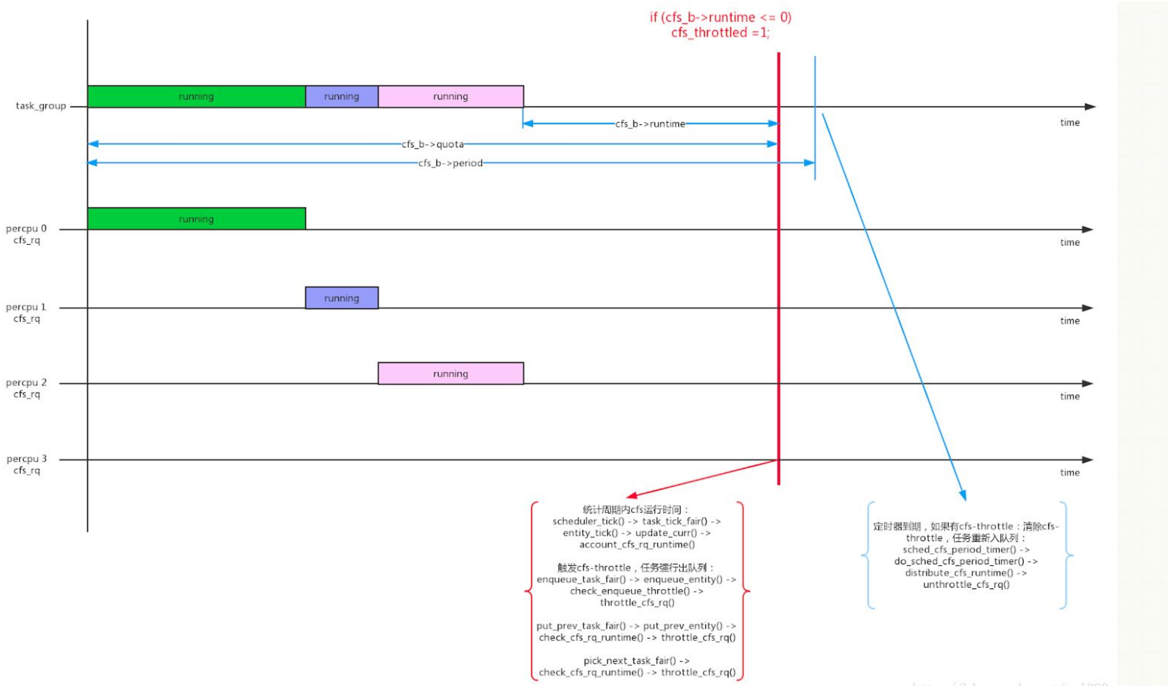
cfs bandwidth



1 cfs bandwidth是针对task_group的配置，一个task_group的bandwidth使用一个struct cfs_bandwidth *cfs_b数据结构来控制。

```
struct cfs_bandwidth {
#ifdef CONFIG_CFS_BANDWIDTH
    raw_spinlock_t lock;
    /*cfs bandwidth的监控周期，默认值是default_cfs_period() 0.1s
    */
    ktime_t period;
    /* quota:cfs task_group 在一个监控周期内的运行时间配额，默认值是RUNTIME_INF，
   无限大

    runtime:cfs task_group 在一个监控周期内剩余可运行的时间*/
    u64 quota, runtime;
    s64 hierarchical_quota;
    u64 runtime_expires;

    int idle, period_active;
    /*period_timer周期性的throttle动作，slack_timer是idle时候的timer*/
    struct hrtimer period_timer, slack_timer;
    struct list_head throttled_cfs_rq;

    /* statistics */
    int nr_periods, nr_throttled;
    u64 throttled_time;
#endif
};
```

我们首先通过运行图来简单的了解其工作原理:

- 系统首先会预算一个运行时间配额和剩余运行时间,两者默认是相等的
- 当某个task_group里的task开始运行一段时间之后,比如为delta,则剩余运行时间变成了初始的剩余运行时间-delta,更新新的剩余运行时间

- 如果在一个周期里面,剩余运行时间用光了,可以尝试那补偿5ms的时间,即总的运行时间减少了5ms,而剩余运行时间增加了5ms.
- 随着时间的流逝,剩余运行时间逐渐减少到0甚至为负值,如果在检测过程中,检测到了剩余运行时间已经使用完毕,那么系统就会额外的补偿给剩余运行时间数值为5-runtime_remaining(unit:ms).
- 在每次pick task的时候都会检测是否可以throttle,如果可以,则强制将enqueue的task dequeue,并有一个period timer(100ms)定时检测是否有rq throttle了,如果有则cfs调度算法重新对task进行调度操作.

下面是它的初始化:

```
/*执行slack_timer的回调函数*/
static enum hrtimer_restart sched_cfs_slack_timer(struct hrtimer *timer)
{
    struct cfs_bandwidth *cfs_b =
        container_of(timer, struct cfs_bandwidth, slack_timer);

    do_sched_cfs_slack_timer(cfs_b);

    return HRTIMER_NORESTART;
}
/*running period timer*/
static enum hrtimer_restart sched_cfs_period_timer(struct hrtimer *timer)
{
    struct cfs_bandwidth *cfs_b =
        container_of(timer, struct cfs_bandwidth, period_timer);
    int overrun;
    int idle = 0;

    raw_spin_lock(&cfs_b->lock);
    for (;;) {
        overrun = hrtimer_forward_now(timer, cfs_b->period);
        if (!overrun)
            break;

        idle = do_sched_cfs_period_timer(cfs_b, overrun);
    }
    if (idle)
        cfs_b->period_active = 0;
    raw_spin_unlock(&cfs_b->lock);

    return idle ? HRTIMER_NORESTART : HRTIMER_RESTART;
}
/*
 * default period for cfs group bandwidth.
 * default: 0.1s, units: nanoseconds
 */
static inline u64 default_cfs_period(void)
{
    return 100000000ULL;
}
/*cfs bandwidth的初始化*/
void init_cfs_bandwidth(struct cfs_bandwidth *cfs_b)
```

```
{
    raw_spin_lock_init(&cfs_b->lock);
    cfs_b->runtime = 0;
    cfs_b->quota = RUNTIME_INF;
    cfs_b->period = ns_to_ktime(default_cfs_period());

    INIT_LIST_HEAD(&cfs_b->throttled_cfs_rq);
    /*周期性处理cfs bandwidth上的task_group*/
    hrtimer_init(&cfs_b->period_timer, CLOCK_MONOTONIC,
HRTIMER_MODE_ABS_PINNED);
    cfs_b->period_timer.function = sched_cfs_period_timer;
    hrtimer_init(&cfs_b->slack_timer, CLOCK_MONOTONIC, HRTIMER_MODE_REL);
    cfs_b->slack_timer.function = sched_cfs_slack_timer;
}

static void init_cfs_rq_runtime(struct cfs_rq *cfs_rq)
{
    cfs_rq->runtime_enabled = 0;
    INIT_LIST_HEAD(&cfs_rq->throttled_list);
}

void start_cfs_bandwidth(struct cfs_bandwidth *cfs_b)
{
    lockdep_assert_held(&cfs_b->lock);

    if (!cfs_b->period_active) {
        cfs_b->period_active = 1;
        hrtimer_forward_now(&cfs_b->period_timer, cfs_b->period);
        hrtimer_start_expires(&cfs_b->period_timer,
HRTIMER_MODE_ABS_PINNED);
    }
}
```

2. 因为一个task_group是在percpu上都创建了一个cfs_rq，所以cfs_b->quota的值是这些percpu cfs_rq中的进程共享的，每个percpu cfs_rq在运行时需要向tg->cfs_bandwidth->runtime来申请；
scheduler_tick() -> task_tick_fair() -> entity_tick() -> update_curr() -> account_cfs_rq_runtime()

```
scheduler_tick() -> task_tick_fair() -> entity_tick() ->
update_curr() -> account_cfs_rq_runtime()

↓

static __always_inline
void account_cfs_rq_runtime(struct cfs_rq *cfs_rq, u64
delta_exec)
{
    if (!cfs_bandwidth_used() || !cfs_rq->runtime_enabled)
        return;

    __account_cfs_rq_runtime(cfs_rq, delta_exec);
```

```
}

|→

static void __account_cfs_rq_runtime(struct cfs_rq *cfs_rq,
u64 delta_exec)
{
    /* (1) 用cfs_rq已经申请的时间配额(cfs_rq->runtime_remaining)
减去已经消耗的时间 */
    /* dock delta_exec before expiring quota (as it could
span periods) */
    cfs_rq->runtime_remaining -= delta_exec;

    /* (2) cfs_b与cfs_rq的 runtime_expire的比较之后做出决策 */
    expire_cfs_rq_runtime(cfs_rq);

    /* (3) 如果cfs_rq已经申请的时间配额还没用完，返回 */
    if (likely(cfs_rq->runtime_remaining > 0))
        return;

    /*
     * if we're unable to extend our runtime we resched so
that the active
     * hierarchy can be throttled
     */
    /* (4) 如果cfs_rq申请的时间配额已经用完，尝试向tg的
cfs_b->runtime申请新的时间片
        如果申请新时间片失败，说明整个tg已经没有可运行时间了，把本进程设
置为需要重新调度，
        在中断返回，发起schedule()时，发现
cfs_rq->runtime_remaining<=0，会调用throttle_cfs_rq()对cfs_rq进
行实质的限制
     */
    if (!assign_cfs_rq_runtime(cfs_rq) &&
likely(cfs_rq->curr))
        resched_curr(rq_of(cfs_rq));
}

||→

static int assign_cfs_rq_runtime(struct cfs_rq *cfs_rq)
{
    struct task_group *tg = cfs_rq->tg;
    struct cfs_bandwidth *cfs_b = tg_cfs_bandwidth(tg);
    u64 amount = 0, min_amount, expires;

    /* (4.1) cfs_b的分配时间片的默认值是5ms */
    /* note: this is a positive sum as runtime_remaining <= 0
*/
    min_amount = sched_cfs_bandwidth_slice() -
cfs_rq->runtime_remaining;
```

```
    raw_spin_lock(&cfs_b->lock);
    if (cfs_b->quota == RUNTIME_INF)
        /* (4.2) RUNTIME_INF类型，时间是分配不完的 */
        amount = min_amount;
    else {
        start_cfs_bandwidth(cfs_b);

        /* (4.3) 剩余时间cfs_b->runtime减去分配的时间片,runtime
        - amount目的是告知系统,我增加了amount数量的配额,所以
        runtime需要减去amount,表示仅仅运行了runtime-amount时间
        目的还是按照period做判决throttle */
        if (cfs_b->runtime > 0) {
            amount = min(cfs_b->runtime, min_amount);
            cfs_b->runtime -= amount;
            cfs_b->idle = 0;
        }
    }
    expires = cfs_b->runtime_expires;
    raw_spin_unlock(&cfs_b->lock);

    /* (4.4) 增加分配的时间片赋值给cfs_rq原先的配额 */
    cfs_rq->runtime_remaining += amount;
    /*
     * we may have advanced our local expiration to account for allowed
     * spread between our sched_clock and the one on which runtime was
     * issued.
     */
    if ((s64)(expires - cfs_rq->runtime_expires) > 0)
        cfs_rq->runtime_expires = expires;

    /* (4.5) 判断分配时间是否足够? */
    return cfs_rq->runtime_remaining > 0;
}
```

3. 在enqueue_task_fair()、put_prev_task_fair()、pick_next_task_fair()这几个时刻，会 check cfs_rq是否已经达到throttle，如果达到cfs throttle会把cfs_rq dequeue停止运行；

```
enqueue_task_fair() -> enqueue_entity() -> check_enqueue_throttle() -> throttle_cfs_rq()
put_prev_task_fair() -> put_prev_entity() -> check_cfs_rq_runtime() -> throttle_cfs_rq()
pick_next_task_fair() -> check_cfs_rq_runtime() -> throttle_cfs_rq()


/*
 * When a group wakes up we want to make sure that its quota is not already
 * expired/exceeded, otherwise it may be allowed to steal additional ticks of
 * runtime as update_curr() throttling can not not trigger until it's on-rq.
 */
static void check_enqueue_throttle(struct cfs_rq *cfs_rq)
```

```
{
    if (!cfs_bandwidth_used())
        return;
    /*检测进程组上下节点是否throttle,并做对应的参数update*/
    /* Synchronize hierarchical throttle counter: */
    if (unlikely(!cfs_rq->throttle_uptodate)) {
        struct rq *rq = rq_of(cfs_rq);
        struct cfs_rq *pcfs_rq;
        struct task_group *tg;

        cfs_rq->throttle_uptodate = 1;

        /* Get closest up-to-date node, because leaves go first: */
        for (tg = cfs_rq->tg->parent; tg; tg = tg->parent) {
            pcfs_rq = tg->cfs_rq[cpu_of(rq)];
            if (pcfs_rq->throttle_uptodate)
                break;
        }
        if (tg) {
            cfs_rq->throttle_count = pcfs_rq->throttle_count;
            cfs_rq->throttled_clock_task = rq_clock_task(rq);
        }
    }

    /* an active group must be handled by the update_curr()->put() path */
    if (!cfs_rq->runtime_enabled || cfs_rq->curr)
        return;
     /*如果已经throttle,则直接返回*/
    /* ensure the group is not already throttled */
    if (cfs_rq_throttled(cfs_rq))
        return;
    /*update last runtime*/
    /* update runtime allocation */
    account_cfs_rq_runtime(cfs_rq, 0);
    /*配额用完,进行throttle*/
    if (cfs_rq->runtime_remaining <= 0)
        throttle_cfs_rq(cfs_rq);
}

/* conditionally throttle active cfs_rq's from put_prev_entity() */
static bool check_cfs_rq_runtime(struct cfs_rq *cfs_rq)
{
    if (!cfs_bandwidth_used())
        return false;

    /* (2.1) 如果cfs_rq->runtime_remaining还有运行时间, 直接返回 */
    if (likely(!cfs_rq->runtime_enabled || cfs_rq->runtime_remaining > 0))
        return false;

    /*
     * it's possible for a throttled entity to be forced into a running
     * state (e.g. set_curr_task), in this case we're finished.
     */
    /* (2.2) 如果已经throttle, 直接返回 */
```

```
    if (cfs_rq_throttled(cfs_rq))
        return true;

    /* (2.3) 已经throttle，执行throttle动作 */
    throttle_cfs_rq(cfs_rq);
    return true;
}

static void throttle_cfs_rq(struct cfs_rq *cfs_rq)
{
    struct rq *rq = rq_of(cfs_rq);
    struct cfs_bandwidth *cfs_b = tg_cfs_bandwidth(cfs_rq->tg);
    struct sched_entity *se;
    long task_delta, dequeue = 1;
    bool empty;

    se = cfs_rq->tg->se[cpu_of(rq_of(cfs_rq))];

    /* freeze hierarchy runnable averages while throttled */
    rcu_read_lock();
    walk_tg_tree_from(cfs_rq->tg, tg_throttle_down, tg_nop, (void *)rq);
    rcu_read_unlock();

    task_delta = cfs_rq->h_nr_running;
    for_each_sched_entity(se) {
        struct cfs_rq *qcfs_rq = cfs_rq_of(se);
        /* throttled entity or throttle-on-deactivate */
        if (!se->on_rq)
            break;

        /* (3.1) throttle的动作1：将cfs_rq dequeue停止运行 */
        if (dequeue)
            dequeue_entity(qcfs_rq, se, DEQUEUE_SLEEP);
        qcfs_rq->h_nr_running -= task_delta;

        if (qcfs_rq->load.weight)
            dequeue = 0;
    }

    if (!se)
        sub_nr_running(rq, task_delta);

    /* (3.2) throttle的动作2：将cfs_rq->throttled置位 */
    cfs_rq->throttled = 1;
    cfs_rq->throttled_clock = rq_clock(rq);
    raw_spin_lock(&cfs_b->lock);
    empty = list_empty(&cfs_b->throttled_cfs_rq);

    /*
     * Add to the _head_ of the list, so that an already-started
     * distribute_cfs_runtime will not see us
     */
    list_add_rcu(&cfs_rq->throttled_list, &cfs_b->throttled_cfs_rq);

```

```
        /*
         * If we're the first throttled task, make sure the bandwidth
         * timer is running.
         */
        if (empty)
            start_cfs_bandwidth(cfs_b);  /*启动定时器throttle检测*/

        raw_spin_unlock(&cfs_b->lock);
    }
```

4.对每一个tg的cfs_b，系统会启动一个周期性定时器cfs_b->period_timer，运行周期为cfs_b->period。主 要作用是period到期后检查是否有cfs_rq被throttle，如果被throttle恢复它，并进行新一轮的监控；

```
sched_cfs_period_timer() -> do_sched_cfs_period_timer()

↓

static int do_sched_cfs_period_timer(struct cfs_bandwidth
*cfs_b, int overrun)
{
    u64 runtime, runtime_expires;
    int throttled;

    /* no need to continue the timer with no bandwidth
constraint */
    if (cfs_b->quota == RUNTIME_INF)
        goto out_deactivate;

    throttled = !list_empty(&cfs_b->throttled_cfs_rq);
    cfs_b->nr_periods += overrun;

    /*
     * idle depends on !throttled (for the case of a large
deficit), and if
     * we're going inactive then everything else can be
deferred
     */
    if (cfs_b->idle && !throttled)
        goto out_deactivate;

    /* (1) 新周期的开始，给cfs_b->runtime重新赋值为cfs_b->quota
     并更新runtime_expires = now + ktime_to_ns(cfs_b->period)
*/
    __refill_cfs_bandwidth_runtime(cfs_b);

    if (!throttled) {
        /* mark as potentially idle for the upcoming period
*/
        cfs_b->idle = 1;
        return 0;
    }
```

```c

    /* account preceding periods in which throttling occurred */
    cfs_b->nr_throttled += overrun;

    runtime_expires = cfs_b->runtime_expires;

    /*
     * This check is repeated as we are holding onto the new bandwidth while
     * we unthrottle. This can potentially race with an unthrottled group
     * trying to acquire new bandwidth from the global pool. This can result
     * in us over-using our runtime if it is all used during this loop, but
     * only by limited amounts in that extreme case.
     */
    /* (2) 解除cfs_b->throttled_cfs_rq中所有被throttle住的cfs_rq */
    while (throttled && cfs_b->runtime > 0) {
        runtime = cfs_b->runtime;
        raw_spin_unlock(&cfs_b->lock);
        /* we can't nest cfs_b->lock while distributing bandwidth */
        runtime = distribute_cfs_runtime(cfs_b, runtime,
                         runtime_expires);
        raw_spin_lock(&cfs_b->lock);

        throttled = !list_empty(&cfs_b->throttled_cfs_rq);

        cfs_b->runtime -= min(runtime, cfs_b->runtime);
    }

    /*
     * While we are ensured activity in the period following an
     * unthrottle, this also covers the case in which the new bandwidth is
     * insufficient to cover the existing bandwidth deficit. (Forcing the
     * timer to remain active while there are any throttled entities.)
     */
    cfs_b->idle = 0;

    return 0;

out_deactivate:
    return 1;
}
```

```
|→

static u64 distribute_cfs_runtime(struct cfs_bandwidth *cfs_b,
        u64 remaining, u64 expires)
{
    struct cfs_rq *cfs_rq;
    u64 runtime;
    u64 starting_runtime = remaining;

    rcu_read_lock();
    list_for_each_entry_rcu(cfs_rq, &cfs_b->throttled_cfs_rq,
                throttled_list) {
        struct rq *rq = rq_of(cfs_rq);

        raw_spin_lock(&rq->lock);
        if (!cfs_rq_throttled(cfs_rq))
            goto next;

        runtime = -cfs_rq->runtime_remaining + 1;
        if (runtime > remaining)
            runtime = remaining;
        remaining -= runtime;

        cfs_rq->runtime_remaining += runtime;
        cfs_rq->runtime_expires = expires;

        /* (2.1) 解除throttle */
        /* we check whether we're throttled above */
        if (cfs_rq->runtime_remaining > 0)
            unthrottle_cfs_rq(cfs_rq);

next:
        raw_spin_unlock(&rq->lock);

        if (!remaining)
            break;
    }
    rcu_read_unlock();

    return starting_runtime - remaining;
}

||→

void unthrottle_cfs_rq(struct cfs_rq *cfs_rq)
{
    struct rq *rq = rq_of(cfs_rq);
    struct cfs_bandwidth *cfs_b = tg_cfs_bandwidth(cfs_rq->tg);
```

```
	struct sched_entity *se;
	int enqueue = 1;
	long task_delta;

	se = cfs_rq->tg->se[cpu_of(rq)];

	cfs_rq->throttled = 0;

	update_rq_clock(rq);

	raw_spin_lock(&cfs_b->lock);
	cfs_b->throttled_time += rq_clock(rq) -
cfs_rq->throttled_clock;
	list_del_rcu(&cfs_rq->throttled_list);
	raw_spin_unlock(&cfs_b->lock);

	/* update hierarchical throttle state */
	walk_tg_tree_from(cfs_rq->tg, tg_nop, tg_unthrottle_up,
(void *)rq);

	if (!cfs_rq->load.weight)
		return;

	task_delta = cfs_rq->h_nr_running;
	for_each_sched_entity(se) {
		if (se->on_rq)
			enqueue = 0;

		cfs_rq = cfs_rq_of(se);
		/* (2.1.1) 重新enqueue运行 */
		if (enqueue)
			enqueue_entity(cfs_rq, se, ENQUEUE_WAKEUP);
		cfs_rq->h_nr_running += task_delta;

		if (cfs_rq_throttled(cfs_rq))
			break;
	}

	if (!se)
		add_nr_running(rq, task_delta);

	/* determine whether we need to wake up potentially idle
cpu */
	if (rq->curr == rq->idle && rq->cfs.nr_running)
		resched_curr(rq);
}
```

明白其思路就可以.但是我看了好几个手机平台都没有定义CONFIG_CFS_BANDWIDTH,似乎都没有使用.目前cpu速度越来越快,处理能力一般都没什么问题,不需要throttle.