Memory Consistency

Writes to a location become visible to all in the same order

But when does a write become visible

- How to establish orders between a write and a read by different procs?
  - Typically use event synchronization, by using more than one location

P_1	P_2
/*Assume initial value of A and ag is 0*/	
`A = 1;`	`while (flag == 0);` /*spin idly*/
`flag = 1;`	`print A;`

- Intuition not guaranteed by coherence
- Sometimes expect memory to respect order between accesses to *different* locations issued by a given process
  - to preserve orders among accesses to same location by different processes
- Coherence doesn't help: pertains only to single location

Another Example of Orders

P_1	P_2
/*Assume initial values of A and B are 0*/	
`(1a) A = 1;`	`(2a) print B;`
`(1b) B = 2;`	`(2b) print A;`

- What's the intuition?
- Whatever it is, we need an ordering model for clear semantics
  - across different locations as well
  - so programmers can reason about what results are possible
- This is the memory consistency model

Memory Consistency Model

Specifies constraints on the order in which memory operations (from any process) can *appear to execute* with respect to one another

- What orders are preserved?
- Given a load, constrains the possible values returned by it

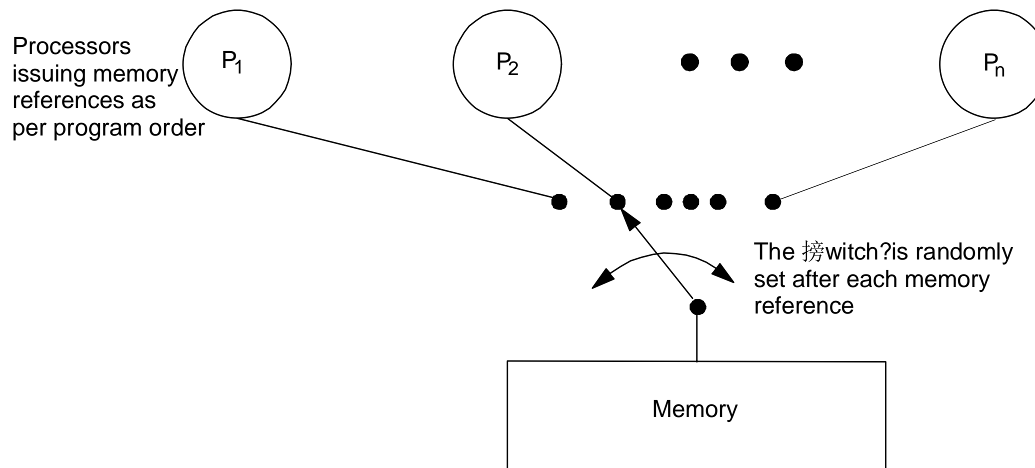
Without it, can't tell much about an SAS program's execution

Implications for both programmer and system designer

- Programmer uses to reason about correctness and possible results
- System designer can use to constrain how much accesses can be reordered by compiler or hardware

Contract between programmer and system

Sequential Consistency



- (as if there were no caches, and a single memory)
- Total order achieved by *interleaving* accesses from different processes
- Maintains *program order*, and memory operations, from all processes, appear to [issue, execute, complete] atomically w.r.t. others
- Programmer's intuition is maintained

"A multiprocessor is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program." [Lamport, 1979]

What Really is Program Order?

Intuitively, order in which operations appear in source code

- Straightforward translation of source code to assembly
- At most one memory operation per instruction

But not the same as order presented to hardware by compiler

So which is program order?

Depends on which layer, and who’s doing the reasoning

We assume order as seen by programmer

SC Example

What matters is order in which *appears to execute*, not *executes*

P_1	P_2

/*Assume initial values of A and B are 0*/

```
(1a) A = 1;                    (2a) print B;
(1b) B = 2;                    (2b) print A;
```

– possible outcomes for (A,B): (0,0), (1,0), (1,2); impossible under SC: (0,2)

– we know 1a->1b and 2a->2b by program order

– A = 0 implies 2b->1a, which implies 2a->1b

– B = 2 implies 1b->2a, which leads to a contradiction

– BUT, actual execution 1b->1a->2b->2a is SC, despite not program order

  - appears just like 1a->1b->2a->2b as visible from results

– actual execution 1b->2a->2b-> is not SC

Implementing SC

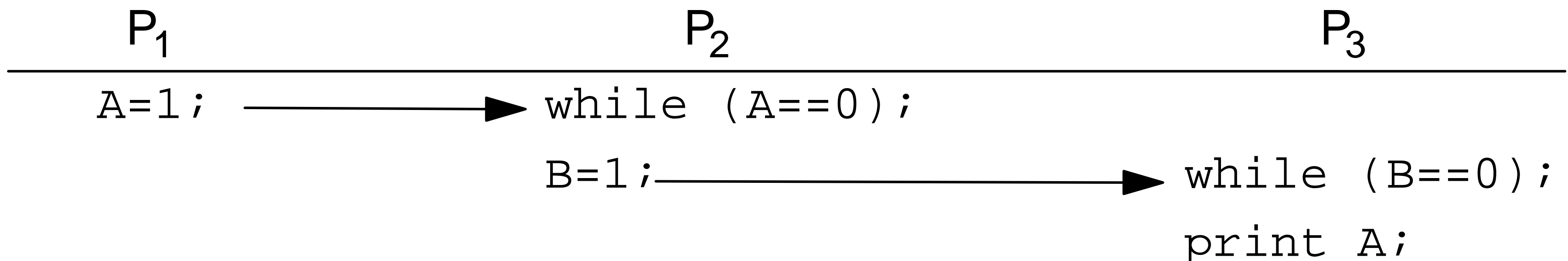
Two kinds of requirements

- Program order
  - memory operations issued by a process must appear to become visible (to others and itself) in program order
- Atomicity
  - in the overall total order, one memory operation should appear to complete with respect to all processes before the next one is issued
  - needed to guarantee that total order is consistent across processes
  - tricky part is making writes atomic

Write Atomicity

Write Atomicity: Position in total order at which a write appears to perform should be the same for all processes

- Nothing a process does after it has seen the new value produced by a write W should be visible to other processes until they too have seen W
- In effect, extends write serialization to writes from multiple processes



- Transitivity implies A should print as 1 under SC
- Problem if P_2 leaves loop, writes B, and P_3 sees new B but old A (from its cache, say)

More Formally

Each process's program order imposes partial order on set of all operations

Interleaving of these partial orders defines a total order on all operations

Many total orders may be SC (SC does not define particular interleaving)

SC Execution: An execution of a program is SC if the results it produces are the same as those produced by some possible total order (interleaving)

SC System: A system is SC if any possible execution on that system is an SC execution

Sufficient Conditions for SC

1. Every process issues memory operations in program order
2. After a write operation is issued, the issuing process waits for the write to complete before issuing its next operation
3. After a read operation is issued, the issuing process waits for the read to complete, and for the write whose value is being returned by the read to complete, before issuing its next operation (provides write atomicity)

Sufficient, not necessary, conditions

Clearly, compilers should not reorder for SC, but they do!

- Loop transformations, register allocation (eliminates!)

Even if issued in order, hardware may violate for better performance

- Write buffers, out of order execution

Reason: uniprocessors care only about dependences to same location

- Makes the sufficient conditions very restrictive for performance

Our Treatment of Ordering

Assume for now that compiler does not reorder

Hardware needs mechanisms to detect:

- Detect write completion (read completion is easy)
- Ensure write atomicity

For all protocols and implementations, we will see

- How they satisfy coherence, particularly write serialization
- How they satisfy sufficient conditions for SC (write completion and write atomicity)
- How they can ensure SC but not through sufficient conditions

Will see that centralized bus interconnect makes it easier

SC in Write-through Example

Provides SC, not just coherence

Extend arguments used for coherence

- Writes and read misses to *all locations* serialized by bus into bus order
- If read obtains value of write W, W guaranteed to have completed
  - since it caused a bus transaction
- When write W is performed w.r.t. any processor, all previous writes in bus order have completed