CFS [调度程序](https://www.zhihu.com/search?q=%E8%B0%83%E5%BA%A6%E7%A8%8B%E5%BA%8F&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra={"sourceType":"article","sourceId":72754729}" \t "/Users/wanghaoyu/Documents\\x/_blank)并不采用严格规则来为一个优先级分配某个长度的时间片，而是为每个任务分配一定比例的 CPU 处理时间。每个任务分配的具体比例是根据nice值来计算的。nice值的范围从 -20 到 +19，数值较低的nice值表示较高的相对优先级。具有较低nice值的任务，与具有较高nice值的任务相比，会得到更高比例的处理器处理时间。默认nice值为 0。

**为什么叫nice？当一个任务增加了它的nice，说明它的优先级降低了，进而对其他任务变得nice**

CFS 调度程序没有直接分配优先级。相反，它通过每个任务的变量 vruntime 以便维护虚拟运行时间，进而记录每个任务运行多久。虚拟运行时间与基于任务优先级的衰减因子有关，更低优先级的任务比更高优先级的任务具有更高[衰减速率](https://www.zhihu.com/search?q=%E8%A1%B0%E5%87%8F%E9%80%9F%E7%8E%87&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra={"sourceType":"article","sourceId":72754729}" \t "/Users/wanghaoyu/Documents\\x/_blank)。对于正常优先级的任务（nice值为 0），[虚拟运行时间](https://www.zhihu.com/search?q=%E8%99%9A%E6%8B%9F%E8%BF%90%E8%A1%8C%E6%97%B6%E9%97%B4&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra={"sourceType":"article","sourceId":72754729}" \t "/Users/wanghaoyu/Documents\\x/_blank)与实际物理运行时间是相同的。下面分析一下 CFS 调度程序是如何工作的。

假设有两个任务，它们具有相同的nice值。一个任务是 I/O [密集型](https://www.zhihu.com/search?q=%E5%AF%86%E9%9B%86%E5%9E%8B&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra={"sourceType":"article","sourceId":72754729}" \t "/Users/wanghaoyu/Documents\\x/_blank)而另一个为 CPU 密集型。通常，I/O 密集型任务在运行很短时间后就会阻塞以便等待更多的 I/O；而 CPU 密集型任务只要有在处理器上运行的机会，就会用完它的时间片。

因此，I/O 密集型任务的虚拟运行时间最终将会小于 CPU 密集型任务的，从而使得 I/O 密集型任务具有更高的优先级。这时，如果 CPU 密集型任务在运行，而 I/O 密集型任务变得有资格可以运行（如该任务所等待的 I/O 已成为可用)，那么 I/O 密集型任务就会抢占 CPU 密集型任务。

更糟糕的是，还有许多其他因素，包括缓存位置、非统一

内存访问(NUMA)拓扑、CPU亲和力、CPU带宽控制和异构平

台中的CPU容量，这些因素与CPU共享无关，但影响进程分

配给运行队列的方式。





在一个调度期间内运行队列中的所有进程都有机会运行，CPU时间的数量根据运行队列中所有进程的权重划分为时间段。



因此，任务权重只标示相对的CPU共享，而不是用户期望他们的容器根据预订获得的绝对CPU共享。由于权重只是相对的，一旦一个容器与另一个容器共享运行队列，就不能再保证其承诺的数量资源。

sysctl\_sched\_min\_granularity

表示进程最少运行时间，防止频繁的切换

CFS把调度周期sched\_latency按照进程的数量平分，给每个进程平均分配CPU时间片（当然要按照nice值加权，为简化起见不再强调），但是如果进程数量太多的话，就会造成CPU时间片太小，如果小于sched\_min\_granularity\_ns的话就以sched\_min\_granularity\_ns为准；而调度周期也随之不再遵守sched\_latency\_ns，而是以 (sched\_min\_granularity\_ns \* 进程数量) 的乘积为准。

sysctl\_sched\_wakeup\_granularity参数。该参数的作用是在唤醒新任务时，只有在当前任务curr的虚拟时间比被唤醒任务p的虚拟时间多于sysctl\_sched\_wakeup\_granularity参数的加权平均值时才会考虑让新唤醒任务p抢占当前任务curr。

幻影CPU时间例子：

假设我们有两个容器，它们的配置列在表中。这里有两个容器T和N在一个有三个cpu的主机上运行。容器T只有一个线程，而容器N有三个线程：N1−N3。T和N请求的cpu分别为1和2。由于应用程序的性质、负载平衡和一些其他的原因，T和N1被分配给CPU0的同一运行队列。N2和N3分别被分配给CPU1和CPU2。

当T是一个批处理应用程序时的情况。

如图3a所示，CPU1和CPU2被相邻的线程 N2和N3完全消耗。容器T只得到CPU0的60%，因为其CPU权重为1024，N1的CPU权重为683。这只是T请求的cpu的60%。

当N1−N3消耗其所有的CPU配额时(i。e.,2)，所有这三个线程将在当前调度期间暂停，CPU0-CPU2将可用。





因此，无论邻居是受限的还是不受限的，目标 容器接收到的实际CPU时间总是明显低于其请求量，尽管系统中没有过度承诺。由于负载平衡和对CPU共享概念的定义方式（不考虑并行消耗），一个容器拥有的线程越多，与线程较少的其他容器混合时的破坏性就越大，而且在发生突发事件的情况下，它可以不公平地使用的cpu就越多。

当T是一个交互式应用程序时 的情况。与批处理应用程序不同，交互式应用程序可能会醒来，并且在返回睡眠之前只做相对较少的工作。例如，内存缓存接收一个请求，处理它，响应客户端，然后返回

休眠状态。但是，它需要快速做出反应，即在用户请求到达时立即醒来。当邻居结构稳定时，如图4a所示，由于执行最小粒度，T不能及时醒来。

理论上T可以在CPU1或CPU2上运行，但是由于缓存亲和力和一些其他的因素，频繁的迁移很少发生。

因此，T必须继续与N1共享CPU0，从而由于频繁的优先获取和上下文交换的限制，导致请求处理延迟增加。

限制邻居并没 有帮助。如图4b所示，只有当相邻应用程序的线程耗尽所有CPU配额时，T才会比以前更频繁地醒来。然而，T仍然不能利用CPU1和CPU2上的幻影CPU时间，因为T仅仅只有一个线程。

现代调度程序的设计至少有两个目标：最大化资源利用

和最大化交互性能。

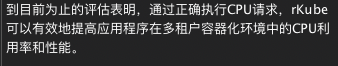
具体来说，尽管目标容器可以保留整个CPU的数量，以便其线程可以在任何时间内运行，但当这样的线程产生阻塞时，OS将把其他线程移到预留的CPU上，使其保持忙碌。因此，（强制共享运行队列）FRS是调度器的负载平衡机制的结果，这是最大化利用率的目标的结果。

类似地，PCT是两个设计考虑组合的结果:1)线程是调度实体，它会有效地偏向具有更多线程的容器，进而是在线程间执行系统范围公平性的结果。2）线程不能无条件抢占其他线程，这反过来是为了减少调度开销而有一个较低边界的时间片的结果。



rKube首先根据目标容器的请求选择一组预留给它们的cpu。

然后，它为所有容器设置cpuset.cpus，以保证所选cpu的运行队列对目标容器是独占的。这样，其他相邻容器就不会被调度到这些cpu上。最终，系统和其他容器将不会受到预留cpu的调度开销的影响，其成本将由有需求的用户支付



结果表明，在保证可预测的性能时，rKube比垂直扩展更有效。仅仅增加资源并不能阻止目标应用程序和邻居线程共享相同的CPU运行队列，从而从目标“窃取”CPU资源。

在执行垂直缩放时，有两种策略：1) 线程数等于CPU请求，从而随着CPU请求的增加而增加；2) 当CPU请求增加时，线程数保持不变。我们在图中将这两种 策略表示为垂直1和垂直2。默认情况下，流集群的线程数 和CPU请求都设置为12个。然后，我们在CPU请求从10到22之间改变CPU请求，在这种情况下，策略“垂直2”保持线程数量12，作为比较，rKube（虚线）

不需要额外的CPU资源

图a和图b显示了归一化的完成时间，越小代表完成越快

a当邻居应用程序处于受限和内存密集型时

b当邻居应用程序处于不受限和内存密集型时

此外，垂直缩放可能会进一步导致性能下降，因为随着垂直缩放产生更多的线程，CPU争用会增加。从图9b中的“垂直1”的结果中可以观察到这一趋势。

水平缩放

例如，要使内存缓存集群能够以更高的吞吐量提供服务，可以增加内存缓存副本的数量，并将负载分发 到所有副本中。

图a和图b中的最小复制品数量分别为7和9。

我们观察到，通过增加复制品的数量，尾部延迟逐渐 减少。另外，在rKube的帮助下，两种邻居的最小复制数为5个，尾部延迟分别为606和613ms。

结果表明，虽然扩展可以提高内存缓存的吞 吐量，但由于邻居容器，延迟也增加了。RKube可以减轻这种影响，为用户提供更好的服务质量