Very clear

<http://www.cnblogs.com/wwwjieo0/archive/2014/05/21/3743251.html>

Linux System Programming 学习笔记(六) 进程调度

2. 时间片

如果时间片过大，那么 挂起进程 开始执行前的等待时间过长，这将减小 并发执行的粒度，甚至用户觉察到延迟

如果时间片过小，那么 系统在进程之间切换的时间花销将很大，时间局部性的优势将丧失

CPU密集型：processes are hungry for CPU time，比如科学计算、数学计算、图像处理

I/O密集型：spend more time blocked waiting for some resource than executing，often issuing and waiting for file or network I/O, blocking on keyboard input

3. CFS调度

传统Unix进程调度中，有两个最基础的变量：**优先级**和**时间片**

**CFS引进 fair scheduling 算法，CFS赋予每个进程一定比例的处理器时间，而不是时间片.** 开始时，CFS赋予N个进程等同的 1/N 处理器时间，然后权衡每个进程比例进行动态调整

Processes with the default nice value of zero have a weight of one, so their proportion is unchanged

Processes with a smaller nice value (higher priority) receive a larger weight, increasing their fraction of the processor

process with a larger nice value (lower priority) receive a smaller weight, decreasing their fraction of the processor

virtual time

virtual time += actual run time \* weight (become faster or slower)

nice = > weight =>

为了确定每个进程每次运行的实际时间，CFS needs to divide the proportions into a fixed period

例如：20毫秒，2个相同优先级的进程，则每个进程赋予相同的权重和处理器时间比例，即每个进程10毫秒

如果是5个进程，则每个进程分配4毫秒，如果是200个进程呢？

按照常理应该是 每个进程分配100微秒，但是由于 上下文切换开销巨大，并且 局部性优势丧失，这将严重影响系统的吞吐率

->这时CFS引入另一个变量：minimum granularity (最小粒度)

minimum granularity是任一进程运行时间长度的下限，这将保证 上下文切换开销占 系统总时间开销的比例 不会过大

**-> By assigning proportions of the processor and not fixed timeslices, CFS is able to enforce fairness: each process gets its fair share of the processor**

4. nice val

**processes are assigned priorities that affect how long they run，Unix has historically called these priorities nice values.**

Legal nice values range from −20 to 19 inclusive, with a default value of 0， nice值越大，优先级越低，nice值越小，优先级越高

/\* increments a process's nice value by inc and returns the newly updated value \*/

#include <unistd.h>

int nice (int inc);

Passing 0 for inc is an easy way to obtain the current nice value

nice函数错误时返回-1，但是 新的nice值也可能就是-1，为了区分 函数错误 和 返回nice新值为-1，则如下使用：

int ret;

errno = 0;

ret = nice (10); /\* increase our nice by 10 \*/

if (ret == −1 && errno != 0)

perror ("nice");

else

printf ("nice value is now %d\n", ret);

获取/设置优先级：

#include <sys/time.h>

#include <sys/resource.h>

int getpriority (int which, int who);

int setpriority (int which, int who, int prio);

/\* returns the current process’s priority \*/

int ret;

**ret = getpriority (PRIO\_PROCESS, 0);**

**printf ("nice value is %d\n", ret);**

/\* sets the priority of all processes in the current process group to 10 \*/ int ret;

ret = setpriority (PRIO\_PGRP, 0, 10);

if (ret == −1)

perror ("setpriority");

5. 处理器关联

the process scheduler must decide which processes run on each CPU

如果一个进程在一个CPU核上被调度，the process scheduler should aim to schedule it on the same CPU in the future

因为 进程从一个CPU核迁移到另一个CPU核的代价是巨大的（主要是缓存影响）

if a process moves to a new CPU and writes new data into memory, the data in the old CPU's cache can become stale

这样，进程调度器必须在 进程迁移CPU花销 和 多个CPU负载均衡 之间取得平衡

The Linux scheduler attempts to schedule the same processes on the same processors for as long as possible, migrating a process from one CPU

to another only in situations of extreme load imbalance. This allows the scheduler to minimize the cache effects of migration but still ensure that

all processors in a system are evenly loaded

7. Linux实时调度策略

FIFO调度：A FIFO-classed process will continue running so long as no higher-priority process becomes runnable

时间片轮转：When an RR-classed process exhausts its timeslice, the scheduler moves it to the end of the list of processes at its priority

注意：这两种实时调度策略中，如果存在更高优先级的进程，那么低优先级进程将不会运行

int policy; /\* get our scheduling policy \*/ policy = sched\_getscheduler (0); switch (policy) { case SCHED\_OTHER: printf ("Policy is normal\n"); break; case SCHED\_RR: printf ("Policy is round-robin\n"); break; case SCHED\_FIFO: printf ("Policy is first-in, first-out\n"); break; case -1: perror ("sched\_getscheduler"); break; default: fprintf (stderr, "Unknown policy!\n"); }

实时调度的注意事项：

设计实时程序必须格外小心，can easily bring down the entire system

(1) 如果系统中没有更高优先级的实时进程，CPU密集型 循环程序将会 一直运行到结束

(2) Take care not to starve the rest of the system of processor time

(3) If a real-time process busy-waits for a resource held by a lower-priority process, the real-time process will busy-wait forever.

（因为低优先级进程不会运行从而释放资源）

**關於實現CFS細節**

<http://www.tqcto.com/article/framework/3392.html>

說到些細節, 也點出

<http://blog.csdn.net/ustc_dylan/article/details/7303851>

二、CFS关键点总结

1. 虚拟运行时间(vruntime)变化公式

vruntime += delta \* (1024/se.load.weight);

/\*delta：进程实际增加的运行时间，即从调度实体被选择获得cpu到调度实体放弃CPU这段时间\*/

结论：在实际运行时间相同的情况下，调度实体权重越大，vruntime增加的越慢。

2. 进程的理想运行时间计算公式

ideal\_time = slice \*(se.load.weight/cfs\_rq.load.weight);

/\*slice为CFS运行队列中所有进程运行一遍所需要的时间\*/

/\*slice的经验计算公式如下：\*/

if(cfs\_rq->nr\_running > 5) 大於五個人就算

　　slice = 4 \* cfs\_rq->nr\_running;

else 小於五個人直接給20ms

slice = 20; /\*单位ms\*/

task/

3. CFS调度时机

在有了上面几个计算公式之后，就可以总结出CFS调度算法的几个调度时机：

(1) 调度实体的状态转换的时刻：进程终止、进程睡眠等，广义上还包括进程的创建(fork)；

(2) 当前调度实体的时机运行时间大于理想运行时间（delta\_exec > ideal\_runtime）,这一步在时钟中断 处理函数中完成；

(3) 调度实体主动放弃CPU，直接调度schedule函数，放弃CPU

(4) 调度实体从中断、异常及系统调用返回到用户态时，回去检查是否需要调度；

(se = sched\_entry task meta data for cfs)

需要調度的時機：进程终止、进程睡眠等，广义上还包括进程的创建(fork) -> 所以這邊會有 執行時間<理想分配時間的問題

我們還是有考慮中斷的

<http://loda.hala01.com/2011/12/linux-kernel-%E6%8E%92%E7%A8%8B%E6%A9%9F%E5%88%B6%E4%BB%8B%E7%B4%B9/>

說到CFS nice value & virtual time 的很多細節

timer interrupt: is implemented in a function

一般來說是每1/100秒或每1/1000秒觸發一次Scheduling Tick,讓系統的排程機制可以去計算每個Task執行的時間,並藉此決定是不是要進行Task Context-Switch. (已經做)

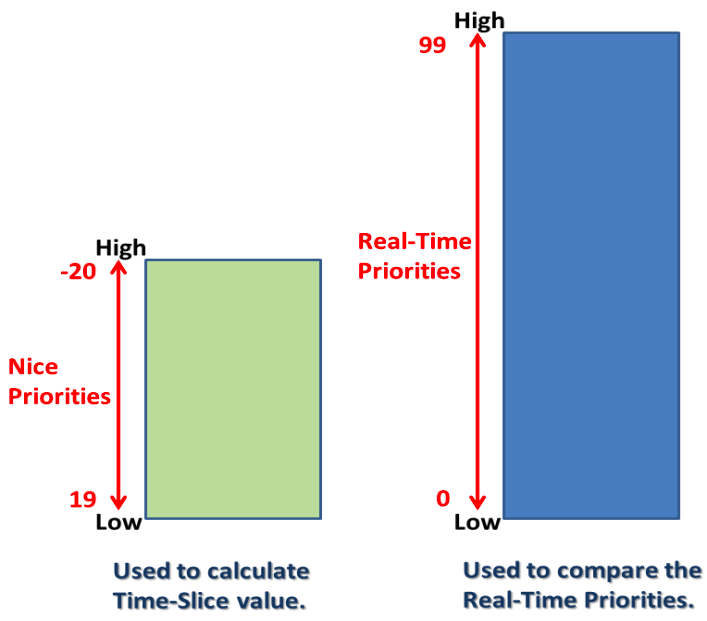
先說原因,再說simulater

Our simulater doesn’t consider the real system overhead. This makes us to get ride of interferences from a system. We consentratly focus on comparing different algorithms.

**CSF RBTREE(Red-Black Tree) – O(Log2(N))**

CFS挑選Task的機制為RB-Tree,概念如下圖所示,包括Task或是Task Group,都會是RBTree中的一個Node支點,每次選擇Task來執行時,都會選擇左邊目前Virtual RunTime最低的Task(也就是最久沒有被執行到的Task)來執行,雖然Virtual RunTime會根據Task的Priority而有增長的差異,例如:Nice Priority低的Task Virutal RunTime會增長的比較快,相對佔據處理器執行的時間就會比較短,反之,Nice Priority 高的Task會因為Virutal RunTime增長的慢,而得到相對比較多實際的處理器執行時間. 但在理想的情況下,如果大家都是Nice 0的情況,基於CFS RBTree的機制,可以讓多數的Task能相對均勻的得到處理器的執行時間,而避免有特定的Task等待比較久的時間沒有得到處理器的執行機會.

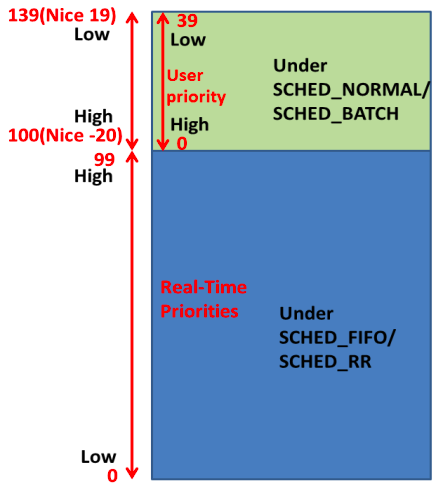
如下圖所示為CFS之前,Linux Kernel 2.6排程的優先級機制,右邊是屬於Task Real-Time優先權，左邊是Task的Nice Value,系統初始化後,一個最原始的行程產生時,預設的Real-Time優先權為0,而Nice優先權亦為0. Linux Kernel會根據Nice Value來計算該Task執行時間Time-Slice值的大小. 當使用者把行程的Nice Value遞減時(最低為-20),就表示該行程的優先權提高,相對的在排程時就會取得比較多的Time-Slice. 相對的,若不斷的增加Nice值(最大為19),就表示該行程的優先權降低,所取得的Time-Slice亦較低.



有關Linux Kernel 2.6.10 核心Nice Value與對應之Time-Slice關係,可以參考下表

|  |  |
| --- | --- |
| Nice Value | Time Slice |
| **-20 -19 -18 -17 -16 -15 -14 -13 -12 -11 -10 -9 -8 -7 -6 -5 -4 -3 -2 -1 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19** | 800ms (DEF\_TIMESLICE \* 4 \* 40 /20) 780ms (DEF\_TIMESLICE \* 4 \* 39 /20) 760ms (DEF\_TIMESLICE \* 4 \* 38 /20) 740ms (DEF\_TIMESLICE \* 4 \* 37 /20) 720ms (DEF\_TIMESLICE \* 4 \* 36 /20) 700ms (DEF\_TIMESLICE \* 4 \* 35 /20) 680ms (DEF\_TIMESLICE \* 4 \* 34 /20) 660ms (DEF\_TIMESLICE \* 4 \* 33 /20) 640ms (DEF\_TIMESLICE \* 4 \* 32 /20) 620ms (DEF\_TIMESLICE \* 4 \* 31 /20) 600ms (DEF\_TIMESLICE \* 4 \* 30 /20) 580ms (DEF\_TIMESLICE \* 4 \* 29 /20) 560ms (DEF\_TIMESLICE \* 4 \* 28 /20) 540ms (DEF\_TIMESLICE \* 4 \* 27 /20) 520ms (DEF\_TIMESLICE \* 4 \* 26 /20) 500ms (DEF\_TIMESLICE \* 4 \* 25 /20) 480ms (DEF\_TIMESLICE \* 4 \* 24 /20) 460ms (DEF\_TIMESLICE \* 4 \* 23 /20) 440ms (DEF\_TIMESLICE \* 4 \* 22 /20) 420ms (DEF\_TIMESLICE \* 4 \* 21 /20) 100ms (DEF\_TIMESLICE \* 1 \* 20 /20) 95ms (DEF\_TIMESLICE \* 1 \* 19 /20) 90ms (DEF\_TIMESLICE \* 1 \* 18 /20) 85ms (DEF\_TIMESLICE \* 1 \* 17 /20) 80ms (DEF\_TIMESLICE \* 1 \* 16 /20) 75ms (DEF\_TIMESLICE \* 1 \* 15 /20) 70ms (DEF\_TIMESLICE \* 1 \* 14 /20) 65ms (DEF\_TIMESLICE \* 1 \* 13 /20) 60ms (DEF\_TIMESLICE \* 1 \* 12 /20) 55ms (DEF\_TIMESLICE \* 1 \* 11 /20) 50ms (DEF\_TIMESLICE \* 1 \* 10 /20) 45ms (DEF\_TIMESLICE \* 1 \* 9 /20) 40ms (DEF\_TIMESLICE \* 1 \* 8 /20) 35ms (DEF\_TIMESLICE \* 1 \* 7 /20) 30ms (DEF\_TIMESLICE \* 1 \* 6 /20) 25ms (DEF\_TIMESLICE \* 1 \* 5 /20) 20ms (DEF\_TIMESLICE \* 1 \* 4 /20) 15ms (DEF\_TIMESLICE \* 1 \* 3 /20) 10ms (DEF\_TIMESLICE \* 1 \* 2 /20) 5m (DEF\_TIMESLICE \* 1 \* 1 /20) |

如下圖所示,筆者把Nice值對應到的Task Priority與對應的排程機制進行對比,我們可以看到在Task Priority 100-139 (對應為Nice -20 – 19 或 User Priority 0-39)的區間內,為一般應用Task的排程機制,使用的Scheduling Class為Normal與Batch,而屬於Real-Time排程機制對應到的Task Priority為0-99,使用的Scheduling Class為Round-Robin與FIFO.



**CFS在Linux Kernel 2.6.23之後,參考Tasks優先級所施行的Time Slice與Real-Time措施 (linux 如何算timeslice)**

基於CFS RBTree的概念,由於會一直確保Virtual RunTime值的平衡,透過抓取佔據Virtual RunTime執行時間最短的Task來執行,因此像是原本排程中會透過Nice值計算對應Task可運作Time Slice的設計方式,就變的需要有所調整,也因此在CFS中會透過函式calc\_delta\_mine (實作在 kernel/sched.c),根據目前Task的優先級計算在排程時每次一個Scheduling Tick要增加的Delta值,也就是說如果該Task Nice值越高(優先級越低)該Delta值就越大,對該執行的Task來說就是Virtual RunTime在運作時,增加的速度越快(等於縮短實際執行的時間),反之,如果該Task Nice值越低(優先級越高)則Delta值就越小,在每次Scheduling Tick觸發時,Virtual RunTime增加的速度越慢,能獲得處理器執行的實際時間也就越長.

Nice -> Delta -> virtual time 大 大 大 (low priority)

讀書會：

static const int prio\_to\_weight[40] = {

/\* -20 \*/ 88761, 71755, 56483, 46273, 36291,

/\* -15 \*/ 29154, 23254, 18705, 14949, 11916,

/\* -10 \*/ 9548, 7620, 6100, 4904, 3906,

/\* -5 \*/ 3121, 2501, 1991, 1586, 1277,

/\* 0 \*/ 1024, 820, 655, 526, 423,

/\* 5 \*/ 335, 272, 215, 172, 137,

/\* 10 \*/ 110, 87, 70, 56, 45,

/\* 15 \*/ 36, 29, 23, 18, 15,

};

weight = 1024 / (1.25 ^ nice\_value)

1.25 <= **将多获得**10%**的**CPU**时间**

很有意思的是计算公式中的1.25是怎么来的？一般的概念是这样的，进程每降低一个nice值，将多获得10%的CPU时间。如果

运行队列里有两个进程，一个nice值为0，另一个nice值为-1。那么按照约定，nice值为0的应该获得45%的CPU时间，而nice值为-1的应该获得

55%的CPU时间。那么两者的权重比例应该是多少？

1/（1+x）＝0.45

根据上面的计算公式，很容易算出，该值约等于1.222左右。内核计算时，选择该值为1.25。具体可阅读prio\_to\_weight定义出的注释。

manu@manu-rush:~$ ps -C nice\_test -o pid,ppid,cmd,etime,nice,pri,psr

PID PPID CMD ELAPSED NI PRI PSR

3885 2695 ./nice\_test 35:02 0 19 0

3886 3885 ./nice\_test 35:02 5 14 0

比如，有两个进程位于CPU的运行队列上，一个nice值是0（权重是1024），另外一个nice值是5（权重是335），按照前面的权重可以推算

出，nice值为0的进程获得CPU的时间应该是nice值为5的3倍。

virtual time += runtime \* (nice / weight)

For example

**在该公式中，**NICE\_0\_LOAD**的值是**nice**值为**0**的进程的权重，即**1024**。前面的例子中，**nice**值为**0**的进程运行了**15**毫秒，因为其权**

**重为**1024**，故其虚拟运行时间也为**15**毫秒；**nice**值为**5**的进程运行时间为**5**毫秒，因为其权重为**335**，所以记账时其虚拟运行时间为：**

5 \* (1024 / 335 ) = 15

nice 0 = 1024 weight

====================================virtual time搞訂

＝＝＝＝＝＝＝＝＝＝＝nice value還沒搞定

=====================**内核的**sched\_slice**函数负责计算进程在本轮调度周期应分得的真实运行时间，其实现代码如下：**

sched\_slice can be larger than timmer interrupt!!!!!!!

**在下一节中可以看到，内核会周期性地检查进程是不是已经耗完了自己的时间片，检查的方法就是判断进程本轮运行时间是否**

**已经超过了**sched\_slice**计算出来的时间片。如果超过，则表示运行时间足够久了，应该发生一次抢占。**

**io\_runtime + cpu\_runtime >** time\_slice => preemption (enq+ deq)

**這我沒用 所以睡覺的定義是不一樣的….. 是另一個queue**

**这条路径之所以很重要，是因为它决定了休眠进程醒来后的虚拟运行时间。回到本节开头的问题：休眠进程的虚拟运行时间会保持不变**

**吗？答案是否定的。很多进程可能会长时间地休眠，在这个过程中，如果虚拟运行时间vruntime保持不变，一旦该进程醒来，它的vruntime就会**

**比运行队列上的其他进程小很多，因为会长时间保持调度的优势。这显然是不合理的。对于这种情况，完全公平调度的做法是，以运行队列的**

min\_vruntime**为基础，给予一定的补偿。**

**补偿多少？这就又要去看看我们的老朋友**place\_entity**函数了。在创建新进程时，曾经走到过该函数，那时该函数负责决定新进程的虚拟运**

**行时间。下面来看看对于被唤醒的休眠进程，该函数是如何决定进程的虚拟运行时间的：**

**更新进程虚拟运行时间的逻辑位于内核的**\_\_update\_curr**函数，该函数里更新了当前进程的真实运行时间和虚拟运行时间，同时也**

**更新了**CFS**运行队列的最小虚拟运行时间。**

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->statistics.exec\_max, max((u64)delta\_exec, curr->statistics.exec\_max));

/\***更新进程的真实运行时间**\*/

curr->sum\_exec\_runtime += delta\_exec;

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

/\*calc\_delta\_fair**用来计算加权后的运行时间**\*/

delta\_exec\_weighted = calc\_delta\_fair(delta\_exec, curr); //

/\***更新进程的虚拟运行时间**\*/

curr->vruntime += delta\_exec\_weighted; //

/\***更新运行队列的最小虚拟运行时间**\*/

update\_min\_vruntime(cfs\_rq); //all run\_queue

#if defined CONFIG\_SMP && defined CONFIG\_FAIR\_GROUP\_SCHED

cfs\_rq->load\_unacc\_exec\_time += delta\_exec;

#endif

**内核会将所有进程中虚拟运行时间的最小值记录到运行队列的最小虚拟运行时间（**vruntime**）中。**

**为什么要维护这个值？**CFS**算法可确保队列上的所有进程步调一致地轮流运行，虚拟运行时间不断增大，大部分进程的虚拟运行**

**时间相差也不会太远。但是记录下队列虚拟运行时间的最小值仍然是有意义的。比如新加入一个进程，应该给它的虚拟运行时间赋**

**初始值，初始值应是多少？**

virtual\_runtime is not always 0

g\_lowest\_vrtime // Linux each queue // we here consider only a global queue // multiple run\_queue -> we need to consider Load Balance

**比如新加入一个进程，应该给它的虚拟运行时间赋**

**初始值，初始值应是多少？再比如进程陷入了漫长的休眠，醒来时已经沧海桑田，相对其他进程，它的虚拟运行时间已经大幅落**

**后。内核应该将该进程的虚拟运行时间调整成何值？又比如内核不得不将某个进程从一个**CPU**的运行队列拉到另一个**CPU**的运行队列**

**中，该进程的虚拟运行时间该如何调整？此时，维护运行队列的最小虚拟运行时间的意义就彰显出来了。运行队列的最小虚拟运行**

**时间给了我们一个基准，根据这个基准值可以知道，该**CPU**运行队列上的大部分进程的虚拟运行时间就在该值附近，且大于该值。**

**在后面分析新创建的进程和唤醒休眠进程时，会分析内核如何调整这些进程的虚拟运行时间。**

**Timmer interrupt**

scheduler\_tick

task\_tick

task\_tick\_fair

**在**task\_tick\_fair**函数中，内核更新了正在运行的进程的时间统计，包括真实运行时间和虚拟运行时间，代码如下：**

static void task\_tick\_fair(struct rq \*rq, struct task\_struct \*curr, int queued)

{

struct cfs\_rq \*cfs\_rq;

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

/\***为了支持组调度，引入了调度实体的概念**

\*/

for\_each\_sched\_entity(se) {

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

entity\_tick(cfs\_rq, se, queued);

}

} static void

entity\_tick(struct cfs\_rq \*cfs\_rq, struct sched\_entity \*curr, int queued)

{

/\***更新正在运行进程的统计信息**

\*/

update\_curr(cfs\_rq);

update\_entity\_shares\_tick(cfs\_rq);

...

/\***如果可运行状态的进程个数大于**

1**，检查是否可以抢占当前进程**

\*/

if (cfs\_rq->nr\_running > 1)

check\_preempt\_tick(cfs\_rq, curr);

}

**What we don’t support**

Linux supports

fork -> 0 virtual runtime

CGroups (process)

huge process(100 tasks) vs small process (1tasks)

Linux 想以process為主

Linux内核实现了cgroups（control groups的缩写）功能，该功能用来限制、记录和隔离一个进程组

群所使用的物理资源（如CPU、内存、磁盘IO、网络等）。为了管理不同的资源，cgroups提供了一系

列子系统，本节将要介绍的cpu和后面CPU亲和力一节介绍的cpuset都属于cgroups的子系统。cpu子系统

只用于限制进程的CPU使用率。接下来介绍如何使用cgroups的cpu子系统来创建组（task group）及按组

来分配CPU资源。

Linux**提供了**RLIMIT\_NICE**资源限制。如果一个进程的**RLIMIT\_NICE**限制为**25**，那么其**nice**值**

**可以提升到**20-25**＝**-5

( mic\_ cfs value new comer)

其中START\_DEBIT特性是用来给新创建的进程略加惩罚的。如果没有START\_DEBIT选项，子进程的虚拟运行时间为：

这个值通常比较小，这就意味着子进程很快就能获得调度的机会，因此也就给了恶意进程可乘之机。因为恶意进程可以通过不停

地fork来获得更多的CPU时间。如果设置了START\_DEBIT选项，会通过增大子进程的虚拟运行时间来惩罚新创建的进程，使新创建的进程晚一点才能获得被调度的机会。

Fork 出來的 task不一定會在同一個cpu上跑 所以virtualtime 也不一定是跟老爸的一樣

our limitation:

在对称多处理器（SMP）的系统上，存在着多个处理器，那么所有处于可运行状态的进程是应该位于一个队列上，还是每个处理

器都要有自己的队列？这大概是进程调度首先要解决的问题。

目前Linux采用的是每个CPU都要有自己的运行队列，即per cpu run queue。每个CPU去自己的运行队列中选择进程，这样就降低了

竞争。这种方案还有另外一个好处：缓存重利用。某个进程位于这个CPU的运行队列上，经过多次调度之后，内核趋于选择相同的

CPU执行该进程。这种情况下上次运行的变量很可能仍然在CPU的缓存中，这样就提升了效率。

所有的CPU共用一个运行队列这种方案的弊端是显而易见的，尤其是在CPU数目很多的情况下。我们可以想象一下如果存在1024

个CPU，都要去同一个运行队列取下一个调度的进程，这种竞争无疑会降低调度器的性能。

但是凡事无绝对，没有最好的，只有最适合的。对于CPU核数比较少的桌面应用来说，只有一个运行队列的Brain Fuck

Scheduler（脑残调度器）却表现的异常出色。 [2] [3]

Linux选择了每一个CPU都有自己的运行队列这种解决方案。这种选择也带来了一种风险：CPU之间负载不均衡，可能出现一些

CPU闲着而另外一些CPU忙不过来的情况。为了解决这个问题，load\_balance就闪亮登场了。load\_balance的任务就是在一定的时机下，

通过将任务从一个CPU的运行队列迁移到另一个CPU的运行队列，来保持CPU之间的负载均衡。

static priority

SP = priority120 + nice = 100~139

Dynamic priority

Calculate Quantum =    (140 − SP ) × 20 if SP < 120

(140 − SP ) × 5 if SP ≥ 120

Other mechanisms used for quick interactive response:

Dynamic priority is calculated from static priority and average sleep time

When process wakes up, record **how long it was sleeping**, up to some **maximum value**

When the process is running, decrease **that** **value** each timer tick

Roughly speaking, the bonus is a number in [0, 10] that measures what percentage of the time the process was sleeping recently; 5 is neutral, **10 helps priority by 5**, **0 hurts priority by 5**

**DP** = max(100, min(SP − bonus + 5, 139))

A process is interactive if

bonus − 5 ≥ (S/4) − 28

120/4=30 30-28=2

bonus5-5

需要模擬的所有進程種類

**此外，不同的进程之间，其行为模式可能存在着巨大的差异。进程的行为模式可以粗略地分成两类：**CPU消耗型（CPU bound）**和**I/O消耗型（I/O bound）**。**

**所谓**CPU**消耗型是指进程因为没有太多的**I/O**需求，始终处于可运行的状态，始终在执行指令。而**I/O**消耗型是指进程会有大量**I/O**请求（比如等待键盘键入、读写块设备上的文件、等待网络**I/O**等），它处于可执行状态的时间不多，而是将更多的时间耗费在等待上。当然这种划分方法并非绝对的，可能有些进程某段时间表现出**CPU**消耗型的特征，另一段时间又表现出**I/O**消耗型的特征。**

**还有另外一种进程分类的方法，如下。**

·**交互型进程：这种类型的进程有很多的人机交互，进程会不断地陷入休眠状态，等待键盘和鼠标的输入。但是这种进程对系统的响应时间要求非常高，用户输入之后，进程必须被及时唤醒，否则用户就会觉得系统反应迟钝。比较典型的例子是文本编辑程序和图形处理程序等。**

·**批处理型进程：这类进程和交互型的进程相反，它不需要和用户交互，通常在后台执行。这样的进程不需要及时的响应。比较典型的例子是编译、大规模科学计算等，一般来说，这种进程总是**“**被侮辱的和被损害的**”**。**

·**实时进程：这类进程优先级比较高，不应该被普通进程和优先级比它低的进程阻塞。一般需要比较短的响应时间。系统之中，有很多性格各异的进程，这就增加了设计调度器的难度。有一个很有意思的比喻来描述调度器的困境** [1] **：**Linux**内核调度器就像处境尴尬的主妇，满足孩子对晚餐的要求便有可能会伤害到老人的食欲，做出一桌让男女老少都满意的饭菜实在是太难**

**了。**

**设计一个优秀的进程调度器绝不是一件容易的事情，它还有很多事情需要考虑，很多目标需要达成：**

**·公平：每一个进程都可以获得调度的机会，不能出现“饿死”的现象。**

**·良好的调度延迟：尽量确保进程在一定的时间范围内，总能够获得调度的机会。**

**·差异化：允许重要的进程获得更多的执行时间。**

**·支持软实时进程：软实时进程，比普通进程具有更高的优先级。**

**·负载均衡：多个CPU之间的负载要均衡，不能出现一些CPU很忙，而另一些CPU很闲的情况。**

**·高吞吐量：单位时间内完成的进程个数尽可能多。**

**·简单高效：调度算法要高效。不应该在调度上花费太长的时间。**

·低耗电量：在系统并不繁忙的情况下，降低系统的耗电量。

**在对称多处理器（**SMP**）的系统上，存在着多个处理器，那么所有处于可运行状态的进程是应该位于一个队列上，还是每个处理**

**器都要有自己的队列？这大概是进程调度首先要解决的问题。**

**目前**Linux**采用的是每个**CPU**都要有自己的运行队列，即**per cpu run queue**。每个**CPU**去自己的运行队列中选择进程，这样就降低了**

**竞争。这种方案还有另外一个好处：缓存重利用。某个进程位于这个**CPU**的运行队列上，经过多次调度之后，内核趋于选择相同的**

CPU**执行该进程。这种情况下上次运行的变量很可能仍然在**CPU**的缓存中，这样就提升了效率。**

**所有的**CPU**共用一个运行队列这种方案的弊端是显而易见的，尤其是在**CPU**数目很多的情况下。我们可以想象一下如果存在**1024

**个**CPU**，都要去同一个运行队列取下一个调度的进程，这种竞争无疑会降低调度器的性能。**

Linux**选择了每一个** CPU**都有自己的运行队列这种解决方案。这种选择也带来了一种风险：** CPU**之间负载不均衡，可能出现一些**

CPU**闲着而另外一些** CPU**忙不过来的情况。为了解决这个问题，** load\_balance**就闪亮登场了。** load\_balance**的任务就是在一定的时机下，**

**通过将任务从一个** CPU**的运行队列迁移到另一个** CPU**的运行队列，来保持** CPU**之间的负载均衡。**

5.4.4**睡眠进程醒来**

**睡眠进程的虚拟运行时间会保持不变吗？**

**如何对待睡眠进程也是调度器需要解决的问题。因为交互型的进程会不断陷入休眠状态中，并等待用户的输入。虽然这类进程对**CPU**的整**

**体消耗并不大，但是要求响应必须及时，否则用户会感觉到系统卡顿，用户体验就会很糟糕。**

**对**CFS**之前的**O**（**1**）调度器来说，交互型进程堪称其阿喀琉斯之踵。该调度算法的交互进程识别启发式算法异常复杂，该启发式算法融入 (this is algorithm it self)**

**了睡眠时间作为考量的标准，但是对于一些特殊的情况，经常判断不准，而且经常是改完一种情况又发现另一种特殊情况。**

CFS**调度算法并没有刻意地区分交互型进程和批处理型进程，依然漂亮地满足了交互型进程需要及时响应的需求。**CFS**算法是如何做到的**

**呢，对于从休眠中醒来的进程，**CFS**进行了哪些处理呢？这是本节要介绍的内容。**

----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

1. 多任务操作系统有2种类型：cooperative and preemptive multitasking, 即非抢占式和抢占式. Linux是抢占式多任务操作系统.

2. Linux 2..5版本使用叫o(1)的scheduler, Linux2.6.23使用叫completely fair scheduler(CFS).

3. 进程可分为I/O-bound和processor-bound两种类型.前者依赖于IO或者输入,如UI等,后者一直占用CPU,如算法等. Linux调度时倾向于IO-bound类型进程，但也不会怱略processor-bound进程.

4. Linux有两种优先级:一种叫nice value,值从-20到19,值越大,优先级越低.通过控制时间片的比例来区分优先级.使用ps-el命令可以查看进程优先级。NI列显示的值.

另一种是real-time优先级范围，值从0到99.PTPRIO列显示的.如以下命令:

ps -eo state,uid,pid,ppid,rtprio,time,comm.

5.Linux支持针对不同进程的不同算法,称之为Scheduler class,定义在kernel/sched.c.

cfs是常规进程的调度算法,SCHED\_NORMAL,定义在kernel/sched\_fair.c.

6. Linux’s CFS scheduler不分配时间片给进程,而是分配占用处理器的比例.进程获得处理器的时间是由一个系统负载函数决定的.这个比例是由进程的nice value影响的,nice value值大的(低优先级)挂起的处理器时间比例较小,反之较大.一个进程如果消耗的处理器时间比例比当前进程小,就会抢占当前进程.

7. (CFG基于一个概念: 调度模型就如同系统有理想的,完美多任务处理器.) CFS不给进程分配timeslice值,而是计算一个进程需要运行多久,基于需要运行的进程数的函数计算而得.CFS不使用nice value来计算timeslice,而是通过nice value计算进程能够获得占用处理器运行的时间比例.value大的(高优先级)获得比重较大,value小的(低优先级)获得比重较小.

CFS使用targeted Latency来计算timeslice,latency越小,越接近完美多任务. CFS用minimum granularity来表示timeslice的最小值,因为越小的值会加重处理器切换进程的负载.

处理器时间的分配比例是由nice value的相对值决定的.CFS给每个进程一个公平的享有处理器时间,它不是绝对完美的(进程特别多的时候),但是是相对完美的.

8.CFS使用scheduler entity structure, struct sched\_entity, defined in <linux/sched.h>,来记录进程运行的时间.

struct sched\_entity {

struct load\_weight load;

struct rb\_node run\_node;

struct list\_head group\_node;

unsigned int on\_rq;

u64 exec\_start;

u64 sum\_exec\_runtime;

u64 vruntime;

u64 prev\_sum\_exec\_runtime;

u64 last\_wakeup;

u64 avg\_overlap;

u64 nr\_migrations;

u64 start\_runtime;

u64 avg\_wakeup;

/\* many stat variables elided, enabled only if CONFIG\_SCHEDSTATS is set \*/

};

该scheduler entity structure是process descriptor, struct task\_stuct的成员,名称为se.

9. 变量vruntime存储了进程的虚拟运行时间,就是根据运行的进程数得到的实际运行时间.单位是ns.CFS使用vruntime计算进程已经运行的时间和还要运行的时间.函数update\_curr(), defined in kernel/sched\_fair.c, 来进行这个工作.

10. CFS选择vruntime最小的进程作为下一个运行的进程.CFS算法的核心就是选择vruntime最小的进程. CFS使用red-black tree来管理就绪进程的表,来有效的寻找vruntime最小的进程.red-black tree,Linux中叫做rbtree.是一个自我平衡的二叉树(self-balancing binary search tree)

11. vruntime最小的进程位于rbtree的最左端(leftmost node),函数\_\_pick\_next\_entity(), defined in kernel/sched\_fair.c完成这样的选择.这个函数并没有遍历这个树,而是从rb\_leftmost直接获得.

12. 进程就绪(wake up)或者第一次通过vfork()创建时,将该进程节点保存到rbtree,并获得leftmost的节点.enqueue\_entity()完成这个工作.enqueue\_entity()更新运行时和统计信息,然后调用\_\_enqueue\_entity()进行实际的插入节点工作.

13. 进程阻塞或者终止时,CFS将该进程从rbtree移除.dequeue\_entity()通过调用 \_\_dequeue\_entity()(\_\_dequeue\_entity()调用rb\_erase())来完成移除.

14. 进程调度的主入口函数是schedule(),定义在kernel/sched.c.schedule()调用 pick\_next\_task()来遍历scheduler class,选择优先级最高的scheduler class中优先级最高的进程.

15. 等待队列(wait queue): wake\_queue\_head\_t, 通过 DECLARE\_WAITQUEUE()静态创建,通过init\_waitqueue\_head()动态创建.

内核任务睡眠的推荐方法:

/\* ‘q’ is the wait queue we wish to sleep on \*/

DEFINE\_WAIT(wait);

add\_wait\_queue(q, &wait);

while (!condition) { /\* condition is the event that we are waiting for \*/

prepare\_to\_wait(&q, &wait, TASK\_INTERRUPTIBLE);

if (signal\_pending(current))

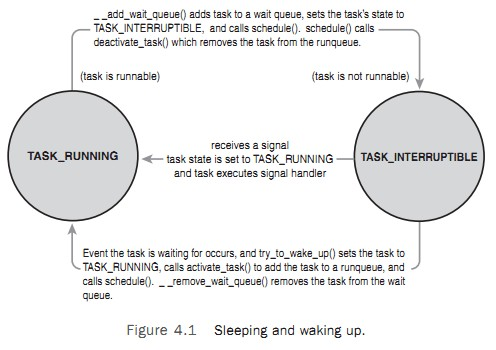
/\* handle signal \*/

schedule();

}

finish\_wait(&q, &wait);

16. wake\_up()来实现唤醒. 调用It calls try\_to\_wake\_up()来将任务状态设置为TASK\_RUNNING,调用enqueue\_task()将任务添加到the red-black tree, 如果优先级比当前运行的任务高就设置need\_resched.下图描述了调度状态之间的关系.



17. context\_switch()来进行任务切换(kernel/sched.c).需要切换任务时, schedule()会调用context\_switch().

调用switch\_mm()(<asm/mmu\_context.h>), 来切换新进程的虚拟内存.

调用switch\_to()(<asm/system.h>), 切换新进程的处理器状态.包括保存和恢复栈信息,处理器寄存器以及其他需要在进程运行之前保存的架构特定的状态.

18. 内核提供变量need\_resched来提示调度需要进行.在scheduler\_tick()中当进程需要被抢占时,need\_resched被设置.当有比当前进程优先级高的进程被唤醒时, try\_to\_wake\_up()设置need\_resched.这个变量是每个进程都有的,而不是简单的全局变量.在2.6内核中, 是thread\_info结构的一个变量的位成员.

19. 用户抢占发生在:从系统调用返回用户空间时;从中断处理函数返回用户空间时.

20. Linux支持内核任务的抢占, 在每个进程的thread\_info结构中有个变量preempt\_count,进行lock操作的计数.当为0的时候,内核进程就可以被抢占.

内核任何调度发生在:

中断处理函数结束,返回内核空间之前时;

内核代码有一次可被抢占时;

内核任务显式调用schedule();

内核任务阻塞时(这会导致调用schedule().

21. Linux提供两种实时调度的策略:SCHED\_FIFO和SCHED\_RR.实时调度不使用CFS,而是real-time scheduler(kernel/sched\_rt.c).

SCHED\_FIFO实现了先进先出的调度算法,没有时间片概念.运行的SCHED\_FIFO任务能够抢占任何SCHED\_NORMAL任务.当SCHED\_FIFO就绪时,它会运行知道自己阻塞或者显式放弃处理器.它没有时间片会一直运行至完成.只有高优先级的SCHED\_FIFO或SCHED\_RR任务能够抢占.

SCHED\_RR与SCHED\_FIFO相似,仅仅在于每个运行运行预先定义好的时间片,就是SCHED\_FIFO with timeslices,real-time, round-robin scheduling algorithm.时间片仅用来调度相同优先级的进程.

22. Real-time scheduling policies提供软实时服务.实时优先级范围从0到MAX\_RT\_PRIO - 1.MAX\_RT\_PRIO默认值是100.这与SCHED\_NORMAL进程共享优先级空间.范围是MAX\_RT\_PRIO to (MAX\_RT\_PRIO + 40),默认值就是100-139,对应的nice value是-20~19.

23. Scheduler-Related System Calls

-----------------

<https://hackpad.com/ep/pad/static/oqzjExf8vmc>

時間片段(timeslice)

一個process從執行到被preempt的時間就是一個timeslice.(Time slice represents how long a task can run until it is preempted.)

在Linux中，process的processor time是依據其nice value來決定。nice value越小，processor time越長；nice value越大processor time越短。(P.45)(Processes with higher nice values (a lower priority) receive a smaller proportion of the processor; processes with smaller nice values (a higher priority) receive a larger proportion of the processor.)

**For nice value equal to 0, factor is 1; vruntime is same as real time spent by task on CPU.**

**For nice value less than 0, factor is < 1; vruntime is less than real time spent. vruntime grows slower than real time.**

**For nice value more than 0, factor is > 1; vruntime is more than real time spent. vruntime grows faster than real time.**

**‘weight’ is roughly equivalent to 1024 \* (1.25)^(-nice)**

static const int prio\_to\_weight[40] = {

/\* -20 \*/ 88761, 71755, 56483, 46273, 36291,

/\* -15 \*/ 29154, 23254, 18705, 14949, 11916,

/\* -10 \*/ 9548, 7620, 6100, 4904, 3906,

/\* -5 \*/ 3121, 2501, 1991, 1586, 1277,

/\* 0 \*/ 1024, 820, 655, 526, 423,

/\* 5 \*/ 335, 272, 215, 172, 137,

/\* 10 \*/ 110, 87, 70, 56, 45,

/\* 15 \*/ 36, 29, 23, 18, 15,

};

2.6以前 ：O(1)排班器

2.6之後 ：CFS(Completely Fair Scheduler)

CFS

CFS的概念是讓每個行程都能公平的分配執行時間

假設有n個行程, 每個行程應該會分到1/n的CPU時間

但實際情況下在切換行程時會有context switch的開銷 所以無法達到完美的分配

CFS選擇執行的行程方式是觀察每個行程的虛擬執行時間( virtual runtime)並選擇虛擬執行時間最小的行程

而nice值在這邊當作權重計算虛擬執行時間

為了計算實際的timeslice, CFS會設定延遲目標(targeted lantency)

假設延遲目標為20ms 並有兩個行程

不管這兩個行程的優先權為何, 在一個行程被搶佔之前, 每個行程都可以執行10ms

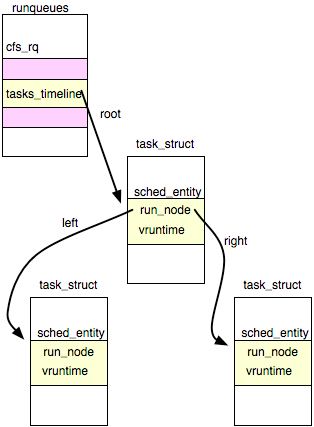
如果有5個行程, 那每個行程都會執行到4ms

所以timeslice是隨行程數量而改變嗎？

按照timeslice的定應:"一個process從執行到被preempt的時間"，是的

如果行程數量越大, 行程分配到的timeslice會越少, context switch得成本也會變得過高

因此timeslice會設定一個下限(預設 1ms) 下限值稱作最小粒度（minimum granularity） 即不管行程再多, 每個行程也會至少執行到1ms



the core of CFS’s scheduling algorithm: Pick the task with the smallest vruntime.

假設target latency為20 milliseconds，因此每個process 至少執行10 milliseconds才會被preempt，在60milliseconds的時間內，假設process :emoji\_1f418: 先被執行，經過10 milliseconds之後會更新process :emoji\_1f418: 的 vruntime，並且在尋找rb tree中最小的vruntime的 process並且執行，排程結果如下圖，在60 milliseconds中process :emoji\_1f418: 與process :emoji\_1f411: 的時間為40milliseconds 和20 milliseconds。

:\_1f418: : \_1f411: : \_1f418: : \_1f418: : \_1f411: : \_1f418: x

|----------| ----------| ----------| ----------| ----------| ----------|

0 10 20 30 40 50 60 (單位:ms)

綿羊大象排排站，別讓:emoji\_1f437: :emoji\_1f437: 不開心

CFS實作重點是虛擬執行時間（vruntime）+ 紅黑樹

CFS的資料結構是紅黑:emoji\_1f332:

紅黑樹的每個節點都代表一個行程, 鍵值代表的是該行程的虛擬執行時間

行程選擇就是選虛擬時間最小的行程, 因此只要一直沿著左節點往下走就可以找到虛擬執行時間最小的行程

CFS有做優化, 將最左邊的節點存在struct cfs\_rq裡的rb\_leftmost, 因此不用每次選行程都重新跑一次紅黑樹

cfs\_rq裡有rb\_root, rt\_node \*rb\_leftmost等成員

Adding Processes to the Tree when a process becomes runnable (wakes up) or is first created via fork():

This is performed by enqueue\_entity() . enqueue\_entity() calls \_\_enqueue\_entity()

\_\_enqueue\_entity() 走訪這棵紅黑數，[宣告一個參數leftmost，初始化為1，用來判斷新加入的process是否落在最左邊]，並比較key,如果新加入的這個process的vruntime是最小的，那它就會落在紅黑數的最左邊，而leftmost就維持1，反過來說，當它不是最小的，那它會被往右邊移動，只要往右邊移動過，leftmost就會被設為0.

if (key < entity\_key(cfs\_rq, entry)) {

link = &parent->rb\_left;

}

else {

link = &parent->rb\_right;

leftmost = 0;

最後，如果這個process是leftmost(leftmost = 1)，則它會被cache起來：

if (leftmost)

cfs\_rq->rb\_leftmost = &se->run\_node;

Removing Processes from the Tree when a process blocks (becomes unrunnable) or terminates:

This is performed by dequeue\_entity() . dequeue\_entity() calls \_\_dequeue\_entity().

當要被移除的process是leftmost的時候，會執行rb\_next()，把vruntime次小的process加到leftmost的位置

if (cfs\_rq->rb\_leftmost == &se->run\_node) {

struct rb\_node \*next\_node;

next\_node = rb\_next(&se->run\_node);

cfs\_rq->rb\_leftmost = next\_node;

virtual runtime

Virtual run time is the weighted time a task has run on the CPU,measured in nanosecond.

vruntime被包在struct sched\_entity<linux/sched.h>

struct task\_struct

{

……

struct sched\_entity se;

……

};

struct sched\_entity

{

……

u64 vruntime;

……

};

kernel/sched\_fair.c中的update\_curr()負責計算vruntime

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;

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

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

curr->vruntime += delta\_exec\_weighted;

update\_min\_vruntime(cfs\_rq);

}

vruntime is updated by the function \_\_update\_curr(),and the function \_\_update\_curr() is called by update\_curr() function.

行程選擇

每個scheduler class都實作自己的pick\_next\_task

是指每個class的pick\_next\_task都不完全一樣嗎，感覺這描述怪怪的

不一樣 CFS選法跟sched\_FIFO的選法就不同

在scheduler選擇行程時

rq->nr\_running == rq.cfs.nr\_running 是否成立 (代表現在的行程是否都屬於CFS排班器)

成立 就會直接使用CFS選擇行程

不成立 則會從最高優先權的sched\_class開始呼叫pick\_next\_task直至選到下一個要執行的行程

/\* pick\_next\_task() \*/

for ( ; ; ) {

p = class->pick\_next\_task(rq);

if (p)

return p;

/\*

\* Will never be NULL as the idle class always

\* returns a non-NULL p:

\*/

class = class->next;

}

pick\_next\_task() calls pick\_next\_task\_fair():

/\* pick\_next\_task\_fair() \*/

if (!cfs\_rq->nr\_running) //如果這個runqueue裡面沒有process

return NULL;

do {

se = pick\_next\_entity(cfs\_rq);

set\_next\_entity(cfs\_rq, se);

cfs\_rq = group\_cfs\_rq(se);

} while (cfs\_rq);

pick\_next\_entity() calls \_\_pick\_next\_entity():

/\* \_\_pick\_next\_entity() \*/

static struct sched\_entity \*pick\_next\_entity(struct cfs\_rq \*cfs\_rq)

{

struct sched\_entity \*se = \_\_pick\_next\_entity(cfs\_rq); //把rbtree最左邊的process挑出來

/\*

\*為避免過度頻繁的context switch，所以加上一些判斷(與granularity有關)

\*/

if (cfs\_rq->next && wakeup\_preempt\_entity(cfs\_rq->next, se) < 1)

return cfs\_rq->next;

if (cfs\_rq->last && wakeup\_preempt\_entity(cfs\_rq->last, se) < 1)

return cfs\_rq->last;

return se;

}

-----------------

<https://hwchen18546.wordpress.com/2014/04/15/linux-scheduling/>

在Linux CFS (Completely Fair Scheduler)的設定下time slice 並非一個固定值

而是依據正在執行的processes它們的nice value來分配各個process 使用CPU的比例(proportion)

也就是會依系統的loading 及其對應的priority value 來作time slice 動態的調整

The period of time for which a process is allowed

to run in a preemptive multitasking system is generally called the “**time slice**”, or “**quantum**”.

Timeslice(quantum)不能太短或太長 (中斷時間)

CPU密集型：processes are hungry for CPU time，比如科学计算、数学计算、图像处理

I/O密集型：spend more time blocked waiting for some resource than executing，often issuing and waiting for file or network I/O, blocking on keyboard input

太短會一直contentswitch(overhead)

太長會沒有concurrent,就會壟斷

一般長度以兩個content switch為基準

在**sleep status** slice會升

在**run status** slice會降

loop雖然cup高

但因為**一直run所以priority**會降,

有人要會直接被幹走

Q: priority?

A: PRI(new) = PRI(old) + nice value

所以nice value越低優先度越高

root權限可以調整-20~19

一般權限只能調整0~19

用nice可以看值、用renice可以設定

在linux裡面總共有140個priority level

每個level都有一個priority queue

schedule在執行先看是否在run queue

然後是否有time slice

一旦開始run就開始扣掉

所有run queue都沒有time slice就重發薪水

……..

**在Linux CFS (Completely Fair Scheduler)的設定下**

**time slice 並非一個固定值**

**而是依據正在執行的processes它們的nice value**

**來分配各個process 使用CPU的比例(proportion)**

**也就是會依系統的loading**

**及其對應的priority value 來作time slice 動態的調整**

**新加入的process是否能preempting current process，**

**是根據該Process 的priority 及available time slice**

At every time tick, ()

decrease the quantum of the current running process

If(時間到還沒執行完){

If the process is non-interactive,

put it aside on the expired list

Else if the process is interactive,

put it at the end of the current priority queue

(Linux認為和使用者互動的程序優先度要高)

If there’s nothing else at that priority,

it will run again immediately

}