第 6 章

ARM 处理器存储访问一致性问题

6.1 存储访问一致性问题介绍

当存储系统中引入了 cache 和写缓冲区（Write Buffer）时，同一地址单元的数据可能在系统中有多个副本，分别保存在 cache、Write Buffer 及主存中，如果系统采用了独立的数据 cache 和指令 cache，同一地址单元的数据还可能在数据 cache 和指令 cache 中有不同的版本。位于不同物理位置的同一地址单元的数据可能会不同，使得数据读操作可能得到的不是系统中“最新的数值”，这样就带来了存储系统中数据的一致性问题。在 ARM 存储系统中，数据不一致的问题则需要通过程序设计时遵守一定的规则来保证，这些规则说明如下。

6.1.1 地址映射关系变化造成的数据不一致性

当系统中使用了 MMU 时，就建立了虚拟地址到物理地址的映射关系，如果查询 cache 时进行的相连比较使用的是虚拟地址，则当系统中虚拟地址到物理地址的映射关系发生变化时，可能造成 cache 中的数据和主存中数据不一致的情况。

在虚拟地址到物理地址的映射关系发生变化前，如果虚拟地址 A1 所在的数据块已经预取到 cache 中，当虚拟地址到物理地址的映射关系发生变化后，如果虚拟地址 A1 对应的物理地

址发生了改变，则当 CPU 访问 A1 时再使用 cache 中的数据块将得到错误的结果。

同样当系统中采用了 Write Buffer 时，如果 CPU 写入 Write Buffer 的地址是虚拟地址，也会发生数据不一致的情况。在虚拟地址到物理地址的映射关系发生变化前，如果 CPU 向虚拟地址为 A1 的单元执行写操作，该写操作已经将 A1 以及对应的数据写入到 Write Buffer 中，当虚拟地址到物理地址的映射关系发生变化后，如果虚拟地址 A1 对应的物理地址发生了改变，当 Write Buffer 将上面被延迟的写操作写到主存中时，使用的是变化后的物理地址，从而使写操作失败。

为了避免发生这种数据不一致的情况，在系统中虚拟地址到物理地址的映射关系发生变化前，根据系统的具体情况，执行下面操作序列中的一种或几种：

1）如果数据 cache 为 write back 类型，清空该数据的 cache；

2）使数据 cache 中相应的块无效；

3）使指令 cache 中相应的块无效；

4）将 Write Buffer 中被延迟的写操作全部执行；

5）有些情况可能还要求相关的存储区域被设置成非缓冲的。

6.1.2 指令 cache 的数据不一致性问题

当系统中采用独立的数据 cache 和指令 cache 时，下面的操作序列可能造成指令不一致的情况：

1）读取地址为 A1 的指令，从而包含该指令的数据块被预取到指令 cache 中。

2）与 A1 在同一个数据块中的地址为 A2 的存储单元的数据被修改，这个数据写操作可能影响数据 cache、Write Buffer 和主存中地址为 A2 的存储单元的内容，但是不影响指令 cache 中地址为 A2 的存储单元的内容。

3）如果地址 A2 存放的是指令，当该指令执行时，就可能发生指令不一致的问题。如果地址 A2 所在的块还在指令 cache 中，系统将执行修改前的指令。如果地址 A2 所在的块不在指令 cache 中，系统将执行修改后的指令。

为了避免这种指令不一致情况的发生，在上面第 1）步和第 2）步之间插入下面的操作序列：

1）对于使用统一的数据 cache 和指令 cache 的系统，不需要任何操作；

2）对于使用独立的数据 cache 和指令 cache 的系统，使指令 cache 的内容无效；

3）对于使用独立的数据 cache 和指令 cache 的系统，如果数据 cache 是 write back 类型的，

清空数据 cache。

当数据操作修改了指令时，最好执行上述操作序列，保证指令的一致性。下面是上述操作序列的一个典型应用场合。当可执行文件加载到主存中后，在程序跳转到入口点处开始执行之前，先执行上述的操作序列，以保证下面指令的是新加载的可执行代码，而不是指令中原来的旧代码。

6.1.3 DMA 造成的数据不一致问题

DMA 操作直接访问主存，而不会更新 cache 和 Write Buffer 中相应的内容，这样就可能造成数据的不一致。

如果 DMA 从主存中读取的数据已经包含在 cache 中，而且 cache 中对应的数据已经被更新，这样 DMA 读到的将不是系统中最新的数据。同样 DMA 写操作直接更新主存中的数据，如果该数据已经包含在 cache 中，则 cache 中的数据将会比主存中对应的数据“老”，也将造成数据的不一致。

为了避免这种数据不一致情况的发生，根据系统的具体情况，执行下面操作序列中的一种或几种：

1）将 DMA 访问的存储区域设置成非缓冲的，即 uncachable 及 unbufferable；

2）将 DMA 访问的存储区域所涉及数据 cache 中的块设置成无效，或者清空数据 cache；

3）清空 Write Buffer（执行 Write Buffer 中延迟的所有写操作）；

4）在 DMA 操作期间限制处理器访问 DMA 所访问的存储区域。

6.1.4 指令预取和自修改代码

在 ARM 中允许指令预取，在 CPU 执行当前指令的同时，可以从存储器中预取其后若干条指令，具体预取多少条指令，不同的 ARM 实现中有不同的数值。

当用户读取 PC 寄存器的值时，返回的是当前指令下面第 2 条指令的地址。比如当前执行的是第 *N* 条指令，当用户读取 PC 寄存器的值时，返回的是指令 *N*+2 的地址。对于 ARM 指令来说，读取 PC 寄存器的值时，返回当前指令地址值加 8 个字节；对于 Thumb 指令来说，读取 PC 寄存器的值时，返回当前指令地址值加 4 个字节。

6.2 Linux 中解决存储访问一致性问题的方法

在 Linux 中，是用 barrier()宏来解决以上存储访问一致性问题的，barrier()的定义如下所示：

```
#define barrier() __asm__ __volatile__("": : :"memory")
```

另外在 barrier()的基础上还衍生出了很多类似的定义，如：

```
#define mb() __asm__ __volatile__ ("" : : : "memory")
#define rmb() mb()
#define wmb() mb()
#define smp_mb()        barrier()
#define smp_rmb()       barrier()
#define smp_wmb()       barrier()
```

barrier 是内存屏障的意思，CPU 越过内存屏障后，将刷新自己对存储器的缓冲状态。barrier()宏定义这条语句实际上不生成任何代码，但可使 gcc 在 barrier()之后刷新寄存器对变量的分配。具体分析如下。

指　令	说　明
__asm__	告诉编译器下面为汇编语句
__volatile__	告诉编译器该对象可能在程序之外被修改，严禁将此处的汇编语句与其他的语句重组合优化，即按原来的样子处理这这里的汇编。在 C 语言中通过使用关键字“volatile”声明存储器映射的 I/O 空间来防止编译器在优化时删掉有用的存储访问操作
Memory	告诉 gcc 编译器所有内存单元均被汇编指令修改过，registers 和 cache 中已缓存的内存单元中的数据将作废，barrier 之后的程序 CPU 必须重新从内存中读取数据而不是使用过期的 registers 和 cache 中的数据
""::::	表示这是个空指令。barrier()不用在此插入一条串行化汇编指令，所谓串行化指令也就是同步指令，即将 cache、Write Buffer 和内存中的数据同步更新

概括起来说 barrier()起到两个作用：

1）告诉编译器不要优化这部分代码，保持原有的指令执行顺序；

2）告诉 CPU 执行完 barrier()之后要进行同步操作，更新 registers、cache、写缓存和内存中的内容，全部重新从内存中取数据。