始化一个跳表：

- 跳表最大级数是8；
- 初始化next和prev指向节点本身；

```c
//MaxNumberOfLevels是跳表的级数，这里定义最多8级
#define MaxNumberOfLevels 8
#define MaxLevel (MaxNumberOfLevels - 1)

void skiplist_init(skiplist_node *slnode)
{
    int i;

    slnode->key = 0xFFFFFFFFFFFFFFFF;
    slnode->level = 0;
    slnode->value = NULL;
    for (i = 0; i < MaxNumberOfLevels; i++)
        slnode->next[i] = slnode->prev[i] = slnode; /*初始slnode的next和prev指自己*/
}
```

个人理解：这里的MaxNumberOfLevels级数不是指下图中的黄色底标的数字，是指从左到右的箭头的层数。



创建一个新的空表：

```
skiplist *new_skiplist(skiplist_node *slnode)
{
    skiplist *l = kzalloc(sizeof(skiplist), GFP_ATOMIC);

    BUG_ON(!l);
    l->header = slnode;
    return l;
}
```

销毁一张表：

```
void free_skiplist(skiplist *l)
{
    skiplist_node *p, *q;

    p = l->header;
    do {
        q = p->next[0];
        p->next[0]->prev[0] = q->prev[0];
        skiplist_node_init(p);
        p = q;
    } while (p != l->header);
    kfree(l);
}
```

插入一个节点：

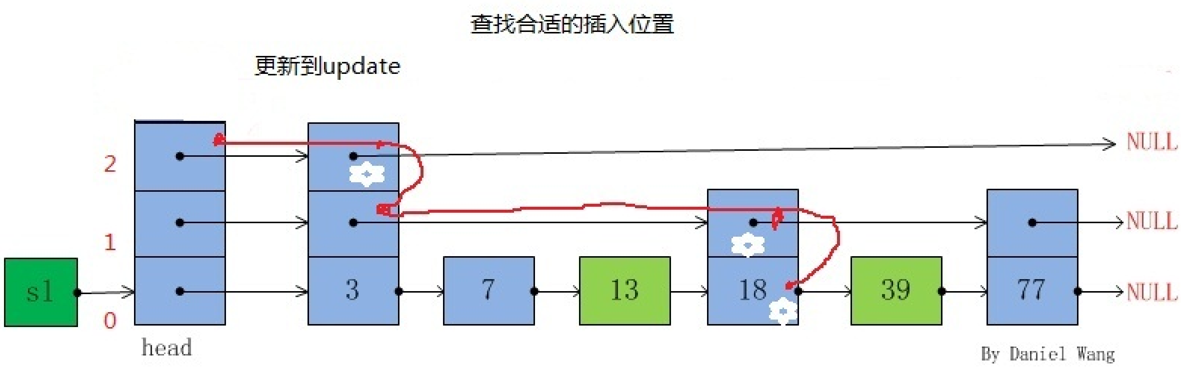
```
void skiplist_insert(skiplist *l, skiplist_node *node, keyType key, valueType value, unsigned int randseed)
{
    skiplist_node *update[MaxNumberOfLevels];
    skiplist_node *p, *q;
    int k = l->level;

    /*步骤1，从最高层一层一层往下找，并更新update数组，update数组中保存的是每次降一层时的节点*/
    p = l->header;
    do {
        while (q = p->next[k], q->key <= key)
            p = q;
        update[k] = p;
    } while (--k >= 0);

    ++l->entries;
    /*步骤2，产生一个随机层数level，如果新生成的层数比跳表的层数大，则设置k为跳表当前level大1的层数，并更新update中k层指向header*/
    k = randomLevel(randseed); /*需要插入的层*/
```

```
    if (k > l->level) {
        k = ++l->level;
        update[k] = l->header;
    }

    /*步骤3，将待插入的节点一层一层的插入*/
    node->level = k; /*node当中的level记录了该节点从哪一层被插入*/
    node->key = key;
    node->value = value;
    do {
        p = update[k];     /*这里逐层往下插入到update[k]之后一个元素*/
        node->next[k] = p->next[k];
        p->next[k] = node;
        node->prev[k] = p;
        node->next[k]->prev[k] = node;
    } while (--k >= 0);
}

/*步骤2中为何不用担心k超过MaxNumberOfLevels?应该是这个计算随机数时会保证在MaxLevel范围内*/
static inline unsigned int randomLevel(const long unsigned int randseed)
{
    return find_first_bit(&randseed, MaxLevel) / 2;
}
```

下图中假设要插入的值是25：

步骤1

我们需要对于每一层进行遍历并保存这一层中下降的节点(其后继节点为NULL或者后继节点的key大于等于要插入的key)，如下图，节点中有白色星花标识的节点保存到update数组。



步骤2

通过一个随机算法产生一个随机的层数，但是当这个随机产生的层数level大于当前跳表的最大层数时，我们此时需要更新当前跳表最大层数到level之间的update内容，这时应该更新其内容为跳表的头节点head，想想为什么这么做？ 然后就是更新跳表的最大层数。

这么做update[k]=l->header是为啥，是因为多加了一层时，只有header么？

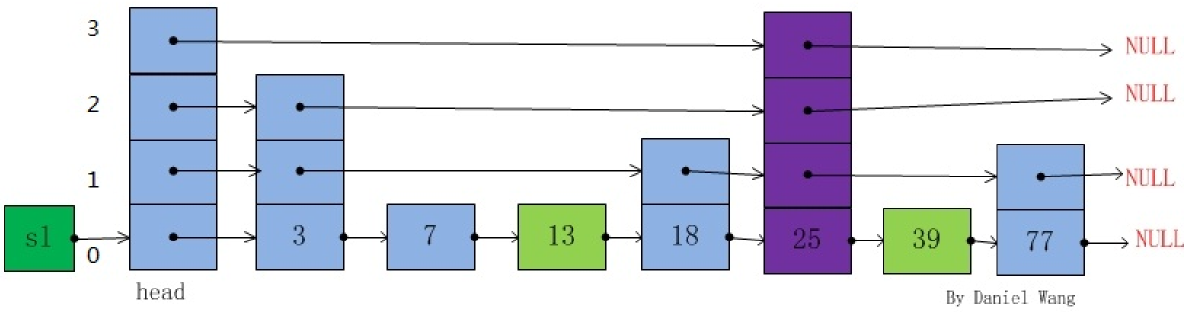
对，我觉得答案就是这样，当计算随机数k已经大于当前list的最大level了，则向上取一层，这一层是新的，里面没有元素，所以update[k]需要从header处开始插入

还有为什么会算出来k大于了当前跳表的最大层数？那样每次都k == ++l->level 会不会k超过了最大跳表值？

函数randomLevel中应该限制了MaxLevel（是MaxNumberOfLevel-1）

步骤3

根据update[k]开始逐层往下插入节点



删除跳表节点：

```
void skiplist_delete(skiplist *l, skiplist_node *node)
{
    int k, m = node->level; /*这里的node-level在插入时设置了*/

    /*for循环从0层开始一直到node所在插入层遍历，将node节点的next和prev移除*/
    for (k = 0; k <= m; k++) {
        node->prev[k]->next[k] = node->next[k];
        node->next[k]->prev[k] = node->prev[k];
    }
    skiplist_node_init(node);  /*只是把node节点内存置空，并未释放这部分内存啊？*/
    /*如果m刚好是最顶层，删除节点后需要检查下是否m层只剩下header，如果是删除m层*/
    if (m == l->level) {
        while (l->header->next[m] == l->header && l->header->prev[m] == l->header && m > 0)
            m--;
        l->level = m;
    }
    l->entries--; /*这个entries计数有何用？待解答*/
}
```

7.4.2 MuQSS详细设计

为何设计MuQSS?

BFS中是所有的CPU共享一个runqueue，这会导致什么呢？会导致每个CPU都需要去搜索整个runqueue去寻找拥有最早的deadline的进程来调度，并且不用管该进程原来是哪个CPU调度的，从而导致BFS的延时会因processed和CPUs的数量增加而增加。**并且，单个runqueue**会导致CPU之间的**锁竞争**，当CPU数量超过16个后，**lock contention**就很严重了。

MuQSS是BFS的一种进化方案，改进在哪里?

- 每个CPU都有自己的runqueue
- skiplist跳表取代链表

那么，当初BFS为何只用**一个runqueue**呢？

是因为有multiple runqueues会需要复杂的交互，因为每个runqueue都只会负责它自己队列的**调度延时和公平性**，这边需要一个复杂的交互系统来保证低延时和公平性，任何增加CPU本地进程调度吞吐量带来优势的同时也会带来劣势，这是因为需要一个复杂的平衡系统，来保证绑定同一个进程到同一个CPU的低延时效果，而不是同一个进程被不同的CPU调度运行。

MuQSS怎么解决多个runqueue带来的劣势问题？

MuQSS通过跳表优先级排序、创新的使用了无锁检查（当它需要因为降低延时需求或者CPU平衡等理由来从其他队列中获取更早的deadline的任务时）。MuQSS仍然没有balancing系统，选择允许下一个任务调度决策和任务唤醒CPU来实现平衡。

详细设计

1. 定制的skiplist实现

MuQSS使用固定的**8 level**跳表，不是动态分配的，这样使得每个队列仅可将O(logN)扩展为64k个任务（**这个地方没搞懂**），但是呢，每个CPU都有一个runqueue的话，这样O(logN)最多可扩展64k * CPUs个任务，和CPU数量相关了

2. 任务插入

MuQSS任务插队就是一个O(logN)的插入skiplist操作。

3. Niffies

jiffies是记录系统启动后到现在的时钟中断次数，它取决于系统的时钟评率，比如1000Hz，那么产生时钟中断是没1/1000s一次，也即1ms一次。

niffies和jiffies不同，niffies是一直单调递增的定时器，纳秒单位，Niffies是根据高分辨率TSC计时器针对每个运行队列计算的，并且为了保持公平性，每当两个运行队列同时锁定时，CPU之间就会进行同步

4. virtual deadline

虚拟期限？，MuQSS中保证**低延时、调度公平性、优先级**的关键核心机制是**virtual deadline machanism**。

rr_interval: roud robin interval，该参数可通过proc系统调节，作用是：当两个任务具有相同的nice级别时（普通进程SCHED_NORMAL或者SCHED_OTHER)，该进程能够运行的最大时间；或者换个角度说，两个相同优先级任务的最大延迟时间。

当一个任务需要CPU时间，它被配置的**时间片（time_slice)**等于一个rr_interval和一个virtual deadline，（这里如何理解？），virtual deadline如何计算：

`niffies + （prio_ratio * rr_interval)`

其中：

- prio_ratio：优先级，是和nice -20的基线进行比的比率，每增加一个nice level，prio_ratio增加10%；

- deadline：（deadline调度器是根据deadline来选择调度的，最先到达截止时间点的进程被有优先调度）；截止时间点，是个虚拟时间，用于比较接下来调度运行那个任务。

  选择哪个进程该运行，通常有三种情况：

  - 时间片耗尽，进程会被重新调度，时间片也会被分配，deadline也会按照上面的公式进行重计算；
  - 进程进入睡眠sleep状态，会让出CPU，这个过程中，time_slice时间片和deadline不会改变，该进程下次被调度时还会恢复；
  - 抢占，一个新的任务比当前正在运行的任务有更高的优先级，可以抢占。

  在前两种情况中，deadline是选择下一个运行任务的关键要素点。

The CPU proportion of different nice tasks works out to be approximately the
(prio_ratio difference)^2
The reason it is squared is that a task's deadline does not change while it is running unless it runs out of time_slice. Thus, even if the time actually passes the deadline of another task that is queued, it will not get CPU time unless the current running task deschedules, and the time "base" (niffies) is constantly moving.

5. 任务查找

由于在skiplist中，任务已经预先根据调度的预期顺序排序了，通常选择下一个待运行的任务就是选择**0 level的第一个entry入口任务**，

查找的时间复杂度是O(k)，这里的k是CPU个数。

6. 延时

通过使用虚拟期限来控制正常任务的调度顺序，可以确保每个运行队列的队列到激活延迟都受rr_interval可调参数约束，该参数默认设置为6ms。 这意味着与CPU绑定的任务等待的最长时间将与正在运行的任务的数量成正比，在通常情况下，每个CPU 0-2个正在运行的任务，将低于7ms的阈值（人类能感到抖动的阈值）。

7.4.3 源码简析

rq

MuQSS对rq结构体进行了定制修改：

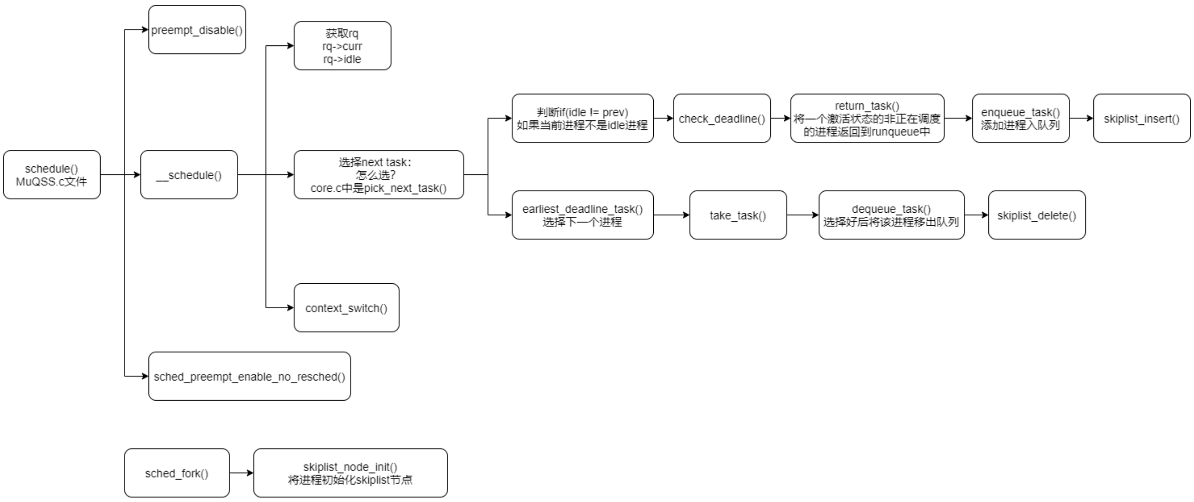
```c
/*
 * This is the main, per-CPU runqueue data structure.
 * This data should only be modified by the local cpu.
 */
struct rq {
	raw_spinlock_t *lock;
	raw_spinlock_t *orig_lock;

	struct task_struct __rcu	*curr;
	struct task_struct	*idle;
	struct task_struct	*stop;
	struct mm_struct *prev_mm;

```

```
    unsigned int nr_running;
    /*
     * This is part of a global counter where only the total sum
     * over all CPUs matters. A task can increase this counter on
     * one CPU and if it got migrated afterwards it may decrease
     * it on another CPU. Always updated under the runqueue lock:
     */
    unsigned long nr_uninterruptible;
#ifdef CONFIG_SMP
    unsigned int        ttwu_pending;
#endif
    u64 nr_switches;

    /* Stored data about rq->curr to work outside rq lock */
    u64 rq_deadline;
    int rq_prio;

    /* Best queued id for use outside lock */
    u64 best_key;

    unsigned long last_scheduler_tick; /* Last jiffy this RQ ticked */
    unsigned long last_jiffy; /* Last jiffy this RQ updated rq clock */
    u64 niffies; /* Last time this RQ updated rq clock */
    u64 last_niffy; /* Last niffies as updated by local clock */
    u64 last_jiffy_niffies; /* Niffies @ last_jiffy */

    u64 load_update; /* When we last updated load */
    unsigned long load_avg; /* Rolling load average */
#ifdef CONFIG_HAVE_SCHED_AVG_IRQ
    u64 irq_load_update; /* When we last updated IRQ load */
    unsigned long irq_load_avg; /* Rolling IRQ load average */
#endif
#ifdef CONFIG_SMT_NICE
    struct mm_struct *rq_mm;
    int rq_smt_bias; /* Policy/nice level bias across smt siblings */
#endif
    /* Accurate timekeeping data */
    unsigned long user_ns, nice_ns, irq_ns, softirq_ns, system_ns,
        iowait_ns, idle_ns;
    atomic_t nr_iowait;

#ifdef CONFIG_MEMBARRIER
    int membarrier_state;
#endif

    skiplist_node *node;
    skiplist *sl;
#ifdef CONFIG_SMP
    struct task_struct *preempt; /* Preempt triggered on this task */
    struct task_struct *preempting; /* Hint only, what task is preempting */

    int cpu;        /* cpu of this runqueue */
    bool online;

    struct root_domain *rd;
    struct sched_domain *sd;

```

```
	unsigned long cpu_capacity_orig;

	int *cpu_locality; /* CPU relative cache distance */
	struct rq **rq_order; /* Shared RQs ordered by relative cache distance */
	struct rq **cpu_order; /* RQs of discrete CPUs ordered by distance */

	bool is_leader;
	struct rq *smp_leader; /* First physical CPU per node */
#ifdef CONFIG_SCHED_THERMAL_PRESSURE
	struct sched_avg	avg_thermal;
#endif /* CONFIG_SCHED_THERMAL_PRESSURE */
#ifdef CONFIG_SCHED_SMT
	struct rq *smt_leader; /* First logical CPU in SMT siblings */
	cpumask_t thread_mask;
	bool (*siblings_idle)(struct rq *rq);
	/* See if all smt siblings are idle */
#endif /* CONFIG_SCHED_SMT */
#ifdef CONFIG_SCHED_MC
	struct rq *mc_leader; /* First logical CPU in MC siblings */
	cpumask_t core_mask;
	bool (*cache_idle)(struct rq *rq);
	/* See if all cache siblings are idle */
#endif /* CONFIG_SCHED_MC */
#endif /* CONFIG_SMP */

#ifdef CONFIG_IRQ_TIME_ACCOUNTING
	u64 prev_irq_time;
#endif /* CONFIG_IRQ_TIME_ACCOUNTING */
#ifdef CONFIG_PARAVIRT
	u64 prev_steal_time;
#endif /* CONFIG_PARAVIRT */
#ifdef CONFIG_PARAVIRT_TIME_ACCOUNTING
	u64 prev_steal_time_rq;
#endif /* CONFIG_PARAVIRT_TIME_ACCOUNTING */

	u64 clock, old_clock, last_tick;
	/* Ensure that all clocks are in the same cache line */
	u64 clock_task ____cacheline_aligned;
	int dither;

	int iso_ticks;
	bool iso_refractory;

#ifdef CONFIG_HIGH_RES_TIMERS
	struct hrtimer hrexpiry_timer;
#endif

	int rt_nr_running; /* Number real time tasks running */
#ifdef CONFIG_SCHEDSTATS

	/* latency stats */
	struct sched_info rq_sched_info;
	unsigned long long rq_cpu_time;
	/* could above be rq->cfs_rq.exec_clock + rq->rt_rq.rt_runtime ? */

	/* sys_sched_yield() stats */
	unsigned int yld_count;
```

```

    /* schedule() stats */
    unsigned int sched_switch;
    unsigned int sched_count;
    unsigned int sched_goidle;

    /* try_to_wake_up() stats */
    unsigned int ttwu_count;
    unsigned int ttwu_local;
#endif /* CONFIG_SCHEDSTATS */

#ifdef CONFIG_CPU_IDLE
    /* Must be inspected within a rcu lock section */
    struct cpuidle_state *idle_state;
#endif
}
```

MuQSS.c直接将core.c原本的框架修改了，直接不涉及调度类等，集成在MuQSS.c中



八、实时调度器

九、自定义调度器

9.1 框架

框架需不需要修改？

MuQSS中的MuQSS.c对应主线的core.c，其中实现了大部分原主线上的函数比如context_switch()等等；

TODO：

实现一个简单的先进先出的调度器，框架没必要修改，还是用原来的core.c中实现的即可；

9.2 调度类

现有的主线core.c中涉及调度类stop > deadline > rt > fair > idle，共5个调度类；

MuQSS暂未找到调度类的用法，其很多功能都是在MuQSS.c封装实现，比如：

enqueue_task()，dequeue_task(), yield_to() 等功能函数，主线core.c中是封装后通过class->enqueue_task()钩子函数来调用具体的调度器算法的功能实现，MuQSS中是enqueue_task()直接实现；

TODO：

设计一个sched_class，集成到内核中；

9.3 具体实现

9.3.1 数据结构

TODO：

使用双链表或者跳表

runqueue需要重新设计一个rq，并嵌入到struct rq{}中去

runqueue

sched_class

policy

9.3.2 调度算法

TODO：

先进先出的算法，不涉及抢占，时间片用完或者自动退出才进行进程调度切换；

9.3.3 函数功能

TODO：

根据sched_class中的钩子函数，参照fair.c和MuQSS，实现每个钩子函数即可；

9.4 功能测试验证

9.4.1 编译

TODO:

涉及内核调度器修改，编译恐怕容易出错，跑起来容易导致panic

9.4.2 验证

TODO：

- 将其他的sched_class全部阉割掉，只留自开发的FIFO调度器；
- FIFO调度器和主线上调度器并存，用户设置进程使用哪种调度器比如FIFO来验证性能；

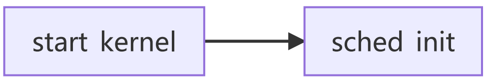
十、调度器框架

模仿fair_sched_class来修改编码fifolist_sched_class：

core.c中涉及的fair_sched_class有以下：

sched_init()	初始化调度器函数，由start_kernel调用
__setscheduler()	sched_setscheduler()函数会调用，用于修改线程的sched policy或者priority
rt_mutex_setprio()	设置当前任务的优先级
set_load_weight()	
set_task_cpu()	
sched_fork()	fork的进程调度
pick_next_task	选择下一个调度的进程任务

10.1 调度器初始化



core.c中的sched_init()函数到底初始化了什么？

```
{
    unsigned long ptr = 0;
    int i;

    /* Make sure the linker didn't screw up */
    BUG_ON(&idle_sched_class + 1 != &fair_sched_class ||
           &fair_sched_class + 1 != &rt_sched_class ||
           &rt_sched_class + 1   != &dl_sched_class);
#ifdef CONFIG_SMP
    BUG_ON(&dl_sched_class + 1 != &stop_sched_class);
#endif

    wait_bit_init();

```

```
#ifdef CONFIG_FAIR_GROUP_SCHED
    ptr += 2 * nr_cpu_ids * sizeof(void **);
#endif
#ifdef CONFIG_RT_GROUP_SCHED
    ptr += 2 * nr_cpu_ids * sizeof(void **);
#endif
    if (ptr) {
        ptr = (unsigned long)kzalloc(ptr, GFP_NOWAIT);

#ifdef CONFIG_FAIR_GROUP_SCHED
        root_task_group.se = (struct sched_entity **)ptr;
        ptr += nr_cpu_ids * sizeof(void **);

        root_task_group.cfs_rq = (struct cfs_rq **)ptr;
        ptr += nr_cpu_ids * sizeof(void **);

        root_task_group.shares = ROOT_TASK_GROUP_LOAD;
        init_cfs_bandwidth(&root_task_group.cfs_bandwidth);
#endif /* CONFIG_FAIR_GROUP_SCHED */
#ifdef CONFIG_RT_GROUP_SCHED
        root_task_group.rt_se = (struct sched_rt_entity **)ptr;
        ptr += nr_cpu_ids * sizeof(void **);

        root_task_group.rt_rq = (struct rt_rq **)ptr;
        ptr += nr_cpu_ids * sizeof(void **);

#endif /* CONFIG_RT_GROUP_SCHED */
    }
#ifdef CONFIG_CPUMASK_OFFSTACK
    for_each_possible_cpu(i) {
        per_cpu(load_balance_mask, i) = (cpumask_var_t)kzalloc_node(
            cpumask_size(), GFP_KERNEL, cpu_to_node(i));
        per_cpu(select_idle_mask, i) = (cpumask_var_t)kzalloc_node(
            cpumask_size(), GFP_KERNEL, cpu_to_node(i));
    }
#endif /* CONFIG_CPUMASK_OFFSTACK */

    init_rt_bandwidth(&def_rt_bandwidth, global_rt_period(), global_rt_runtime());
    init_dl_bandwidth(&def_dl_bandwidth, global_rt_period(), global_rt_runtime());

#ifdef CONFIG_SMP
    init_defrootdomain();
#endif

#ifdef CONFIG_RT_GROUP_SCHED
    init_rt_bandwidth(&root_task_group.rt_bandwidth,
            global_rt_period(), global_rt_runtime());
#endif /* CONFIG_RT_GROUP_SCHED */

#ifdef CONFIG_CGROUP_SCHED
    task_group_cache = KMEM_CACHE(task_group, 0);

    list_add(&root_task_group.list, &task_groups);
    INIT_LIST_HEAD(&root_task_group.children);
    INIT_LIST_HEAD(&root_task_group.siblings);
    autogroup_init(&init_task);
```

```
#endif /* CONFIG_CGROUP_SCHED */

    for_each_possible_cpu(i) {
        struct rq *rq;

        rq = cpu_rq(i);
        raw_spin_lock_init(&rq->lock);
        rq->nr_running = 0;
        rq->calc_load_active = 0;
        rq->calc_load_update = jiffies + LOAD_FREQ;
        init_cfs_rq(&rq->cfs);
        init_rt_rq(&rq->rt);
        init_dl_rq(&rq->dl);
#ifdef CONFIG_FAIR_GROUP_SCHED
        INIT_LIST_HEAD(&rq->leaf_cfs_rq_list);
        rq->tmp_alone_branch = &rq->leaf_cfs_rq_list;
        /*
         * How much CPU bandwidth does root_task_group get?
         *
         * In case of task-groups formed thr' the cgroup filesystem, it
         * gets 100% of the CPU resources in the system. This overall
         * system CPU resource is divided among the tasks of
         * root_task_group and its child task-groups in a fair manner,
         * based on each entity's (task or task-group's) weight
         * (se->load.weight).
         *
         * In other words, if root_task_group has 10 tasks of weight
         * 1024) and two child groups A0 and A1 (of weight 1024 each),
         * then A0's share of the CPU resource is:
         *
         *  A0's bandwidth = 1024 / (10*1024 + 1024 + 1024) = 8.33%
         *
         * We achieve this by letting root_task_group's tasks sit
         * directly in rq->cfs (i.e root_task_group->se[] = NULL).
         */
        init_tg_cfs_entry(&root_task_group, &rq->cfs, NULL, i, NULL);
#endif /* CONFIG_FAIR_GROUP_SCHED */

        rq->rt.rt_runtime = def_rt_bandwidth.rt_runtime;
#ifdef CONFIG_RT_GROUP_SCHED
        init_tg_rt_entry(&root_task_group, &rq->rt, NULL, i, NULL);
#endif
#ifdef CONFIG_SMP
        rq->sd = NULL;
        rq->rd = NULL;
        rq->cpu_capacity = rq->cpu_capacity_orig = SCHED_CAPACITY_SCALE;
        rq->balance_callback = NULL;
        rq->active_balance = 0;
        rq->next_balance = jiffies;
        rq->push_cpu = 0;
        rq->cpu = i;
        rq->online = 0;
        rq->idle_stamp = 0;
        rq->avg_idle = 2*sysctl_sched_migration_cost;
        rq->max_idle_balance_cost = sysctl_sched_migration_cost;

        INIT_LIST_HEAD(&rq->cfs_tasks);

```

```
        rq_attach_root(rq, &def_root_domain);
#ifdef CONFIG_NO_HZ_COMMON
        rq->last_blocked_load_update_tick = jiffies;
        atomic_set(&rq->nohz_flags, 0);

        rq_csd_init(rq, &rq->nohz_csd, nohz_csd_func);
#endif
#endif /* CONFIG_SMP */
        hrtick_rq_init(rq);
        atomic_set(&rq->nr_iowait, 0);
    }

    set_load_weight(&init_task, false);

    /*
     * The boot idle thread does lazy MMU switching as well:
     */
    mmgrab(&init_mm);
    enter_lazy_tlb(&init_mm, current);

    /*
     * Make us the idle thread. Technically, schedule() should not be
     * called from this thread, however somewhere below it might be,
     * but because we are the idle thread, we just pick up running again
     * when this runqueue becomes "idle".
     */
    init_idle(current, smp_processor_id());

    calc_load_update = jiffies + LOAD_FREQ;

#ifdef CONFIG_SMP
    idle_thread_set_boot_cpu();
#endif
    init_sched_fair_class();

    init_schedstats();

    psi_init();

    init_uclamp();

    scheduler_running = 1;
}
```