CHINA LINUX KERNEL
中国Linux内核开发者大会

vivo

对EEVDF的一点优化及思考

--- 臧春鑫

CHINA LINUX KERNEL
中国Linux内核开发者大会

vivo

目录

CONTENTS

01

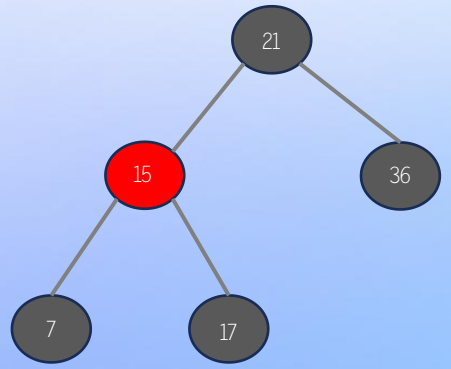
EEVDF 背景

EEVDF Technical Background

EEVDF 背景

为什么引入 EEVDF

- 简单回顾 CFS 的核心实现
  - 维护一个 vruntime
  - vruntime 的增涨速度 $\frac{NICE\_0\_LOAD}{WEITGH}$
  - 按照 vruntime 的大小组织成一颗 RBTree
  - 总是运行 vruntime 最小的任务



EEVDF 背景

- CFS 的不足
  - 缺乏对时间紧迫性的表达
    - 只知道"谁跑得少"，但不知道"谁需要马上跑"

```
static int
wakeup_preempt_entity(struct sched_entity *curr, struct sched_entity *se)
{
        s64 gran, vdiff = curr->vruntime - se->vruntime;

        if (vdiff <= 0)
                return -1;

        gran = wakeup_gran(se);
        if (vdiff > gran)
                return 1;

        return 0;
}
```

```
static unsigned long wakeup_gran(struct sched_entity *se)
{
        unsigned long gran = sysctl_sched_wakeup_granularity;
```

EEVDF 背景

- CFS 的不足
  - 较多的启发式逻辑
    - sched_feat(START_DEBIT)
    - sched_feat(ALT_PERIOD)
    - sched_feat(BASE_SLICE)
    - sysctl_sched_idle_min_granularity
    - sched_nr_latency
    - …

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

	/*
	 * The 'current' period is already promised to the current tasks,
	 * however the extra weight of the new task will slow them down a
	 * little, place the new task so that it fits in the slot that
	 * stays open at the end.
	 */
	if (initial && sched_feat(START_DEBIT))
		vruntime += sched_vslice(cfs_rq, se);

	/* sleeps up to a single latency don't count. */
	if (!initial) {
		unsigned long thresh;

		if (se_is_idle(se))
			thresh = sysctl_sched_min_granularity;
		else
			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;

		vruntime -= thresh;
	}
```

```
static void
check_preempt_tick(struct cfs_rq *cfs_rq, struct sched_entity *curr)
{
	unsigned long ideal_runtime, delta_exec;
	struct sched_entity *se;
	s64 delta;

	/*
	 * When many tasks blow up the sched_period; it is possible that
	 * sched_slice() reports unusually large results (when many tasks are
	 * very light for example). Therefore impose a maximum.
	 */
	ideal_runtime = min_t(u64, sched_slice(cfs_rq, curr), sysctl_sched_latency);

	delta_exec = curr->sum_exec_runtime - curr->prev_sum_exec_runtime;
	if (delta_exec > ideal_runtime) {
		resched_curr(rq_of(cfs_rq));
		/*
		 * The current task ran long enough, ensure it doesn't get
		 * re-elected due to buddy favours.
		 */
		clear_buddies(cfs_rq, curr);
		return;
	}
```

EEVDF 背景

EEVDF 进入主线

- *1995《Earliest Eligible Virtual Deadline First: A Flexible and Accurate Mechanism for Proportional Share Resource Allocation》* Ion Stoica, Hussein Abdel-Wahab

- Peter Zijlstra 于 2023 年末实现 EEVDF 初版合入到 v6.6

EEVDF 背景

EEVDF 简介

EEVDF 是 *Earliest Eligible Virtual Deadline First* 的缩写，即“最早合格虚拟截止时间优先”。

- *Virtual Deadline*
  - 不是一个真实的、用户设置的截止时间。是调度器基于任务的“权重”为每个任务动态计算的一个在理想状态下运行需要截止的虚拟时间节点
  - 权重越大的任务，每次分配的虚拟运行时间越短，Virtual Deadline 越小
- *Eligible*
  - 合格，用于表达进程是否超额运行。
- *Earliest*
  - 最早，即需要对所有运行的任务进行有效组织，可以找到 Virtual Deadline 最小的任务

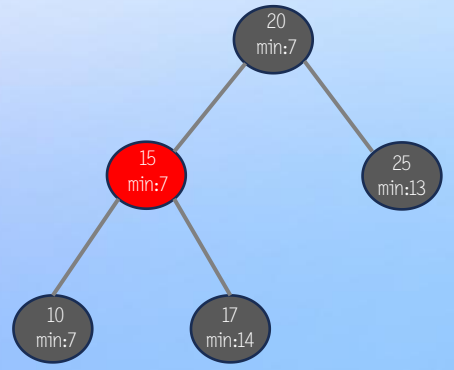
即，优先运行截止时间最小的合格任务

EEVDF 背景

EEVDF 核心实现

- lag（vlag）
  - 用于判断任务是否合格
  - lag =“应得运行时间”-“实际运行时间”
  - lag >= 0，即进程需要补偿运行或正常运行，称“Eligible”。如果 lag < 0，即进程超额运行，称“ ineligible”

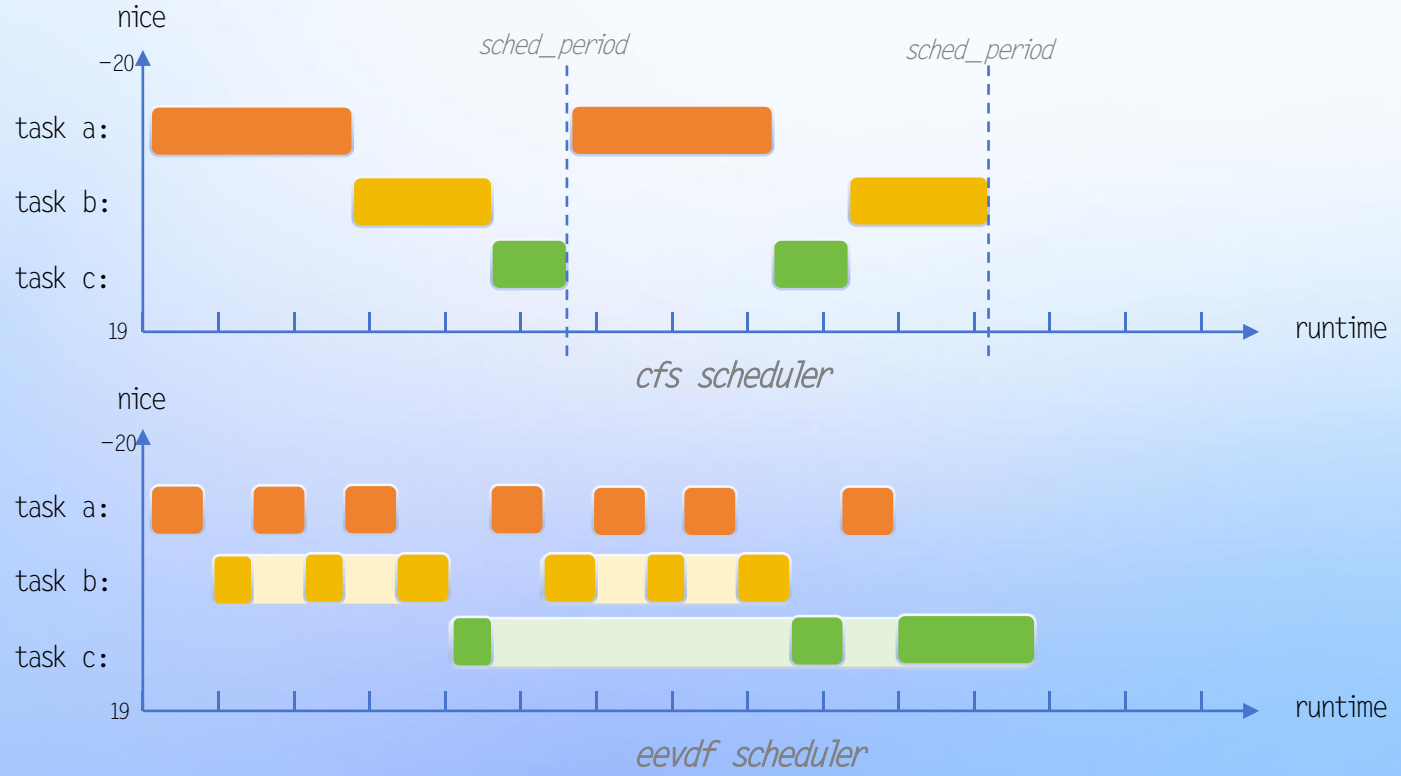
- 进程的组织方式
  - Augmented RBTree
    - 新添加成员 se->min_vruntime
  - 按照 deadline 大小组织



EEVDF 背景

EEVDF 任务运行表现

n 通俗来说“优先级越高的任务，每次运行的时间越短越频繁”



02

EEVDF 性能摸底

EEVDF Performance Baseline Testing

EEVDF 性能摸底

测试环境

- KERNEL: v6.8
- CPU: intel i7-13th

测试方法

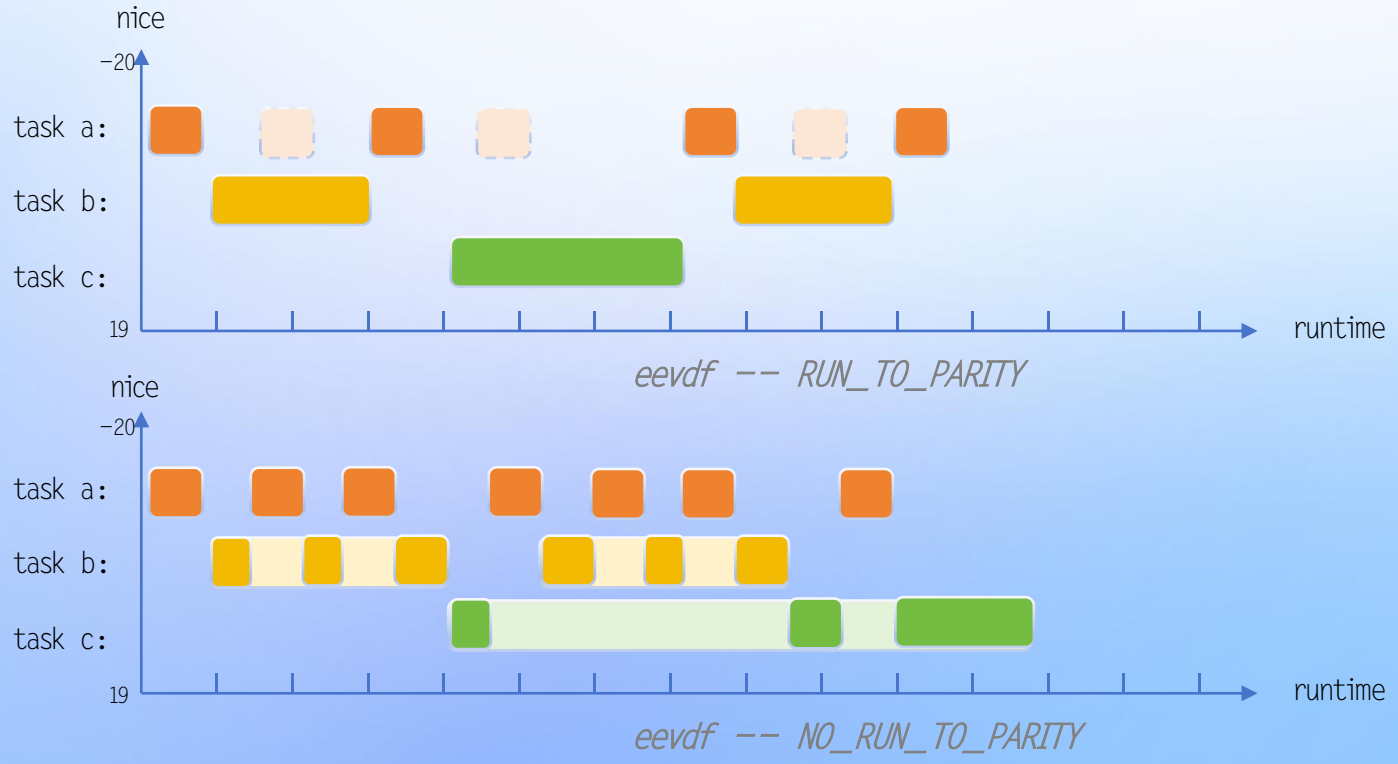
- 使用 hackbench 对系统加压
- 然后使用 cyclictest 进行调度时延测试

nice值	latency(us)	CFS	EEVDF	EEVDF no parity
-19	avg:	66	191	89
	max:	11277	15442	14529
0	avg:	510	466	289
	max:	22247	38917	32665
19	avg:	9362	37151	18293
	max:	166190	2688299	426196

EEVDF 性能摸底

sched_feat(RUN_TO_PARITY)

- *“Inhibit (wakeup) preemption until the current task has either matched the 0-lag point or until is has exhausted it's slice.”*
- 抑制（唤醒）抢占，直到当前任务到达 0-lag 点或其 slice 耗尽



03

调度时延优化

Scheduling Latency Optimization

调度时延优化

此次 eevdf 修改中影响调度时延的关键节点有哪些

- *pick_next_entity*
  - 选择下一个运行se逻辑
  - pick_eevdf，选出deadline最小的合格进程

- *check_preempt_wakeup_fair*
  - 是否抢占 curr 进程

```
        cfs_rq = cfs_rq_of(se);
        update_curr(cfs_rq);

        /*
         * XXX pick_eevdf(cfs_rq) != se ?
         */
        if (pick_eevdf(cfs_rq) == pse)
                goto preempt;

        return;

preempt:
        resched_curr(rq);
}
```

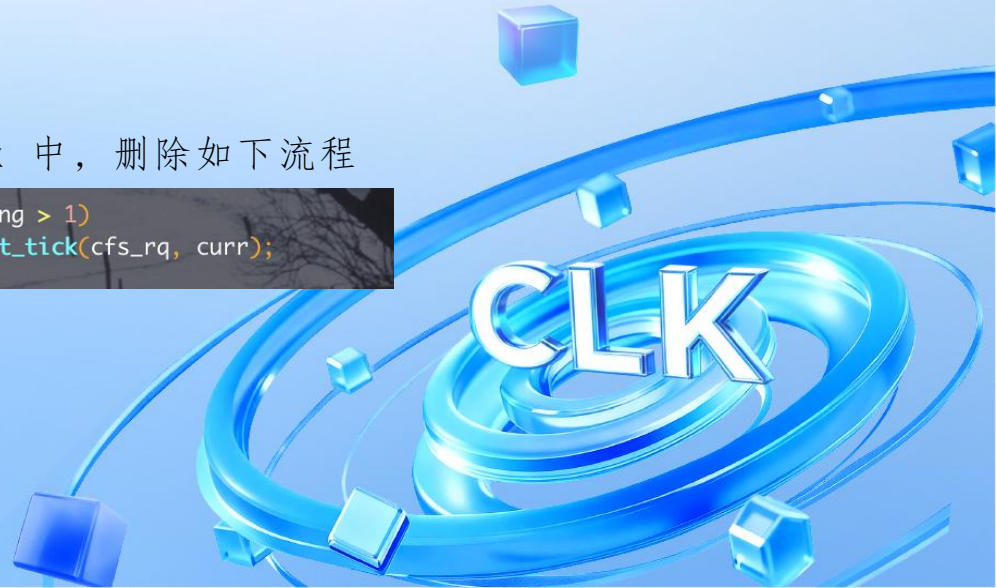
- *update_curr*
  - 新增调用 update_deadline
  - update_deadline 中会判断进程是否resched

```
static void update_deadline(struct cfs_rq *cfs_rq, struct sched_entity *se)
{
        if ((s64)(se->vruntime - se->deadline) < 0)
                return;
```

```
        if (cfs_rq->nr_running > 1) {
                resched_curr(rq_of(cfs_rq));
                clear_buddies(cfs_rq, se);
        }
}
```

- *tick 周期检查*
  - 在entity_tick 中，删除如下流程

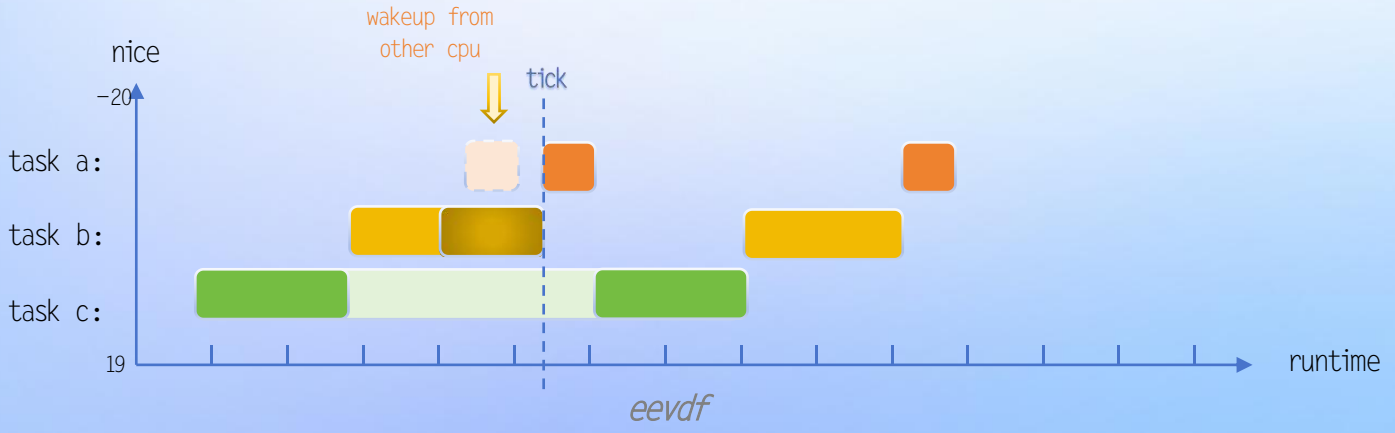
```
if (cfs_rq->nr_running > 1)
        check_preempt_tick(cfs_rq, curr);
```



调度时延优化

发现问题

- 目前流程中基本所有因进程不合格导致进程切换的场景都是由 *update_deadline* 覆盖
- 但会存在如下场景
  - 一个eligible的进程从 cpu1 唤醒/切换到 cpu0上，会导致 cpu0 上的一些进程由 eligible 变为 ineligible



调度时延优化

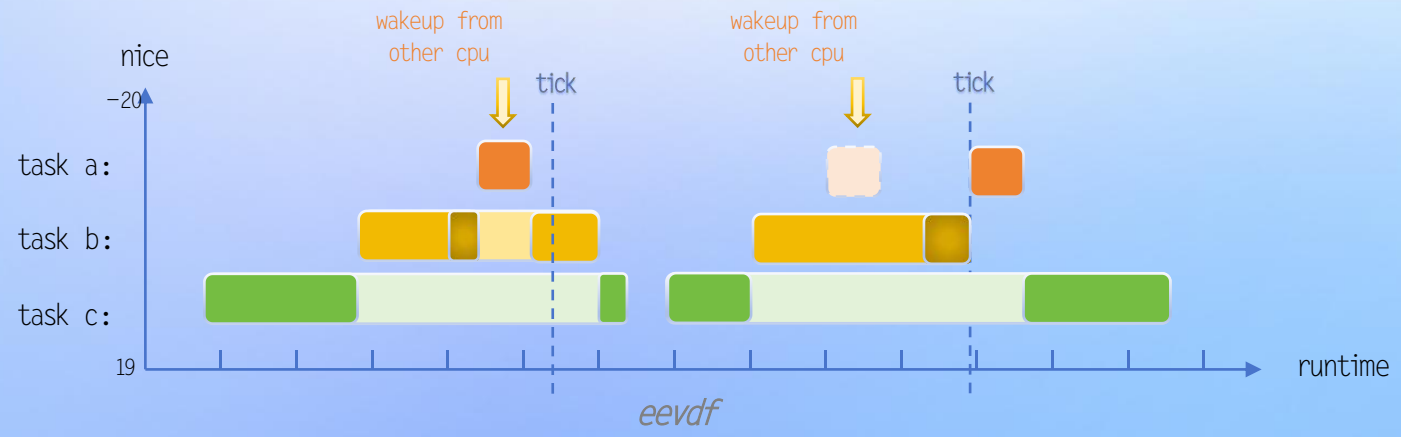
尝试优化

- *update_curr*
  - 新增 *check_entity_need_preempt* 调用
  - 将 *update_deadline* 中的 resched 操作移到 *update_curr* 流程中
  - 不仅检查 curr 进程的 vruntime 是否超过分配的 deadline，同时还要检查是否 eligible

```
resched = update_deadline(cfs_rq, curr);
if (resched || check_entity_need_preempt(cfs_rq, curr)) {
        resched_curr(rq);
        clear_buddies(cfs_rq, curr);
}
```

```
static bool check_entity_need_preempt(struct cfs_rq *cfs_rq, struct sched_entity *se)
{
        if (cfs_rq->nr_running <= 1 || entity_eligible(cfs_rq, se))
                return false;

        return true;
}
```

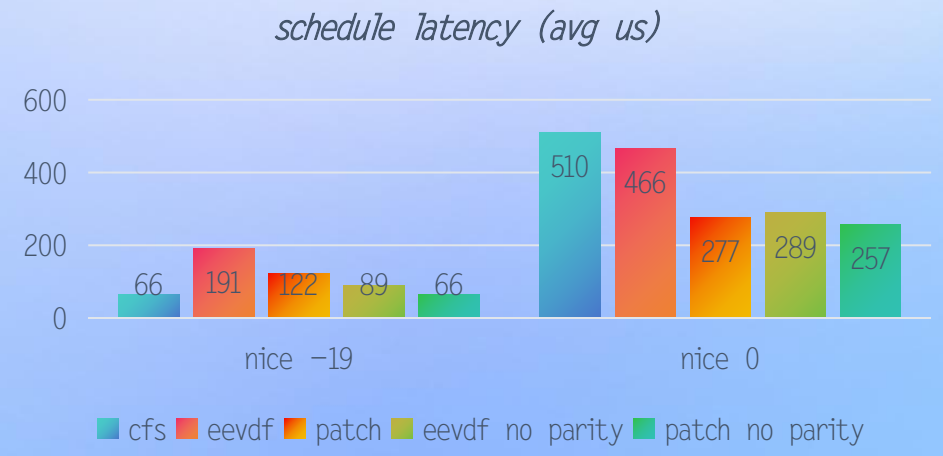


调度时延优化

v2 版本对比数据

n 调度时延对比。-19 nice 进程平均时延降低约30%左右，尾部时延降低约50%

nice值	latency(us)	CFS	EEVDF	PATCH	EEVDF no parity	PATCH no parity
-19	avg:	66	191	122	89	66
	max:	11277	15442	07648	14529	07713
0	avg:	510	466	277	289	257
	max:	22247	38917	32391	32665	17710
19	avg:	9362	37151	31045	18293	23035

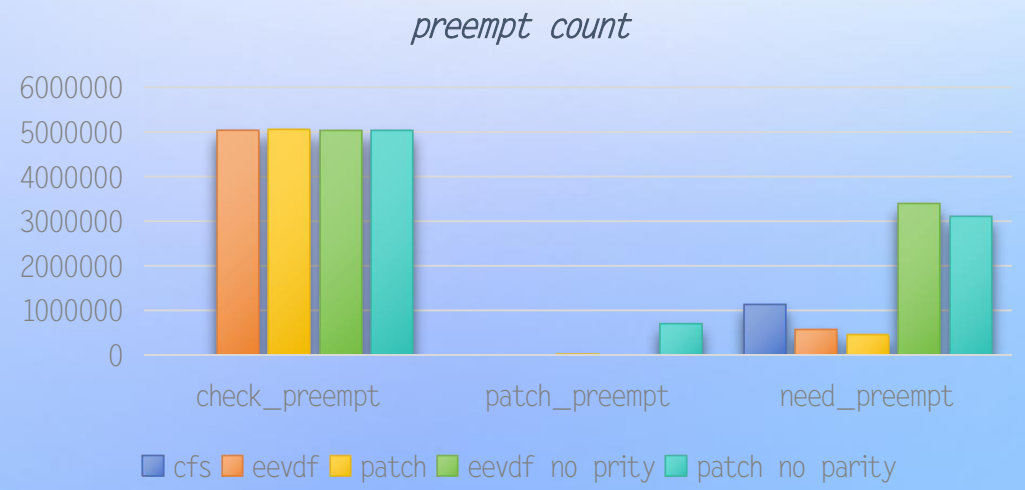


调度时延优化

v2 版本对比数据

n 抢占次数对比

schedstats	CFS	EEVDF	PATCH	EEVDF no parity	PATCH no parity
check_preempt_count	–	5053054	5045388	5018589	5029585
patch_preempt_count	–	–	0020495	–	0700670
need_preempt_count	1140821	0570520	0458947	3380513	3116966



调度时延优化

最终优化与peter的一组patch共同合入v6.12版本

- 合入版本差异
  - 单独添加了一个特性控制这种短时抢占 *sched_feat(PREEMPT_SHORT)*
  - 在wakeup阶段与 *custom_slice* 特性相结合，只有当唤醒 se->slice 小于当前 se->slice 时才可以短时抢占。
- 其中的权衡
  - 社区对抢占的引入是非常谨慎的，更倾向于用户提供灵活的控制从而获得不同的性能倾向。

04

思考与展望

Reflections and Outlook

思考与展望

带宽与时延是调度子系统中两个关键指标，二者关系像是翘翘板的两端，根据不同场景寻找最合适的平衡点。

不同场景下 eevdf 的配置组合

- 对实时性要求高的场景
  - 去使能 *sched_feat(RUN_TO_PARITY)* 特性
  - 但会带来更多的上下文切换
- 相对均衡的场景
  - 使能 *sched_feat(RUN_TO_PARITY)* 特性
  - 使能 *sched_feat(PREEMPT_SHORT)* 特性
- 高吞吐场景
  - 使能 *sched_feat(RUN_TO_PARITY)* 特性
  - 去使能 *sched_feat(PREEMPT_SHORT)* 特性

思考与展望

eevdf 可能优化的方向

- *wakeup* 抢占识别这一块
  - 当前 *sched_feat(PREEMPT_SHORT)* 判断逻辑的粒度有点粗
  - 各家可以针对各自的场景进行定制优化
    - 结合 ebpf 技术可以更灵活使用

- *load balance* 中可能会有更好的处理方式
  - 当前 LB 的逻辑是根据 CPU 上的 load 进行控制
  - 或许可以结合任务对时延的敏感程度进行 LB

思考与展望

展望

调度器作为对应用场景最为敏感的子模块之一，其发展路径正呈现出丰富多元的态势。不同厂商、不同场景对调度器的需求各不相同——手机厂商更关注功耗与实时性，新能源、AI、机器人等领域同样强调极致的实时响应，而在互联网与云服务场景中，则更注重吞吐量与实时性之间的精巧平衡。

随着 sched_ext 新特性的涌现，调度器的设计与演进迎来了前所未有的想象空间。可以预见，未来的调度器领域必将迎来一个百花齐放、创新迸发的时代。

同时也期待，在各场景的持续探索中，能够不断沉淀出通用性强、可复用的方法与框架，持续赋能通用调度器的演进与壮大。让我们携手并进，共同推动通用调度器的成熟与完善，为计算技术的未来奠定更坚实的基础。

祝 Linux Kernel 发展越来越好，祝 China 的 Linux Kernel 发展越来越好！

THANKS