QNX 的调度算法

作为一个硬实时操作系统，QNX 是一个基于优先级抢占的系统。这也导致其基本调度算法相对比较简单。因为不需要像别的通用操作系统考虑一些复杂的“公平性”，只需要保证“优先级最高的线程最优先得到 CPU”就可以了。

基本调度算法

调度算法，是基于优先级的。QNX 的线程优先级，是一个 0-255 的数字，**数字越大优先级越高。**所以，优先级 0 是内核中的 idle 线程。同时，优先级 64 是一个分界岭。就是说，优先级 1 – 63 是非特权优先级，一般用户都可以用，而 64 – 255 必须是有 root 权限的线程才可以设。这个”优先级 64”分界线，如果有必要，还可以通过启动 Procnto 时传 –P <priority> 来改变。

调度算法的对像是线程，而线程在 QNX 上，有大约 20 个状态。（参考 /usr/include/sys/states.h）在这许多状态中，跟调度有关的，其实只有 STATE_RUNNING 和 STATE_READY 两个状态。STATE_RUNNING 是线程当前正在使用 CPU，而 STATE_READY 是等着被执行（被调度）的线程。其他状态的线程，处于某种”阻塞”状态中，调度算法不需要关注。

所以在调度算法看来，整个系统里的线程像这样：

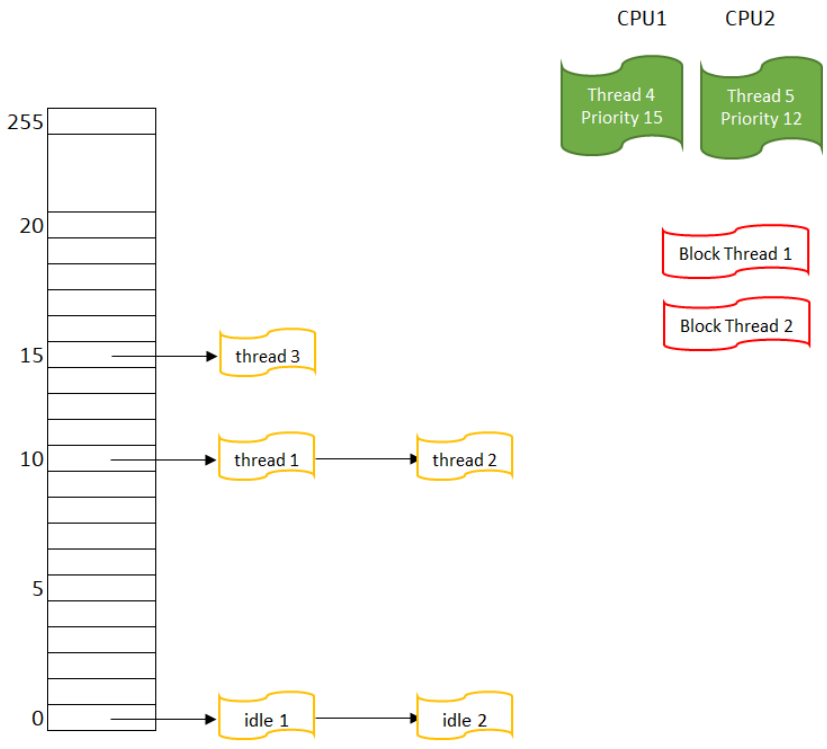

图 1 有两个 CPU 的系统里的线程

这是一个有两个 CPU 的系统，所以可以看到有两个 RUNNING 线程；对于 BLOCK THREAD，它们不参于调度，所以不需要考虑它们的优先级。

调度策略

在 QNX 上实质上只有三种基本调度策略，“轮询”（Round Robin），“先进先出”（First in first out）和 " 零星调度“(Sporadic) 算法。虽然形式上还有一个”其他”，但“其他”跟“轮询”是一样的。这些调度策略，在 /usr/include/sched.h 里有定义。(SCHED_FIFO, SCHED_RR, SCHED_SPORADIC, SCHED_OTHER)

强调一下，调度策略只限于在 READY 队列里的线程，优线级最高的线程有不止一个时，才会用到。如果线程不再 READY，或是有别的更高优先级的线程 READY 了，那就高优先级线程获取 CPU，没有什么策略可言。

“轮询调度“（Round Robin）跟平时生活里排队的情形差不多，晚到的人排在队尾，早到的人排在队首，等到叫号（调度）的时候，队首的人会被先叫到 。如下图所示：

图 2 论询调度示意

- 首先在 CPU 1 上运行的线程 4，被挪入优先级 15 的队列**末尾**
- 然后重新搜索可执行的最高优先级线程，这里有优先级 15 队列上的线程 3 和 4
- 线程 3 因为在队列最前端，它被选择得到 CPU，线程 3 的状态变为 RUNNING，在 CPU1 上执行

可以预期，当下一次调度发生时，线程 3 会被挪入优先级 15 队列末尾，而线程 4 会被调度执行，这样线程 3 和 4 会分别得到 CPU1.

“先进先出”（First in first out）调度则刚好相反，后来的人插在队首，然后在叫号的时候被先叫到。看下图：

图 3 先进先出调度示意

图 4 先进先也

- 首先在 CPU 1 上运行的线程 4，被挪入优先级 15 的队列**队首**
- 然后重新搜索可执行的最高优先级线程，这里有优先级 15 队列上的线程 4 和 3
- 线程 4 因为在队列最前端，它被选择得到 CPU，线程 4 的状态变为 RUNNING，在 CPU1 上执行

可以看到，在这个调度算法下，如果没有别的状态发生，事实上线程 4 就会一直占据 CPU1

如果在优先级 15 上的线程 3 和线程 4 都是 FIFO 会怎样？按上面的描述，线程 3 还是始终无法获得 CPU1，因为线程 4 每次都会插在 3 的前面，再调度就又是 4 获得 CPU1。除非线层 4 进入了阻塞状态（从而不在 READY 队列里了），那么线程 3 才能获得 CPU。

“零星调度”（Sporadic）算法比较特殊，它比较适合长时间占用 CPU 的线程。它的基本设计思想是给一个线程准备两个优先级，”前台“优先级比较高，”后台“优先级稍微底一点。如果线程在高优先级连续占用 CPU 超过一定时间后，线程会被强行降到”后台“低优先级上（这时线程能不能占用 CPU 取决于系统中有没有比”后台“优先级高的别的线程了）；然后线程在低优先级上经过了一段时间后，会重新被调回高优先级。

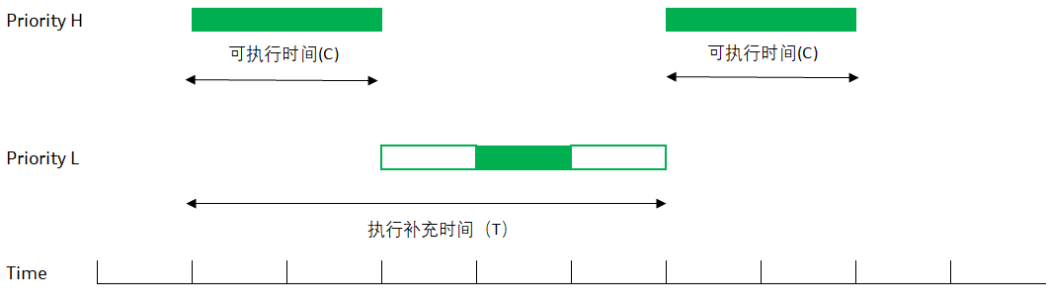

图 5 零星调度示意

上图是一个零星调度线程的示意。

- 开始的时候，线程在比较高的（正常）优先级 H 上运行，一直到用完预先分配给零星调度的时间用完（sched_ss_init_budget）
- 这时，线程会被自动调整为低优先级 L（sched_ss_low_priority）；一旦被调低，线程也可能运行（如果优先级 L 依然是系统里最高优先级的线程），也可能无法运行呆在 READY 队列里（系统里有比 L 更高的优先级）
- 不管线程有没有执行，从最开始运行时间点算起，当线程“执行补充时间”（sched_ss_repl_period）过了以后，线程的优先级被重新提到优先级 H，并试图取得 CPU 来。

“零星调度”看上去比较“公平”，但是实际在用 QNX 的项目中，这个调度算法很少被用户用到。主要是因为一般来说在 QNX 上很少有线程能够“连续占用 CPU”的。而且当系统变得复杂，线程数成百上千后，这种上下调优先级的做法，很容易出现别的后遗症。

什么时候会发生调度?

上面介绍了 QNX 支持的几个调度算法。那么，什么时候才会发生调度呢?

QNX 的设计目标是一个硬实时操作系统，所以，保证最高优先级的线程在第一时间占据 CPU 是很重要的。考虑到线程的状态都是在内存中进行变化的（都是因为线程进行了某个内核调用后变化的），所以 QNX 在每次从内核调用退出时，都会进行一次线程调度，以保证最高优先级的线程可以占据 CPU。

得益于微内核结构，QNX 的内核调用通常都非常短，或者说，每一个内核调用，都能够比较确定地知道要花多少时间。而且，因为微内核系统的基本就是进程间通信，所以在 QNX 上，一段程序非常容易进入内核并进行线程状态切换，很少能有长时间占满 CPU 的，在实际系统上测，现实上很少能有线程执行完整个时间片的。

举个例子，哪怕程序里只写一个 printf(“Hello World!\n”); 可是在 libc 库里，最后这个会变成一个 IO_WRITE 消息，MsgSend() 给控制台驱动；这时，在 MsgSend()这个内核调用里，会把 printf() 的线程置为阻塞状态（REPLY BLOCK），同时会把控制台驱动的信息接收线程（从 RECEIVE BLOCK）改到 READY 状态，并放入 READY 队列。当退出 MsgSend() 内核调用时，线程调度发生，通常情况下（如果没有别的线程 READY 的话）控制台驱动的信息接收线程被激活，并占据 CPU.

如果用户写了一个既不内核调用，也不放弃 CPU 的线程会怎么样？那时候，时钟中断会发生，当内核记时到线程占据了一整个时间片（QNX 上是 4ms）后，内核会强制当前线程进入 READY，并重新调度。如果同一优先级只有这一个线程（这是优先级最高线程），那么调度后，还是这个线程获取 CPU。如果同一优先级有别的线程存在，那么根据调度算法来决定哪个线程获得 CPU。

另一种常见情况是，由于某些别的原因导致高优先级线程被激活，比如网卡驱动中断导致高优先级驱动线程 READY，所设时钟到达导致高优先级线程从阻塞状态返回 READY 状态了，当前线程开放互斥锁之类的线程同步对象，导致别的线程返回 READY 状态了。这些，都会在从内核调用退出时，进行调度。

中断与优先级

上面提到如果用户线程长期占有 CPU，时钟中断会打断用户线程。细心的读者或许会有疑问，那中断的优先级是多少呢?

答案是在 QNX 这样的实时操作系统里，“硬件中断”永远高于任何线程优先级，哪怕你的线程优先级到了 255，只要有中断发生，都要让路，CPU 会跳转去执行中断处理程序，执行完了再回归用户线程。事实上，能够快速稳定地响应中断处理，是一个实时操作系统的硬指标。

我们这里说的是“硬件中断”，就是说，当外部设备，通过中断控制器，向 CPU 发出中断请求时，无论当时 CPU 上执行的线程优先级是什么，都会先跳转到内核的中断处理程序；中断处理程序会去中断控制器找到具体是哪一个源发生了中断（中断号），并据此，跳转到该中断号的中断处理程序（通常是硬件驱动程序通过 InterruptAttach() 挂接的函数）。在这个过程中，如果当前 CPU 正在处理另一个中断，那么这时，会根据中断的优先级来决定是让 CPU 继续处理下去（当前中断进入等待）；或者发生中断抢占，新中断的优先级比旧中断高，所以跳转新中断处理。

当然，实际应用中，特别是微内核环境下，考虑中断其实只是中断设备给出的一个通知，对这中断的响应并不需要真的在中断处理中进行，驱动程序可以选择在普通线程中处理，QNX 上有 InterruptAttachEvent() 就是为了这个设计的。通常这里的“事件”会是一个“脉冲”，也就是说，当硬件中断发生，内核检查到相应中断绑定了事件。这时，不会跳转到用户中断处理程序，而是直接发出那个脉冲，以激活一个外部（驱动器中）线程，在这线程中，做设备中断所需要的处理。这样做，虽然稍微增加了一些中断延迟，但也带来了不少好处。首先，这个外部线程同普通的用户线程一样，所以可以调用任何库函数，而中断服务程序因为执行环境的不同，有好多限制。其次，因为是普通用户线程，就可以用线程调度的方法规定其优先级（脉冲事件是带优先级的），使不同的设备中断处理，跟正常业务逻辑更好地一起使用。

多 CPU 上的线程调度

现在同步多处理器（SMP）已经相当普及了。在 SMP 上，也就是说当有多个 CPU 时，我们的调度算法有什么变化呢？比如一个有 2 个 CPU 的系统，首先肯定，系统上可执行线程中的最高优先级线程，一定在 2 个 CPU 上的某一个上执行；那，是不是第二高优先级的线程就在另一个 CPU 上执行呢？

虽然直觉上我们觉得应该是这样的（系统里的第一，第二高优先级的线程占据 CPU1 和 CPU2），但事实上，第二高优先级的线程占据 CPU2 这件事，并不是必要的。实时抢占系统的要求是最高优先级”必须“能够抢占 CPU，但对第二高优先级并没有规定。拿我们最开始的双 CPU 图再看一眼。

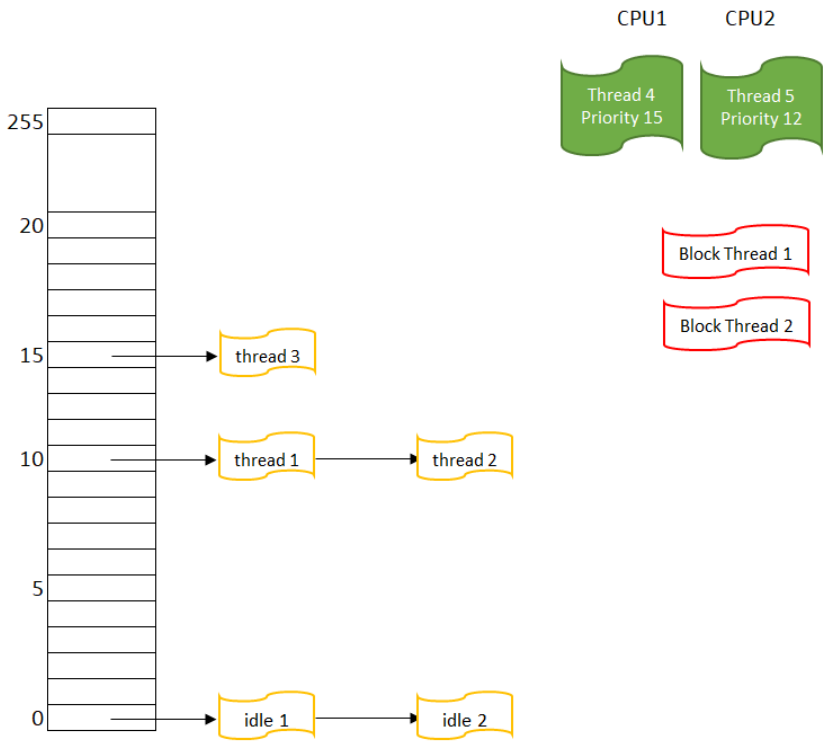

图 6 有两个 CPU 的系统里的线程

线程 4 以优先级 15 占据 CPU1 这是毫无疑问的，但线程 5 只有优先级 12，为什么它可以占据 CPU2，而线程 3 明明也有优先级 15，但只能排队等候，这是不是优先级倒置了？其实并没有，如上所述，系统确实保证了“最高优先级占据 CPU”的要求，但在 CPU2 上执行什么线程，除了线程本身的优先级以外，还有一些别的因素可以权衡，其中一个在 SMP 上比较重要的，就是“线程跃迁”。

“线程跃迁”指的是一个线程，一会儿在 CPU1 上执行，一会儿在 CPU2 上执行。在 SMP 系统上，线程跃迁而导致的缓存清除与重置，会给系统性能带来很大的影响。所以在线程调度时，尽量把线程调度到上次执行时用的 CPU，是 SMP 调度算法里比较重要的一环。上述例子中，很有可能就是线程 3 上一次是在 CPU1 上执行的，而线程 5 虽然优先级比较低，很有可能上一次就是在 CPU2 上执行的。

实际应用中，因为 QNX 的易于阻塞的特性，其实大多数情况下，还是符合”第一，第二高优先级线程在 CPU 上执行“的。只是，如果你观察到了上述情形，也不需要担心，设计上确实有可能不是第二高优先级的线程在运行。

另一个多处理器上常见的应用，是线程绑定。在正常情况下，把可执行线程调度到哪一个 CPU 上，是由操作系统完成的。当然操作系统会考虑“线程跃迁”等情形来做决定。但是，QNX 的用户也可以把线程绑定到某一个（或者某几个）CPU 上，这样操作系统在调度时，会考虑用户的要求来进行。绑定是通过 ThreadCtl() 修改线程的 “RUNMASK” 来进行的，如果你有 0,1,2,3 总共 4 个 CPU，那么 0x00000003 意味着线程可以在 CPU0 和 CPU1 上执行，具体例子可以参考 ThreadCtl()函数说明。更简单的办法，是通过 QNX 特有的 on 命令的 –C 参数来指定，这个指定的 runmask，还会自动继承。所以你可以简单的如下执行：

 # on –C 0x00000003 Navigation &

 # on –C 0x00000004 Media &

 # on –C 0x00000008 System &

这样来把不同的系统部署到不同的 CPU 上。

这样做的好处当然是可以减少比如因为系统繁忙而对导航带来的影响，但不要忘了，另一面，如果所有 Media 线程都处于阻塞状态，上述绑定也限制了导航线程使用 CPU2 的可能，CPU2 这时候就会空转（执行内核 idle 线程）。

自适应分区调度算法

前面我们提到过，在讨论优先级调度时，只是讨论当有多个优先级相同的线程时，系统怎样取舍。优先级不一样时，肯定是优先级高的赢。但是 " 高出多少 " 并不是一个考量因素。两个线程，一个优先级 10，另一个优先级 11 的情况，和一个 10，另一个 40 的情况是一样的。并不会因为 10 和 40 差距比较大而有什么不同。

假如我们有红蓝两个线程，它们的优先级一样，调度策略是 RR，两个线程都不阻塞，那么在 10 时间片的区间里，我们看到的就是这样一个执行结果：

也就是说，各占了 50%的 CPU。但只要把蓝色线程提高哪怕 1，执行结果就成了下面这样。

这种“非黑即白”的情形，是实时系统的基本要求（高优先级抢占 CPU）。但是当然，现实情况有时候比较复杂。比如 “HMI 渲染” 是需要经常占据 CPU 的一个任务（这样画面才会顺畅），但“用户输入”也是需要响应比较快的（不然用户的点击就会没有反应）。如果”用户输入”的优先级太高的话，那用户拖拽时，画面就

会卡顿甚至没有反应？反之，如果”HMI 渲染“的优先级太高，那么有用户输入时，因为处理程序优先级低而造成用户输入反应慢。通常情况下，需要有经验的系统工程师不断调整这两个任务的优先级（因为优先级继承与传统，一个任务可能涉及到多个线程），来达到系统的最优。那么，有没有别的办法呢？

分区调度

传统上，有一种“分区调度”的方法，今天还有一些 Hypervisor 采取这个办法。这个想法很简单，就是把 CPU 算力隔成几个分区，比如 70%，30%这样，然后把不同线程分到这些分区里，当分区里的 CPU 预算被用完以后，那个分区里所有可执行线程都会被”停住“，直到预算恢复。

假设我们把红线程放入 70%红色分区，蓝线程放入 30%蓝色分区，然后以 10 个时间片为预算滑动窗口大小，各线程具体就会如下图占据 CPU：

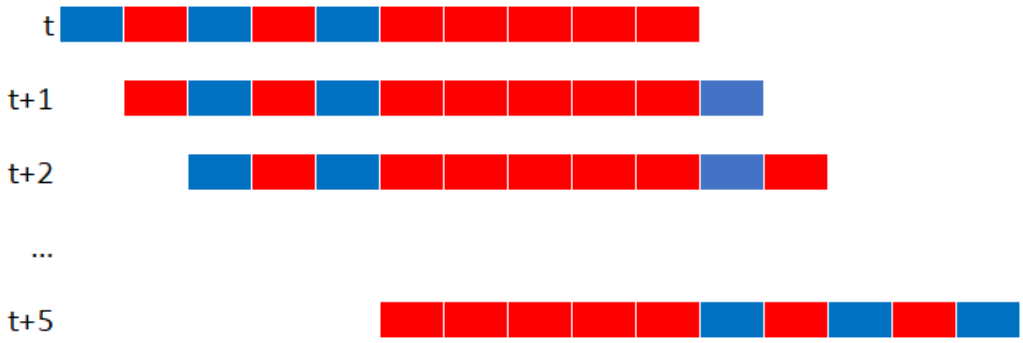

图 7 分区调度算力全满示意

在前 6 个时间片中，蓝红分区分别占据 CPU，注意在第 7 个时间片时，虽然蓝分区中线程跟红分区中线程有相同的优先级，虽然调度策略是轮回，应该轮到蓝线程上了，但是因为蓝线程已经用完了 10 个时间片里的 3 个，所以系统没有执行蓝线程，而是继续让红线程占据 CPU，一直到第 8 第 9 和第 10 个时间片结束。

10 个时间片结束后，窗口向右滑动，这时我们等于又多了一个时间片的预算，在新的 10 个时间片中，蓝线程只占了两个(20%)，这样，新的第 11 个时间片，就分给了蓝分区。

同理再滑动后，第 12 个时间片，分给红线程；一直到 17 个时间片时，同样的事情再度发生，蓝分区线程又用完了 10 个时间片里的 3 个，而被迫等待它的预算重新补充进来。

综上，在任意一个滑动窗口中，蓝色分区总是只占 30%，而红色分区却占了 70%。QNX 的自适应分区调度，跟上面这个是类似的。只是传统的分区调度，有一个明显的弱点。

想一下这个情况，如果红线程因为某些情况被阻塞了，会发生什么呢？

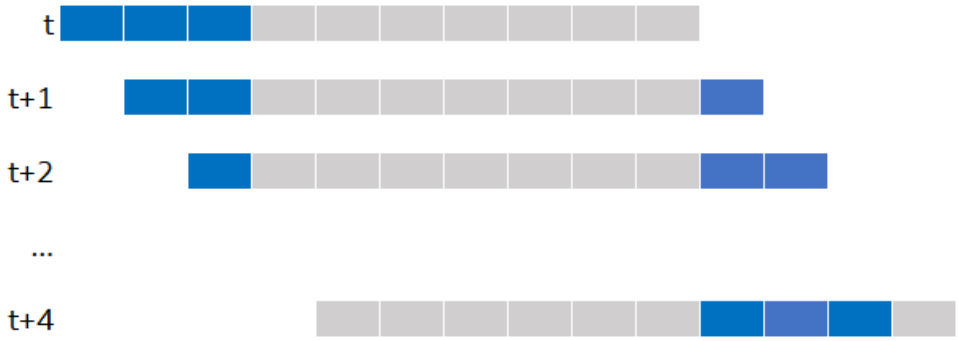

图 8 分区调度算力有富余示意

对，蓝线程是唯一可执行线程，所以它一直占据 CPU。但是，当 3 个时间片轮转之后，因为蓝分区只有 30%的时间预算，它将不再占据 CPU，而因为红线程无法执行，接下来的 7 个时间片 CPU 处于空转状态（执行 Idle 线程）。

一直到时间窗口移动，那时，因为蓝分区只占用了 20%的算力，所以它再次占据 CPU……

所以你也看到了，在传统的分区调度里，当一个分区的算力有富裕的时候，CPU 就被浪费了。

自适应分区调度

QNX 在传统的分区调度上，增加了“自适应”的部份。其基本思想是一样的，给算力加分区，然后把不同的线程分到分区里。这样，当所有的线程都忙起来时，你会发现情况跟图 7 是一样的。但是当分区算力有富裕时，“自适应”允许把多出来的算力”借”给需要更多算力的分区。

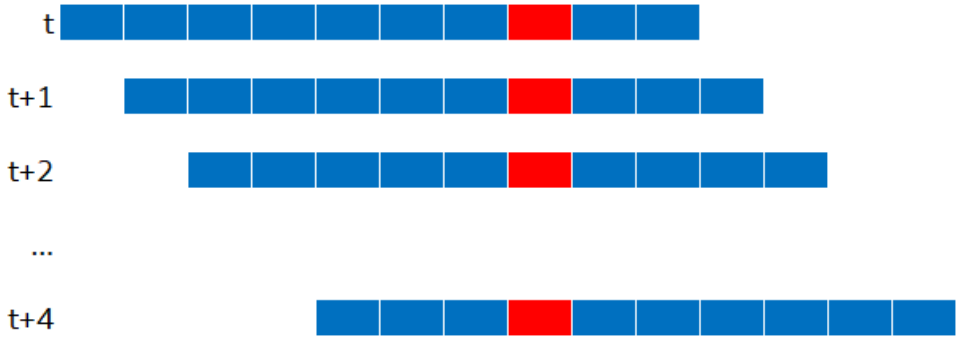

图 9 自适就分区算力有富裕示意

如上，当蓝色分区里的线程消耗完了他自己的分区预算后，自适应分区会把有富裕算力的红色分区的预算，借给蓝色分区，蓝分区内线程得以继续在 CPU 上运行。注意，在第 8 个时间片时，红色分区需要使用 CPU，蓝色分区立即让路，把 CPU 让给红色分区。而当红色分区里的线程被阻塞住以后，蓝色分区线程继续使用 CPU。

自适应分区似乎确实带来了好处，但是也带来了一些潜在的问题，需要在系统设计的时候做好决定。

自适应分区调度与线程优先级

你可能会好奇，在分区调度的系统里，线程的优先级代表了什么？

答案取决于各个分区对各自算力的消耗情况。我们假设蓝色分区里的线程优先级比较高，红色的优先级比较低，当两个分区都有预算时，内核会调度（所有分区里的）最高优先级线程执行。如果系统一直不是很忙，那么不论分区，永远是有最高优先级的线程得到 CPU，这个，跟一个标准的实时操作系统是一致的。

当两个分区中某一个有预算时（意味着那个分区中所有的线程都不在执行状态），那么多出来的 CPU 算力会被分给另一个分区，另一个分区中的最高优先级线程（虽然用完了自己分区的预算，但得到了别的分区的算力），继续占据 CPU。这个，也是跟实时操作系统是一致的。

比较特殊的情况是，当两个分区都没有预算，都需要占据 CPU 时，这时，蓝色线程虽然有较高的优先级，但因为分区算力(30%)被用完，面且没有别的算力可以“借”，所以它被留在 READY 队列中，而比它优先级低的红色线程得以占据 CPU。

自适应分区调度富裕算力分配

我们上面的例子只有两个分区，考虑这样一个例子。假设我们现在有 A (70%)，B (20%)，C (10%) 三个分区，A 分区没有可执行线程，B 分区有个优先级为 10 的线程，C 分区有个优先级为 20 的线程。我们知道 A 分区的 70%会分配给 B 和 C，但具体是怎么分配的呢？

如上所述，当预算有富裕时，系统挑选所有分区中，优先级最高的线程执行，也就是说 C 分区中的线程得到运行。在一个窗口以后，你会发现 A 的 CPU 使用率是 0%，B 是 20%，C 则达到了 80%。也就是说 A 所有的富裕算力，都给了 C 分区（因为 C 中的线程优先级高）。

也许，在某些时候，这个不是你所期望的。也许 C 中有一些第三方程序你无法控制，你也不希望他们偷偷提高优先级而占用全部富裕算力。QNX 提供了 SchedCtl()函数，可以设 SCHED_APS_FREETIME_BY_RATIO 标志。设了这个标志后，富裕算力会按照各分区的预算比例分配给各分区。上面的例子下，最后的 CPU 使用率会变成 A 是 0%， B 是 65%，而 C 是 35%。A 分区富裕的 70% 算力，按照大约 2: 1 的比例，分给了分区 B 和 C。

“关键线程”与“关键分区”

在实际使用中，有一些重要任务，可能需要响应，不论其所在的分区还有没有算力。比如一个紧急中断服务线程，不管分区是不是还有预算，都需要响应。为了解决这种情况，在 QNX 的自适应分区调度里，除了给分区分配算力预算以外，还允许有权限的用户为分区分配“关键响应时间”，并把特定线程定义为“关键线程”。

当一个“关键线程”需要执行时，如果线程所在分区有预算，它就直接使用所在分区预算就好，如同普通线程；如果所在分区没有预算了，但是别的分区还有预算，那么“自适应”部份会把别分区的预算拿过来，并用于关键线程，这个跟普通的自适应分区调度一样。

只有当系统里所有分区都没有预算了，而有一个关键线程需要运行，而且线程所在的分区已经预先分配了关键响应预算，那么线程允许“突破”分区的预算，使用“关键响应预算”来执行。在 QNX 里，一个关键

线程消耗的时间，从退出 RECEIVE_BLOCK 开始，到下一次进入 RECEIVE_BLOCK。而且，关键线程的属性是可传递的，如果关键线程在执行中，给别的线程发送了消息，那个线程也会变成关键线程。

总的来说，关键线程是用来保证关键任务不会因为系统太忙而无法取得 CPU 时间。即使所有的分区都被占满了，至少还有“关键响应时间”可供关键线程来使用。当然，一个系统里不应该有太多的关键线程和关键响应时间。理论上，假设所有的线程都是关键线程，那么整个系统其实就变成了一个普通的按优先级调度的实时系统，所有的分区和预算都不起作用了。

在最紧急的情况下，关键线程可以使用“关键响应时间”来完成它的任务。如果“关键响应时间”还是不够，会怎么样？这个是系统设计问题，在设计系统的时候，你就应该为关键线程分配它能够完成任务所需要的最大时间。如果依然发生“关键响应时间”不够的状况（被称为“破产”状态），这个就是一个设计错误了。

关键线程的破产

如上所述，关键线程的破产是一个设计问题。或者线程完成的任务并不那么“关键”，或者设计时给出的预算不够。这种情况下需要重新审视整个系统设计（因为系统在某些情况下无法保证关键任务在预定时间内完成）。QNX 在自适应分区里提供了侦测到关键线程破产时的多种响应办法，可以是强行忽视，或者重启系统，或者由自适应分区系统自动调整分区的预算。

自适应分区继承

想像这个场景，文件系统在 System 分区里，但另一个 Others 分区里的第三方应用拼命调用文件系统，很有可能造成 System 分区的预算耗尽；这样，首先可能导致别的应用无法使用文件系统；更严重的，可能是 System 分区里别的系统，比如 Audio 也无法正常工作。这个，显然是自适应分区系统带来的安全隐患。

解决办法，就是跟优先级在消息传递上可以继承一样，分区也是可以继承的。文件系统虽然分配在 System 系统里，但根据它响应的是谁的请求，时间被记到请求服务的线程分区里。这样，如果一个第三方应用拼命调用文件系统，最多能做的，也只是消耗它自己的分区，当他自己分区的预算被耗尽时，影响它自己的 CPU 占用率。

自适应分区的小结

自适应分区有一些有趣的用法。比如我们常常被要求“系统需要保留 30%的算力”。有了自适应分区，就可以建一个有 30%预算的分区，在里面跑一个 for (;;); 这样的死循环程序。这样，剩下的系统就只有 70%的算力了，可以在这个环境下检验一下系统的性能和稳定性。

自适应分区的具体操作方法，可以参考 QNX 的文档。不同版本的 QNX 有稍微不同的命令行，但基本设计是一样的。这篇文章只是介绍了自适应分区的基本概念，实际使用上，还是有许多细节需要考虑的，真的要使用，还是需要详细参考 QNX 对应文档。