# linux重点

<http://wenku.baidu.com/view/ec464a3210661ed9ad51f368.html>

嵌入式系统Linux内核开发工程师必须掌握的三十道题

附有答案：<http://blog.csdn.net/zhe563323884/article/details/8642817>

1) Linux中主要有哪几种内核锁？

顺便谈一下Windows上几种典型的同步机制：

CriticalSection

Mutex

Semaphore

Event - 和Mutex的区别：http://stackoverflow.com/questions/6913478/mutex-vs-event-in-windows

Event的应用场景类似于：一个线程做了一些初始化工作，然后一声令下，其他等待Event的线程都开始启动的情况，对这种例子，就要用Event，而不可以用Mutex。

I/O completion ports provide an efficient threading model for processing multiple asynchronous I/O requests on a multiprocessor system.

自旋锁 spinlock

读写自旋锁

信号量 semaphore

#补充 completion

Linux的内核锁主要是自旋锁和信号量。  
**自旋锁**最多只能被一个可执行线程持有，如果一个执行线程试图请求一个已被争用（已经被持有）的自旋锁，那么这个线程就会一直进行忙循环——旋转——等待锁重新可用。要是锁未被争用，请求它的执行线程便能立刻得到它并且继续进行。自旋锁可以在任何时刻防止多于一个的执行线程同时进入临界区。  
Linux中的**信号量**是一种睡眠锁。如果有一个任务试图获得一个已被持有的信号量时，信号量会将其推入等待队列，然后让其睡眠。这时处理器获得自由去执行其它代码。当持有信号量的进程将信号量释放后，在等待队列中的一个任务将被唤醒，从而便可以获得这个信号量。  
信号量的睡眠特性，使得信号量适用于锁会被长时间持有的情况；只能在进程上下文中使用，因为中断上下文中是不能被调度的；另外当代码持有信号量时，不可以再持有自旋锁。

另一个类似的话题：Linux**内核**的同步机制有哪几种：

参考<http://blog.csdn.net/lucien_cc/article/details/7440225>

\*）**atomic**（原子操作）：

所谓原子操作，就是该操作绝不会在执行完毕前被任何其他任务或事件打断，也就说，它的最小的执行单位，不可能有比它更小的执行单位，因此这里的原子实际是使用了物理学里的物质微粒的概念。  
原子操作需要硬件的支持，因此是架构相关的，其API和原子类型的定义都定义在内核源码树的include/asm/atomic.h文件中，它们都使用汇编语言实现，因为C语言并不能实现这样的操作。  
  
原子操作主要用于实现资源计数，很多引用计数(refcnt)就是通过原子操作实现的。  
  
原子类型定义如下：  
typedefstruct { volatile int counter; }atomic\_t;  
volatile修饰字段告诉gcc不要对该类型的数据做优化处理，对它的访问都是对内存的访问，而不是对寄存器的访问。

\*）**Spinlock(**自旋锁**)**

自旋锁与互斥锁有点类似，**只是自旋锁不会引起调用者睡眠**，如果自旋锁已经被别的执行单元保持，调用者就一直循环在那里看是否该自旋锁的保持者已经释放了锁，"自旋"一词就是因此而得名。由于自旋锁使用者一般保持锁时间非常短，因此选择自旋而不是睡眠是非常必要的，自旋锁的效率远高于互斥锁。

信号量和读写信号量适合于保持时间较长的情况，它们会导致调用者睡眠，因此只能在进程上下文使用（\_trylock的变种能够在中断上下文使用），而自旋锁适合于保持时间非常短的情况，它可以在任何上下文使用。如果被保护的共享资源只在进程上下文访问，使用信号量保护该共享资源非常合适，如果对共巷资源的访问时间非常短，自旋锁也可以。但是如果被保护的共享资源需要在中断上下文访问（包括底半部即中断处理句柄和顶半部即软中断），就必须使用自旋锁。

自旋锁保持期间是抢占失效的，而信号量和读写信号量保持期间是可以被抢占的。自旋锁只有在内核可抢占或SMP的情况下才真正需要，在单CPU且不可抢占的内核下，自旋锁的所有操作都是空操作。

跟互斥锁一样，一个执行单元要想访问被自旋锁保护的共享资源，必须先得到锁，在访问完共享资源后，必须释放锁。如果在获取自旋锁时，没有任何执行单元保持该锁，那么将立即得到锁；如果在获取自旋锁时锁已经有保持者，那么获取锁操作将自旋在那里，直到该自旋锁的保持者释放了锁。

无论是互斥锁，还是自旋锁，在任何时刻，最多只能有一个保持者，也就说，在任何时刻最多只能有一个执行单元获得锁。

**需要特别提醒读者，spin\_lock可用于阻止在不同CPU上的执行单元对共享资源的同时访**

**写SpinLock代码要注意的几点问题：**

* **任何拥有自旋锁的代码必须不能休眠,原因是当一段代码在某个CPU上调用自旋锁函数时内核会在这个CPU上禁止抢占，所以一旦拥有自旋锁的代码休眠后不能及时唤醒，CPU就无法执行其它的任务了。所以我们要注意仔细检查每一个我们调用的函数，防止该函数会引起休眠。**
* **要仔细权衡是否需要在调用自旋锁时同时禁止中断。内核提供了两套的API，一个是允许中断的自旋锁（spin\_lock），一种是禁止中断的自旋锁(spin\_lock\_irq, spin\_lock\_irqsave)。考虑在未禁止中断的前提下，驱动的一段代码在一个CPU上运行时获得了一个自旋锁，正在执行中未退出时，设备发生了一个中断，同一个CPU被该中断抢占并且在该中断中又尝试去获取同一个自旋锁并进入自旋，基于前一个问题，因为本地CPU此时已经被禁止抢占，所以中断将永远自旋下去无法退出。所以如果你有一个可能被在(硬件或软件)中断上下文运行的代码获得的自旋锁，则要禁止中断。**
* **自旋锁的代码必须拥有尽可能短的CPU时间，保证快速执行完成。**

\*）**mutex(**互斥锁**)**

互斥锁主要用于实现内核中的互斥访问功能。内核互斥锁是在原子API之上实现的，但这对于内核用户是不可见的。对它的访问必须遵循一些规则：同一时间只能有一个任务持有互斥锁，而且只有这个任务可以对互斥锁进行解锁。互斥锁不能进行递归锁定或解锁。一个互斥锁对象必须通过其API初始化，而不能使用memset或复制初始化。一个任务在持有互斥锁的时候是不能结束的。互斥锁所使用的内存区域是不能被释放的。使用中的互斥锁是不能被重新初始化的。并且互斥锁不能用于中断上下文。但是**互斥锁比当前的内核信号量选项更快**，并且更加紧凑，因此如果它们满足您的需求，那么它们将是您明智的选择。

注: pthread\_mutex\_lock就是mutex\_lock的封装

<http://unix.derkeiler.com/Newsgroups/comp.unix.solaris/2006-02/msg01200.html。>但这是Sun的解释，在Linux上需要再看看。

\*）**semaphore (**信号量**)**

Linux内核的信号量在概念和原理上与用户态的SystemV的IPC机制信号量是一样的，但是它绝不可能在内核之外使用，因此它与SystemV的IPC机制信号量毫不相干。

信号量在创建时需要设置一个初始值，表示同时可以有几个任务可以访问该信号量保护的共享资源，**初始值为1就变成互斥锁（Mutex）（但请注意这仍然是用Linux的Semaphore实现的类互斥锁，他是采用init\_MUTEX定义的，这个和前面描述的用mutex\_init定义的linux原生互斥锁是两个概念， 参考**http://blog.csdn.net/zhangjie201412/article/details/7403123**）**，即同时只能有一个任务可以访问信号量保护的共享资源。一个任务要想访问共享资源，首先必须得到信号量，获取信号量的操作将把信号量的值减1，若当前信号量的值为负数，表明无法获得信号量，该任务必须挂起在该信号量的等待队列等待该信号量可用；若当前信号量的值为非负数，表示可以获得信号量，因而可以立刻访问被该信号量保护的共享资源。当任务访问完被信号量保护的共享资源后，必须释放信号量，释放信号量通过把信号量的值加1实现，如果信号量的值为非正数，表明有任务等待当前信号量，因此它也唤醒所有等待该信号量的任务。

\*）**rw\_semaphore** （读写信号量）

读写信号量对访问者进行了细分，或者为读者，或者为写者，读者在保持读写信号量期间只能对该读写信号量保护的共享资源进行读访问，如果一个任务除了需要读，可能还需要写，那么它必须被归类为写者，它在对共享资源访问之前必须先获得写者身份，写者在发现自己不需要写访问的情况下可以降级为读者。读写信号量同时拥有的读者数不受限制，也就说可以有任意多个读者同时拥有一个读写信号量。如果一个读写信号量当前没有被写者拥有并且也没有写者等待读者释放信号量，那么任何读者都可以成功获得该读写信号量；否则，读者必须被挂起直到写者释放该信号量。如果一个读写信号量当前没有被读者或写者拥有并且也没有写者等待该信号量，那么一个写者可以成功获得该读写信号量，否则写者将被挂起，直到没有任何访问者。因此，写者是排他性的，独占性的。

读写信号量有两种实现，一种是通用的，不依赖于硬件架构，因此，增加新的架构不需要重新实现它，但缺点是性能低，获得和释放读写信号量的开销大；另一种是架构相关的，因此性能高，获取和释放读写信号量的开销小，但增加新的架构需要重新实现。在内核配置时，可以通过选项去控制使用哪一种实现。

\*）**seqlock（顺序锁）**

用于能够区分读与写的场合，并且是读操作很多、写操作很少，写操作的优先权大于读操作。  
seqlock的实现思路是，用一个递增的整型数表示sequence。写操作进入临界区时，sequence++；退出临界区时，sequence再++。写操作还需要获得一个锁（比如mutex），这个锁仅用于写写互斥，以保证同一时间最多只有一个正在进行的写操作。  
当sequence为奇数时，表示有写操作正在进行，这时读操作要进入临界区需要等待，直到sequence变为偶数。读操作进入临界区时，需要记录下当前sequence的值，等它退出临界区的时候用记录的sequence与当前sequence做比较，不相等则表示在读操作进入临界区期间发生了写操作，这时候读操作读到的东西是无效的，需要返回重试。  
  
seqlock写写是必须要互斥的。但是seqlock的应用场景本身就是读多写少的情况，写冲突的概率是很低的。所以这里的写写互斥基本上不会有什么性能损失。  
而读写操作是不需要互斥的。seqlock的应用场景是写操作优先于读操作，对于写操作来说，几乎是没有阻塞的（除非发生写写冲突这一小概率事件），只需要做sequence++这一附加动作。而读操作也不需要阻塞，只是当发现读写冲突时需要retry。  
  
seqlock的一个典型应用是时钟的更新，系统中每1毫秒会有一个时钟中断，相应的中断处理程序会更新时钟（见《[linux时钟浅析](http://hi.baidu.com/_kouu/blog/item/c4d1c8fb1117f19d59ee9010.html)》）（写操作）。而用户程序可以调用gettimeofday之类的系统调用来获取当前时间（读操作）。在这种情况下，使用seqlock可以避免过多的gettimeofday系统调用把中断处理程序给阻塞了（如果使用读写锁，而不用seqlock的话就会这样）。中断处理程序总是优先的，而如果gettimeofday系统调用与之冲突了，那用户程序多等等也无妨。

\*）**RCU（read-copy-update）**

\*）**BKL**（大内核锁）

\*）**rwlock** （读写锁）

2) Linux中的用户模式和内核模式是什么含意？

MS-DOS等操作系统在单一的CPU模式下运行，但是一些类Unix的操作系统则使用了双模式，可以有效地实现时间共享。在Linux机器上，CPU要么处于受信任的内核模式，要么处于受限制的用户模式。除了内核本身处于内核模式以外，所有的用户进程都运行在用户模式之中。

内核模式的代码可以无限制地访问所有处理器指令集以及全部内存和I/O空间。如果用户模式的进程要享有此特权，它必须通过系统调用向设备驱动程序或其他内核模式的代码发出请求。另外，用户模式的代码允许发生缺页，而内核模式的代码则不允许。

在2.4和更早的内核中，仅仅用户模式的进程可以被上下文切换出局，由其他进程抢占。除非发生以下两种情况，否则内核模式代码可以一直独占CPU：

(1)它自愿放弃CPU；

(2)发生中断或异常。

2.6内核引入了内核抢占，大多数内核模式的代码也可以被抢占。

**对Linux内核中进程上下文和中断上下文的理解**

<http://www.embedu.org/Co lumn/Column240.htm>

3) 怎样申请大块内核内存？

通过伙伴系统申请内核内存,以页框为单位申请

alloc\_pages()/alloc\_page

free\_pages/free\_page

在系统启动完之前（也就是mem\_init()函数执行之前）用类似alloc\_bootmem()的函数来申请就可以，这样申请的内存空间系统启动之后不会被Linux内核管理，也就是不会被内核回收，内核不知道有这段内存，你可以自己放心使用；

参考ldd3 P230如果必须使用引导时分配，应该将驱动程序直接链接到内核。

4) 用户进程间通信主要哪几种方式？

（1）管道（Pipe）：管道可用于具有亲缘关系进程间的通信，允许一个进程和另一个与它有共同祖先的进程之间进行通信。这里指的应该是最普通的匿名管道。

* 他们是半双工的 ，
* 他们只能在具有公共祖先的进程之间使用，通常，一个管道由一个进程创建(调用pipe系统调用)，然后该进程fork，由于文件描述符的进程可继承性，此后父子进程之间就可以应用该管道。int pipe(int filedes[2]); 调用成功后filedes[1]向管道写入数据，而另一个filedes[0]可以读出。具体描述参考APUE.

对比Windows的匿名管道机制：匿名管道

　　管道(Pipe)是一种具有两个端点的通信通道：有一端句柄的进程可以和有另一端句柄的进程通信。管道可以是单向－一端是只读的，另一端点是只写的；也可以是双向的一管道的两端点既可读也可写。

　　匿名管道(Anonymous Pipe)是 在父进程和子进程之间，或同一父进程的两个子进程之间传输数据的无名字的单向管道。通常由父进程创建管道，然后由要通信的子进程继承通道的读端点句柄或写 端点句柄，然后实现通信。父进程还可以建立两个或更多个继承匿名管道读和写句柄的子进程。这些子进程可以使用管道直接通信，不需要通过父进程。

　　匿名管道是单机上实现子进程标准I/O重定向的有效方法，它不能在网上使用，也不能用于两个不相关的进程之间。

（2）命名管道（named pipe）：命名管道克服了管道没有名字的限制，因此，除具有管道所具有的功能外，它还允许无亲缘关系进程间的通信。命名管道在文件系统中有对应的**文件名**。命名管道通过命令mkfifo或系统调用mkfifo来创建。int mkfifo(const char\* pathname, mode\_t mode);

（3）信号（Signal）：信号是比较复杂的通信方式，用于通知接受进程有某种事件发生，除了用于进程间通信外，进程还可以发送信号给进程本身；linux除了支持Unix早期信号语义函数sigal外，还支持语义符合Posix.1标准的信号函数sigaction（实际上，该函数是基于BSD的，BSD为了实现可靠信号机制，又能够统一对外接口，用sigaction函数重新实现了signal函数）。

信号的实现机制应该不是软中断，具体参考ULK and：

<http://www.oldlinux.org/oldlinux/viewthread.php?tid=10564>

<http://www.ibm.com/developerworks/cn/linux/l-ipc/part2/index1.html>

（4）消息（Message）队列：消息队列是消息的链接表，包括Posix消息队列systemV消息队列。有足够权限的进程可以向队列中添加消息，被赋予读权限的进程则可以读走队列中的消息。消息队列克服了信号承载信息量少，管道只能承载无格式字节流以及缓冲区大小受限等缺。Windows无此说法

（5）共享内存：使得多个进程可以访问同一块内存空间，是最快的可用IPC形式。是针对其他通信机制运行效率较低而设计的。往往与其它通信机制，如信号量结合使用，来达到进程间的同步及互斥。

shmget

shmctl

**（6）信号量（semaphore）：主要作为进程间以及同一进程不同线程之间的同步手段。- *我觉得这条应该不是属于进程间通讯的概念，而是内核同步的概念。*** 但是参考APUE，却发现在进程间通信一章，将消息队列，信号量和共享内存并列作为SystemV引入的三大进程间通信机制。Windows无此说法

（7）内存映射（mapped memory）：内存映射允许任何多个进程间通信，每一个使用该机制的进程通过把一个共享的文件映射到自己的进程地址空间来实现它。

void \*mmap(void\*start,size\_t length,int prot,int flags,int fd,off\_t offset);

int munmap(void\* start,size\_t length);

（8）套接口（Socket）：更为一般的进程间通信机制，可用于不同机器之间的进程间通信。起初是由Unix系统的BSD分支开发出来的，但现在一般可以移植到其它类Unix系统上：Linux和System V的变种都支持套接字。

5) 通过buddy伙伴系统申请内核内存的函数有哪些？

alloc\_pages()/alloc\_page

free\_pages/free\_page

简而言之，buddy系统适合分配page（4k）为单位的内存，而slab系统适合分配更小的内存，slab是基于buddy开发，会先向buddy申请再细分后分配给用户。

6) 通过slab分配器申请内核内存的函数有？

kmem\_cache\_create()/kmem\_cache\_destroy()

kmem\_cache\_alloc()/kmem\_cache\_free()

kmalloc()/kfree() ，内部调用了kmem\_catch\_alloc

# （经典）Linux slab 分配器剖析:

# <http://www.ibm.com/developerworks/cn/linux/l-linux-slab-allocator/>

Linux 的記憶體管理: http://ccckmit.wikidot.com/lk:memory

[内存管理简介之Buddy算法和slab分配](http://blog.csdn.net/yming0221/article/details/6591412): <http://blog.csdn.net/yming0221/article/details/6591412>

一个更详细的介绍**Linux内存管理Slab分配器：**<http://www.uml.org.cn/embeded/201210165.asp>

7) Linux的内核空间和用户空间是如何划分的（以32位系统为例）？

每个进程的虚存空间都是4G，其中顶端的1G是系统空间（内核空间），剩下的3G即为用户空间；这是就虚存空间而言的。而具体到物理内存上，系统空间却总是从0开始，这其间就有一个0XC0000000的偏移，而那也正好就是用户虚存空间的上限。

不知有没说清楚？

PAGE\_OFFSET

8) vmalloc()申请的内存有什么特点？

Linux设备驱动编程之内存与I/O操作：

<http://tech.sina.com.cn/s/2006-10-27/15181207636.shtml>

9) 用户程序使用malloc()申请到的内存空间在什么范围？

最大0x40000000, 最小取决于读写段的上限。

10) 在支持并使能MMU的系统中，Linux内核和用户程序分别运行在物理地址模式还是虚拟地址模式？

虚拟地址模式

11) ARM处理器是通过几级页表进行存储空间映射的？

两级页表。一级页表中包含以段为单位的地址变换条目或者指向二级页表的指针，一级页表实现的地址映射粒度较大。二级页表中包含以大页、小页和极小页为单位

12) Linux是通过什么组件来实现支持多种文件系统的？

VFS

Linux 文件系统剖析 - <http://www.ibm.com/developerworks/cn/linux/l-linux-filesystem/>

# LINUX VFS精华版PPT

<http://wenku.baidu.com/view/e77321768e9951e79b892705.html>

上文中讲到一个current->namespace的概念：linux/namespace.h

找到如下文章对这个概念有较好的解释：

linux驱动current,引用当前进程，及task\_struct

<http://blog.csdn.net/tommy_wxie/article/details/7282775>

linux vfs 解析 之 mount 文件系统

<http://blog.csdn.net/qqzhangchang/article/details/8116684>

解析 Linux 中的 VFS 文件系统机制

<http://www.ibm.com/developerworks/cn/linux/l-vfs/>

dentry与inode - <http://blog.chinaunix.net/uid-20184656-id-3151111.html>

dentry是一个内存实体,inode是在磁盘上的。

13) Linux虚拟文件系统的关键数据结构有哪些？（至少写出四个）

inode file super\_block dentry

还是可以参考 dentry与inode - <http://blog.chinaunix.net/uid-20184656-id-3151111.html>

14) 对文件或设备的操作函数保存在那个数据结构中？

file\_operations

15) Linux中的文件包括哪些？

* 普通文件 ： 通常是流式文件
* 目录文件 ： 用于表示和管理系统中的全部文件
* 符号链接文件 ： 用于不同目录下文件的共享
* 设备文件 ： 包括块设备文件和字符设备文件，Linux 中的设备有2种类型：字符设备(无缓冲且只能顺序存取)、块设备(有缓冲且可以随机存取)。块设备文件表示磁盘文件、光盘等，字符设备文件包括字符操作终端、键盘等设备。对于USB设备来说，既有USB字符设备，也有USB块设备。
* 管道/命名管道(FIFO)文件 : 提供进程建通信的一种方式
* 套接字(socket) 文件： 该文件类型与网络通信有关

16) 创建进程的系统调用有那些？

参考一个比较精彩的描述：<http://www.chinaunix.net/old_jh/23/62446.html>

1)exec族函数：

头文件 #include<unistd.h> 原型

int execl(const chat \*path,const char \*args,...)

int execv(const char \*path,char const \*argv[])

int execle(const cahr \*path,const char \*arg,...,char \*const envp[])

int execve(const char \*path,char \*const argv[],char \*const envp[])

int execlp(const char \*file,char \*arg,...)

int execvp(const cahr \*file,char \*const argv[])

返回 返回-1表示出错

2)system()函数

头文件 #include<stdlib.h>

功能 在进程中开始另一个进程

原型 int system(const char \*string)

传入值 系统变量

返回值 成功则返回执行shell命令后的返回值，调用/bin/sh数百返回127，其他

失败返回－1，三叔string为空返回非零值

3)fork函数：

头文件 #include<unistd.h>

功能 建立一个新进程（复制进程）

原型 pid\_t fork(void);

返回值 执行成功则在子进程中返回0，在父进程中返回新建进程的PID,

失败返回－1

17) 调用schedule()进行进程切换的方式有几种？

这里是针对2.6内核来说的。

Linux下面调度有以下两种形式的调度方式

主动启动

被动调度

主动启动

当前进程因等待资源而需进入被阻塞状态时，调度程序将进行如下的操作：

首先把当前进程放到适当的等待队列里

把当前进程的状态设为TASK\_INTERRUPTIBLE或者TASK\_UNINTERRUPTIBLE

调用schedule()，让新的进程运行

被动调度

通过在当前进程的need\_resched设为1来请求调度。

设置need\_resched的时机：

当前进程用完了它的CPU时间片，update\_process\_times()重新进行计算

当一个进程被唤醒，而且它的优先级比当前进程高

当sched\_setschedler()或sched\_yield()系统调用被调用时

2.4 kernel:

A. 主动式

　　在核心应用中直接调用schedule()。这通常发生在因等待核心事件而需要将进程置于挂起（休眠）状态的时候--这时应该主动请求调度以方便其他进程使用CPU。

将进程添加到事件等待队列中；

　　置进程状态为TASK\_INTERRUPTIBLE（或TASK\_UNINTERRUPTIBLE）；

　　在循环中检查等待条件是否满足，不满足则调用schedule()，满足了就退出循环；

　　将进程从事件等待队列中删除

B. 被动式

　　在系统调用执行结束后，控制由核心态返回到用户态之前，Linux都将在ret\_from\_sys\_call入口检查当前进程的need\_resched值，如果该值为1，则调用schedule()：

因此，只需要设置当前进程（current）的need\_resched，就有机会启动调度器。

通常有如下几种场合会设置need\_resched：

　　update\_process\_times()，由时钟中断触发，负责管理除0号进程（idle进程）以外的其他各个进程的时间片消耗。如果当前进程（SCHED\_FIFO实时进程除外）的时间片用完了（counter==0），则设置need\_resched为1；（注意：此时并不计算或重置counter值，这个工作在所有进程的时间片都耗完以后在schedule()中进行）

　　reschedule\_idle()，此函数的功能在"调度器工作流程"一节中已经详细描述了，不过，最经常的调用者是在某一事件等待队列上休眠的进程的唤醒过程--wake\_up\_process()及其他一系列wake\_up函数（见上"主动式调度"）；

　　sched\_setscheduler()、sched\_yield()系统调用，以及系统初始化（rest\_init()中）、创建新进程（do\_fork()中）等从语义上就希望启动调度器工作的场合。

　　由于启动schedule()的时机实际上由当前进程决定，因此设置了need\_resched并不意味着就能及时调度，这也是"Linux内核不可抢占"的原因

18) Linux调度程序是根据进程的动态优先级还是静态优先级来调度进程的？

动态优先级----完全公平调度器CFS

19) 进程调度的核心数据结构是哪个？

task\_struct

首先明确一点，多线程并不是指多个线程同时执行（除非你有多个CPU），而是一种并发地执行。从宏观上来看好像是多个线程同时执行，但对于单CPU来讲只能是一个一个的顺序执行。但为了让线程的运行看起来是同时的，就必须由操作系统来统一地进行调度，也就是给每个线程分配CPU资源。

再明确一个概念：抢先式多任务（preemptive task），它指的是一个线程是否需要主动放弃CPU才能使其他线程获得CPU资源。如果是，则为非抢先式多任务（比如windows 3.1）；如果不需要，也就是说一个线程在执行时可以由其他的途径被剥夺CPU使用权（比如中断）的话，就是抢先式多任务。从目前的情况来看几乎所有的操作系统都支持抢先式多任务，所以这个概念提的也就不多了。

任何一个线程在运行时都会在用户空间与内核空间切换（内核线程除外），线程在用户态执行时，操作系统的调度器要想剥夺它的CPU使用权并移交给另一个线程是比较简单的，但如果线程正运行在内核空间那么情况就复杂得多。并不是所有支持抢先式多任务的操作系统的内核都能对一个正在内核空间执行的线程进行线程切换，而能够做到这一点的内核就称之为抢占式内核（preemptible kernel），比如linux 2.6.x内核，windows 2000/xp的内核。当然即使是抢占式内核也并不是说他在内核中所有的地方都能实现抢占。

当然啦，内核可抢占就是为了加快一些进程的响应速度，具体哪些进程可以抢占内核就看它的调度策略和内核的调度算法了。2.6的进对用户进程来说是件好事，因为以前只有被内核抢占，现在可以反过来抢占内核，这就是为什么会感觉2.6的系统会运行得比较快的原因。

刚才略微看了一下2.6，在bh和定时器都有检查点，特别是定时器，驱动没少用定时器的，所以在那做检查，性能应该会有不少提升。建议你多搜2.6的调度算法的资料（当然，熟悉2.4调度那就更好了），再来看，会更清晰的。

Linux内核多线程：

<http://www.cnblogs.com/zhuyp1015/archive/2012/06/11/2545624.html>

Linux内核线程之深入浅出 - 有点老，可能是基于2.4，但有点参考意义

<http://hi.baidu.com/koxkuemvypbabud/item/e60cb0cad45ff30bc610b2c1>

20) 如何加载、卸载一个模块？

21) 模块和应用程序分别运行在什么空间？

22) Linux中的浮点运算由应用程序实现还是内核实现？

由内核实现

23) 模块程序能否使用可链接的库函数？

No. 内核模块有所不同。在hello world模块中您也许已注意到了我们使用的函数 printk() 却没有包含标准I/O库。这是因为模块是在insmod加 载时才连接的目标文档。那些要用到的函数的符号链接是内核自己提供的。 也就是说，您能够在内核模块中使用的函数只能来自内核本身。

24) TLB中缓存的是什么内容？

TLB:Translation Lookaside Buffer.

根据功能可以译为快表，直译可以翻译为旁路转换缓冲，也可以把它理解成页表缓冲。里面存放的是一些页表文件（虚拟地址到物理地址的转换表）。当处理器要在主内存寻址时，不是直接在内存的物理地址里查找的，而是通过一组虚拟地址转换到主内存的物理地址，TLB就是负责将虚拟内存地址翻译成实际的物理内存地址，而CPU寻址时会优先在TLB中进行寻址。处理器的性能就和寻址的命中率有很大的关系。

25) Linux中有哪几种设备？

字符设备

一个字符( char ) 设备是一种可以当作一个字节流来存取的设备( 如同一个文件 ); 一个字符驱动负责实现这种行为. 这样的驱动常常至少实现 open, close, read, 和 write 系统调用. 文本控制台( /dev/console )和串口( /dev/ttyS0 及其友 )是字符设备的例子, 因为它们很好地展现了流的抽象. 字符设备通过文件系统结点来存取, 例如 /dev/tty1 和 /dev/lp0. 在一个字符设备和一个普通文件之间唯一有关的不同就是, 你经常可以在普通文件中移来移去, 但是大部分字符设备仅仅是数据通道, 你只能顺序存取.然而, 存在看起来象数据区的字符设备, 你可以在里面移来移去. 例如, frame grabber 经常这样, 应用程序可以使用 mmap 或者 lseek 存取整个要求的图像.

块设备

如同字符设备, 块设备通过位于 /dev 目录的文件系统结点来存取. 一个块设备(例如一个磁盘)应该是可以驻有一个文件系统的. 在大部分的 Unix 系统, 一个块设备只能处理这样的 I/O 操作, 传送一个或多个长度经常是 512 字节( 或一个更大的 2 的幂的数 )的整块. Linux, 相反, 允许应用程序读写一个块设备象一个字符设备一样 -- 它允许一次传送任意数目的字节. 结果就是, 块和字符设备的区别仅仅在内核在内部管理数据的方式上, 并且因此在内核/驱动的软件接口上不同. 如同一个字符设备, 每个块设备都通过一个文件系统结点被存取的, 它们之间的区别对用户是透明的. 块驱动和字符驱动相比, 与内核的接口完全不同.

网络接口

任何网络事务都通过一个接口来进行, 就是说, 一个能够与其他主机交换数据的设备. 通常, 一个接口是一个硬件设备, 但是它也可能是一个纯粹的软件设备, 比如环回接口. 一个网络接口负责发送和接收数据报文, 在内核网络子系统的驱动下, 不必知道单个事务是如何映射到实际的被发送的报文上的. 很多网络连接( 特别那些使用 TCP 的)是面向流的, 但是网络设备却常常设计成处理报文的发送和接收. 一个网络驱动对单个连接一无所知; 它只处理报文.

既然不是一个面向流的设备, 一个网络接口就不象 /dev/tty1 那么容易映射到文件系统的一个结点上. Unix 提供的对接口的存取的方式仍然是通过分配一个名子给它们( 例如 eth0 ), 但是这个名子在文件系统中没有对应的入口. 内核与网络设备驱动间的通讯与字符和块设备驱动所用的完全不同. 不用 read 和 write, 内核调用和报文传递相关的函数.

有其他的划分驱动模块的方式, 与上面的设备类型是正交的. 通常, 某些类型的驱动与给定类型设备的其他层的内核支持函数一起工作. 例如, 你可以说 USB 模块, 串口模块, SCSI 模块, 等等. 每个 USB 设备由一个 USB 模块驱动, 与 USB 子系统一起工作, 但是设备自身在系统中表现为一个字符设备( 比如一个 USB 串口 ), 一个块设备( 一个 USB 内存读卡器 ), 或者一个网络设备( 一个 USB 以太网接口 ).

26) 字符设备驱动程序的关键数据结构是哪个？

file\_operations

file

inode

27) 设备驱动程序包括哪些功能函数？

register prob release remove

28) 如何唯一标识一个设备？

主设备号＋次设备号

29) Linux通过什么方式实现系统调用？

[Linux下系统调用的实现](http://blog.csdn.net/lrlsqrf/article/details/4638668)： <http://blog.csdn.net/lrlsqrf/article/details/4638668>

**在每种平台上，都有特定的指令可以使进程的执行由用户态转换为核心态，这种指令称作操作系统陷入（operating system trap）。进程通过执行陷入指令后，便可以在核心态运行系统调用代码。**

**在Linux中是通过软中断来实现这种陷入的，在x86平台上，这条指令是int 0x80。**

30) Linux软中断和工作队列的作用是什么？

软中断包括Tasklet，Tasklet也是基于软中断机制实现的

工作队列是另一种延迟执行技术，但不是基于软中断技术实现的。

**Linux中软中断机制分析** <http://bbs.ednchina.com/BLOG_ARTICLE_135152.HTM>

[linux 中断分析专辑](http://blog.chinaunix.net/uid-28236237-id-3434417.html) - <http://blog.chinaunix.net/uid-28236237-id-3434417.html> 以及该blog下的一系列文章，有关软中断和工作队列。

linux2.6软中断-tasklet（很长需要仔细看）<http://www.cnblogs.com/flyxiang2010/archive/2010/12/24/1915821.html>

软中断和tasklet运行于中断上下文，而工作队列则运行于进程上下文。因此，软中断和tasklet处理函数中不能睡眠，而工作队列处理函数中允许睡眠

软中断在smp中是可能被同时运行的，所以软中断的处理函数必须被编写成可重入的函数而且必须明确地使用自旋锁保护其数据结构。  
但tasklet是不会在多个cpu之中同时运行的，因为内核在基于软中断定义的基础上定义的Tasklet，对Tasklet的执行添加了更加严格的控制，保证相同类型的Tasklet总是串行，即一个CPU上如果正在执行，另一个CPU上就必须等待。类型不同的Tasklet还是可以在SMP上并发执行的。所以tasklet的处理函数可以编写成不可重入的函数，这样就减轻了编程人员（特别是设备驱动开发人员）的负担。

软中断： 1、软中断是在编译期间静态分配的。

2、最多可以有32个软中断。

3、软中断不会抢占另外一个软中断，唯一可以抢占软中断的是中断处理程序。

4、可以并发运行在多个CPU上（即使同一类型的也可以）。所以软中断必须设计为可重入的函数（允许多个CPU同时操作），

因此也需要使用自旋锁来保护其数据结构。

5、目前只有两个子系直接使用软中断：网络和SCSI。

6、执行时间有：从硬件中断代码返回时、在ksoftirqd内核线程中和某些显示检查并执行软中断的代码中。

tasklet： 1、tasklet是使用两类软中断实现的：HI\_SOFTIRQ和TASKLET\_SOFTIRQ。本质上没有什么区别，只不过HI\_SOFTIRQ的优先级更高一些，建立在HI\_SOFTIRQ上的tasklet会早于TASKLET\_SOFTIRQ执行。

2、可以动态增加减少，没有数量限制。

3、同一类tasklet不能并发执行。

4、不同类型可以并发执行。

5、大部分情况使用tasklet。

工作队列： 1、由内核线程去执行，换句话说总在进程上下文执行。

2、可以睡眠，阻塞。