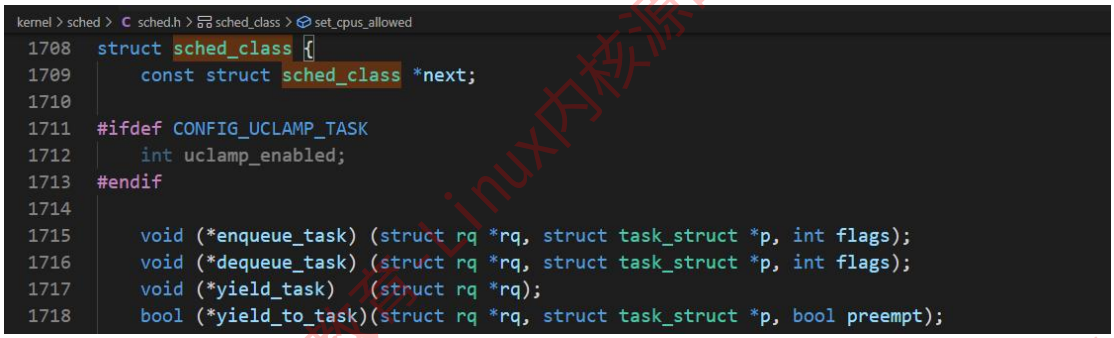
第 0003 讲 调度器及 CFS 调度器

1、调度：就是按照某种调度的算法设计，从进程的就绪队列当中选取进程分配 CPU，主要是协调对 CPU 等等相关资源使用。进程调度目的：最大限度利用 CPU 时间。

2、如果调度器支持就绪状态切换到执行状态，同时支持执行状态切换到就绪状，称该调度器为抢占式调度器。

3、调度类 sched_class 结构体源码分析




```
// 调度类 sched_class 结构体如下：
struct sched_class {
    /* 操作系统当中有多个调度类，按照调度优先级排成一个链表*/
    const struct sched_class *next;
#ifdef CONFIG_UCLAMP_TASK
    int uclamp_enabled;
#endif
```

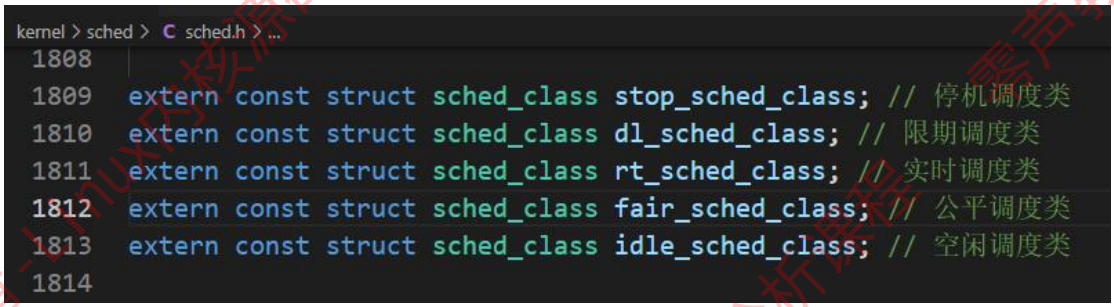
```
/* 将进程加入到执行队列当中，即将调度实体（进程）存放到红黑树当中，并对 nr_running 变量自动加 1 */
void (*enqueue_task) (struct rq *rq, struct task_struct *p, int flags);
/* 从执行队列当中删除进程，并对 nr_running 变量自动减 1 */
void (*dequeue_task) (struct rq *rq, struct task_struct *p, int flags);
/* 放弃 CPU 执行权限，实际上此函数执行先出队后入队，在这种情况它直接将调度实体存放在红黑树的最右端 */
void (*yield_task)  (struct rq *rq);
bool (*yield_to_task)(struct rq *rq, struct task_struct *p, bool preempt);
/* 专门用于检查当前进程是否可被新进程抢占 */
void (*check_preempt_curr)(struct rq *rq, struct task_struct *p, int flags);
/* 选择下一个要运行的进程*/
struct task_struct *(*pick_next_task)(struct rq *rq);
/* 将进程施加到运行队列当中*/
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);
   /* 为进程选择一个适合的 CPu*/
    int  (*select_task_rq)(struct task_struct *p, int task_cpu, int
sd_flag, int flags);
   /*迁移任务到另一个 CPU*/
   void (*migrate_task_rq)(struct task_struct *p, int new_cpu);
   /* 专门用于唤醒进程*/
      void  (*task_woken)(struct  rq  *this_rq,  struct  task_struct
*task);
   /* 修改进程在 CPU 的亲和力*/
   void (*set_cpus_allowed)(struct task_struct *p,
            const struct cpumask *newmask);
    /* 启动/禁止运行队列 */
   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
};
```

4、调度器类可分为：stop_sched_class、dl_sched_class、rt_sched_class、fair_sched_class 及 idle_sched_class。



```
kernel > sched > C sched.h > ...
1808
1809  extern const struct sched_class stop_sched_class; // 停机调度类
1810  extern const struct sched_class dl_sched_class; // 限期调度类
1811  extern const struct sched_class rt_sched_class; // 实时调度类
1812  extern const struct sched_class fair_sched_class; // 公平调度类
1813  extern const struct sched_class idle_sched_class; // 空闲调度类
1814
```

这 5 种调度类的优先级从高到低依次为：停机调度类、限期调度类、实时调度类、公平调度类、空闲调度类。

停机调度类：优先级是最高的调度类，停机进程是优先级最高的进程，可以抢占所有其它进程，其他进程不可能抢占停机进程。

限期调度类：最早使用优先算法，使用红黑树把进程按照绝对截止期限从小到大排序，每次调度时选择绝对截止期限最小的进程。

实时调度类：为每个调度优先级维护一个队列。

公平调度类：使用完全公平调度算法。完全公平调度算法引入虚拟运行时间的相关概念：虚拟运行时间=实际运行时间*nice0 对应的权重/进程的权重。

空闲调度类：每个 CPU 上有一个空闲线程，即 0 号线

程。空闲调度类优先级别最低，仅当没有其他进程可以调度的时候，才会调度空闲线程。

5、进程优先级

Linux 内核优先级源码如下：

```
include > linux > sched > C prio.h > ...
22
23  // Linux内核优先级如下
24  #define MAX_USER_RT_PRIO    100
25  #define MAX_RT_PRIO     MAX_USER_RT_PRIO
26
27  #define MAX_PRIO        (MAX_RT_PRIO + NICE_WIDTH)
28  #define DEFAULT_PRIO        (MAX_RT_PRIO + NICE_WIDTH / 2)
29
```

【进程分类】

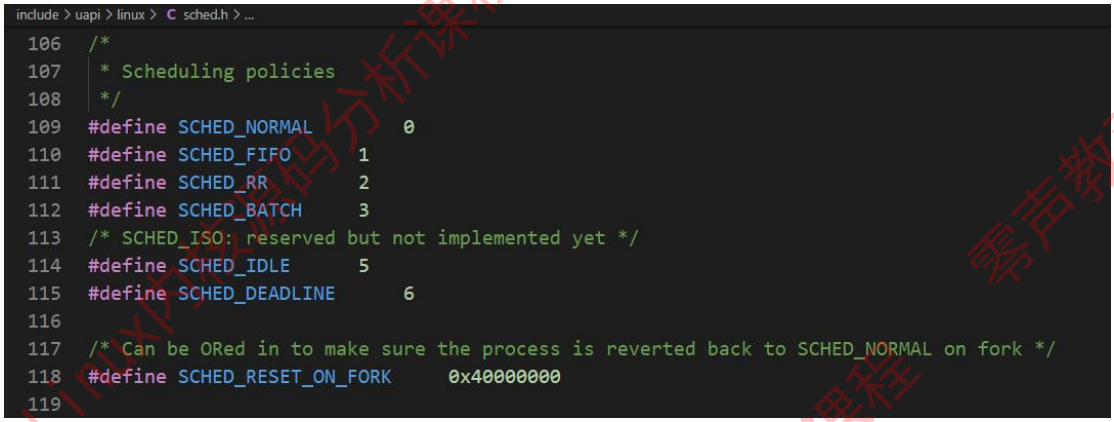
实时进程（Real-Time Process）：优先级高、需要立即被执行的进程。

普通进程（Normal Process）：优先级低、更长执行时间的进程。

进程的优先级是一个 0--139 的整数直接来表示。数字越小，优先级越高，其中优先级 0-99 留给实时进程，100-139 留给普通进程。我们使用如下图所示：

6、内核调度策略

Linux 内核提供一些调度策略供用户应用程序来选择调度器。Linux 内核调度策略源码分析如下：



SCHED_NORMAL：普通进程调度策略，使 task 选择 CFS 调度器来调度运行；

SCHED_FIFO：实时进程调度策略，先进先出调度没有时间片，没有更高优先级的状态下，只有等待主动让出 CPU；

SCHED_RR：实时进程调度策略，时间片轮转，进程使用完时间片之后加入优先级对应运行队列当中的尾部，把 CPU 让给同等优先级的其它进程；

SCHED_BATCH：普通进程调度策略，批量处理，使 task 选择 CFS 调度器来调度运行；

SCHED_IDLE：普通进程调度策略，使 task 以最低优先级选择 CFS 调度器来调度运行；

SCHED_DEADLINE：限期进程调度策略，使 task 选择 Deadline 调度器来调度运行；

备注：其中 Stop 调度器和 IDLE-task 调度器，仅使用于内核，用户没有办法进行选择。

7、实际运行时间

CFS 是 Completely Fair Scheduler 简称，完全公平调度器。在实际当中必须会有进程优先级高或者进程优先级低，CFS 调度器引入权重，使用权重代表进程的优先级，各个进程按照权重比例分配 CPU 时间。

假设有 2 个进程 X 和 Y，X 权重为 1024，Y 权重为 2048。

X 获得 CPU 时间比例为：1024/(1024+2048)=33%左右

Y 获得 CPU 时间比例为：2048/(1024+2048)=66%左右

在引入权重之后分配给进程的时间计算公式如下：

实际运行时间=调度曲*进程权重/所有进程权重之和。

8、虚拟运行时间

虚拟运行时间=实际运行时间*NICE_0_LOAD/进程权重=（调度周期*进程权重/所有进程权重之后）*NICE_0_LOAD/进程权重=调度周期*1024/所有进程总权重。

在一个调度周期里面，所有进程的虚拟运行时间是相同的，所以在进程调度时，只需要找到虚拟运行时间最小的进程调度运行即可。

9、调度子系统各个组件模块

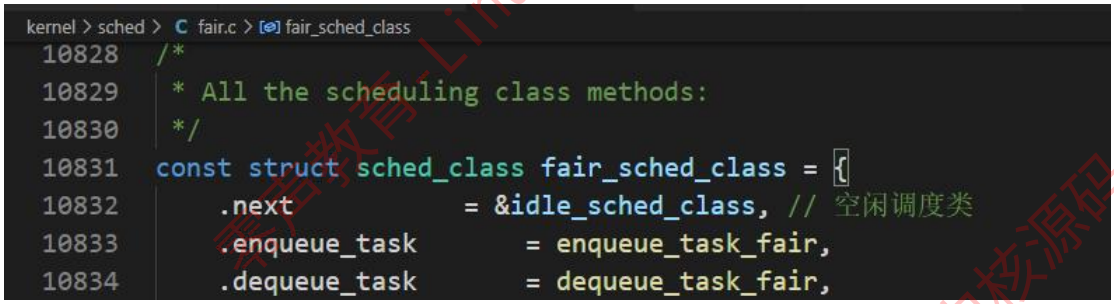
主调度器：通过调用 schedule()函数来完成进程的选择和切换。

周期性调度器：根据频率自动调用 scheduler_tick 函数，作用根据进程运行时间触发调度。

上下文切换：主要做两个事情（切换地址空间、切换寄存器和栈空间）。

10、CFS 调度器类

CFS 调度器类在 Linux 内核源码如下：



enqueue_task_fair：当任务进入可运行状态时，用此函数将调度实体存放到红黑树，完成入队操作。

dequeue_task_fair：当任务退出可运行状态进，用此函数将调度实体从红黑树中移除，完成出队操作，

11、CFS 调度器就绪队列内核源码

```
/* CFS-related fields in a runqueue */
struct cfs_rq {
    struct load_weight  load;
    unsigned long       runnable_weight;
    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;
#ifndef CONFIG_64BIT
    u64         min_vruntime_copy;
#endif
    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).
     */
    // sched_entity 可被内核调度的实体
    struct sched_entity *curr;
```

```
    struct sched_entity *next;
    struct sched_entity *last;
    struct sched_entity *skip;
#ifdef  CONFIG_SCHED_DEBUG
    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_sum;
    } 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 */
};
```

cfs_rq：跟踪就绪队列信息以及管理就绪态调度实体，并维护一查按照虚拟时间排序的红黑树。tasks_timeline->rb_root 是红黑树的根，tasks_timeline->rb_leftmost 指向红黑树中最左边的调度实体，即虚拟赶时间最小的调度实体。