

Advanced Multiprocessor Programming

Jesper Larsson Träff

traff@par.tuwien.ac.at

Research Group Parallel Computing

Faculty of Informatics, Institute of Computer Engineering

Vienna University of Technology (TU Wien)

This is confusing!

It's all about what can be observed!

Memory behavior (hardware) and memory models (software)

Two issues:

1. Which memory behaviors do actual hardware allow? What can be observed? "What set of values is a read allowed to return"? Is this reasonable?
2. How can possible behaviors be modeled or controlled such that it becomes possible to reason about correctness of real programs? What is desirable? How can undesirable behaviors be constrained?

This can all be very confusing. Many subtleties. No entirely satisfactory answers to all questions

Desirable(?)

Something like sequential consistency (SC) for memory behavior:

- Reads and writes are observed by any thread in the order issued, but can be delayed
- Outcome of concurrent program is some interleaving of the core's instructions in program order for each core

This is not what hardware and compilers actually provide

Sequential computing: Programs on Random Access Machine

```
// initially A==0, B==0
register int x = A;
```

```
A = 7;
```

```
A = 27;
```

```
register int x = A;
```

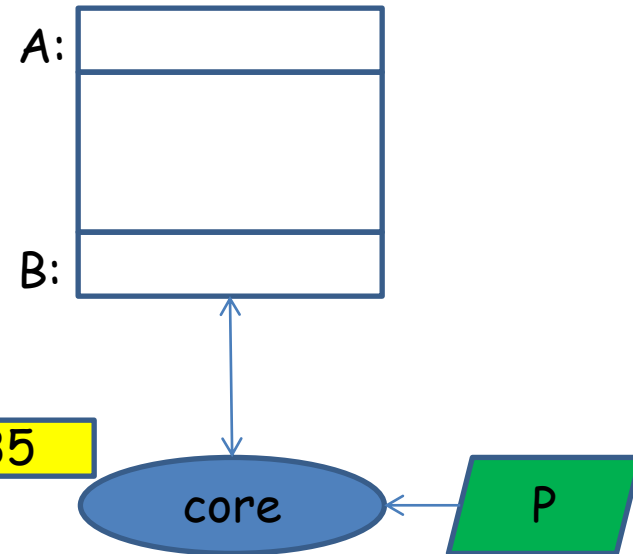
```
A = 35;
```

```
register int y = A;
```

```
assert(x==27);
```

```
assert(y==35);
```

← Not 7, not 35

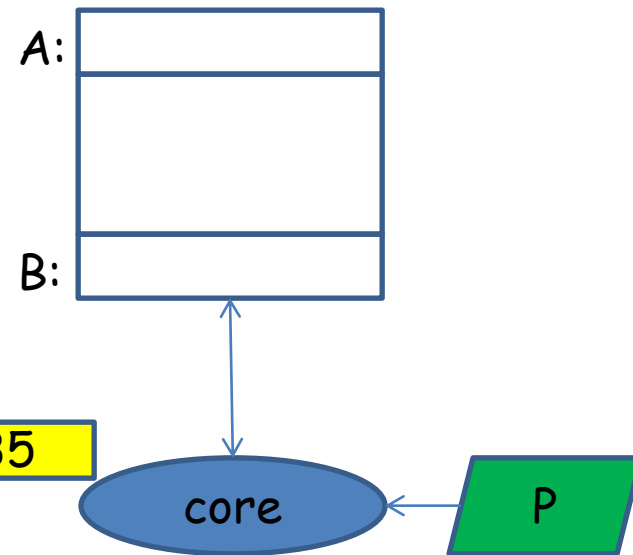


Loads and stores take place in **program order**: Load from address A returns the value of the most recent store to A. **No values** from the distant past (earlier stores are overwritten) or the near future (no speculation, "out of thin air")

Sequential computing: Programs on Random Access Machine

```
// initially A==0, B==0  
register int x = A;  
  
A = 7;  
A = 27;  
register int x = A;  
A = 35;  
register int y = A;  
assert(x==27);  
assert(y==35);
```

Not 7, not 35



This makes it “easy” to reason about program properties (correctness): Assertions and invariants

Sequential program order:

Running program P imposes a **total order** \prec_P on loads and stores $L(A)$, $S(B)$, A and B may be same or different memory location.

- $L(A) \prec_P S(B)$ if $L(A)$ is **executed** before $S(B)$ by P

And also between multiple loads and stores $L1$, $L2$, $S1$, $S2$:

- $L1(A) \prec_P L2(B)$ if $L1(A)$ is executed before $L2(B)$ by P
- $S1(A) \prec_P S2(B)$ if $S1(A)$ is executed before $S2(B)$ by P

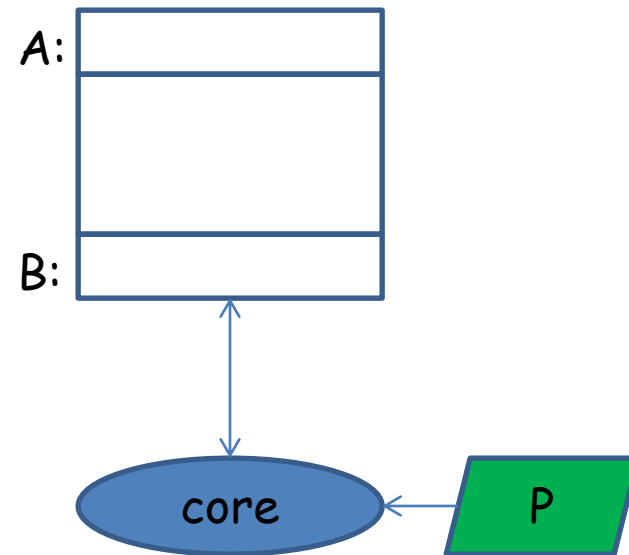
Program: Sequence of instructions/code (C, Java, C++, machine code) executed by a thread running on a core

Hidden assumption: Compiler essentially preserves program order (where it matters, reordering must be observationally transparent)

Sequential semantics

```
// initially A==0, B==0
register int x = A;

A = 7;
A = 27;
register int x = A;
A = 35;
register int y = A;
assert(x==27);
assert(y==35);
```



$$S(A,7) \prec_p S(A,27) \prec_p x=L(A) \prec_p S(A,35) \prec_p y=L(A)$$

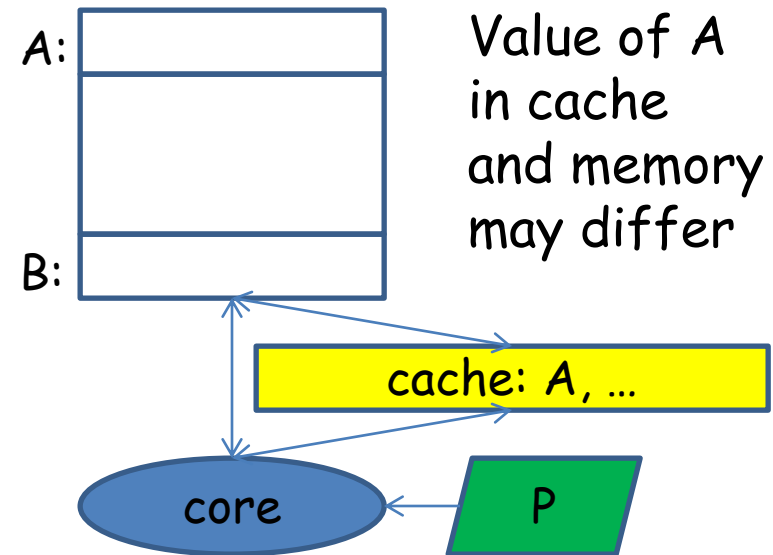
For any memory location A:

$L(A)$ returns the value written by $\text{MAX}\{S(A) \mid S(A) \prec_p L(A)\}$

Sequential semantics allows the use of coherent caches

```
// initially A==0, B==0
register int x = A;

A = 7;
A = 27;
register int x = A;
A = 35;
register int y = A;
assert(x==27);
assert(y==35);
```

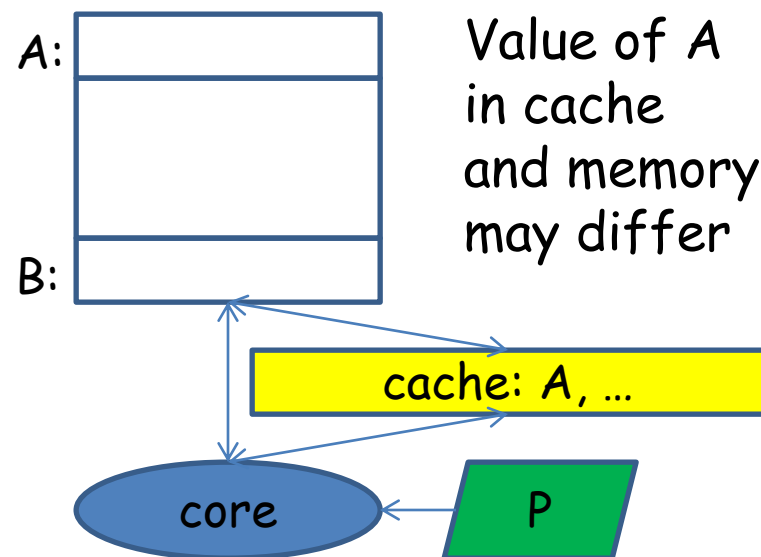


If A is stored in cache, all $S(A)$ must update cache, $L(A)$ must return value in cache

Sequential semantics allows the use of coherent caches

```
// initially A==0, B==0
register int x = A;

A = 7;
A = 27;
register int x = A;
A = 35;
register int y = A;
assert(x==27);
assert(y==35);
```

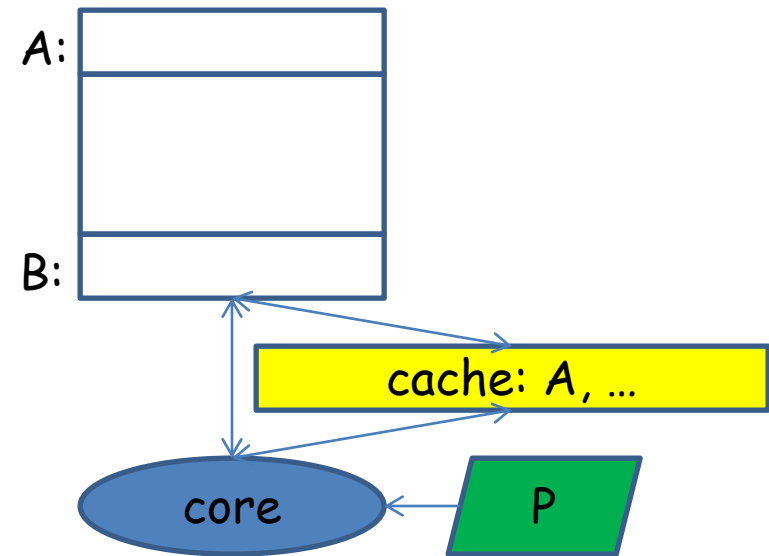


A coherent cache is **observationally transparent** (caching is **functionally invisible**): There is no way to determine by analyzing the outcomes of loads and stores whether the processor has cache(s)

Except by timing the operations (pragmatics)

Sequential semantics permits compiler optimizations

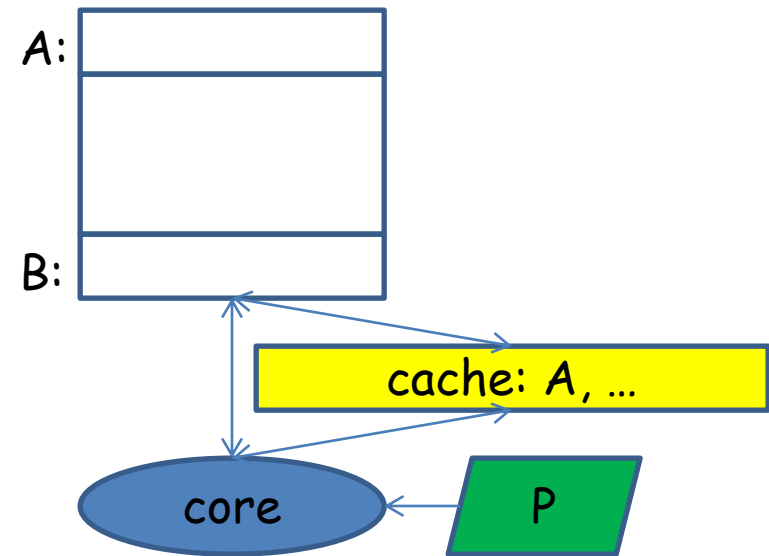
```
// initially A==0, B==0  
register int x = A;  
  
A = 7; B = 13;  
A = 27;  
register int x = A;  
A = 35;  
register int y = A;  
register z = B;  
assert(z==13);  
assert(x==27);  
assert(y==35);
```



Sequential semantics permits compiler optimizations

```
// initially A==0, B==0
register int x = A;

A = 7; B = 13;
register z = B;
A = 27;
register int x = A;
A = 35;
register int y = A;
assert(z==13);
assert(x==27);
assert(y==35);
```

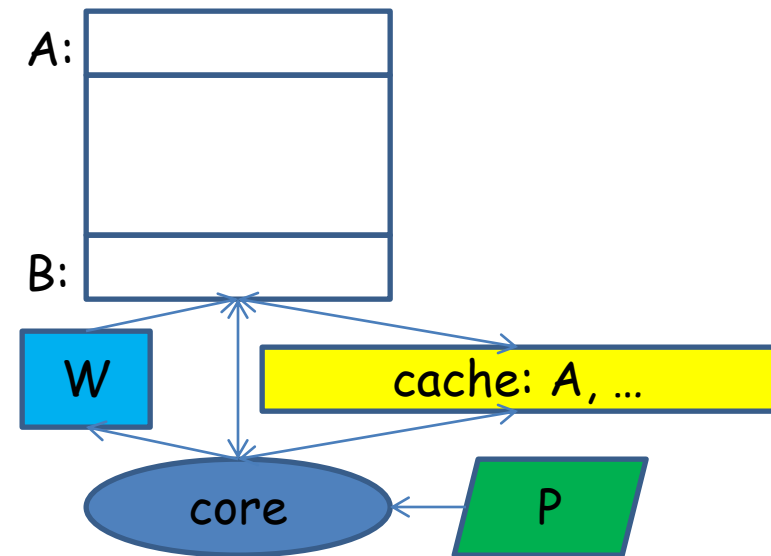


and other hardware optimizations (prefetching)... as long as functional dependencies are respected (A and B must be loaded before $C = A+B$; cannot be reordered after store to C)

Sequential semantics permits further hardware optimizations

```
// initially A==0, B==0
register int x = A;

A = 7; B = 13;
register z = B;
A = 27;
register int x = A;
A = 35;
register int y = A;
assert(z==13);
assert(x==27);
assert(y==35);
```



Write buffer: Batches writes to memory (FIFO or other order).
Any load A must return latest value in write buffer (or cache)

Why hardware optimizations

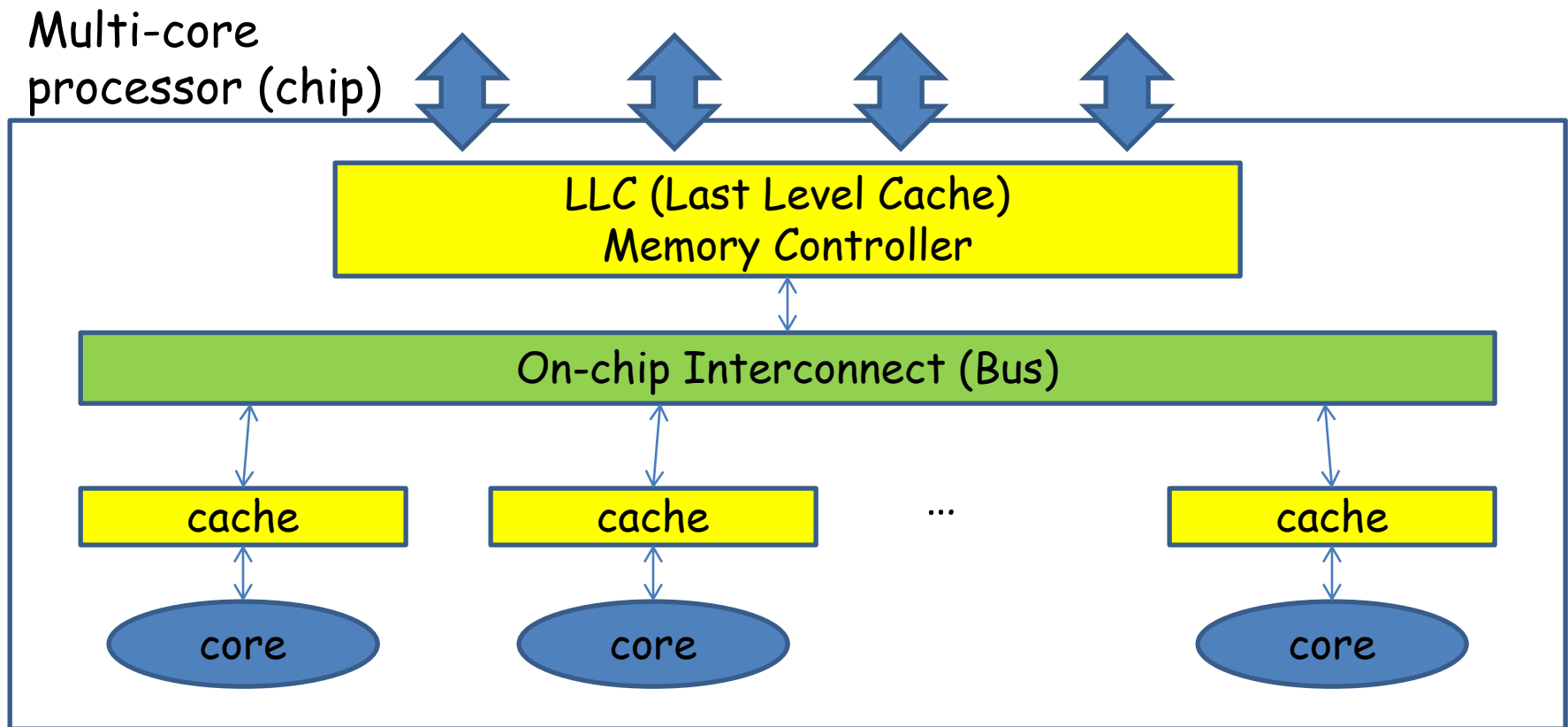
Cost of actual, direct memory access order(s) of magnitudes slower than performing an operation on registers (and cache lines)

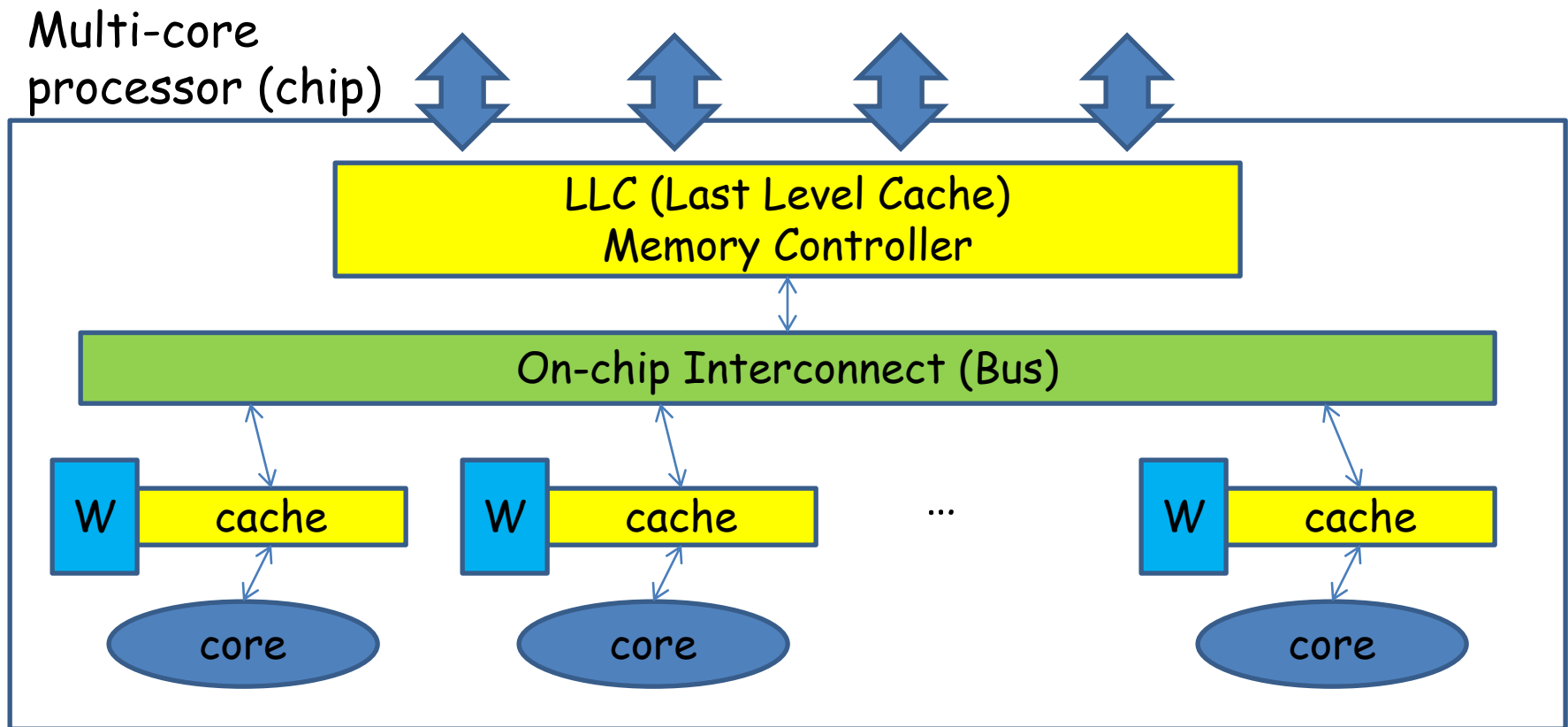
Caches, write buffers (prefetching, instruction reordering) can reduce average cost of memory accesses:

- Caches: Exploit (temporal and spatial) locality
- Write buffers: Hide/postpone latency of writes

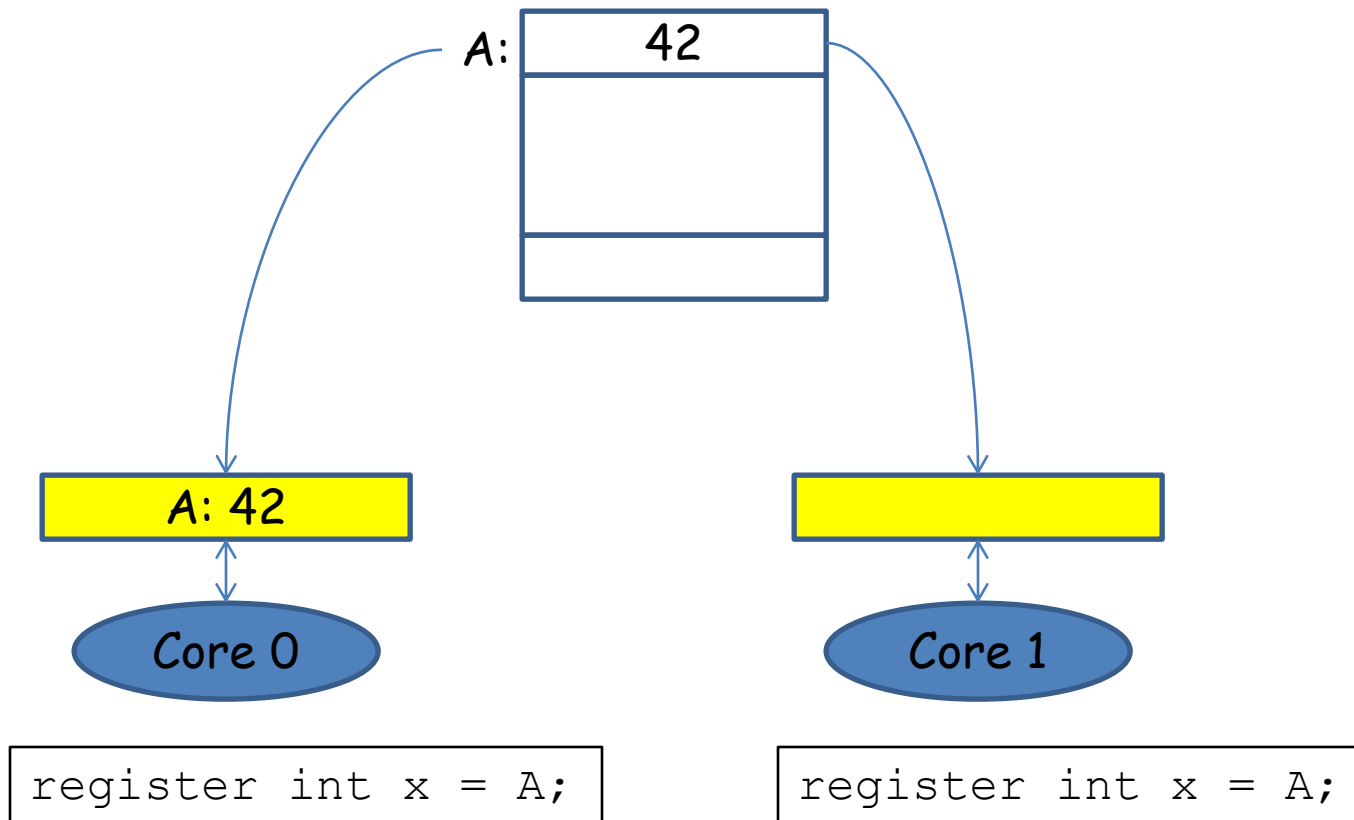
Sequential semantics: What is not observed (aka dependencies), does not matter (speculative, superscalar hardware)

Template multi-core parallel processor

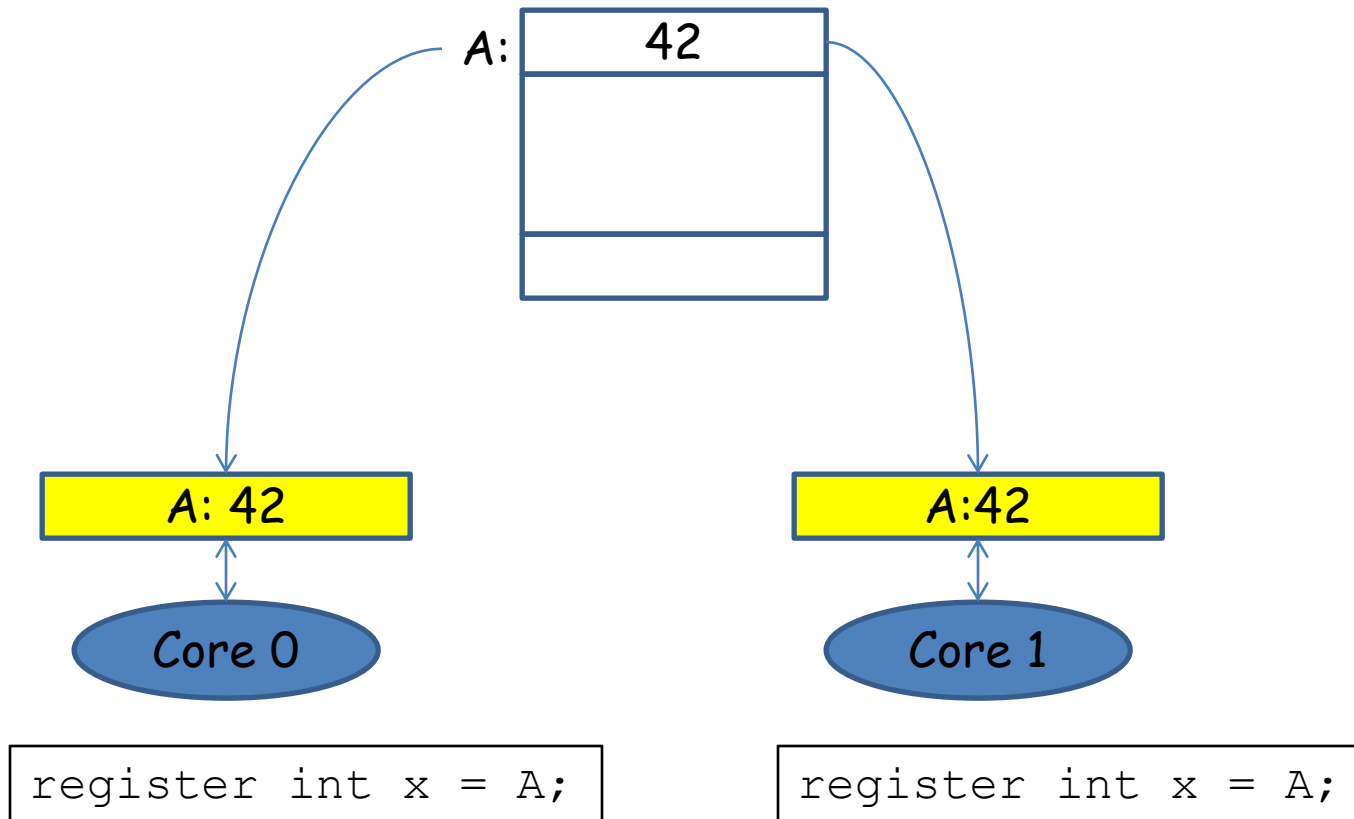




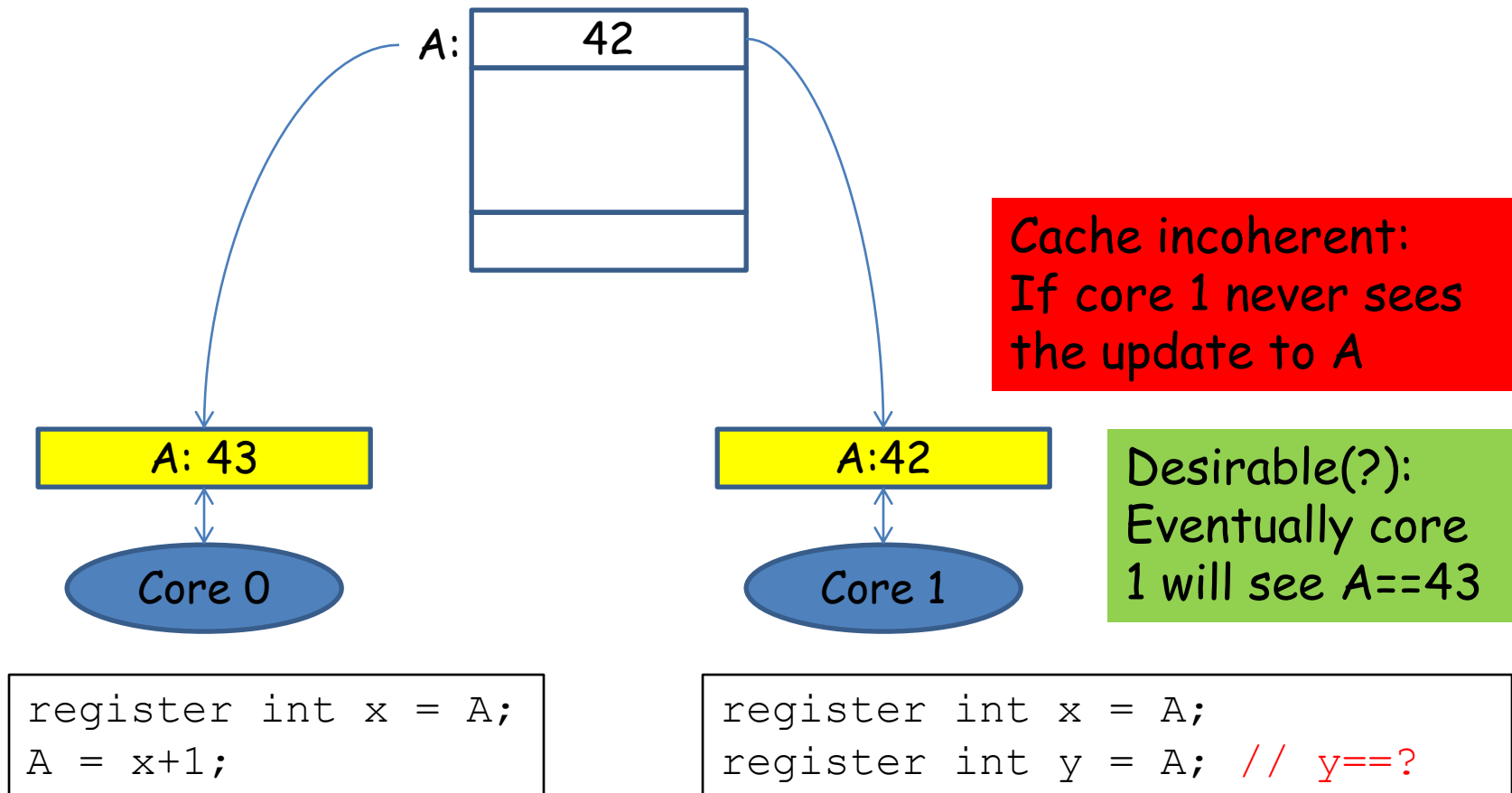
Cache coherence: Keeping caches observationally transparent



Cache coherence: Keeping caches observationally transparent



Cache coherence: Keeping caches observationally transparent



Definition 1 (SC like):

A cores' loads and stores to a single memory location must happen in a total order that respects the program order of each thread

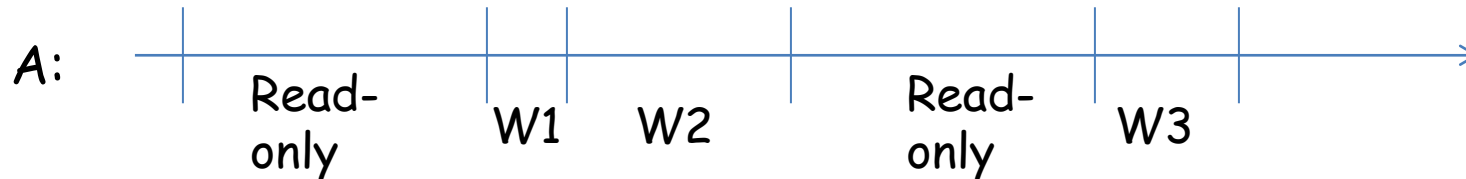
Definition 2 (Gharachorloo):

- Every store eventually becomes visible to each core
- Stores to the same memory location are serialized

Definition 3 (Hennessy&Patterson):

1. A load from memory location A returns the value of the most recent store to A by the same core, unless another core stored to A inbetween
2. A load from memory location A returns the value of the most recent store from another core, provided the load and the store are "sufficiently separated in time"
3. Stores to the same memory location are serialized

Definition 4 (epoch invariants): Assume lifetime of a memory location A divided into separate epochs



Cache is coherent if it maintains the following invariants:

- **Single-Writer, Multiple-reader:** For any memory location A , at any given time, either a single core may write to A (and read from A), or some number of cores may read from A
- **Data-value:** The value of the memory location at the start of an epoch is the same as the value at the end of the last write epoch.



Definitions 1-4 are equivalent.

Cache coherence protocol implements cache coherence (in hardware, transparently)

Cache coherence protocol (e.g, invalidate-based, MESI, ...) maintains the invariants of Definition 4.

Reminder: Cache granularity (performance and pragmatics)

Caches (normally) works on larger units than words, or programming language objects (int's, char's, double's, ...):

- **Cache block** (cache line: cache block + meta information)

Coherence is at the level of cache blocks; this may lead to **false sharing** in coherent caches

Other reminders (performance factors):

Cache capacity, associativity, replacement strategy, ...

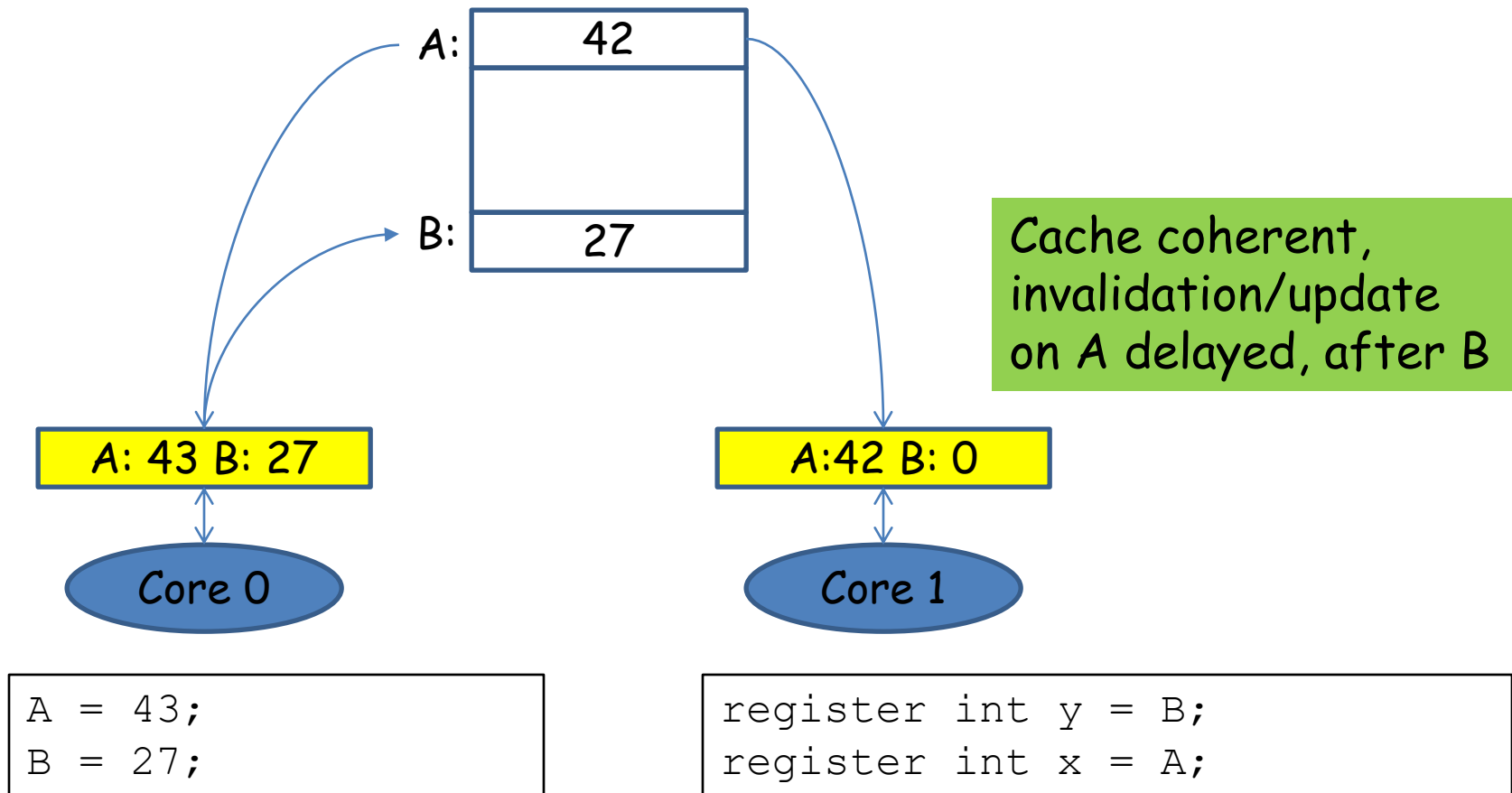
Cache coherence vs. memory consistency

Cache coherence is about the behavior of operations on a single memory location:

- Coherent: Eventually, other threads will “see” the stores, and stores will be seen in interleaved program order

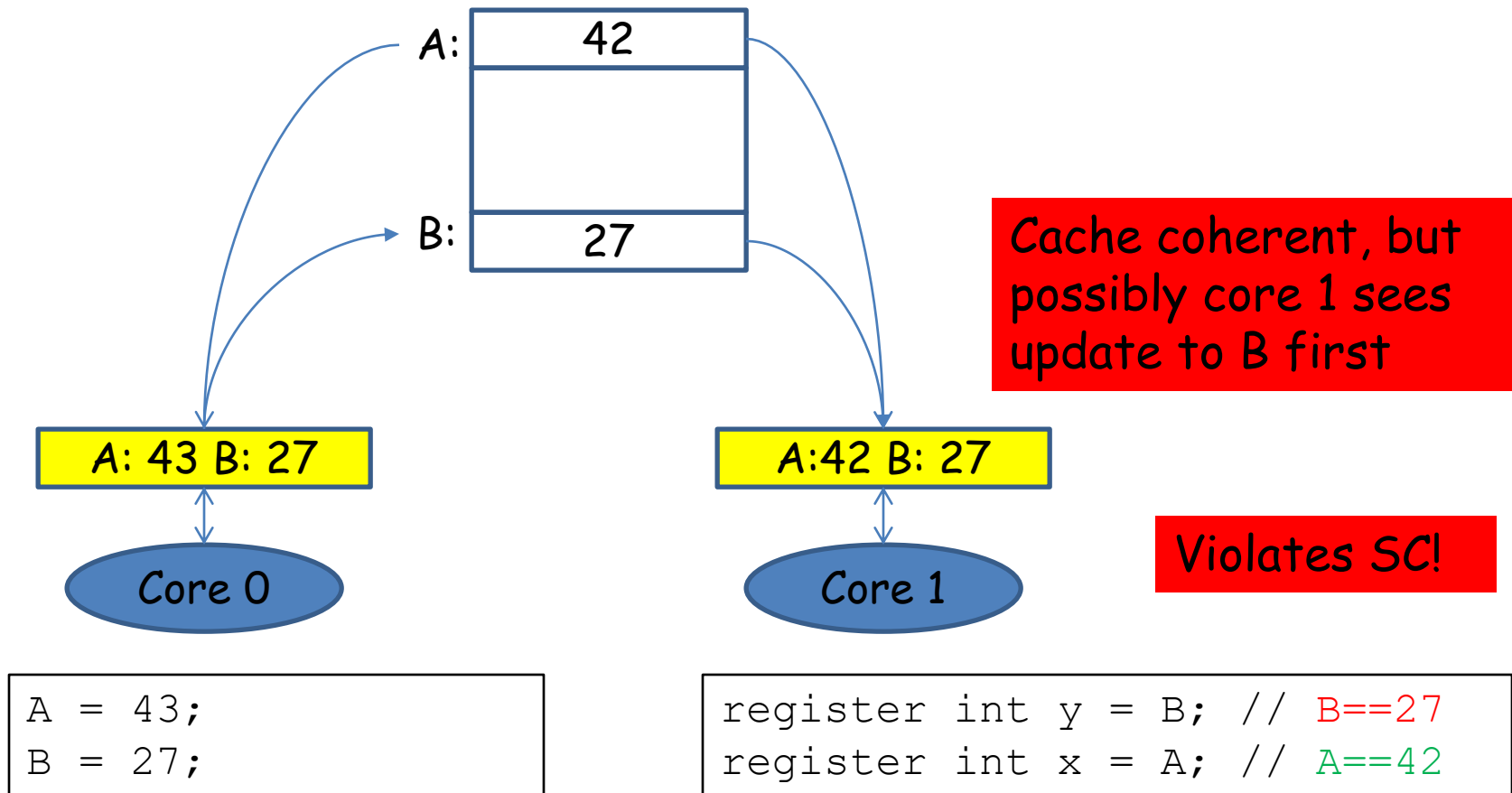
What about operations on different locations?

Cache coherence and different memory locations



Cache coherence and different memory locations

NOT $S(A, 43) <_M S(B, 27) <_M L(B == 27) <_M L(A == 42)$



Initially A and B are 0

Thread 0

```
A = data;  
B = true;
```

Thread 1

```
while (!B);  
register y = A;
```

Desirable (SC):
Thread 1 has $y == \text{data}$ (and **not** $y == 0$)

Common pattern:
Flag synchronization
between two threads

Basic synchronization pattern between two threads:
Flag B determines when one thread has finished and the other can take over, all data written by thread 0 should be available to thread 1

Initially A and B are 0

Thread 0



Thread 1

```
while (!B);  
register y = A;
```

Store to A goes to write buffer, is delayed, and effectively reordered after store to B

Thread 1 could read $A == 0$ into y

With non-FIFO write buffer, a reorder of both writes to A and B could be possible

Initially A and B are 0

Thread 0

```
A = value1;  
register int x = B;
```

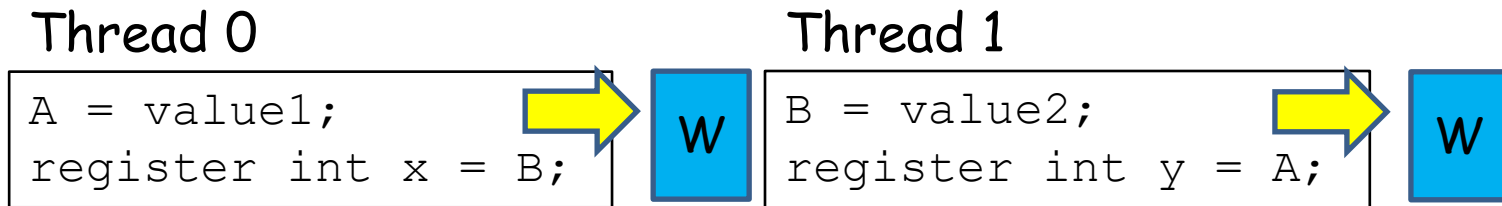
Thread 1

```
B = value2;  
register int y = A;
```

Obviously possible outcomes:

- $(x,y) = (value2,value1)$
- $(x,y) = (0,value1)$
- $(x,y) = (value2,0)$
- Threads run in lock-step
- Thread 0 before thread 1
- Thread 1 before thread 0

Initially A and B are 0



Obviously possible outcomes:

- $(x,y) = (value2,value1)$
- $(x,y) = (0,value1)$
- $(x,y) = (value2,0)$
- Threads run in lock-step
- Thread 0 before thread 1
- Thread 1 before thread 0

Is $(x,y) = (0,0)$ possible?

Yes: Write buffers (even FIFO) may delay the (single) write per thread

Initially A and B are 0

Thread 0

```
A = 1;  
if (B==0) {  
    // alone?  
}
```

Thread 1

```
B = 1;  
if (A==0) {  
    // alone?  
}
```

Simplified Dekker's algorithm: Is mutual exclusion guaranteed?

No. Write buffers may delay both stores to A and B

Initially A and B are 0

Non-hardware issue

Thread 0

```
A = value1;  
register int x = B;
```



```
register int x = B;  
A = value1;
```

Compiler orders load
early to hide latency

Thread 1

```
B = value2;  
register int y = A;
```



```
register int y = A;  
B = value2;
```

Is $(x,y) = (0,0)$ possible?

Yes: Compiler could reorder
independent assignments

Consistency problems, consistency models

Hardware aspects: Cores read from and write to memory.

What can other cores observe? Why? How to model and axiomatize possible hardware behaviors

Software aspects: Threads read from and write to memory as determined by their programs.

What can threads observe? Why (hardware and compiler)? How to reason about possible behaviors? How write correct and portable programs?

Memory consistency model

Definition 1:

A memory consistency model **constrains** what a core (or thread) can observe on load and store operations to different locations

Definition 2 (Sorin, Hill, Wood):

A memory consistency model is a specification of the **allowed load/store behavior** of a multi-core (multi-threaded) program executing with shared memory.

An implementation (in hardware, or software) of a memory consistency model ensures that only allowed behaviors can happen (**safety**), and that some allowed behavior will happen (**liveness**)

Sequential consistency (SC)

Lamport:

A multi-processor (-core) system 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**"

Leslie Lamport: How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. IEEE Trans. Computers 28(9): 690-691 (1979)

"Some sequential order..."

- All operations of all processor-cores can be totally ordered
- All operations (loads and stores to memory locations) in this order take place instantaneously

Call the total order of the operations on the (fictitious) global memory the **memory order**:

- $S(x) \prec_M L(y)$, or $L(y) \prec_M S(x)$, and
- $L1(x) \prec_M L2(y)$, or $L2(y) \prec_M L1(x)$, or
- ...

for any loads and stores by any processor-cores

Sequential consistency (definition):

"...in the order specified by its program" (program order, \prec_p)

1. $L(x) \prec_p L(y)$ implies $L(x) \prec_M L(y)$
2. $L(x) \prec_p S(y)$ implies $L(x) \prec_M S(y)$
3. $S(x) \prec_p S(y)$ implies $S(x) \prec_M S(y)$
4. $S(x) \prec_p L(y)$ implies $S(x) \prec_M L(y)$

Furthermore, a load must return the value from memory written by the most recent store

$L(a)$ returns value written by $\text{MAX}\{S(a) \mid S(a) \prec_M L(a)\}$

Sequential consistency (definition):

"...in the order specified by its program" (program order, \prec_p)

Another way to specify SC: The following instruction (program) orders by a single processor must be **preserved**, **not reordered**:

1. Load \rightarrow Load
2. Load \rightarrow Store
3. Store \rightarrow Store
4. Store \rightarrow Load

"cannot be reordered":

If a thread performs loads and stores in order 1-4, no thread must be able to observe these loads and stores in a different order

SC: No reordering of memory instructions

Flag-synchronization works under SC (initially $A, B = (0,0)$)

Thread 0

```
A = value1;
register int x = B;
```

Thread 1

```
B = value2;
register int y = A;
```

Under SC $(x,y) = (0,0)$ is **not possible!**

Assume for the sake of contradiction that $(x,y) = (0,0)$.

Then

- $L(B) <_M S(B)$
- $L(A) <_M S(A)$

By program order

- $S(A) <_p L(B)$
- $S(B) <_p L(A)$

Contradiction, by transitivity and SC: $L(B) <_M S(B) <_M L(A) <_M S(A)$

Implementing sequential consistency

Can SC be implemented in hardware? At what cost/penalty?

- Trivial implementation 1: Software multi-threading
- Trivial implementation 2: Funnel access to memory through switch (one core at a time has control over memory)
- Non-trivial implementations: Use cache-coherency protocol

Many believe that SC is too constraining on desired hardware optimizations (e.g. comes with a too high penalty). **But not all:**

Milo M. K. Martin, Mark D. Hill, Daniel J. Sorin: Why on-chip cache coherence is here to stay. *Commun. ACM* 55(7): 78-89 (2012)

The fictions of shared memory and global time (god&Newton)

The mental model of SC (and later TSO) is an actual, shared memory, and some globally ordered sequence of operations on this memory

However:

- Storing information takes physical space: Memory is always distributed (over the chip, over several chips, ...)
- Updating memory takes (signal propagation) time, and the time to reach different physical locations may be different
- The order in which one processor observes updates may therefore be different from the order in which another/any other processor observes updates

Total Store Order (TSO)

Allow the use of FIFO write buffer by allowing stores to be ordered after loads

Write buffer:

- Stores are sent to the write buffer, and later written to memory
- Load $L(A)$ must return value most recently written to A by processor (**bypassing**)

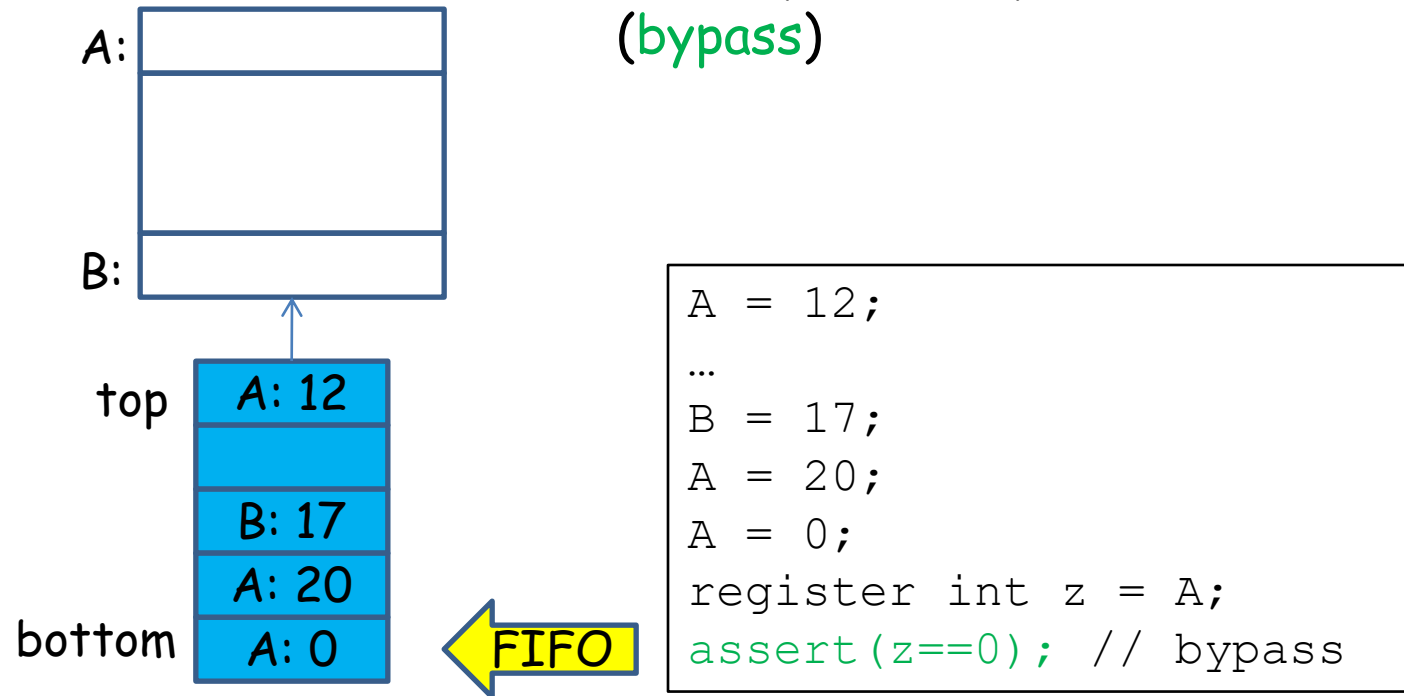
Thus: A store can be postponed till after a load in the memory order

Further non-FIFO write buffer optimizations

- Coalescing
- Sorting
- ...

See later remarks

Memory receives from top of FIFO,
processor writes to bottom, and
reads (searches) from bottom
(bypass)



Coalescing would maintain only one entry for A

Total Store Order (definition):

- | | |
|--|--|
| 1. $L(x) \prec_p L(y)$ implies $L(x) \prec_M L(y)$ | 1. Load \rightarrow Load |
| 2. $L(x) \prec_p S(y)$ implies $L(x) \prec_M S(y)$ | 2. Load \rightarrow Store |
| 3. $S(x) \prec_p S(y)$ implies $S(x) \prec_M S(y)$ | 3. Store \rightarrow Store |
| 4. NOT: $S(x) \prec_p L(y)$ implies $S(x) \prec_M L(y)$ | 4. NOT Store \rightarrow Load |

Furthermore, a load must return the value written by the most recent store whether from memory or from write buffer

$L(a)$ returns value of $\text{MAX}\{S(a) \mid S(a) \prec_M L(a) \text{ or } S(a) \prec_p L(a)\}$

Flag based synchronization works under TSO:

1. Thread 0 reads and writes A_0, A_1, \dots, A_n
2. Thread 0 writes synchronization flag F
3. Thread 1 waits on synchronization flag F
4. Thread 1 reads what thread 0 has written in A_0, A_1, \dots, A_n

Even if the write to F of thread 0 is postponed, all loads and stores before must be observed as before F by thread 1

Initially A and B are 0

Thread 0

```
A = 1;  
if (B==0) {  
    // alone?  
}
```

Thread 1

```
B = 1;  
if (A==0) {  
    // alone?  
}
```

Simplified Dekker's algorithm. Is mutual exclusion guaranteed?

Not under TSO

Initially A and B are 0

Thread 0

```
A = value1;  
register int a = A;  
register int b = B;
```

Thread 1

```
B = value2;  
register int x = B;  
register int y = A;
```

Possible that $(a,x) = (\text{value1}, \text{value2})$ even if $(b,y) \neq (0,0)$?

Natural **counter-argument**: If $(b,y) \neq (0,0)$ then the stores must have taken place after the loads into a and x, thus the values of a and x should also be 0

However, **bypassing** makes it possible for $a = A$; to load value1 (from core 0's write buffer) and $x = B$; to load value2, and load $b = B$; and $y = A$; from memory, thus $(a,x) = (\text{value1}, \text{value2})$ and $(b,y) \neq (0,0)$

Memory consistency models

A memory consistency model MC1 is **weaker than** a memory consistency model MC2 if the executions allowed by MC2 is a subset of the executions allowed by MC1 (the **stronger model** is **more constraining** on allowed executions)

Theorem:

TSO is weaker than SC

Two memory models are **incomparable** if neither is weaker than the other (weaker than only a partial order)

Enforcing order: The memory FENCE

A mechanism for enforcing order (preventing reordering) is needed

Memory FENCE (hardware instruction): Core-local memory operation across which other memory operations cannot be reordered

Bad names (but often used): Memory barrier, sync



FENCE is core/thread local, does not involve other cores/threads, it is not a synchronization operation

Total Store Order FENCE (definition):

1. $L(x) \prec_p \text{FENCE}$ implies $L(x) \prec_M \text{FENCE}$
2. $S(y) \prec_p \text{FENCE}$ implies $S(y) \prec_M \text{FENCE}$
3. $\text{FENCE} \prec_p \text{FENCE}$ implies $\text{FENCE} \prec_M \text{FENCE}$
4. $\text{FENCE} \prec_p L(x)$ implies $\text{FENCE} \prec_M L(x)$
5. $\text{FENCE} \prec_p S(y)$ implies $\text{FENCE} \prec_M S(y)$

For TSO, it suffices to state

1. $S(y) \prec_p \text{FENCE}$ implies $S(y) \prec_M \text{FENCE}$
2. $\text{FENCE} \prec_p L(x)$ implies $\text{FENCE} \prec_M L(x)$

Total Store Order FENCE (definition):

Equivalently, the following orders must be preserved

1. Load -> FENCE
2. Store -> FENCE
3. FENCE -> FENCE
4. FENCE -> Load
5. FENCE -> Store

In addition to

1. Load -> Load
2. Load -> Store
3. Store -> Store

But **NOT** Store -> Load

Using FENCE for SC with TSO

In TSO, Store can be postponed and ordered after Load (same: Load can be ordered before Store), only difference to SC model

To get SC behavior, FENCE must be inserted after each Store that may be followed by a Load (conservative, but expensive: after each Store)

TSO in hardware allows the use of FIFO write buffers

TSO seems to be the memory model of x86-processors

- No formal specification published
- Vendors (Intel) do not reveal all details

Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, Magnus O. Myreen: x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM* 53(7): 89-97 (2010)

Exercise:

Write and execute a program that shows that your x86 is not SC (but hopefully TSO)

Atomic operations (registers)

- Load
- Store
- RMW (read-modify-write)

Atomic load/store operations: Totally ordered, when some thread observes the effect of an atomic operation, all threads can observe the same effect (**atomicity**)

RMW: Atomically read and update memory location (globally ordered) without any intervening updates to the location

Implementing atomic RMW operations

- Special hardware (combining circuits), rare
- Exploiting cache-system: Get exclusive access to cacheline for the duration of the atomic operation, cache lines of other threads must be invalidated

Write buffer: Must be drained

On some hardware, atomic operations often entails a (or some type of) FENCE, but not always so

C11 and C++11 atomic operations

exchange (get-and-set), compare_exchange (CAS), fetch_add (fetch-and-add), ...

See earlier lecture

```
#include <stdatomic.h>
void atomic_thread_fence(memory\_order order);
```

```
#include <atomic>
extern "C"
void atomic_thread_fence(std::memory\_order order);
```

Neither C nor C++ were designed for multi-treading(!):

- Compilers were/are mostly thread-unaware and might perform optimizations that break expectations on memory model
- Threading support was originally added through libraries
- Thread libraries provide synchronization primitives to prevent data races (see later)
- Compilers explicitly forbidden to reorder across synchronization primitives

Hans-Juergen Boehm: Threads cannot be implemented as a library. PLDI 2005: 261-268

With C11 and C++11, threading support and explicit memory models were built into the languages

Enforcing load from memory: `volatile`

“volatile” means that the location can be modified from outside (other thread or device). Volatile accesses not reordered inside threads. **But** “volatile” does not mean “atomic”. Use case:

```
volatile bool locked;

void lock() { // test-and-test-and-set lock
    while (true) {
        while (locked);
        if (!atomic_exchange(&locked, true)) return;
    }
}

void unlock() {
    locked = false;
}
```

Does this work
without volatile?

```
bool locked;

void lock() { // test-and-test-and-set lock
    while (true) {
        while (locked);
        if (!atomic_exchange(&locked,true)) return;
    }
}
```



Compiler might optimize load away

```
bool locked;

void lock() {
    while (true) {
        if (locked) while (true);
        if (!atomic_exchange(&locked,true)) return;
    }
}
```

Infinite loop

We will also need this (volatile pointer):

```
node_t *volatile pred, *volatile curr,  
        *volatile * succ;
```

e.g., list-based set, skip list, pointers with stolen bits (marks, time stamps) that may/will be updated by other threads

Even weaker (relaxed) consistency models: Motivation

Flag synchronization does actually not require TSO:

1. Thread 0 reads and writes A_0, A_1, \dots, A_n
2. Thread 0 writes synchronization flag F
3. Thread 1 waits on synchronization flag F
4. Thread 1 reads what thread 0 has written in A_0, A_1, \dots, A_n

Load/Store order in Step 1 (and 4) does not have to be respected. It is only required that thread 1 “sees” all updates after the flag has been set

Some hardware/memory organizations permit (almost) all reorderings

Flag synchronization does actually not require TSO:

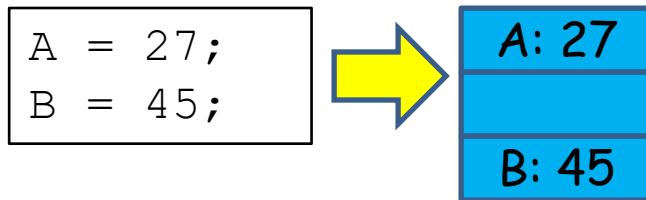
1. Thread 0 reads and writes A_0, A_1, \dots, A_n
2. Thread 0 writes synchronization flag F
3. Thread 1 waits on synchronization flag F
4. Thread 1 reads what thread 0 has written in A_0, A_1, \dots, A_n

Required order for correct flag synchronization (hand-over):

$$L(A_i), S(A_i) \prec_M S(F) \prec_M L(F) \prec_M L(A_i), S(A_i)$$

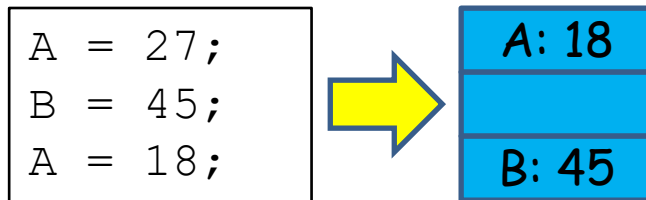
Further hardware optimization potential with less constraints

1. Coalescing write buffer



Further hardware optimization potential with less constraints

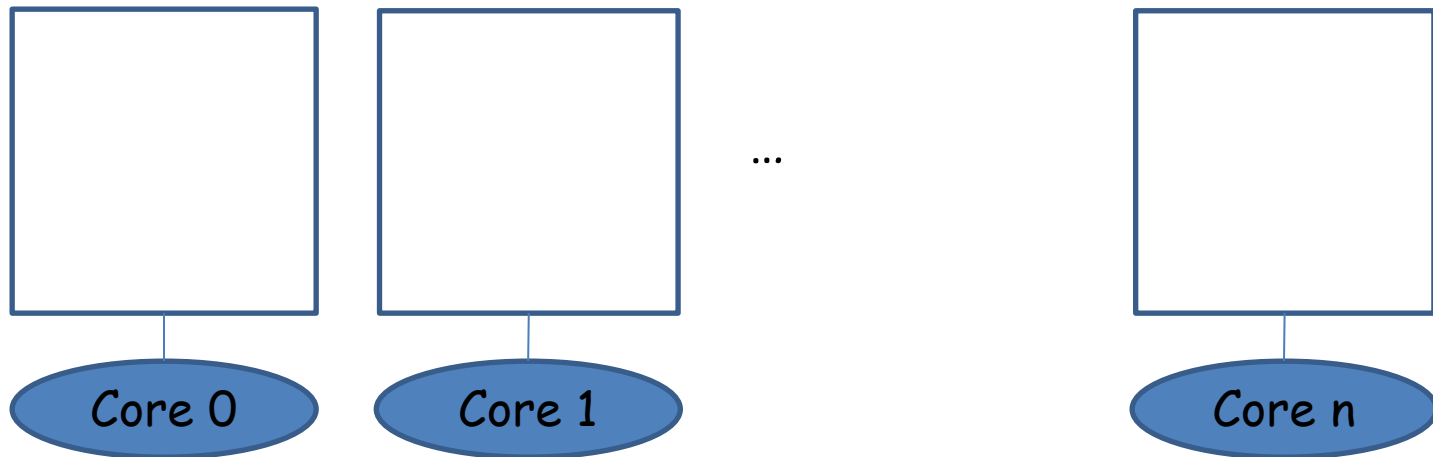
1. Coalescing write buffer



Other threads may see $A = 18$; before $B = 45$;

- NOT Store -> Store

NUMA (Non-Uniform Memory Access): Memory affinity

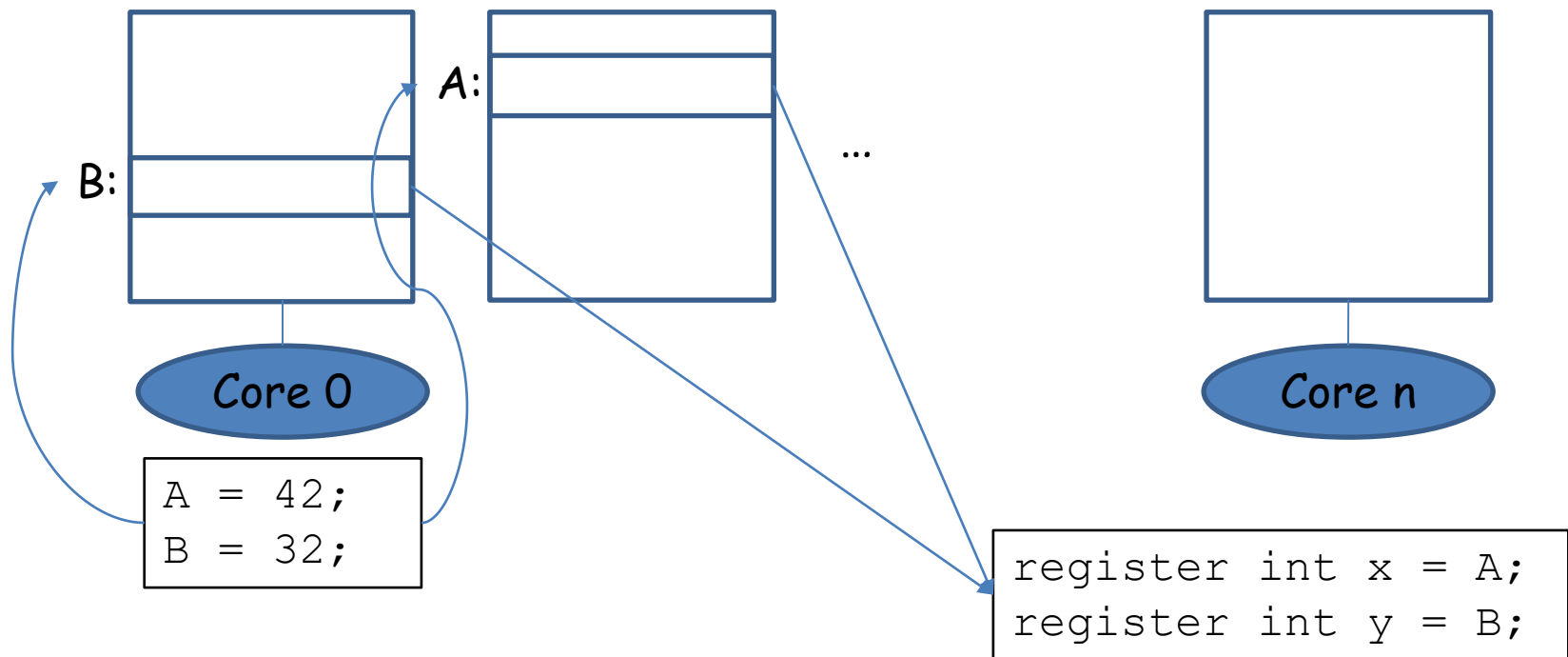


Shared memory divided physically into modules, each module is closer to some cores than to other cores

Performance/Pragmatics: Placement of program memory can be controlled (to some extent), e.g., first-touch policy

NUMA (Non-Uniform Memory Access): Memory affinity

Initially A and B are 0

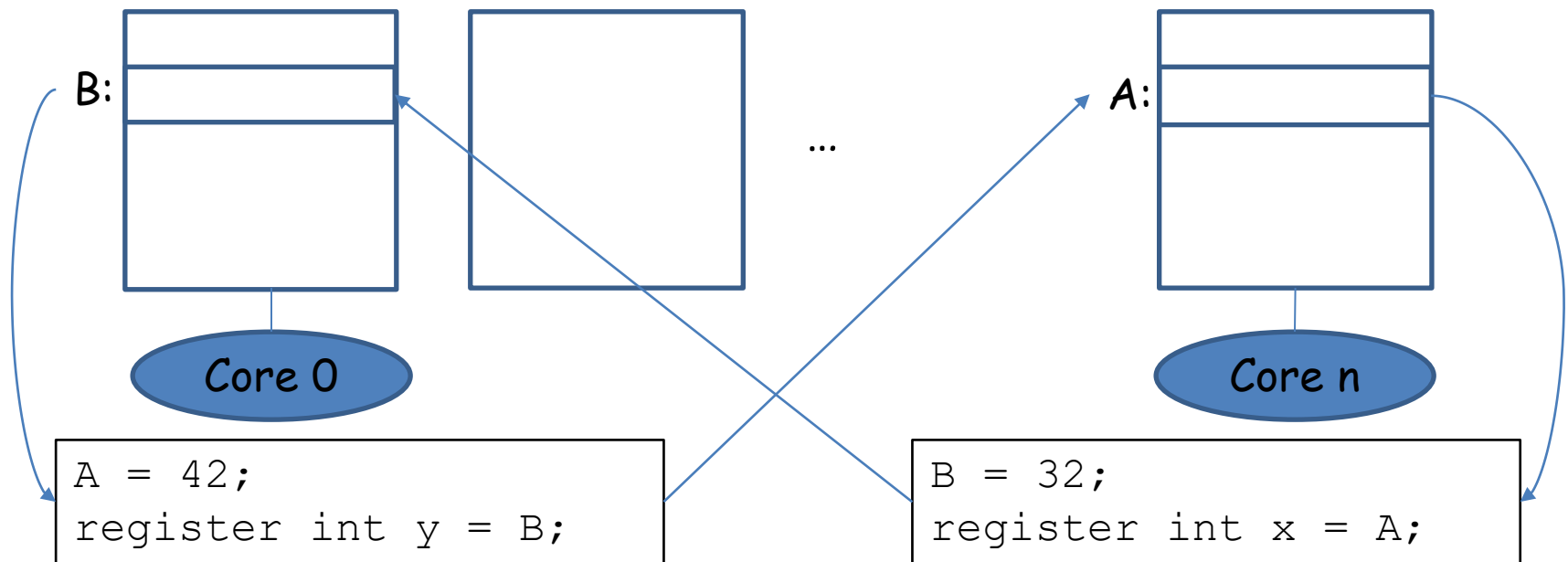


NOT Store -> Store

May see **A==0, B==32** (if write to A in other module is slow)

NUMA (Non-Uniform Memory Access): Memory affinity

Initially A and B are 0



$(x,y) = (0,0)$ may happen, even without write buffer (writes to the locations far away slow, reading the close locations fast):

NOT Store -> Load

Independent Reads of Independent Writes (IRIW)

Core 0:

```
A = 27;
```

There are architectures (POWER, ARM) where this can happen

Core 1:

```
B = 35;
```

Core 2:

```
register int x = A;
register y = B;
```

Core 3:

```
register int b = B;
register a = A;
```

Assume $x == 27$ and $b == 35$.

- If core 1 is farther away from core 2 than core 0, core 2 may load $y == 0$
- If core 0 is farther away from core 3 than core 1, core 3 may load $a == 0$

$(x, b) = (27, 35)$ but $(y, a) = (0, 0)$

- Core 2 observes: $S(A) < S(B)$
- Core 3 observes: $S(B) < S(A)$

No (total) memory order!

Independent Reads of Independent Writes (IRIW) under SC

Core 0:

```
A = 27;
```

Core 1:

```
B = 35;
```

Core 2:

```
register int x = A;  
register y = B;
```

Core 3:

```
register int b = B;  
register a = A;
```

Assume $x==27$ and $b==35$. Assume further $y==0$

Then $S(A,27) \prec_M L(A==27) \prec_M L(B==0) \prec_M S(B,35) \prec_M L(B==35) \prec_M L(A)$, so it must be that $a==27$

Under SC (and TSO), it cannot happen that both $(y,a)==(0,0)$

Independent Reads of Independent Writes (IRIW)

Core 0:

```
A = 27;
```

Core 1:

```
B = 35;
```

Core 2:

```
register int x = A;  
register y = B;
```

Core 3:

```
register int b = B;  
register a = A;
```

Assume $x == 27$ and $b == 35$.

FENCE operations cannot prevent $y == 0$, $a == 0$ (without further axioms), core 2 and core 3 can read $A == 27$ and $B == 35$ before the FENCEs that would have prevented both $y == 0$ and $a == 0$

Independent Reads of Independent Writes (IRIW)

Core 0:

```
A = 27;
```

Core 1:

```
B = 35;
```

Core 2:

```
register int x = A;  
register y = B;
```

Core 3:

```
register int b = B;  
register a = A;
```

Assume $x == 27$ and $b == 35$.

FENCE operations cannot prevent $y == 0$, $a == 0$ (without further axioms)

Problem: Stores do not take place immediately, possible that some threads observe the outcome of a store, but others not (yet)

Write atomicity (store atomicity):

Stores/writes take place immediately: If some core can observe the result of a store, then so can all cores

IRIW example:

Weak hardware memory models (e.g, NUMA systems) can **violate write atomicity**

Write atomicity implies that IRIW is correctly handled:

If $x==27$ and $b==35$, and furthermore $y==0$, then $S(A,27) <_M L(A==27) <_M L(B==0) <_M S(B,35) <_M L(B==35)$ which implies $L(A==27)$ by core 3, so $a=27$

Causality: If a location is read by one thread, a strictly later thread will see the same (or later) value

```
A = 27;
```

```
while (A!=27);  
FENCE;  
B = 35;
```

```
while (B!=35);  
FENCE;  
register int x = A;
```

By causality: $S(A,27) \prec_M L(A) \prec_M S(B) \prec_M x=L(A)$ and $x=27$

Observations:

- Write atomicity implies causality
- Causality does **not** imply write atomicity

Causality does **not** imply write atomicity (counter example)

Core 0, hyperthread 0:

```
A = 27;
```

W

Core 0, hyperthread 1:

```
register int x = A;  
register y = B;
```

Core 1, hyperthread 0:

```
B = 35;
```

W

Core 1, hyperthread 1:

```
register int b = B;  
register a = A;
```

Hyperthreads share write buffer (and other hw): Possible that $x==27$ and $b==35$ (bypassing), and $y==0$ and $a==0$ (stores are in write buffer), so write atomicity **does not hold**.

Core 0, hyperthread 0:

```
A = 27;
```

W

Core 1, hyperthread 0:

```
while (B!=35); FENCE;  
register int x = A;
```

W

Core 1, hyperthread 1:

```
while (A!=27);  
FENCE;  
B = 35;
```

On this example architecture, causality holds, A must have left write buffer before core 1, hyperthread 1 writes to B, and can therefore be read also by hyperthread 0

Core 0, hyperthread 0:

```
A = 27;
```

W

Core 0, hyperthread 1:

```
while (A!=27);  
FENCE;  
B = 35;
```

Core 1, hyperthread 0:

```
while (B!=35); FENCE;  
register int x = A;
```

W

Core 1, hyperthread 1:

Causality holds, FENCE enforces $L(A==27) \prec_M S(B,35)$ in memory order observed by core 1

Not a proof, example(s) from

Daniel J. Sorin, Mark. D. Hill, David A. Wood: A primer on memory consistency and cache coherence. Synthesis Lectures on Computer Architecture. Morgan&Claypool, 2011

Generic, weak memory consistency model XC (definition):

1. $L(x) \prec_p \text{FENCE}$ implies $L(x) \prec_M \text{FENCE}$
2. $S(y) \prec_p \text{FENCE}$ implies $S(y) \prec_M \text{FENCE}$
3. $\text{FENCE} \prec_p \text{FENCE}$ implies $\text{FENCE} \prec_M \text{FENCE}$
4. $\text{FENCE} \prec_p L(x)$ implies $\text{FENCE} \prec_M L(x)$
5. $\text{FENCE} \prec_p S(y)$ implies $\text{FENCE} \prec_M S(y)$

The (total) \prec_M relation describes what can be observed, not a physical memory.

Loads and stores to same location respect

1. $L(a) \prec_p L'(a)$ implies $L(a) \prec_M L'(a)$
2. $L(a) \prec_p S(a)$ implies $L(a) \prec_M S(a)$
3. $S(a) \prec_p S'(a)$ implies $S(a) \prec_M S'(a)$

Load gets value of last local or global store

$L(a)$ returns value of $\text{MAX}\{S(a) \mid S(a) \prec_M L(a) \text{ or } S(a) \prec_p L(a)\}$

Equivalently, the following orders are preserved

1. Load -> FENCE
2. Store -> FENCE
3. FENCE -> FENCE
4. FENCE -> Load
5. FENCE -> Store

For accesses to same location (same as TSO):

1. Load -> Load
2. Load -> Store
3. Store -> Store

Model	Impl.	Load/Store	Fence
SC	Optional/Default	L→L, L→S, S→L, S→S	F→S, F→L, L→F, S→F, F→F
TSO (Processor Consistency)	x86, IBM 370, DEC VAX, (Sparc)	L→L, L→S, S→S	Same
Partial Store Order	Sparc	L→L, S→S	Same
Weak Ordering	PowerPC (IBM)		Same
Release Consistency	Alpha, MIPS		FA→S, FA→L, L→FA, W→FR, FA→FA, FA→FR, FR→FA, FR→FR

From Hennessy/Patterson, "Computer Architecture", 2nd Ed. 1996

Release consistency

FENCE operations for "ending a load store sequence" (RELEASE FENCE, FR) and "beginning a load store sequence" (ACQUIRE FENCE, FA):

1. Load, Store -> RELEASE
2. ACQUIRE -> Load, Store
3. ACQUIRE -> ACQUIRE
4. ACQUIRE -> RELEASE
5. RELEASE -> ACQUIRE
6. RELEASE -> RELEASE

Scott's notation

Distinguish between synchronizing (atomic, fences) and ordinary memory operations (non-atomic, load, store). Together, call these "operations"

From:

Michael L. Scott: Shared-memory synchronization. Synthesis Lectures on Computer Architecture. Morgan&Claypool, 2013

Terminology, assumptions:

- **Coherence:** All operations (synchronizing, ordinary) to any single location appear to occur in some single, total order from the perspective of all threads
- **Global order:** Synchronizing operations happen in some total order, in which operations by the same thread happen in program order
- **Program order:** Operations happen in the order specified by the (compiled) program from the perspective of the issuing thread; other threads may see operations in a different order
- **Local order:** Ordinary operations may be reordered wrt. synchronizing operations by the issuing thread, except when explicitly forbidden
- **Value read:** Any read operation returns the value most recently written value in the order observed by the thread

Local order of ordinary operations wrt. synchronizing operations is specified explicitly with the following annotation on the synchronizing operation

`x.load/store/rmw/fence(arg,P||S)`

`||`: A fence across which the specified kinds of local operations cannot be reordered

P: Preceding, local R(ead) and/or W(rite)

S: Succeeding, local R(ead) and/or W(rite)

```
a1 = x1; a2 = x2; x3 = a3;  
a = x.load(R||W); // atomic  
b = x4;
```

Reads of x1, x2 cannot be ordered after read of x (write to x3 can). Write to b cannot be ordered before read of x

Locks (and data structures):

Lock release must ensure that all data read and written in the critical section must be visible (all load/stores) completed. Lock release must have at least a `RW||` fence (as part of a synchronizing store)

Lock acquire should ensure that the thread cannot read/write shared data before it has the lock. Acquire should be prefixed by a `||RW` fence (as part of synchronizing load/store)

Lock implementations (pthreads library, OpenMP, C++) usually make this guarantee

Annotations and hardware memory consistency

Hardware SC: No $||$ annotations needed, hardware guarantees global order consistent with program order

Hardware TSO: $R||R$, $R||W$ and $W||W$ annotations not needed, these orderings are guaranteed by hardware; but $W||R$ must be asserted when needed

Relaxed models: Any reorderings are possible, all required orders must be asserted

fence($RW||RW$): Full fence

Portable (pseudo-)code: Make all ordering requirements explicit

Example: Test-and-test-and-set lock

```
int f = 0;

void lock() {
    while (test_and_set(&f))
        while (f); // spin on f
    fence(||RW);
}

void unlock() {
    f.store(0,RW||);
}
```


Memory consistency in programming languages

Concurrent/multi-threaded programming language must present a clear memory model to the programmer (which may or may not be identical to the model provided by the hardware):

- Portability
- (Formal) Reasoning about correctness
- Shielding from (too) strange effects of hardware model
- Control and constrain compiler optimizations (reordering, register usage)
- Safety (Java)

Adopting SC as the model is generally thought too restrictive and expensive (but feasible, compiler inserts FENCES)

C++/Java approach

- Programming language makes memory consistency model explicit (and threading model as well)
- Memory model can be used by programmer to enforce required load/store order
- Memory model must be respected by compiler optimizations
- Desirable if weaker (but faster) hardware memory consistency can be exploited (portably) by programmer

Data race, race condition

Data race (**race condition**):

Two or more cores (threads) read/write to a memory location with ordinary read/write operations, and at least one operation is a write

Outcome of a program with a race condition depends on the relative order of execution and of reads/writes, and is considered **bad** (non-deterministic, undefined behavior...)

Another view: Order of memory operations matter only where threads interact, through accessing shared memory locations (threads locally obey program order). Control order by controlling the races...

Race conditions avoided by protecting all updates to shared memory locations by synchronization constructs (e.g., locks) or by using atomic operations that establish an inter-thread order

Data Race Free (DRF):

A program is DRF if it has **no data races** when executed under SC: All accesses to shared locations are either via synchronization operations (atomics), or separated (ordered) by synchronization operations.

Alternatively:

A program is properly synchronized if all accesses to shared locations are ordered by synchronization operations

A DRF program can be treated as if it was executed under SC

- Between synchronization points, SC is established by local rules on program order.
- At synchronization points, FENCES may have to be added to ensure that memory operations are properly ordered

Sequential Consistency for Data Race Free programs: SC for DRF

- Programmer must ensure that the program is DRF if executed under SC
- Implementer must ensure that all executions of a DRF program behave like an SC execution (by inserting FENCES)

Note: Optimal placement of FENCES is undecidable; conservative placement not (extreme: FENCE after every load/store)

SC for DRF:

- Programmer must ensure that the program is DRF if executed under SC
- Implementer must ensure that all executions of a DRF program behave like an SC execution (by inserting FENCES)

Advantage:

SC for DRF allows (almost) all common compiler and hardware optimizations between synchronization points, order must be enforced at synchronization points possibly with fence operations

In particular, SC for DRF can be implemented on hardware with only weak memory ordering

Definition: SC for DRF

- Operations are either access (Load/Store) or synchronization
- Two **data operations D_i and D_j conflict** if they are from different cores (threads), access the same location and at least one is a Store
- Two **synchronization operations conflict** if they are from different cores (threads), if they access the same location, and are **not compatible**
- Synchronization conflict is transitive
- Two **data operations race** if they conflict, and they are not separated by a conflicting synchronization
- An SC execution is Data Race Free (DRF) if no operations race
- A program is DRF if all its SC executions are DRF
- A memory consistency model supports "SC for DRF" if all executions of all DRF programs are SC executions

C++11/C11 Memory model

- SC for DRF, undefined semantics for programs with data races (unlike Java)
- Atomic objects act as synchronization operations (establish order)
- Load/store/atomic (RMW) operations with explicit memory model
- Default model (for atomic loads and stores) is SC

Hans-Juergen Boehm, Sarita V. Adve: Foundations of the C++ concurrency memory model. PLDI 2008: 68-78


Resource: www.cplusplus.com

Establishing order in C/C++

- Atomic operations by different threads establish **synchronizes-with** relationship
- **Synchronizes-with** is transitive between threads
- Local operations has a **happens-before** (local) order(*)
- Inter-thread happens-before:
- Operation A (by thread 0) **happens-before** S (by thread 0) **synchronizes-with** S (by thread 1) **happens-before** B (by thread 1): A inter-thread **happens-before** B
- Memory operations are visible to threads in **happens-before** order

(*) not quite that: happens-before by sequenced-before

C++ flag synchronization with atomics



```
#include <atomic>
#include <thread>
std::atomic<bool> f;
```

...

a = data;

f.store(true);

...


synchronizes-with

```
#include <atomic>
#include <thread>
std::atomic<bool> f;
```

...

while (!f.load());

b = a;



Store to a happens-before load from a, **no race**

Atomic store synchronizes with atomic load (under the default memory model: SC)

Explicit memory model orderings on atomic variables:

- Acquire-Release
- Consume (?)
- Relaxed (no order, used in combination with other operations)

Load

```
memory_order_seq_cst  
memory_order_relaxed  
memory_order_acquire  
memory_order_consume
```

Store

```
memory_order_seq_cst  
memory_order_relaxed  
memory_order_release
```

RMW

```
memory_order_seq_cst  
memory_order_relaxed  
memory_order_acquire  
memory_order_release  
memory_order_acq_rel  
memory_order_consume
```

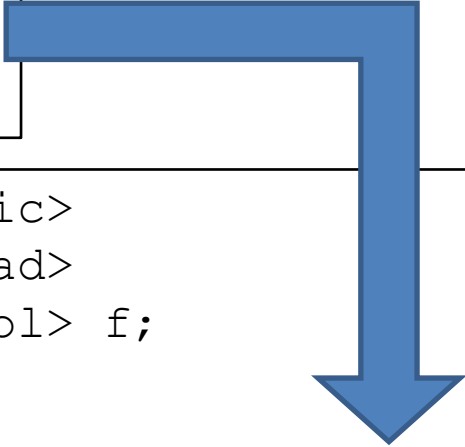
Synchronizes-with relationship (approximate):

- `memory_order_seq_cst` **with** `memory_order_seq_cst`
- `memory_order_acquire` (on load/RMW) **with** `memory_order_release` (on store/RMW)
- `memory_order_acq_rel` (on RMW) **with** `memory_order_release` (on store)
- `memory_order_acq_rel` (on RMW) **with** `memory_order_acquire` (on load)
- `memory_order_acq_rel` **with** `memory_order_acq_rel` (on RMW)
- `memory_order_release` (on store) **with** `memory_order_consume` (on load) (*)
- `memory_order_relaxed` **does not synchronize with anything**

(*) The `memory_order_consume` model may be deprecated (2016)

```
#include <atomic>
#include <thread>
std::atomic<bool> f;

...
a = data;
f.store(true,
        memory_order_release);
...
```



```
#include <atomic>
#include <thread>
std::atomic<bool> f;

...
while (!f.load(memory_order_acquire));
b = a;
...
```

```
#include <atomic>
#include <thread>
std::atomic<bool> f;

...
a = data;
f.store(true,
        memory_order_release);
A = someotherdata;
```

Operations before
atomic store **cannot be
ordered after**

With acquire-
release, the
atomic store **may
be ordered after**
succeeding local
operations

```
#include <atomic>
#include <thread>
std::atomic<bool> f;

...
while (!f.load(memory_order_acquire));
b = a;
B = A; // ?? No happens-before
...
```

```
#include <atomic>
#include <thread>
std::atomic<bool> f;

...
a = data;
A = someotherdata;
f.store(true,
       memory_order_release);
```

Operations after
atomic load **cannot be
ordered before**

With acquire-
release, the
preceding local
operations **may be
ordered after the
atomic load**

```
#include <atomic>
#include <thread>
std::atomic<bool> f;

...
B = A; // ?? No happens-before
while (!f.load(memory_order_acquire));
b = a;

...
```

Scott's notation:

- `.store(val, memory_model_release)` **like**
`.store(val, RW||)`
- `.load(memory_model_acquire)` **like** `.load(||RW)`

C++ IRIW under `memory_order_seq_cst`

```
A.store(27,  
        memory_order_seq_cst);
```

```
B.store(35,  
        memory_order_seq_cst);
```

```
while (A.load(memory_order_seq_cst) != 27);  
y = B.load(memory_order_seq_cst);
```

```
while (B.load(memory_order_seq_cst) != 35);  
a = A.load(memory_order_seq_cst);
```

Under *SC*, either $y=35$, or $a = 27$ (or both), since the stores are totally ordered

C++ IRIW under `memory_order_acq_rel`

```
A.store(27,  
        memory_order_release);
```

```
B.store(35,  
        memory_order_release);
```

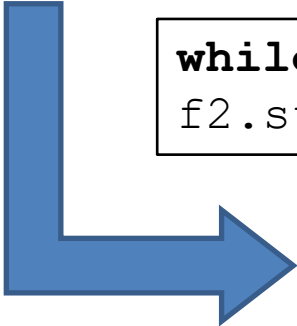
```
while (A.load(memory_order_acquire) != 27);  
y = B.load(memory_order_acquire);
```

```
while (B.load(memory_order_acquire) != 35);  
a = A.load(memory_order_acquire);
```

Possible that `y==0` and `a==0`, because no happens-before between stores to A and B. **Acquire-release does not establish a total order, and does not guarantee atomicity**

Acquire-release guarantees causality

```
A.store(27,memory_order_release);  
f1.store(true,memory_order_release);
```



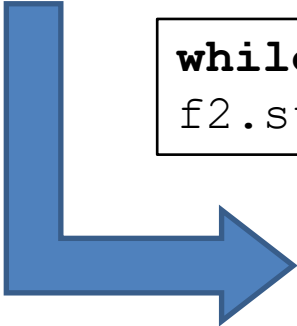
```
while (!f1.load(memory_order_acquire));  
f2.store(true,memory_order_release);
```

```
while (!f2.load(memory_order_acquire));  
a = A.load(memory_order_acquire);
```

Store to A happens-before load from A, since synchronizes with is transitive

Acquire-release guarantees causality (also under relaxed store/load)

```
A.store(27, memory_order_relaxed);  
f1.store(true, memory_order_release);
```



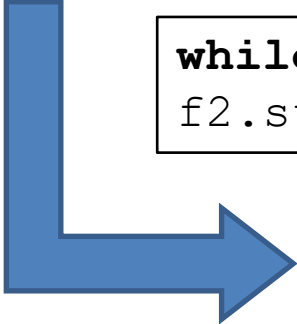
```
while (!f1.load(memory_order_acquire));  
f2.store(true, memory_order_release);
```

```
while (!f2.load(memory_order_acquire));  
a = A.load(true, memory_order_relaxed);
```

Store to A happens-before load from A, since "synchronizes with" is transitive

Acquire-release guarantees causality (also under relaxed store/load)

```
A = 27;  
f1.store(true, memory_order_release);
```



```
while (!f1.load(memory_order_acquire));  
f2.store(true, memory_order_release);
```

```
while (!f2.load(memory_order_acquire));  
a = A;
```

Store to A happens-before load from A, since "synchronizes with" is transitive

```

#include <atomic>
#include <thread>
std::atomic<bool> f;

...
a = data;
atomic_thread_fence(memory_order_release);
f.store(true, memory_order_relaxed);
...

```

The two
fences
synchronizes
-with each
other, and
establish
happens-
before on a

```

#include <atomic>
#include <thread>
std::atomic<bool> f;

...
while (!f.load(memory_order_relaxed));
atomic_thread_fence(memory_order_acquire);
b = a;
...

```

CLH lock in C++11

```
struct Node {  
    Node *pred;  
    atomic<bool> locked;  
};
```

```
void lock() {  
    node = new Node();  
    node->locked.store(true, memory_order_seq_cst);  
    node->pred = tail.exchange(node, memory_order_seq_cst);  
    while (node->pred->locked.load(memory_order_seq_cst));  
}  
  
void unlock() {  
    delete node->pred;  
    node->locked.store(false, memory_order_seq_cst);  
}
```

SC is the default, can be left out



First relaxation: Release for writes, acquire for reads

```
struct Node {  
    Node *pred;  
    atomic<bool> locked;  
};  
  
void lock() {  
    node = new Node();  
    node->locked.store(true, memory_order_release);  
    node->pred = tail.exchange(node, memory_order_acq_rel);  
    while (node->pred->locked.load(memory_order_acquire);  
}  
  
void unlock() {  
    delete node->pred;  
    node->locked.store(false, memory_order_release);  
}
```


Second relaxation: Relax first write (will be released by second)

```
struct Node {  
    Node *pred;  
    atomic<bool> locked;  
};  
  
void lock() {  
    node = new Node();  
    node->locked.store(true, memory_order_relaxed);  
    node->pred = tail.exchange(node, memory_order_acq_rel);  
    while (node->pred->locked.load(memory_order_acquire);  
    }  
  
void unlock() {  
    delete node->pred;  
    node->locked.store(false, memory_order_release);  
}
```

Further reading

Sarita V. Adve, Hans-Juergen Boehm: Memory models: a case for rethinking parallel languages and hardware. *Commun. ACM* 53(8): 90-101 (2010)

Hans-Juergen Boehm, Sarita V. Adve: You don't know jack about shared variables or memory models. *Commun. ACM* 55(2): 48-54 (2012)

Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon-Pharabod, Peter Sewell: The Problem of Programming Language Concurrency Semantics. *ESOP 2015*: 283-307

Hans-Juergen Boehm, Sarita V. Adve: Foundations of the C++ concurrency memory model. *PLDI 2008*: 68-78

Java

Adapts SC for DRF. **But** Java also needs to provide semantics for programs that are not DRF:

- safe and secure language
- must be able to run untrusted code

Arbitrary behavior is not acceptable

New memory model with Java 5 (earlier models were broken)

```
Singleton get() {  
    if (singleton==null) {  
        synchronized (this) {  
            if (singleton==null)  
                singleton = new Singleton();  
        }  
    }  
    return singleton;  
}
```

Double checked locking (**idea**: Prevent expensive lock if singleton already initialized), **broken** in old Java memory model: **Pointer might be set before constructor has been executed**

OpenMP, pthreads

Idea (similar to MPI one-sided communication):
Conceptually separated local and global views on memory, threads operate on local view, OpenMP constructs propagate memory updates to global view

Greg Bronevetsky, Bronis R. de Supinski: Complete Formal Specification of the OpenMP Memory Model. International Journal of Parallel Programming 35(4): 335-392 (2007)
Jay P. Hoeflinger, Bronis R. de Supinski: The OpenMP Memory Model. IWOMP 2005: 167-177

MPI one-sided communication model

Conceptually, local and global views of memory; global view visible to other threads, closing and opening of epoch determine when local views become global, and global view locally visible; additional synchronization (fence) constructs

Torsten Hoefler, James Dinan, Rajeev Thakur, Brian Barrett, Pavan Balaji, William Gropp, Keith D. Underwood:
Remote Memory Access Programming in MPI-3. TOPC 2(2): 9:1-9:26 (2015)

Further reading (and there's lots more):

Daniel J. Sorin, Mark. D. Hill, David A. Wood: A primer on memory consistency and cache coherence. Synthesis Lectures on Computer Architecture. Morgan&Claypool, 2011

Michael L. Scott: Shared-memory synchronization. Synthesis Lectures on Computer Architecture. Morgan&Claypool, 2013

Anthony Williams: C++ Concurrency in Action. Manning, 2012

S. V. Adve, K. Gharachorloo: Shared Memory Consistency Models: A Tutorial. IEEE Computer, 29(12):66-76, 1996

Bjarne Stroustrup: The C++ Programming Language. 4th Ed., Addison-Wesley, 2013

The concurrent memory reclamation problem

In non-garbage collected languages (C, C++)

```
free(p) ;
```



gives node pointed to by p back to memory manager

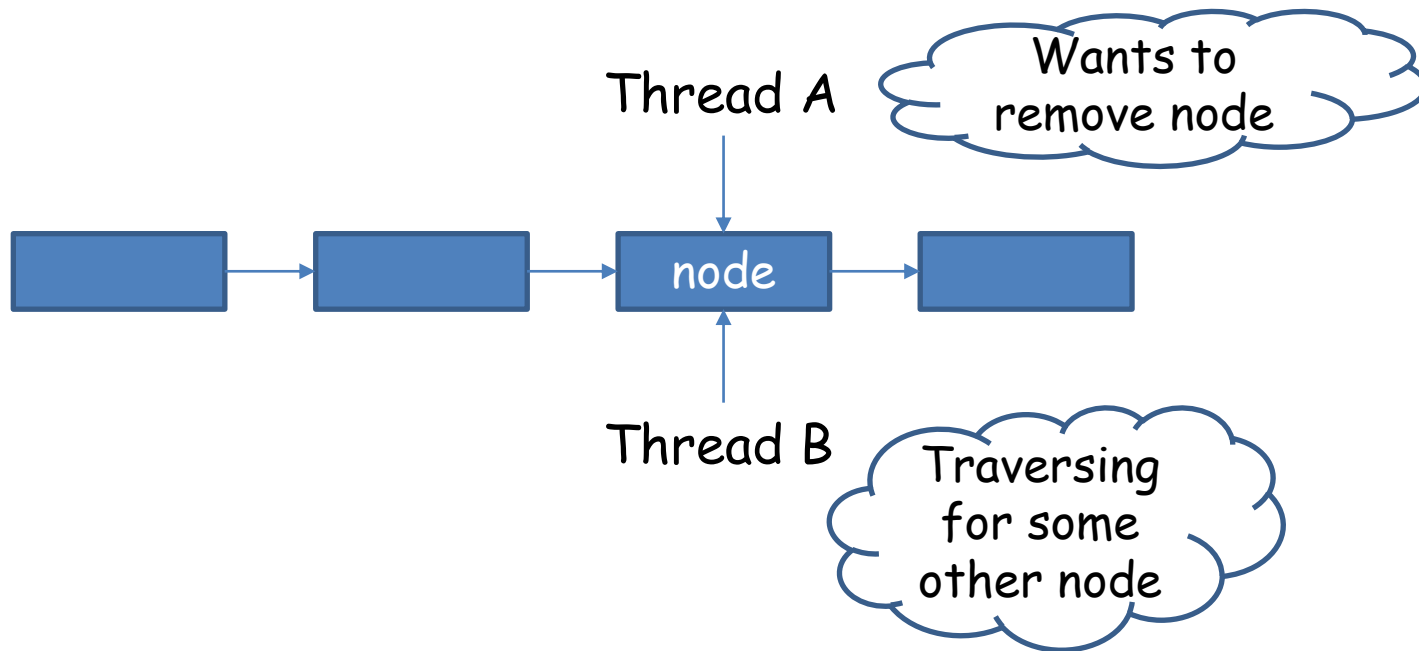
With **only one** thread:

(mostly; if not, use reference counting) easy to know that there is only one reference (**beware**: circular references!), node can **safely** be given back, since it will never be referenced again

With **more than one** thread:

Safe to give node back only when no threads are referencing.

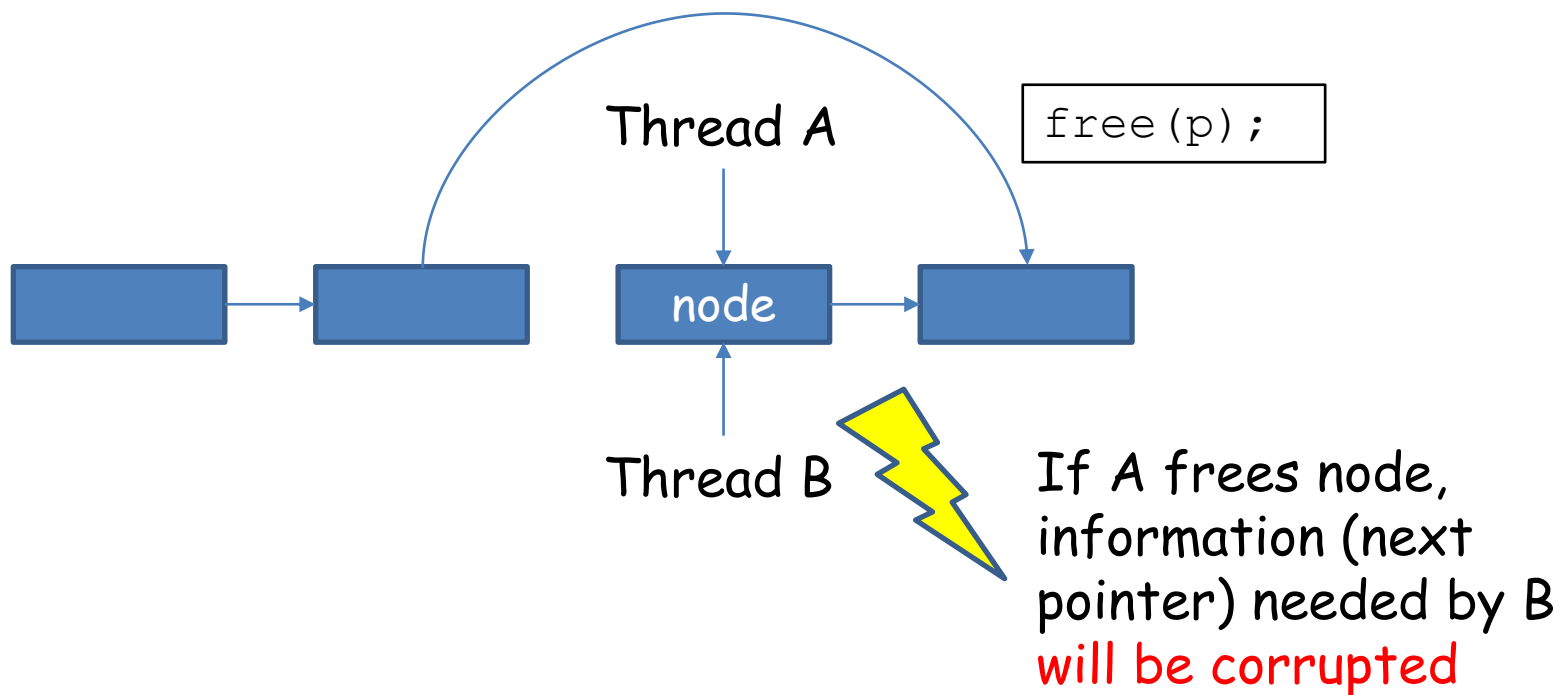
How can free'ing thread know that no other threads reference the node?



With **more than one** thread:

Safe to give node back only when no threads are referencing.

How can free'ing thread know that no other threads reference the node?



Memory reclamation scheme wish list:

- **General**, should work for any data structure, no constraints on structure of allocated nodes, no special coding discipline
- **Portable**, should require no OS support, no not commonly supported hardware instructions (DCAS etc.)
- **Thread oblivious**, no fixed upper bound on number of threads
- **Strong progress guarantees** (wait-free, lock-free, **not reclamation blocking**)
- **Efficient**, constant-time overhead per reclaimed node
- ...

Reclamation blocking: A stalled (crashed) thread can prevent reclamation of an unbounded amount of memory

Lock-Free Reference Counting (LFRC)

Memory allocated and free'd as "nodes".

Add a reference count to each node. When a thread references a node, the reference count is incremented, when a thread drops the reference, the reference count is decremented. When a reference count drops to 0, this node may be reclaimed, reclaiming thread sets "reclaim" bit.

Drawback of reference counting:

Nodes in cyclic data structures may never be freed (memory is leaked)

LFRC

- Get reference: FAA on reference count, validate reference with CAS (on failure, decrease reference count, drop reference, retry)
- Drop reference: CAS on reference count, if 0 reclaim (by setting reclaim bit with the CAS)

Because getting a reference involves two steps, LFRC can only be used when nodes are being reused in the same data structure, the reference count must be available indefinitely

LFRC: Not general, portable, thread oblivious, reclamation blocking, not efficient

John D. Valois: Lock-free data structures. PhD thesis, Rensselaer Polytechnic Institute, 1995

Maged M. Michael, Michael L. Scott: Correction of a memory management method for lock-free data structures. TR, University of Rochester, 1995

Håkan Sundell: Wait-Free Reference Counting and Memory Management. IPDPS 2005

Anders Gidenstam, Marina Papatriantafilou, Håkan Sundell, Philippas Tsigas: Efficient and Reliable Lock-Free Memory Reclamation Based on Reference Counting. IEEE Trans. Parallel Distrib. Syst. 20(8): 1173-1187 (2009)

Hazard Pointers (HP)

Sensitive nodes are referenced through special pointers: Hazard pointers.

Each thread has some fixed number K of hazard pointers (total number of hazard pointers is pK when p threads are running).

The hazard pointers is the way for a thread to communicate to other threads that it is holding a reference to a shared node.

When a thread wants to reclaim a node, it must scan the hazard pointers of all other threads to find out whether there are other references to the node

HP: Not general, portable, not thread oblivious, not reclamation blocking, not efficient. And **patented**(!)

Maged M. Michael: Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. IEEE Trans. Parallel Distrib. Syst. 15(6): 491-504 (2004)

Stamp-it: A new, epoch-based reclamation scheme

- **General**: Little special coding discipline required
- **Portable**: Uses FAA, CAS, EX, no OS support
- **Thread oblivious**
- **Good progress**: lock-less (but **reclamation blocking**)
- **Amortized constant-time reclamation** per node

Design goal: Avoid inspection of all threads on reclamation

Ingredients:

- Local retire list of nodes to be reclaimed
- Global retire list
- Stamp Pool of thread elements

Epoch-based schemes (ER, NER, IBR, ...)*:

- Programmer has to mark **critical regions** (**not to be confused** with **critical sections**...) where nodes may be shared. Can be supported with C++ interface (not yet in standard)
- Scheme tracks entry to and exit from critical regions

Stamp-it key idea:

Stamp Pool data structure to order entry to critical regions

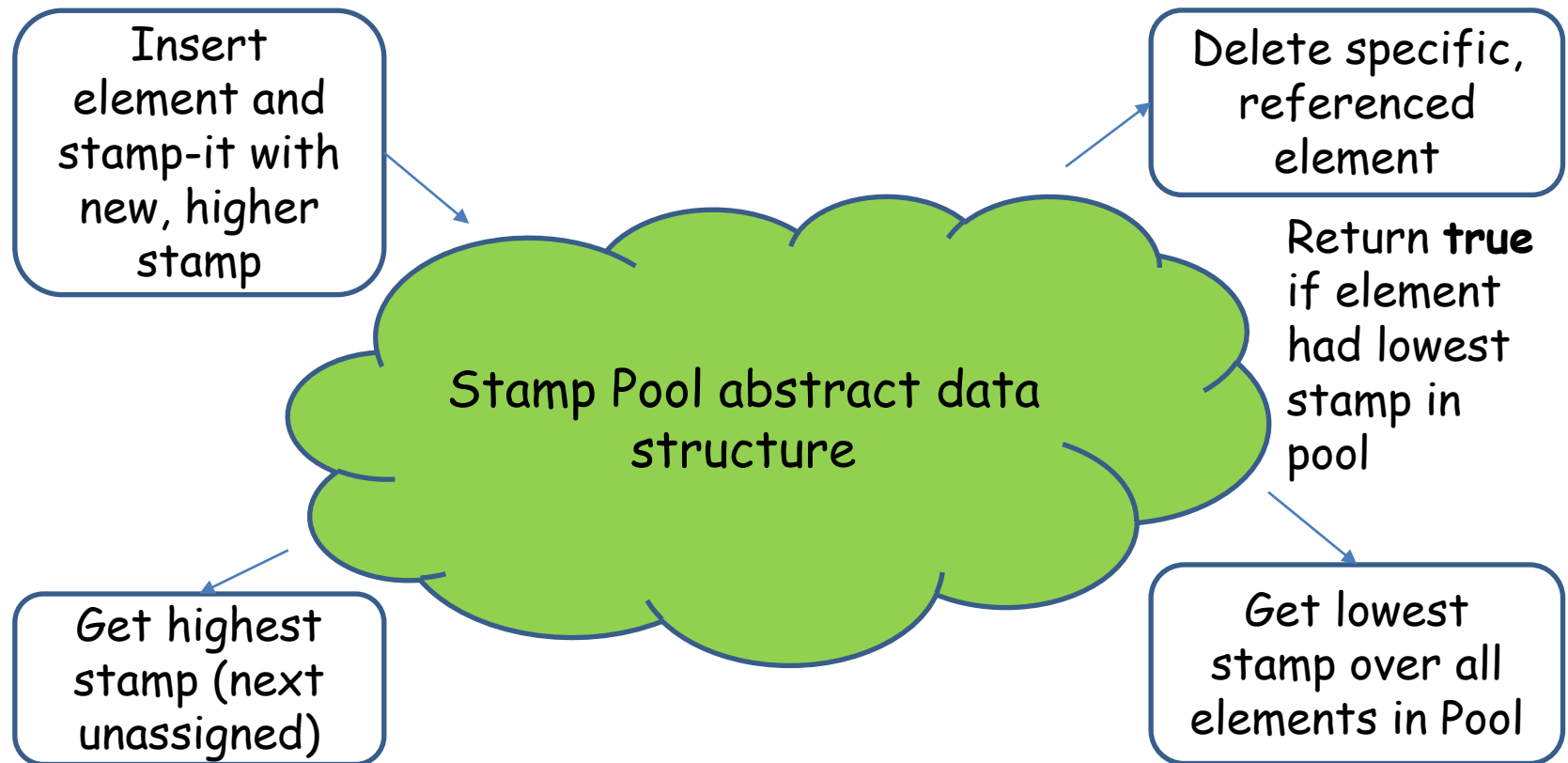
*

Keir Fraser: Practical lock-freedom. University of Cambridge, UK
2004

Thomas E. Hart, Paul E. McKenney, Angela Demke Brown,
Jonathan Walpole: Performance of memory reclamation for
lockless synchronization. J. Parallel Distrib. Comput. 67(12):
1270-1285 (2007)

Stamp Pool:

Maintains time-stamped elements (one per thread), stamps monotonically increasing



Stamp Pool:

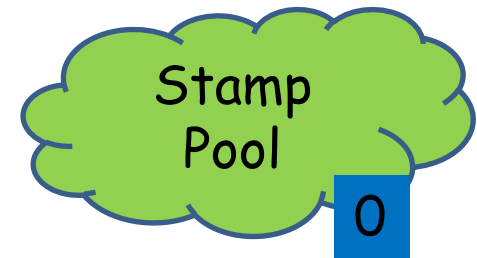
Maintains time-stamped elements (one per thread), stamps monotonically increasing

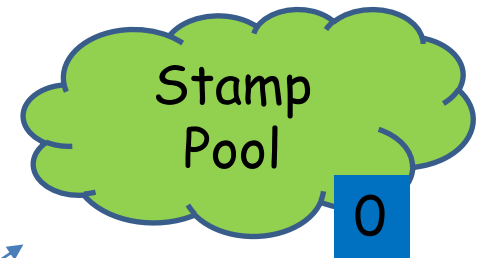
Implementation:

Lock-free doubly linked list (CAS), based on Sundell/Tsigas, $O(1)$ operations in absence of conflicts. New stamp by FAA

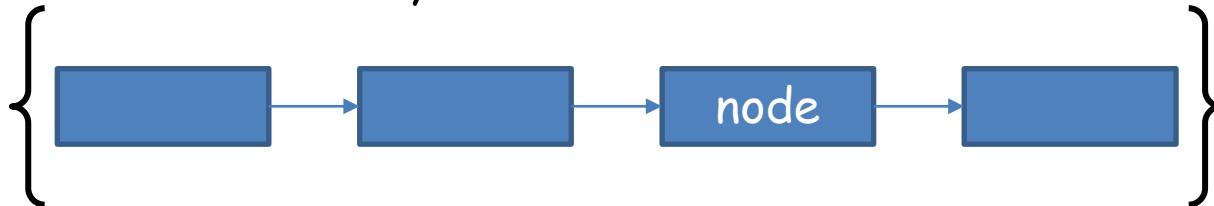
Correct in C++ memory model (acquire-release)

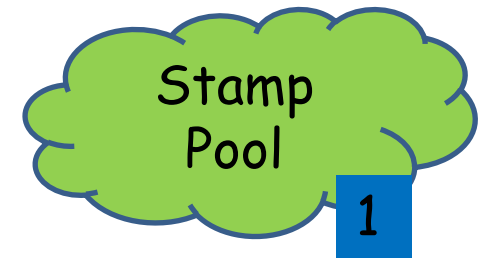
Håkan Sundell, Philippos Tsigas: Lock-free dequeues and doubly linked lists. J. Parallel Distrib. Comput. 68(7): 1008-1020 (2008)



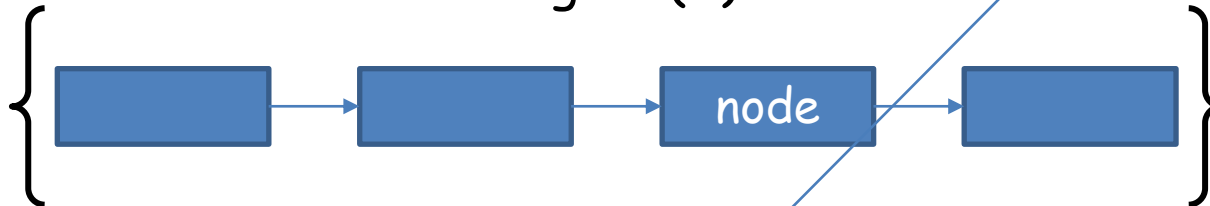


Thread A enters, Pool insert

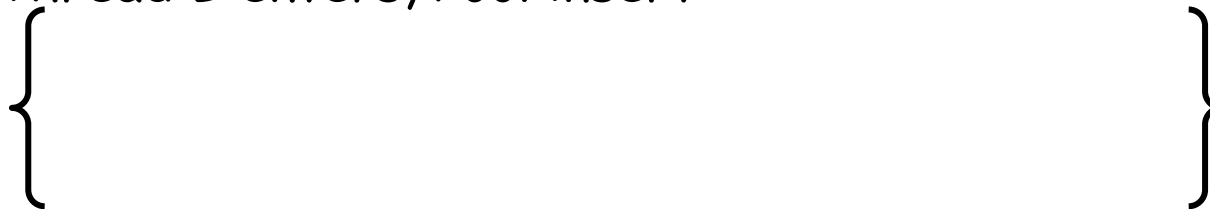


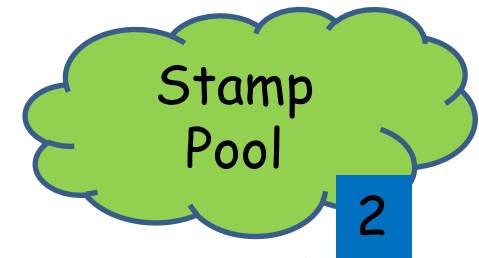


Thread A in critical region (0)



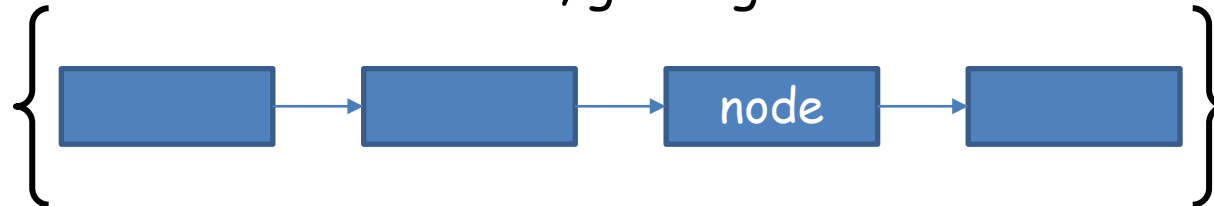
Thread B enters, Pool insert





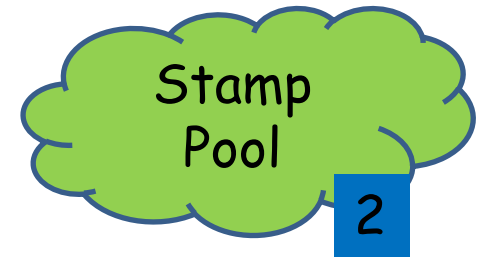
Thread A retires node, get highest

retire

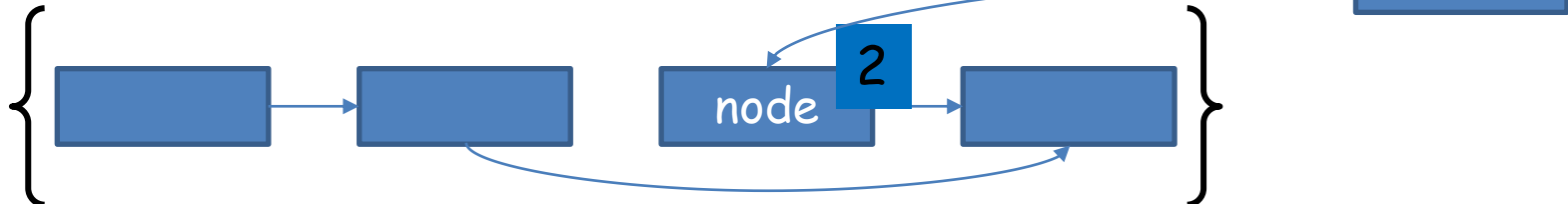


Thread B in critical region (1)

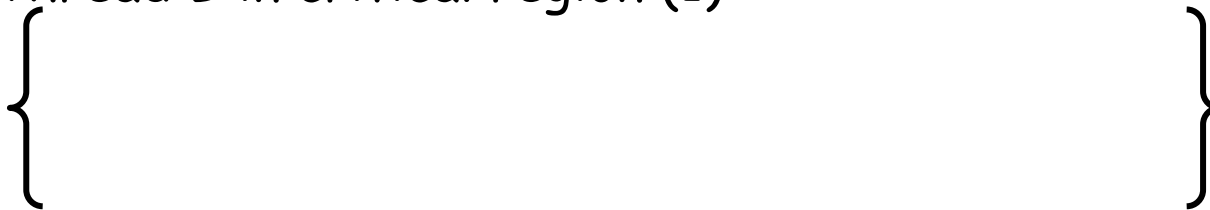


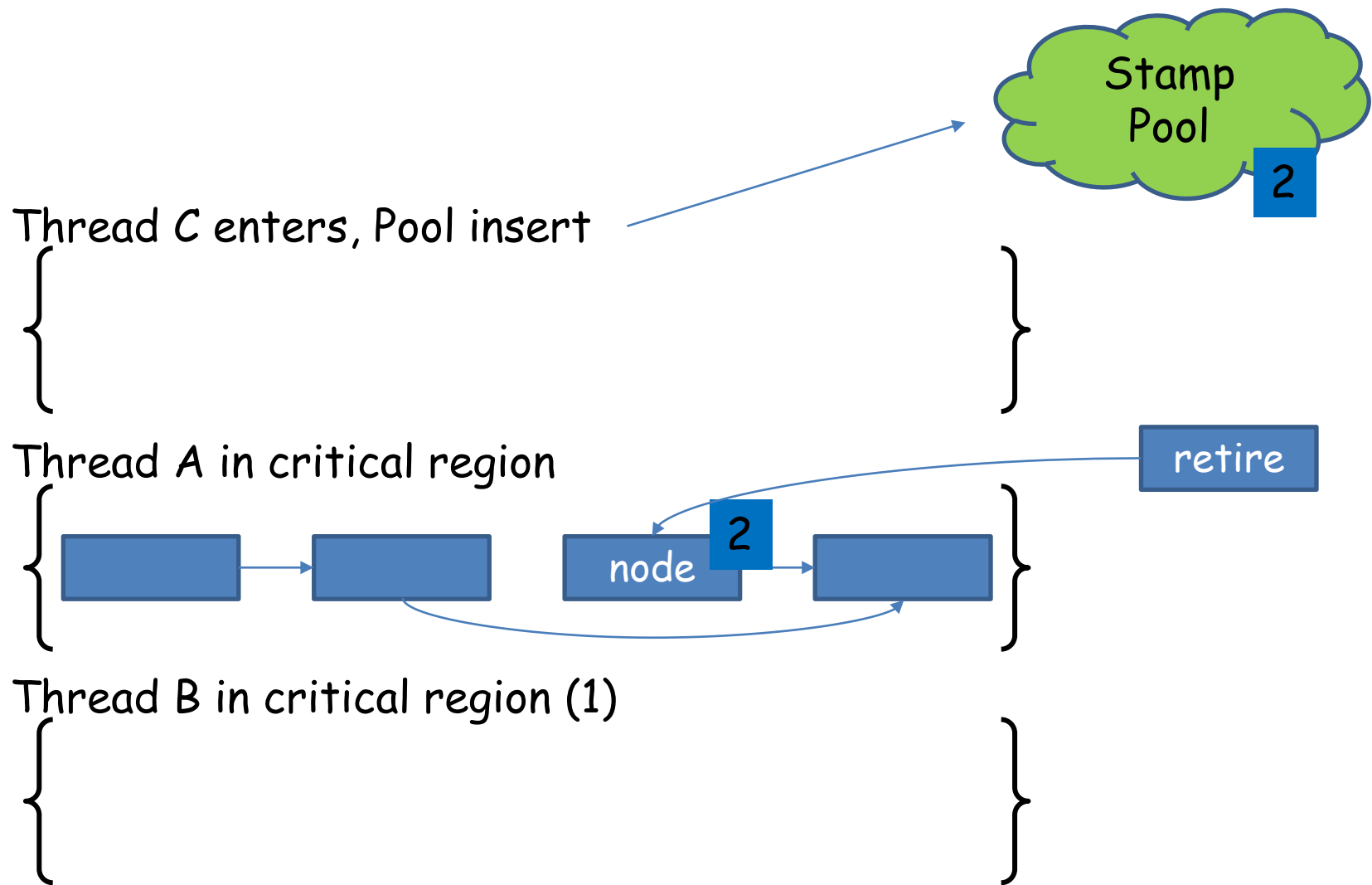


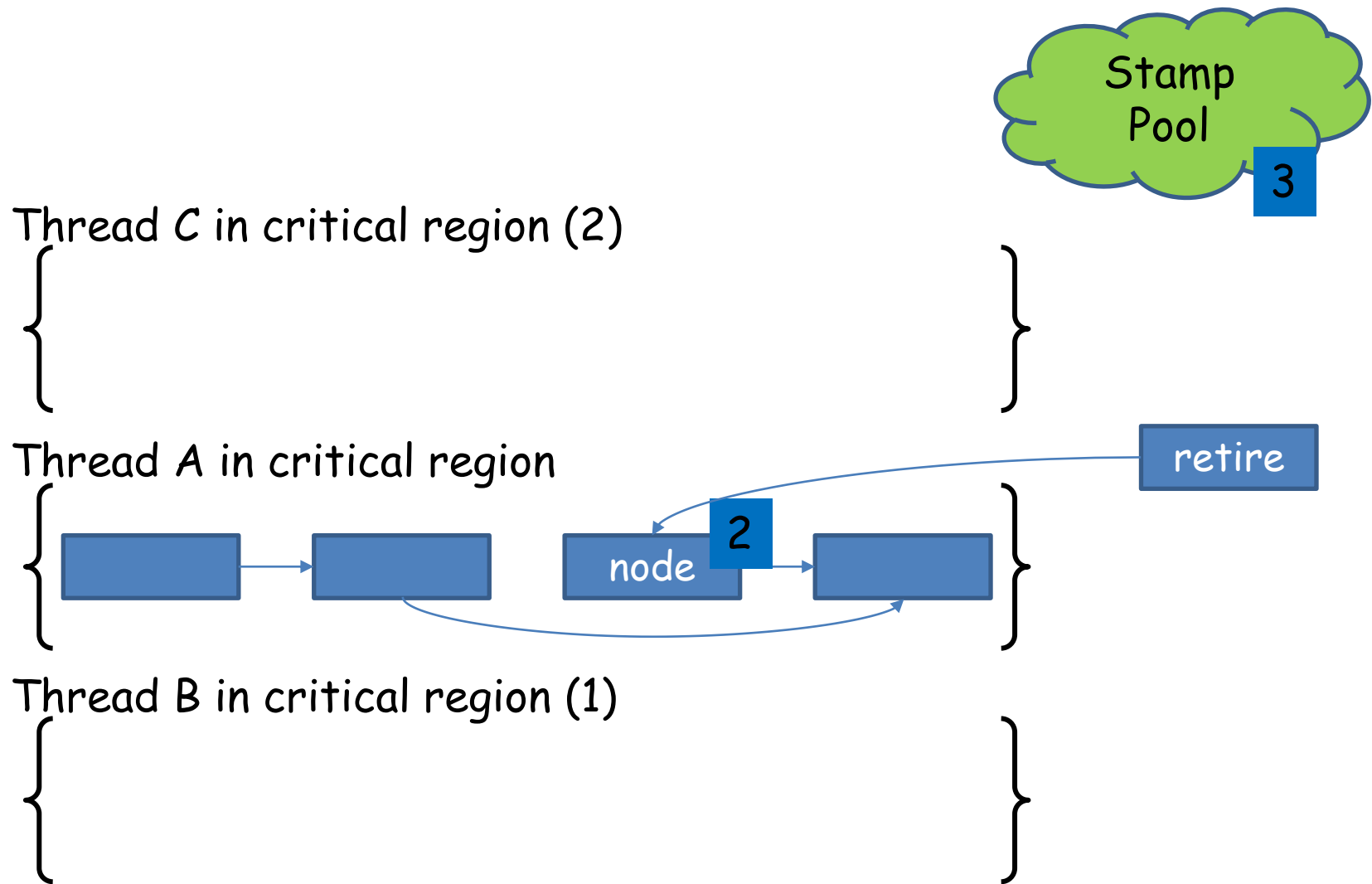
Thread A retires node

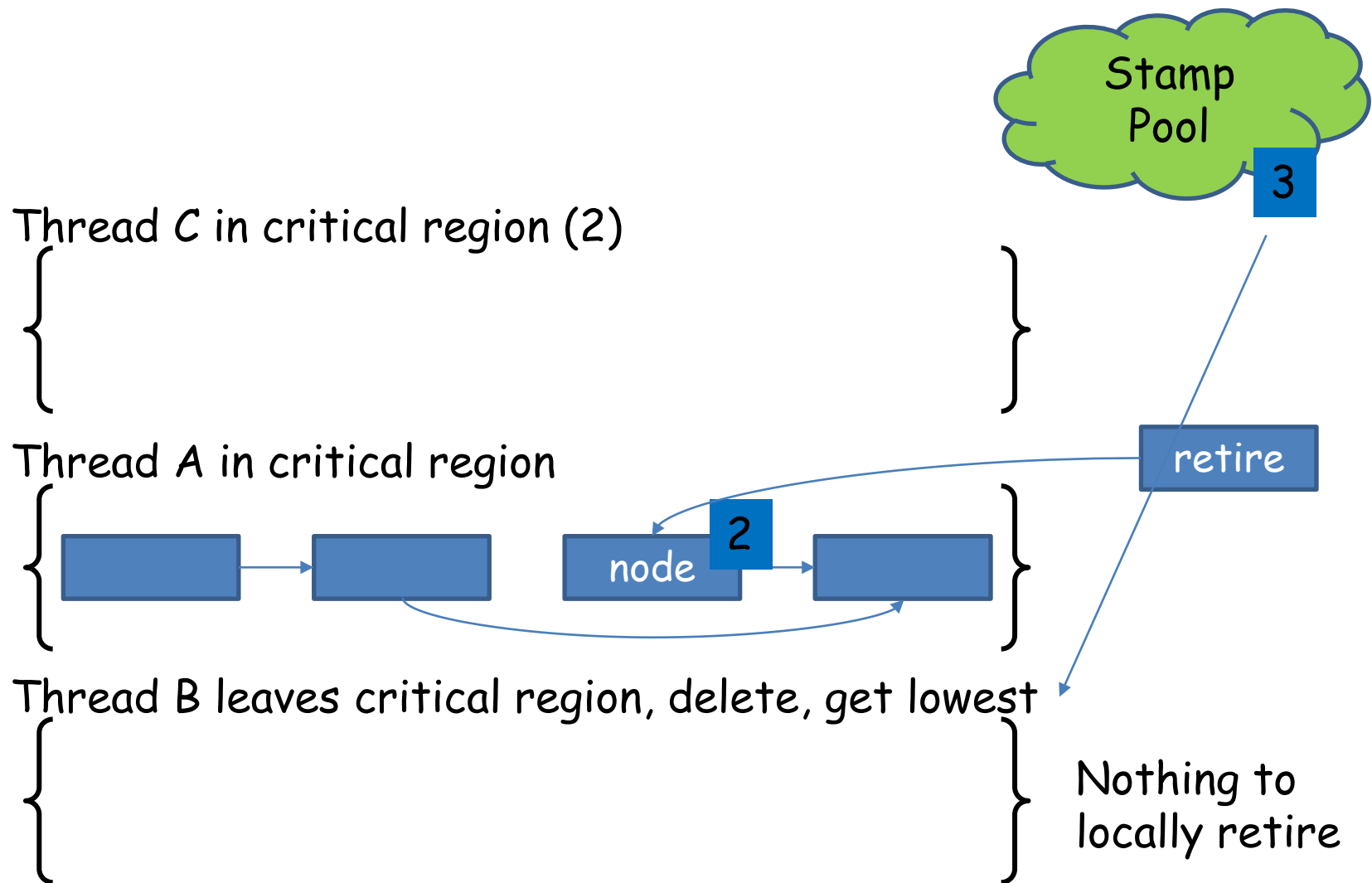


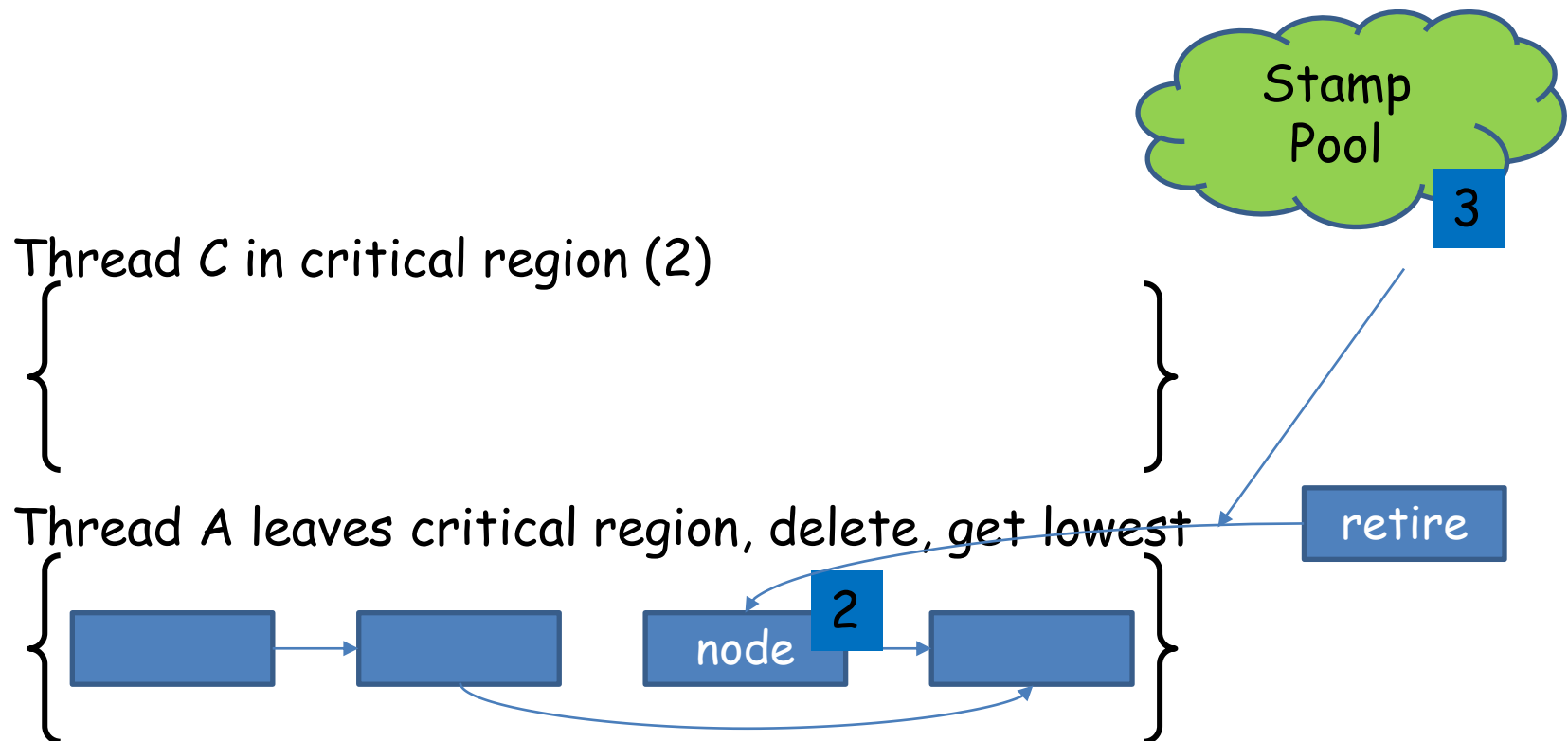
Thread B in critical region (1)



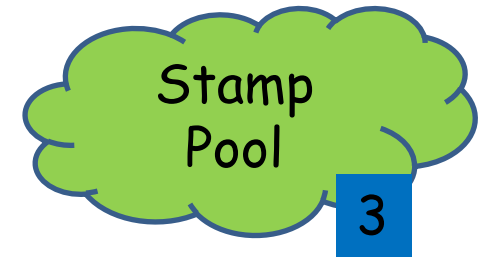








Scan local retire list, all retired nodes with stamp smaller than lowest can be freed



Thread C in critical region (2)

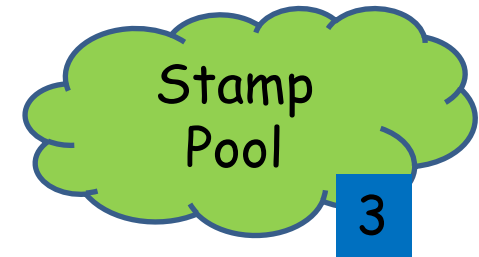


Thread A leaves critical region



retire

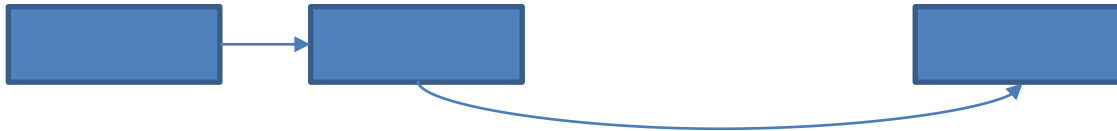
Scan local retire list, all retired nodes with stamp smaller than lowest can be freed

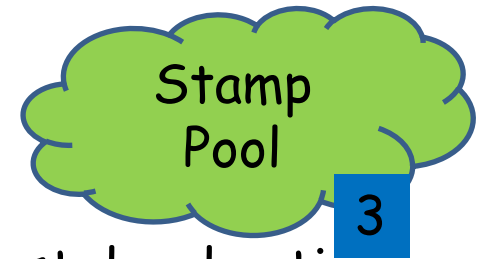


Thread C leaves critical region



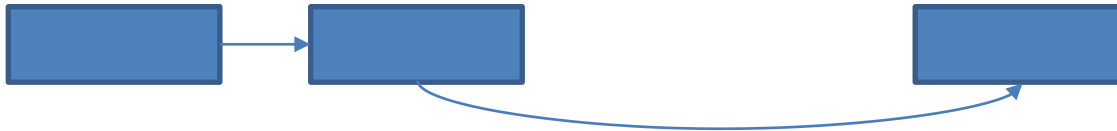
retire

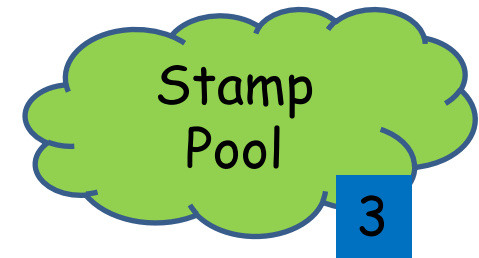




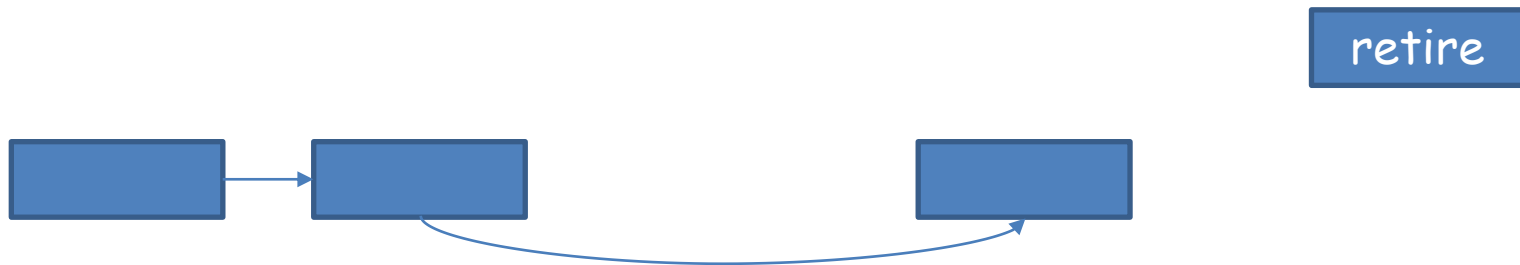
Thread C leaves critical region, delete, get lowest, local retire

retire





If local retire list gets too large, move to global retire list



Properties:

- Nodes in local retire lists are in increasing stamp order. Reclaiming all nodes with stamp less than currently lowest stamp in $O(1)$ per reclaimed node
- Global retire list is list of ordered sublists. Carefully chosen threshold for amortization argument

Retire list operations:

- Append local retire list to global retire list: CAS
- Reclaim from global retire list: Last thread (remove=**true**) grabs list with EX

Claim 1:

Stamp-it is reclamation safe (correct). A node is reclaimed only when it is referenced by no thread

Claim 2:

Stamp-it reclaims any node in amortized constant time in the number of data structure (Stamp Pool) operations

Claim 3:

Stamp-it is lock-less. All Stamp Pool operations and other operations for entering/leaving critical regions are lock-free. If threads do not conflict, all operations are $O(1)$

Claim 4:

The Stamp-it implementation is correct under the C++ memory model

Practically efficient:

Comparison to 6 other general schemes

- LFRC, HP, ER, NER, QSR, DEBRA

on 4 different architectures with quite a lot of hardware threads (48-core AMP Opteron, 160-thread Intel Xeon, 244-thread Intel Xeon Phi, 512-thread SPARC)

Different benchmarks, various criteria (speed, reclamation efficiency)

Stamp-it performs on par with, or (considerably) better in almost all cases

Details:

Manuel Pöter, Jesper Larsson Träff: Brief Announcement: Stamp-it, a more Thread-efficient, Concurrent Memory Reclamation Scheme in the C++ Memory Model. SPAA 2018: 355-358

Manuel Pöter, Jesper Larsson Träff: Stamp-it, amortized constant-time memory reclamation in comparison to five other schemes. PPOPP 2018: 413-414

Manuel Pöter, Jesper Larsson Träff: Stamp-It: A more Thread-efficient, Concurrent Memory Reclamation Scheme in the C++ Memory Model. CoRR abs/1805.08639 (2018)

Manuel Pöter: Effective Memory Reclamation for Lock-free Data Structures in C++. Master's thesis, TU Wien (2018)

Full code and benchmarks: <https://github.com/mpoeter/emr>