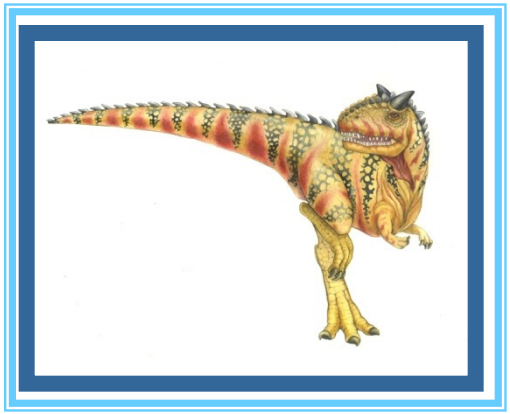
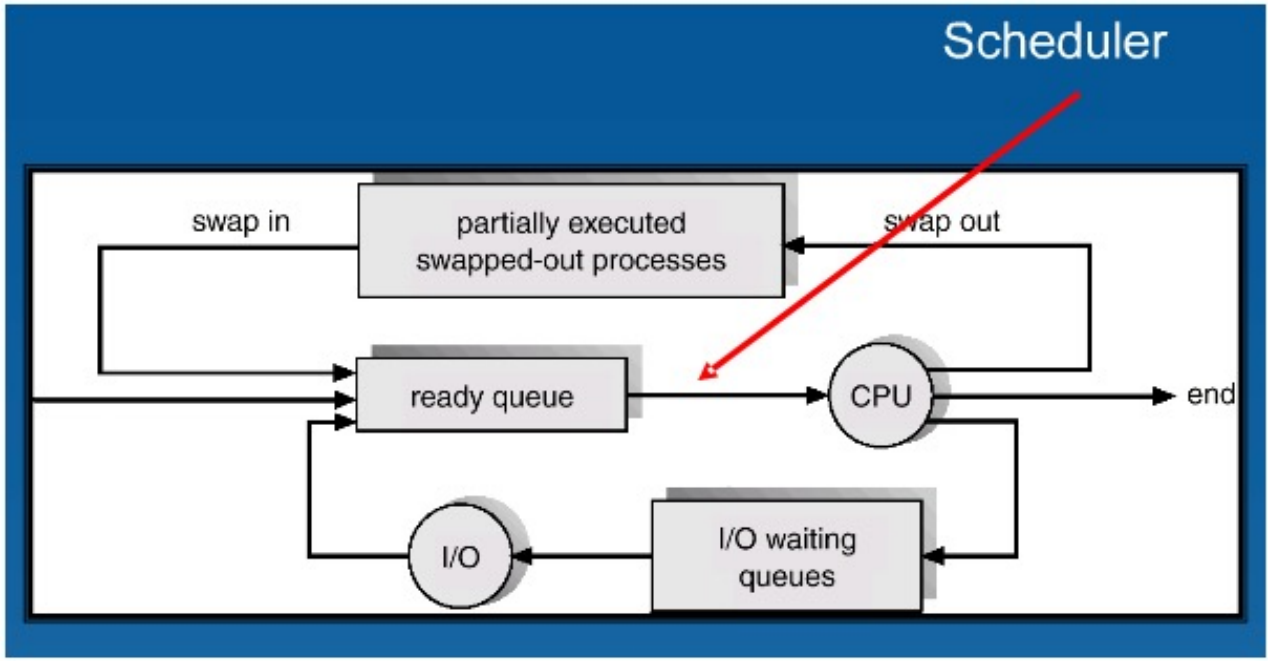
进程调度算法



调度器的任务

- 通过CPU 调度器，计算机进程（PCB）有四个去向：
  1）留在就绪队列
  2）进入CPU
  3）等待队列
  4）I/O操作



最简单的版本：Linux0.11

```
void schedule(void)
{
    int i,next,c;
    struct task_struct ** p;
    //NR_TASKS 是最大的进程数目

/* this is the scheduler proper: */

    while (1) {
        c = -1;
        next = 0;
        i = NR_TASKS;
        p = &task[NR_TASKS];
        while (--i) {
            if (!*--p)
                continue;
            if ((*p)->state == TASK_RUNNING && (*p)->counter > c)
                c = (*p)->counter, next = i;
        }

        if (c) break;
        for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
            if (*p)
                (*p)->counter = ((*p)->counter >> 1) +
                        (*p)->priority;
    }
    switch_to(next);
}
```

最简单的版本：Linux0.11

```
void do_timer(...){
        if(- -current->counter>0) return 0;
        current->counter=0;
        schdule(); //如果时间片到期
    }

   _timer_interrput:
        call _do_timer 当发生时间片中断的时候
```

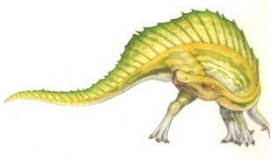
Linux支持的调度策略

- task_struct内包含一个字段标识task的调度策略：

  unsigned int policy;

- 其可选值如下：

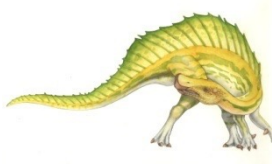
```
// kernel/sched/sched.h

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



实时任务 vs 普通任务

- 普通任务采用的nice值，我们可以通过nice库函数为task设置nice值，nice值越小，调度执行的机会就越大，其取值范围是 -20 ~ 19
- 实时任务采用的实时优先级rt_priority，其取值范围是0 ~ 99
- 在task_struct中有下面几个字段标识着task的优先级的值：

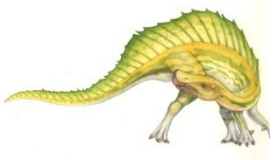
```
// 动态优先级，prio 的值是调度器最终使用的优先级数值，prio越小，表示优先级越高，其取值范围是0-139
// 它又被分为两个区间，0~99表示实时任务优先级，100~139表示普通任务优先级（对于nice值-20~19）
int prio;
// 静态优先级不会随时间改变，内核不会主动修改它，只能通过系统调用 nice 去修改 static_prio
// 有效范围是 100 ~ 139，默认值为120，0~99 没有意义 \
int static_prio;
// 归一化优先级
int normal_prio;
// 实时优先级，取值范围是0~99，仅对实时任务有效
unsigned int rt_priority;
```



优先级定义

- 在 <include/linux/sched/prio.h> 头文件中看到内核表示进程优先级的单位（scale）和宏定义（macros），它们用来将用户空间优先级映射到到内核空间

```
#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)
/*
* Convert user-nice values [ -20 ... 0 ... 19 ]
* to static priority [ MAX_RT_PRIO..MAX_PRIO-1 ],
* and back.
*/
#define NICE_TO_PRIO(nice) ((nice) + DEFAULT_PRIO)
#define PRIO_TO_NICE(prio) ((prio) - DEFAULT_PRIO)
/*
* 'User priority' is the nice value converted to something we
* can work with better when scaling various scheduler parameters,
* it's a [ 0 ... 39 ] range.
*/
#define USER_PRIO(p) ((p)-MAX_RT_PRIO)
#define TASK_USER_PRIO(p) USER_PRIO((p)->static_prio)
#define MAX_USER_PRIO (USER_PRIO(MAX_PRIO))
```

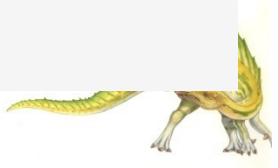


实时任务 vs 普通任务

- **实时任务调度策略：**
  - SCHED_FIFO，按照任务的先后顺序执行，高优先级的任务可以抢占低优先级的任务；
  - SCHED_RR，为每一个任务分配一定大小的时间片，时间片用完后会将任务准移到队列的尾部，高优先级的任务可以抢占低优先级的任务；
  - SCHED_DEADLINE，优先选择当前时间距离任务截止时间近的任务。
- **普通任务调度策略：**
  - SCHED_NORMAL，普通任务调度策略；
  - SCHED_BATCH，后台任务；
  - SCHED_IDLE，空闲时才执行。

PCB中和调度相关的代码

```
1  struct task_struct {
2      ........
3      /* 表示是否在运行队列 */
4      int on_rq;
5
6      /* 进程优先级
7       * prio: 动态优先级，范围为100~139，与静态优先级和补偿(bonus)有关
8       * static_prio: 静态优先级，static_prio = 100 + nice + 20 (nice值为-20~19,所以static_prio值为100~139)
9       * normal_prio: 没有受优先级继承影响的常规优先级，具体见normal_prio函数，跟属于什么类型的进程有关
10      */
11     int prio, static_prio, normal_prio;
12     /* 实时进程优先级 */
13     unsigned int rt_priority;
14     /* 调度类，调度处理函数类 */
15     const struct sched_class *sched_class;
16     /* 调度实体(红黑树的一个结点) */
17     struct sched_entity se;
18     /* 调度实体(实时调度使用) */
19     struct sched_rt_entity rt;
20 #ifdef CONFIG_CGROUP_SCHED
21     /* 指向其所在进程组 */
22     struct task_group *sched_task_group;
23 #endif
24     ........
25 }
```

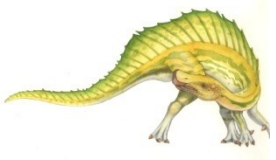


Linux调度类

- 调度类是负责真正执行调度策略的逻辑，在task_struct内部有一个调度类的结构体指针：

  const struct sched_class *sched_class;

- sched_class只是一个抽象的结构体，其内部定义了一些调度先关的方法



Linux调度类

```
struct sched_class {
        // 调度类是一个链表，按照优先顺序排列，next执行下一个调度类
        const struct sched_class *next;
        // 添加任务
        void (*enqueue_task) (struct rq *rq, struct task_struct *p, int flag
        // 移除任务
        void (*dequeue_task) (struct rq *rq, struct task_struct *p, int flag
        // 校验是否当前任务应该被抢占
        void (*check_preempt_curr)(struct rq *rq, struct task_struct *p, int
        // 或取下一个待执行的任务
        struct task_struct * (*pick_next_task)(struct rq *rq,
                                               struct task_struct *prev,
                                               struct rq_flags *rf);
        void (*put_prev_task)(struct rq *rq, struct task_struct *p);

        void (*set_curr_task)(struct rq *rq);
        // 时钟中断处理
        void (*task_tick)(struct rq *rq, struct task_struct *p, int queued);

        /*
         * The switched_from() call is allowed to drop rq->lock, therefore w
         * 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);
  ......
};
```

Schedule Class的实现

- **sched_class本身定义了一个抽象接口，具体是现实可以是多样化的，目前的实现包括：**

```
// 会停止所有其他线程，不会被其他任务打断，优先级最高的任务会使用
extern const struct sched_class stop_sched_class;
// 对于上面deadline调度策略的执行
extern const struct sched_class dl_sched_class;
// 对应上面FIFO与RR调度策略的执行，具体哪一个策略，由policy字段指定
extern const struct sched_class rt_sched_class;
// 对应NORMAL普通调度策略的执行，我们称为公平调度类，其内部采用的是cfs调度算法，后面会详细说明
extern const struct sched_class fair_sched_class;
// 对应IDLE调度策略的执行
extern const struct sched_class idle_sched_class;
```

Schedule Class的实现

- 每一个调度类针对上面的接口函数都有各自的实现机制，同时调度类其实是链表的数据结构，按照优先顺序依次为：

stop_sched_class → dl_sched_class → rt_sched_class → fair_sched_class → idle_sched_class

```
const struct sched_class stop_sched_class = {
    .next                       = &dl_sched_class,
    ..
}

const struct sched_class dl_sched_class = {
    .next                       = &rt_sched_class,
    ...
}

const struct sched_class rt_sched_class = {
    .next                       = &fair_sched_class
    ...
}
...
```

Linux调度实体

- 调度实体用于维护task调度相关的元信息，其有以下几种：
  - 普通任务使用的调度实体 **struct sched_entity se;**
  - 实时任务使用的调度实体 **struct sched_rt_entity rt;**
  - **deadline调度类的调度实体 struct sched_dl_entity dl;**

```
struct sched_entity {
    // 当前任务的权重值，linux通过nice函数为任务设置优先级，每一个nice值都对应一个权重值
    struct load_weight          load;
    unsigned long                       runnable_weight;
    struct rb_node                      run_node;
    struct list_head            group_node;
    unsigned int                        on_rq;

    u64                         exec_start;
    u64                         sum_exec_runtime;
    // 当前任务的虚拟运行时间，cfs调度算法的概念，基于task的实际运行时间与优先级权重计算出
    u64                         vruntime;
    u64                         prev_sum_exec_runtime;
}

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

CFS调度算法

- CFS（Complete Fair Schedule）调度算法的思想是为每一个task维护一个拟的运行时间vruntime，
- 调度程序优先选择vruntime值最小的任务执行，之所以引入vruntime的概念，是为了支持优先级调度。
- vruntime其实是基于task的实际运行时间及优先级权重计算出来的值，其计算公式如下：

  vruntime += delta_exec * NICE_0_LOAD / weight

  - vruntime：虚拟运行时间
  - delta_exec: 实际的执行时间
  - NICE_0_LOAD：进程优先级nice值为0对应的权重值
- nice值越小其权重值越大，则最终计算出来的vruntime就会偏小
- 为了优先获取vruntime最小任务的时间复杂度，LinuxCFS算法的实现上采用红黑树的数据结构，其具体实现是运行队列中的cfs_rq。

调度权重：Weight

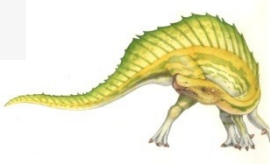
- 为了让 nice 值的变化反映到 CPU 时间变化片上更加合理，Linux 内核中定义了一个数组，用于映射 nice 值到权重
- 假设有两个优先级都是 0 的任务，每个都能获得 50% 的 CPU 时间（1024 / (1024 + 1024) = 0.5）
- 如果突然给其中的一个任务优先级提升了 1 （nice 值 -1）。一个任务应该会获得额外 10% 左右的 CPU 时间，而另一个则会减少 10% CPU 时间
- 来看看计算结果：1277 / (1024 + 1277) ≈ 0.55，1024 / (1024 + 1277) ≈ 0.45，二者差距刚好在 10% 左右，符合预期

```
static const int prio_to_weight[40] = {
    /* -20 */ 88761, 71755, 56483, 46273, 36291,
    /* -15 */ 29154, 23254, 18705, 14949, 11916,
    /* -10 */ 9548, 7620, 6100, 4904, 3906,
    /* -5 */ 3121, 2501, 1991, 1586, 1277,
    /* 0 */ 1024, 820, 655, 526, 423,
    /* 5 */ 335, 272, 215, 172, 137,
    /* 10 */ 110, 87, 70, 56, 45,
    /* 15 */ 36, 29, 23, 18, 15,
};
```

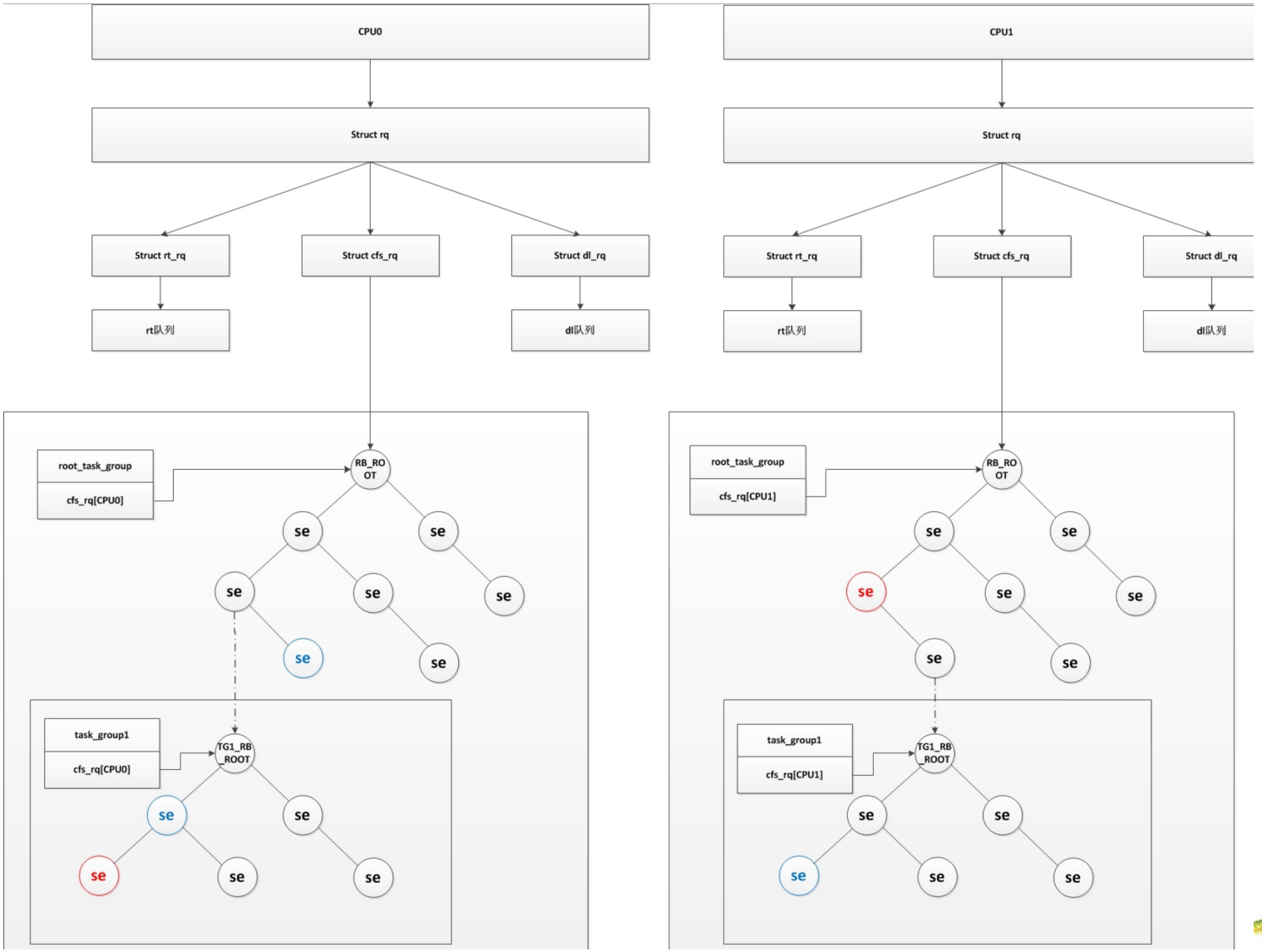
Linux运行队列

- 对于每一个CPU都会有一个rq的结构体，维护着所有待运行的任务，我们称之为运行队列（running queue）

```
struct rq {
    // cfs运行队列
    struct cfs_rq                   cfs;
    // 实时任务运行队列
    struct rt_rq                    rt;
    // deadline任务运行队列
    struct dl_rq                    dl;

    struct task_struct  *curr;

}
```



进程的调度：Run-Queue



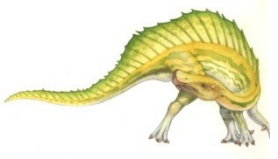
cfs_rq：CFS运行队列

```
// cfs任务队列，qi
struct cfs_rq {
    struct load_weight  load;
    unsigned long               runnable_weight;
    unsigned int                nr_running;
    unsigned int                h_nr_running;

    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).
     */
    // 指向cfs_rq上正在运行的运行实体
    struct sched_entity *curr;
    struct sched_entity *next;
    struct sched_entity *last;
    struct sched_entity *skip;

};

/*
struct rb_root_cached {
        struct rb_root rb_root;
        struct rb_node *rb_leftmost;
};
```

fair_sched_class调度类的实现

- fair_sched_class作为一个抽象类sched_class的具体实现，主要是实现了sched_class里面定义的若干函数

```
const struct sched_class fair_sched_class = {
    .next                   = &idle_sched_class,
    .check_preempt_curr = check_preempt_wakeup,
    .pick_next_task         = pick_next_task_fair,
    .set_curr_task    = set_curr_task_fair,
    .task_tick          = task_tick_fair,
};
```

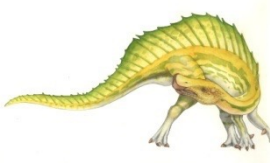


下一个待执行任务pick_next_task

```
static struct task_struct *
pick_next_task_fair(struct rq *rq, struct task_struct *prev, struct rq_flags
{
        struct cfs_rq *cfs_rq = &rq->cfs;
        struct sched_entity *se;
        struct task_struct *p;
        int new_tasks;

again:
        if (!cfs_rq->nr_running)
                goto idle;

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

        do {
                struct sched_entity *curr = cfs_rq->curr;
                if (curr) {
                        if (curr->on_rq)
                                // 更新当前任务，主要包括总时间、vruntime等
                                update_curr(cfs_rq);
                        else
                                curr = NULL;
```

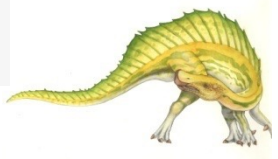
下一个待执行任务pick_next_task

```
                        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);

        p = task_of(se);
```



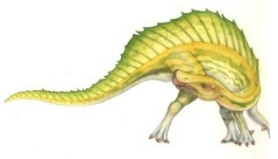
下一个待执行任务pick_next_task

```
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);
                }
        }
        // 修改完时间后，将任务重新放回cfs队列
        put_prev_entity(cfs_rq, pse);
        set_next_entity(cfs_rq, se);
}

goto done;
```



下一个待执行任务pick_next_task

```
simple:
#endif
        // 将任务放回rq
        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);
  ......
}
```



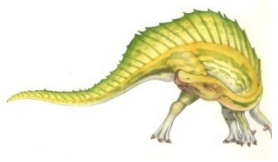
更新当前任务的运行时间统计update_curr函数

- 更新task执行总时间sum_exec_runtime
- 更新task的vruntime

```
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));
        u64 delta_exec;
        delta_exec = now - curr->exec_start;
        curr->sum_exec_runtime += delta_exec;
        // 里面执行的就是vruntime的计算公式
        curr->vruntime += calc_delta_fair(delta_exec, curr);
        update_min_vruntime(cfs_rq);
        。。。
  。。。
}
```

实时任务和普通任务的混合调度

- 调度程序在执行核心调度逻辑的首要工作是获取下一个应该被执行的任务，调度算法其实是一个按照优先顺序的链表，调度程序会依次遍历每一个调度类，分别调用每一个调度类的**pick_next_task()**函数
- 实时任务调度类会优先调用，首先从**rt_rq**中获取任务，如果没有获取到才会交给公平调度类**fair_sched_class**从**cfs_rq**中获取任务，通过此机制实现了实时任务优先于普通任务执行
- 因此，当系统中存在实时任务的时候，普通任务将得不到执行的机会。



实时任务和普通任务的混合调度

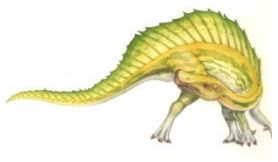
```
/*
 * Pick up the highest-prio task:
 */
static inline struct task_struct *
pick_next_task(struct rq *rq, struct task_struct *prev, struct rq_flags *rf)
{
        const struct sched_class *class;
        struct task_struct *p;

        /*
         * Optimization: we know that if all tasks are in the fair class we
         * call that function directly, but only if the @prev task wasn't of
         * higher scheduling class, because otherwise those loose the
         * opportunity to pull in more work from other CPUs.
         */
        // 一个优化机制，因为通常大部分任务都是普通任务
        // 如果当前rq中所有的任务都是普通任务，就可以直接从fair_sched_class调度类开始
        // 而不用像下面for_each_class(class){} 从头开始遍历每一个调度类
        if (likely((prev->sched_class == &idle_sched_class ||
                    prev->sched_class == &fair_sched_class) &&
                   rq->nr_running == rq->cfs.h_nr_running)) {

                p = fair_sched_class.pick_next_task(rq, prev, rf);
                if (unlikely(p == RETRY_TASK))
                        goto again;
```

实时任何和普通任务的混合调度

```
                /* Assumes fair_sched_class->next == idle_sched_class */
                if (unlikely(!p))
                        p = idle_sched_class.pick_next_task(rq, prev, rf);

                return p;
        }
        // 遍历每一个调度类，获取下一个待执行的任务，例如fair_sched_class的具体参考上-
        for_each_class(class) {
                p = class->pick_next_task(rq, prev, rf);
                if (p) {
                        if (unlikely(p == RETRY_TASK))
                                goto again;
                        return p;
                }
        }

}
```

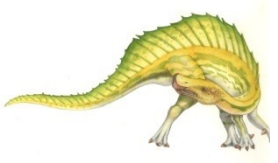
主动调度与抢占式调度

- 主动调度：任务执行schedule库函数（schedule系统调用），主动让出CPU资源，例如当任务在等待IO资源时，此时任务处于不能推进的状态，就可以调用schedule让出CPU资源给别的task
- 抢占式调度，主要有两种：进程执行的时间太长了，时钟中断处理函数触发抢占当前正在执行的任务；当一个任务被唤醒的时候，如果被唤醒任务的优先级高于当前运行任务的优先级，则会触发抢占



主动调度的执行过程 schedule()

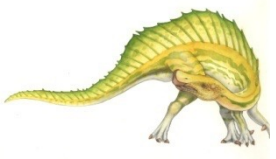
```
static void __sched notrace __schedule(bool preempt)
{
        struct task_struct *prev, *next;
        unsigned long *switch_count;
        struct rq_flags rf;
        struct rq *rq;
        int cpu;

        cpu = smp_processor_id();
        // 获取当前cpu上的运行队列rq
        rq = cpu_rq(cpu);
        // 去取当前运行队列上正在执行的任务，记做prev，因为一旦切换后，当前任务就变成pre
        prev = rq->curr;
        ......
        // 获取下一个待执行的任务,内部逻辑就是遍历每一个调度类，获取任务，
        // 具体实现源码请参考上面一小节：如何实现实时任务优先普通任务执行
        next = pick_next_task(rq, prev, &rf);
        ......
        // 如果下一个待执行任务与prev不等，则需要进程上下文切换，next任务载入运行
        if (likely(prev != next)) {
                rq->nr_switches++;
                rq->curr = next;
                ++*switch_count;

                trace_sched_switch(preempt, prev, next);

                rq = context_switch(rq, prev, next, &rf);
        } else {
                rq->clock_update_flags &= ~(RQCF_ACT_SKIP|RQCF_REQ_SKIP);
                rq_unlock_irq(rq, &rf);
        }
}
```



时钟中断触发抢占

- 时钟中断处理函数会调用scheduler_tick()函数：
- 取出当前运行任务的task_struct
- 调用调度类的时钟中断函数task_tick，等到特定的时机执行真正的调度逻辑具体逻辑参考上述fair_sched_class的实现

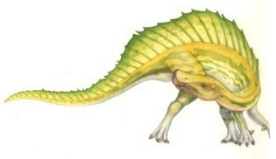
```
/*
 * This function gets called by the timer code, with HZ frequency.
 * We call it with interrupts disabled.
 */
void scheduler_tick(void)
{
        int cpu = smp_processor_id();
        // 取出当前CPU的运行队列
        struct rq *rq = cpu_rq(cpu);
        // 取出当前运行队列正在执行的任务的task_struct
        struct task_struct *curr = rq->curr;

        update_rq_clock(rq);
        // 调用当前任务的调度类的task_tick函数
        curr->sched_class->task_tick(rq, curr, 0);
  ◦ ◦ ◦
  ◦ ◦ ◦
}
```



实时调度 rt_sched_class

- 实时调度类实现在kernel/sched_rt.c中rt_sched_class

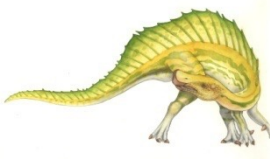
```
static const struct sched_class rt_sched_class = {
    .next           = &fair_sched_class,
    .enqueue_task       = enqueue_task_rt,
    .dequeue_task       = dequeue_task_rt,
    .yield_task     = yield_task_rt,

    .check_preempt_curr = check_preempt_curr_rt,

    .pick_next_task     = pick_next_task_rt,
    .put_prev_task      = put_prev_task_rt,

#ifdef CONFIG_SMP
    .select_task_rq     = select_task_rq_rt,

    .load_balance       = load_balance_rt,
    .move_one_task      = move_one_task_rt,
    .set_cpus_allowed       = set_cpus_allowed_rt,
    .rq_online              = rq_online_rt,
    .rq_offline             = rq_offline_rt,
    .pre_schedule       = pre_schedule_rt,
    .post_schedule      = post_schedule_rt,
    .task_woken     = task_woken_rt,
    .switched_from      = switched_from_rt,
#endif

    .set_curr_task          = set_curr_task_rt,
    .task_tick      = task_tick_rt,
```



实时调度实体：sched_rt_entity

- 在linux/sched.h中定义了面向实时调度的实体

```
struct sched_rt_entity {
    struct list_head run_list;
    unsigned long timeout;
    unsigned int time_slice;
    int nr_cpus_allowed;

    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
};
```



实时优先级队列rt_prio_array

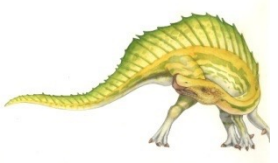
- 在kernel/sched.c中，是一组链表，每个优先级对应一个链表。还维护一个由101 bit组成的bitmap，其中实时进程优先级为0—99，占100 bit，再加1 bit的定界符。
- 当某个优先级别上有进程被插入列表时，相应的比特位就被置位。用sched_find_first_bit()函数查询该bitmap，它返回当前被置位的最高优先级的数组下标

```
struct rt_prio_array {
    DECLARE_BITMAP(bitmap, MAX_RT_PRIO+1); /* 包含1 bit的定界符 */
    struct list_head queue[MAX_RT_PRIO];
};
```

实时运行队列rt_rq

- 在kernel/sched.c中，用于组织实时调度的相关信息

```
struct rt_rq {
    struct rt_prio_array active;
    unsigned long rt_nr_running;
#if defined CONFIG_SMP || defined CONFIG_RT_GROUP_SCHED
    struct {
        int curr; /* 最高优先级的实时任务 */
#ifdef CONFIG_SMP
        int next; /* 下一个最高优先级的任务 */
#endif
    } highest_prio;
#endif
#ifdef CONFIG_SMP
    unsigned long rt_nr_migratory;
    unsigned long rt_nr_total;
    int overloaded;
    struct plist_head pushable_tasks;
#endif
    int rt_throttled;
    u64 rt_time;
    u64 rt_runtime;
    /* Nests inside the rq lock: */
    spinlock_t rt_runtime_lock;
```



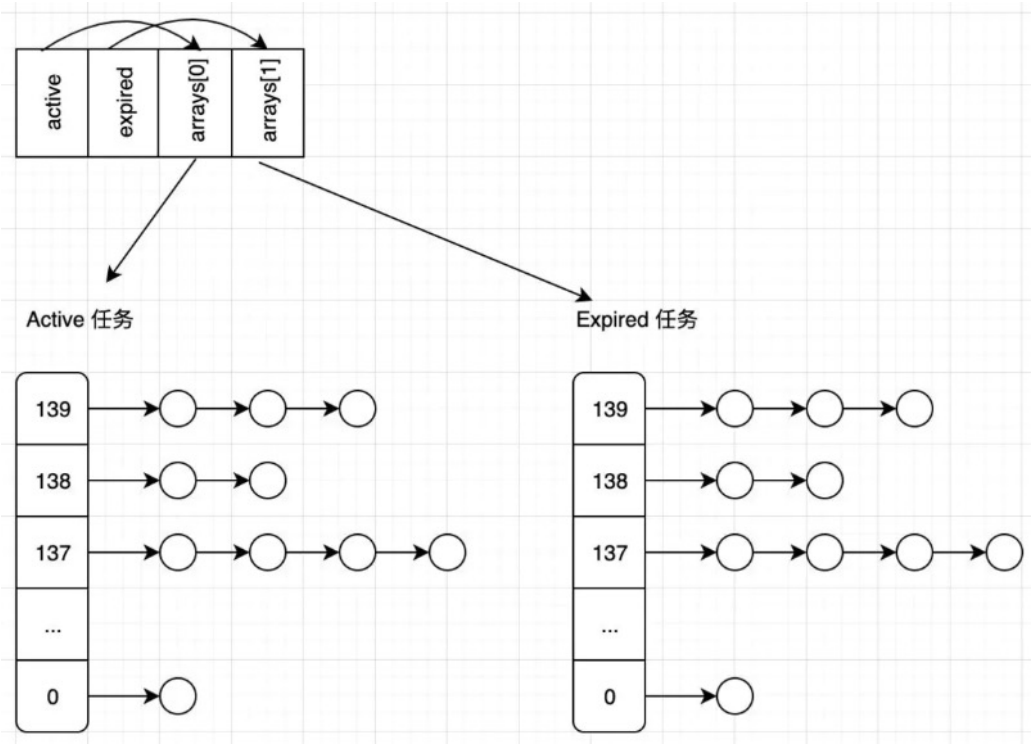
实时调度：选择下一个调度任务

```c
static struct sched_rt_entity *pick_next_rt_entity(struct rq *rq,
                                                   struct rt_rq *rt_rq)
{
    struct rt_prio_array *array = &rt_rq->active;
    struct sched_rt_entity *next = NULL;
    struct list_head *queue;
    int idx;
    /* 找到第一个可用的 */
    idx = sched_find_first_bit(array->bitmap);
    BUG_ON(idx >= MAX_RT_PRIO);
    /* 从链表组中找到对应的链表 */
    queue = array->queue + idx;
    next = list_entry(queue->next, struct sched_rt_entity, run_list);
    /* 返回找到的运行实体 */
    return next;
}
```

新的调度算法（since Linux 2.6）

- **Ingo Molnár** 大佬在 **2.6** 版本的内核中加入了全新的调度算法，它能够在常数时间内调度任务，因此被称为 **O(1)** 调度器

```
struct runqueue {
    unsigned long nr_running; /* 可运行的任务总数（某个 CPU） */
    struct prio_array *active; /* 指向 active 的队列的指针 */
    struct prio_array *expired; /* 指向 expired 的队列的指针 */
    struct prio_array arrays[2]; /* 实际存放不同优先级对应的任务链表 */
}
```



RR调度实现： task_tick_rt()

```
static void task_tick_rt(struct rq *rq, struct task_struct *p, int queued)
{
    update_curr_rt(rq);

    watchdog(rq, p);

    /*
     * RR tasks need a special form of timeslice management.
     * FIFO tasks have no timeslices.
     */
    if (p->policy != SCHED_RR) //RT任务只有两种算法，RR和FIFO；FIFO直到进程运行完毕都不会被调度，因此，直接返回。
        return;

    if (--p->rt.time_slice) //如果时间片尚未用完，则返回。
        return;

    p->rt.time_slice = DEF_TIMESLICE; //重新将p的时间片设为DEF_TIMESLICE，一般为100ms。

    /*
     * Requeue to the end of queue if we are not the only element
     * on the queue:
     */
    if (p->rt.run_list.prev != p->rt.run_list.next) {
        requeue_task_rt(rq, p, 0); //将p调整到队列的尾部，在下一次调度时不可能被调度到。
        set_tsk_need_resched(p); //将p设为可调度的。
    }
}
```