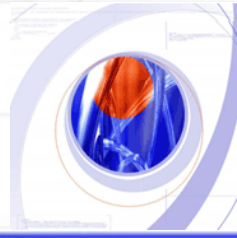
第8章 多处理器

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



8.3 同 步

同步机制通常是在硬件提供的**基本原语**基础上，通过**软件例程**来建立的。

8.3.1 基本硬件原语

在多处理机中实现同步，所需的主要功能是：

- 一组能以原子操作的方式读出并修改存储单元的硬件原语。它们都能以原子操作的方式读/修改存储单元，并指出所进行的操作是否以原子的方式进行。
- 通常情况下，用户不直接使用基本的硬件原语，原语主要供系统程序员用来编制同步库函数。

1. 典型操作：原子交换（atomic exchange）

- 功能：将一个存储单元的值和一个寄存器的值进行交换。

  建立一个锁，锁值：

  - 0：表示开的（可用）
  - 1：表示已上锁（不可用）

- 处理器上锁时，将对应于该锁的存储单元的值与存放在某个寄存器中的1进行交换。如果返回值为0，存储单元的值此时已置换为1，防止了别的进程竞争该锁。
- 实现同步的关键：操作的原子性

2. 测试并置定（test_and_set）
   - 先测试一个存储单元的值，如果符合条件则修改其值。
3. 读取并加1（fetch_and_increment）
   - 它返回存储单元的值并自动增加该值。
4. 使用指令对
   - LL(load linked或load locked)的取指令
   - SC(store conditional)的特殊存指令

- 指令顺序执行：
  - 如果由LL指明的存储单元的内容在SC对其进行写之前已被其它指令改写过，则第二条指令SC执行失败；
  - 如果在两条指令间进行切换也会导致SC执行失败。
  - LL返回该存储单元初始值。
  - SC将返回一个值来指出该指令操作是否成功：
    - “1”：成功
    - “0”：不成功

例：实现对由R1指出的存储单元进行原子交换操作。

```
try: OR    R3, R4, R0      // R4中为交换值。把该值送入R3
     LL    R2, 0（R1）
                           // 把单元0（R1）中的值取到R2
     SC    R3, 0（R1）     // 若0（R1）中的值与R3中的值相
                              同，则置R3的值为1，否则置为0
     BEQZ  R3, try         // 存失败（R3的值为0）则转移
     MOV   R4, R2          // 将取的值送往R4
```

最终R4和由R1指向的单元值进行原子交换，在LL和SC之间如有别的处理器插入并修改了存储单元的值，SC将返回0并存入R3中，从而使这段程序再次执行。

- LL / SC原语的另一个优点：读写操作明显分开

  LL命中不产生总线数据传输，这使下面代码与使用经过优化交换的代码具有相同的特点：

```
lockit:   LL      R2, 0(R1)
          BNEZ    R2, lockit
          DADDIU  R2, R0, #1
          SC      R2, 0(R1)
          BEQZ    R2, lockit
```

第一个分支形成环绕的循环体，第二个分支解决了两个处理器同时看到锁可用的情况下的争用问题。尽管旋转锁机制简单并且具有吸引力，但难以将它应用于处理器数量很多的情况。

8.3.2 用一致性实现锁

- 采用多处理机的一致性机制来实现旋转锁。
- 旋转锁

处理器环绕一个锁不停地旋转而请求获得该锁。适合于这样的场合：锁被占用的时间很少，在获得锁后加锁过程延迟很小。

1. 无Cache一致性机制

在存储器中保存锁变量，处理器可以不断地通过一个原子操作请求使用权。

比如：利用原子交换操作，并通过测试返回值而知道锁的使用情况。释放锁的时候，处理器只需简单地将锁置为0。

例：用原子交换操作对旋转锁进行加锁，R1中存放的是该旋转锁的地址。

```
        DADDIU    R2, R0, #1
lockit: EXCH      R2, 0(R1)
        BNEZ      R2, lockit
```

2. 支持Cache一致性

- 将锁调入Cache，并通过一致性机制使锁值保持一致。
- 优点：
  - 可使“环绕”的进程只对本地Cache中的锁（副本）进行操作，而不用在每次请求占用锁时都进行一次全局的存储器访问；
  - 可利用访问锁时所具有的局部性，即处理器最近使用过的锁不久又会使用。

（减少为获得锁而花费的时间）

➢ 改进旋转锁（获得第一条好处）

- 只对本地Cache中锁的副本进行读取和检测，直到发现该锁已经被释放。然后，该程序立即进行交换操作，去跟在其它处理器上的进程争用该锁变量。
- 修改后的旋转锁程序：

```
lockit:  LD      R2, 0(R1)
         BNEZ    R2, lockit
         DADDIU  R2, R0, #1
         EXCH    R2, 0(R1)
         BNEZ    R2, lockit
```

- 3个处理器利用原子交换争用旋转锁所进行的操作

3个处理器利用原子交换争用旋转锁所进行的操作

步骤	处理器P0	处理器P1	处理器P2	锁的状态	总线/目录操作
1	占有锁	环绕测试 是否lock=0	环绕测试 是否lock=0	共享	无
2	将锁置为0	（收到作废命令）	（收到作废命令）	专有（P0）	P0发出对锁变量的作废消息
3		Cache不命中	Cache不命中	共享	总线/目录收到P2 Cache不命中；锁从P0写回
4		（因总线/目录忙而等待）	lock=0	共享	P2 Cache不命中被处理
5		Lock=0	执行交换， 导致Cache不命中	共享	P1 Cache不命中被处理
6		执行交换， 导致Cache不命中	交换完毕：返回0 并置lock=1	专有（P2）	总线/目录收到P2 Cache不命中；发作废消息
7		交换完毕： 返回1	进入关键程序段	专有（P1）	总线/目录处理P1 Cache不命中；写回
8		环绕测试 是否lock=0			无

- LL / SC原语的另一个优点：读写操作明显分开

  LL命中不产生总线数据传输，这使下面代码与使用经过优化交换的代码具有相同的特点：

```
lockit:   LL      R2, 0(R1)
          BNEZ    R2, lockit
          DADDIU  R2, R0, #1
          SC      R2, 0(R1)
          BEQZ    R2, lockit
```

第一个分支形成环绕的循环体，第二个分支解决了两个处理器同时看到锁可用的情况下的争用问题。尽管旋转锁机制简单并且具有吸引力，但难以将它应用于处理器数量很多的情况。