Memory Consistency
after Advi & Gharachorloo
Program A

int flag1, flag2;

void p1 () {
    flag1 = 1;
    if (!flag2) { /* critical section */ }
}

void p2 () {
    flag2 = 1;
    if (!flag1) { /* critical section */ }
}
Program B

int data, ready;

void p1 () {
    data = 2000;
    ready = 1;
}

void p2 () {
    while (!ready)
        use (data);
}
Program C

```c
int a, b;

void p1 () { a = 1; }

void p2 () {
    if (a == 1)
        b = 1;
}

void p3 () {
    if (b == 1)
        use (a);
}
```
Sequential Consistency

- *Sequential consistency*: The result of execution is as if all operations were executed in some sequential order, and the operations of each processor occurred in the order specified by the program. [Lamport]

- Boils down to two requirements:
  1. Maintaining *program order* on individual processors
  2. Ensuring *write atomicity*
S.C. thwarts hardware optimizations

• Write buffers

• Overlapping write operations
  - Coalescing writes to same cache line

• Non-blocking reads

• Cache coherence
  - Write completion only after invalidation/update
  - Can’t have overlapping updates (Program C)
S.C. thwarts compiler optimizations

• Code motion

• Caching value in register
  - E.g., ready flag in Program B

• Common subexpression elimination

• Loop blocking

• Software pipelining
Possible optimizations

• “Prefetch” writes
  - Invalidate memory in other CPU’s caches while waiting for previous reads to complete

• Speculatively execute reads (optimistically)
  - If program order violated, roll back state
Relaxed Consistency Models

- **Relax program order**
  - Relax Write to Read order
    - E.g., Re-order read wrt. writes from same proc, breaks A
  - Relax Write to Read and Write to Write order
    - E.g., Read own writes before other people
  - Relax Read to Read and Read to Write order

- **Relax write atomicity**
  - Read others’ writes early

- **Relax both**
  - Read own writes early (in conjunction with other relaxation)
Weak ordering

- Define two classes of memory operation
  - data
  - synchronization

- System can reorder any operations between sync references

- Easy to implement:
  - Processor keeps counter of outstanding operations
How to classify memory accesses?

- Find variables that race under S.C.:
  - Two operations access variable
  - At least one is a write
  - No intervening references (in S.C.)

- E.g., in Prog B, ready races, not data
Release consistency

- **4 types of memory operation:**
  - ordinary, nsync, acquire, release

- **Preserve the following orderings [RCsc]:**
  - acquire → all
  - all → release
  - \{release, nsync\} → \{acquire, nsync\}

- **Perfect for data protected by mutexes**