第 0004 讲 1 实时调度类及 SMP 和 NUMA

1、Linux 进程可分为两大类：实时进程和普通进程。实时进程与普通进程根本不同之处，如果系统中有一个实时进程且可执行，那么调度器总是会选择它，除非另有一个优先级更高实时进程。

SCHED_FIFO：没有时间片，在被调度器选择之后，可以运行任意长赶时间；

SCHED_RR：有时间片，其值在进程运行时会减少。

2、实时调度实体 sched_rt_entity 数据结构

表示实时调度实体，包含整个实时调度数据信息。具体内核源码如下：

include > linux > C sched.h > sched_dl_entity

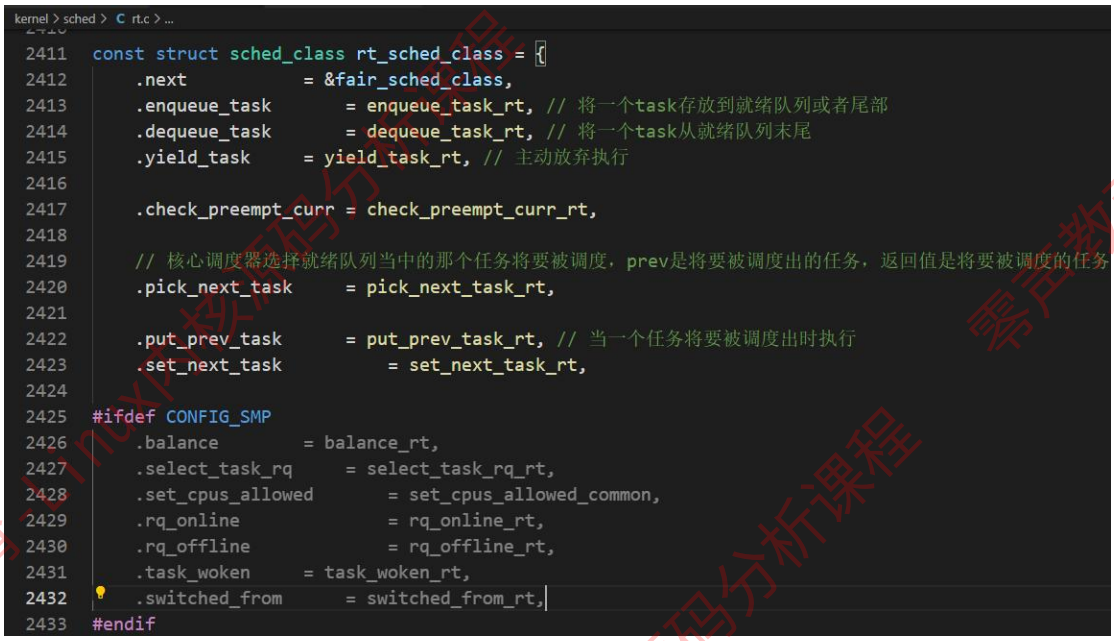
```
// 表示实时调度实体
struct sched_rt_entity {
    struct list_head        run_list; // 专门用于加入到优先级队列当中
    unsigned long           timeout; // 设置的时间超时
    unsigned long           watchdog_stamp; // 用于记录jiffies值
    unsigned int            time_slice; // 时间片
    unsigned short          on_rq;
    unsigned short          on_list;

    struct sched_rt_entity      *back; // 临时用于从上往下连接RT调度实体使用
#ifdef CONFIG_RT_GROUP_SCHED
    struct sched_rt_entity      *parent; // 指向父RT调度实体
    /* rq on which this entity is (to be) queued: */
    // 实时类
    struct rt_rq            *rt_rq; // RT调度实体所属的实时运行队列，被调度
    /* rq "owned" by this entity/group: */
    struct rt_rq            *my_q; // RT调度实体所拥有的实时运行队列，用于管理子任务或子组任务
#endif
} __randomize_layout;

```

有时间大家可以研究一下这个实时类：struct rt_rq

3、实时调度类 rt_sched_class 数据结构　　此数据结构的 Linux 内核源码如下：



```
kernel > sched > C rt.c > ...
2411  const struct sched_class rt_sched_class = {
2412      .next           = &fair_sched_class,
2413      .enqueue_task       = enqueue_task_rt, // 将一个task存放到就绪队列或者尾部
2414      .dequeue_task       = dequeue_task_rt, // 将一个task从就绪队列末尾
2415      .yield_task     = yield_task_rt, // 主动放弃执行
2416
2417      .check_preempt_curr = check_preempt_curr_rt,
2418
2419      // 核心调度器选择就绪队列当中的那个任务将要被调度，prev是将要被调度出的任务，返回值是将要被调度的任务
2420      .pick_next_task     = pick_next_task_rt,
2421
2422      .put_prev_task      = put_prev_task_rt, // 当一个任务将要被调度出时执行
2423      .set_next_task          = set_next_task_rt,
2424
2425  #ifdef CONFIG_SMP
2426      .balance        = balance_rt,
2427      .select_task_rq     = select_task_rq_rt,
2428      .set_cpus_allowed       = set_cpus_allowed_common,
2429      .rq_online              = rq_online_rt,
2430      .rq_offline             = rq_offline_rt,
2431      .task_woken     = task_woken_rt,
2432      .switched_from      = switched_from_rt,
2433  #endif
```

```
/* * Adding/removing a task to/from a priority array: * */
// 更新调度信息，将调度实体插入到相应优先级队列的末尾
static void
enqueue_task_rt(struct rq *rq, struct task_struct *p, int flags)
{
    struct sched_rt_entity *rt_se = &p->rt;

    if (flags & ENQUEUE_WAKEUP)
        rt_se->timeout = 0;

    enqueue_rt_entity(rt_se, flags);
```

```
    if (!task_current(rq, p) && p->nr_cpus_allowed > 1)
        enqueue_pushable_task(rq, p);
}

// 选择进程函数
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;
    // 第一个找到一个可用的实体
    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);
    return next; // 返回找到运行实体
}
```

```
// 删除进程
static void dequeue_task_rt(struct rq *rq, struct task_struct *p,
int flags)
{
    struct sched_rt_entity *rt_se = &p->rt;
    update_curr_rt(rq); // 更新调度数据信息等等
    // 将 rt_se 从运行队列当中删除，然后添加到队列尾部
    dequeue_rt_entity(rt_se, flags);
    dequeue_pushable_task(rq, p); // 从 hash 表当中进行删除
}
```

4、SMP 服务器 CPU 利用率最好的情况下是 2 至 4 个 CPU。这个结论经过实践证明。

5、从应用层系统架构，目前商用服务器大体分为三类：SMP、NUMA、MPP。

NUMA 优势：一台物理服务器内部集成多个 CPU，使系统具有较高事务处理能力。MUNA 架构适合 OLTP 事务处理环境。

SMP 优势：当前使用的 OTLP 程序当中，用户访问一个中断数据库，如果采用 SMP 系统构架，它的效率要比 MPP 架构要快。

6、内核对 CPU 管理通过 bitmap 来操作

include > linux > C cpumask.h > ...

```
91  extern struct cpumask __cpu_possible_mask;
92  extern struct cpumask __cpu_online_mask;
93  extern struct cpumask __cpu_present_mask;
94  extern struct cpumask __cpu_active_mask;
95
96  // 表示系统当中有多少个可以执行的CPU核心
97  #define cpu_possible_mask ((const struct cpumask *)&__cpu_possible_mask)
98
99  // 表示系统当中有多少个正在处于运行状态的CPU核心
100 #define cpu_online_mask   ((const struct cpumask *)&__cpu_online_mask)
101
102 // 表示系统当中有多少个具备online条件的CPU核心 ，它们不一定都处于online，有的CPU核心可能被热插拔
103 #define cpu_present_mask  ((const struct cpumask *)&__cpu_present_mask)
104
105 // 表示系统当中有多个少活跃的CPU核心
106 #define cpu_active_mask   ((const struct cpumask *)&__cpu_active_mask)
107
```