PURDUE UNIVERSITY

Principles of Concurrency

Week 8

Software Memory Models

Principles

- Most language definitions do not define a memory model
  - E.g., Rust, Python, Haskell, etc.
- But, there are several important exceptions
  - C/C++11
  - Java
- A language memory model provides guidance to developers and compiler writers on expected behaviors

```
   Initially X=Y=0
T1                T2
r1=X              r2=Y
if (r1==1)        if (r2==1)
    Y=1               X=1

Is outcome r1=r2=1 allowed?
```

Reasoning about the behavior of this program requires a precise definition on the values a thread is allowed to witness

---

```
struct s { char a; char b; } x;

Thread 1:          Thread 2:
  x.a = 1;           x.b = 1;

Thread 1 is not equivalent to:
  struct s tmp = x;
  tmp.a = 1;
  x = tmp;
```

Compiler transformations must be aware of threads

C11 Memory Model

- If a program has a sequentially consistent (SC) execution which contains a data race, then its behavior is undefined. The definition has no notion of a benign race.
- Otherwise, the program behaves according to one of its SC executions

Enforcing SC behavior is expensive because it requires that all writes execute atomically with respect to subsequent reads

```
                Initially X=Y=0
T1              T2              T3              T4
X=1             Y=1             r1=X            r3=Y
                                fence           fence
                                r2=Y            r4=X

r1=1, r2=0, r3=1, r4=0 violates write atomicity
```

If writes execute non-atomically, then they might get propagated to reads out-of-order

Data Races and Compiler Transformations


```
unsigned i = x;
if (i < 2) {
    foo: ...
    switch (i) {
        case 0: ...; break;
        case 1: ...; break;
        default: ...;
    }
}
```

- Suppose i needs to be spilled because of foo
- Since i and x have the same (unmodified) value, can simply reload x, rather than spilling
- Since switch is either 0 or 1, and i is in that range, there's no need to have a branch bounds check

* Suppose there is a data race that causes another thread to update x to 5 while this thread is executing foo

Low-Level Atomics

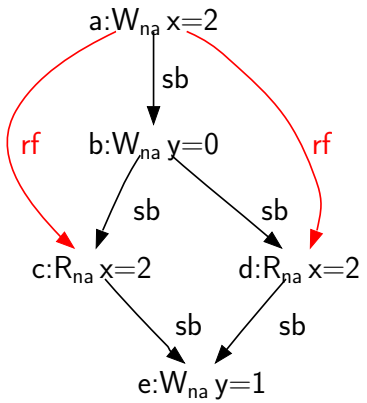
To enable high-performance applications greater control of memory visibility, the C11 memory model provides support for low-level atomics, annotations on memory accesses, that allow for weaker visibility guarantees than provided by SC

- SC annotations enforce a total ordering on all read and write accesses
- Release (REL)/Acquire (ACQ) annotations enforce a causally-consistent ordering between the operations that precede the release and follow the acquire
- Relaxed (RLX) annotations compile to single hardware loads and stores with no additional synchronization other than basic coherence guarantees

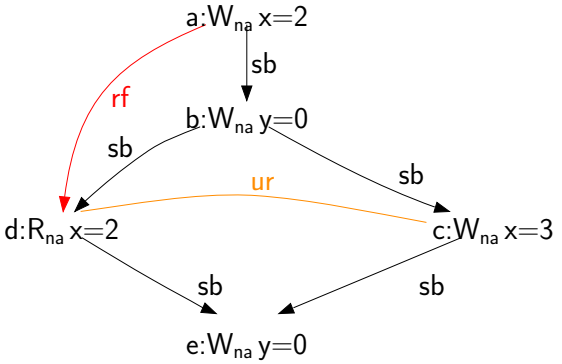
Formalize these notions using relations defined over execution events that identify potential visibility properties based on the semantics of these annotations

Single-Threaded Programs


```
int main() {
  int x = 2;
  int y = 0;
  y = (x==x);
  return 0; }
```




```
int main() {
  int x = 2;
  int y = 0;
  y = (x == (x=3));
  return 0; }
```



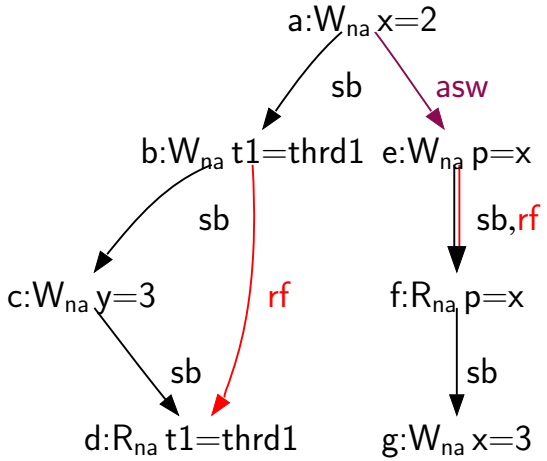
"ur" = "unsequenced race"

Constraints on reads cannot be simply that they read from the "most recent" write since there is no global linear time in a concurrent execution. Use a happens-before relation (and relations derived from it) to reason about allowed behaviors.

In a sequential execution, the "sequenced-before" (sb) corresponds to happens-before.

Threads and Non-Atomic Accesses

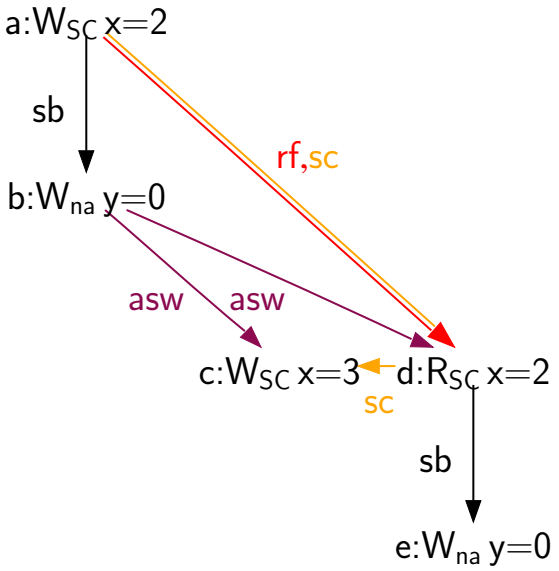

```
void foo(int* p) {*p = 3;}
int main() {
  int x = 2;
  int y;
  thread t1(foo, &x);
  y = 3;
  t1.join();
  return 0; }
```



additional-synchronizes-with edges added between parent and child thread creation event

SC Atomics


```
int main() {
  atomic_int x;
  x.store(2);
  int y = 0;
  {{{ x.store(3);
  ||| y = ((x.load())==3);
  }}};
  return 0; }
```



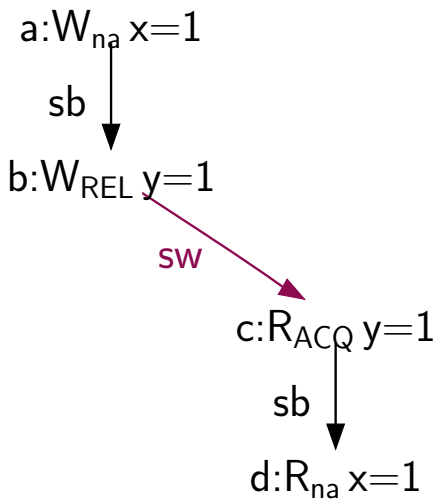
- concurrent access to x no longer considered a data race
- SC atomic operations are totally ordered
  - form an interleaving in a global timeline
- Initialization of an atomic object by non-atomic stores can potentially race however
- All other accesses are through atomic stores and loads

Release-Acquire Synchronization


```
// sender
x = ...
y = 1;
```

```
// receiver
while (0 == y);
r = x;
```

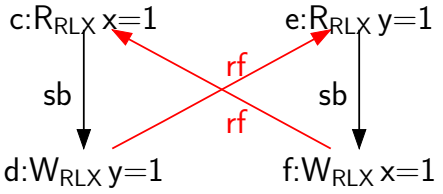
- Message-passing program
- Desired behavior:
  - Causal ordering between events that precede the release-write to y by the sender and the events that follow the acquire-read to y by the receiver



Relaxed Atomics

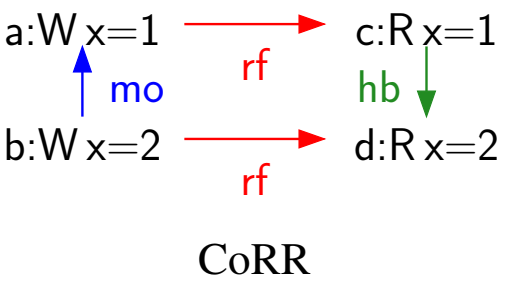
- No additional synchronization edges introduced
- Only provides basic coherence guarantees

```
int main() {
  int r1, r2;
  atomic_int x = 0;
  atomic_int y = 0;
  {{{ { r1 = x.load(mo_relaxed));
        y.store(r1,mo_relaxed); }
  ||| { r2 = y.load(mo_relaxed));
        x.store(r2,mo_relaxed); }
  }}}
  return 0; }
```

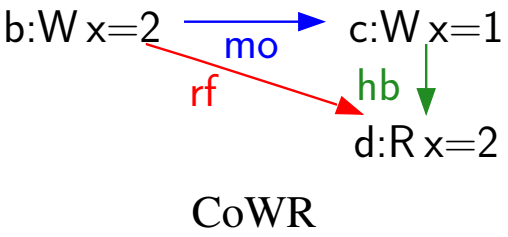


- The definition of relaxed atomics as strictly interpreted permits load buffering and out-of-thin-air reads
- The draft standard proposes conditions to prohibit such scenarios but at a cost of reasoning complexity

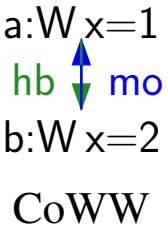
Coherence Rules



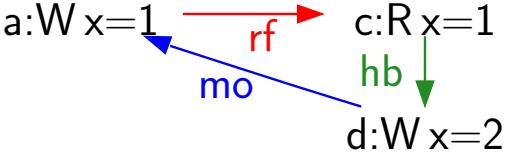
CoRR



CoWR



CoWW

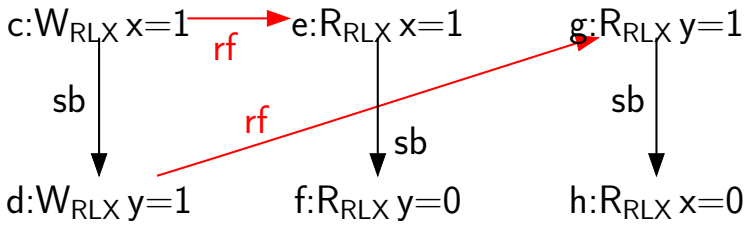


CoRW

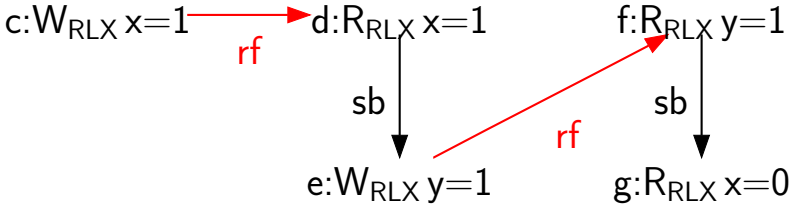
Forbidden executions

Examples

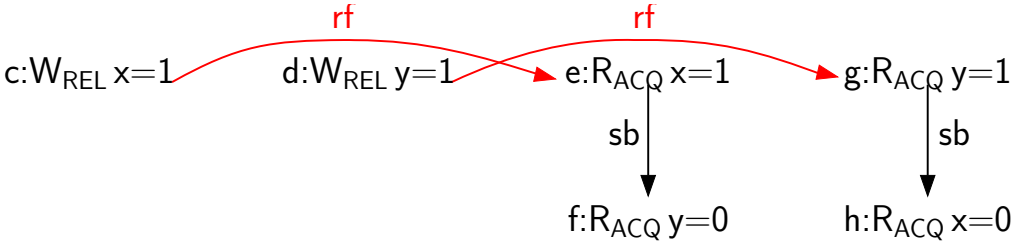
Message passing not sound with relaxed atomics:



Write-to-read causality guarantees not respected with relaxed atomics:



IRIW admitted:



Compiler Transformations

Sequentialization

$$C_1 \| C_2 \rightsquigarrow C_1; C_2$$

$$\begin{array}{l} r_1 = x.\text{load}(\text{RLX}); \\ y.\text{store}(1, \text{RLX}); \end{array} \Bigg\| \begin{array}{l} r_2 = y.\text{load}(\text{RLX}); \\ x.\text{store}(1, \text{RLX}); \end{array}$$

$$r_1 = r_2 = 1$$

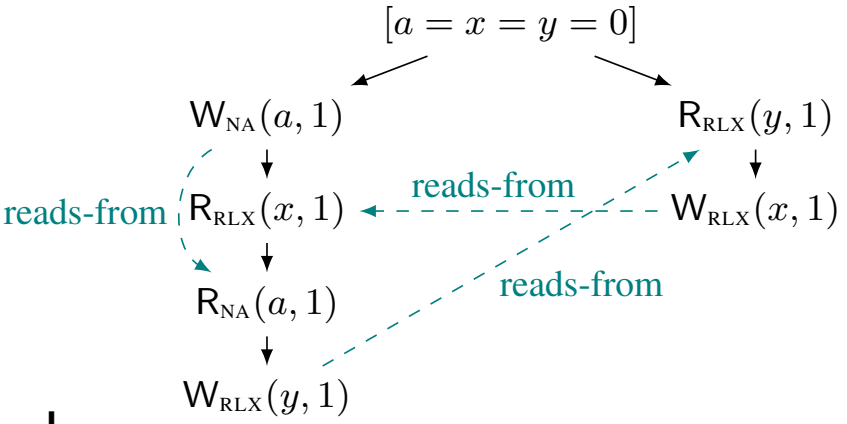
$$\begin{array}{l} \text{if } (x.\text{load}(\text{RLX})) \\ \quad y.\text{store}(1, \text{RLX}); \end{array} \Bigg\| \begin{array}{l} \text{if } (y.\text{load}(\text{RLX})) \\ \quad x.\text{store}(1, \text{RLX}); \end{array}$$

$$x = y = 1$$

$$a = 1; \Bigg\| \begin{array}{l} \text{if } (x.\text{load}(\text{RLX})) \\ \quad \text{if } (a) \\ \quad\quad y.\text{store}(1, \text{RLX}); \end{array} \Bigg\| \begin{array}{l} \text{if } (y.\text{load}(\text{RLX})) \\ \quad x.\text{store}(1, \text{RLX}); \end{array}$$

- No consistent execution in which load of a occurs
- Now, apply sequentialization:

$$\begin{array}{l} a = 1; \\ \text{if } (x.\text{load}(\text{RLX})) \\ \quad \text{if } (a) \\ \quad\quad y.\text{store}(1, \text{RLX}); \end{array} \Bigg\| \begin{array}{l} \text{if } (y.\text{load}(\text{RLX})) \\ \quad x.\text{store}(1, \text{RLX}); \end{array}$$



- An execution in which `a = x = y = 1` is now permitted

Compiler Transformations

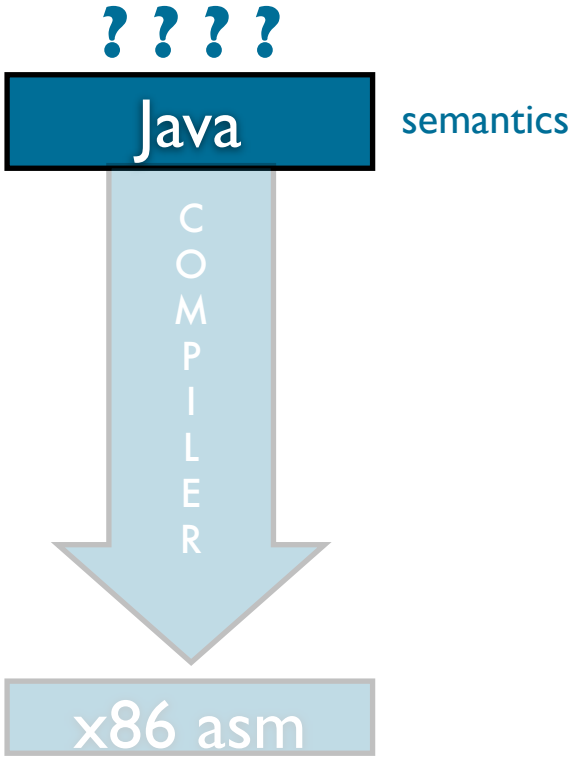
$$
\begin{array}{l}
z.\text{store}(1, \text{REL}); \\
a = 1;
\end{array}
\;\Big\|\;
\begin{array}{l}
\text{if } (x.\text{load}(\text{RLX})) \\
\quad \text{if } (z.\text{load}(\text{ACQ})) \\
\quad\quad \text{if } (a) \\
\quad\quad\quad y.\text{store}(1, \text{RLX});
\end{array}
\;\Big\|\;
\begin{array}{l}
\text{if } (y.\text{load}(\text{RLX})) \\
\quad x.\text{store}(1, \text{RLX});
\end{array}
$$

- The only possible final state of the program is `a = z = 1` and `x = y = 0`
- Reordering the two stores in the first thread now allows an execution in which `a = z = x = y = 1` is allowed.

Java Semantics

- The sequential semantics of Java is well-specified and well-understood.

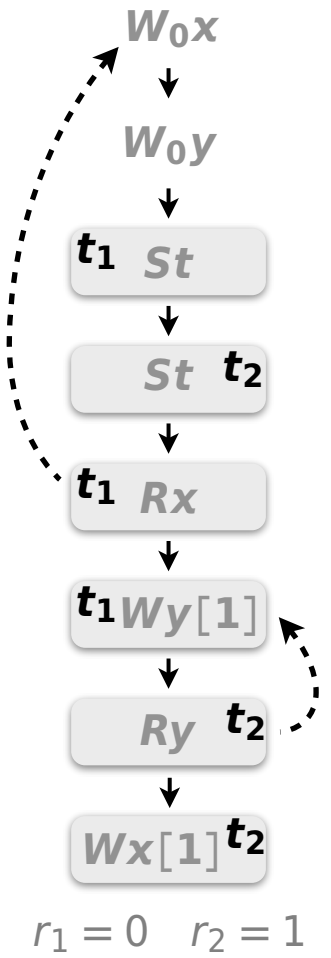
- About Java concurrency:
  - ★ Do we have a formal definition?
  - ★ Do people understand it?



First Try: Sequential Consistency

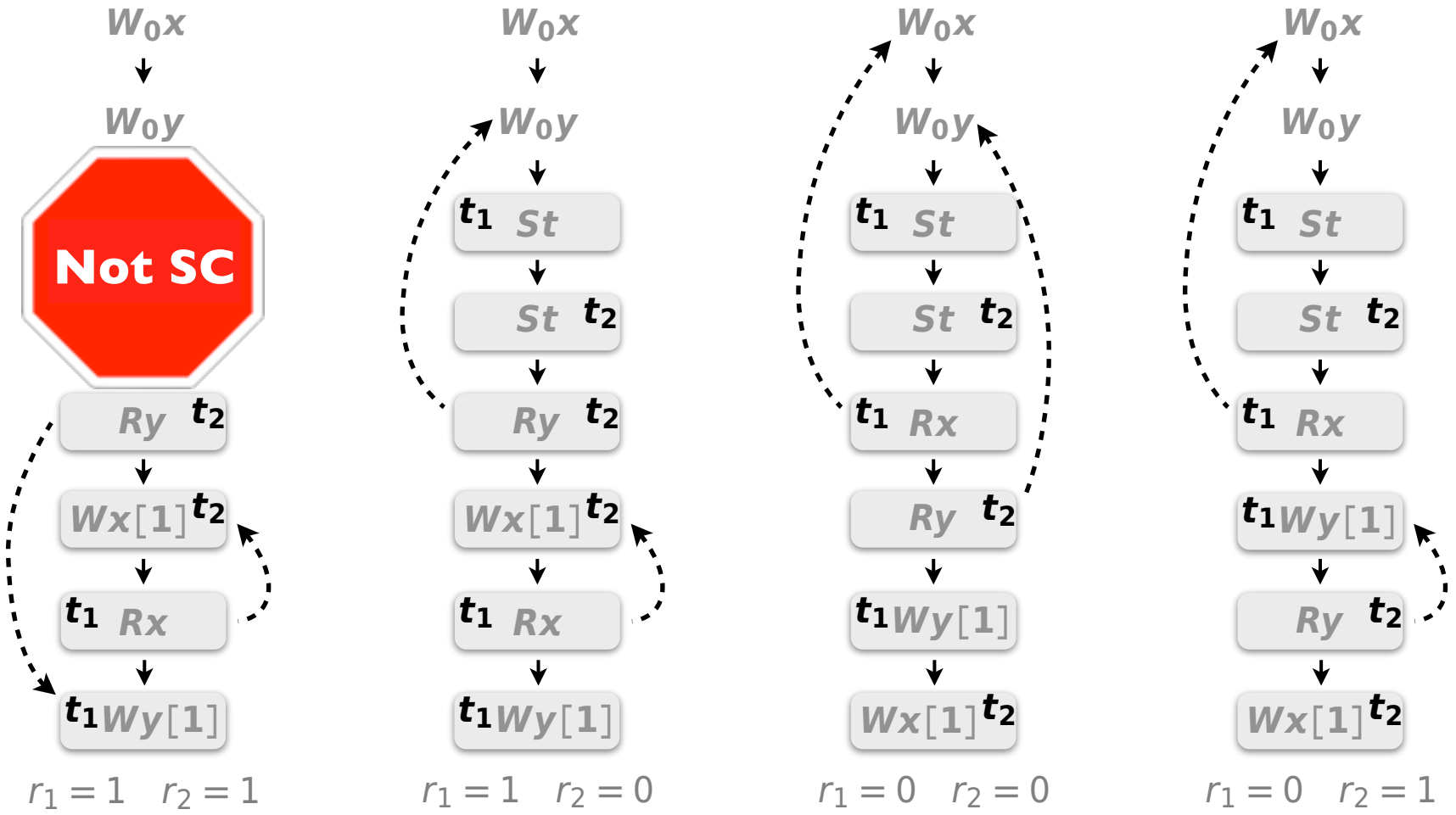
- During execution, memory actions should appear to execute one at a time in an imaginary interleaving of actions on a shared memory.
- In particular, reads of a shared variable should return the value written most recently to the memory.
- Example:

$x \leftarrow 0; y \leftarrow 0$	
$r_1 \leftarrow x$	$r_2 \leftarrow y$
$y \leftarrow 1$	$x \leftarrow 1$



Some More SC Executions

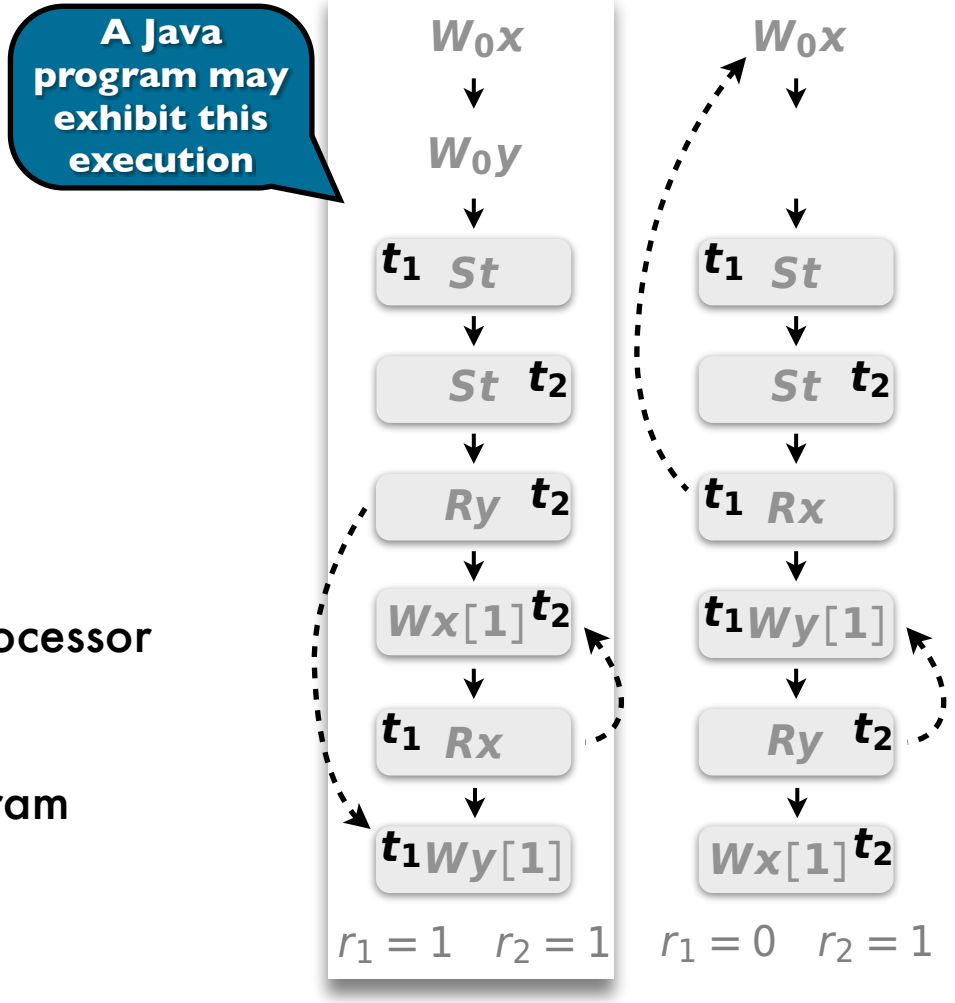
$x \leftarrow 0; y \leftarrow 0$	
$r_1 \leftarrow x$ $y \leftarrow 1$	$r_2 \leftarrow y$ $x \leftarrow 1$



Sequential Consistency and Java

SC memory model

- Do we have a formal definition? yes
- Do people understand it? yes
- Is it faithful to Java semantics and processor behavior? **no**
  - compilers will often reorder program statements
  - and hardware will do the same



Sequential Consistency and Java

Let's try...

$x \leftarrow 0; y \leftarrow 0$	
$y \leftarrow 1$	$x \leftarrow 1$
$r_1 \leftarrow x$	$r_2 \leftarrow y$
$r_1 = 0$	$r_2 = 0$

is not visible in SC

```
public class Relaxed1 extends Thread {
        public static volatile long iter = 0;
        public static volatile int fence = 0;
        public static int res0 = 1;
        public static int res1 = 1;
        private static int x = 0;
        private static int y = 0;
        private int tid;
        private long wait;

        public Relaxed1(int tid, long wait){
                this.tid = tid;
                this.wait = wait;
        }

        public void run(){
                if (this.tid == 0) {
                        while(iter == this.wait);
                        x = 1;
```

```
Administrator — bash — 80×24
bash-3.2$ javac Relaxed1.java
bash-3.2$ java Relaxed1
Relaxed after 5055 iterations!
bash-3.2$
```

```
                while (res0 != 0 || res1 != 0){
                        Relaxed1 th1 = new Relaxed1(0,iter);
                        Relaxed1 th2 = new Relaxed1(1,iter);
                        th1.start();
                        th2.start();
                        iter++;
                        try {
                                th1.join();
                                th2.join();
                        } catch (InterruptedException e) {}
                        x = 0;
                        y = 0;
                }
                if (res0 == 0 && res1 == 0)
                    System.out.println("Relaxed after "+iter+" iterations!");
        }
}
```

The Java Memory Model

- Proposed in 1995, but broken.
- New version in 2004 (JSR-133). Published POPL'05 (Manson, Pugh, Adve)
  - Also broken :-)

- JMM tries *squaring the circle*
  - ★Guarantees for programmers
    - ✦Data-race free programs execute like in a SC model
    - ✦A (safe) formal semantics for all Java programs (incl. those with races)
  - ★Guarantees for optimizers
    - ✦Allows all various hardware reordering semantics
    - ✦Allows aggressive compiler optimizations

Happens-Before Model

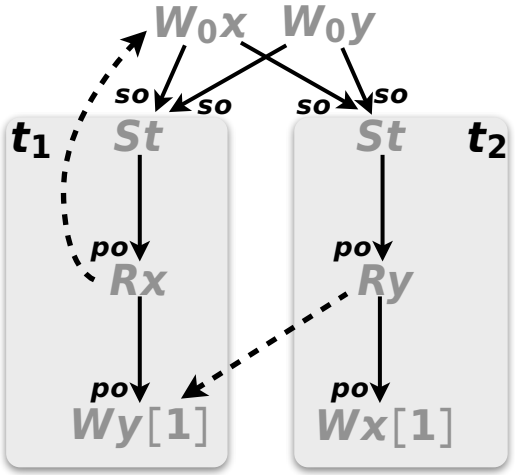
An interleaving of actions is better described without global time.

It is in fact a consistent extension of a partial order called **happens before**.

The **hb** relation is the smallest transitive relation that respects

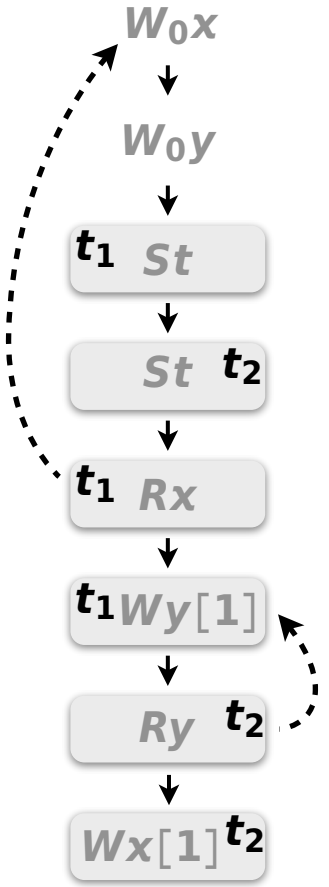
- program order between actions of same thread
- synchronization order between actions of threads

This relation should constrain the write actions that a read action can legally see.



$$x \leftarrow 0; y \leftarrow 0$$

$r_1 \leftarrow x$	$r_2 \leftarrow y$
$y \leftarrow 1$	$x \leftarrow 1$



Happens-Before Model

An axiomatic execution is described by a tuple

$$\left\langle P, A, \xrightarrow{po}, \xrightarrow{so}, W \right\rangle$$

Among all executions, the happens-before model selects the so-called **well-formed executions**

Ex: a read action must not see a write that happens after it.

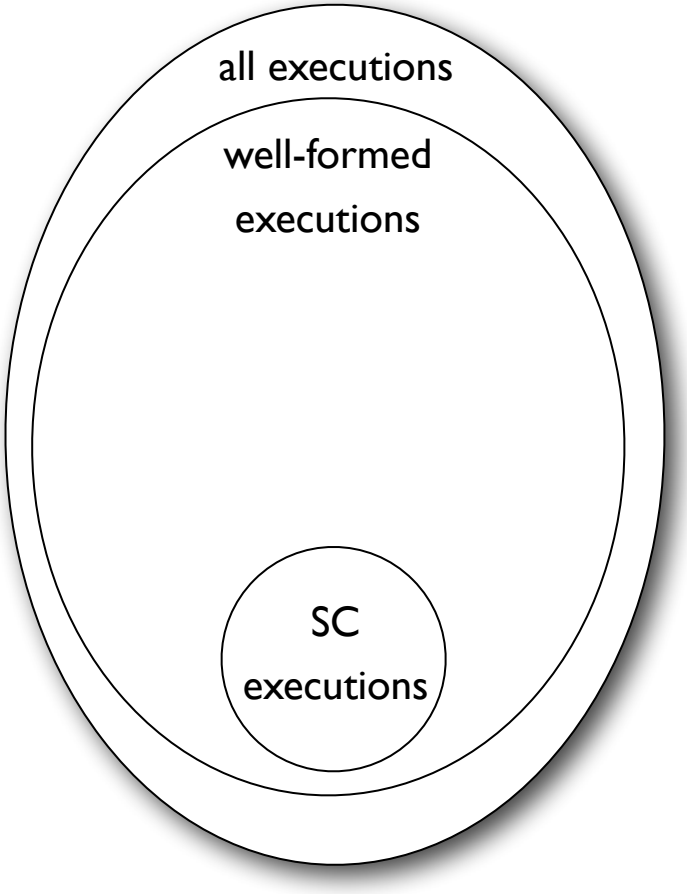
Such a model

is quite easy to understand,

contains all SC executions,

but is too relaxed: it does not respect the DRF property

allows cyclic dependencies

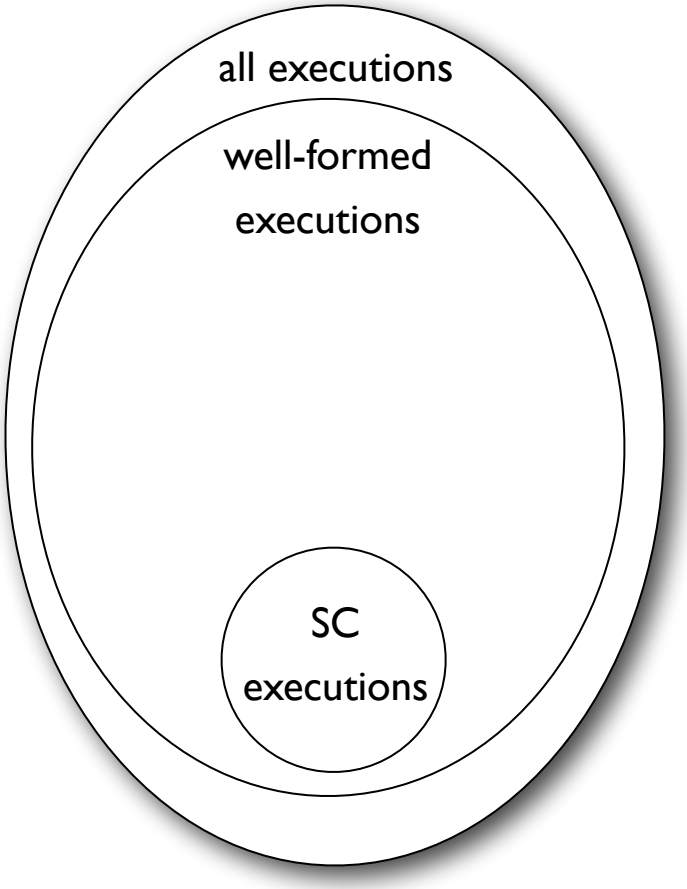


Data Race Freedom

Data race free program:

- Intuitively: a program where two threads don't access at the same time a non-volatile shared location (at least one of the access is a write)
- Formally: in all executions of the model, conflicting actions must be in the **hb** relation

Data race free model: data-race free programs only have SC executions.



Race Committing Sequence

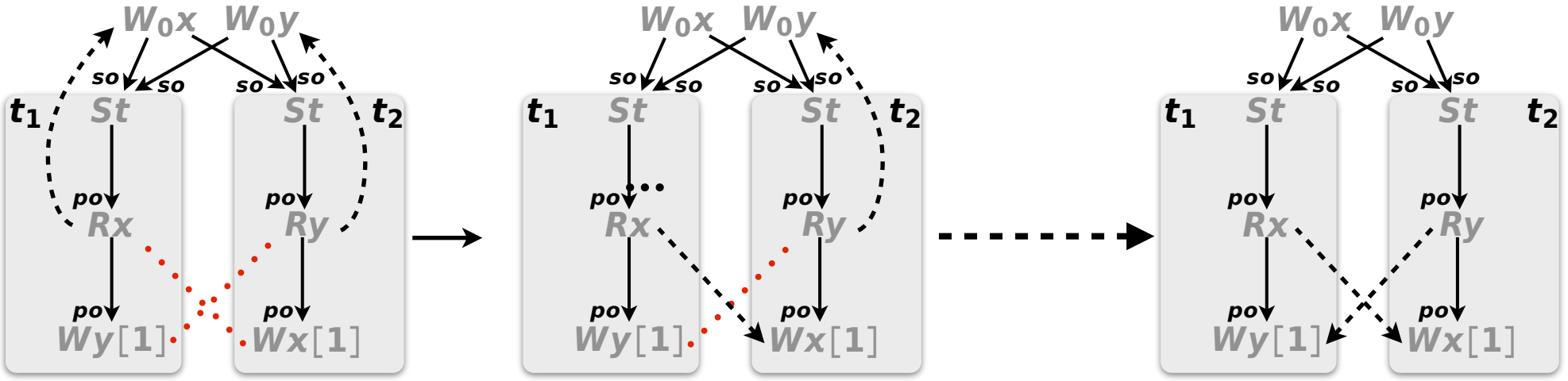
$x \leftarrow 0; y \leftarrow 0$	
$r_1 \leftarrow x$ $y \leftarrow 1$	$r_2 \leftarrow y$ $x \leftarrow 1$

Is $r_1 = 1 \quad r_2 = 1$ allowed ?

each read sees a write that happens before it

1. Start from a well-behaved execution
2. Commit races: modifies the write-seen
3. Restart the execution taking the race into account
4. Goto 2 until all actions are committed

+ constraints on the sequence to rule out cyclic causality



a well-behaved execution — one write-seen is updated — final JMM execution

Examples

initially x = y = 0

r1 = x	r2 = y
y = 1	x = 1

A. (allowed)

initially x = y = 0

lock m1	lock m2
r1=x	r2=y
unlock m1	unlock m2
lock m2	lock m1
y=1	x=1
unlock m2	unlock m1

B. (prohibited)

initially x = y = 0

r1 = x	r2 = y
y = r1	x = r2

C. (prohibited)

Is it possible to get r1 = r2 = 1 at the end of an execution?

Compiler Transformations

The meaning of a Java program is given by its set of traces:

- A transformation that does not change this set is trivially valid

```
if (r1==1)                    x=1
     {x=1;y=1}      ⇄         y=1
else {x=1;y=1}
```

Independent writes can be reordered

```
x=1          y=1
y=1    ⟶    x=1
```

Elimination of redundant reads

```
r1 = x              r1 = x
r2 = x       ⟶     r2 = r1
if (r1==r2)         if (r1==r2)
  y = 1               y = 1
```

(read after read)

```
x = r1              x = r1
r2 = x       ⟶     r2 = r1
if (r1==r2)         if (r1==r2)
  y = 1               y = 1
```

(read after write)

Is this valid?

Compiler Transformations

Irrelevant Read Introduction (Speculation)

```
if (r1==1) {          if (r1==1) {
  r2=x                  r2=x
  y=r2          →       y=r2          ⇄   r2=x
}                     } else r2=x         if (r1==1)
                                            y=r2
```

Is this valid?

Roach Motel Semantics

```
x=1                       lock m
lock m                      x=1
  y=1            ⟶         y=1
unlock m                  unlock m
```

Interestingly, this transformation is invalid in general

Compiler Transformations

Redundant Write-After-Read Elimination

initially x = 0

lock m1	lock m2	lock m1
x=2	x=1	lock m2
unlock m1	unlock m2	r1=x
		x=r1
		r2=x
		unlock m2
		unlock m1

Is this valid?

- No well-behaved execution contains a datarace
- The read "r2 = x" must always see the write "x=r1"
- Removing the redundant write allows r1 and r2 to see different values

Compiler Transformations

x = y = 0	
r1=x	r2=y
y=r1	if (r2==1) {
	r3=y
	x=r3
	} else x=1

Suppose we rewrite to r3=r2.
Can we now observe r1=r2=1?

Redundant read-after-read elimination

- Initially only one well-behaved execution, r1=r2=0
- Two dataraces:
  - between y=r1 and r2=y:
    - r2=0 is guaranteed
  - between r1=x and x=1 with value 1
    - now, commit datarace between y=r1 and r2=y with value 1
    - if r1=r2=1, then r3=y must read a value that happens-before it
      - there's no datarace between r3=y and y=r1
      - thus, the only value that it can read is y=0
      - then x=r3 must write 0: contradiction
- Transforming r3=y to r3=r2 allows r1=r2=1
  - commit the datarace betwen r1=x and x=1
  - then, commit the datarace between y=r1 and r2=y
  - can keep commitment to write x=1 since r1=r2=1

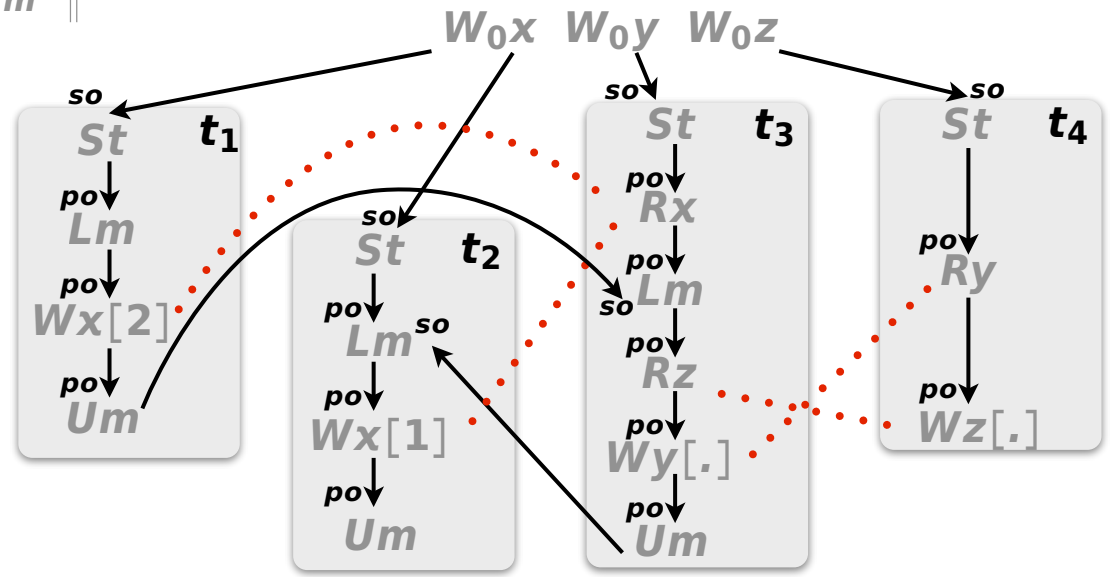
Compiler Transformations

$x \leftarrow 0; y \leftarrow 0; z \leftarrow 0$

lock m
$x \leftarrow 2$
unlock m
	lock m
	$x \leftarrow 1$
	unlock m
		$r_1 \leftarrow x$
		lock m
		$r_2 \leftarrow z$
		if($r_1 == 2$)
		$y \leftarrow 1$
		else
		$y \leftarrow r_2$
		unlock m
			$r_3 \leftarrow y$
			$z \leftarrow r_3$

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

Complex chain of logic says no!



Compiler Transformations

$x \leftarrow 0; y \leftarrow 0; z \leftarrow 0$

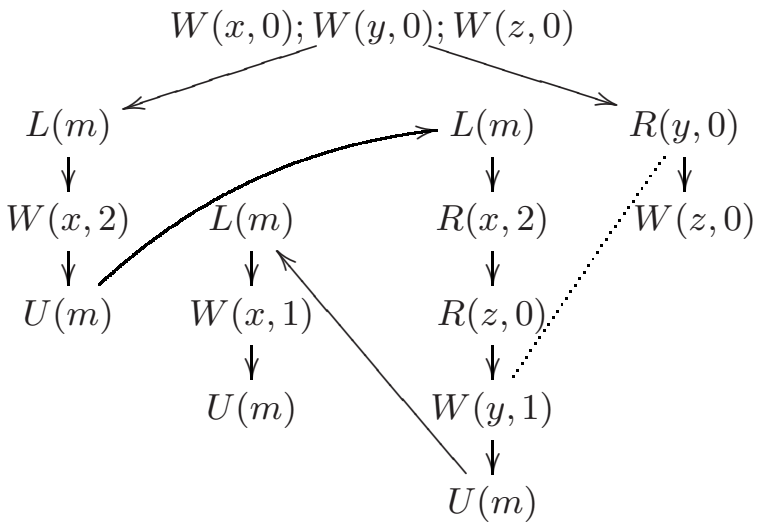
lock m
$x \leftarrow 2$
unlock m
	lock m
	$x \leftarrow 1$
	unlock m
		lock m
		$r_1 \leftarrow x$
		$r_2 \leftarrow z$
		if($r_1 == 2$)
		$y \leftarrow 1$
		else
		$y \leftarrow r_2$
		unlock m
			$r_3 \leftarrow y$
			$z \leftarrow r_3$

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

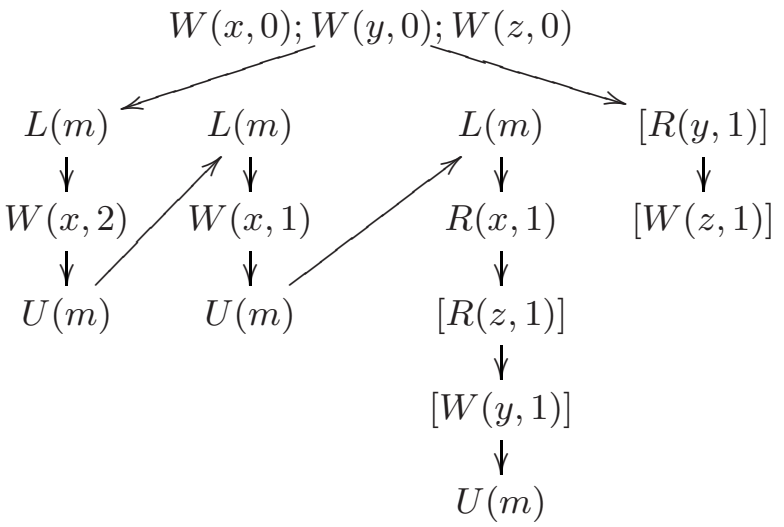
Swapping the lock and the load allows this execution!

Swapping offers more freedom for well-behaved executions:

- r1=x can see value 2 (without committing any action on x; no race)
- then, commit data race on y with value 1
- restart, commit data race on z with value 1
- restart, change synchronization order so that x=1 overwrites x=2



A.



B.