.

**显式并行编程模型 哪几种？分别 述 种并行编程模型的 征？**

### 数据并行模型

数据并行（Data-Parallel）模型主要任务是局部计算和数据选路操作，通常应用于细粒度问题的求解[27]。该模型在SIMD计算机与SPMD计算机上均可以实现，这取决于粒度的大小。

数据并行具有以下特点：

1）单线程：从程序员角度看，这类程序具有单一控制线，即只有一个进程在执行；

2）并行操作于聚合数据结构上：一个单步语句，可以同时作用在不同数组元素的多个操作上；

3）松散同步：在每条语句之后，都有一个隐含的同步；

4）全局命名空间：数据并行程序中所有变量都存储于单一地址空间内，因而只要满足变量的作用域规则，任何语句可以访问任何变量；

5）隐式相互作用：由于每条语句之后隐式同步的存在，所以数据并行程序中不需要显示同步语句，通信也可以隐式完成；

6）隐式数据分配：数据可以自动分配，而不需要程序员干预。

### 消息传递模型

消息传递（Message Passing）模型中，不同处理器节点上的进程需要通过网络发送消息来实现通信。这里的消息包括数据、指令以及信号等。相比于数据并行模型，消息传递模型更加灵活，不仅可以在共享存储式多处理机上运行，也可以在分布存储式多处理机上运行[28]，而且更加适合与开发粒度比较大的并行程序。

消息传递模型有以下特点：

1）多进程：在消息传递模型中，程序包含多个进程，而且各进程拥有独立的控制线，可以执行不同的程序代码，可以实现控制并行以及数据并行；

2）异步并行性：在此模型中，各进程异步执行，因此需要用户显示同步各个进程；

3）分开的地址空间：各个进程存于不同的地址空间内，每个进程的变量对于其他进程是不可访问的；

4）显示相互作用：各进程只能在其所拥有的数据上进行计算，通信、同步等相互作用问题需要程序员显示解决；

5）显示分配：数据和负载需要程序员显示分配给各个进程。

目前，消息传递模型已经成为MPP和COW的主要编程方式，标准库PMV和MPI的使用也增强了这种模型的可移植性。其中，MPI（Message Passing Interface）于1994年5月由美国国家实验室和几所大学共同开发成功，它实际上是一种消息传递函数库的标准，由于在开发过程中吸取了许多其它消息传递系统的优点，MPI目前已经成为最主流的并行编程环境之一。MPI具有以下优点：易用性和可移植性；异步通信功能完备；定义精确且详细。

在基于MPI的编程模型中，各进程通过调用库函数发送、接收消息，共同组成计算过程。通常，在程序初始化时就会生成一组固定的进程，一般一个处理器生成一个进程，这些进程所运行的程序可以相同也可以不同，在进程间的通信也可以是点对点通信或者是群组通信。MPI编程实际上就是计算与通信交织的过程。

### 共享存储模型

在共享存储模型中，各进程可以通过访问公共存储器中的共享变量而实现通信。和数据并行相比，相同点在于拥有全局命名空间；和消息传递模型相比，相同点在于多线程和异步。这种编程模式具有可编程性强和系统可用性高的优点，因而在科学和工程计算中被广泛应用。

OpenMP标准是共享存储体系结构上的一种编程模型，具有简单、移植性好以及可扩展等优点，目前已经成为共享存储系统编程的工业标准。OpenMP本质上来说，是对基本语言的扩展，其规范中定义了制导指令、运行库和环境变量，从而能够保证程序的可移植性，并且根据标准将程序并行化。其中，制导指令对程序语言进行了扩展，从而可以支持并行区域、工作共享以及同步构造，实现数据共享和私有化。而运行库和环境变量则支持用户对程序执行环境的调整。目前，已经有很多软件和硬件厂商支持OpenMP，可以应用于UNIX、Windows NT等多种平台，C、C++、Fortran等语言也都指定了相应的实现规范。

[**https://blog.csdn.net/qq100440110/article/details/51823828**](https://blog.csdn.net/qq100440110/article/details/51823828)

**Windows 事件和临界区机制，相应api：**

**\* 临界区**

**\* 运行在用户模式**

**\* \_其实是管程\_**

**```c**

**void InitializeCriticalSection( LPCRITICAL\_SECTION );**

**void EnterCriticalSection( LPCRITICAL\_SECTION );**

**void LeaveCriticalSection( LPCRITICAL\_SECTION );**

**void DeleteCriticalSection( LPCRITICAL\_SECTION );**

**```**

**```c++**

**CRITICAL\_SECTION csMyCriticalSection;**

**void CriticalSectionExample (void)**

**{**

**InitializeCriticalSection (&csMyCriticalSection); //初始化临界区变量**

**\_\_try**

**{**

**EnterCriticalSection (&csMyCriticalSection); //开始保护机制**

**//此处编写代码**

**}**

**\_\_finally //异常处理，无论是否异常都执行此段代码**

**{**

**LeaveCriticalSection (&csMyCriticalSection); //撤销保护机制**

**}**

**}**

**```**

**\* 事件**

**```c**

**HANDLE CreateEvent(**

**PSECURITY\_ATTRIBUTES psa;**

**BOOL fManualReset,**

**BOOL fInitialState,**

**PCTSTR pszName);**

**BOOL SetEvent(HANDLE hEvent);**

**BOOL ResetEvent(HANDLE hEvent);**

**```**

事件是内核对象，用于线程间通信和同步。事件分为无信号状态和有信号状态，无信号状态会阻塞到WaitForSingleObject()函数，直到触发事件(即调用SetEvent)。有信号状态则会忽略事件触发信号，不会阻塞到WaitForSingleObject()函数。

ResetEvent(), WaitForSingleObject(), CreateEvent(),SetEvent()

[**https://blog.csdn.net/niu2212035673/article/details/78892801**](https://blog.csdn.net/niu2212035673/article/details/78892801)

多线程程序中，有些代码是共享资源，需将这些代码作为临界区。如果有多个线程试图同时访问临界区，那么在一个线程进入后，其他线程将被挂起，并一直持续到进入临界区的线程离开。临界区在被释放后，其他线程可以继续抢占。

    临界区的同步速度很快；不是内核对象，因而不能跨进程同步；不能指定阻塞时的等待时间(只能无限等待下去)。

InitializeCriticalSection()

                           EnterCriticalSection()

                           LeaveCriticalSection()

                           DeleteCriticalSection()

 这四个函数的形参都是一个指向CRITICAL\_SECTION结构体的指针，因而必须先定义一个CRITICAL\_SECTION类型的变量。

    InitializeCriticalSection()的作用是初始化一个临界资源对象；EnterCriticalSection()的作用是查看CRITICAL\_SECTION结构成员变量的值，判断是否有线程访问临界区的资源。如果没有，则更新CRITICAL\_SECTION结构成员变量的值，并将当前的线程赋予资源访问权；如果有线程正在访问临界区的资源，则该函数将线程置为等待状态；LeaveCriticalSection()释放临界区资源的所有权，使其他等待临界区资源的线程能够有机会获得临界区资源的所有权。

**3、事件 - Event**

    (1)说明

    事件是内核对象，具有“激发状态”和“未激发状态”两种状态。事件主要分为两类：

    人工重置事件：用程序手动设置。

    自动重置事件：一旦事件发生并被处理后，自动恢复到没有时间状态。

    (2)有关函数

    使用使用事件对象要涉及的API函数主要有：

                                   CreateEvent()           // 创建事件对象

                                   SetEvent()              // 设置事件对象

                                   ResetEvent()            // 设置事件对象

                                   PulseEvent()            // 设置事件对象

                                   OpenEvent()             // 跨进程时使用

                                   WaitforSingleEvent()    // 等待

                                   WaitForMultipleObjects()// 等待

    (3)应

**4、信号量 - Semaphore**

    (1)说明

    信号量允许多个线程在同一时刻访问统一资源，但是限制了在同一时刻访问共享资源的最大线程数。

    信号量是内核对象，允许跨进程使用。

    (2)有关函数

    使用信号量要涉及的API函数主要有:

                               CreateSemaphore()  // 创建信号量

                               ReleaseSemaphore() // 释放信号量

                               OpenSemaphore()    // 跨进程使用

    在用 CreateSemaphore()创建信号量时即要同时指出允许的最大资源计数和当前可用资源计数。一般是将当前可用资源计数设置为最大资源计数，每增加一个线程对共享资源的访问，当前可用资源计数就会减1，只要当前可用资源计数是大于0的，就可以发出信号量信号。但是当前可用计数减小到0时则说明当前占用资源的线程数已经达到了所允许的最大数目，不能在允许其他线程的进入，此时的信号量信号将无法发出。线程在处理完共享资源后，应在离开的同时通过 ReleaseSemaphore()函数将当前可用资源计数加1。在任何时候当前可用资源计数决不可能大于最大资源计数。

互斥对象的作用是保证每次只能有一个线程获得互斥对象而得以继续执行。互斥对象主要包含使用数量、线程ID和递归计数器等信息。其中，线程ID表示当前拥有互斥对象的线程号，递归计数器表示线程拥有互斥对象的次数。

    互斥对象是是Windows的内核对象，可跨进程互斥，并且能指定阻塞时的等待时间。

    (2) 有关函数

    使用互斥对象要涉及的API函数主要有：

                                CreateMutex()         // 创建互斥对象

                                ReleaseMutex()        // 释放互斥对象

                                OpenMutex()           // 跨进程时使用

                                WaitForSingleObject() // 等待指定时间

    使用互斥编程的一般方法是：

## 写出Flynn的四种机器分类并分别举例

Flynn分类法，是基于指令流和数据流的数量对计算机进行分类的方法。

一系列修改那些流经数据处理单元的数据（数据流）的命令，可以被认为是一个指令流。

以下是四种不同的情况：

* **单指令流单数据流（SISD）**——传统的计算机包含单个CPU，它从存储在内存中的程序那里获得指令，并作用于单一的数据流（本例中就是一个指令处理一条数据）。
* **单指令流多数据流（SIMD）**——单个的指令流作用于多于一个的数据流上。例如有数据4、5和3、2，一个单指令执行两个独立的加法运算：4+5和3+2，就被称为单指令流多数据流。SIMD的一个例子就是一个数组或向量处理系统，它可以对不同的数据并行执行相同的操作。
* **多指令流单数据流（MISD）**——用多个指令作用于单个数据流的情况实际上很少见。这种冗余多用于容错系统。
* **多指令流多数据流（MIMD）**——这种系统类似于多个SISD系统。实际上，MIMD系统的一个常见例子是多处理器计算机，如Sun的企业级服务器。
* **SISD（单指令流单数据流）系统：**
* 传统的计算机只有一块CPU，因此准许一条指令流顺序执行。目前许多大型主机有一块以上的CPU，但各CPU执行的指令流互不相关，因此它们仍被看作是SISD机器，不过是很多运行在不同数据空间上的SISD的组合。很多[HP](http://www.yesky.com/key/214/214.html" \t "_blank)，[DEC](http://www.yesky.com/key/3579/3579.html" \t "_blank)和SUN的工作站属于这种类型。

* **SIMD（单指令流多数据流）系统：**
* 这类系统往往有大量的处理器（从1024到16384），它们按时钟节奏同时对不同的数据执行相同的指令，也即同一指令并行处理多个数据，这种类型的系统有CPP DAP [Gamma](http://www.yesky.com/key/852/5852.html) II和Alenia Quadrics。
* SIMD（单指令流多数据流）系统的一个子类是向量处理，向量处理不处理单个的数据元素而是使用特殊结构的CPU处理由同类数据构成的数组，当数据被向量单元处理时，结果可以按1、2（在特殊情况下3）个时钟周期的频率输出，这样，当微量处理量在向量工作方式下处理数据时，它们基本上也是在并行处理数据，这将比标量工作方式快好几倍，因此大多数人同意微量处理是SIMD的一个变种。[Hitachi](http://www.yesky.com/key/3884/13884.html" \t "_blank)（[日立](http://www.yesky.com/key/1309/16309.html)）[S36](http://www.yesky.com/key/111/10111.html)00是这种系统的一个实例。

* **MISD（多指令流单数据流）系统：**
* 理论上这种系统的多个指令流共同操作同一条数据流，然而到目前为止尚未发明这样的系统，这样的系统也的确难以构思设计。（01年的文章如此写，06年现在如何还不得而知啊）

* **MIMD（多指令流多数据流）系统：**
* 这种系统并行的在不同数据上执行不同的指令，这种类型与上面提到的多处理器SISD系统的本质区别在于不同处理器执行的指令与数据是彼此相关的，它们实际上是在并行执行同一工作的不同部分。因此MIMD系统可以将一个主任务分解为众多子任务并行执行以缩短工作时间。事实上，MIMD系统有大量的子类型这使传统的Flynn分类法显得力不从心，四处理器的NEC SX-5与上千个处理器的SGI/Cray T3E都属于这一类型而它们的工作原因却大不相同。因此，我们不得不将之划分为若干子类以区别对待：
  + 共享内存系统：这种系统的多个CPU共享同一地址空间，这意味着数据的实际存放位置对用户而言并不重要，因为所有的CPU都使用同一内存访问地址，同享内存系统可以是SIMD或MIMD系统，使用向量处理器的系统可以看成是前者，而使用多处理器的系统可以看成是后者，我们用[SM](http://www.yesky.com/key/2711/167711.html" \t "_blank)-SIMD和[SM](http://www.yesky.com/key/347/347.html" \t "_blank)-MIMD分别标志这两 种子类型。
  + 分布式内存系统：这种情况下每个CPU有其自己独立的内存，这些CPU通过网络相连并在需要时交换各自内存的数据。与共享内存系统相比，用户必需清楚的知道数据存放的实际位置并在需要时显示的分发数据到其它地址，这种系统与可以是SIMD类型或MIDM类型，分布式内存SIMD系统的多个处理器都有自己的内存空间并按时钟频率操作它们各自的数据，分布式内存MIMD系统在CPU的连接上采用了多种多样的拓扑布局，而这样布局的细节对用户而言是透明的，这有助于编程的便捷。这两种子类我们分别用DM-SIMD和DM-MIMD标志。

1. 常用的PC机采用什么并行计算系统？描述这个并行计算系统的特点（9分）

SMP 并行计算系统

特点：

\* SMP \*

\* 结构对称，采用单一操作系统

\* 所有处理器通过高速总线或交叉开关与共享存储器相连，具有单一的地址空间

\* 通过共享变量通信，快捷且编程容易

\* 存储器和I/O负载大，容易成为系统的瓶颈，限制了系统中处理器的数量

\* 容错性极低（单点失效就会导致整个系统崩溃）

\* 拓展性差

用户态自旋锁：

自旋锁最多可能被一个可执行线程所持有。一个被征用的自旋锁使得请求它的线程在等待锁重新可用时自旋（特别浪费处理器时间）。所以自旋锁不应该被长时间持有。

**自旋锁是不可递归的!**

（1）自旋锁相关函数

       用户态的自旋锁相关函数包含在头文件<pthread.h>中。

       相关函数：

       int pthread\_spin\_destroy(pthread\_spinlock\_t \*lock);

       销毁自旋锁lock，并且回收被锁lock使用的任何资源。

    int pthread\_spin\_init(pthread\_spinlock\_t \*lock, int pshared);

       分配使用自旋锁lock所需要的资源，并且初始化锁lock为未锁状态。

       int pthread\_spin\_lock(pthread\_spinlock\_t \*lock);

       锁住自旋锁lock。当锁没有被某个线程持有时，调用的线程将获得锁，否则线程将不断自旋，知道锁可用。

    int pthread\_spin\_trylock(pthread\_spinlock\_t \*lock);

       如果锁没有被某个线程持有，自旋锁lock将被锁住，否则将会失败。

       int pthread\_spin\_unlock(pthread\_spinlock\_t \*lock);

       释放被锁住的自旋锁lock。

（2）基本使用形式

       pthread\_spin\_init(&lock, 0);

       …

       pthread\_spin\_lock(&lock);

       /\*临界区资源…\*/

       pthread\_spin\_unlock(&lock);

       pthread\_spin\_destroy(&lock);

       真正需要保护的是数据而不是代码，采用特定的锁保护自己的共享数据。

缓存一致性：用于保证多个CPU cache之间缓存共享数据的一致。

    至于MESI，则是缓存一致性协议中的一个，到底怎么实现，还是得看具体的处理器指令集。

## 1. cache的写方式

    cache的写操作方式可以追溯到大学教程《计算机组成原理》一书。

    A. write through（写通）：每次CPU修改了cache中的内容，立即更新到内存，也就意味着每次CPU写共享数据，都会导致总线事务，因此这种方式常常会引起总线事务的竞争，高一致性，但是效率非常低；

    B. write back（写回）：每次CPU修改了cache中的数据，不会立即更新到内存，而是等到cache line在某一个必须或合适的时机才会更新到内存中；

    无论是写通还是写回，在多线程环境下都需要处理缓存cache一致性问题。为了保证缓存一致性，处理器又提供了写失效（write invalidate）和写更新（write update）两个操作来保证cache一致性。

    写失效：当一个CPU修改了数据，如果其他CPU有该数据，则通知其为无效；

    写更新：当一个CPU修改了数据，如果其他CPU有该数据，则通知其跟新数据；

    写更新会导致大量的更新操作，因此在MESI协议中，采取的是写失效（即MESI中的I：ivalid，如果采用的是写更新，那么就不是MESI协议了，而是MESU协议）。

为什么需要CPU cache？因为CPU的频率太快了，快到主存跟不上，这样在处理器时钟周期内，CPU常常需要等待主存，浪费资源。所以cache的出现，是为了缓解CPU和内存之间速度的不匹配问题（结构：cpu -> cache -> memory）。

    CPU cache有什么意义？cache的容量远远小于主存，因此出现cache miss在所难免，既然cache不能包含CPU所需要的所有数据，那么cache的存在真的有意义吗？当然是有意义的——局部性原理。

    A. 时间局部性：如果某个数据被访问，那么在不久的将来它很可能被再次访问；

    B. 空间局部性：如果某个数据被访问，那么与它相邻的数据很快也可能被访问；

## 2. cache和寄存器

    存储器的三个性能指标——速度、容量和每位价格——导致了计算机组成中存储器的多级层次结构，其中主要是缓存和主存、主存和磁盘的结构。那么在主存之上，cache和寄存器之间的关系是？

Cache由标记存储器和数据存储器两个基本部分组成。标记存储器是用来储存Cache的控制位与块地址标签，控制位用于管理Cache的读写操作，而块地址标签则记录着Cache中各块的地址。这个地址包含了与主内存映射的块地址，并且都与Cache中的一块“数据”相对应。而这块“数据”正是贮存于Cache的数据存储器中。当CPU读取数据时，先通过地址总线把物理地址送到Cache中，与Cache中的块地址标签进行对比。若相符合，则表示此数据已经存在于Cache中（此情况被戏称为“命中”），这时只需把Cache中的对应数据经由数据总线直接传送给CPU即可。但如果CPU送来的物理地址无法与Cache中的块地址标签相符，则表明这一数据不在Cache中（称为“失误”），这时，需要由主内存把CPU所需的数据地址拷贝到Cache中，再由Cache把数据传送给CPU。从这个过程我们可以看到，若CPU读取“命中”，存取速度确实可以提高许多，但如果“失误”，则Cache的存在反而减慢了CPU的读取速度。因此，采用何种技术和方法提高读写命中率、减少失误率，就成了Cache设计的关键。加大Cache的容量当然可以提高命中率，但因成本问题，Cache不可能无限增大，但可以通过采用适当的映射方式和块替代方式来提高命中率。  
  
　　所谓映射方式是指Cache中的数据如何与主内存中的数据相对应的问题。映射方式通常有以下三种：  
  
直接映射  
  
　　如果主内存上的块只能映射到Cache中的特定块，我们称这种映射方式为直接映射。直接映射的存取速度最快，但失误率也最高。  
  
完全映射  
  
　　在这种映射方式下，主内存上的块可以映射到Cache的任意块之中，当CPU欲读取某一个块时，Cache会把CPU送来的地址与Cache中的所有地址标签进行对比。由于是完全对比，因此存取时间最长，但失误率也最低。  
  
结合映射  
  
　　这种映射方式是把Cache分成若干个页面，每个页面会有相同数目的块。主内存中数据块可以映射到Cache中指定页面的任一块中。这种映射方式可以看成是直接映射与完全映射的折衷，是效率最高的映射方式。  
  
　　由上可见，结合映射方式最为理想，也是应用最为广泛的映射方式。但由于Cache的容量比主内存要小得多，因此无论采用何种映射方式，失误都在所难免。由于当CPU的存取出现“失误”时，必须从主内存把相应的块地址与数据写入Cache中，若此时Cache已经饱和，写入的数据必然会覆盖掉Cache中原有的数据，这就是“块替代”。那么，采用何种块替代方式可以最大限度的减低失误率呢？可以采用的方式又是什么呢？是这么三种：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| fork | | | 创建一个新进程 | |
| clone | | | 按指定条件创建子进程 | |
| execve | | | 运行可执行文件 | |
| exit | | | 中止进程 | |
| \_exit | | | 立即中止当前进程 | |
| getdtablesize | | | 进程所能打开的最大文件数 | |
| getpgid | | | 获取指定进程组标识号 | |
| setpgid | | | 设置指定进程组标志号 | |
| getpgrp | | | 获取当前进程组标识号 | |
| setpgrp | | | 设置当前进程组标志号 | |
| getpid | | | 获取进程标识号 | |
| getppid | | | 获取父进程标识号 | |
| getpriority | | | 获取调度优先级 | |
| setpriority | | | 设置调度优先级 | |
| modify\_ldt | | | 读写进程的本地描述表 | |
| nanosleep | | | 使进程睡眠指定的时间 | |
| nice | | | 改变分时进程的优先级 | |
| pause | | | 挂起进程，等待信号 | |
| personality | | | 设置进程运行域 | |
| prctl | | | 对进程进行特定操作 | |
| ptrace | | | 进程跟踪 | |
| sched\_get\_priority\_max | | | 取得静态优先级的上限 | |
| sched\_get\_priority\_min | | | 取得静态优先级的下限 | |
| sched\_getparam | | | 取得进程的调度参数 | |
| sched\_getscheduler | | | 取得指定进程的调度策略 | |
| sched\_rr\_get\_interval | | | 取得按RR算法调度的实时进程的时间片长度 | |
| sched\_setparam | | | 设置进程的调度参数 | |
| sched\_setscheduler | | | 设置指定进程的调度策略和参数 | |
| sched\_yield | | | 进程主动让出处理器,并将自己等候调度队列队尾 | |
| vfork | | | 创建一个子进程，以供执行新程序，常与execve等同时使用 | |
| wait | | | 等待子进程终止 | |
| wait3 | | | 参见wait | |
| waitpid | | | 等待指定子进程终止 | |
| wait4 | | | 参见waitpid | |
| capget | | | 获取进程权限 | |
| capset | | | 设置进程权限 | |
| getsid | | | 获取会晤标识号 | |
| setsid | | | 设置会晤标识号 | |
| ipc | 进程间通信总控制调用 | |

##### 1、信号

|  |
| --- |
| sigaction | 设置对指定信号的处理方法 |
| sigprocmask | 根据参数对信号集中的信号执行阻塞/解除阻塞等操作 |
| sigpending | 为指定的被阻塞信号设置队列 |
| sigsuspend | 挂起进程等待特定信号 |
| signal | 参见signal |
| kill | 向进程或进程组发信号 |
| \*sigblock | 向被阻塞信号掩码中添加信号,已被sigprocmask代替 |
| \*siggetmask | 取得现有阻塞信号掩码,已被sigprocmask代替 |
| \*sigsetmask | 用给定信号掩码替换现有阻塞信号掩码,已被sigprocmask代替 |
| \*sigmask | 将给定的信号转化为掩码,已被sigprocmask代替 |
| \*sigpause | 作用同sigsuspend,已被sigsuspend代替 |
| sigvec | 为兼容BSD而设的信号处理函数,作用类似sigaction |
| ssetmask | ANSI C的信号处理函数,作用类似sigaction |

##### 2、消息

|  |
| --- |
| msgctl | 消息控制操作 |
| msgget | 获取消息队列 |
| msgsnd | 发消息 |
| msgrcv | 取消息 |

##### 3、管道

|  |
| --- |
| pipe | 创建管道 |

##### 4、信号量

|  |
| --- |
| semctl | 信号量控制 |
| semget | 获取一组信号量 |
| semop | 信号量操作 |

##### 5、共享内存

|  |
| --- |
| shmctl | 控制共享内存 |
| shmget | 获取共享内存 |
| shmat | 连接共享内存 |
| shmdt | 拆卸共享内存 |

单处理器环境不需要自旋锁，调用了自旋锁的函数，里面也不是自旋锁的实现，只不过是用了层壳而已

原因是：

如果是非抢占的系统：（高优先级的进程不能中止正在内核中运行的低优先级的进程而抢占CPU运行。进程一旦处于核心态(例如用户进程执行系统调用)，则除非进程自愿放弃CPU，否则该进程将一直运行下去，直至完成或退出内核。）

一个进程（进入到了内核态），或者线程用了自旋锁，没有执行完毕之前，没有别的进程或者内核线程跟他抢，即使是时间片到了之 后，接着还是分给本进程或者线程。

如果是软中断或者是中断打断了怎么办？ 没关系，关中断即可。也就是说，这时自旋锁退化成关开中断

如果是抢占的系统：（即当进程位于内核空间时，有一个更高优先级的任务出现时，如果当前内核允许抢占，则可以将当前任务挂起，执行优先级更高的进程。）

自旋锁的定义变成非抢占即可，这时高优先级进程就没法跟他抢了。中断是一样的处理。

总之，单处理器来说，非抢占的话，自旋锁退化为 关开中断；

对于抢占来说，自旋锁变成  禁止/打开抢占+关开中断