Why Threads Can't be a library
- and -
The Java Memory Model

CS 458
28 February 2007

Memory consistency models

- A memory model specifies the orders that reads and writes to memory by one processor must become available to itself and to other processors
- Are a requirement for multiprocessors

Sequential consistency (SC)

If X is listed before Y in program order, then the effects of X are guaranteed to have been committed when Y is executed

- This is intuitively appealing, but has some major problems
  - Hardware and compiler optimizations that were possible under uniprocessor SC are no longer possible (e.g., reordering, delayed writes)
  - We really only need SC semantics for synchronization purposes, so this restriction is overkill

Relaxed memory models (RM)

- Enforcing SC everywhere prevents lots of necessary compiler and hardware optimizations – it's too strong
- Relaxed memory models allow us to rollback the overly stringent requirements of SC, and make use of *safety nets* to enforce it when it's needed
- In addition to providing room for optimizations, relaxed memory models are good because they prevent the programmer from relying on subtle interactions of code for synchronization (possible under sequential consistency)
  - **Lesson** synchronization should be made explicit

Threads cannot be (safely) implemented as a library

TCBSIAAL

- Why not?
- The specifications for such libraries (e.g., pthreads) don't provide enough detail about what sorts of behavior is prohibited
- Because they (pthreads authors) didn't control the library and language specifications, they defined no memory model, instead insisting that a certain class of function calls would force memory consistency with respect to other threads
- The compiler is not sufficiently constrained because it doesn't know about threads
  - When does a data race exist? When can the compiler introduce one?

Example 1: concurrent modifications

(a)

```
if (x == 1) ++y;

if (y == 1) ++x;
```


(b)

```
++y; if (x != 1) --y;

++x; if (y != 1) --x;
```

- A compiler could do this
- The code is no longer free of a data race

Example 2: rewriting adjacent data

- Rewriting of adjacent data
- Data that is not synchronized gets rewritten because it is adjacent to synchronized data
  - bit fields, adjacent writes in structs
- Especially a potential problem for globals, which often get reordered

Example 3: register promotion

```
for (...) {
  ...
  if (mt) pthread_mutex_lock(...);
  x = ... x ...
  if (mt) pthread_mutex_unlock(...);
}
```

```
r = x;
for (...) {
  ...
  if (mt) {
    x = r; pthread_mutex_lock(...); r = x;
  }
  r = ... r ...
  if (mt) {
    x = r; pthread_mutex_unlock(...); r = x;
  }
}
x = r;
```

- Register promotion

Performance hits

- Threads-as-a-library is expensive
  - pthreads essentially allows optimizations that don't cross the explicit synchronization boundaries provided by its library calls
  - Each synchronization call is a very coarse safety net
- Certain algorithms don't require expensive locks
- Fine-grained use of atomic operations, as well as wait- and lock-free programming, are necessary for performance

Performance hit example

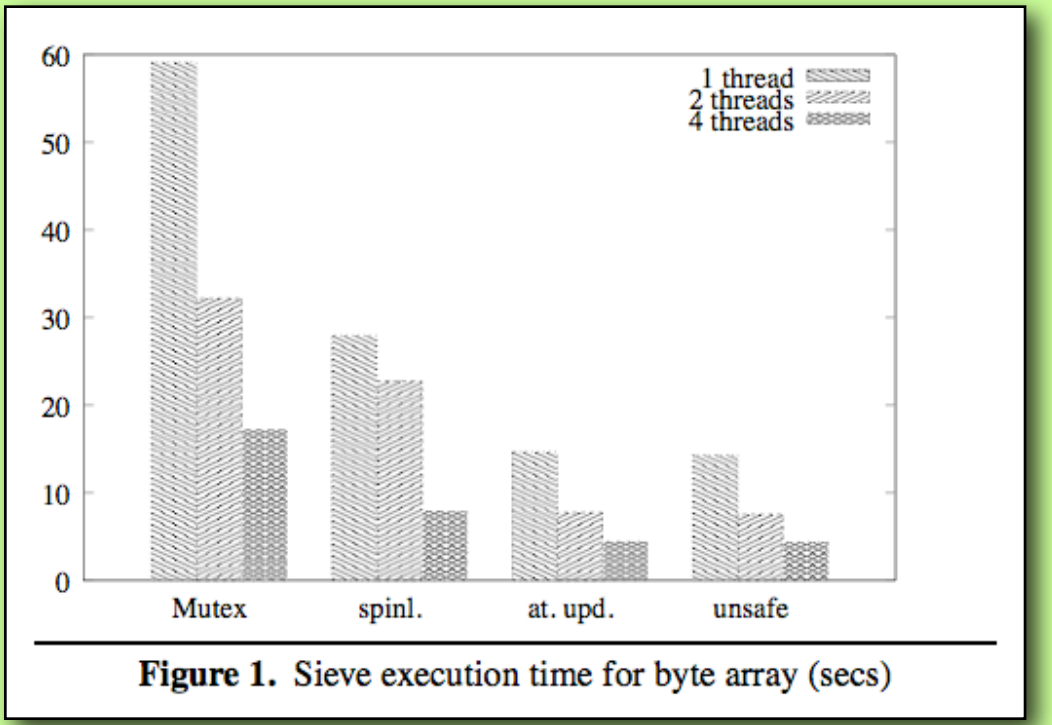


Figure 1. Sieve execution time for byte array (secs)

- the pthreads-consistent versions of this algorithm (stacks 1 & 2) are much slower

The Java memory model

The Java memory model

- The JMM is a relaxed memory model, so it allows for some of the reorderings and optimizations we need
- It guarantees the semantics of sequential consistency *for programs that are data-race-free*

Example

Initially, x == y == 0

Thread 1	Thread 2
1: r2 = x;	3: r1 = y
2: y = 1;	4: x = 2

r2 == 2, r1 == 1 violates sequential consistency.

- r2 == 2, r1 == 1 is not permitted under SC (why?)
- Why is this bad?
  - If it's not permitted, then basically no optimizations can be allowed
- How can we characterize this code to claim that it is bad?
  - We claim it is incorrectly (=not) synchronized

Definition of correct synch.

correct synchronization

A program is ***correctly synchronized*** iff all sequentially consistent executions of the program are free of data races

data race

Two memory accesses from separate threads, at least one of which is a write, and which are not ordered by happens-before

happens-before

The transitive closure of program order and synchronizes-with order

program order

the order that instructions are written in a program

synchronization order

A total ordering (consistent with program order) of synchronization actions for *each execution* of a program

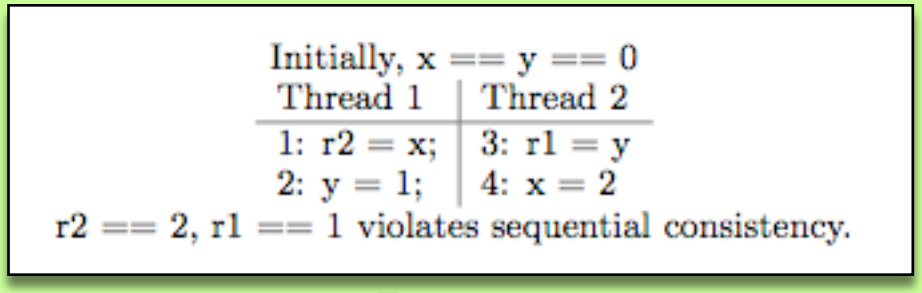
synchronizes-with order

a relationship in the synch. order between synchronization actions on the same lock or variable

synchronization action

lock, unlocks, and reads and writes to volatile variables

Example explained



Initially, x == y == 0

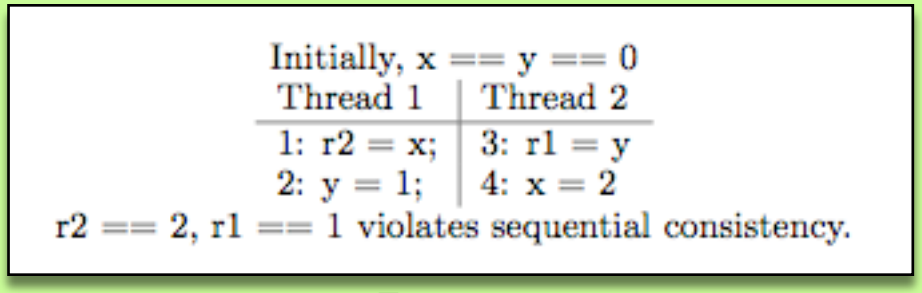
Thread 1	Thread 2
1: r2 = x;	3: r1 = y
2: y = 1;	4: x = 2

r2 == 2, r1 == 1 violates sequential consistency.

Figure 1

- By the definition on the previous slide, we see that this code is not correctly synchronized, because there is a data race
- Therefore the JMM would not guarantee sequential consistency

A further problem



Initially, x == y == 0

Thread 1	Thread 2
1: r2 = x;	3: r1 = y
2: y = 1;	4: x = 2

r2 == 2, r1 == 1 violates sequential consistency.

Figure 1

- Under SC, the only permissible results of this code are (r1,r2) ∈ {(0,0),(0,1),(2,0)} – *not* (2,1)
- What results are possible under other consistency models?
  - Any result from (r1,r2) ∈ {0,1} X {0,2} is possible

    (i.e., (0,0), (0,2), (1,0), (1,2))
- We can't allow just anything to happen in Java because it is supposed to be secure

A further problem (cont'd)

Initially, x == y == 0

Thread 1	Thread 2
1: r2 = x;	3: r1 = y
2: y = 1;	4: x = 2

r2 == 2, r1 == 1 violates sequential consistency.

Figure 1

Initially, x == y == 0

Thread 1	Thread 2
r1 = x;	r2 = y;
y = r1;	x = r2;

Incorrectly synchronized, but we want to disallow r1 == r2 == 42.

Figure 2: An Out Of Thin Air Result

Figure 2

- For example, consider Figure 2
- Both of these are **incorrectly synchronized** (meaning...)
  - They are not data-race-free in all SC executions of the code – there are data races on both *x* and *y*

A further problem (cont'd)

Initially, x == y == 0

Thread 1	Thread 2
1: r2 = x;	3: r1 = y
2: y = 1;	4: x = 2

r2 == 2, r1 == 1 violates sequential consistency.

Figure 1

Initially, x == y == 0

Thread 1	Thread 2
r1 = x;	r2 = y;
y = r1;	x = r2;

Incorrectly synchronized, but we want to disallow r1 == r2 == 42.

Figure 2: An Out Of Thin Air Result

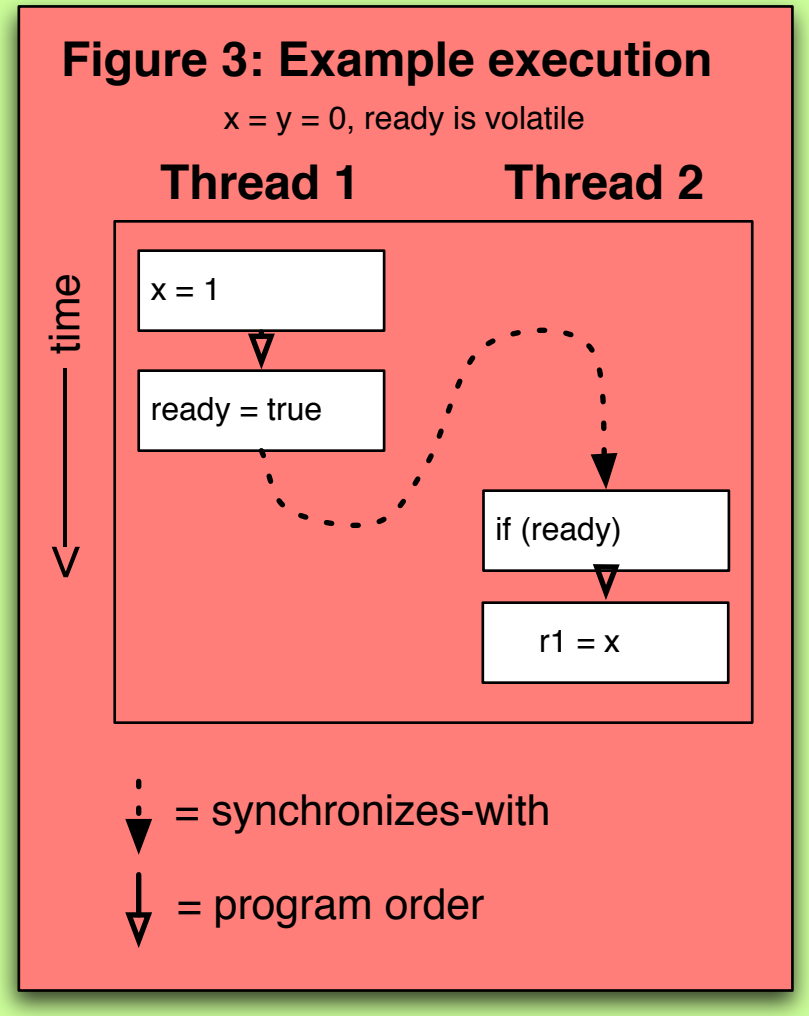
Figure 2

- In a language like C++, that's all that matters – the programmer messed up, so the language doesn't have to make any guarantees about what will happen
- Java must make security guarantees
  - such as preventing Figure 2 from predicting itself a reference to an object it's not supposed to see (an admittedly contrived example)

The Happens-before MM

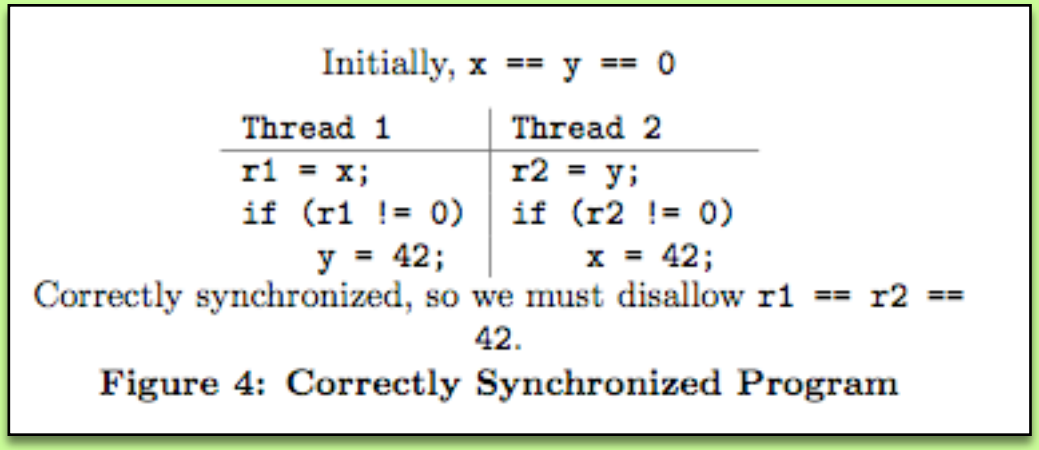
- Subtle formalization
- Important points
  - remember that synchronization order is a function of an execution that is consistent with program order
  - matched synch actions in this order have *synchronizes-with* edges between them
  - program order and synch-with edges together form the *happens-before* partial order
  - we have SC over this happens-before order

Synchronization order



- There is a total order over synchronization actions (reads and writes to *ready*) which produces a **synchronizes-with** edge from the write to the read of *ready*
- **happens-before** order is the transitive closure of program order and synchronizes-with order
- there is no path through the graph of this execution in which *r1* reads the old value of *x*
- notice that nothing would prevent us from reordering the write to *x* with, say, a write to *y* just above it

Problems with happens-before



Initially, x == y == 0

Thread 1	Thread 2
r1 = x;	r2 = y;
if (r1 != 0)	if (r2 != 0)
y = 42;	x = 42;

Correctly synchronized, so we must disallow r1 == r2 == 42.

Figure 4: Correctly Synchronized Program

- What's wrong with this?
  - Is correctly synchronized (definition?)
  - "all sequentially consistent executions exhibit no data races"
- However, we still have the problem from Figure 2 – the assignments (r1 = x, r2 = y) could predict the value 42 and then use the following code to justify that

Problems with happens-before

Initially, x == y == 0

Thread 1	Thread 2
r1 = x;	r2 = y;
y = r1;	x = r2;

Incorrectly synchronized, but we want to disallow r1 == r2 == 42.

Figure 2: An Out Of Thin Air Result

Initially, x == y == 0

Thread 1	Thread 2
r1 = x;	r2 = y;
if (r1 != 0)	if (r2 != 0)
y = 42;	x = 42;

Correctly synchronized, so we must disallow r1 == r2 == 42.

Figure 4: Correctly Synchronized Program

- So, happens-before does not guarantee sequential semantics when the code is data-race-free
- This is bad
- How can we characterize the problems in Figures 2 and 4, and adjust happens-before so that it prohibits these?

Causality

- In the illegal behavior in figures 2, and 4, the speculative writes caused subsequent code to justify those writes
- So apparently our issue is **causality** – we can just disallow this sort of self-justification
- These speculative writes were data or control dependent on the read that eventually justifies them (the `if` statement controlling their execution)
- Unfortunately, cause can't rely on notions of data or control dependence, for subtle reasons shown in Figures 5 & 6

Causality isn't dependence

Before compiler transformation

Initially, a = 0, b = 1

Thread 1	Thread 2
1: r1 = a;	5: r3 = b;
2: r2 = a;	6: a = r3;
3: if (r1 == r2)	
4: b = 2;	

Is r1 == r2 == r3 == 2 possible?

After compiler transformation

Initially, a = 0, b = 1

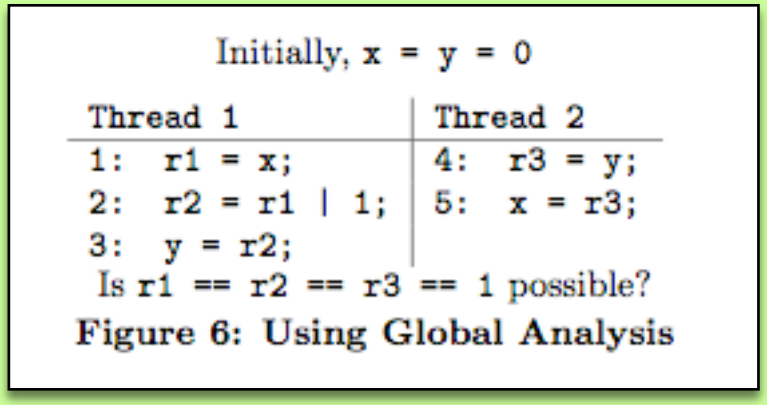
Thread 1	Thread 2
4: b = 2;	5: r3 = b;
1: r1 = a;	6: a = r3;
2: r2 = r1;	
3: if (true) ;	

r1 == r2 == r3 == 2 is sequentially consistent

Figure 5: Effects of Redundant Read Elimination

- Figure 5 is similar to Figures 2 & 4 (the write *b* = *2* seems to justify itself), but it should be permitted because a valid rewrite removes *b*'s dependences
- The "self-justifying behavior" is a function of the code itself (some sequentially consistent execution permits it), rather than of some predictive assignment

Another example



Initially, x = y = 0

Thread 1	Thread 2
1: r1 = x;	4: r3 = y;
2: r2 = r1 | 1;	5: x = r3;
3: y = r2;	

Is r1 == r2 == r3 == 1 possible?

Figure 6: Using Global Analysis

- The situation is the same in this example: global compiler optimizations eliminate data dependences by deducing that `r2` is always 1 in Thread 1
- The dependence of `r2` on `r1` can then be eliminated

Solution

PREVENT THIS

Initially, x == y == 0

Thread 1	Thread 2
r1 = x;	r2 = y;
if (r1 != 0)	if (r2 != 0)
y = 42;	x = 42;

Correctly synchronized, so we must disallow r1 == r2 == 42.

Figure 4: Correctly Synchronized Program

ALLOW THIS

Initially, a = 0, b = 1

Thread 1	Thread 2
4: b = 2;	5: r3 = b;
1: r1 = a;	6: a = r3;
2: r2 = r1;	
3: if (true) ;	

r1 == r2 == r3 == 2 is sequentially consistent

- What's different between these two?
- In Fig. 4 there is *no* SC execution in which `r1=r2=42`

Definition

Initially, x == y == 0

Thread 1	Thread 2
r1 = x;	r2 = y;
if (r1 != 0)	if (r2 != 0)
y = 42;	x = 42;

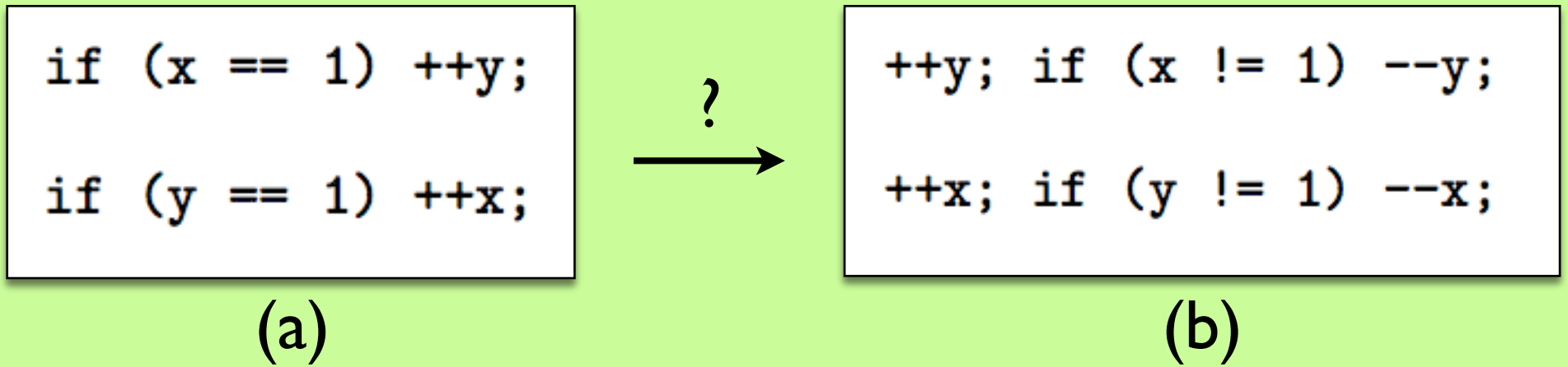
Correctly synchronized, so we must disallow r1 == r2 == 42.

Figure 4: Correctly Synchronized Program

- Using the notion of SC to justify rearrangements isn't quite enough (for some "subtle reasons")
- Crux: early execution (say of a write) is permitted "*if its occurrence is not dependent on a read returning a value from a data race.*"
- Optimizations cannot introduce data races

Examples

- From §4.1 of the Boehm paper (`x = y = 0` at start)



- Can we make the above transformation?
- No
  - The code in (a) is correctly synchronized
  - The transformation to (b) introduces two data races

Model summary

Model	Allows result from figure...					
	1	2	3*	4	5	6
sequential consistency	no	yes	yes	no	yes	yes
happens-before	yes	yes	yes	yes	no	no
Java memory model	yes	no	yes	no	yes	yes

* implicit desired behavior: `r1=x=1` iff Thread 2 executes after Thread 1

good bad

Implications

- Java implementors have to take care to implement the details of the model
- Java programmers need not worry about code transformations if their code is correctly synchronized