第3章 コヒーレンス・プロトコルとプログラムを書くために用いる不可分命令

CPUごとに持つキャッシュの整合性を保つ仕組み

塩谷 亮太

複数のCPUコアからなるマルチコア・プロセッサが当たり前のものとなってから久しく，現代では組み込み機器においてもマルチコアがよく搭載されています．このマルチコアにおいて性能を引き出すためには処理を複数のスレッドに分割し協調して動作させる並列プログラミングが欠かせません．しかし一般に，高速かつ正しく動くよう並列プログラミングを行うのはとても難しいです．

この難易度を上げている要因の1つとして，ハードウェアの（一見）不思議な挙動があります．

キャッシュにまつわるハードウェアの不思議な挙動

● 2つの変数をインクリメントするコード

例えば**リスト1**のようなコードについて，関数taとtbをそれぞれ別のスレッドで並列に実行した場合を考えます．それぞれのスレッドにおいて，taはdata[0]をインクリメントし続ける一方で，tbはdata[B]をインクリメントし続けます．

ここでBを0，1，32と変えた場合，実行時間はどのように変化するでしょうか？Bを0にした場合，taを実行しているスレッドとtbを実行しているスレッドは共にdata[0]を更新するため，並列な処理が妨げられるかもしれません．

一方，Bを1や32にした場合，taがdata[0]を更新している一方で，tbは独立したdata[1]やdata[32]を更新しているのですから，処理時間はBが0の場合の半分で済むかもしれません．

このコードを筆者の手元のマシン（AMD Ryzen 5950X, gcc 9.3 -O3, N=0x70000000）で実行した結果は，

・B＝0で30秒　・B＝1で30秒　B＝32で0.4秒

というものでした．

● なぜ実行時間に違いが生ずるのか

B＝32がB＝0より速いのは良いとして，なぜ2けた近くも実行時間に違いがあるのかでしょうか．また，なぜ独立な変数を操作しているはずのB＝1がB＝0と同じように遅いのか，これは一見不思議に見えるかもしれません．

リスト1　2つの変数をインクリメントするコード

```
#define B 0  // tbがアクセスする場所．0 or 1 or 32を設定する
volatile int data[128];
void ta() {
    for (int i = 0; i < N; i++)
        data[0]++;  // 配列の0番目をインクリメント
}
void tb() {
    for (int i = 0; i < N; i++)
        data[B]++;  // 配列のB番目をインクリメント
}
```

この実行時間の差は，本稿で説明するコヒーレント・キャッシュというハードウェアの仕組みに由来します．以降ではその構造や，その上でプログラムを行うのに欠かせない不可分（atomic）命令などについて説明します．そしてその後に，それらがプログラムの性能にどのような影響を与えるのかを説明します．

ハードウェアに普段なじみがない人でも，ざっとその意味をつかめるよう，キャッシュの基本の解説から始め，なるべく予備知識がなくても読めるように書いたつもりです．逆に，これらの基本的なことを既に知っている場合，部分ごとに適宜読み飛ばしていただくのが良いと思います．

キャッシュの基礎知識

以下ではまず，CPUの持つキャッシュ・メモリ（以降，キャッシュと省略）について簡単に説明します．

● メモリには複数の階層がある

データを記憶するメモリは，高速にアクセスでき大容量であることが理想的です．しかし一般に，メモリのアクセス速度と容量には相反する関係があり，高速性と大容量を同時に満たすことはできません．

このため，今日のCPUは一般に記憶階層と呼ばれる，低速で大容量なメモリと，その一部を保持する高速で小容量なキャッシュ（cache）を階層的に組み合わ