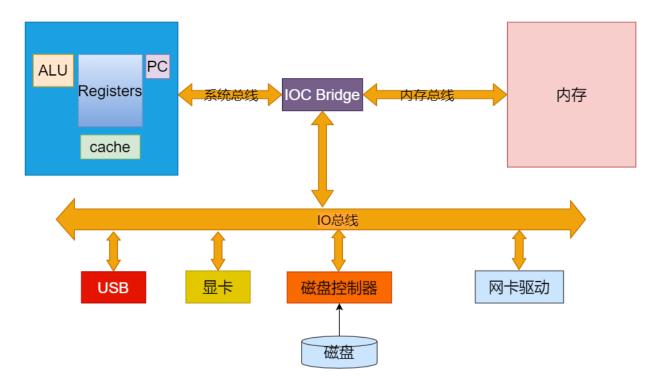
1、计算机组成结构



- 当使用一个程序,程序从硬盘中加载到内存当中,然后CPU将程序中的下一条指令地址读取到PC中,然后将相关数据存储到Registers(寄存器)中
- PC (Program Counter 程序计数器)

记录这一个地址存放下一条执行的指令在哪里,cpu执行完一条就去内存取下一条。

• Register寄存器

执行指令中的数据放到CPU执行,存储数据的就是寄存器

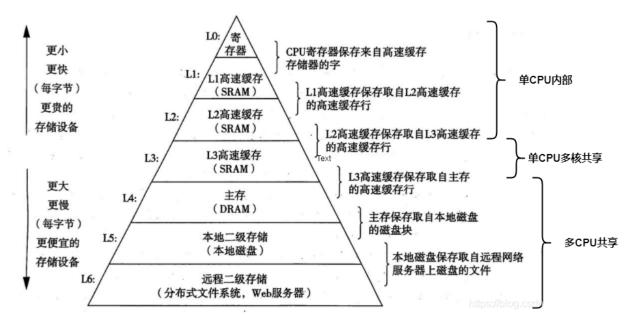
• ALU (Arithmetic Logic Unit) 运算单元

数据放到寄存器之后使用运算单元alu来运算,运算完写回到寄存器,寄存器再写回到内存里面 去。

计算机从磁盘读取到CPU中的过程很慢,所以我们的计算机中有好几层存储结构,会将磁盘中的数据进行缓存,而不是每次都重新读取

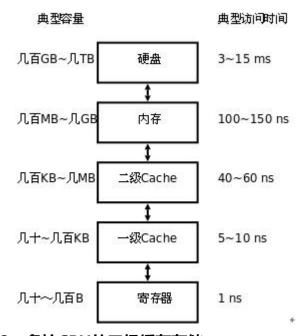
2、存储器的层次解耦股

2.1、层次缓存结构



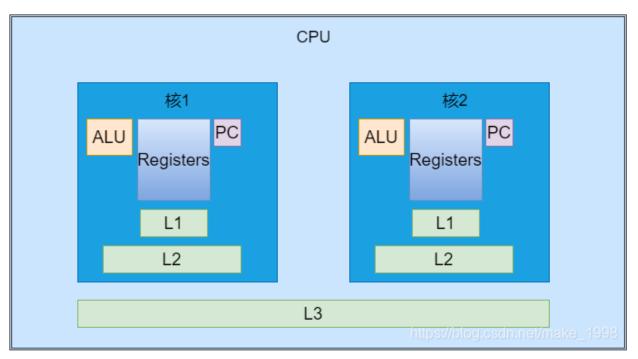
这几层的存储的访问和存储容量也是不同的

2.2、缓存容量和访问时间



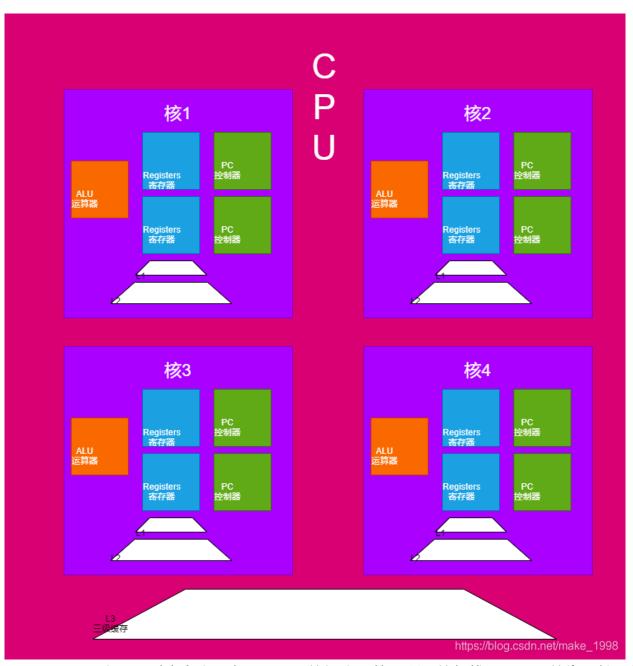
3、多核CPU的三级缓存存储

那咱们现在尝尝说的多核的CPU结构又是什么样的呢?



- 每个核中的一级缓存和二级缓存是每个核中单独拥有的,但是三级缓存是共同拥有的
- 对于一块CPU,可以有多个处理核心。每个核心内有自己的L1,L2缓存,多个核心共用同一个L3缓存。
- 但一个电脑如果有多个CPU插槽,各个CPU有自己的L3。
- 对于一个CPU核心来说,每个核心都有ALU,逻辑运算单元。负责对指令进行计算。Register 寄存器,记录线程执行对应的数据。PC:指令寄存器,记录线程执行到了哪个位置。里面存的是指令行数。通俗讲,就是记录线程执行到了哪一行指令(代码在进入CPU运行前,会被编译成指令)了。

5、超线程



- 一个ALU对应多个PC|Registers的组合,就是所谓的超线程。图示的为四核八线程
- 线程在执行的时候,将当前线程对应的数据放入寄存器,将执行行数放到指令寄存器,然后执行过一个时间片后,如果线程没有执行完,将数据和指令保存,然后其他线程进入执行。
- 一般来说,同一个CPU核在同一个时间点,只能执行同一个线程,但是,如果一个核里面有两组寄存器,两个pc。那么就可以同时执行两组线程,在切换线程的时候,没必要再去等待寄存器的数据保存和数据载入。直接切换到下一组寄存器就可以。这就是超线程。

5、缓存行 (cache line)

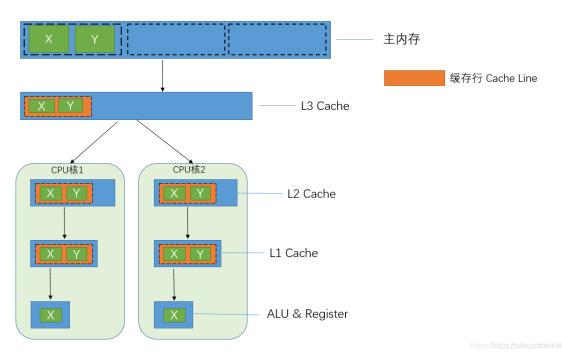
5.1、什么是缓存行

5.1.1、概念

- Cache 中的数据是按块读取的,当CPU访问某个数据时,会假设该数据附近的数据以后会被访问到,因此,第一次访问这一块区域时,会将该数据连同附近区域的数据(共64字节)一起读取进缓存中,那么这一块数据称为一个Cache Line 缓存行。
- 缓存系统是以缓存行为单位存储的。目前主流的CPU Cache的Cache Line大小都是64字节。

注: 并不是所有数据都会被缓存,比如一些较大的数据,缓存行无法容下,那么就只能每次都去主内存中读取。

5.1.2、例子



CPU到内存之间有很多层的内存,如图所示,CPU需要经过L1,L2,L3及主内存才能读到数据。从主内存读取数据时的过程如下

读取数据

• 当我左侧的CPU读取x的值的时候,按照下面L1——>L2——>L3,只有下一级 没有才会去上一级查找。最后从主内存读入的时候,首先将内存数据读入L3,然后L2 最后L1,然后再进行运算。

读取的块就叫做缓存行, cache line 。

读取的块就叫做缓存行, cache line 。

• 缓存行越大,局部性空间效率越高,但读取时间慢。

缓存行越小,局部性空间效率越低,但读取时间快。

目前多取一个平衡的值,64字节。

缓存行的效率问题

如果你的X和y在同一块缓存行中,且两个字段都用volatile修饰了

那么将来两个线程再修改的时候,就需要将x和y发生修改的消息告诉另外一个线程,让它重新加载对应缓存,然而另外一个线程并没有使用该缓存行中对应的内容,只是因为缓存行读取的时候跟变量相邻,也会需要重新读取。这就会产生效率问题。

5.2、伪共享

5.2.1、定义

伪共享的非标准定义为:

• 缓存系统中是以缓存行 (cache line) 为单位存储的, 当多线程修改互相独立的 变量时, 如果这些变量共享同一个缓存行, 就会无意中影响彼此的性能, 这就是伪共享。

5.2.2、 伪共享的含义

什么是伪共享???

如5.1.2的例子图所示:

- CPU1想要修改X, CPU2想要修改Y, 这两个频繁改动的变量是同意个缓存行, 两个争夺缓存行的拥有权。
- CPU1抢到后,更新X,那么CPU2上的缓存行的状态就会变成I状态(无效)
- ——状态含义(MESI协议)
- 当CPU2抢到,更新Y,CPU1上缓存行就会变成I状态(无效)

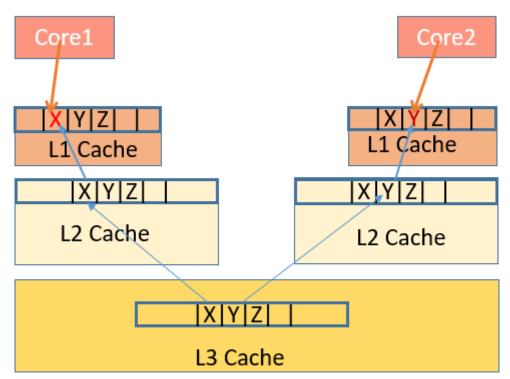
轮番抢夺,不仅会带来大量的RFO消息,而且某个线程读取此行数据时,L1和L2缓存上都是 失效数据,只有L3是同步好的。

表面上 X 和 Y 都是被独立线程操作的,而且两操作之间也没有任何关系。只不过它们共享了一个缓存行,但所有竞争冲突都是来源于共享。

因此, 当两个以上CPU都要访问同一个缓存行大小的内存区域时, 就会引起冲突, 这种情况就叫"伪共享"

当一个CPU高速缓存行发生改变,为了不影响其他CPU的相关数据的正确性,必须通过某些手段让其他拥有该缓存行数据的CPU高速缓存进行数据一致性同步,这就是造成"伪共享"的所在,试想运行在两个不同CPU核心上的两个线程,如果他们操作的内存数据在物理上处于相同或相邻的内存块区域(或者干脆就是操作的相同的数据),那么在将它们各自操作的数据加载到各自的一级缓存L1的时候,在很大概率上它们刚好位于一个缓存行(这是很有可能的,毕竟一个缓存行的大小一般有64个字节),即共享一个缓存行,这时候只要其中一个CPU核心更改了这个缓存行,都会导致另一个CPU核心的相应缓存行失效,如果它们确实操作的是同一个数据变量(即共享变量)这无可厚非,但如果它们操作的是不同的数据变量呢,依然会因为共享同一个缓存行导致整个缓存行失效,不得不重新进行缓存一致性同步,出现了类似串行化的运行结果,严重影响性能,这就是所谓的"伪共享",即从逻辑层面上讲这两个处理器核心并没有共享内存,因为他们访问的是不同的内容(变量)。但是因为cache line缓存行,产生了事实上的"共享"。显然,由于cache line大小限制带来的这种"伪共享"是我们不想要的,会浪费系统资源。

缓存行上的写竞争是运行在SMP系统中并行线程实现可伸缩性最重要的限制因素。有 人将伪共享描述成无声的性能杀手,因为从代码中很难知道两个不同的变量是否会出现伪共 享。



两个不同的线程,操作同一CacheLine中的不同字节,产生了不是真正共享的伪共享。

在上图中,展示了伪共享的种表现形式,数据X、Y、Z被加载到同一Cache Line中,线程A在Core1修改X,线程B在Core2上修改Y。根据MESI缓存一致性协议,假设是Core1是第一个发起写操作的CPU核,Core1上的L1 Cache Line由S(共享)状态变成M(修改,脏数据)状态,然后告知其他的CPU核,图例则是Core2,引用同一地址的Cache Line已经无效了;当Core2发起写操作时,首先导致Core1将X写回主存,而后才是Core2从主存重新读取该地址内容以便后面的修改。

可见多个线程操作在同一Cache Line上的不同数据,相互竞争同一Cache Line,导致线程彼此牵制影响,变成了串行程序,降低了并发性。此时我们则需要将共享在多线程间的数据进行隔离,使他们不在同一个Cache Line上,从而提升多线程的性能。当然上图只是数据在同一个CPU的多个核心直接产生的伪共享,数据仅仅通过共享的三级缓存L3就能得到同步,如果是多CPU或者处理器核心位于不同的插槽上,带来的性能问题才更糟。

为什么叫做伪共享。

但是,这种情况里面又包含了"其实不是共享"的"伪共享"情况。

比如,两个处理器各要访问一个word,这两个word却存在于同一个cache line大小的区域里,这时:

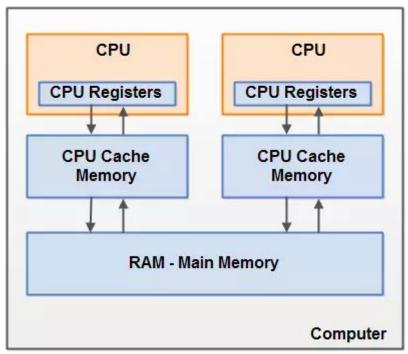
- 从应用逻辑层面说,这两个处理器并没有共享内存,因为他们访问的是不同的内容(不同的word)。
- 但是因为cache line的存在和限制,这两个CPU要访问这两个不同的word时,却一定要访问同一个cache line块,产生了事实上的"共享"。 I
- 显然,由于cache line大小限制带来的这种"伪共享"是我们不想要的,会浪费系统资源。

5.2.3、解决方案

- 1) 让不同线程操作的对象处于不同的缓存行。
- 2) 使用编译指示,强制使每一个变量对齐。

6、缓存一致性问题

CPU中的缓存和主存之间的结构



https://blog.csdn.net/make_1998

缓存一致性问题

在多处理器系统中,每个处理器都有自己的高速缓存,而它们又共享同一主内存(MainMemory)。

基于高速缓存的存储交互很好地解决了处理器与内存的速度矛盾,但是也引入了新的问题:缓存一致性(CacheCoherence)。

当多个处理器的运算任务都涉及同一块主内存区域时,将可能导致各自的缓存数据不一致的情况,如果真的发生这种情况,那同步回到主内存时以谁的缓存数据为准呢?



图 12-1 处理器、高速缓存、主内存间的交互关系 net/make 1998

为了解决一致性的问题,需要各个处理器访问缓存时都遵循一些协议,在读写时要根据协议来进行操作,这类协议有MSI、MESI(IllinoisProtocol)、MOSI、Synapse、Firefly及DragonProtocol,等等