深入理解 Linux 内核 第五章内核同步

Contents

RT Embedded http://www.kontronn.com

第二章 内存寻址

内核如何为不同的请求提供服务

内核抢占

如果进程正执行内核函数时，即它在内核态运行时，允许发生内核切换，这个内核就是抢占的。

同步原语

各种同步技术：

技术	说明	适用范围
每 CPU 变量	在 CPU 之间复制数据结构	所有 CPU
原子操作	对一个计数器原子地“读-修改-写”指令	所有 CPU
内存屏障	避免指令重新排序	本地 CPU 或所有 CPU
自旋锁	加锁时忙等	所有 CPU
信号量	加锁时阻塞等待	所有 CPU
顺序锁	基于访问计数器的锁	所有 CPU
本地中断的禁止	禁止单个 CPU 上的中断处理	本地 CPU
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
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

每 CPU 变量

per-cpu variable 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。一个 CPU 不应该访问与其他 CPU 对应的数据元素。

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止抢占的情况下访问每 CPU 变量。（原因：一个内核控制路径获得了其每 CPU 变量本地副本的地址，然后因被抢占而转移到另外一个 CPU 上，但仍然引用原来 CPU 元素的地址。）

原子操作

atomic operations 确保相关的操作在芯片级是原子的，任何一个操作都必须以单个指令执行，中间不能终端，且避免其他 CPU 访问同一存储器单元。

优化和内存屏障

当使用优化的编译器时，编译器可能重新安排汇编语言指令以使寄存器以最优的方式使用。此外，现代 CPU 通常并行地执行若干条指令，且可能重新安排内存访问。

优化屏障 optimization barrier 原语保证编译器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。**Linux** 中，优化屏障就是 barrier()宏，展开为 asm volatile(“”:::”memory”)。asm 告知编译程序要插入汇编语言片段，volatile 关键字禁止编译器把 asm 指令与程序中的其他指令重新组合。memory 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。
注： 优化屏障并不包装不使当前 CPU 把汇编语言指令混在一起执行——这是内存屏障的作用。

内存屏障 memory barrier 原语确保，在原语之后的操作开始执行之前，原语之前的操作已经完成。
Linux 中的内存屏障

宏	说明
mb	适用于 MP 和 UP 的内存屏障
rmb	适用于 MP 和 UP 的读内存屏障
wmb	适用于 MP 和 UP 的写内存屏障
smp_mb	适用于 MP 的内存屏障
smp_rmb	适用于 MP 的读内存屏障
smp_wmb	适用于 MP 的写内存屏障

自旋锁

spin lock 是用来在多处理器环境中工作的一种特殊的锁。自旋锁的循环指令表示忙等。如果内核控制路径发现自旋锁“开着”，就会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，就在周炜旋转，反复执行一条紧凑的循环指令，知道锁被释放。

在单处理器系统上，这种锁并不起作用。

顺序锁

seqlock 与读写自旋锁非常相似，但是它为写者赋予了较高的优先级：即使在读者正在读的时候也运行写者继续运行。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多次读相同的数据直到其获得有效的副本。

顺序锁 seqlock_t 结构包含：一个 spinlock_t 的 lock 字段和一个整型的 sequence 字段。每个读者必须在读数据前后两次读顺序计数器 sequence，并检查前后读的值是否相同，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，按时读者刚读到的数据是无效的。

一般来说，满足下列条件才能使用顺序锁：

- 被保护的数据结构不包括被写者修改和被读者间接引用的指针。
- 读者的临界区代码应该没有副作用（否则，多个读者的操作会与单独的读操作有不同的结果）

读-拷贝-更新 RCU

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一种同步技术。它允许多个读者和写者并发执行。RCU 不使用锁。

信号量

信号量实现了一个加锁原语，即让等待者睡眠，直到等待的资源变为空闲。

Linux 提供两种信号量：

- 内核信号量，由内核控制路径使用
- system v IPC 信号量，由用户态进程使用

完成量

completion 是为了解决多处理器系统上发生的一种微妙的竞争条件。

补充原语和信号量之间的真正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 complete 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。

There are two reasons you might want to use a completion instead of a semaphore. First, multiple threads can wait for a completion, and they can all be released with one call to complete_all(). It’s more complex to have a semaphore wake up an unknown number of threads.

Second, if the waiting thread is going to deallocate the synchronization object, there is a race condition if you’re using semaphores. That is, the waiter might get woken up and deallocate the object before the waking thread is done with up(). This race doesn’t exist for completions. (See Lasse’s post.)

禁止本地中断

禁止本地终端并不保护运行在另一个 CPU 上的中断处理程序对数据结构的并发访问，因此，在多处理器系统上，禁止本地中断经常与自旋锁结合使用。

对内核数据结构的同步访问

系统的并发度取决于两个主要因素：

- 同时运转的 IO 设备数
- 进行有效工作的 CPU 数

内核控制路径访问的数据结构所需要的保护

访问数据结构的内核控制路径	单处理器保护	多处理器进一步保护
异常	信号量	无
中断	本地中断禁止	自旋锁
可延迟函数	无	无或自旋锁
异常与中断	本地中断禁止	自旋锁
异常与可延迟函数	本地软中断禁止	自旋锁
中断与可延迟函数	本地中断禁止	自旋锁
异常、中断与可延迟函数	本地中断禁止	自旋锁

避免竞争条件的实例

引用计数器广泛地用在内核中以避免由于资源的并发分配和释放而产生的竞争条件。reference counter 只不过是一个 atomic_t 计数器，与特定的资源，如内存页、模块或文件相关。

RT Embedded http://www.kontronn.com