## 内核抢占：

1. 请求内核服务的用户态进程必须发出适当的指令。这些指令引起一个异常，它迫使CPU从用户态切换到内核态。
2. 定义：
3. 如果进程正在执行内核函数时，即它在内核态运行时，允许发生内核切换（被替换的进程是正执行内核函数的进程），这个内核就是抢占的。
4. 无论是抢占内核还是非抢占内核中，运行在内核态的进程都可以自动放弃CPU，比如，其原因可能是，进程由于等待资源而不得不转入睡眠状态。我们将这种进程切换叫做计划性进城切换。但是，抢占式内核在响应引起进程切换的异步事件（如唤醒高优先权进程的中断处理程序）的方式上与非抢占的内核是有差别的，我们将这种进程切换称作强制性进程切换。
5. 所有进程切换都由宏switch\_to来完成。在抢占式内核和非抢占内核中，当进程执行完某些内核功能的线程，而且调度程序被调用后，就发生进程切换。不过，在非抢占内核中，当前进程是不可能被替换的，除非它打算切换到用户态。
6. 主要特点：一个在内核态运行的进程，可能在执行内核函数期间被另外一个进程取代。
7. 抢占内核和非抢占内核的例子：
8. 在进程A执行异常处理程序时（肯定是在内核态），一个具有较高优先级的进程B变为可执行状态。这种情况是有可能的，例如，发生了中断请求而且相应的处理程序唤醒了进程B。如果内核是抢占的，就会发生强制性进程切换，让进程B取代进程A。异常处理程序的执行被暂停，直到调度程序再次选择进程A时才恢复它的执行。相反，如果内核是非枪战的，在进程A完成异常处理程序的执行之前是不会发生进城切换的，除非A主动放弃CPU。
9. 一个异常处理程序的进程已经用完了它的时间配额的情况。如果内核是抢占的，进程可能会立即被取代，但如果内核是非抢占的，进程继续运行直到它执行完异常处理程序或自动放弃CPU。
10. 使内核可抢占的目的是减少用户态进程的分派延迟（从进程变为可执行状态到它实际开始运行之间的时间间隔）。内核抢占对执行及时被调度的任务（如：硬件控制器、环境监视器、电影播放器等等）的进程确实是有好处的，因为它降低了这种进程被另外一个运行在内核态的进程延迟的风险。
11. 只有当内核正在执行异常处理程序（尤其是系统调用），而且内核抢占没有被显式的禁止时，才可能抢占内核。本地CPU必须打开本地中断，否则无法完成内核抢占。
12. preempt\_count字段：为了保证Linux内核在不能被抢占状态下不会被抢占，抢占式内核使用了一个变量preempt\_count变量，称为内核抢占锁。这一个变量被设置在进程的PCB结构task\_struct中。每当内核进入不能被抢占状态时，变量preempt\_count加1，指示内核不允许被抢占。当内核从不能被抢占状态退出时，变量preempt\_count减1，同时进行可抢占的判断与调度。
13. 处理preempt\_count字段的抢占计数器的宏：
14. preempt\_count() ：在thread\_info描述符中选择preempt\_count字段
15. preempt\_disable() ：使抢占计数器的值加１
16. preempt\_enable\_no\_resched() ：使抢占计数器的值减1
17. preempt\_enable() ：使抢占计数器的值减1，并在thread\_info描述符的TIF\_NEED\_RESCHED标志被置为1的情况下，调用preempt\_schedule()
18. get\_cpu() ：与preempt\_disable()相似，但要返回本地CPU的数量。
19. put\_cpu() ：与preempt\_enable()相同
20. put\_cpu\_no\_resched() ：与preempt\_enable\_no\_resched()相同

// 相关的宏：

1. PREEMPT\_ACTIVE：防止将处于非TASK\_RUNNING状态的进程并且没有没有在任何睡眠队列的进程移出运行队列。
2. irqs\_disabled()：检查是否禁用本地中断。
3. 内核抢占可能在内核控制路径（通常是一个中断处理程序）结束时发生，也可能在异常处理程序调用preemt\_enable()重新允许内核抢占时发生。

10、内核抢占会引起不容忽视地开销。因此，Linux2.6允许用户在编译内核时通过设置选项来禁用或启用内核抢占。

11、竞争条件：多个进程并发访问和操作同一数据且执行结果与访问的特定顺序有关

## 同步原语：

1. 内核使用的各种同步技术：

技术 说明 适用范围

（1）每CPU变量 在CPU之间复制数据结构 所有CPU

（2）原子操作 对一个存储单元原子地“读-修改-写”的指令 所有CPU

（3）内存屏障 避免指令重新排序 所有CPU

（4）自旋锁 加锁时忙等 所有CPU

（5）顺序锁 基于访问计数器的锁 所有CPU

（6）本地中断的禁止 禁止单个CPU上的中断处理 本地CPU

（7）本地软中断的禁止 禁止单个CPU上的可延迟函数处理 本地CPU

（8）读-拷贝-更新（RCU）通过指针而不是锁来访问共享数据结构 所有CPU

1. 每CPU变量：
2. 最简单也是最重要的同步技术包括把内核变量声明为每CPU变量。每CPU变量主要是数据结构的数组，系统的每个CPU对应数组的一个元素。
3. 一个CPU不应该访问与其他CPU对应的数组元素。另外，它可以随意读或修改它自己的元素而不用担心出现竞争条件。因为它是唯一有资格这么做的CPU。每CPU变量只有在确定系统的CPU上的数据在逻辑上是独立的时候才可以使用。
4. 每CPU的数组元素在主存中被排列以使每个数据结构存放在硬件高速缓存的不同行。因此，对每CPU数组的并发访问不会导致高速缓存行的窃用和失效。
5. 虽然CPU变量为来自不同CPU的并发访问提供保护，但对来自异步函数（中断处理程序和可延迟函数）的访问不提供保护，在这种情况下需要另外的同步原语。
6. 在单处理器和多处理器系统中，内核抢占都可能使每CPU变量产生竞争条件。总的原则是内核控制路径应该在禁用抢占的情况下访问没CPU变量。
7. 为每CPU变量提供的函数和宏：

宏或函数名 说明

DEFINE\_PER\_CPU(type, name) 静态分配一个每CPU数组，数组名为name，结构类型为type；

per\_cpu(name, cpu) 为CPU选择一个每CPU数组元素，CPU由参数cpu指定，数组名为name；

\_\_get\_cpu\_var(name)选择每CPU数组name的本地CPU元素；

get\_cpu\_var(name)先禁用内核抢占，然后再每CPU数组name中，为本地CPU选择元素；

put\_cpu\_var(name)启用内核抢占（不使用name）

alloc\_percpu(type)动态分配type类型数据结构的每CPU数组，并返回它的地址。

free\_percpu(pointer)释放被动态分配的每CPU数组，pointer指示其地址

per\_cpu\_ptr(pointer, cpu)返回CPU数组中与参数cpu对应的CPU元素地址，参数pointer给出数组地址。

1. 原子操作：
2. 概念：对一个存储单元原子地“读-修改-写”的指令
3. Linux内核提供了一个专门的atomic\_t类型（一个原子访问计数器）和一个专门的函数和宏，这些函数和宏作用于atomic\_t类型的变量，并当作单独的、原子的汇编语言指令来使用。在多处理系统中，每条这样的指令都有一个lock字节的前缀（操作码前缀是lock字节(0xf0)的“读-修改-写”汇编语言指令即使在多处理器系统中也是原子的，当控制单元检测到这个前缀，就锁定内存总线，直到这条指令执行完成为止，因此，当加锁的指令执行时，其他处理器就不能访问这个内存单元）。
4. Linux中的原子操作：

函数 说明

atomic\_read(v) 返回\*v

atomic\_set(v, i) 把\*v设置成i

atomic\_add(i, v) 给\*v增加i

atomic\_sub(I, v) 给\*v减去i

atomic\_sub\_and\_test(i, v) 从\*v中减去i，如果结果为0，则返回1，否则，返回0

atomic\_inc(v) 给\*v加1

atomic\_dec(v) 给\*v减1

atomic\_dec\_and\_test(v) 从\*v减1，如果结果为0，则返回1，否则，返回0

atomic\_inc\_and\_test(v) 把\*v加1，如果结果为0，则返回1，否则，返回0

atomic\_add\_negative(i, v) 把\*v加上i，若结果为负数，则返回1，否则，返回0

atomic\_inc\_return(v) 把\*v加上1，返回\*v的新值

atomic\_dec\_return(v) 把\*v减1，返回\*v的新值

atomic\_add\_return(i, v) 把\*v加上i，返回\*v的新值

atomic\_sub\_return(i, v) 把\*v减i，返回\*v的新值

1. Linux中的原子位处理函数：

函数 说明

test\_bit(nr, addr) 返回\*addr的第nr位的值

1. 内存屏障：
2. 优化屏障原语：优化屏障原语保证编译程序不会混淆放在原语操作之前的汇编语言指令和放在原语操作之后的汇编语言指令，这些汇编语言指令在C中都有对应的语句。在Linux优化屏障就是barrier()宏，它展开为asm volatile(“”:::”memory”)。指令asm告诉编译器程序要出入汇编语言片段。Volatile关键字禁止编译器把asm指令与程序中其他指令重新组合。Memoey关键字强制编译器假定RAM中的所有内存单元已经被汇编语言指令修改，因此，编译器不能使用存放在CPU寄存器中的内存单元的值类优化asm指令前的代码。注意，优化屏障原语不保证不使当前CPU把汇编语言指令混在一起执行——这是内存屏障的工作。
3. 内存屏障原语：确保在这个原语之后的操作开始执行之前，原语之前的操作已经完成。因此，内存屏障类似于防火墙，让任何汇编语言指令都不能通过。
4. 自旋锁：
5. 当内核控制路径必须访问共享数据结构或进入临界区时，就需要为自己获取一把“锁”。
6. 概念：自旋锁是用来处理多处理器环境中工作的一种特殊的锁。如果内核控制路径发现自旋锁“开着”，就获取锁并继续自己的执行。相反，如果内核控制路径发现锁由运行在另一个CPU上的内核控制路径“锁着”，就在周围“旋转”，反复执行一条紧凑的循环指令，直到锁被释放。
7. 一般来说，由自旋锁所保护的每个临界区都是禁止内核抢占的。在单处理器系统上，这种锁本身并不起锁的作用，自旋锁原语仅仅是禁止或启动内核抢占。请注意，在自旋锁忙等期间，内核抢占还是有效的，因此，等待自选所释放的进程有可能被更高优先级进程替代。
8. 在Linux中，每个自旋锁都用spinlock\_t结构表示，其中包含两个字段：

slock: 该字段表示自旋锁的状态，值为1表示“未加锁”状态，而任何负数和0都表示“加锁”状态。

Break\_lock：表示有进程正在忙等自旋锁（只有内核支持SMP和内核抢占的情况下使用该标志）。通过检测该字段，拥有锁并在其他CPU上运行的进程可以知道是否有其他进程在等待这个锁。

1. 自旋锁宏：

spin\_lock\_init()：把自旋锁置为1（未锁）

spin\_lock()：循环，直到自旋锁变为1（未锁），然后把自旋锁置为0（锁上）

spin\_unlock()：把自旋锁置为1（未锁）

spin\_unlock\_wait()：等待，直到自旋锁变为1（未锁）

spin\_is\_locked()：如果自旋锁被置为1（未锁），返回0，否则，返回1

spin\_trylock()：把自旋锁置为0（锁上），如果原来锁的值是1，则返回1，否则，返回0

read\_lock：作用于读/写自旋锁的地址

1. 读写自旋锁：
2. 读/写自旋锁的引入是为了增加内核的并发能力。只要没有内核控制路径对数据结构进行修改，读/写自旋锁就允许多个内核控制路径同时读同一数据结构。如果一个内核控制路径想对这个结构进行写操作，那么它必须首先获取读/写锁的写锁，写锁授权独占访问这个资源。
3. 每个读/写锁都是一个rwlock\_t结构
4. 当使用读/写自旋锁时，内核控制路径发出的执行的read\_lock和write\_lock操作的请求具有相同的优先权：读者必须等待，直到写操作完成。同样的，写者也必须等待，直到读操作完成。
5. 顺序锁：
6. 顺序者与读/写自旋锁非常相似，只是它为写者赋予了较高的优先级：事实上，即使读者正在读的时候也允许写者继续运行。这种策略的好处是写者永远不会等待（除非另外一个写者正在写），缺点是有些时候读者不得不反复读相同的数据直到它获得有效的副本。
7. 每个顺序锁都是包括两个字段的seqlock\_t结构：一个类型为spinlock\_t的lock字段和一个整形的sequence字段，第二个字段是一个顺序计数器。每个读者都必须在读数据前后两次读顺序计数器，并检查两次读到的值是否相同，如果不相同，如果不相同，说明新的写者已经开始写并增加了顺序计数器，因此暗示读者刚读到数据是无效的。
8. 读-拷贝-更新（RCU）
9. RCU是为了保护在多数情况下被多个CPU读的数据结构而设计的另外一种同步技术。RCU允许多个读者和写者并发执行。
10. 限制：
11. RCU只保护被动态分配并通过指针引用的数据结构。
12. 在被RCU保护的临界区中，任何内核控制路径都不能睡眠
13. 信号量：
14. Linux提供两种信号量：
15. 内核信号量，由内核控制路径使用
16. System V IPC信号量，由用户态进程使用
17. 内核信号量：当内核控制路径试图获取内核信号量所保护的忙资源时，相应的进程被挂起。只有资源被释放时，进程才再次变为可运行的。因此，只有可以睡眠的函数才能获取内核信号量，中断处理成和可延迟函数都不能使用内核信号量。
18. 读/写信号量：
19. 读/写信号量类似于前面描述的”读/写自旋锁”，有一点不同：在信号量再次变为打开之前，等待进程挂起而不是自旋。
20. 很多读者内核控制路径可以并发地获取读/写信号量。但是，任何写者内核控制路径必须有对被保护资源的互斥访问。因此，只有在没有内核控制路径为读访问或者写访问持有信号量时，写者才可以获取信号量。读/写信号量可以提高内核中的并发度，并改善了整个系统的性能。
21. 内核以严格的FIFO顺序处理等待读/写信号量的所有进程。如果读者或写者进程发现信号量关闭，这些进程就被插入到信号量等待队列链表的末尾。当信号量被释放时，就检查处于等待队列链表第一个位置的进程。第一个进程常被唤醒。如果是一个写者进程，等待队列上其他的进程就继续睡眠。如果是一个读者进程，那么紧跟第一个进程的其他所有读者进程也被唤醒并获得锁。不过，在写者进程之后排队的读者进程继续睡眠。
22. 补充原语：
23. 补充原语和信号量之间真正的差别在于如何使用等待队列中包含的自旋锁。在补充原语中，自旋锁用来确保compelete()和wait\_for\_completion()不会并发执行。在信号量中，自旋锁用于避免并发执行的down()函数弄乱信号量的数据结构。

## 禁止和激活可延迟函数：

1. 可延迟函数可能在不可预知的时间执行。因此，必须保护可延迟函数访问的数据结构使其避免竞争条件。
2. 禁止可延迟函数在一个CPU上执行的一种简单方式就是禁止那个CPU上的中断。因为没有中断处理程序被激活，软中断操作就不能异步地开始。
3. 如果preempt\_count字段中存放的软中断计数器是正数，do\_softirq()函数就不会执行软中断，而且，因为tasklet在软中断之前被执行，把这个计数器设置为大于0的值，由此禁止了在给定CPU上的所有可延迟函数和软中断函数的执行。