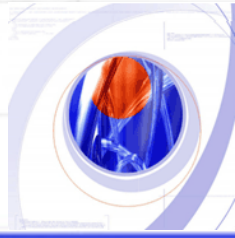
第8章 多处理器

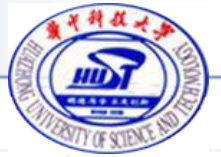
8.1 多处理器概念

8.2 对称式共享储存器系统结构

8.3 同步

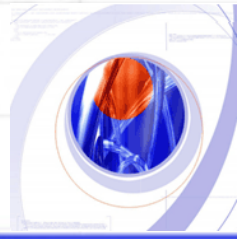
8.4 同步性能问题





8.4.3 同步性能问题

简单旋转锁不能很好地适应可缩扩性。大规模多处理机中，若所有的处理器都同时争用同一个锁，则会导致大量的争用和通信开销。



例8.3 假设某条总线上有10个处理器同时准备对同一变量加锁。如果每个总线事务处理（读不命中或写不命中）的时间是100个时钟周期，而且忽略对已调入Cache中的锁进行读写的时间以及占用该锁的时间。

（1）假设该锁在时间为0时被释放，并且所有处理器都在旋转等待该锁。问：所有10个处理器都获得该锁所需的总线事务数目是多少？

（2）假设总线是非常公平的，在处理新请求之前，要先全部处理好已有的请求。并且各处理器的速度相同。问：处理10个请求大概需要多少时间？

解 当i个处理器争用锁的时候，它们都各自完成以下操作序列，每一个操作产生一个总线事务：

- 访问该锁的i个LL指令操作
- 试图占用该锁（并上锁）的i个SC指令操作
- 1个释放锁的存操作指令

因此对于i个处理器来说，一个处理器获得该锁所要进行的总线事务的个数为2i+1。

由此可知，对n个处理器，总的总线事务个数为：

$$\sum_{i=1}^{n}(2i+1)=n(n+1)+n=n^2+2n$$

对于10个处理器来说，其总线事务数为120个，需要12000个时钟周期。

- 本例中问题的根源：锁的争用、对锁进行访问的串行性以及总线访问的延迟。
- 旋转锁的主要优点：总线开销或网络开销比较低，而且当一个锁被同一个处理器重用时具有很好的性能。

1. 如何用旋转锁来实现一个常用的高级同步原语：栅栏

- 栅栏强制所有到达该栅栏的进程进行等待，直到全部的进程到达栅栏，然后释放全部的进程，从而形成同步。

- 栅栏的典型实现

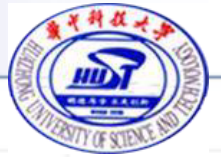
  用两个旋转锁：

  - 用来保护一个计数器，它记录已到达该栅栏的进程数；
  - 用来封锁进程直至最后一个进程到达该栅栏。

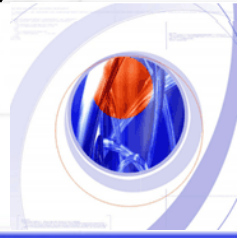
- 一种典型的实现

  其中：

  - lock和unlock提供基本的旋转锁
  - 变量count记录已到达栅栏的进程数
  - total规定了要到达栅栏的进程总数



```
lock (counterlock) ;                    //  确保更新的原子性
    if (count==0) release=0;    //  第一个进程则重置release
    count=count+1;                      //  到达进程数加1
    unlock (counterlock) ;              //  释放锁
    if (count==total) {                 //  进程全部到达
        count=0;                        //  重置计数器
        release=1;                      //  释放进程
        }
    else {                              //  还有进程未到达
        spin (release=1) ;              //  等待别的进程到达
        }
```



- 对counterlock加锁保证增量操作的原子性。
- release用来封锁进程直到最后一个进程到达栅栏。
- spin（release=1）使进程等待直到全部的进程到达栅栏。

实际情况中会出现的问题

栅栏通常是在循环中使用，从栅栏释放的进程运行一段后又会再次返回栅栏，这样有可能出现某个进程永远离不开栅栏的状况(它停在旋转操作上)。

- 一种解决方法

  当进程离开栅栏时进行计数（和到达时一样），在上次栅栏使用中的所有进程离开之前，不允许任何进程重用并初始化本栅栏。但这会明显增加栅栏的延迟和竞争。

- 另一种解决办法
  - 采用sense_reversing栅栏，每个进程均使用一个私有变量local_sense，该变量初始化为1。
  - sense_reversing栅栏的代码

优缺点：使用安全，但性能比较差。

对于10个处理器来说，当同时进行栅栏操作时，如果忽略对Cache的访问时间以及其它非同步操作所需的时间，则其总线事务数为204个，如果每个总线事物需要100个时钟周期，则总共需要20400个时钟周期。

```
local_sense=! local_sense;          //  local-sense取反
    lock (counterlock) ;            //  确保更新的原子性
    count++;                        //  到达进程数加1
    unlock (counterlock) ;          //  释放锁
    if (count==total) {             //  进程全部到达
        count=0;                    //  重置计数器
        release=local_sense;        //  释放进程
        }
    else {                          //  还有进程未到达
        spin (release==local_sense) ; //  等待信号
        }
```

2. 当竞争不激烈且同步操作较少时，我们主要关心的是一个同步原语操作的延迟。
   - 即单个进程要花多长时间才完成一个同步操作。
   - 基本的旋转锁操作可在两个总线周期内完成：
     - 一个读锁
     - 一个写锁

   我们可用多种方法改进，使它在单个周期内完成操作。
3. 同步操作最严重的问题：进程进行同步操作的串行化。它大幅度地增加了完成同步操作所需要的时间。