## 1、调度策略：

1. /\*
2. \* Scheduling policies
3. \*/
4. #define SCHED\_NORMAL        0   /\*CFS\*/
5. #define SCHED\_FIFO      1   /\*RT\*/
6. #define SCHED\_RR        2   /\*RT\*/
7. #define SCHED\_BATCH     3   /\*CFS\*/
8. /\* SCHED\_ISO: reserved but not implemented yet \*/
9. #define SCHED\_IDLE      5   /\*CFS-IDLE\*/
10. #define SCHED\_DEADLINE      6   /\*DL\*/

SCHED\_NORMAL：普通的分时进程，使用CFS调度器

SCHED\_FIFO：先进先出的实时进程。当调用程序把CPU分配给进程的时候，它把该进程描述符保留在运行队列链表的当前位置。此调度策略的进程一旦使用CPU则一直运行。如果没有其他可运行的更高优先级实时进程，进程就继续使用CPU，想用多久就用多久，即使还有其他具有相同优先级的实时进程处于可运行状态。

SCHED\_RR：时间片轮转的实时进程。当调度程序把CPU分配给进程的时候，它把该进程的描述符放在运行队列链表的末尾。这种策略保证对所有具有相同优先级的SCHED\_RR实时进程进行公平分配CPU时间

SCHED\_BATCH：是SCHED\_NORMAL的分化版本。采用分时策略，根据动态优先级，分配CPU资源。在有实时进程的时候，实时进程优先调度。但针对吞吐量优化，除了**不能抢占外**与常规进程一样，允许任务运行更长时间，更好使用高速缓存，适合于成批处理的工作

SCHED\_IDLE：优先级最低，在系统空闲时运行

SCHED\_DEADLINE：新支持的实时进程调度策略，针对突发型计算，并且对延迟和完成时间敏感的任务使用，基于EDF（earliest deadline first）

处于TASK\_RUNNING状态的进程才会被进程调度器选择，其他状态不会进入调度器。系统发生调度的时机如下：

🡪调用cond\_resched()时

🡪显式调用schedule()时

🡪从中断上下文返回时

当内核开启抢占时，会多出几个调度时机如下：

🡪在系统调用或者中断上下文中调用preemt\_enable()时（多次调用系统只会在最后一次调用时会调度）

🡪在中断上下文中，从中断处理函数返回到可抢占的上下文时

## 2、调度类

|  |  |  |  |
| --- | --- | --- | --- |
| **调度器类** | **描述** | **调度策略** | **文件** |
| fair\_sched\_class | 采用CFS算法调度普通的非实时进程 | SCHED\_NORMAL, SCHED\_BATCH | Fair.c |
| rt\_sched\_class | 采用提供Roound-Robin算法或者FIFO算法调度实时进程，具体调度策略由进程task\_struct->policy指定 | SCHED\_FIFO, SCHED\_RR | Rt.c |
| idle\_sched\_class | 采用CFS算法调用idle进程 | SCHED\_IDLE | Idle\_task.c |
| dl\_sched\_class | 采用EDF最早截至时间优先算法调度实时进程 | SCHED\_DEADLINE | Deadline.c |
| stop\_sched\_class | 优先级最高的线程，会中断所有其他线程，且不会被其他任务打断作用 | 无 | Stop\_task.c |
|  |  |  |  |
|  |  |  |  |

1. **struct** sched\_class {
2. /\*系统中有多个调度类，按照优先级排成一个链表
3. \*优先级顺序\*stop\_sched\_class->dl\_sched\_class->rt\_sched\_class->fair\_sched\_class->idle\_sc\*hed\_class
4. \*/
5. **const** **struct** sched\_class \*next;
6. /\*将进程加入到运行队列中，即将调度实体放入红黑树中，并对nr\_running变量+1\*/
7. **void** (\*enqueue\_task) (**struct** rq \*rq, **struct** task\_struct \*p, **int** flags);
8. /\*从运行队列中删除进程，并对nr\_running变量-1\*/
9. **void** (\*dequeue\_task) (**struct** rq \*rq, **struct** task\_struct \*p, **int** flags);
10. /\*放弃CPU,在compat\_yield sysctl关闭的情况下，该函数实际上执行先出队后入队，在这种情况下他将调度实体放在红黑树最右\*/
11. **void** (\*yield\_task) (**struct** rq \*rq);
12. **bool** (\*yield\_to\_task) (**struct** rq \*rq, **struct** task\_struct \*p, **bool** preempt);
13. /\*检查当前进程是否可被进程抢占\*/
14. **void** (\*check\_preempt\_curr) (**struct** rq \*rq, **struct** task\_struct \*p, **int** flags);
15. /\*选择下一个应该要运行的进程\*/
16. **struct** task\_struct \* (\*pick\_next\_task) (**struct** rq \*rq);
17. /\*将进程放回运行队列\*/
18. **void** (\*put\_prev\_task) (**struct** rq \*rq, **struct** task\_struct \*p);
20. #ifdef CONFIG\_SMP
21. /\*为进程选择一个合适的CPU\*/
22. **int**  (\*select\_task\_rq)(**struct** task\_struct \*p, **int** task\_cpu, **int** sd\_flag, **int** flags);
23. /\*迁移任务到另一个CPU\*/
24. **void** (\*migrate\_task\_rq)(**struct** task\_struct \*p, **int** next\_cpu);
26. **void** (\*pre\_schedule) (**struct** rq \*this\_rq, **struct** task\_struct \*task);
27. **void** (\*post\_schedule) (**struct** rq \*this\_rq);
28. /\*用于进程唤醒\*/
29. **void** (\*task\_waking) (**struct** task\_struct \*task);
30. **void** (\*task\_woken) (**struct** rq \*this\_rq, **struct** task\_struct \*task);
31. /\*修改进程的CPU亲和力（affinity）\*/
32. **void** (\*set\_cpus\_allowed)(**struct** task\_struct \*p,
33. **const** **struct** cpumask \*newmask);
34. /\*启动运行队列\*/
35. **void** (\*rq\_online)(**struct** rq \*rq);
36. /\*禁止运行队列\*/
37. **void** (\*rq\_offline)(**struct** rq \*rq);
38. #endif
39. /\*当进程改变它的调度类或者进程组时被调用\*/
40. **void** (\*set\_curr\_task) (**struct** rq \*rq);
41. /\*该函数通常调用自time tick函数，它可能引起进程切换，这将驱动运行时running抢占\*/
42. **void** (\*task\_tick) (**struct** rq \*rq, **struct** task\_struct \*p, **int** queued);
43. /\*在进程创建时调用，不同调度策略的进程初始化不一样\*/
44. **void** (\*task\_fork) (**struct** task\_struct \*p);
45. /\*在进程退出时会使用\*/
46. **void** (\*task\_dead) (**struct** task\_struct \*p);
47. /\*用于进程切换\*/
48. **void** (\*switched\_from) (**struct** rq \*this\_rq, **struct** task\_struct \*task);
49. **void** (\*switched\_to) (**struct** rq \*this\_rq, **struct** task\_struct \*task);
50. /\*改变优先级\*/
51. **void** (\*prio\_changed) (**struct** rq \*this\_rq, **struct** task\_struct \*task,
52. **int** oldprio);
54. unsigned **int** (\*get\_rr\_interval) (**struct** rq \*rq,  **struct** task\_struct \*task);
56. #ifdef CONFIG\_FAIR\_GROUP\_SCHED
57. **void** (\*task\_move\_group) (**struct** task\_struct \*p, **int** on\_rq);
58. #endif
59. };

## 3、调度实体

|  |  |  |
| --- | --- | --- |
| **调度实体** | **名称** | **描述** |
| sched\_dl\_entity | DEADLINE调度实体 | 采用EDF算法的调用实体 |
| sched\_rt\_entiy | RT调度实体 | 采用Roound-Robin或者FIFO算法的调度实体 |
| sched\_entity | CFS调度实体 | 采用CFS算法的普通非实时进程的调度实体 |
|  |  |  |

1. /\*
2. 一个调度实体（红黑树的一个节点），其包含一组或一个指定的进程，包含一个自己的运行队列，一个父亲指针，一个指向需要调度的队列
3. \*/
4. **struct** sched\_entity {
5. /\*权重，在数组prio\_to\_weight[]包含优先级转权重的数值\*/
6. **struct** load\_weight  load;       /\* for load-balancing \*/
7. /\*实体在红黑树对应的节点信息\*/
8. **struct** rb\_node      run\_node;
9. /\*实体所在的进程组\*/
10. **struct** list\_head    group\_node;
11. /\*实体是否处于红黑树运行队列中\*/
12. unsigned **int**        on\_rq;
14. /\*开始运行时间\*/
15. u64         exec\_start;
16. /\*总运行时间\*/
17. u64         sum\_exec\_runtime;
18. /\*
19. 虚拟运行时间，在时间中断或者任务状态发生改变时会更新
20. 其会不停的增长，增长速度与load权重成反比，load越高，增长速度越慢，就越可能处于红黑树最左边被调度
21. 每次时钟中断都会修改其值
22. 具体见calc\_delta\_fair()函数
23. \*/
24. u64         vruntime;
25. /\*进程在切换进cpu时的sum\_exec\_runtime值\*/
26. u64         prev\_sum\_exec\_runtime;
27. /\*此调度实体中进程移到其他cpu组的数量\*/
28. u64         nr\_migrations;
30. #ifdef CONFIG\_SCHEDSTATS
31. /\*用于统计一些数据\*/
32. **struct** sched\_statistics statistics;
33. #endif
35. #ifdef CONFIG\_FAIR\_GROUP\_SCHED
36. /\*
37. 父亲调度实体指针，如果是进程则指向其运行队列的调度实体，如果是进程组则指向其上一个进程组的调度实体
38. 在set\_task\_rq函数中设置
39. \*/
40. **struct** sched\_entity \*parent;
41. /\* rq on which this entity is (to be) queued: \*/
42. /\*实体所处红黑树运行队列\*/
43. **struct** cfs\_rq       \*cfs\_rq;
44. /\* rq "owned" by this entity/group: \*/
45. /\*实体的红黑树运行队列，如果为NULL表明其是一个进程，若非NULL表明其是调度组\*/
46. **struct** cfs\_rq       \*my\_q;
47. #endif
49. #ifdef CONFIG\_SMP
50. /\* Per-entity load-tracking \*/
51. **struct** sched\_avg    avg;
52. #endif
53. };
55. **struct** sched\_rt\_entity {
56. **struct** list\_head run\_list;
57. unsigned **long** timeout;
58. unsigned **long** watchdog\_stamp;
59. unsigned **int** time\_slice;
61. **struct** sched\_rt\_entity \*back;
62. #ifdef CONFIG\_RT\_GROUP\_SCHED
63. **struct** sched\_rt\_entity  \*parent;
64. /\* rq on which this entity is (to be) queued: \*/
65. **struct** rt\_rq        \*rt\_rq;
66. /\* rq "owned" by this entity/group: \*/
67. **struct** rt\_rq        \*my\_q;
68. #endif
69. };
71. **struct** sched\_dl\_entity {
72. **struct** rb\_node  rb\_node;
74. /\*
75. \* Original scheduling parameters. Copied here from sched\_attr
76. \* during sched\_setscheduler2(), they will remain the same until
77. \* the next sched\_setscheduler2().
78. \*/
79. u64 dl\_runtime;     /\* maximum runtime for each instance    \*/
80. u64 dl\_deadline;    /\* relative deadline of each instance   \*/
81. u64 dl\_period;      /\* separation of two instances (period) \*/
82. u64 dl\_bw;      /\* dl\_runtime / dl\_deadline     \*/
84. /\*
85. \* Actual scheduling parameters. Initialized with the values above,
86. \* they are continously updated during task execution. Note that
87. \* the remaining runtime could be < 0 in case we are in overrun.
88. \*/
89. s64 runtime;        /\* remaining runtime for this instance  \*/
90. u64 deadline;       /\* absolute deadline for this instance  \*/
91. unsigned **int** flags; /\* specifying the scheduler behaviour   \*/
93. /\*
94. \* Some bool flags:
95. \*
96. \* @dl\_throttled tells if we exhausted the runtime. If so, the
97. \* task has to wait for a replenishment to be performed at the
98. \* next firing of dl\_timer.
99. \*
100. \* @dl\_new tells if a new instance arrived. If so we must
101. \* start executing it with full runtime and reset its absolute
102. \* deadline;
103. \*
104. \* @dl\_boosted tells if we are boosted due to DI. If so we are
105. \* outside bandwidth enforcement mechanism (but only until we
106. \* exit the critical section).
107. \*/
108. **int** dl\_throttled, dl\_new, dl\_boosted;
110. /\*
111. \* Bandwidth enforcement timer. Each -deadline task has its
112. \* own bandwidth to be enforced, thus we need one timer per task.
113. \*/
114. **struct** hrtimer dl\_timer;
115. };

## 4、CFS

CFS不再跟踪进程的睡眠时间，也不再企图区分交互式进程。它将所有的进程公平对待。

例如：当系统又两个进程，CPU的计算时间被分为2份，每个进程可获取50%。当前进程运行了10ms，在完全理想的多任务处理器中，10ms应该平分个两个进程，但是它却运行了10ms，所以CFS将惩罚当前进程，使其它进程能够在下次调度时尽可能取代当前进程。最终实现进程的公平调度。

CFS对每个CPU维护一个以时间为顺序的红黑树。原因在于：

🡪红黑树可以始终保持平衡，意味着树上没有路径比任何其他路径长两倍以上

🡪红黑树是二叉树，查找时间复杂度为O(log n)。但是除了最左侧查找以外，很难执行其他查找，并且最左侧的节点指针始终被缓存

🡪红黑树可通过内部存储实现，不需要使用外部分配，即可对数据结构进行维护。

1、实现pick next

所有可运行的任务通过不断地操作最终都存储在以时间为顺序的红黑树中（由sched\_entity）对象表示，对处理器需求最多的任务存储在树的左侧，需求最少的任务存储在树的右侧。为了公平，CFS调度器会选择红黑树最左边的叶子节点作为下一个将获得cpu的任务。这样树左侧的进程就被给予时间运行。

2、tick中断

tick中断首先更新调度信息。然后调整当前进程在红黑树中的位置。调整完成以后如果当前进程不再是最左边的叶子，就标记为Need\_resched标志，中断返回时就会调用scheduler()完成切换、否则当前进程继续占用CPU。从这里可以看出CFS抛弃了传统时间片概念。Tick中断只需要更新红黑树。

1. 红黑树键值计算

红黑树键值由三个因子计算而得：进程已经占用cpu时间、当前进程的nice值、当前cpu负载。其实很大程度上理解CFS可以简单认为键值就等于进程已占用cpu时间。因此占用时间越大，键值越大，从而使得当前进程向红黑树的右侧移动。另外CFS规定，nice值为1的进程为Nice值为0的进程多获取10%的cpu时间。计算键值也考虑这个因素。Nice值越大，键值越大。

1. 调度器管理器

为了支持实时进程，CFS提供了调度器模块管理器。各种不同的调度器算法都可以作为一个模块注册到该管理器中。不同的进程可以选择使用不同的调度模块。

1. CFS组调度

|  |  |
| --- | --- |
| **字段** | **描述** |
| load | 指定了权重, 决定了各个实体占队列总负荷的比重, 计算负荷权重是调度器的一项重任, 因为CFS所需的虚拟时钟的速度最终依赖于负荷, 权重通过优先级转换而成，是vruntime计算的关键 |
| Run\_node | 调度实体在红黑树对应的结点信息, 使得调度实体可以在红黑树上排序 |
| Sum\_exec\_runtime | 记录程序运行所消耗的CPU时间, 以用于完全公平调度器CFS |
| On\_rq | 调度实体是否在就绪队列上接受检查, 表明是否处于CFS红黑树运行队列中，需要明确一个观点就是，CFS运行队列里面包含有一个红黑树，但这个红黑树并不是CFS运行队列的全部，因为红黑树仅仅是用于选择出下一个调度程序的算法。很简单的一个例子，普通程序运行时，其并不在红黑树中，但是还是处于CFS运行队列中，其on\_rq为真。只有准备退出、即将睡眠等待和转为实时进程的进程其CFS运行队列的on\_rq为假 |
| vruntime | 虚拟运行时间，调度的关键，其计算公式：一次调度间隔的虚拟运行时间 = 实际运行时间 \* (NICE\_0\_LOAD / 权重)。可以看出跟实际运行时间和权重有关，红黑树就是以此作为排序的标准，优先级越高的进程在运行时其vruntime增长的越慢，其可运行时间相对就长，而且也越有可能处于红黑树的最左结点，调度器每次都选择最左边的结点为下一个调度进程。注意其值为单调递增，在每个调度器的时钟中断时当前进程的虚拟运行时间都会累加。单纯的说就是进程们都在比谁的vruntime最小，最小的将被调度 |
| Cfs\_rq | 此调度实体所处于的CFS运行队列 |
| My\_q | 如果此调度实体代表的是一个进程组，那么此调度实体就包含有一个自己的CFS运行队列，其CFS运行队列中存放的是此进程组中的进程，这些进程就不会在其他CFS运行队列的红黑树中被包含(包括顶层红黑树也不会包含他们，他们只属于这个进程组的红黑树) |

**Sum\_exec\_runtime：**

* + 跟踪运行时间是由update\_curr不断累积完成的. 内核中许多地方都会调用该函数, 例如, 新进程加入就绪队列时, 或者周期性调度器中. 每次调用时, 会计算当前时间和exec\_start之间的差值, exec\_start则更新到当前时间. 差值则被加到sum\_exec\_runtime.
  + 在进程执行期间虚拟时钟上流逝的时间数量由vruntime统计
  + 在进程被撤销时, 其当前sum\_exec\_runtime值保存到prev\_sum\_exec\_runtime, 此后, 进程抢占的时候需要用到该数据, 但是注意, 在prev\_sum\_exec\_runtime中保存了sum\_exec\_runtime的值, 而sum\_exec\_runtime并不会被重置, 而是持续单调增长

每个进程task\_struct中都嵌入了sched\_entity对象，所以进程是可调度的实体，但是请注意，可调度的实体不一定是进程，也可能是进程组

## 5、调度函数

**Scheduler\_tick**

**主要两个任务：**

1. 更新相关统计量

管理内核中与整个系统和各个进程的调度相关的统计量，其间执行的主要操作是对各种统计器+1

1. 激活负载当前进程调度类的周期性调度方法

检查进程执行的时间是否超过了它对应的ideal\_runtime,如果超过了，则告诉系统，需要启动主调度器(schedule)进行进程切换。

1. **void** scheduler\_tick(**void**)
2. {
3. /\*获取当前cpu号\*/
4. **int** cpu = smp\_processor\_id();
5. /\*取得当前cpu运行队列rq\*/
6. **struct** rq \*rq = cpu\_rq(cpu);
7. /\*获取就绪队列上正在运行的进程\*/
8. **struct** task\_struct \*curr = rq->curr;
10. sched\_clock\_tick();
12. raw\_spin\_lock(&rq->lock); //运行队列上锁
13. update\_rq\_clock(rq); //更新rq的当前时间戳，即rq->clock变为当前时间戳
14. curr->sched\_class->task\_tick(rq, curr, 0); //执行当前运行进程所在调度类的task\_tick回调函数进程周期性调度
15. update\_cpu\_load\_active(rq); //更新rq中的cpu\_load，本质上是将数组中存储的负载值向后移动一个位置，将当前负载值计入数组的第一个位置
16. raw\_spin\_unlock(&rq->lock);//运行队列解锁
18. perf\_event\_task\_tick();
20. #ifdef CONFIG\_SMP
21. rq->idle\_balance = idle\_cpu(cpu);
22. trigger\_load\_balance(rq); //如果下一次负载平衡到则触发SCHED\_SOFTIRQ
23. #endif
24. rq\_last\_tick\_reset(rq);
25. }

**\_\_schedule**

**主要步骤：**

1. 禁止内核抢占
2. 获取当前CPU的rq
3. 取消当前进程运行的高精度计数器(hrtimer)
4. 获取rq锁
5. 更新队列时钟
6. 清楚当前进程的need\_resched标志
7. 如果当前进程为非运行状态，并且当前非内核抢占：
   1. 如果当前进程有信号待处理，设置当前进程为就绪态
   2. 否则，该进程出rq
8. 如果当前rq没有可运行进程，则通过load balance从其他cpu搬进程
9. 通知调度类用另外一个进程代替当前进程
10. 通知调度类选择下一个可运行进程
11. 如果下一个可运行进程非当前进程，则执行进程切换
12. 否则释放队列锁
13. 开启内核抢占
14. 判断当前进程是否被设置need\_resched，是则重复1
15. **static** **void** \_\_sched \_\_schedule(**void**)
16. {
17. **struct** task\_struct \*prev, \*next;
18. unsigned **long** \*switch\_count;
19. **struct** rq \*rq;
20. **int** cpu;
22. need\_resched:
23. preempt\_disable();
24. /\*获取当前CPU\*/
25. cpu = smp\_processor\_id();
26. /\*取得当前cpu的运行队列rq\*/
27. rq = cpu\_rq(cpu);
28. /\*更新全局状态，标识当前CPU发生上下文切换\*/
29. rcu\_note\_context\_switch(cpu);
30. prev = rq->curr;  /\*当前运行的进程存入Prev中\*/
32. schedule\_debug(prev);
34. **if** (sched\_feat(HRTICK)) //取消为当前进程运行的hrtimer
35. hrtick\_clear(rq);
37. /\*
38. \* Make sure that signal\_pending\_state()->signal\_pending() below
39. \* can't be reordered with \_\_set\_current\_state(TASK\_INTERRUPTIBLE)
40. \* done by the caller to avoid the race with signal\_wake\_up().
41. \*/
42. smp\_mb\_\_before\_spinlock(); //内存屏障
43. raw\_spin\_lock\_irq(&rq->lock); //上锁该队列
45. switch\_count = &prev->nivcsw; //记录当前进程切换次数
46. **if** (prev->state && !(preempt\_count() & PREEMPT\_ACTIVE)) { //当前进程非运行状态，并且非内核抢占
47. **if** (unlikely(signal\_pending\_state(prev->state, prev))) { //当前进程有信号待处理，设置进程为就绪态
48. prev->state = TASK\_RUNNING; //设置为RUNNING
49. } **else** {
50. deactivate\_task(rq, prev, DEQUEUE\_SLEEP); //将其从队列中删除
51. prev->on\_rq = 0;
53. /\*
54. \* If a worker went to sleep, notify and ask workqueue
55. \* whether it wants to wake up a task to maintain
56. \* concurrency.
57. \*/
58. **if** (prev->flags & PF\_WQ\_WORKER) {
59. **struct** task\_struct \*to\_wakeup;
61. to\_wakeup = wq\_worker\_sleeping(prev, cpu);
62. **if** (to\_wakeup)
63. try\_to\_wake\_up\_local(to\_wakeup);
64. }
65. }
66. switch\_count = &prev->nvcsw;
67. }
69. pre\_schedule(rq, prev); //通知调度器，即将发生进程切换
71. **if** (unlikely(!rq->nr\_running))
72. idle\_balance(cpu, rq);
74. put\_prev\_task(rq, prev); //通知调度器，即将用另一个进程替换当前进程
75. next = **pick\_next\_task**(rq); //挑选一个优先级高的任务，使用调度类的方法
76. clear\_tsk\_need\_resched(prev); //清空TIF\_NEED\_RESCHED标志
77. clear\_preempt\_need\_resched(); //清空PREEMPT\_NEED\_RESCHED标志
78. rq->skip\_clock\_update = 0;
80. **if** (likely(prev != next)) { //如果如果选出的进程和当前进程不是同一个进程
81. rq->nr\_switches++; //队列切换次数更新
82. rq->curr = next; //将当前进程换成刚才挑选的进程
83. ++\*switch\_count; //进程切换次数更新
85. context\_switch(rq, prev, next); /\* unlocks the rq \*/ /\*进程上下文切换\*/
86. /\*
87. \* The context switch have flipped the stack from under us
88. \* and restored the local variables which were saved when
89. \* this task called schedule() in the past. prev == current
90. \* is still correct, but it can be moved to another cpu/rq.
91. \*/
92. cpu = smp\_processor\_id();
93. rq = cpu\_rq(cpu);
94. } **else**
95. raw\_spin\_unlock\_irq(&rq->lock);
97. post\_schedule(rq); //通知调度器，完成了进程切换
99. sched\_preempt\_enable\_no\_resched(); //开启内核抢占
100. **if** (need\_resched()) //如果该进程被其他进程设置了TIF\_NEED\_RESCHED,则需要重新调度
101. **goto** need\_resched;
102. }