SWEN430 - Compiler Engineering

Lecture 22 - Memory Models II

David J. Pearce

School of Engineering and Computer Science
Victoria University of Wellington

Overview

Thread 1	Thread 2
`x = 1;`	`while(done == 0) { ... }`
`done = 1;`	`print(x);`

- **Q)** Can this program print `0` ?
- Programming language memory models determine the answer to this!

Overview

```
int x;
int done;
```

`x = 1;`	`while(done == 0) { ... }`
`done = 1;`	`print(x);`

Thread 1	Thread 2
`...`	`...`
`movl $1, x(%rip)`	`movl done(%rip), %eax`
`movl $1, done(%rip)`	`.L3:`
`...`	`testl %eax, %eax`
`...`	`je .L3`
`...`	`...`

- **Q)** *What's wrong with this translation?*

Volatile

If a field is declared as **`volatile`**, *any value written to it is flushed and made visible by the writer thread before the writer thread performs any further memory operation (i.e., for the purposes at hand it is flushed immediately). Reader threads must reload the values of volatile fields upon each access.*

– Concurrent Programming in Java, Doug Lea.

Example Reworked

```
int x;
volatile int done;
```

`x = 1;`	`while(done == 0) { ... }`
`done = 1;`	`print(x);`

Thread 1	Thread 2
`...`	`...`
`movl $1, x(%rip)`	`.L3`
`movl $1, done(%rip)`	`movl done(%rip), %eax`
`...`	`testl %eax, %eax`
`...`	`je .L3`
`...`	`...`

- On X86, **`volatile`** in C11 / Java forces reloading of variable.

Compiler Optimisations

```
int x;
int done;
```

Before Optimisation	After Optimisation
`while(done == 0) { }`	`print(x);`
`print(x);`	`while(done == 0) { }`

- Compiler permitted to *reorder* instructions to improve performance.
- Necessary to expose *Instruction-Level Parallelism (ILP*.
- Compiler reasoning about non-**`volatile`** variables is **single-threaded**.

Understanding Volatile (C11)

```
int x;
volatile int done;
```

Thread 1	Thread 2
`x = 1;`	`while(done == 0) { ... }`
`done = 1;`	`print(x);`

- Unfortunately, in C11 this program can still print `0`!
- This is because `volatile` is not *synchronizing*.
- Compiler free to **reorder** accesses between `x` and `done`!

Understanding Volatile (Java 5 or later)

```
int x;
volatile int done;
```

Thread 1	Thread 2
`x = 1;`	`while(done == 0) { ... }`
`done = 1;`	`print(x);`

- In Java 5 (or later), this program cannot print `0`!
- This is because `volatile` is *synchronizing*.

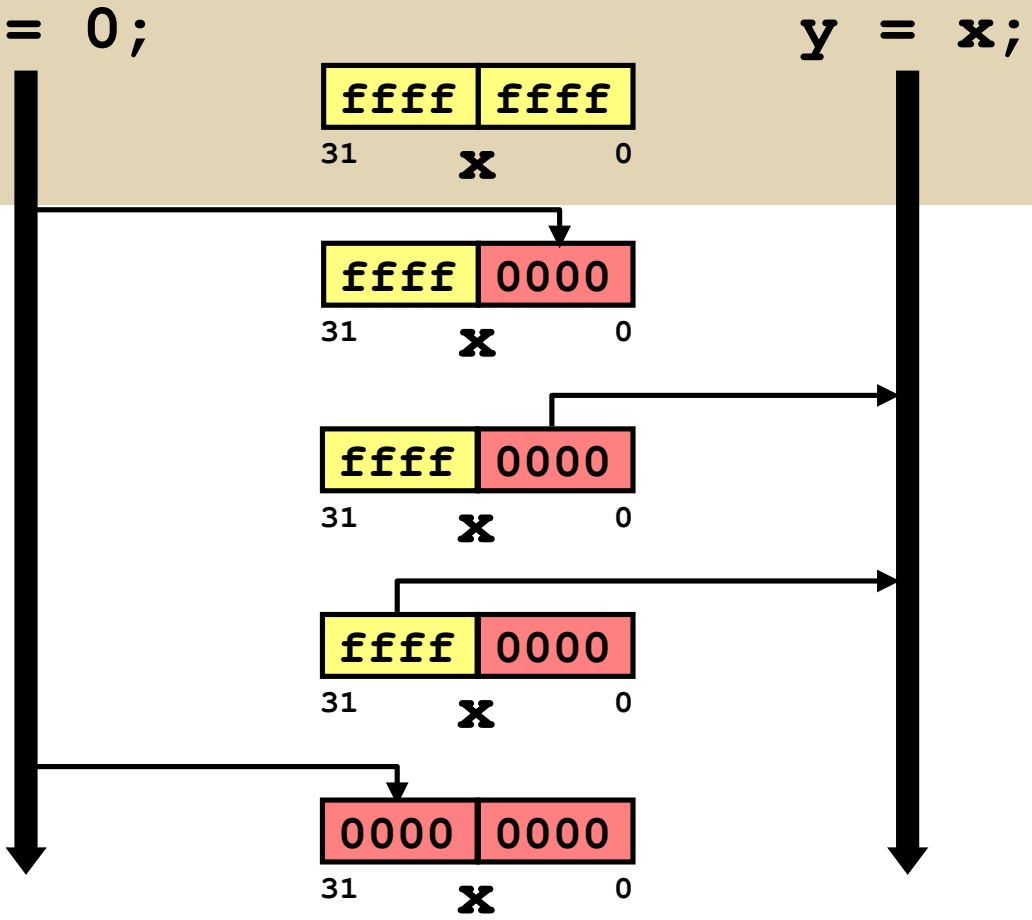
Data Races

```
int done;
```

Thread 1	Thread 2
`done = 1;`	`while(done) { ... }`

- Thread 1 might write `done` at same time Thread 2 reads it!
- This is a *data race* — two threads are “racing” for access.
- Data races always involve *at least one write*.

Data Races (Cont'd)



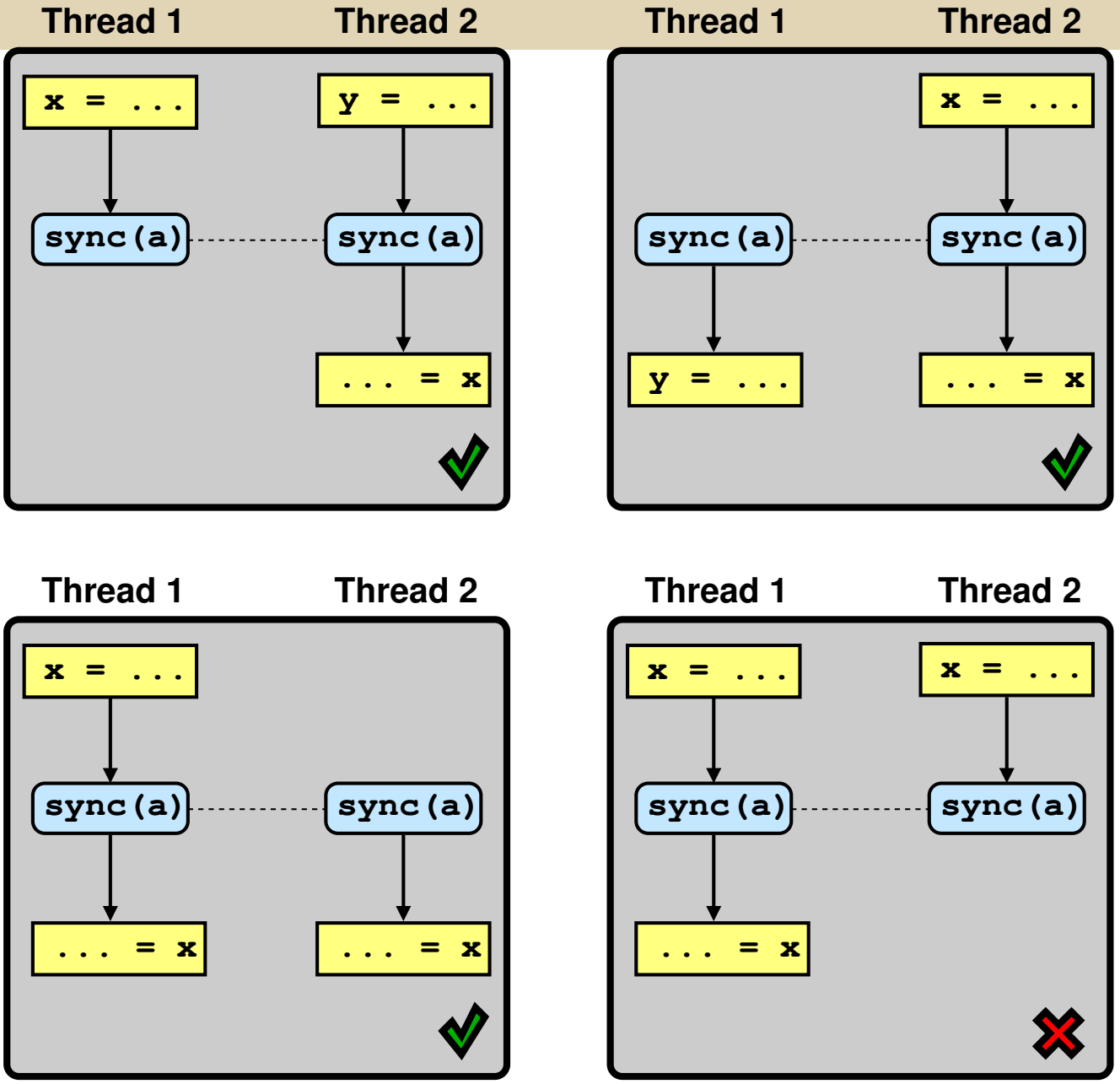
- Above is a *torn write*, where `y` holds `-65536` afterwards!
- Could easily happen on a 16-bit machine.
- Could also happen on 32-bit machines depending on hardware!

Data-Race-Free Sequential Consistency (DRF-SC)

... today's processors guarantee a property called "data-race-free sequential-consistency", or DRF-SC (sometimes also written SC-DRF). A system guaranteeing DRF-SC must define specific instructions called **synchronizing instructions**, *which provide a way to coordinate different processors (equivalently, threads). Programs use those instructions to create a* **"happens before"** *relationship between code running on one processor and code running on another.*

– Cox'21

DRF-SC (Cont'd)


Thread 1
Thread 2
x = ...
y = ...
sync(a)
sync(a)
... = x
Thread 1
Thread 2
x = ...
sync(a)
sync(a)
y = ...
... = x
Thread 1
Thread 2
x = ...
sync(a)
sync(a)
... = x
... = x
Thread 1
Thread 2
x = ...
x = ...
sync(a)
sync(a)
... = x

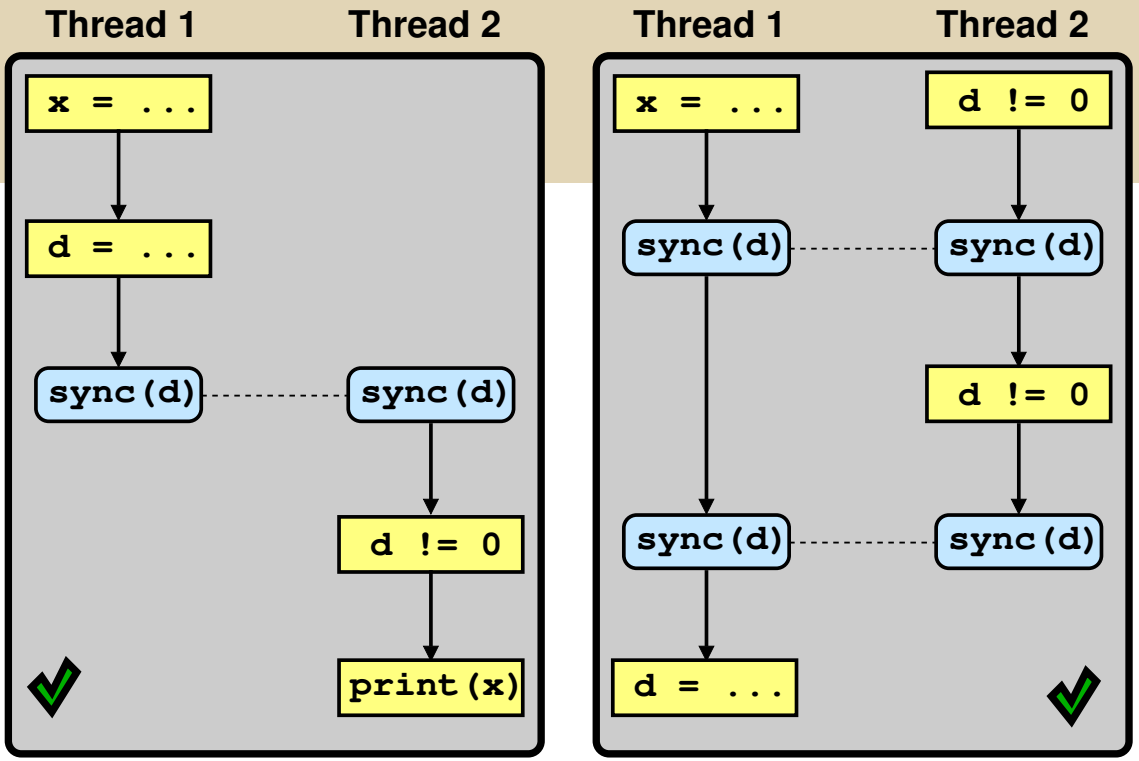
The Java Memory Model

Synchronisation Points

```
public class Buffer<T> {
  synchronized void write(T item) { ... }
  synchronized T read() { ... }
}
```

- Thread creation **happens before** thread begins execution
- Unlocking `m` **happens before** any subsequent lock of `m`
- Writing `volatile` variable `v` **happens before** any subsequent read of `v`

Happens Before




```
int x;
volatile int done;
```

```
x = 1;                          while(done == 0) { ... }
done = 1;                       print(x);
```

Out of Thin Air!

```
int x, y;
int r1, r2;
```

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

Q) Can this program see `r1 = 42` and `r2 = 42` together?

A) Probably not, *but the Java Memory Model does not rule it out!*

(here, `42` is called an *out-of-thin-air* value)

References

- **Programming Language Memory Models**, Russ Cox, 2021. `https://research.swtch.com/plmm`
- **Concurrent Programming in Java: Design Principles and Patterns**, Doug Lea, 1999.
- **The Java Memory Model**, Jeremy Manson, Bill Pugh and Sarita Adve, POPL, 2005.
- **Threads Cannot be Implemented As a Library**, Hans-J. Boehm, PLDI, 2005.