Weak memory models

November 23, 2017

Overview

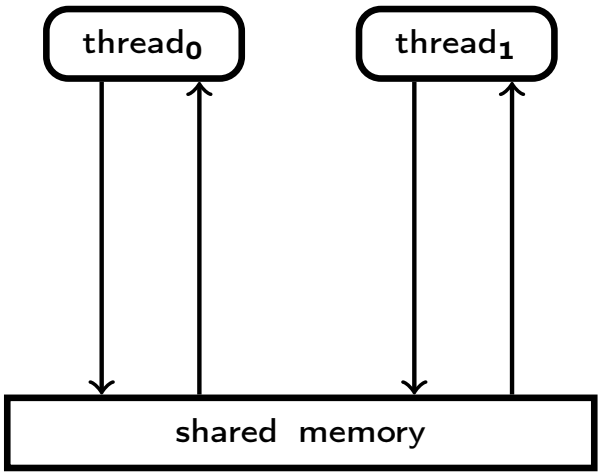
Introduction

Concurrency

"Concurrency is a property of systems in which several computations are executing simultaneously, and potentially interacting with each other" (Wikipedia)

- performance increase, better latency
- many forms of concurrency/parallelism: multi-core, multi-threading, multi-processors, distributed systems ...

Shared memory: a simplistic picture



- one way of "interacting" (i.e., communicating and synchronizing): via shared memory
- a number of threads/processors: access common memory/address space
- interacting by sequence of reads/writes (or loads/stores, etc.)

However: considerably *harder* to get correct and efficient programs

Dekker's solution to mutex

- As known, shared memory programming requires synchronization: e.g. mutual exclusion

Dekker

- simple and first known mutex algo
- here simplified

initially: $\texttt{flag}_0 = \texttt{flag}_1 = 0$

$\texttt{flag}_0$:= 1;	$\texttt{flag}_1$:= 1;
if ($\texttt{flag}_1 = 0$)	**if** ($\texttt{flag}_0 = 0$)
then CRITICAL	**then** CRITICAL

Dekker's solution to mutex

- As known, shared memory programming requires synchronization: e.g. mutual exclusion

Dekker

- simple and first known mutex algo
- here simplified

initially: $\texttt{flag}_0 = \texttt{flag}_1 = 0$

$\texttt{flag}_0$:= 1;	$\texttt{flag}_1$:= 1;
if ($\texttt{flag}_1$ = 0)	**if** ($\texttt{flag}_0$ = 0)
then CRITICAL	**then** CRITICAL

Known textbook "fact":

Dekker is a software-based solution to the mutex problem (or is it?)

A three process example

Initially: `x,y = 0`, `r`: register, local var

thread_0	thread_1	thread_2
`x := 1`	`while (x = 0) do skip;`	`while (y = 1) do skip`
	`y:=1`	`r:=x`

"Expected" result

Upon termination, register r of the third thread will contain $r = 1$.

A three process example

Initially: `x,y = 0`, `r`: register, local var

thread_0	thread_1	thread_2
`x := 1`	`while (x = 0) do skip;`	`while (y = 1) do skip`
	`y:=1`	`r:=x`

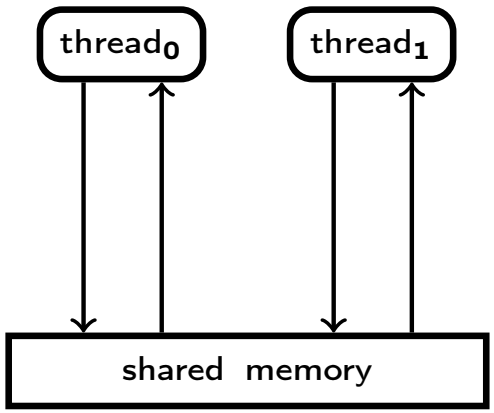
"Expected" result

Upon termination, register r of the third thread will contain $r = 1$.

But:

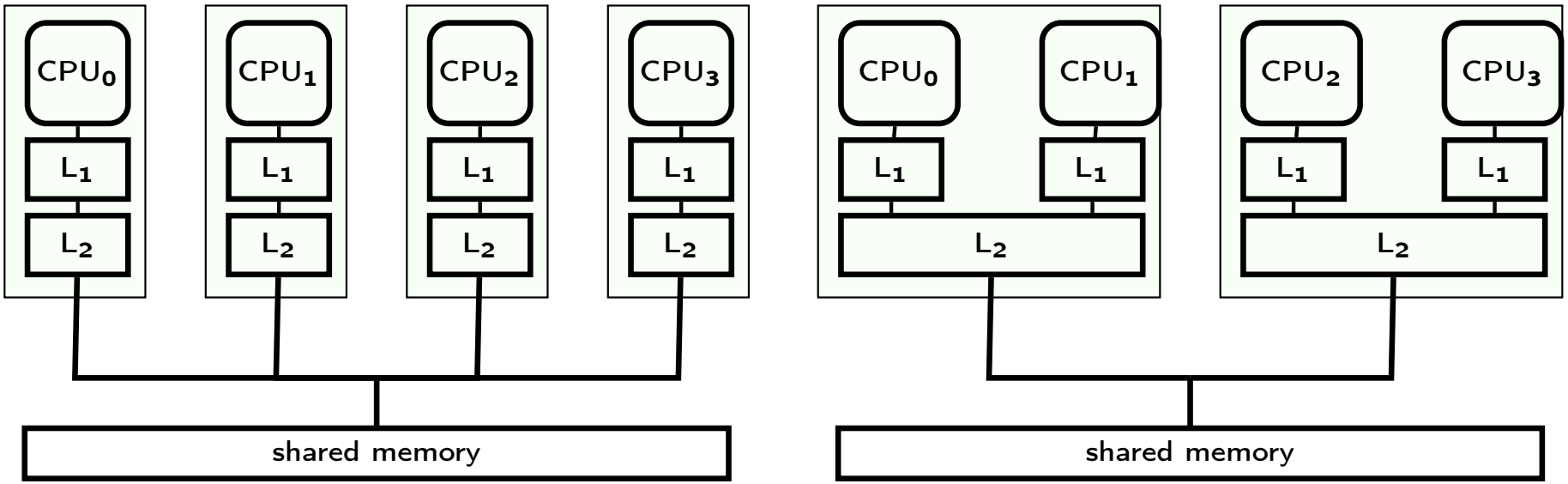
Who ever said that there is only one identical copy of x that thread_1 and thread_2 operate on?

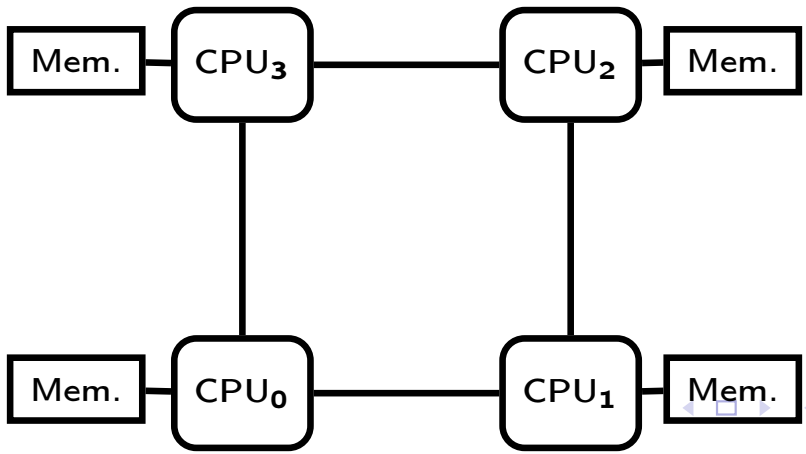
Shared memory concurrency in the real world



- the memory architecture does not reflect reality
- out-of-order executions: 2 interdependent reasons:
  1. modern HW: complex memory hierarchies, caches, buffers ...
  2. compiler optimizations

SMP, multi-core architecture, and NUMA



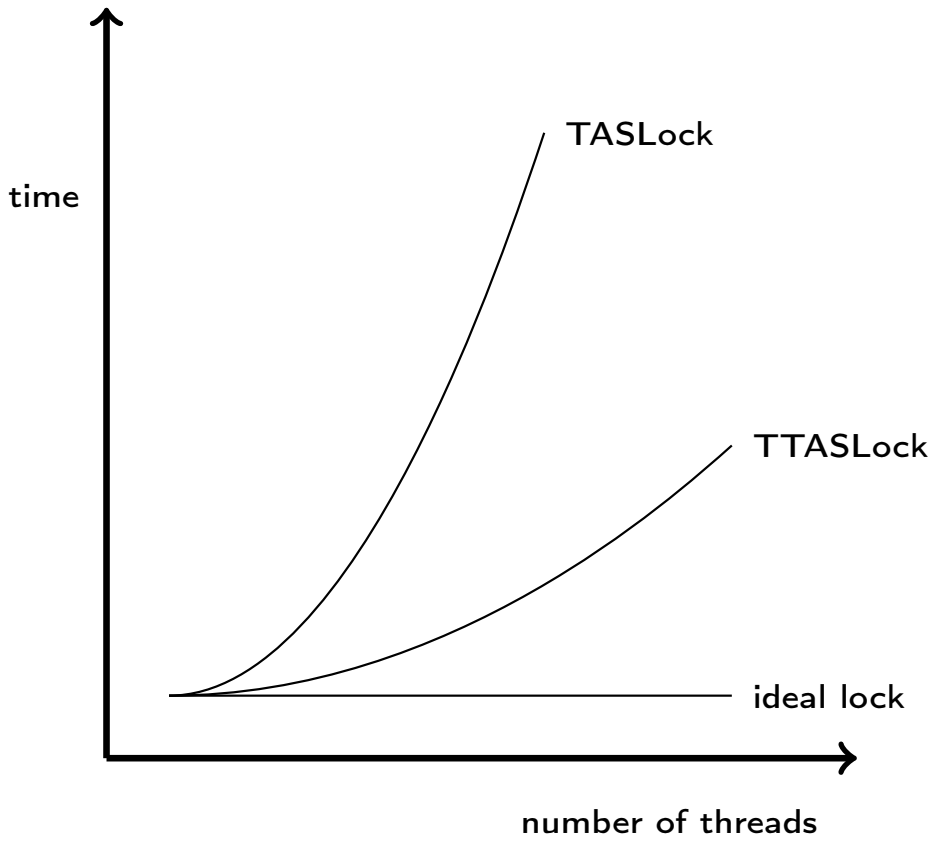


"Modern" HW architectures and performance

```
public class TASLock implements Lock {
    ...
    public void lock() {
        while (state.getAndSet(true)) { }  // spin
    }
    ...
}
```

```
public class TTASLock implements Lock {
    ...
    public void lock() {
        while (true) {
            while (state.get()) {}; //spin
            if (!state.getAndSet(true))
                return;
        }
        ...
    }
}
```

Observed behavior



(cf. [Anderson, 1990] [Herlihy and Shavit, 2008, p.470])

Compiler optimizations

- many optimizations with different forms:
  - **elimination** of reads, writes, sometimes synchronization statements
  - **re-ordering** of independent, non-conflicting memory accesses
  - **introductions** of reads
- examples
  - constant propagation
  - common sub-expression elimination
  - dead-code elimination
  - loop-optimizations
  - call-inlining
  - ... and many more

Code reodering

Initially: `x = y = 0`

thread_0	thread_1
`x := 1`	`y:= 1;`
r_1 `:= y`	r_2 `:= x;`
`print` r_1	`print` r_2

possible print-outs
$\{(0,1),(1,0),(1,1)\}$

$\Longrightarrow$

Initially: `x = y = 0`

thread_0	thread_1
r_1 `:= y`	`y:= 1;`
`x := 1`	r_2 `:= x;`
`print` r_1	`print` r_2

possible print-outs
$\{(0,0),(0,1),(1,0),(1,1)\}$

Common subexpression elimination

Initially: `x = 0`

$thread_0$	$thread_1$

`x := 1`	r_1 `:= x;`
	r_2 `:= x;`
	`if` $r_1 = r_2$
	`then print 1`
	`else print 2`

$\Longrightarrow$

Initially: `x = 0`

$thread_0$	$thread_1$

`x := 1`	r_1 `:= x;`
	r_2 `:=` r_1`;`
	`if` $r_1 = r_2$
	`then print 1`
	`else print 2`

Is the transformation from the left to the right correct?

Common subexpression elimination

Initially: `x = 0`

$thread_0$	$thread_1$

`x := 1`	`r1 := x;`
	`r2 := x;`
	`if r1 = r2`
	`then print 1`
	`else print 2`

$\Longrightarrow$

Initially: `x = 0`

$thread_0$	$thread_1$

`x := 1`	`r1 := x;`
	`r2 := r1;`
	`if r1 = r2`
	`then print 1`
	`else print 2`

Is the transformation from the left to the right correct?

$thread_0$	$W[x] := 1;$			
$thread_1$		$R[x] = 1;$	$R[x] = 1;$	`print(1)`
$thread_0$		$W[x] := 1;$		
$thread_1$	$R[x] = 0;$		$R[x] = 1;$	`print(2)`
$thread_0$			$W[x] := 1;$	
$thread_1$	$R[x] = 0;$	$R[x] = 0;$		`print(1)`
$thread_0$				$W[x] := 1;$
$thread_1$	$R[x] = 0;$	$R[x] = 0;$	`print(1);`	

2nd prog: only 1 read from memory $\Rightarrow$ only `print(1)` possible

Golden rule of compiler optimization

Change the code (for instance re-order statements, re-group parts of the code, etc) in a way that leads to

- better performance (at least on average), but is otherwise
- unobservable to the programmer (i.e., does not introduce new observable result(s))

Golden rule of compiler optimization

Change the code (for instance re-order statements, re-group parts of the code, etc) in a way that leads to

- better performance (at least on average), but is otherwise
- unobservable to the programmer (i.e., does not introduce new observable result(s)) when executed single-threadedly, i.e. without concurrency! :-O

In the presence of concurrency

- more forms of "interaction"

$\Rightarrow$ more effects become observable

- standard optimizations become observable (i.e., "break" the code, assuming a naive, standard shared memory model)

Is the *Golden Rule* outdated?

Golden rule as task description for compiler optimizers:

- Let's assume for convenience, that there is no concurrency, how can I make make the code faster ....
- and if there's concurrency? too bad, but not my fault ...

Is the *Golden Rule* outdated?

Golden rule as task description for compiler optimizers:

- Let's assume for convenience, that there is no concurrency, how can I make make the code faster ....
- and if there's concurrency? too bad, but not my fault ...

- unfair characterization
- assumes a "naive" interpretation of shared variable concurrency (interleaving semantics, SMM)

Is the *Golden Rule* outdated?

Golden rule as task description for compiler optimizers:

- Let's assume for convenience, that there is no concurrency, how can I make make the code faster . . . .
- and if there's concurrency? too bad, but not my fault . . .

What's needed:

- golden rule must(!) still be upheld
- but: relax naive expectations on what shared memory is

$\Rightarrow$ weak memory model

DRF

golden rule: also core of "data-race free" programming principle

Compilers vs. programmers

Programmer

- wants to understand the code

$\Rightarrow$ profits from strong memory models

$\leftrightsquigarrow$

Compiler/HW

- want to optimize code/execution (re-ordering memory accesses)

$\Rightarrow$ take advantage of weak memory models

$\Longrightarrow$

- What are valid (semantics-preserving) compiler-optimations?
- What is a good memory model as compromise between programmer's needs and chances for optimization

Sad facts and consequences

- incorrect concurrent code, "unexpected" behavior
  - Dekker (and other well-know mutex algo's) is incorrect on modern architectures[1]
  - in the three-processor example: $r = 1$ not guaranteed
- unclear/obstruse/informal hardware specifications, compiler optimizations may not be transparent
- understanding of the memory architecture also crucial for performance

Need for unambiguous description of the behavior of a chosen platform/language under shared memory concurrency $\Longrightarrow$ memory models

[1] Actually already since at least IBM 370.

What's a memory model?

"A formal specification of how the memory system will appear to the programmer, eliminating the gap between the behavior expected by the programmer and the actual behavior supported by a system." [Adve and Gharachorloo, 1995]

MM specifies:

- How threads interact through memory?
- Which values a read can return?
- When does a value update become visible to other threads?
- What assumptions are allowed to make about memory when writing a program or applying some program optimization?

Sequential consistency

- in the previous examples: unspoken assumptions

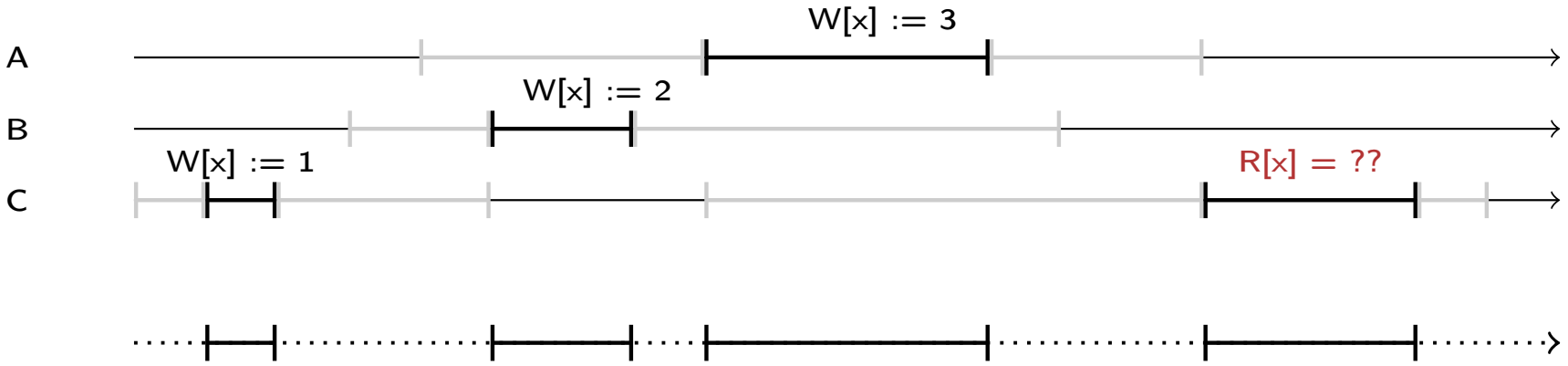
1. Program order: statements executed in the order written/issued (Dekker).
2. atomicity: memory update is visible to everyone at the same time (3-proc-example)

Lamport [Lamport, 1979]: Sequential consistency

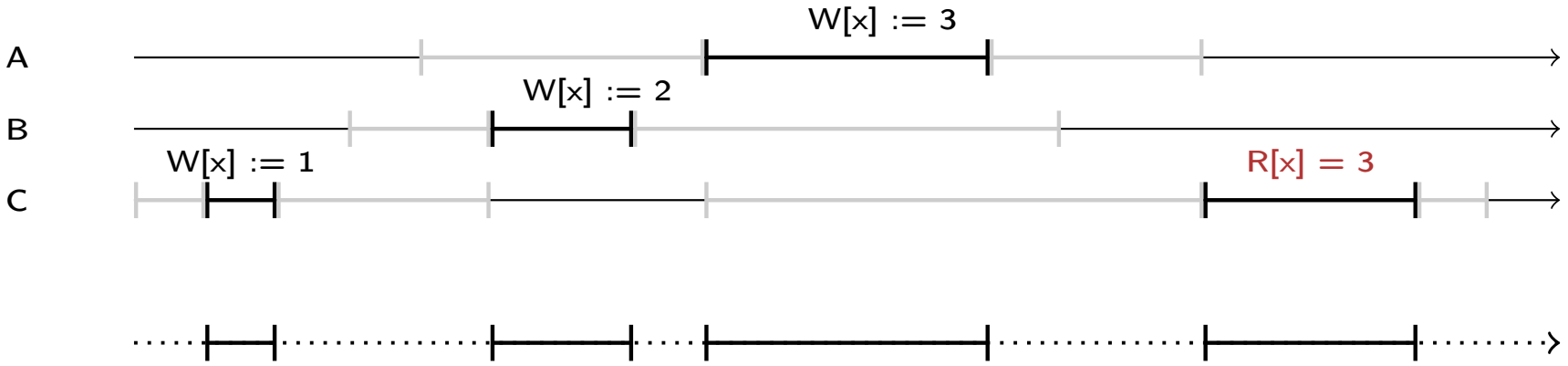
"...the results of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program."

- "classical" model, (one of the) oldest correctness conditions
- simple/simplistic $\Rightarrow$ (comparatively) easy to understand
- straightforward generalization: single $\Rightarrow$ multi-processor
- weak means basically "more relaxed than SC"

Atomicity: no overlap

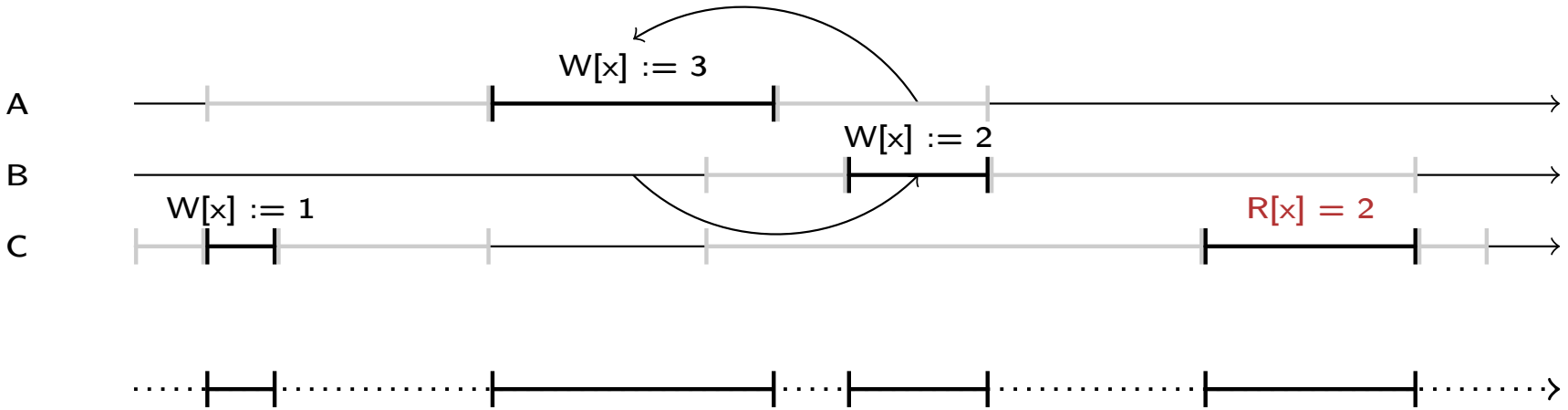


Atomicity: no overlap



Which values for x consistent with SC?

Some order consistent with the observation



- read of 2: observable under sequential consistency (as is 1, and 3)
- read of 0: contradicts program order for thread C.

Weak memory models

Spectrum of available architectures

(from `http://preshing.com/20120930/weak-vs-strong-memory-models`)

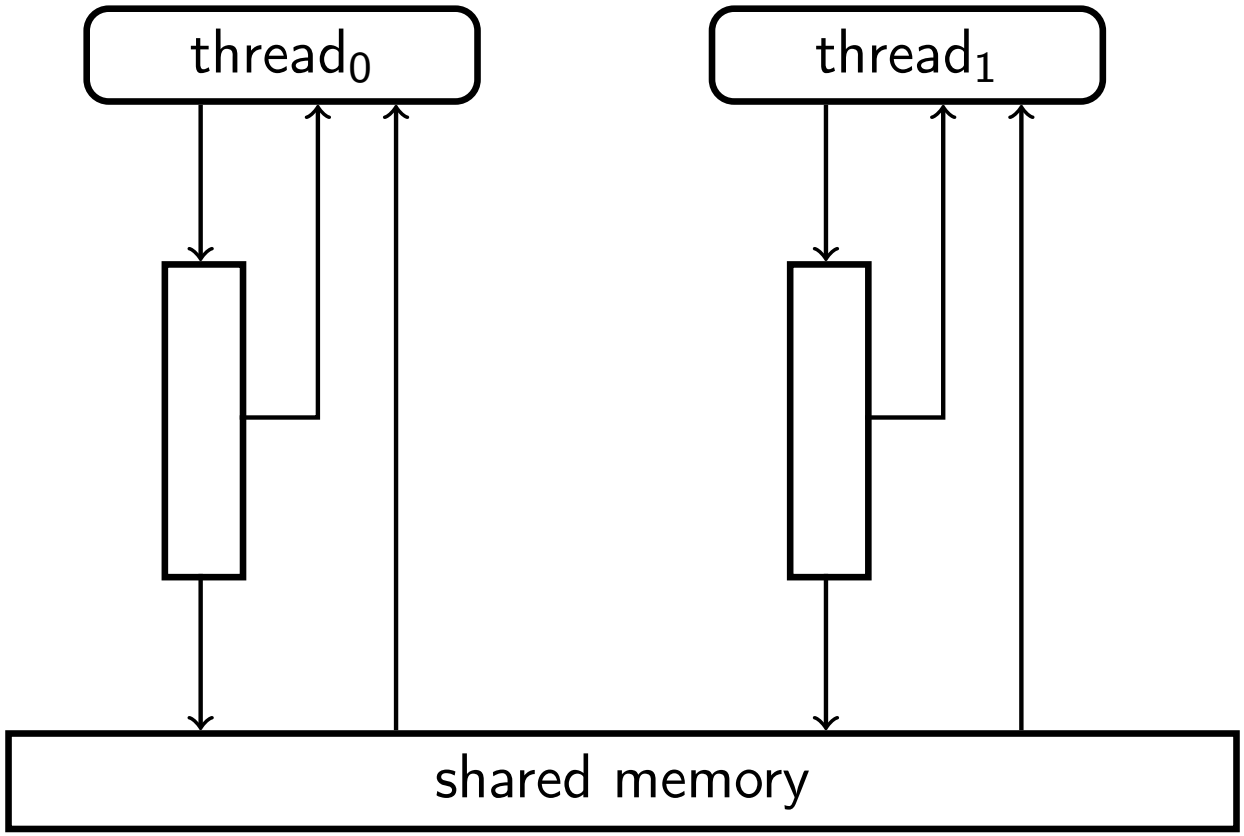
Trivial example

thread_0	thread_1
`x := 1`	`y := 1`
`print y`	`print x`

Result?

Is the printout `0,0` observable?

Hardware optimization: Write buffers



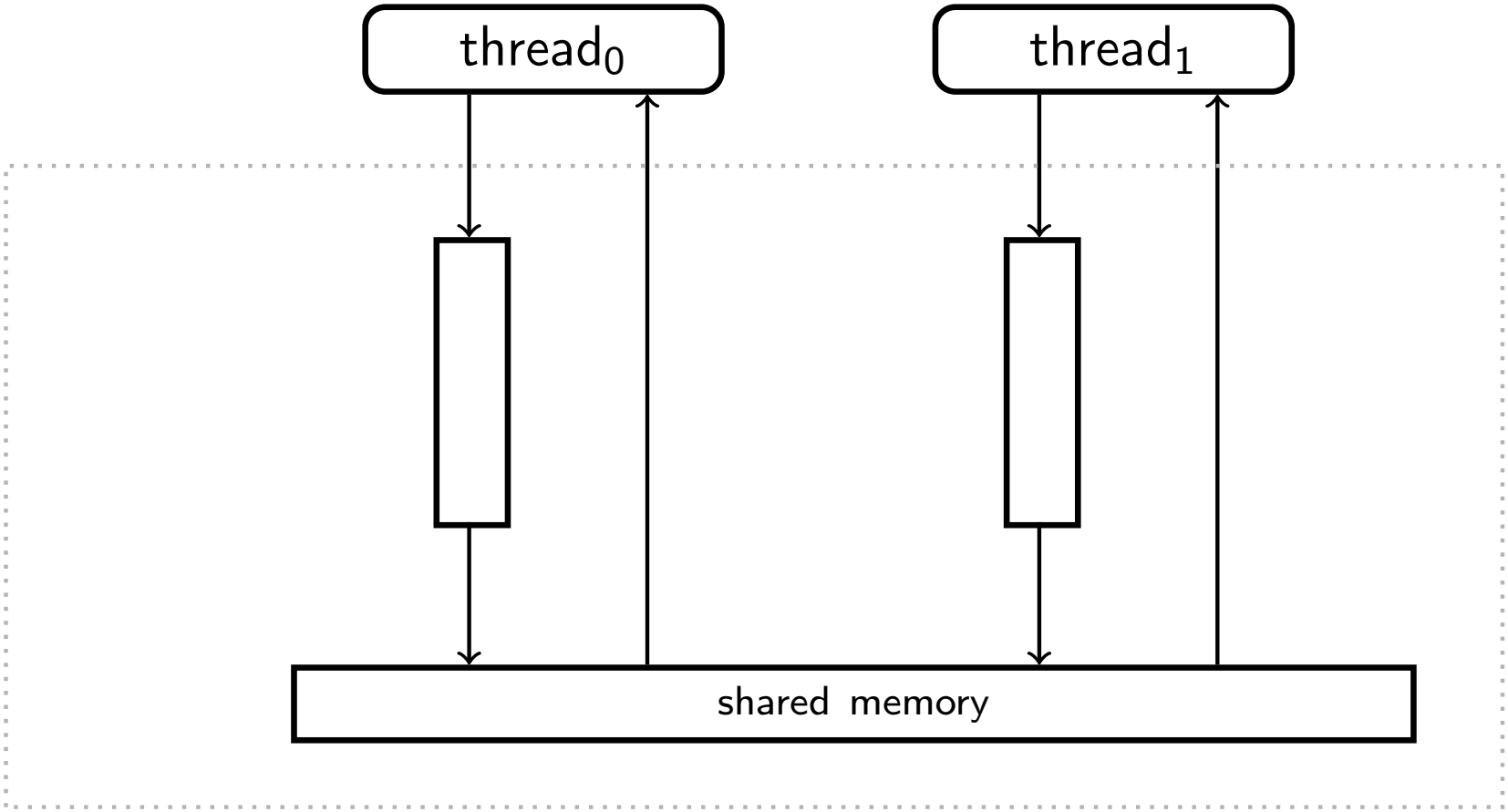
Total store order

- TSO: SPARC, pretty old already
- x86-TSO
- see [Owens et al., 2009] [Sewell et al., 2010]

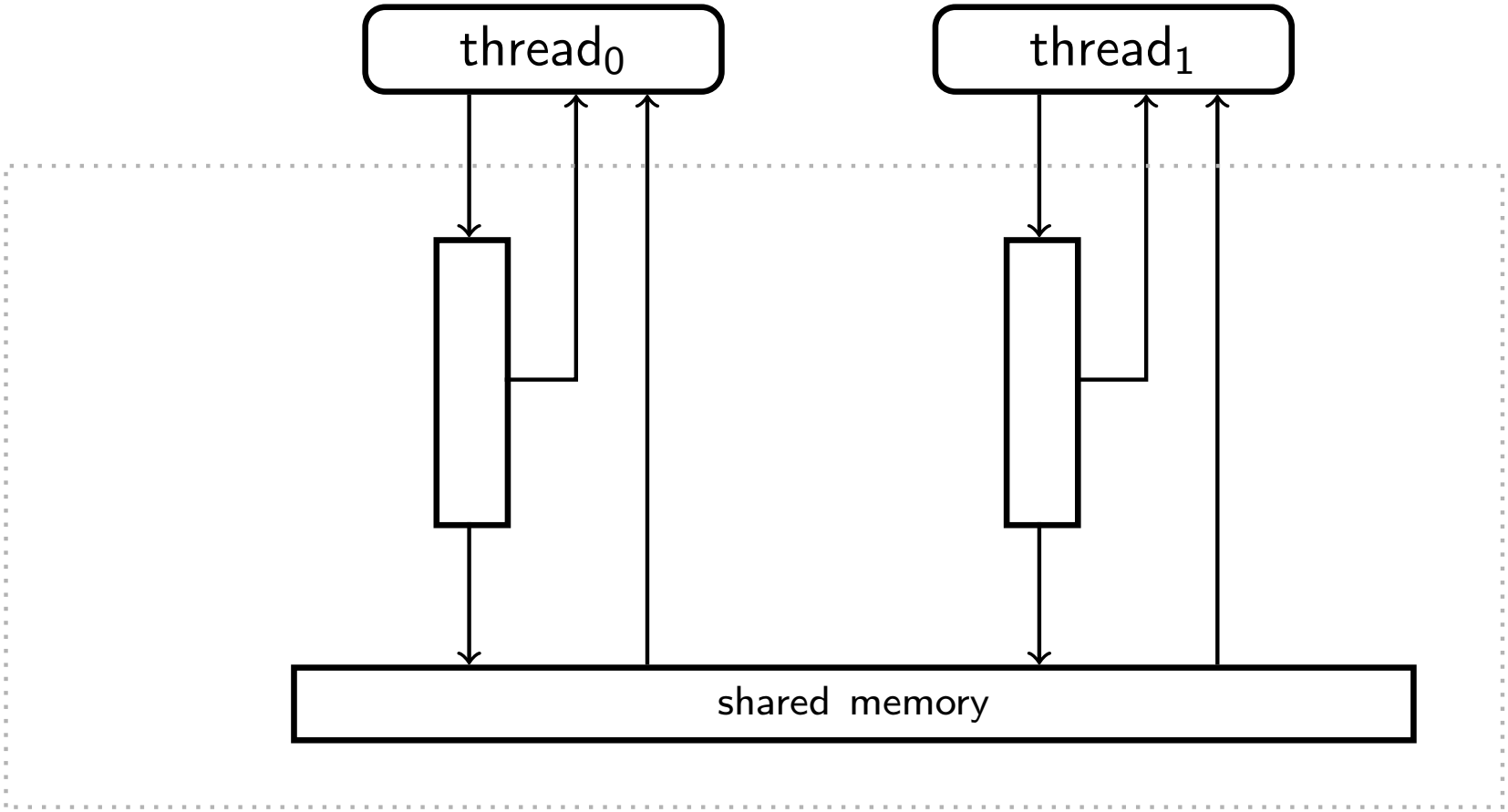
Relaxation

1. architectural: adding store buffers (aka write buffers)
2. axiomatic: relaxing program order $\Rightarrow$ W-R order dropped

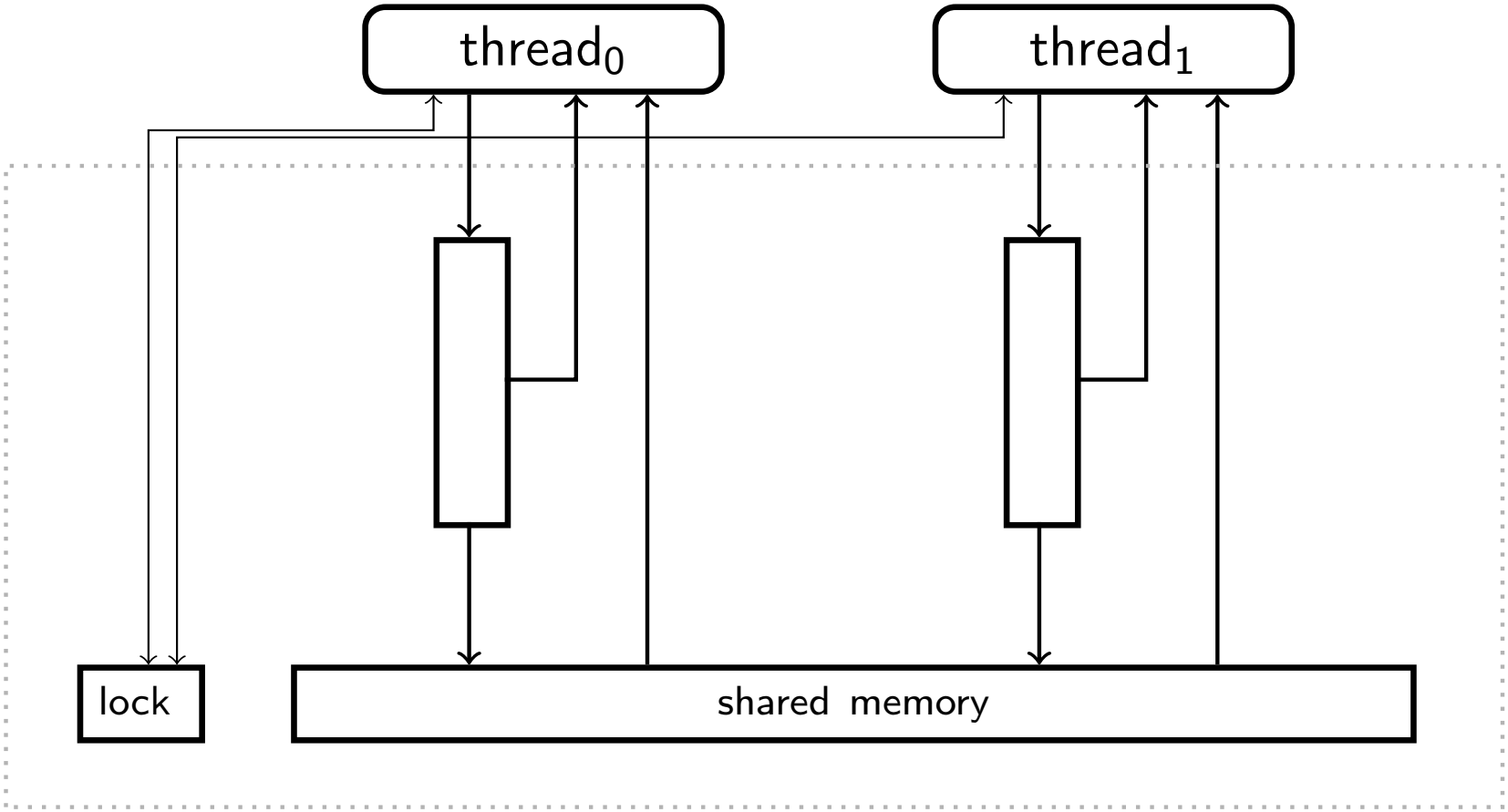
Architectural model: Write-buffers (IBM 370)



Architectural model: TSO (SPARC)



Architectural model: x86-TSO



Directly from Intel's spec

Intel 64/IA-32 architecture sofware developer's manual [int, 2013] (over 3000 pages long!)

- single-processor systems:
  - Reads are not reordered with other reads.
  - Writes are not reordered with older reads.
  - Reads may be reordered with older writes to different locations but not with older writes to the same location.
  - ...
- for multiple-processor system
  - Individual processors use the same ordering principles as in a single-processor system.
  - Writes by a single processor are observed in the same order by all processors.
  - Writes from an individual processor are NOT ordered with respect to the writes from other processors ...
  - Memory ordering obeys causality (memory ordering respects transitive visibility).
  - Any two stores are seen in a consistent order by processors other than those performing the store
  - Locked instructions have a total order

- FIFO store buffer
- read = read the most recent buffered write, if it exists (else from main memory)
- buffered write: can propagate to shared memory at any time (except when lock is held by other threads).

behavior of LOCK'ed instructions

- obtain global lock
- flush store buffer at the end
- release the lock
- note: no reading allowed by other threads if lock is held

SPARC V8 Total Store Ordering (TSO):

a read can complete before an earlier write to a different address, but a read cannot return the value of a write by another processor unless all processors have seen the write (it returns the value of *own* write before others see it)

Consequences: In a thread: for a write followed by a read (to different addresses) the order can be *swapped*

Justification: Swapping of $W - R$ is *not observable* by the programmer, it does not lead to *new, unexpected* behavior!

Example

thread	thread′
`flag := 1`	`flag′ := 1`
`A := 1`	`A := 2`
$\texttt{reg}_1$ `:= A`	$\texttt{reg}'_1$ `:= A`
$\texttt{reg}_2$ `:= flag′`	$\texttt{reg}'_2$ `:= flag`

Result?

In TSO[a]

- $(\texttt{reg}_1, \texttt{reg}'_1) = (1,2)$ observable (as in SC)
- $(\texttt{reg}_2, \texttt{reg}'_2) = (0,0)$ observable

[a] Different from IBM 370, which also has write buffers, but not the possibility for a thread to read from its own write buffer

Axiomatic description

- consider "temporal" ordering of memory commands (read/write, load/store etc)
- program order $<_p$:
  - order in which memory commands are issued by the processor

  $=$ order in which they appear in the program code
- memory order $<_m$: order in which the commands become effective/visible in main memory

Order (and value) conditions

RR: $l_1 <_p l_2 \implies l_1 <_m l_2$

WW: $s_1 <_p s_2 \implies s_1 <_m s_2$

RW: $l_1 <_p s_2 \implies l_1 <_m s_2$

Latest write wins: $val(l_1) = val(\max_{<_m}\{s_1 <_m l_1 \quad \vee \quad s_1 <_p l_1\})$

ARM and Power architecture

- ARM and POWER: similar to each other
- ARM: widely used inside smartphones and tablets (battery-friendly)
- POWER architecture = **P**erformance **O**ptimization **W**ith **E**nhanced **R**ISC., main driver: IBM

Memory model

much weaker than x86-TSO

- exposes multiple-copy semantics to the programmer

"Message passing" example in POWER/ARM

thread_0 wants to pass a message over "channel" x to thread_1, shared var y used as flag.

Initially: `x = y = 0`

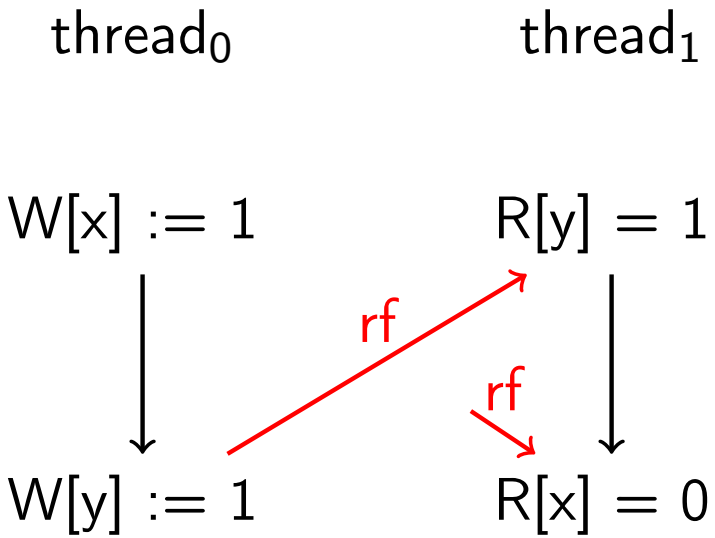
thread_0	thread_1
`x := 1`	`while (y=0) { };`
`y := 1`	`r := x`

Result?

Is the result `r = 0` observable?

- impossible in (x86-)TSO
- it would violate W-W order

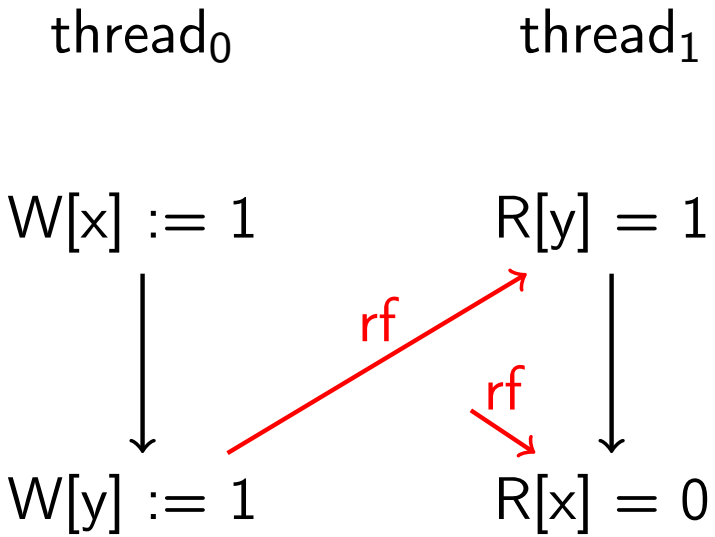
Analysis of the example



How could that happen?

1. thread does stores out of order
2. thread does loads out of order
3. store propagates between threads out of order.

Analysis of the example

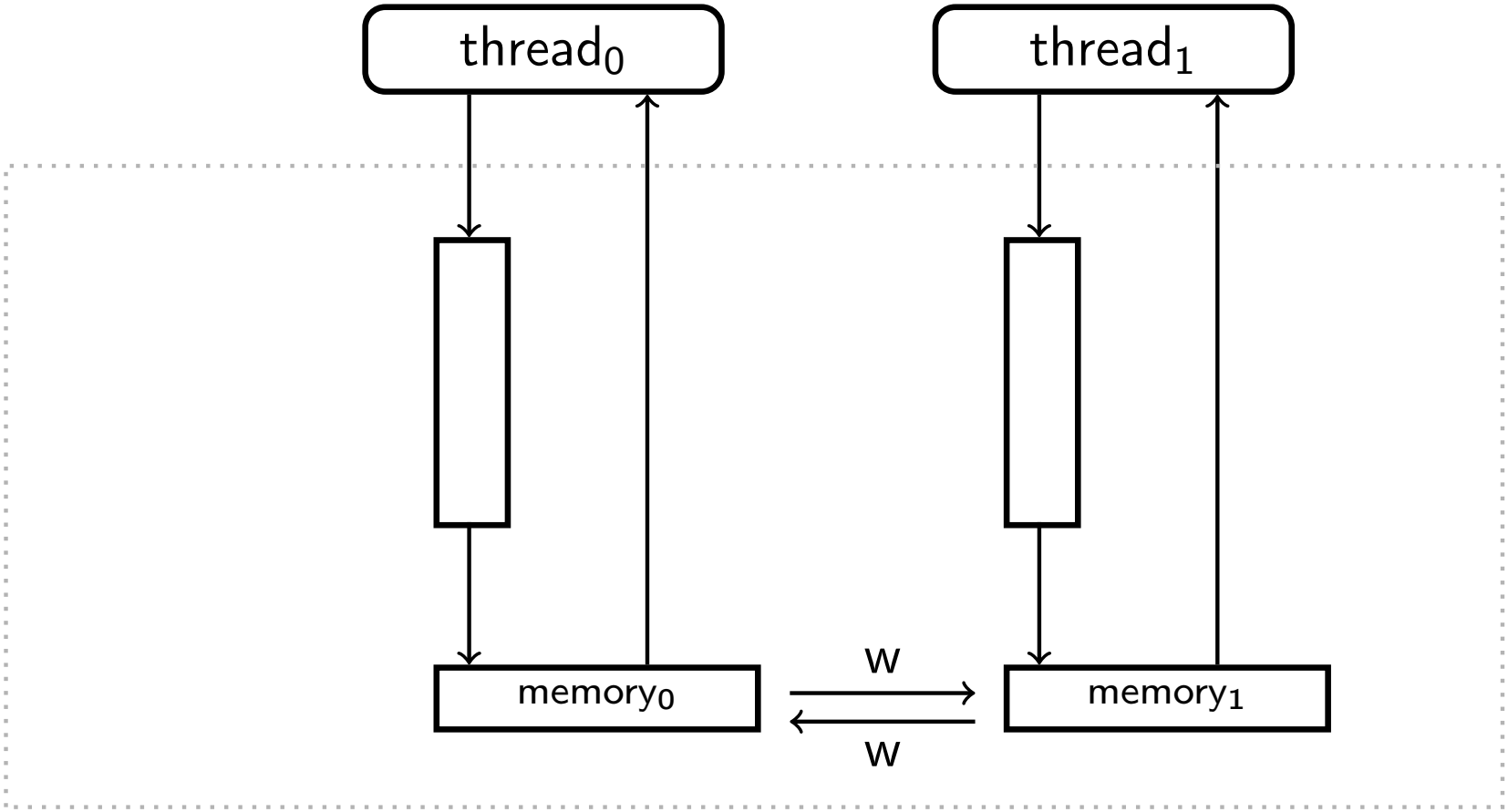


How could that happen?

1. thread does stores out of order
2. thread does loads out of order
3. store propagates between threads out of order.

Power/ARM do all three!

Conceptual memory architecture



Power and ARM order constraints

basically, program order is not preserved[2] (!) unless.

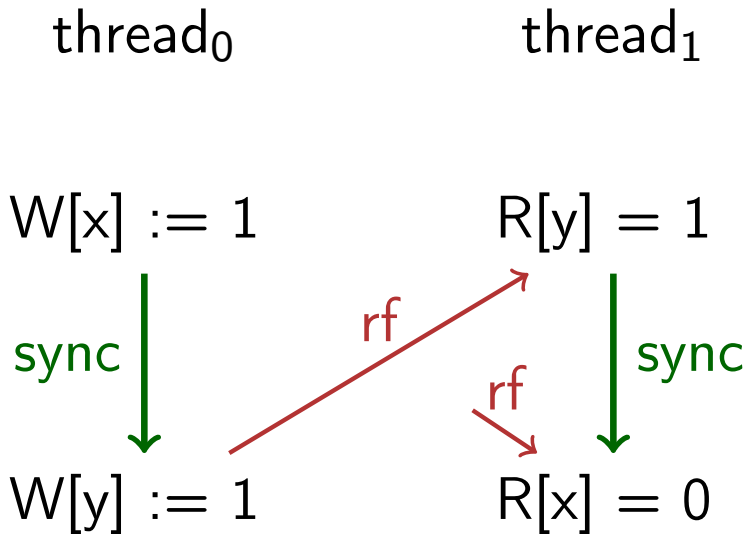
- writes to the same location
- address dependency between two loads
- dependency between a load and a store,
  1. address dependency
  2. data dependency
  3. control dependency
- use of synchronization instructions.

[2] in other words: "semicolon" etc is meaningless

Repair of the MP example

To avoid reorder: Barriers

- heavy-weight: sync instruction (POWER)
- light-weight: lwsync



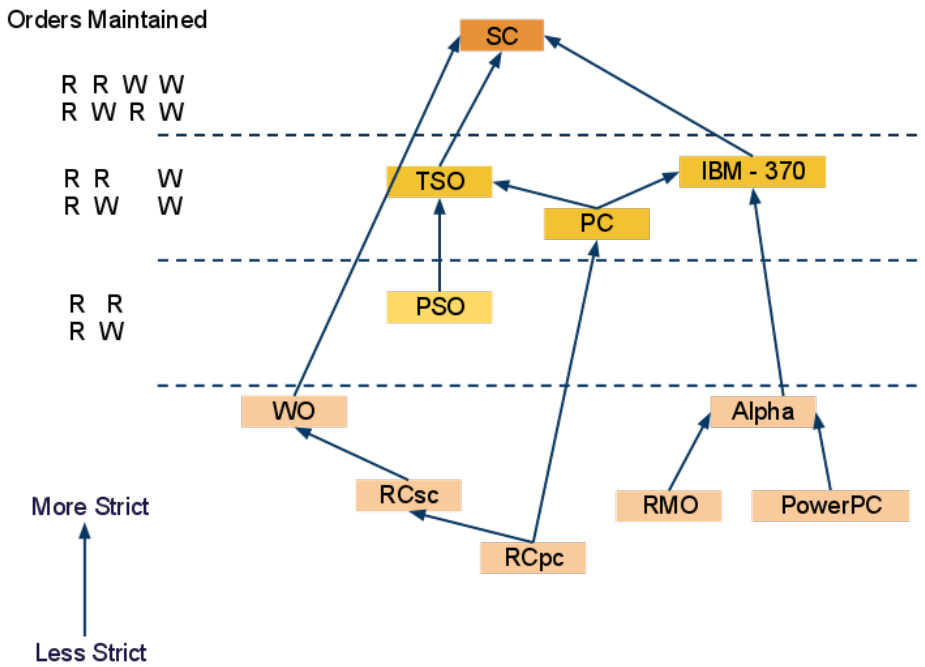
Stranger still, perhaps

thread_0	thread_1
`x := 1`	`print y`
`y := 1`	`print x`

Result?

Is the printout `y = 1, x = 0` observable?

Relationship between different models



(from

`http://wiki.expertiza.ncsu.edu/index.php/CSC/ECE_506_Spring_2013/10c_ks`)

Java memory model

- known, influential example for a memory model for a programming language.
- specifies how Java threads interact through memory
- weak memory model
- under long development and debate
- original model (from 1995):
  - widely criticized as flawed
  - disallowing many runtime optimizations
  - no good guarantees for code safety
- more recent proposal: Java Specification Request 133 (JSR-133), part of Java 5
- see [Manson et al., 2005]

Correctly synchronized programs *and* others

1. Correctly synchronized programs: correctly synchronized, i.e., data-race free, programs are sequentially consistent ("Data-race free" model [Adve and Hill, 1990])
2. Incorrectly synchronized programs: A clear and definite semantics for incorrectly synchronized programs, without breaking Java's security/safety guarantees.

tricky balance for programs with data races:

disallowing programs violating Java's security and safety guarantees vs. flexibility still for standard compiler optimizations.

Data race free model

data race free programs/executions are sequentially consistent

Data race

- A data race is the "simultaneous" access by two threads to the same shared memory location, with at least one access a write.
- a program is race free if no execution reaches a race.

- note: the definition seems ambiguous!

Data race free model

data race free programs/executions are sequentially consistent

Data race with a twist

- A data race is the "simultaneous" access by two threads to the same shared memory location, with at least one access a write.
- a program is race free if no sequentially consistent execution reaches a race.

Order relations

synchronizing actions: locking, unlocking, access to volatile variables

Definition

1. synchronization order $<_{sync}$: total order on all synchronizing actions (in an execution)
2. synchronizes-with order: $<_{sw}$
   - an unlock action *synchronizes-with* all $<_{sync}$-subsequent lock actions by any thread
   - similarly for volatile variable accesses
3. happens-before ($<_{hb}$): transitive closure of program order and synchronizes-with order

Happens-before memory model

- simpler than/approximation of Java's memory model
- distinguising `volative` from non-`volatile` reads
- happens-before

Happens before consistency

In a given execution:

- if $R[x] <_{hb} W[X]$, then the read cannot observe the write
- if $W[X] <_{hb} R[X]$ and the read observes the write, then there does not exists a $W'[X]$ s.t. $W[X] <_{hb} W'[X] <_{hb} R[X]$

Synchronization order consistency (for `volatile`-s)

- $<_{sync}$ consistent with $<_p$.
- If $W[X] <_{hb} W'[X] <_{hb} R[X]$ then the read sees the write $W'[X]$

Incorrectly synchronized code

Initially: $x = y = 0$

$thread_0$	$thread_1$
$r_1 := x$	$r_2 := y$
$y := r_1$	$x := r_2$

- obviously: a race
- however:

out of thin air

observation $r_1 = r_2 = 42$ not wished, but consistent with the happens-before model!

Happens-before: volatiles

- cf. also the "message passing" example

`ready` volatile

Initially: `x = 0, ready = false`

thread_0	thread_1
`x := 1`	`while (!ready) do skip`
`ready := true`	r_1 `:= x`

- `ready` volatile $\Rightarrow r_1 = 1$ guaranteed

Problem with the happens-before model

Initially: $x = 0, y = 0$

$thread_0$	$thread_1$
$r_1 := x$	$r_2 := y$
if $(r_1 \neq 0)$	if $(r_2 \neq 0)$
$y := 42$	$x := 42$

- the program is correctly synchronized!

$\Rightarrow$ observation $y = x = 42$ disallowed

- However: in the happens-before model, this is allowed!

violates the "data-race-free" model

$\Rightarrow$ add causality

Causality: second ingredient for JMM

JMM

Java memory model = happens before + causality

- circular causality is unwanted
- causality eliminates:
  - data dependence
  - control dependence

Causality and control dependency

Initially: a = 0; b = 1

thread$_0$	thread$_1$
r$_1$:= a	r$_3$:= b
r$_2$:= a	a := r$_3$;
if (r$_1$ = r_2)	
b := 2;	

$\Longrightarrow$

Initially: a = 0; b = 1

thread$_0$	thread$_1$
b := 2	r$_3$:= b;
r$_1$:= a	a := r$_3$;
r$_2$:= r$_1$	
if (true) ;	

is $r_1 = r_2 = r_3 = 2$ possible?

$r_1 = r_2 = r_3 = 2$ is sequentially consistent

Optimization breaks control dependency

Causality and data dependency

Initially: $x = y = 0$

$thread_0$	$thread_1$
$r_1 := x;$	$r_3 := y;$
$r_2 := r_1 \vee 1;$	$x := r_3;$
$y := r_2;$	

$\Longrightarrow$

Initially: $x = y = 0$

$thread_0$	$thread_1$
$r_2 := 1;$	$r_3 := y;$
$y := 1$	$x := r_3;$
$r_1 := x$	

using *global* analysis

Is $r_1 = r_2 = r_3 = 1$ possible?

$\vee$ = bit-wise or on integers

Optimization breaks data dependence

Summary: Un-/Desired outcomes for causality

Disallowed behavior

Initially: x = y = 0

thread$_0$	thread$_1$
r_1 := x	r_2 := y
y := r_1	x := r_2

$r_1 = r_2 = 42$

Initially: x = 0, y = 0

thread$_0$	thread$_1$
r_1 := x	r_2 := y
if ($r_1 \neq 0$)	if ($r_2 \neq 0$)
y := 42	x := 42

$r_1 = r_2 = 42$

Allowed behavior

Initially: a = 0; b = 1

thread$_0$	thread$_1$
r_1 := a	r_3 := b
r_2 := a	a := r_3;
if (r_1 = r_2)	
b := 2;	

is $r_1 = r_2 = r_3 = 2$ possible?

Initially: x = y =0

thread$_0$	thread$_1$
r_1 := x;	r_3 := y;
r_2 := $r_1 \vee 1$;	x := r_3;
y := r_2;	

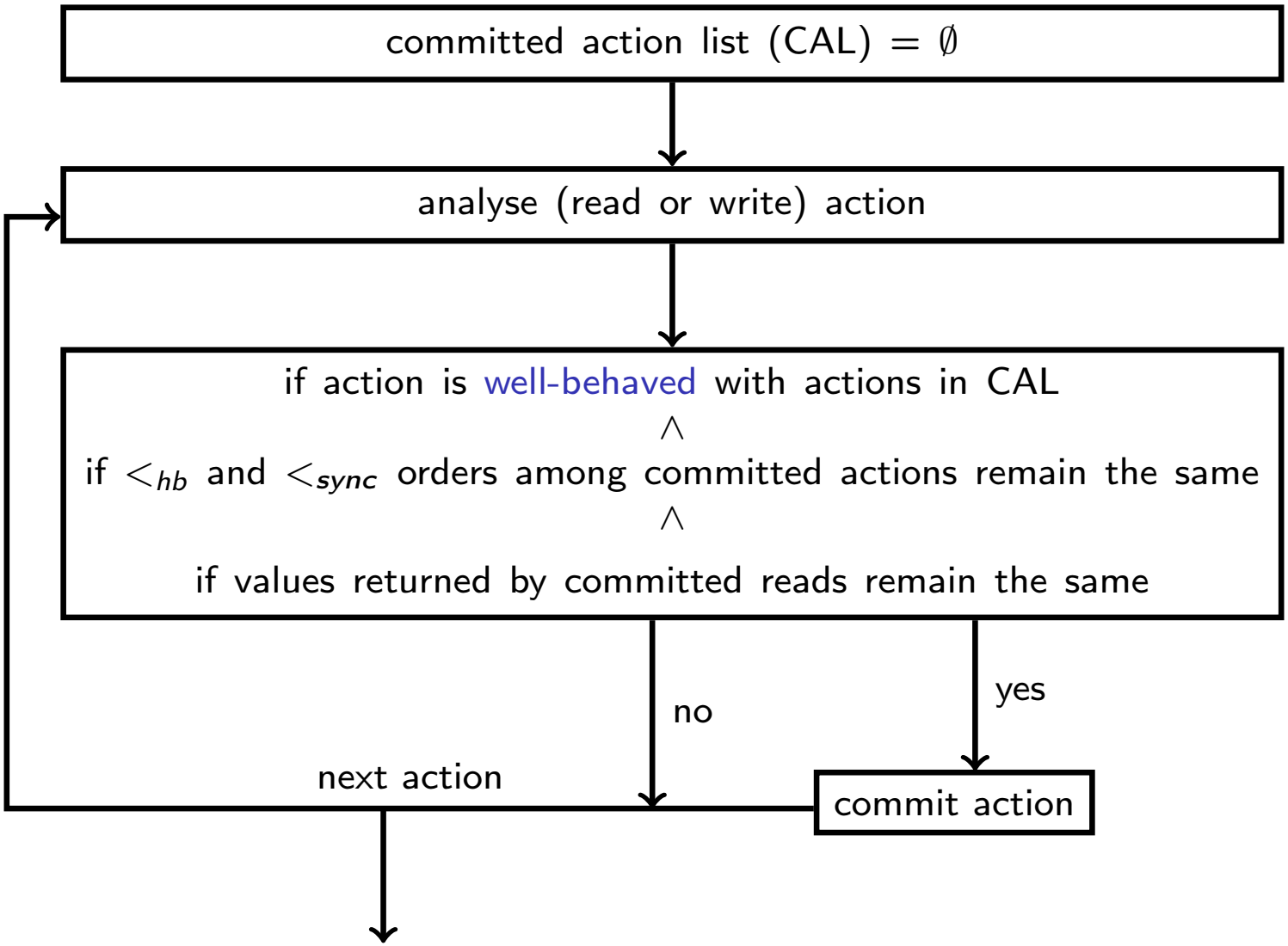
Is $r_1 = r_2 = r_3 = 1$ possible?

- key of causality: well-behaved executions (i.e. consistent with SC execution)
- non-trivial, subtle definition
- writes can be done early for well-behaved executions

Well-behaved

a not yet commited read must return the value of a write which is $<_{hb}$.

Iterative algorithm for well-behaved executions



JMM impact

- considerations for implementors
  - control dependence: should not reorder a write above a non-terminating loop
  - weak memory model: semantics allow re-ordering,
  - other code transformations
    - synchronization on thread-local objects can be ignored
    - volatile fields of thread local obects: can be treated as normal fields
    - redundant synchronization can be ignored.
- Consideration for programmers
  - DRF-model: make sure that the program is correctly synchronized $\Rightarrow$ don't worry about re-orderings
  - Java-spec: no guarantees whatsoever concerning pre-emptive scheduling or fairness

Go language and weak memory

- Go: supports shared var (but frowned upon)
- favors *message passing* (channel communication)
- "standard" modern-flavored WMM (like Java, C++11)
- based on happens-before
- specified in `https://golang.org/ref/mem` (in English)

Advice for average programmers[a] [Go memory model, 2014]

[a] But of course participants of this course well-trained enough to make sense of the document.

> *"If you must read the rest of this document to understand the behavior of your program, you are being too clever.*
>
> *Don't be clever"*

Go MM: Programs-order implies happens-before

program order [Go memory model, 2014]

> *"Within a single goroutine, the happens-before order is the order expressed by the program."*

- goroutine: Go-speak for thread/process/asynchronously executing function body/unit-of-concurrency

Allowed and guaranteed observability

May observation [Go memory model, 2014]

A read r of a variable v is allowed to observe a write w to v if both of the following hold:

1. r does not happen before w.
2. There is no other write w' to v that happens after w but before r.

Must observation [Go memory model, 2014]

r is guaranteed to observe w if both of the following hold:

1. w happens before r.
2. Any other write to the shared variable v either happens before w or after r.

Synchronization?

- so far: only statements without sync-power (reads, writes)
- without synchronization (and in WMM): concurrent programming impossible (beyond independent concurrency)
- a few synchronization statements in Go
  - initialization, package loads
  - Go goroutine start
  - via `sync`-package: locks and mutexes, once-operation
  - channels

Channels as communication and synchronization construct

- central in Go
- message passing: fundamental for concurrency
- cf. producer/consumer problem, bounded-buffer data structure, also *Oblig-1*

Role of channels:

Communication: one can transfer data from sender to receiver, but not only that:

Channels as communication and synchronization construct

- central in Go
- message passing: fundamental for concurrency
- cf. producer/consumer problem, bounded-buffer data structure, also *Oblig-1*

Role of channels:

Communication: one can transfer data from sender to receiver, but not only that:

Synchronization:

- receiver has to wait for value
- sender has to wait, until place free in "buffer"

Channels as communication and synchronization construct

- central in Go
- message passing: fundamental for concurrency
- cf. producer/consumer problem, bounded-buffer data structure, also *Oblig-1*

Role of channels:

Communication: one can transfer data from sender to receiver, but not only that:

Synchronization:

- receiver has to wait for value
- sender has to wait, until place free in "buffer"
- and: channels introduce "barriers"

Channels as communication and synchronization construct

- central in Go
- message passing: fundamental for concurrency
- cf. producer/consumer problem, bounded-buffer data structure, also *Oblig-1*

Role of channels:

Communication: one can transfer data from sender to receiver, but not only that:

Synchronization:

- receiver has to wait for value
- sender has to wait, until place free in "buffer"
- and: channels introduce "barriers"

- technically: happens-before relation for channel communication

Happens-before for send and receive

```
x := 1     |   y := 2
c!()       |   c?()
print y    |   print x
```

which read is guaranteed / may happen?

Message passing and happens-before

Send before receive [Go memory model, 2014]

"A send on a channel happens before the corresponding receive from that channel completes."

Receives before send [Go memory model, 2014]

"The kth receive on a channel with capacity C happens before the $k + C$th send from that channel completes."

Message passing and happens-before

Send before receive [Go memory model, 2014]

"A send on a channel happens before the corresponding receive from that channel completes."

Receives before send [Go memory model, 2014]

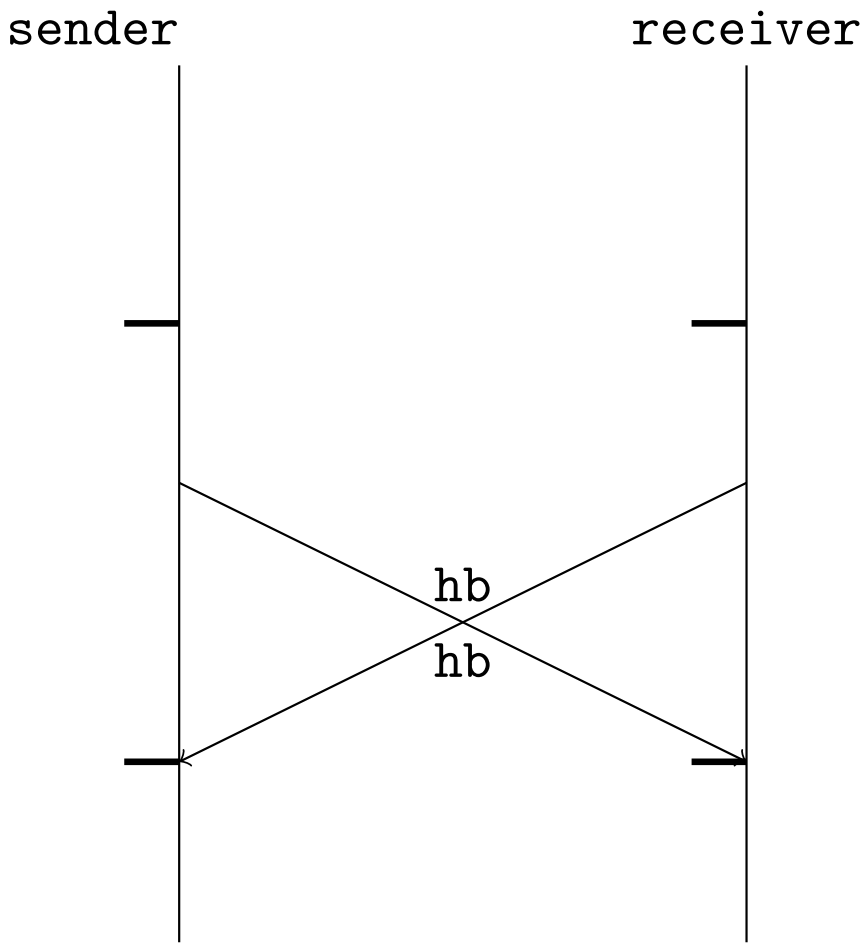
"The kth receive on a channel with capacity C happens before the $k + C$th send from that channel completes."

Receives before send, unbuffered[Go memory model, 2014]

A receive from an unbuffered channel happens before the send on that

Happens-before for send and receive

```
x := 1     |   y:=2
c!()       |   c?()
print(y)   |   print x
```



Go memory model

- catch-fire / out-of-thin-air ($\neq$ Java)
- standard: DRF programs are SC
- Concrete implementations:
  - more specific
  - platform dependent
  - difficult to "test"

```
[msteffen@rijkaard wmm] go run reorder.go
 1 reorders detected after 329 interations
 2 reorders detected after 694 interations
 3 reorders detected after 911 interations
 4 reorders detected after 9333 interations
 5 reorders detected after 9788 interations
 6 reorders detected after 9951 interations
 ...
```

Summary and conclusion

Memory/consistency models

- there are memory models for HW and SW (programming languages)
- often given informally/prose or by some "illustrative" examples (e.g., by the vendor)
- it's basically the semantics of concurrent execution with shared memory.
- interface between "software" and underlying memory hardware
- modern complex hardware $\Rightarrow$ complex(!) memory models
- defines which compiler optimizations are allowed
- crucial for correctness and performance of concurrent programs

Take-home lesson

it's impossible(!!) to produce

- correct and
- high-performance

concurrent code without clear knowledge of the chosen platform's/language's MM

- that holds: not only for system programmers, OS-developers, compiler builders . . . but also for "garden-variety" SW developers
- reality (since long) much more complex than "naive" SC model

Take home lesson for the impatient

Avoid data races at (almost) all costs (by using synchronization)!

References I

[int, 2013] (2013).
Intel 64 and IA-32 Architectures Software Developer s Manual. Combined Volumes:1, 2A, 2B, 2C, 3A, 3B and 3C.
Intel.

[Adve and Gharachorloo, 1995] Adve, S. V. and Gharachorloo, K. (1995).
Shared memory consistency models: A tutorial.
Research Report 95/7, Digital WRL.

[Adve and Hill, 1990] Adve, S. V. and Hill, M. D. (1990).
Weak ordering — a new definition.
SIGARCH Computer Architecture News, 18(3a).

[Anderson, 1990] Anderson, T. E. (1990).
The performance of spin lock alternatives for shared-memory multiprocessors.
IEEE Transactions on Parallel and Distributed System, 1(1):6–16.

[Andrews, 2000] Andrews, G. R. (2000).
Foundations of Multithreaded, Parallel, and Distributed Programming.
Addison-Wesley.

[Go memory model, 2014] Go memory model (2014).
The Go memory model.
`https://golang.org/ref/mem`.
Version of May 31, 2014, covering Go version 1.9.1.

[Herlihy and Shavit, 2008] Herlihy, M. and Shavit, N. (2008).
The Art of Multiprocessor Programming.
Morgan Kaufmann.

References II

[Lamport, 1979] Lamport, L. (1979).
How to make a multiprocessor computer that correctly executes multiprocess programs.
IEEE Transactions on Computers, C-28(9):690–691.

[Manson et al., 2005] Manson, J., Pugh, W., and Adve, S. V. (2005).
The Java memory memory.
In *Proceedings of POPL '05*. ACM.

[Owens et al., 2009] Owens, S., Sarkar, S., and Sewell, P. (2009).
A better x86 memory model: x86-TSO.
In Berghofer, S., Nipkow, T., Urban, C., and Wenzel, M., editors, *Theorem Proving in Higher-Order Logic: 10th International Conference, TPHOLs'09*, volume 5674 of *Lecture Notes in Computer Science*.

[Sewell et al., 2010] Sewell, P., Sarkar, S., Nardelli, F., and O.Myreen, M. (2010).
x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors.
Communications of the ACM, 53(7).