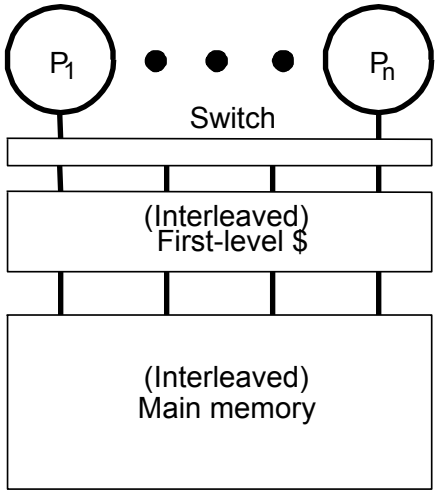
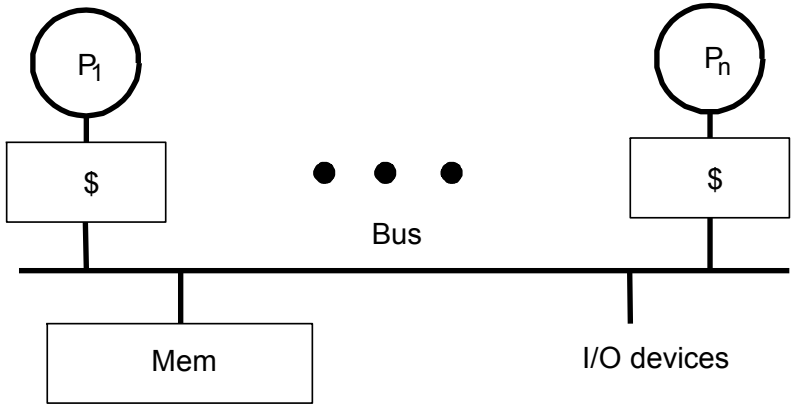
Shared Memory Multiprocessors

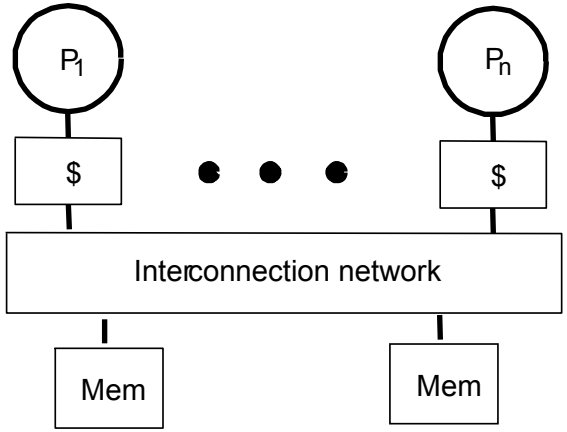
Natural Extensions of Memory System



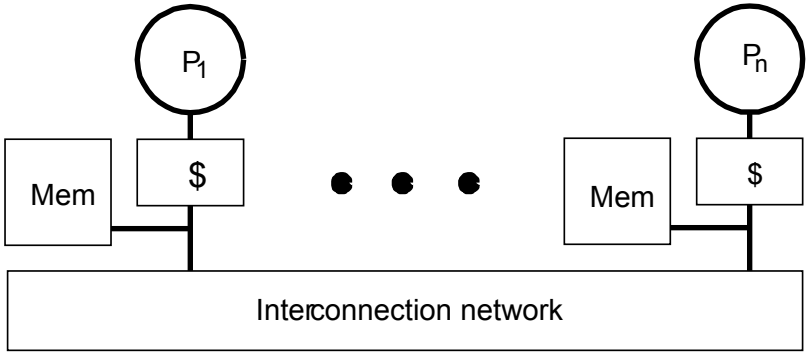
(a) Shared cache



(b) Bus-based shared memory



(c) Dancehall



(d) Distributed-memory

Caches and Cache Coherence

Caches play key role in all cases

- Reduce average data access time
- Reduce bandwidth demands placed on shared interconnect

But private processor caches create a problem

- Copies of a variable can be present in multiple caches
- A write by one processor may not become visible to others
  - They'll keep accessing stale value in their caches
- *Cache coherence* problem

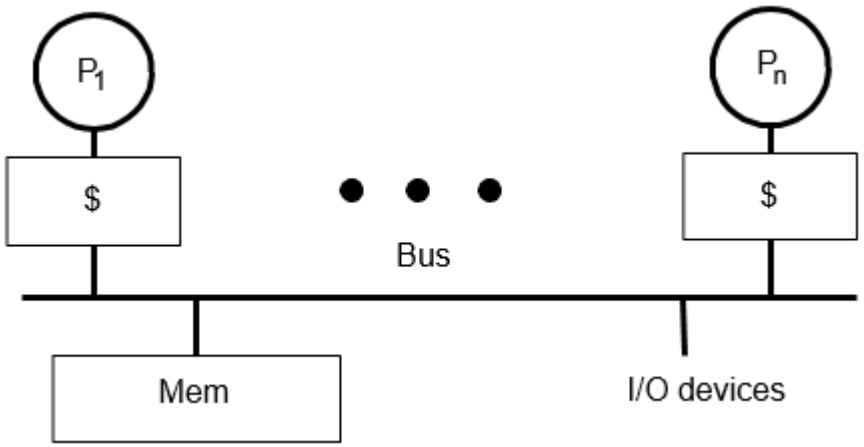
Need for Coherence

a = b =c = d = 0 initially

P1	**P2**
a = 1;	b = 1;
if (b == 0)	if (a == 0)
c = 1;	d = 1;

Need to take actions to ensure writes are visible everywhere!

First Focus: Bus-based SMPs



- Low-latency sharing and prefetching across processors
- Sharing of working sets
- But high bandwidth needs and negative interference (e.g. conflicts)
- Mid 80s: to connect couple of processors on a board (Encore, Sequent)
- Today: for multiprocessor on a chip (for small-scale systems or nodes)

A Coherent Memory System: Intuition

Reading a location should return latest value written (by any process)

Easy in uniprocessors

- Except for I/O: coherence between I/O devices and processors
- But infrequent so software solutions work
  - uncacheable memory, uncacheable operations, flush pages, pass I/O data through caches

Would like same to hold when processes run on different processors

- E.g. as if the processes were interleaved on a uniprocessor

But coherence problem much more critical in multiprocessors

- Pervasive
- Performance-critical
- Must be treated as a basic hardware design issue

Problems with the Intuition

Recall: Value returned by read should be last value written

But “last” is not well-defined

Even in seq. case, last defined in terms of program order, not time

- Order of operations in the machine language presented to processor
- “Subsequent” defined in analogous way, and well defined

In parallel case, program order defined within a process, but need to make sense of orders across processes

Must define a meaningful semantics

Some Basic Definitions

Extend from definitions in uniprocessors to those in multiprocessors

Memory operation: a single read (load), write (store) or read-modify-write access to a memory location

- Assumed to execute atomically w.r.t each other

Issue: a memory operation issues when it leaves processor's internal environment and is presented to memory system (cache, buffer …)

Perform: operation appears to have taken place, as far as processor can tell from other memory operations it issues

- A write performs w.r.t. the processor when a subsequent read by the processor returns the value of that write or a later write
- A read perform w.r.t the processor when subsequent writes issued by the processor cannot affect the value returned by the read

In multiprocessors, stay same but replace "the" by "a" processor

- Also, *complete*: perform with respect to all processors
- Still need to make sense of order in operations from different processes

Sharpening the Intuition

Imagine a single shared memory and no caches

- Every read and write to a location accesses the same physical location
- Operation completes when it does so

Memory imposes a *serial* or *total order* on operations to the location

- Operations to the location from a given processor are in program order
- The order of operations to the location from different processors is some interleaving that preserves the individual program orders

“Last” now means most recent in a hypothetical serial order that maintains these properties

For the serial order to be consistent, all processors must see writes to the location in the same order (if they bother to look, i.e. to read)

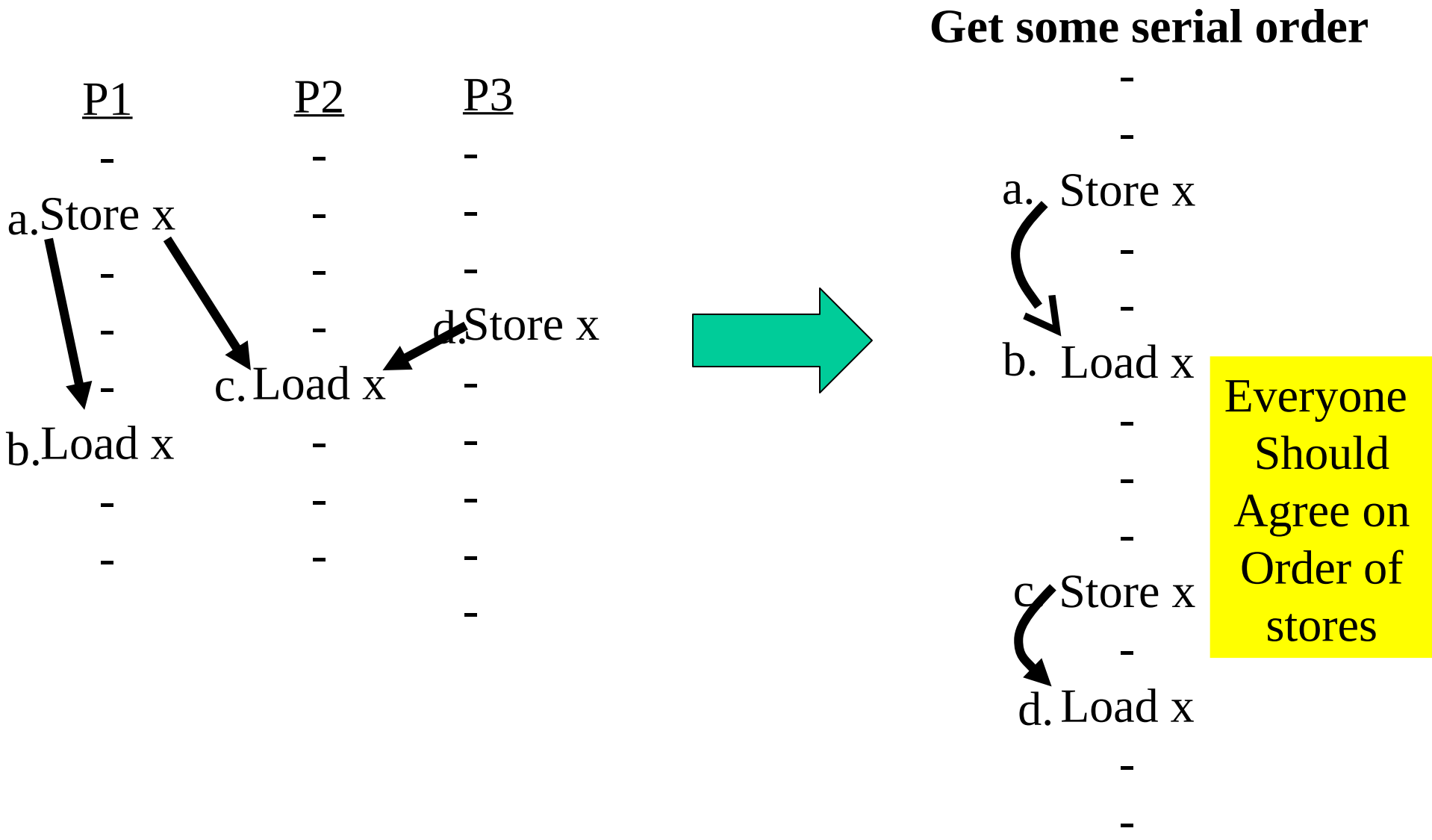
Note that the total order is never really constructed in real systems

- Don’t even want memory, or any hardware, to see all operations

But program should behave as if some serial order is enforced

- Order in which things appear to happen, not actually happen

What does “last” mean?



Formal Definition of Coherence

Results of a program: values returned by its read operations

A memory system is *coherent* if the results of any execution of a program are such that each location, it is possible to construct a hypothetical serial order of all operations to the location that is consistent with the results of the execution and in which:

1. operations issued by any particular process occur in the order issued by that process, and

2. the value returned by a read is the value written by the last write to that location in the serial order

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

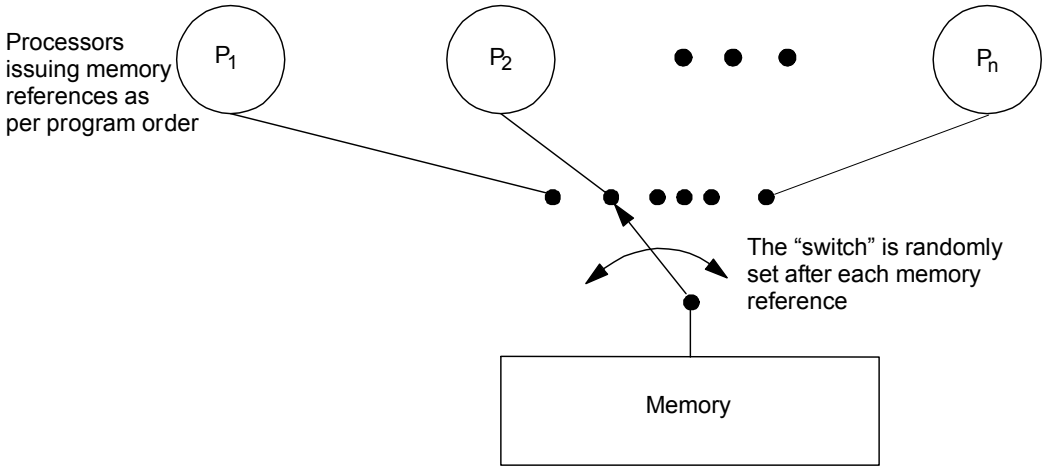
Without it, can't tell much about an SAS program's execution

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

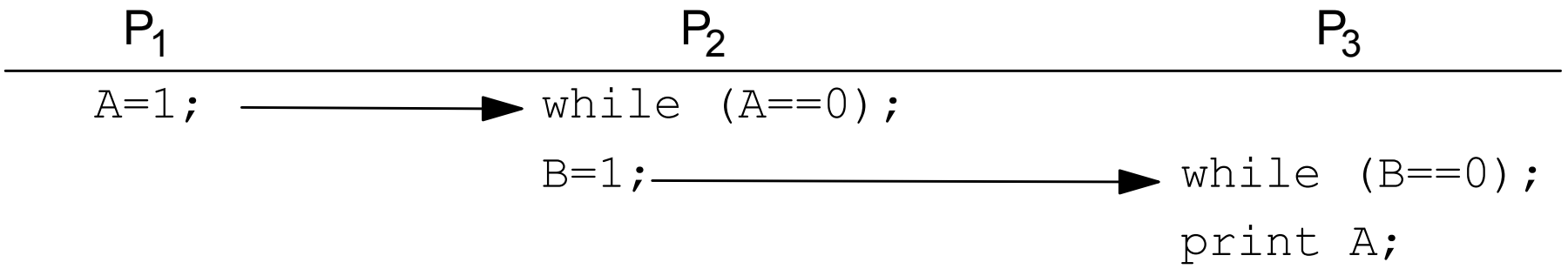
Two kinds of requirements

- 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

- Every process issues memory operations in program order
- After a write operation is issued, the issuing process waits for the write to complete before issuing its next operation
- 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

Two Required Features

Write propagation: value written must become visible to others

Write serialization: writes to location seen in same order by all

if I see w1 after w2, you should not see w2 before w1