**CFS调度算法分析**

计科1201 唐佳娣 1212300112

CFS（completely fair schedule）是最终被内核采纳的调度器。它从RSDL/SD中吸取了完全公平的思想，不再跟踪进程的睡眠时间，也不再企图区分交互式进程。它将所有的进程都统一对待，这就是公平的含义。CFS的算法和实现都相当简单，众多的测试表明其性能也非常优越。

CFS 背后的主要想法是维护为任务提供处理器时间方面的平衡（公平性）。这意味着应给进程分配相当数量的处理器。分给某个任务的时间失去平衡时（意味着一个或多个任务相对于其他任务而言未被给予相当数量的时间），应给失去平衡的任务分配时间，让其执行。

按照作者Ingo Molnar的说法："CFS百分之八十的工作可以用一句话概括：CFS在真实的硬件上模拟了完全理想的多任务处理器"。在“完全理想的多任务处理器”下，每个进程都能同时获得CPU的执行时间。当系统中有两个进程时，CPU的计算时间被分成两份，每个进程获得50%。然而在实际的硬件上，当一个进程占用CPU时，其它进程就必须等待。这就产生了不公平。

其中涉及到几个重要的结构体：

1.task\_struct ：

struct task\_struct {

...

int prio, static\_prio, normal\_prio; 进程的优先级

unsigned int rt\_priority;实时进程的优先级

const struct sched\_class \*sched\_class; 调度类，一堆函数指针，实现调度

struct sched\_entity se;调度实体 一个进程对应一个调度实体，，

struct sched\_rt\_entity rt;

....

}

2．sched\_class：（调度相关的函数指针）

struct sched\_class {

...

void (\*enqueue\_task) (struct rq \*rq, struct task\_struct \*p, int wakeup); 入列

void (\*dequeue\_task) (struct rq \*rq, struct task\_struct \*p, int sleep);出列

void (\*yield\_task) (struct rq \*rq);

void (\*check\_preempt\_curr) (struct rq \*rq, struct task\_struct \*p, int sync);检查当前进程可否被新进程抢占

struct task\_struct \* (\*pick\_next\_task) (struct rq \*rq); 选择下一个进程运行

void (\*put\_prev\_task) (struct rq \*rq, struct task\_struct \*p);

#ifdef CONFIG\_SMP

int (\*select\_task\_rq)(struct task\_struct \*p, int sync);

....

#ifdef CONFIG\_FAIR\_GROUP\_SCHED 跟组调度相关的宏void (\*moved\_group) (struct task\_struct \*p);

#endif

};

3.调度实体

struct sched\_entity {

struct load\_weight load; /\* for loadbalancing\*/ nice对应的load值

struct rb\_node run\_node; 红黑树结点

struct list\_head group\_node;

unsigned int on\_rq;

u64 exec\_start; 上次开始调度时的运行时间

u64 sum\_exec\_runtime; 总运行时间

u64 vruntime;

u64 prev\_sum\_exec\_runtime; 上次调度总运行时间

...

#ifdef CONFIG\_FAIR\_GROUP\_SCHED 如果是组调度的话，就多了些部分。

struct sched\_entity \*parent;

/\* rq on which this entity is (to be) queued: \*/

struct cfs\_rq \*cfs\_rq;

/\* rq "owned" by this entity/group: \*/

struct cfs\_rq \*my\_q;

#endif

}

4.cfs 运行队列

/\* CFS-related fields in a runqueue \*/

struct cfs\_rq {

struct load\_weight load;

unsigned long nr\_running;

u64 exec\_clock;

u64 min\_vruntime;

5.运行队列

struct rq {

...

unsigned long nr\_running;

#define CPU\_LOAD\_IDX\_MAX 5

unsigned long cpu\_load[CPU\_LOAD\_IDX\_MAX];

...

struct cfs\_rq cfs;

...

struct task\_struct \*curr, \*idle;

}

6.调度相关类

static const struct sched\_class fair\_sched\_class = {

.next = &idle\_sched\_class,

.enqueue\_task = enqueue\_task\_fair,

.dequeue\_task = dequeue\_task\_fair,

.yield\_task = yield\_task\_fair,

.check\_preempt\_curr = check\_preempt\_wakeup,

.pick\_next\_task = pick\_next\_task\_fair,

.put\_prev\_task = put\_prev\_task\_fair,

#ifdef CONFIG\_SMP

.select\_task\_rq = select\_task\_rq\_fair,

.load\_balance = load\_balance\_fair,

.move\_one\_task = move\_one\_task\_fair,

#endif

.set\_curr\_task = set\_curr\_task\_fair,

.task\_tick = task\_tick\_fair,

.task\_new = task\_new\_fair,

.prio\_changed = prio\_changed\_fair,

.switched\_to = switched\_to\_fair,

#ifdef CONFIG\_FAIR\_GROUP\_SCHED

.moved\_group = moved\_group\_fair,

#endif

};

以下将对内核中CFS相关的部分代码进行分析。

1.wake\_up\_new\_task

void wake\_up\_new\_task(struct task\_struct \*p, unsigned long clone\_flags)

{

unsigned long flags;

struct rq \*rq;

rq = task\_rq\_lock(p, &flags);顺序操作运行队列

BUG\_ON(p->state != TASK\_RUNNING);

update\_rq\_clock(rq);

p->prio = effective\_prio(p); 计算priority,普通进程的priority就是static priority

if (!p->sched\_class->task\_new || !current->se.on\_rq) {如果条件不满足，直接入列，但请注意active\_task()的最后参数0，不唤醒

activate\_task(rq, p, 0);

} else {

p->sched\_class->task\_new(rq, p);调用完全公平类里面的task\_new做些初始化的操作。

inc\_nr\_running(rq);增加运行队列的运行进程数目

}

trace\_sched\_wakeup\_new(rq, p);

check\_preempt\_curr(rq, p, 0);

#ifdef CONFIG\_SMP

if (p->sched\_class->task\_wake\_up)

p->sched\_class->task\_wake\_up(rq, p);

#endif

task\_rq\_unlock(rq, &flags);

}

2.task\_new

static void task\_new\_fair(struct rq \*rq, struct task\_struct \*p)

{

struct cfs\_rq \*cfs\_rq = task\_cfs\_rq(p);

struct sched\_entity \*se = &p->se, \*curr = cfs\_rq->curr;

int this\_cpu = smp\_processor\_id();

sched\_info\_queued(p);

update\_curr(cfs\_rq); 更新cfs的一些信息

place\_entity(cfs\_rq, se, 1);初始化se在cfs里的信息，包括vruntime

if (sysctl\_sched\_child\_runs\_first && this\_cpu == task\_cpu(p) &&curr && curr->vruntime < se->vruntime) {

swap(curr->vruntime, se->vruntime);

resched\_task(rq->curr);

}

enqueue\_task\_fair(rq, p, 0); 放进队列里面

}

3.update\_curr

static void update\_curr(struct cfs\_rq \*cfs\_rq)

{

struct sched\_entity \*curr = cfs\_rq->curr;

u64 now = rq\_of(cfs\_rq)->clock;

unsigned long delta\_exec;

if (unlikely(!curr))

return;

delta\_exec = (unsigned long)(now curr->exec\_start);计算自上次调度此进程到这次又调度，经过的时间

\_\_update\_curr(cfs\_rq, curr, delta\_exec); 把刚计算出来的运行时间作为参数传进去,下面详解

curr->exec\_start = now;更新完了立刻打个时间戳。

if (entity\_is\_task(curr)) {

struct task\_struct \*curtask = task\_of(curr);

cpuacct\_charge(curtask, delta\_exec);

account\_group\_exec\_runtime(curtask, delta\_exec);

}

}

4.\_\_update\_curr

static inline void \_\_update\_curr(struct cfs\_rq \*cfs\_rq, struct sched\_entity \*curr,unsigned long delta\_exec) {

unsigned long delta\_exec\_weighted;

schedstat\_set(curr->exec\_max, max((u64)delta\_exec, curr->exec\_max));

curr->sum\_exec\_runtime += delta\_exec; 直接加到总运行时间变量里去，

schedstat\_add(cfs\_rq, exec\_clock, delta\_exec);

delta\_exec\_weighted = calc\_delta\_fair(delta\_exec, curr); 详解见下

curr->vruntime += delta\_exec\_weighted; 把上步计算出来的值加到了vruntime里

update\_min\_vruntime(cfs\_rq);

}

5. calc\_delta\_fair

static inline unsigned long calc\_delta\_fair(unsigned long delta, struct sched\_entity \*se){

if (unlikely(se->load.weight != NICE\_0\_LOAD)) 如果进程实体有默认的load值，直接返回delta

delta = calc\_delta\_mine(delta, NICE\_0\_LOAD, &se->load);计算应该修正的这个值

return delta;

}

在往下走之前，先弄明白nice 和 se->load之间的关系吧。

se->load是指调度实体的负载。同理一个cfs\_rq也有负载。一般是所有task的load之和。这个load是通过nice与 一个静态数组转换来的。一般普通进程的nice值在20~19之间，其对应load.weight值是递减的。而 NICE\_0\_LOAD就是nice=0的对应的load值。

6.place\_entity

static void place\_entity(struct cfs\_rq \*cfs\_rq, struct sched\_entity \*se, int initial){

u64 vruntime = cfs\_rq->min\_vruntime;

if (initial && sched\_feat(START\_DEBIT))

vruntime += sched\_vslice(cfs\_rq, se);

if (!initial) {

if (sched\_feat(NEW\_FAIR\_SLEEPERS)) {

unsigned long thresh = sysctl\_sched\_latency;

if (sched\_feat(NORMALIZED\_SLEEPER))

thresh = calc\_delta\_fair(thresh, se);

vruntime =thresh;

}

/\* ensure we never gain time by being placed backwards. \*/

vruntime = max\_vruntime(se->vruntime, vruntime);

}

se->vruntime = vruntime;

}

7. enqueue\_task\_fair

static void enqueue\_task\_fair(struct rq \*rq, struct task\_struct \*p, int wakeup){

struct cfs\_rq \*cfs\_rq;

struct sched\_entity \*se = &p->se;

for\_each\_sched\_entity(se) {遍历所有的se及它的父亲。。。这个在非组调度时，就是se,组调度时，会将其dad一同入列

if (se->on\_rq)如果已经在运行队列，就停止

break;

cfs\_rq = cfs\_rq\_of(se);

enqueue\_entity(cfs\_rq, se, wakeup);真正的入队函数，把se插入到rbtree.

wakeup = 1;

}

hrtick\_update(rq);

}

8. enqueue\_entity() 会更新时间，最终调──enqueue\_entity()，把schedule\_entity往红黑树中存放。每个结点的值是通过 entity\_key()来实现的。