**https://blog.csdn.net/redseazhaojianertao/article/details/53843300**

**VxWorks内核解读-2**

2016年12月23日 19:53:20 [redseazhaojianertao](https://me.csdn.net/redseazhaojianertao" \t "_blank) 阅读数：1046

**本文讨论Wind内核的设计思想，正如前文所述，VxWorks的Wind内核采用可裁剪的微内核设计，具有多任务并发执行、可抢占的优先级调度、可选的时间片调度、任务间通信和同步机制、快速的上下文切换，低中断时延、快速的中断响应、支持中断嵌套、支持256个优先级、支持优先级继承、以及任务删除保护机制，Wind内核运行在特权模式，不使用陷阱指令和跳转表，所有的系统调用均采用函数调用的形式实现。**

Wind内核是一种强实时操作系统内核，但和当前其它成熟的操作系统一样，并不是“硬实时”的操作系统。所谓“硬实时”是指当某种事件发生时，系统必须在预定时间之内，或者说在限期之前做出反应，否则就会造成灾难性的后果。具有这一特征的操作系统要做到对于所提交的每一项作业及其时间要求，或者做出承诺，或者立即拒绝(以使提交者可以考虑采取其他措施)，对于做出的承诺则保证其实现。提交给硬实时系统的作业也可以是不带时间要求的，但是对这些作业的承诺当然也就不包含时间的因素，那将当作后台作业来对待。

VxWorks 没有使用基于时间限（deadline）的调度算法，也就是不接受任何作业提交的时间要求。为了提高实时性，目前采用的方法都是优先级抢占式调度，使得高优先级的任务的到优先执行。这样，在一定计算资源前提下，通过恰当的任务划分和设定任务优先级可以满足实时性的要求。

备注：网上有人说VxWorks内核是一种硬实时内核，这是不准确的。准确的说VxWorks系统是实时系统，但是它的实时性是在一定计算资源前提下，通过恰当的任务划分和设定任务优先级来实现的，其它系统比如uC/OS系统，FreeRTOS都是如此。

**2.1 Wind内核结构**

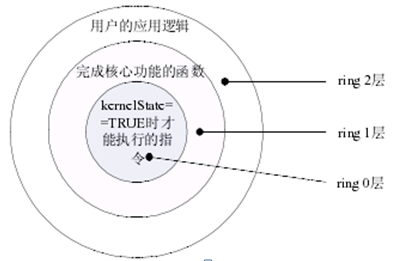
为了提高系统的实时性，大多数操作系统都提供了各种机制，如可抢占的内核、中断的后半段处理等。VxWorks 为了提高实时性，构造成了一个带有微内核的层次结构。

在计算机的早期，操作系统大部分是一个统一的实体(monolithic)。在这样的系统中，提供不同功能的模块是独立考虑的，如处理器管理、内存管理、文件管理等，较少考虑模块间的关系。这样的操作系统结构清晰，构造简单，但是由于操作系统本身很复杂，在这种大粒度的操作系统中难以划分可抢占部分和不可抢占部分的边界，难以避免有冗余的操作在操作系统的不可抢占部分中执行，造成系统的实时性比较差，不适合实时应用环境。

而带有微内核的层次结构的操作系统则较好的解决了这个问题。在这样的操作系统中，以内核作为层次结构的起点，每一层功能对较低层进行封装。内核仅需要包含最重要的操作指令，提供高层软件与下层硬件的抽象层，形成操作系统其它部分所需的最小操作集。这样就可以比较容易的精确确定可抢占部分与不可抢占部分的边界，减少了需要在内核不可抢占部分执行的操作，有利于实现更快的内核抢占，提高系统的实时性。

操作系统的最好的内部结构模型是一个层次性的结构，最底层是内核。这些层次可以看成为一个倒置的金字塔，每一层都建立在较低层的功能之上。 内核仅包含一个操作系统执行的最重要的底层功能。正象一个统一结构的操作系统，内核提供了在高层软件与下层硬件之间的抽象层。然而，内核仅提供了构造操作系统其他部分所需的最小操作集。

VxWorks 的Wind内核就是这样的一个有利于构造层次结构的微内核，它由 kernelLib库、taskLib库、semLib库、tickLib库、wdLib库、schedLib库、workQLib库、windLib库、windAlib库、semAlib库、以及workQAlib库组成的。其中其中kernelLib库、taskLib库、semLib库、tickLib库、以及wdLib库组成了VxWorks内核的基本功能，同时也是Wind内核底层最基本、最核心的函数。在这样的内核中，容易实现很强的实时性；而对内核高层次的封装则使得用户在开发 VxWorks 的应用程序时，只需要调用顶层的函数，不必关心底层的实现，程序设计也很方便。VxWorks内核结构从逻辑上可以分成3层，如图2.1所示。



2.1 VxWorks内核层次结构

其中由全局变量kernelState包含的例程构成了Wind内核的核心态，当kernelState设置为TRUE时，表示此时存在代码正在内核态运行。VxWorks的内核态的本质是保护内核数据结构，以防止多处代码对内核数据结构同时进行访问，所以其不同于通用操作系统内核态的概念。

进入内核态只需要简单的将全局变量kernelState置为TRUE，从这一点开始所有的内核数据将会被保护，而避免竞争。当结束内核操作时，VxWorks通过windExit()例程将kernelState重新设置为FALSE。让我们考虑一下当处在内核态时，谁会竞争使用内核数据结构，显然中断服务例程是仅有可能的在Wind内核处于内核时，请求内核额外工作的始作俑者。这也就意味着，一旦系统进入内核态，应用程序请求内核服务的仅有方式通过中断服务程序。

kernelState在设置之前总是会被检查是否已经被置位。这就是互斥机制的精髓，VxWorks内核使用将工作延迟来作为互斥机制实现方式的技术，当kernelState为TRUE的时候，要做的工作不会立即执行，而是作为一个延迟的Job放到延时工作队列中。内核延时工作队列只能在恢复前一个将要执行的任务上下文之前，被windExit()清空(或者当中断ISR首先进入内核态的条件下的，intExit()清空)。在内核态即将结束的时刻关中断，windExit()会检查工作队列是否为空，并进入一个被选择的任务上下文。

正如前文所述，Wind内核使用kernelState全局变量从软件上模拟了一个特权态，处于特权态禁止任务抢占。处于特权态例程的函数在windLib库中。执行windLib库中的例程，更高层次的调用例程将会进入内核态(kernelState=1)，获得对所有内核队列的互斥使用。windLib库中的例程具有自由支配任务内核数据结构的能力。内核态(kernelState=1)是一种强有力的互斥工具，处于内核态的时间内，抢占是被禁止的。高的抢占时延破坏了实时系统的响应时延的快速性，因此必须非常保守的使用这种机制(目前使用这种设计机制的开源RTOS，只有RTEMS)。实际上微内核的设计理念是保持内核足够小的同时，使其仍具有支持更高级别应用的能力，还记得我在前一章说过的话吗？“一个优美的内核不是还有什么样的功能还可以增加，而是还有什么样的功能还可以减少”。

Wind内核在处于特权态时是开中断的，这就意味此时内核仍可以继续响应外部中断，Wind内核的创新之处在于提出了工作队列(Work Queue)的概念。由于内核态只能被高层次的应用互斥访问，当中断发生时，如果没有程序访问内核态例程，当前中断ISR首先会进入内核态(将kernelState置为TRUE)，然后执行相应的中断ISR；如果内核态已经被占用(kernelState=FALSE)，那边当前中断的ISR将会被放置到内核工作队列后直接返回。等到占用内核态的程序退出内核态(调用windExit()例程)时会执行内核工作队列中的工作(即Job)，其中的中断ISR将会得到处理(我会在后续的博文中结合代码具体分析O(∩\_∩)O)。

wind内核中kernelState包含的内核态例程在windLib库中，以wind\*开头的例程，见图2.2。

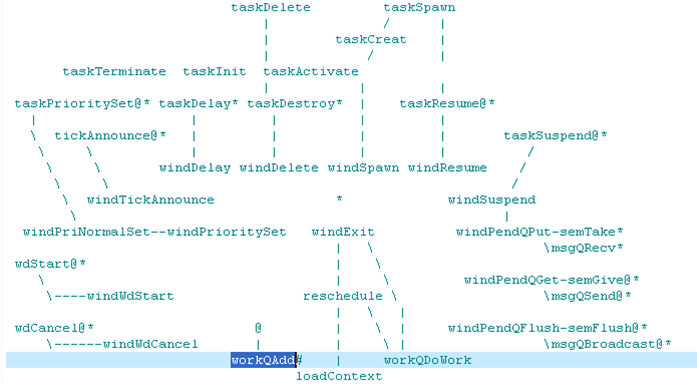


图2.2 VxWorks内核态例程示意图

其中：

\*表示中断ISR不能调用该例程，只能在任务级被调用；

@\*表示可以在中断ISR中调用该例程；

#表示将在wind内核内部使用；

@表示可以在内核态运行；

VxWorks系统的组成部分见图2.3。

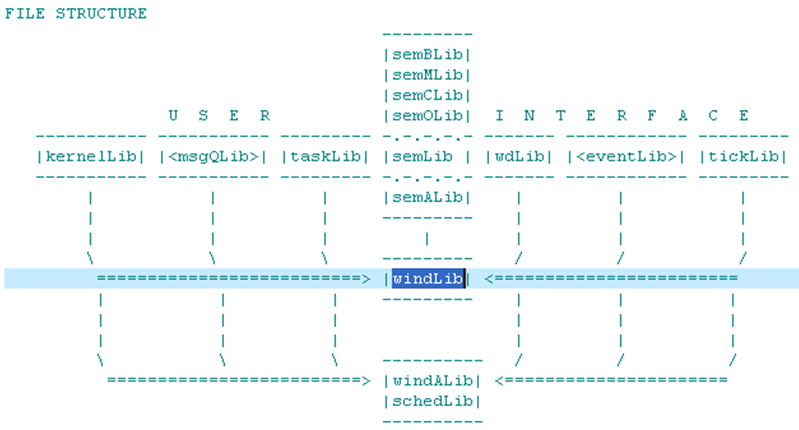


图2.3 VxWorks系统构成图

**2.2 Wind内核的类和对象**

VxWorks的Wind内核采用类和对象的思想将wind内核的5个组成部分：任务管理模块、内存管理模块、消息队列管理模块、信号量管理模块、以及看门狗管理模块组织起来。

在Wind内核中，所有的对象都是类的组成部分，类定义了操作对象的方法(Method)，同时维护着对所有对象的操作记录。Wind内核采用了C++的语义，但是采用C语言来实现。整个Wind内核通过显式编码实现，其编译过程并不依赖于具体的编译器。这意味着Wind内核不但可以在VxWorks自带的diab编译上编译，也可以使用开源的GNU/GCC编译器。VxWorks为Wind内核设计了一个元类(Meta-class)，所有的对象类(Obj-class)都是基于该元类。每个对象类只负责维护各自对象(Object)的操作方法(比如创建对象、初始化对象、注销对象等)、以及管理统计记录(比如一个创建对象的数据、销毁对象的数目等)。类管理模式不是VxWorks内核的特性，它是操作系统的组成部分；但是所有的内核对象都依赖于它。Wind内核各个组成模块间对象类、对象和元类的关系如图2.4所示。

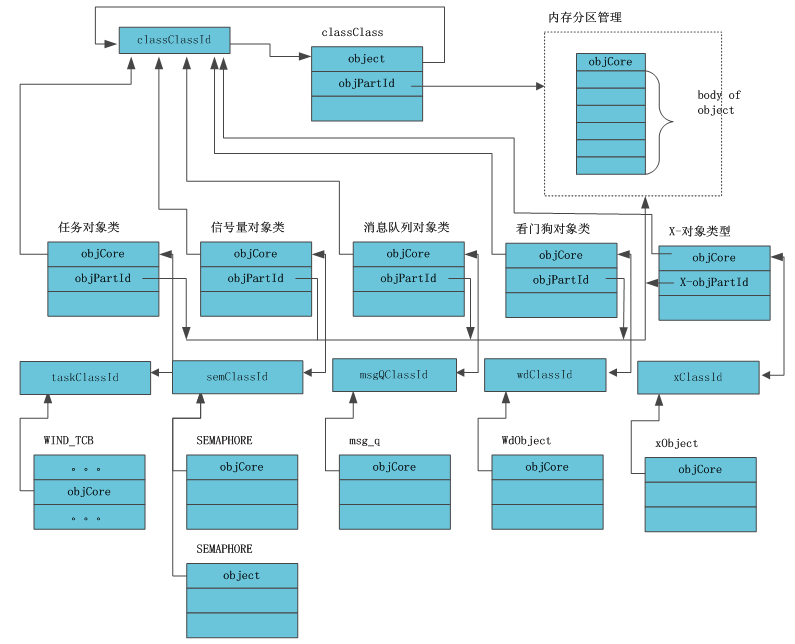


图2.4 对象类、对象和元类的关系示意图

备注：采用对象和类的设计思想，可以将VxWorks的Wind内核的各个组件有机的组织起来，创建相同组件的各个实例时都便于验证实例类型的正确性，同时所有的组件对象类都来源于基类，对所有对象的操作记录。

**2.3 wind内核特性**

多任务：内核的基本功能是提供一个多任务环境。多任务使得许多程序在表面上表现为并发执行，而事实上内核是根据基本的调度算法使他们分段执行。每个明显独立的程序被成为一个任务。每个任务拥有自己的上下文，其中包含在内核调度使该任务执行的时候它所看到的CPU环境和系统资源。

任务状态：内核维护系统中的每个任务的当前状态。状态迁移发生在应用程序调用内核功能服务的时候。下面定义了Wind内核状态：

就绪态----一个任务当前除了CPU不等待任何资源

阻塞态----一个任务由于某些资源不可获得而被阻塞

延迟态----一个任务睡眠一段时间

挂起态----主要用于调试的一个辅助状态，挂起禁止任务的执行

任务被创建以后进入挂起态，需要通过特定的操作使被创建的任务进入就绪态，这一操作执行速度很快，使应用程序能够提前创建任务，并以一种快捷的方式激活该任务。

调度控制：多任务需要一个调度算法分配CPU给就绪的任务。在VxWorks中默认的调度算法是基于优先级的抢占调度，但应用程序也可以选择使用时间片轮转调度。

* 基于优先级抢占调度：基于优先级的抢占调度，每个任务被指定一个优先级，内核分配CPU给处于就绪态的优先级最高的任务。调度采用抢占的方式，是因为当一个优先级高于当前任务的任务变为就绪态时，内核将立即保存当前任务的上文，并切换到高优先级任务的上文。VxWorks有从0到255共256个优先级。在创建的时候任务被指定一个优先级，在任务运行的过程中可以动态地修改优先级以便跟踪真实世界的事件优先级。外部中断被指定优先于任何任务的优先级，这样能够在任何时候抢占一个任务。
* 时间片轮转：基于优先级抢占调度可以扩充时间片轮转调度。时间片轮转调度允许在相同优先级的处于就绪态的任务公平地共享CPU。没有时间片轮转调度，当有多个任务在同一优先级共享处理器时，一个任务可能独占CPU，不会被阻塞直到被一个更高优先级的任务抢占，而不给同一优先级的其他任务运行的机会。如果时间片轮转被使能，执行任务的时间计数器在每个时钟滴答递增。当指定的时间片耗尽，计数器会被清零，该任务被放在同一优先级任务队列的队尾。加入特定优先级组的新任务被放在该组任务的队尾，并将运行计数器初始化为零。

基本的任务函数：用于状态控制的基本任务函数包括一个任务的创建、删除、挂起和唤醒。一个任务也可以使自己睡眠一个特定的时间间隔不去运行。许多其他任务例程提供由任务上下文获得的状态信息。这些例程包括访问一个任务当前处理器寄存器控制。

任务删除问题： wind内核提供防止任务被意外删除的机制。通常，一个执行在临界区或访问临界资源的任务要被特别保护。我们设想下面的情况：一个任务获得一些数据结构的互斥访问权，当它正在临界区内执行时被另一个任务删除。由于任务无法完成对临界区的操作，该数据结构可能还处于被破坏或不一致的状态。而且，假想任务没有机会释放该资源，那么现在其他任何任务现在就不能获得该资源，资源被冻结了。

任何要删除或终止一个设定了删除保护的任务的任务将被阻塞。当被保护的任务完成临界区操作以后，它将取消删除保护以使自己可以被删除，从而解阻塞删除任务。

正如上面所展示的，任务删除保护通常伴有互斥操作。    这样，为了方便性和效率，互斥信号量包含了删除保护选项（我会在后续的博文中详细介绍）

任务间通信： 为了提供完整的多任务系统的功能，Wind内核提供了一套丰富的任务间通信与同步的机制。这些通信功能使一个应用中各个独立的任务协调他们的活动。

共享地址空间： Wind内核的任务间通信机制的基础是所有任务所在的共享地址空间。通过共享地址空间，任务能够使用共享数据结构的指针自由地通信。管道不需要映射一块内存区到两个互相通信任务的寻址空间。

备注：不幸的是，共享地址空间具有上述优点的同时，带来了未被保护内存的重入访问的危险。UNIX，Linux操作系统通过隔离进程提供这样的保护，但同时带来了对于实时操作系统来说巨大的性能损失。

互斥操作： 当一个共享地址空间简化了数据交换，通过互斥访问避免资源竞争就变为必要的了。用来获得一个资源的互斥访问的许多机制仅在这些互斥所作用的范围上存在差别。实现互斥的方法包括禁止中断、禁止任务抢占和通过信号量进行资源锁定。

* 中断禁止：最强的互斥方法是屏蔽中断。这样的锁定保证了对CPU的互斥访问。这种方法当然能够解决互斥的问题，但它对于实时是不恰当的，因为它在锁定期间阻止系统响应外部事件。长的中断延时对于要求有确定的响应时间的应用来说是不可接受的。
* 抢占禁止：禁止抢占提供了强制性较弱的互斥方式。 当前任务运行的过程中不允许其他任务抢占，而中断服务程序可以执行。这也可能引起较差的实时响应，就象被禁止中断一样，被阻塞的任务会有相当长时间的抢占延时，就绪态的高优先级的任务可能会在能够执行前被强制等待一段不可接受的时间。为避免这种情况，在可能的情况下尽量使用信号量实现互斥。
* 互斥信号量：信号量是用于锁定共享资源访问的基本方式。不象禁止中断或抢占，信号量限制了互斥操作仅作用于相关的资源。一个信号量被创建来保护资源。VxWorks的信号量遵循Dijkstra的P()和V()操作模式。

当一个任务请求信号量，P()操作根据在发出调用时信号量的置位或清零的状态， 会发生两种情况。如果信号量处于置位态， 信号量会被清零，并且任务立即继续执行。如果信号量处于清零态，任务会被阻塞来等待信号量。

当一个任务释放信号量，V()操作会发生几种情况。如果信号量已经处于置位态，释放信号量不会产生任何影响。如果信号量处于清零态且没有任务等待该信号量，信号量只是被简单地置位。如果信号量处于清零态且有一个或多个任务等待该信号量，最高优先级的任务被解阻塞，信号量仍为清零态。

通过将一些资源与信号量关联，能够实现互斥操作。当一个任务要操作资源，它必须首先获得信号量。只要任务拥有信号量，所有其他的任务由于请求该信号量而被阻塞。当一个任务使用完该资源，它释放信号量，允许等待该信号量的另一个任务访问该资源。

Wind内核提供了二值信号量来解决互斥操作所引起的问题。 这些问题包括资源拥有者的删除保护，由资源竞争引起的优先级逆转：

* 删除保护：互斥引起的一个问题会涉及到任务删除。在由信号量保护的临界区中，需要防止执行任务被意外地删除。删除一个在临界区执行的任务是灾难性的。资源会被破坏，保护资源的信号量会变为不可获得，从而该资源不可被访问。通常删除保护是与互斥操作共同提供的。由于这个原因，互斥信号量通常提供选项来隐含地提供前面提到的任务删除保护的机制。
* 优先级逆转/优先级继承：优先级逆转发生在一个高优先级的任务被强制等待一段不确定的时间以便一个较低优先级的任务完成执行。考虑下面的假设(我已经在前面的博文中介绍过，这里在啰嗦下O(∩\_∩)O)：

T1，T2和T3分别是高、中、低优先级的任务。T3通过拥有信号量而获得相关的资源。当T1抢占T3，为竞争使用该资源而请求相同的信号量的时候，它被阻塞。如果我们假设T1仅被阻塞到T3使用完该资源为止，情况并不是很糟。毕竟资源是不可被抢占的。然而，低优先级的任务并不能避免被中优先级的任务抢占，一个抢占的任务如T2将阻止T3完成对资源的操作。这种情况可能会持续阻塞T1等待一段不可确定的时间。这种情况成为优先级逆转，因为尽管系统是基于优先级的调度，但却使一个高优先级的任务等待一个低优先级的任务完成执行。 互斥信号量有一个选项允许实现优先级继承的算法。优先级继承通过在T1被阻塞期间提升T3的优先级到T1解决了优先级逆转引起的问题。这防止了T3，间接地防止T1，被T2抢占。通俗地说，优先级继承协议使一个拥有资源的任务以等待该资源的任务中优先级最高的任务的优先级执行。当执行完成，任务释放该资源并返回到它正常的或标准的优先级。因此，继承优先级的任务避免了被任何中间优先级的任务抢占。

同步：信号量另一种通常的用法是用于任务间的同步机制。在这种情况下，信号量代表一个任务所等待的条件或事件。最初，信号量是在清零态。一个任务或中断通过置位该信号量来指示一个事件的发生。等待该信号量的任务将被阻塞直到事件发生、该信号量被置位。一旦被解阻塞，任务就执行恰当的事件处理程序。信号量在任务同步中的应用对于将中断服务程序从冗长的事件处理中解放出来以缩短中断响应时间是很有用的。

消息队列：消息队列提供了在任务与中断服务程序或其他任务间交换变长消息的一种较低层的机制。这种机制在功能上类似于管道，但有较少的开销。

管道、套接字、远程过程调用和许多高层的vxWorks机制提供任务间通信的更高层的抽象，包括管道、TCP/IP套接字、远程过程调用和更多。为了保持裁减内核为仅包含足够支持高层功能的一个最小函数集的设计目标，这些特性都是基于上面描述的内核同步方式的。

**2.4 Wind 内核设计的优点**

Wind内核的一个重要的设计特性是最小的抢占延时。其他的主要设计的优点包括史无前例的可配置性，对不可预见的应用需求的可扩展性，在各种微处理器应用开发中的移植性。

最小的抢占延时： 正如前面所讨论的，禁止抢占是获得代码临界资源互斥操作的通常手段。这种技巧的不期望的负面影响是高的抢占延时，这可以通过尽量使用信号量实现互斥和保持临界区尽量紧凑。但即使广泛地使用信号量也不能解决所有的可能导致抢占延时的根源。内核本身就是一个导致抢占延时的根源。为了理解其原因，我们必须更好地理解内核所需的互斥操作。

内核级和任务级：在任何多任务系统中，大量的应用是发生在一个或多个任务的上下文中。然而，有些CPU时间片不在任何任务的上下文。这些时间片发生在内核改变内部队列或决定任务调度。在这些时间片中，CPU在内核级执行而非任务级。

为了内核安全地操作它的内部的数据结构，必须有互斥操作。内核级没有相关的任务上下文，内核不能使用信号量保护内部链表。内核使用工作延期作为实现互斥的方式。当有内核参与时，中断服务程序调用的函数不是被直接激活，而是被放在内核的工作队列中。内核完成这些请求的执行而清空内核工作队列。

当内核正在执行已经被请求服务时系统将不响应到达内核的函数调用。可以简单地认为内核状态类似于禁止抢占。如前面所讨论的，抢占延时在实时系统中是不期望有的，因为它增加了对于会引起应用任务重新调度的事件的响应时间。尽管操作系统在内核级（此时禁止抢占）完全避免消耗时间是不可能的，但减少这些时间是很重要的。这是减少由内核执行的函数的数量的主要原因，也是不采用统一结构的系统设计方式的原因。

VxWorks显示了采用任务级操作系统服务设计的一个最小内核是能够满足需求的。VxWorks是现在能够获得的独立于任何处理器的、拥有相当小内核的、功能完全的层次结构的实时操作系统。

VxWorks系统在Wind内核之上提供了大量的功能。它包括内存管理，一个完整的BSD4.3网络包，TCP/IP，网络文件系统（NFS），远程过程调用（RPC），UNIX兼容的链接加载模块，C语言的解释界面，各种类型的定时器，性能监测组件，调试工具，额外的通信工具如管道、信号和套接字，I/O和文件系统，和许多功能例程。这些都不是运行在内核级，所以不会禁止中断或任务抢占。

可配置性： 实时应用有多种内核需求。没有哪个内核有一个用来满足每种需求的很好的设计折衷。然而，一个内核可以通过配置来调整特定的性能特性，裁减实时系统来最好地适应一个应用的要求。不可预见的内核配置性以用户可选择的内核排队算法的形式提供给应用。

排队策略：vxWorks中的排队库是独立于使用他们的内核队列功能而执行的，这样提供了将来增加新的排队方式的灵活性。

在VxWorks中有各种内核队列。就绪队列是一个按优先级索引的所有等待调度的任务队列。滴答队列用于定时功能。信号量队列是一个等待信号量的被阻塞任务的链表。活动队列是一个系统中所有任务的一个先进先出（FIFO）的链表。这些队列中的每个队列都需要一个不同的排队算法。这些算法不是被内嵌在内核中，而是被抽取到一个自治的、可转换的排队库中。这种灵活的组织形式是满足特殊的配置需求的基础。

可扩展性： 支持不可预见的内核扩展的能力与以有功能的可配置性是同样重要的。简单的内核接口和互斥方法使内核级功能扩展相当容易； 在某些情况下，应用可以仅利用内核钩子函数来实现特定的扩展。

内部钩子函数：为了不修改内核而能够向系统增加额外的任务相关的功能，VxWorks提供了任务创建、切换和删除的钩子函数。这些允许在任务被创建、 上下文切换和任务被删除的时候额外的例程被调用执行。这些钩子函数可以利用任务上下文中的空闲区创建Wind内核的任务特性。

未来考虑： 有许多系统函数现在变得越来越重要，而且会影响到内核设计时的抢占延时。尽管涉及这些问题一个完整的讨论超出了本博文的范围，但值得简单地提一下。

设计一个独立于CPU的操作系统一直是一个挑战。随着新的RSIC（精简指令集）处理器变得很流行，这些难度也加大了。为了在RISC环境下有效地执行，内核和操作系统需要有执行不同策略的灵活性。

例如，考虑在任务切换时内核执行的例程。在CISC（复杂指令集，如680x0或80x86）CPU，内核为每个任务存储一套完整的寄存器，在运行任务的时候将这些寄存器换入换出。在一个RISC机器上，这样是不合理的，因为涉及到太多的寄存器。所以内核需要一个更精密复杂的策略，如为任务缓存寄存器，允许应用指定一些寄存器给特殊的任务。

移植性：为了使Wind内核在他们出现的结构上能够运行，需要有一个可移植的内核版本。这使移植是可行的，但不是最优化的。

多处理：支持紧耦合的多处理需求要求实时内核的内部功能包含，在理想情况下，在远端请求内核调用，如从一个处理器到另一个处理器。这就要涉及到信号量调用（为处理器间同步）和任务调用（为了控制另一个CPU上的任务）。这种复杂性无疑会增加内核级功能调用的开销，但是许多服务如对象标识可以在任务级执行。在多处理系统中保持一个最小内核的优点是处理器之间的互锁可以有较好的时间粒度。大的内核将在内核级消耗额外的时间，仅能获得粗糙的互锁时间粒度。

实时内核的重要尺度：许多性能特性被用来比较已有的实时内核，这些包括：

* 快速的任务上下文切换----由于实时系统的多任务的特性，系统能够快速地从一个任务切换到另一个任务是很重要的。在分时系统中，如UNIX，上下文切换是在ms级。Wind内核执行原始上下文切换用us级来衡量。
* 最小的同步开销----因为同步是实现资源互斥访问的基本方法，这些操作所引起的开销最小化是很重要的。在vxWorks中，请求和释放二值信号量也是用us级来衡量。
* 最小的中断延时----因为外部世界来的事件通常以中断的形式到来，操作系统快速的处理这些中断是很重要的。内核在操作一些临界数据结构的时候必须禁止中断。为了减小中断延时，必须使这些时间最小化。Wind内核的中断延时也是us级别。

备注：具体的数值性能指标，只有在具体的目标板上直接测量后，才能获取。

抢占延时对性能指标的影响：  当许多的实时解决方案被提交给应用工程师时， 性能指标对于评估供应商的产品变得越来越重要。不象上下文切换和中断延时，抢占延时很难测量。所以它很少在说明中被提及。但是考虑到当内核通常禁止上下文切换会长达数百微妙，而声称一个50us的固定长度（与任务个数无关）的上下文切换时间是毫无意义的。除了很难测量外，抢占延时可能会削弱许多性能指标的有效性。Wind内核通过减小内核的大小来尽量减小抢占延时。 包含繁多功能的内核必将引起长的抢占延时。

此致，Wind内核我做了一个粗粒度的介绍，接下来的文章，我将就Wind内核具体方面，结合代码来详细的阐述。。。。O(∩\_∩)O