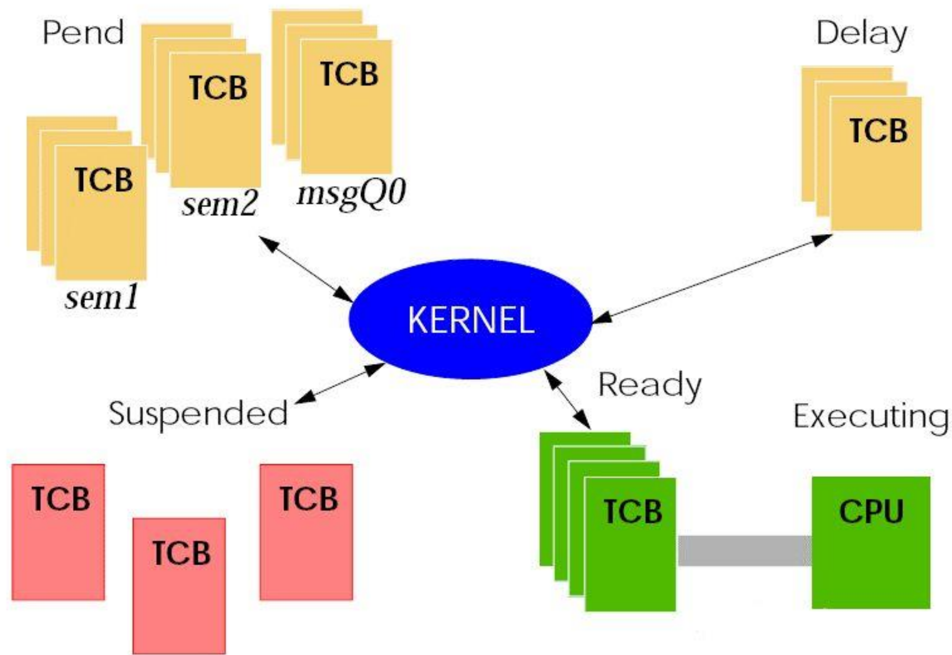
VxWorks Task 的调度策略

我们知道 VxWorks 是一个典型的 Multitasking OS(多任务操作系统)，每个 Task 都可能有多种状态，其中处于 Ready 状态的 Task 一旦拿到 CPU，就可以执行了。不过 CPU 的数量再多，也不太可能比 Task 的数量多。也就是说，总会有 Task 即使进入了 Ready 状态，也抢不到 CPU，还是不能执行。



那么当有了空闲的 CPU 时，到底是分配给哪个 Ready 的 Task 呢？这就涉及到系统的 Scheduling(调度策略)了。如果硬件是 Multicore Processors，VxWorks 运行的又是 SMP(Symmetric MultiProcessor)模式，那情况稍微复杂些，咱们以后再聊。今天先看看 UP(UniProcessor)模式的情况。

Priority

不管哪种策略，都基于一个很重要的概念，即 Task 的 Priority(优先级)。Task 的优先级是在创建时指定的，而且后期也可以动态调整，不过不建议在应用中频繁修改 Task 的优先级，因为可能带来了调度的不确定性。优先级的取值范围是 0-255，其中 0 的优先级最高，255 的优先级最低。具体每个 Task 的优先级取值是没有限制的，只是建议应用 Task 的优先级不小于 100，驱动 Task 的不大于 99。

Priority-based Preemptive Scheduling

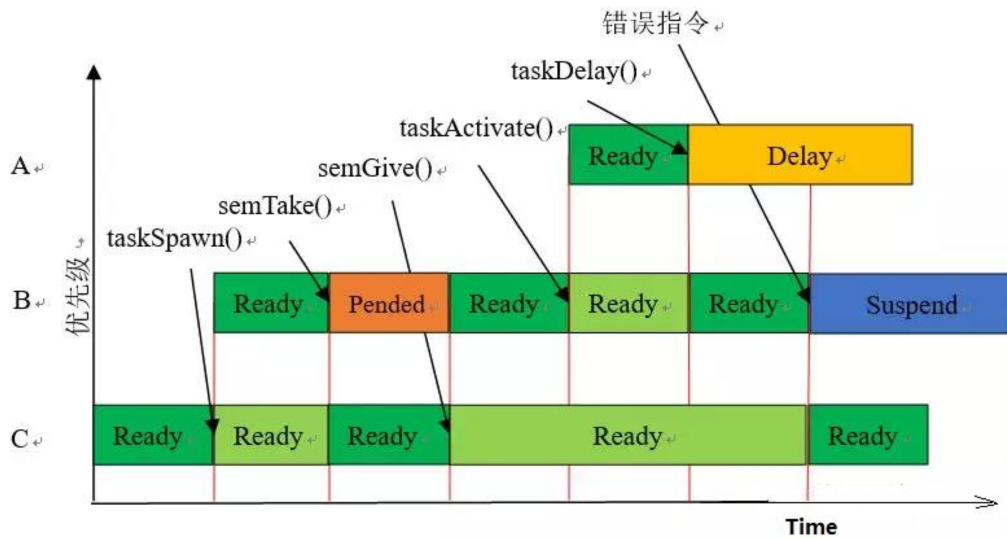
VxWorks 传统的调度策略是基于优先级抢占，这也是默认的策略，在 Vxworks Image Project 中对应的 component 是 INCLUDE_VX_TRADITIONAL_SCHEDULER

Components

Component Configuration

Description	Name
kernel components (default)	**FOLDER_KERNEL**
kernel scheduler selection (default)	**SELECT_KERNEL_SCHEDULER**
POSIX thread scheduler in RTPs	INCLUDE_POSIX_PTHREAD_SCHEDULER
VxWorks traditional kernel scheduler (default)	**INCLUDE_VX_TRADITIONAL_SCHEDULER**
user-specified kernel scheduler	INCLUDE_CUSTOM_SCHEDULER
memory allocator selection	**SELECT_MEM**
spinLock libraries API	**API_SPINLOCK**

在这种策略下，如果一个 Ready 状态的 Task 的优先级比正在运行的 Task 的优先级高，就会发生抢占。



例如上图中，开始时只有 Task C 处于 Ready 状态，那么就是 C 占用 CPU 来执行，当 Task B 通过某种原因(例如 taskSpawn())进入了 Ready 状态，就会发生抢占，即 Kernel 立即保存 C 的上下文，切换到 B 的上下文。只有当 B 通过某种原因(例如申请信号量，进入 Pended 状态)退出 Ready 状态，Kernel 才能再次切换到 C 。因此，永远都是 Ready 状态中，优先级最高的 Task 在执行。

Kernel 为每种状态的 Task 都维护着一个队列，其中 Ready 队列又有多个 List，相同优先级的 Task 处于同一个 List。当某一优先级的 Task 可以执行时，都是该 List 最前端的 Task 来执行。Task 在队列中的位置可能发生变化，情景如下

- Task 被其它高优先级的 Task 抢占后，还保持在其 List 的头部
- Task 退出 Ready 队列(例如进入 Pended、Delayed、Suspended 等)后，又返回 Ready 队列的，排在其原 List 的尾部
- Task 的优先级被 taskPrioritySet()修改后，排在新 List 的尾部
- Task 的优先级被互斥信号量的继承策略临时提高后，又恢复原有优先级的，排在其原 List 的尾部

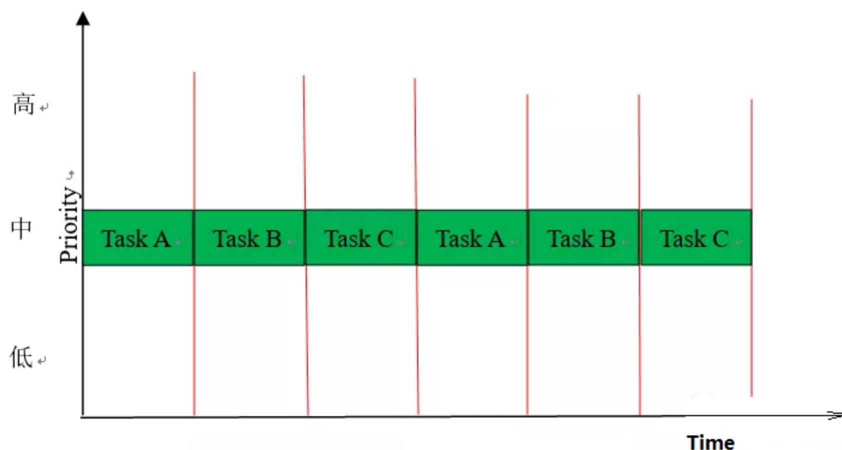
函数 taskRotate()可以把一个 Task 从其 List 的头部移到尾部。例如 taskRotate(100)就是把优先级 100 的 List 的头部的 Task 移到该 List 的尾部。如果正在执行的 Task 要把自己移到当

前 List 的尾部，可以直接调用 taskRotate(TASK_PRIORITY_SELF)。我们在《Task 之任务的控制》中，也介绍过一种类似的方法，谁还记得？

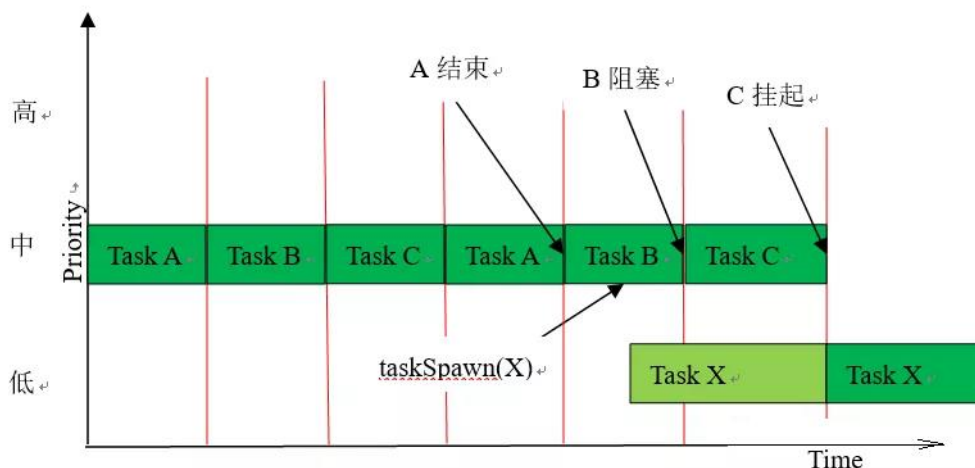
```
01: STATUS taskRotate (int priority);
```

Round-Robin Scheduling

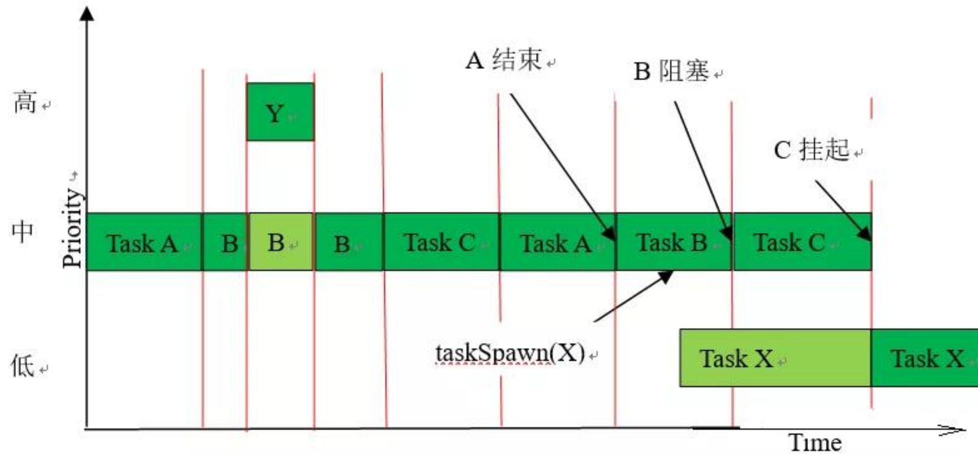
优先级抢占策略有一个缺点，就是当多个优先级相同的 Task 中的某个占用 CPU 后，如果一直不退出 Ready 状态的话，其它同优先级的 Task 就一直得不到执行。刚刚提到的 taskRotate()可以缓解这个问题，不过这会对应用开发人员有一定的要求。其实 Kernel 还提供了一种调度策略，时间片轮转策略（Round-robin scheduling），来解决这个问题。时间片轮转策略也是一种传统策略，不过默认没有使能。这样策略就是给同优先级 Task 分配一定的时间片(Time-Slicing)来共享 CPU。同一 List 中，每个 Task 执行特定的时间片后，就把 CPU 让给下一个 Task，自己移到该 List 的尾部。如果时间片还没有执行完，同一 List 中的 Task 是不能抢占它的，除非它主动让出 CPU 或退出 Ready 队列。



上图中，3 个 Task （A、B、C)的优先级相同，它们依次执行，每次都消耗同样的时间片。如果还有其它优先级不同的 Task，是不会影响 Task A、B、C 的时间片长度的。



上图中 Task B 创建了低优先级的 Task X 之后，X 并不会影响 A、B、C 的时间片。因为 X 的优先级低一些，即使它已经在 Ready 队列了，也要等 A、B、C 全部退出 Ready 状态后，它才有可能占用 CPU 执行。



而如果有高优先级的 Task 进入 Ready 状态时，还是会立即发生抢占。上图中 Task B 被高优先级的 Task Y 抢占后，会排在同优先级 List 的头部。等到 Y 退出 Ready 队列后，B 会继续执行之前剩余的时间片。猜一猜 Kernel 怎么知道 B 的时间片还剩余多少呢？
从上面几个例子可以看到，优先级抢占策略是始终存在的，不同优先级的 Task 之间一直都会遵循抢占的策略。时间片轮转策略只在同优先级的 Task 之间生效。而且 VxWorks 的这种时间片轮转默认还是关闭的。

```
01: STATUS kernelTimeSlice (int ticks);
02: STATUS kernelTimeSliceGet (unsigned long *pTimeSlice);
```

要想打开时间片轮转策略，只需要调用函数 kernelTimeSlice()，其参数为 0 就表示关闭该策略，参数大于零时就表示打开策略并设置时间片的长度。这个 ticks 指的是系统时钟的频率，因此 kernelTimeSlice(sysClkRateGet()/2)表示将时间片设置为半秒。参数的数据类型使用的是 int，这是因为要兼容老版本的原因，而代码中实际是按照 unsigned long 来操作的。
可以看到，VxWorks 中所有 Task 的时间片都是相同的。而在有的操作系统中，优先级不同的 Task 会使用不同长度的时间片。例如 Linux 中默认的 SCHED_OTHER 策略，高优先级的进程/线程会占用多一些的时间片。
那 VxWorks 的这个时间片轮转策略在什么情况下才需要打开呢？一般是应用程序中有多个 Task 的优先级相同，而且它们还会同时长时间的处于 Ready 状态。事实上，这种场景出现的并不多。还有一点，这个策略尽量不要在运行时动态改变，不然就给 Kernel 的调度带来了不确定性，这可不是实时系统希望看到的。如果必须要设置时间片的话，最好放在系统启动之后，应用启动之前。

Task Locks
VxWorks 在 UP 模式里还提供了一种特殊的机制，taskLock()/taskUnlock()，可以让当前 Task 不被任何其它 Task 抢占。一个 Task 在调用 taskLock()之后，调用 taskUnlock()之前，只要它还是 Ready 状态，优先级再高的 Task 也无法抢占它；如果打开了时间片轮转，那么它

的时间片计数也不会增加。这种状态只有调用了 taskUnlock()才会取消。如果在 taskLock()和 taskUnlock()中间，这个 Task 阻塞或挂起了，那 Kernel 就恢复原有的调度策略，一旦这个 Task 返回 Ready 队列，就会再次禁止对它的抢占。而且 taskLock()/taskUnlock()还是可以嵌套使用的，不过 Unlock 的次数要一致。

```
01: extern STATUS taskLock(void);
02: extern STATUS taskUnlock(void);
03:
04: void bbb()
05:     {
06:     (void)taskLock();
07:     .
08:     . /* critical region of code that cannot be interrupted */
09:     .
10:     (void)taskUnlock();
11:     }
12:
13: void aaa()
14:     {
15:     (void)taskLock();
16:     bbb();
17:     (void)taskUnlock();
18:     }
```

可以看到，调用 taskLock()之后，禁止了所有 Task(包含与当前 Task 没有关联的高优先级的 Task)的抢占，这就影响了系统的实时响应。因此这种机制尽量少用，而且 taskLock()和 taskUnlock()之间的代码要尽量的简短，不要有耗时的操作。迫不得已的时候，还可以考虑使用互斥信号量来代替。
还有一点，taskLock()不能阻止 Interrupt 的抢占。必要的话，可以加上 intLock()/intUnlock()的组合，把 Interrupt 的抢占也禁掉。

Other Scheduling
除了默认的优先级抢占和时间片轮转，VxWorks 系统还为 RTP 里的 POSIX thread 提供了 POSIX threads scheduling，INCLUDE_POSIX_PTHREAD_SCHEDULER。咱们在介绍 RTP 时，再介绍这种策略。
另外，VxWorks 还提供了用户自定义框架，INCLUDE_CUSTOM_SCHEDULER。我们可以添加自己的调度策略。