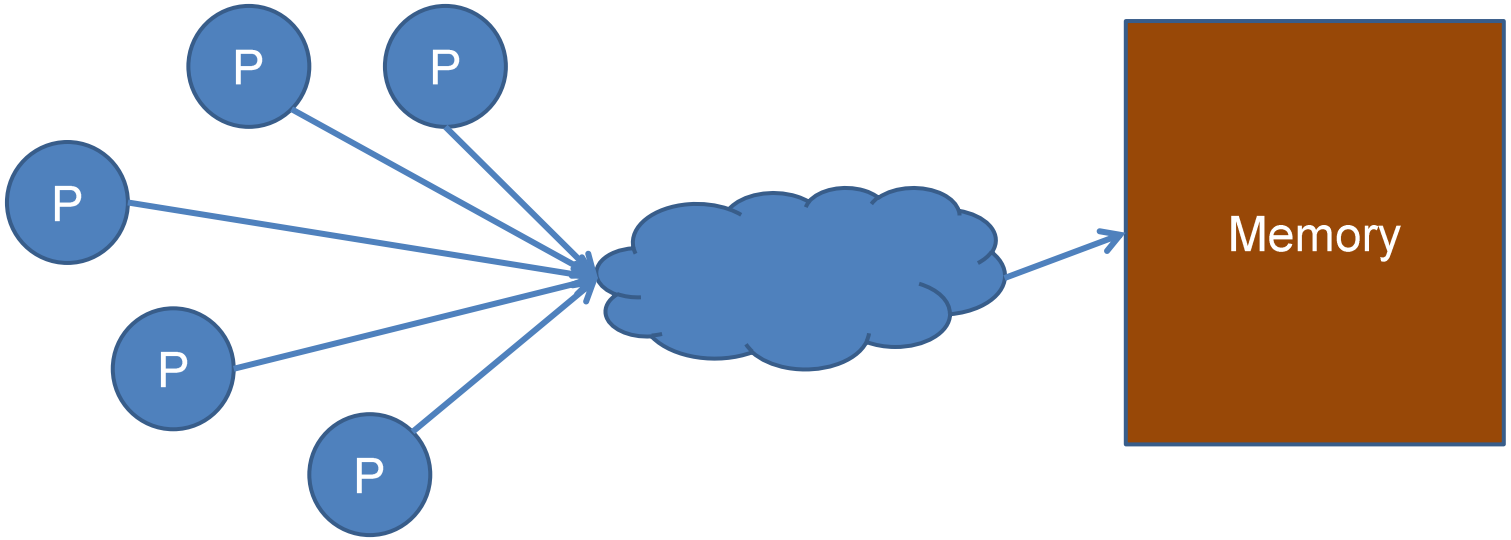
COMPUTER ARCHITECTURE

Memory consistency models

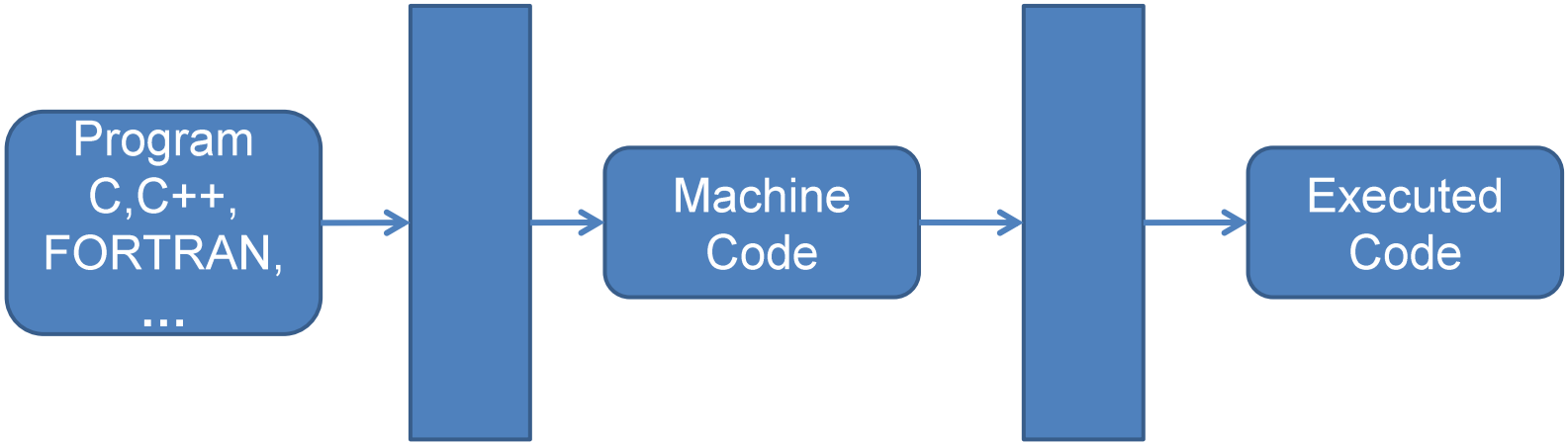
Memory consistency models



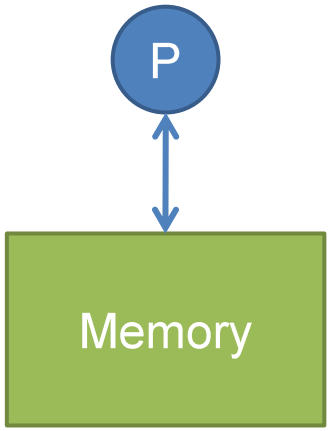
- A memory consistency model:
  - Set of rules defining how the memory system processes memory operations from multiple processors.
  - Contract between programmer and system.
  - Determines which optimizations ara valid on correct programs.

Memory model

- Interface between program and its transformations.
  - Defines which values can be returned by a read.
- Language memory model has implications on the hardware.



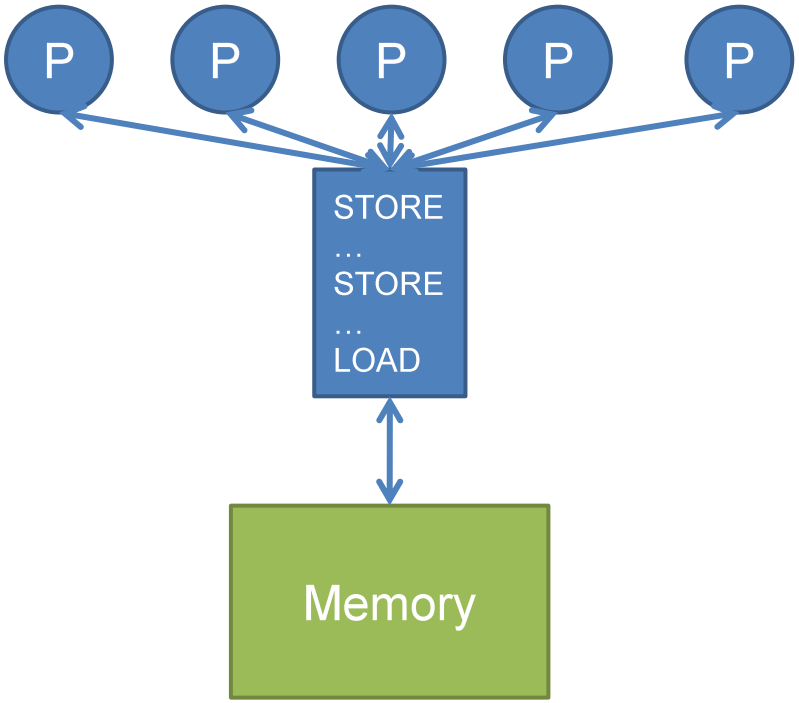
Single processor memory model



STORE
…
LOAD
…
STORE
…
LOAD

- Memory behavior model:
  - Memory operations happen ins program order.
    - A read returns the value from last write in program order.
- Semantics defined by sequential program order.
  - Simple reasoning but constrained.
    - Solve data and control dependencies.
  - Independent operations may be executed in parallel.
  - Optimizations preserve semantics.

Sequential consistency



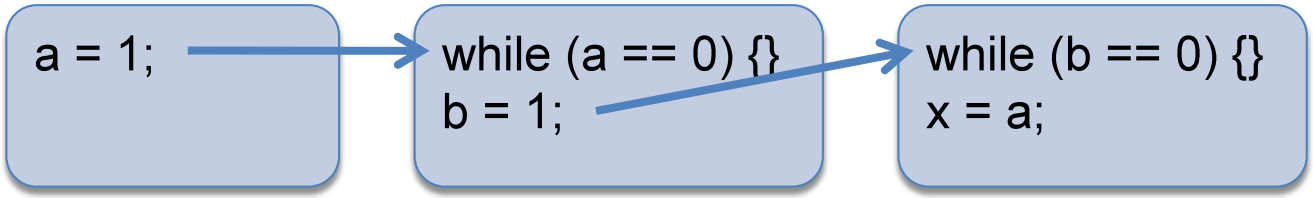
- *A multiprocessor system is sequentially consistent if the result of any execution is the same that would be obtained if operations from all processors were executed in some sequential order, and operations from each individual processor appear in that sequence in the order established by the program.*
- **[Lamport 1979]**
  - Turing Prize 2014.

Sequential Consistency: Constraints

- Program order.
  - Memory operations from a program must be made visible to all processes in program order.

- Atomicity.
  - Total execution order between processes must be consistent requiring all operations to be atomic.
    - Nothing that a processor does after it has seen the new value from a write is made visible to other processes before they have seen the value from that write.

Atomicity



- Non atomic writes:
  - Write on b could bypass the while loop and read from a could bypass the write.
    - X=0.
- Atomic writes:
  - Sequential consistency is preserved.

Sequential consistency

- SC constraints all memory operations:
  - Write → Read.
  - Write → Write.
  - Read → Read, Read → Write.
- Simple model to reason about parallel programs.
- But, simple reordering for single-processors may violate the sequential consistency model:
  - Hardware reordering to improve performance.
    - Write buffers, overlapped writes.
  - Compiler optimizations apply transformations with memory operations reordering. Scalar replacement, register allocation, instruction scheduling.
  - Transformations by programmers or refactoring tools also modify program semantics.

Sequential consistency violation: Write Buffers


```
flag1=0; flag2=0;
```

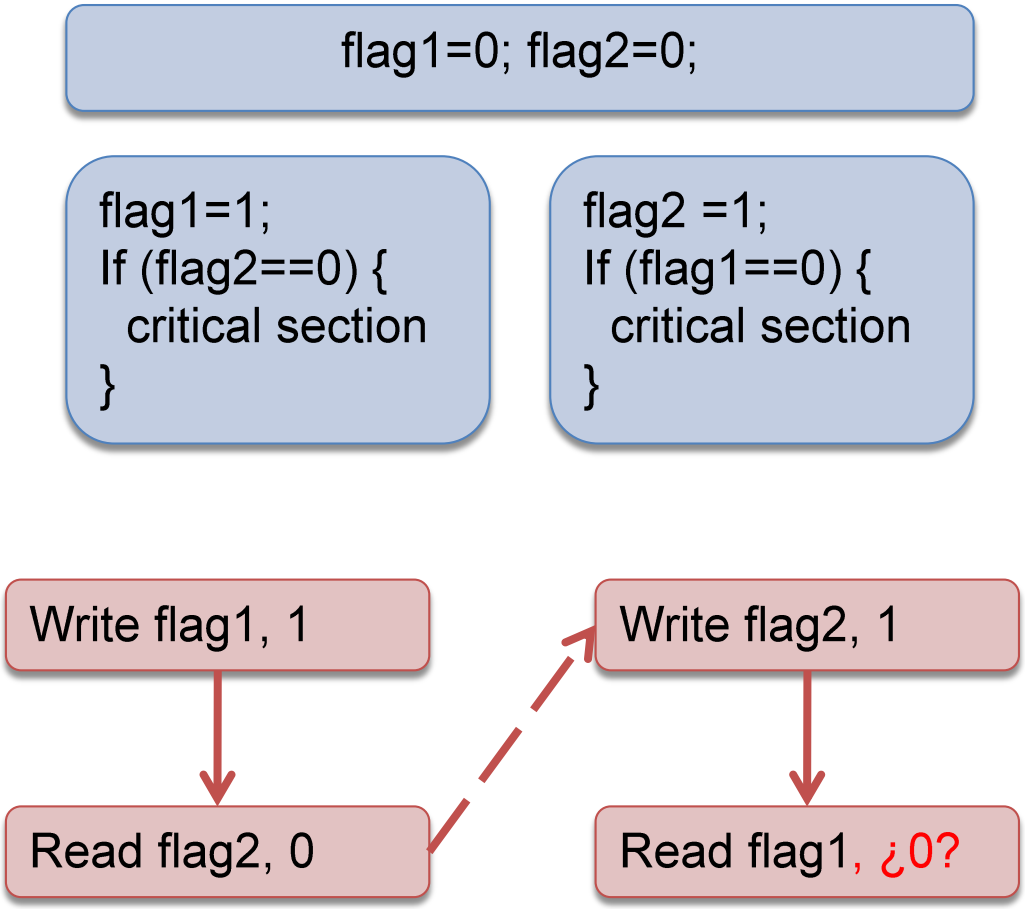
```
flag1=1;
If (flag2==0) {
  critical section
}
```

```
flag2 =1;
If (flag1==0) {
  critical section
}
```

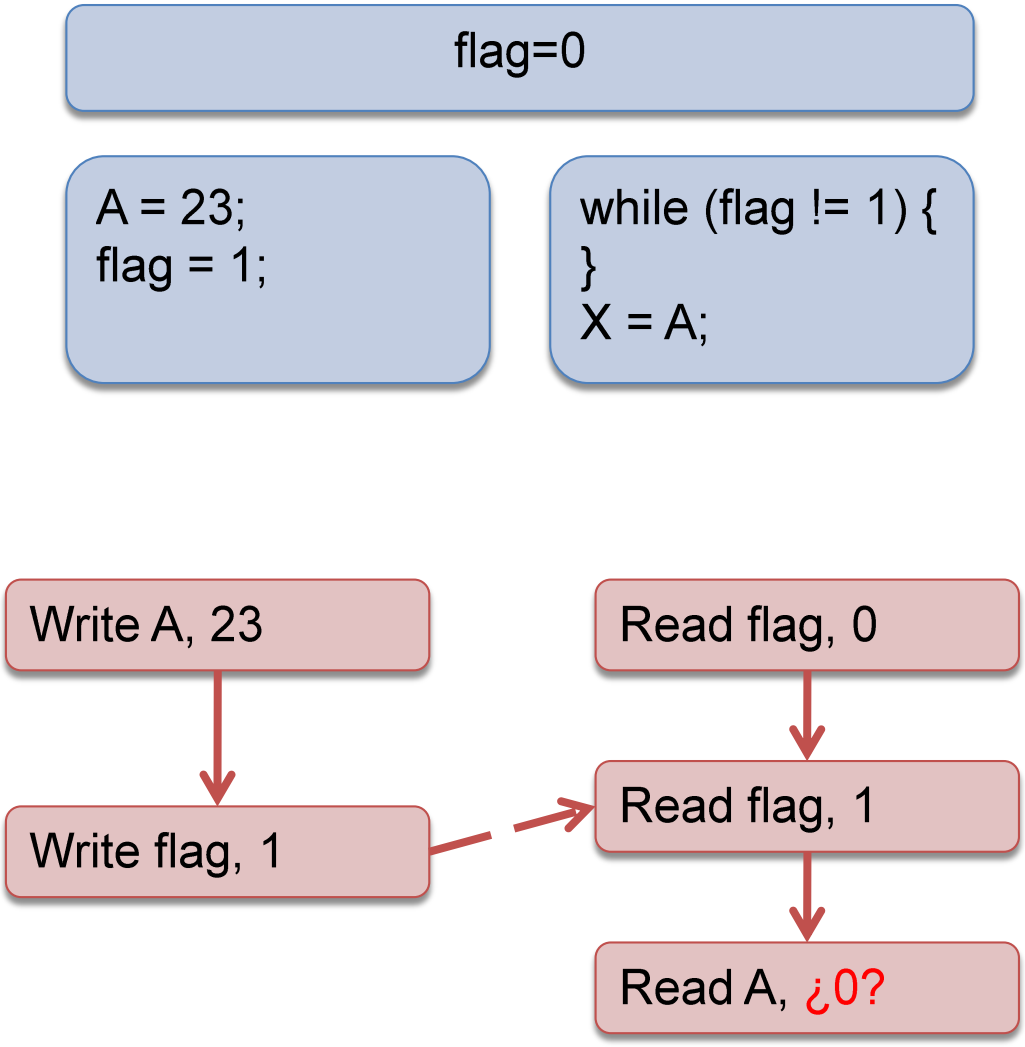
```
assert(flag1!=0 || flag2!=0)
```

- If caches use write buffer:
  - Writes are delayed in buffer.
  - Read obtain old value.
  - Dekker algorithm is invalidated.
    - Dekker algorithm is the first known solution to the critical section problem.

Program order



Program order



Sufficient conditions for sequential consistency

- Sufficient conditions:
  - Each process emits memory operations in program order.
  - After emission of a write, emitting process waits to complete the write before emitting other operation.
  - After a read emission, emitting process waits until read is completed and that the write of the value being read is completed.
    - Wait write propagation to all processes.
- These are very exigent conditions.
  - There might be necessary conditions that are less exigent.

Optimizations

- Models relaxing execution program order.
  - W → R
  - W → W
  - R → W, W → W

Reorderings

Processor	R → R	R → W	W → W	W → R
Alpha	X	X	X	X
PA-RISC	X	X	X	X
POWER	X	X	X	X
SPARC				X
X86				X
AMD64				X
IA64	X	X	X	X
zSeries				X

Reads bypass writes
W → R

- A read may execute before a preceding write.

- Typical in systems with write buffer.
  - Check consistency with buffer.
  - Allow read buffer.

Other models

- R → W, W → R
  - Allow that writes may arrive into memory out of program order.

- R → W, W → R, R → R, W → W
  - Avoid only data and control dependencies within processor.
  - Alternatives:
    - Weak consistency.
    - Release consistency.

Weak ordering

- Weak ordering:
  - Divide memory operations in data operations and synchronization operations.
  - Synchronization operations act as a barrier.
    - All preceding data operations in program order to a synchronization must complete before synchronization is executed.
    - All subsequent data operations in program order to a synchronization operation must wait until synchronization ins completed.
    - Synchronization are performed in program order.
  - Hardware implementation of barrier.
    - Processor keeps a counter.
      - Data operation emission → increment.
      - Data operation completion → decrement

Acquire/release consistency

- More *relaxed* than weak consistency.
- Synchronization accesses divided into:
  - Acquire.
  - Release.
- Semantics:
  - Acquire:
    - Must complete before all subsequent memory accesses.
  - Release
    - Must complete all previous memory accesses.
    - Subsequent memory accesses MAY initiate.
    - Operations following a *release* and must wait, must be protected with an acquire.

Memory consistency in Intel processors

- Until 2005 Intel had not completely clarified its memory consistency model.
  - Formalizing model highly complex.
  - Problems for language implementations (Java, C++, …)
- Currently the model is clarified and public.

Intel initial model

- i486 y Pentium:
  - Operations in program order.
    - Exception: Read misses bypass writes in write buffer only if all writes are cache hits.
    - It is impossible that a read miss matches with a write.

Atomic operations

- Since i486:
  - Read or write 1 byte.
  - Read or write a 16-bit aligned word.
  - Read or write a 32-bit aligned double word.
- Since Pentium:
  - Read or write a 64-bit aligned *quadword*.
  - Non-cached memory access that fits in 32 bit data bus.
- Since P6:
  - Non aligned access to data of 16, 32, or 64 bits that fit in a cache line.

Bus blocking

- A processor may emit a signal to block the bus.
  - Other elements cannot access the bus.
- Automatic bus blocking.
  - XCHG instruction.
  - Update segment descriptors, page directory, and page table.
  - Interruption acceptance.
- Bus software blocking:
  - Use LOCK prefix in:
    - Instructions for bit checking and modification (BTS, BTR, BTC).
    - Exchange instructions (XADD, CMPXCHG, CMPXCHG8B).
    - 1 operand arithmetic instructions (INC, DEC, NOT, NEG).
    - 2 operand arithmetic-logic instructions (ADD, ADC, SUB, SBB, AND, OR, XOR).

Barrier instructions

- **LFENCE**
  - Barrier for ***load*** operations.
  - Every ***load*** preceding a **LFENCE** is globally made visible before any subsequent load.
- SFENCE
  - Barrier for ***store*** operations.
  - Every ***store*** preceding **SFENCE** is globally made visible before any subsequent store.
- MFENCE
  - Barrier for ***load/store*** operations.
  - All the ***load*** and ***store*** preceding **MFENCE** are globally made visible before any other subsequent ***load*** or ***store*** operation.

Current memory model (in a single processor)

- Reads do not bypass other reads (R → R).
- Writes do not bypass reads (R → W).
- Writes do not bypass writes (W → W).
  - Exceptions for strings and non-temporal moves.
- Reads bypass preceding writes (W → R) to different addresses.
- Reads/writes do not bypas I/O operations, locked instructions, serializing instructions.
- Reads cannot bypass preceding LFENCE or MFENCE.
- Writes cannot bypass preceding LFENCE, SFENC or MFENCE.
- LFENCE cannot bypass a preceding read.
- SFENCE cannot bypass a preceding write.
- MFENCE cannot bypass a preceding read or write.

Multiprocessor memory model

- Each processor is individually compliant with former rules.
- Writes from a processor are observed in the same order by all the other processors.
- Writes of a processor are NOT ordered with respect to writes from other processors.
- Memory ordering is transitive.
- Two writes are viewed in a consistent order by any other processor distinct from those two.
- Lock instructions have a total order.

Example: Write ordering

Processor A	Processor B	Processor C
Write A.1 Write A.2 Write A.3	Write B.1 Write B.2 Write B.3	Write C.1 Write C.2 Write C.3

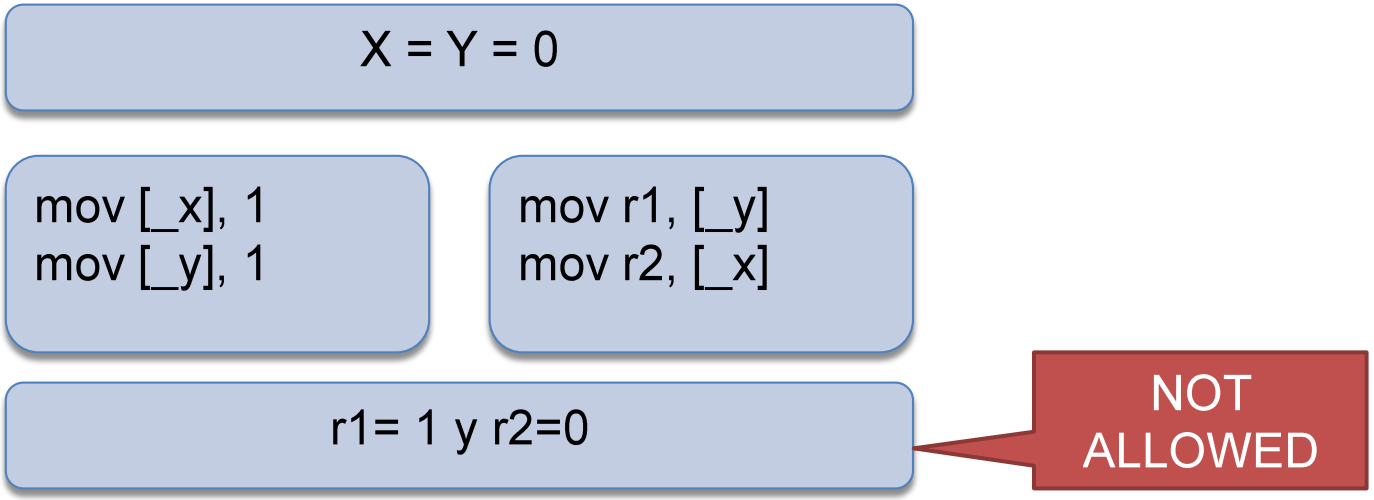
Writes from every processor keep order

Write A.1
Write B.1
Write B.2
Write C.1
Write A.2
Write B.3
Write A.3
Write C.2
Write C.3

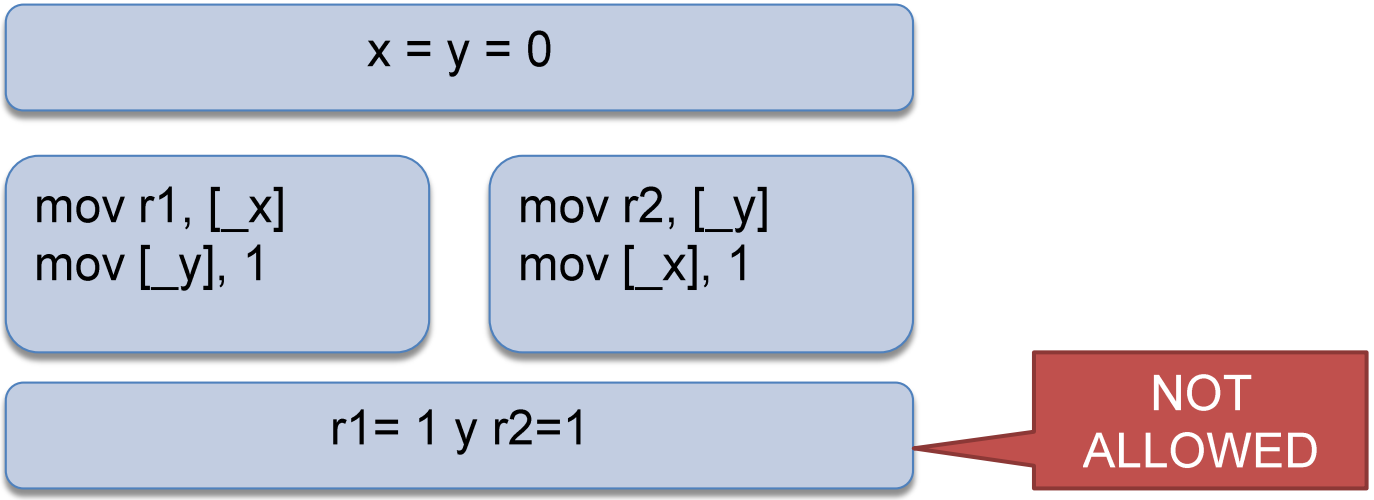
Order for every process is kept

No order is guaranteed across processes

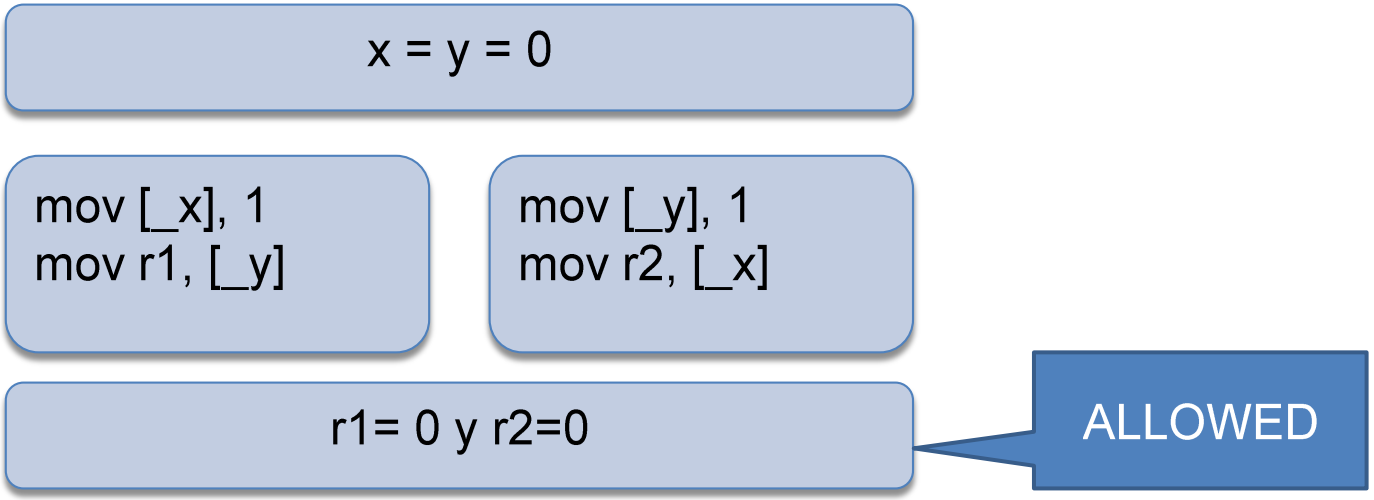
No reordering
R → R, W → W



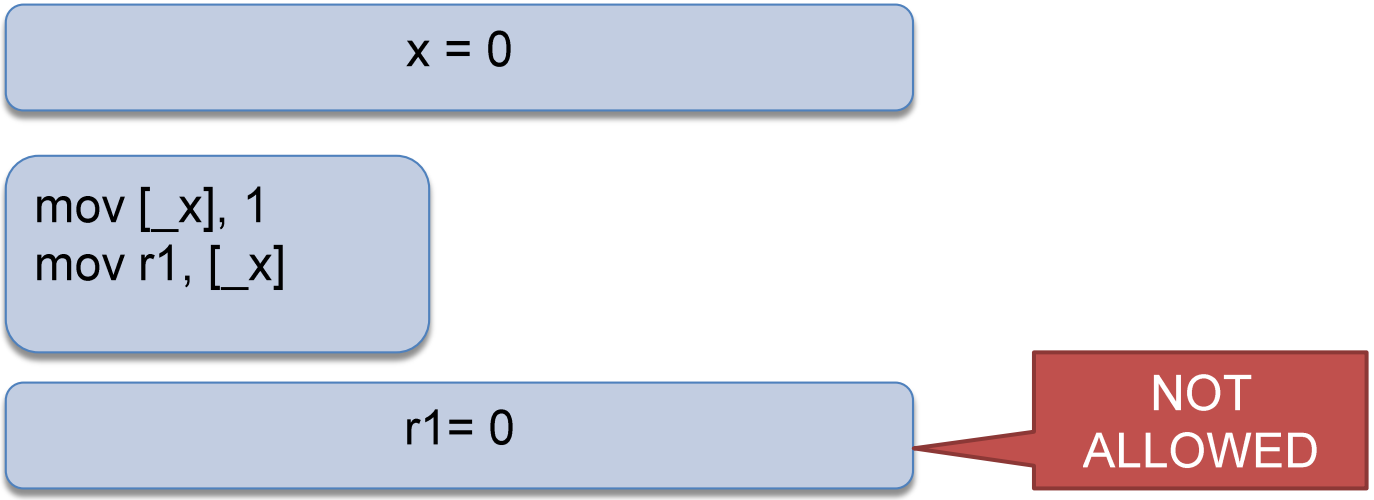
No reordering
R → W



Reordering
W(a) → R(b)

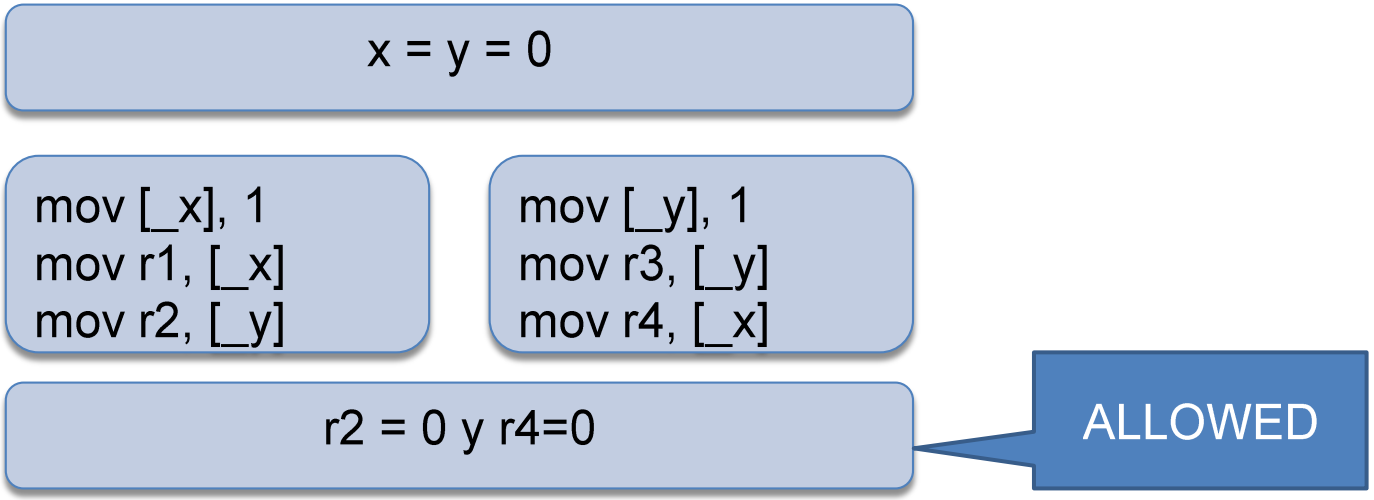


No reordering
W → R

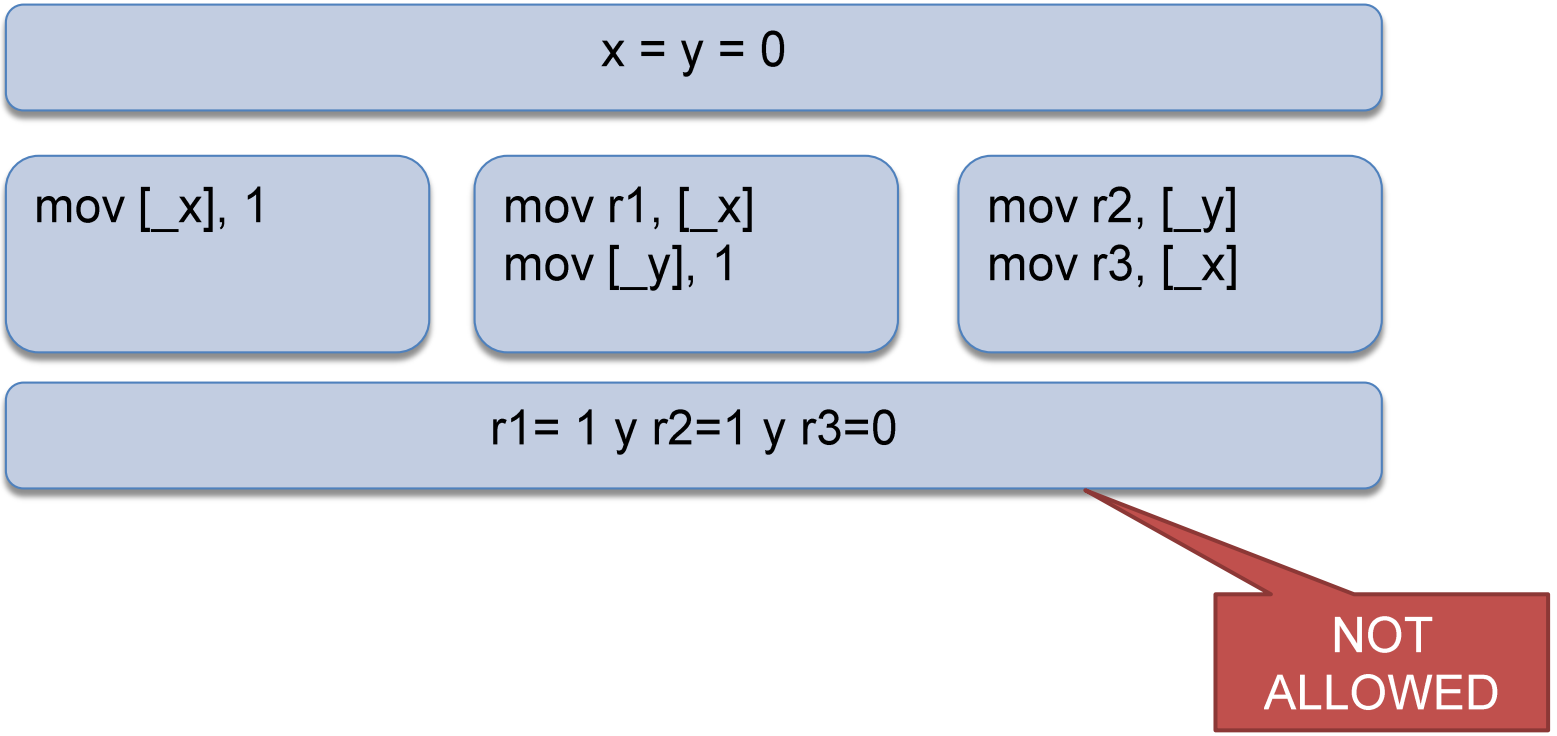


Visibility of writes from other processor

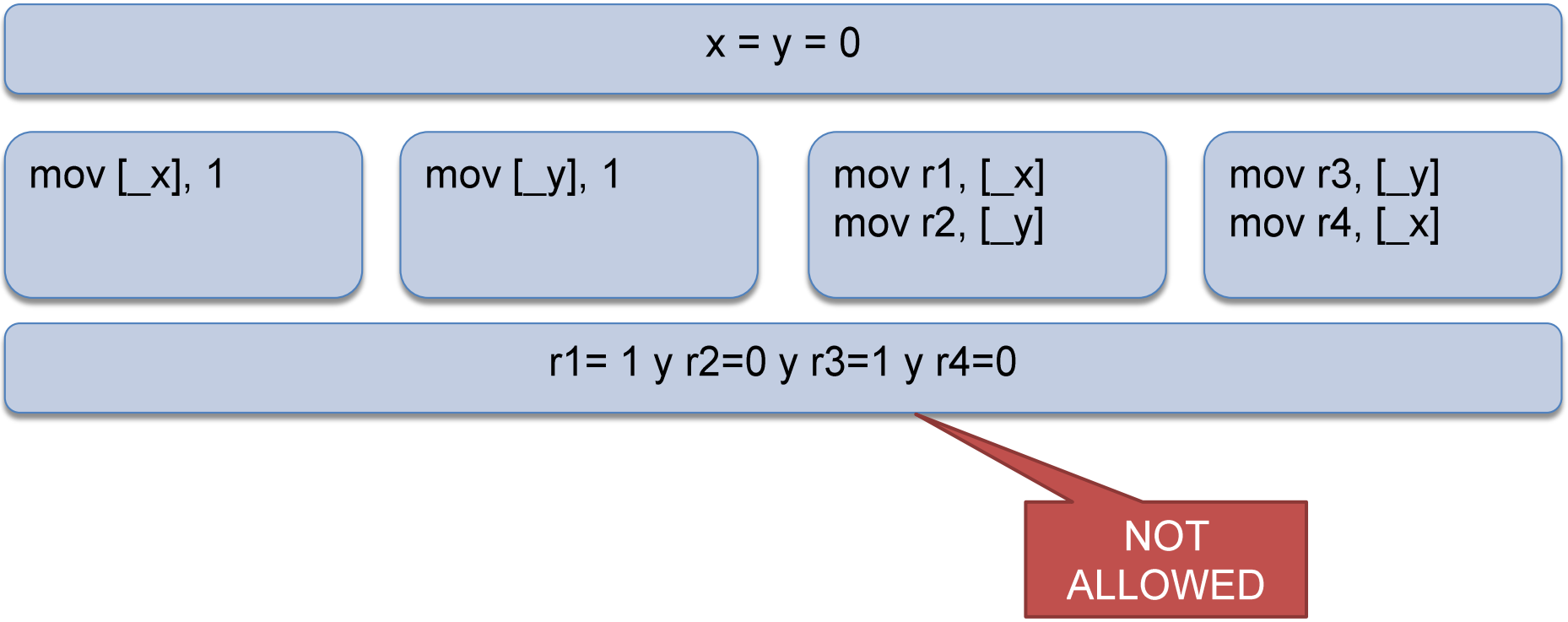
Writes may be perceived in different order by every processor



Transitive visibility of writes



Consistent write order for other processors



Locked instructions define total order

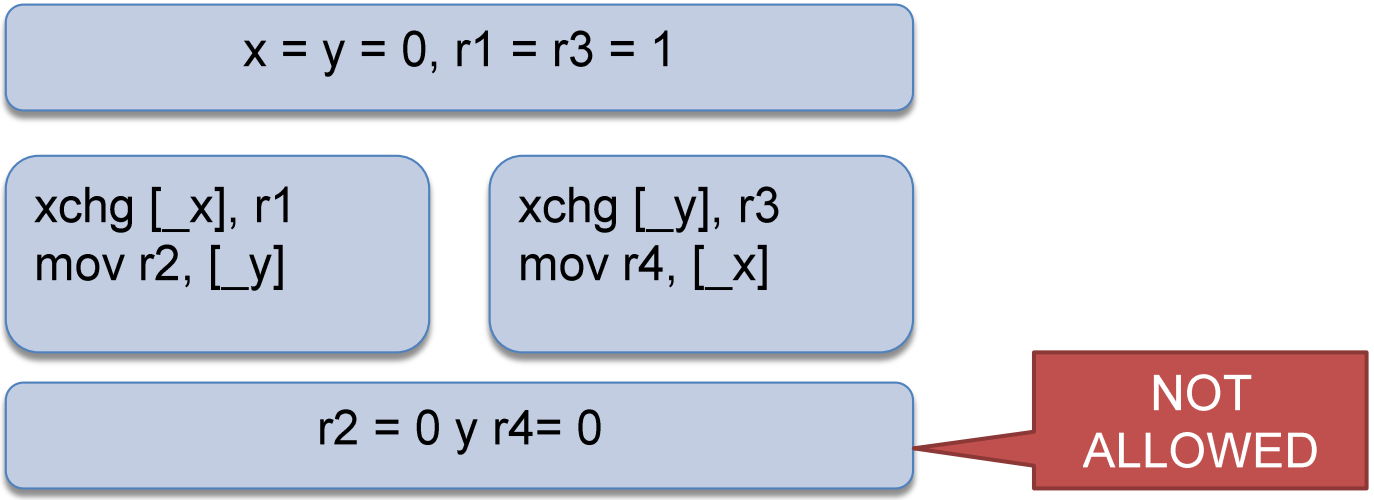
r1 = r2 = 1, x = y = 0

xchg [_x], r1	xchg [_y], r2	mov r3, [_x] mov r4, [_y]	mov r5, [_y] mov r6, [_x]

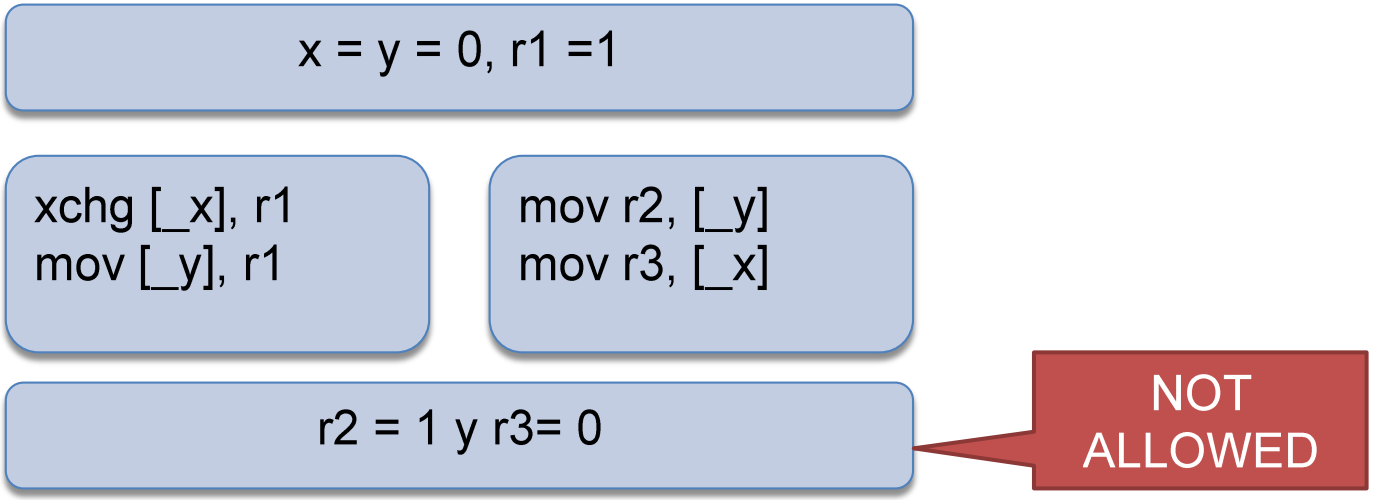
r1= 1 y r2=0 y r3=1 y r4=0

NOT ALLOWED

Reads not reordered with locks



Writes not reordered with locks



Practical effects

- Sequential consistency
  - Load: mov reg, [mem]
  - Store: xchg [mem], reg
- Relaxed consistency
  - Load: mov reg, [mem]
  - Store: mov [mem], reg
- Release/acquire consistency
  - Load: mov reg, [mem]
  - Store: mov [mem], reg

Referencia

- Computer Architecture. A Quantitative Approach. Fifth Edition.
  Hennessy y Patterson.
  Section: 5.6
- Adve, S. V., and Gharachorloo, K. Shared memory consistency models: A tutorial. IEEE Computer 29, 12 (December 1996), 66-76.
- Intel 64 and IA-32 Architectures Software Developer Manuals. Volume 3: Systems Programming Guide.
  - 8.2: Memory Ordering