CONCURRENCY:
SEQUENTIAL CONSISTENCY, DATA RACES, AND DYNAMIC ANALYSES

Lecture by Rohan Padhye

17-355/17-665/17-819: Program Analysis

Material from past lectures by Jonathan Aldrich, based in large part on slides by John Erickson, Stephen Freund, Madan Musuvathi, Mike Bond, and Man Cao

Lecture Goals

- What is sequential consistency and why is it important?
- What is a data race, and what is data-race-free execution?
- Subtleties of data races and memory models
  - Why taking advantage of “harmless races” is almost certainly a bad idea
- Lockset analysis for data race detection
- Happens-before based data race detection
  - And high performance implementations, e.g. as in FastTrack

SEQUENTIAL CONSISTENCY

First things First
Assigning Semantics to Concurrent Programs

int X = F = 0;

```
X = 1;
F = 1;
```

```
t = F;
u = X;
```

- What does this program mean?

- Sequential Consistency [Lamport '79]

  Program behavior = set of its thread interleavings

Recall: Semantics of WHILE$_{||}$ from midterm

$$\frac{\langle E, S_1 \rangle \rightarrow \langle E', S_1' \rangle}{\langle E, S_1; S_2 \rangle \rightarrow \langle E', S_1'; S_2 \rangle} \; \textit{small-seq-congruence}$$

$$\frac{}{\langle E, \texttt{skip}; S_2 \rangle \rightarrow \langle E, S_2 \rangle} \; \textit{small-seq}$$

$$\frac{\langle E, S_1 \rangle \rightarrow \langle E', S_1' \rangle}{\langle E, S_1 \parallel S_2 \rangle \rightarrow \langle E', S_1' \parallel S_2 \rangle} \; \textit{small-par-congruence-1}$$

$$\frac{\langle E, S_2 \rangle \rightarrow \langle E', S_2' \rangle}{\langle E, S_1 \parallel S_2 \rangle \rightarrow \langle E', S_1 \parallel S_2' \rangle} \; \textit{small-par-congruence-2}$$

$$\frac{}{\langle E, \texttt{skip} \parallel \texttt{skip} \rangle \rightarrow \langle E, \texttt{skip} \rangle} \; \textit{small-par-skip}$$

Exercise 1:

int X = F = 0;

```
X = 1;
F = 1;
```

```
t = F;
u = X;
```

- What are the possible final values for variables `t` and `u` after running this program, assuming sequential consistency?

Sequential Consistency Explained

int X = F = 0; // F = 1 implies X is initialized

```
X = 1;
F = 1;
```

```
t = F;
u = X;
```

X = 1;	X = 1;	X = 1;	t = F;	t = F;	t = F;
F = 1;	t = F;	t = F;	u = X;	X = 1;	X = 1;
t = F;	F = 1;	u = X;	X = 1;	u = X;	F = 1;
u = X;	u = X;	F = 1;	F = 1;	F = 1;	u = X;
t=1, u=1	t=0, u=1	t=0, u=1	t=0, u=0	t=0, u=1	t=0, u=1

t=1 implies u=1

Naturalness of Sequential Consistency

- Sequential Consistency provides two crucial abstractions
- Program Order Abstraction
  - Instructions execute in the order specified in the program

A ; B

means "Execute A and then B"

- Shared Memory Abstraction
  - Memory behaves as a global array, with reads and writes done immediately
- We implicitly assume these abstractions for sequential programs
  - As we will see, we can only rely on these abstractions under certain conditions in a concurrent context

WHAT IS A DATA RACE ?

- The term “data race” is often overloaded to mean different things
- Precise definition is important in designing a tool

Data Race

- Two accesses *conflict* if
  - they access the same memory location, and
  - at least one of them is a write

Write X – Write X

Write X – Read X

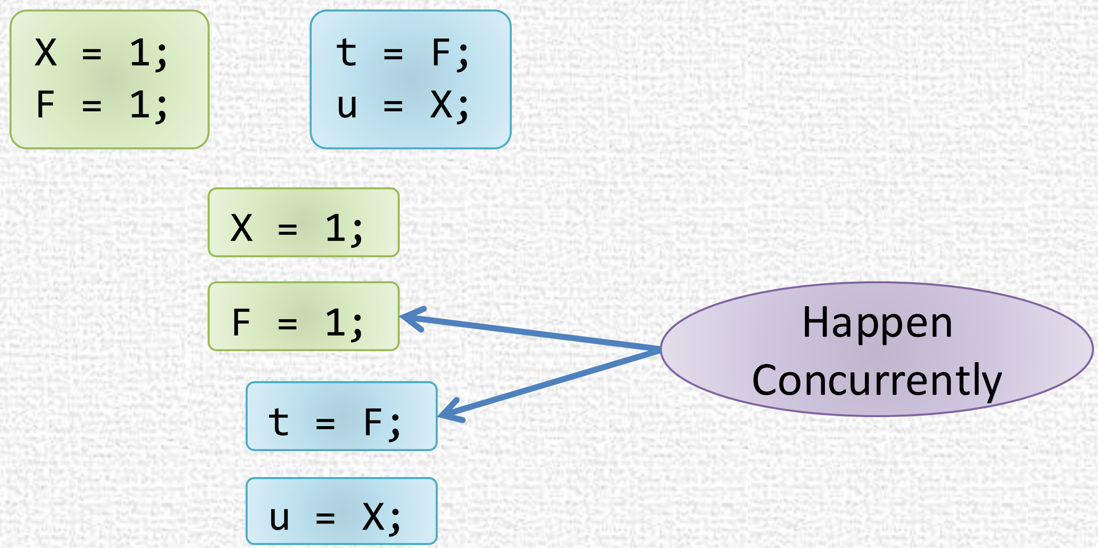
Read X – Write X

Read X – Read X

- A data race is a pair of conflicting accesses that happen concurrently

“Happen Concurrently”

- A and B happen concurrently if
- there exists a sequentially consistent execution in which they happen one after the other

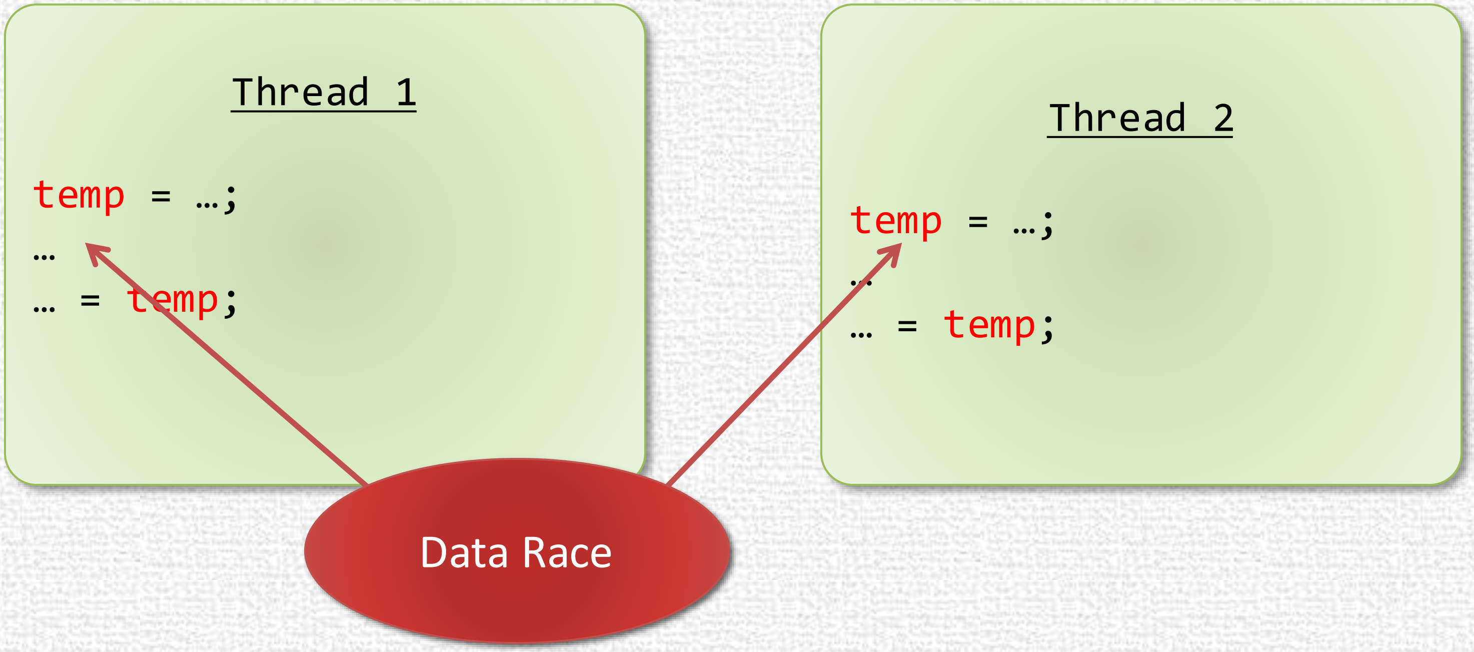


Data races are almost always no good

- What are some consequences of a data race, even when assuming sequential consistency?

Unintended Sharing

- Threads accidentally sharing data that should not be global
- *Solution*: Change allocation (e.g., stack var or static thread-local)



Atomicity Violation

- When code that is meant to execute *atomically* (that is, perform a single undivisible operation) suffers interference from some other thread
- *Solution*: Surround critical sections with locks

Thread 1

```
void Bank::Update(int a)
{
    int t = bal;
    bal = t + a;
}
```

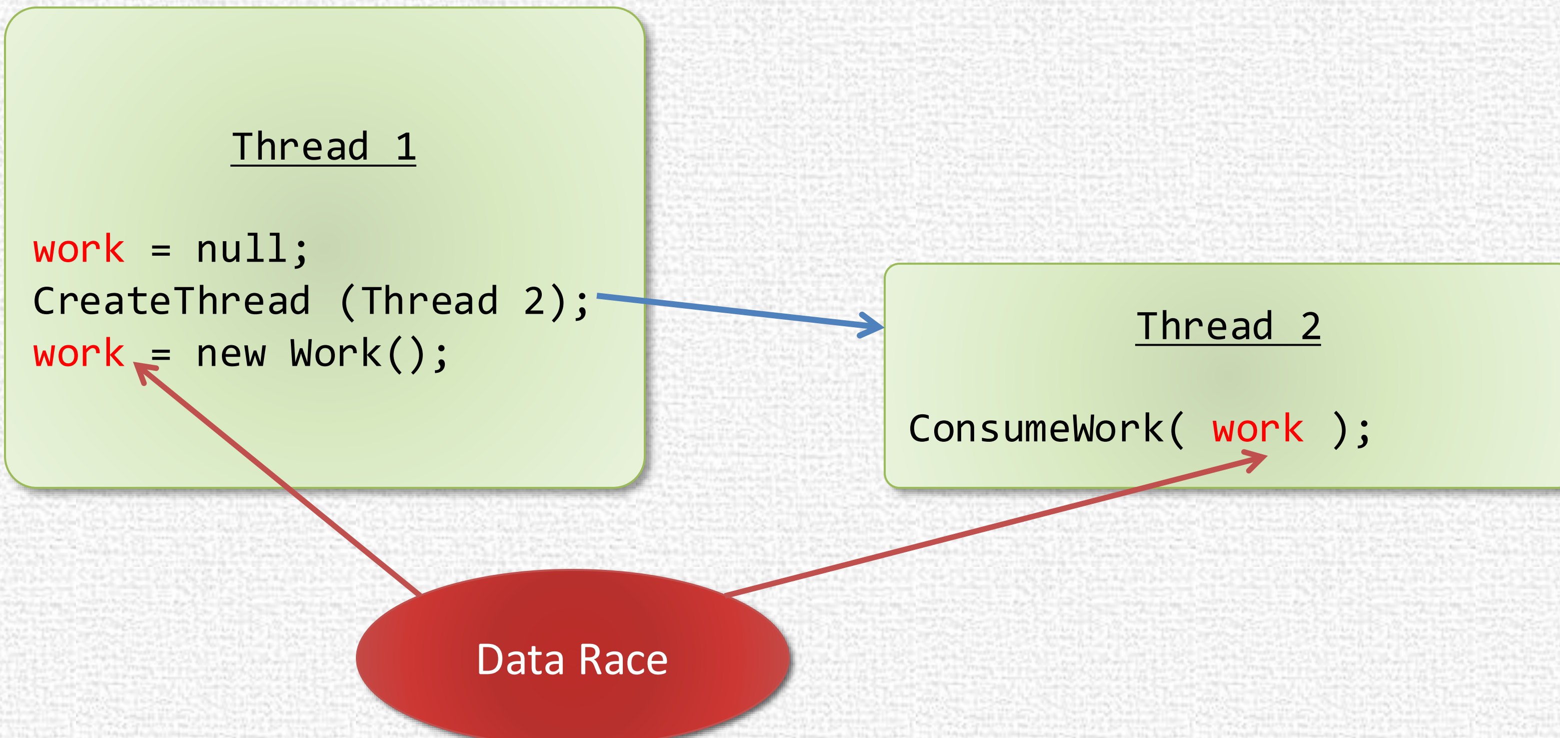
Thread 2

```
void Bank::Withdraw(int a)
{
    int t = bal;
    bal = t - a;
}
```

Data Race

Ordering Violation

- Incorrect signaling between a producer and a consumer
- *Solution*: Reorder operations or use synchronization (e.g., signals)



But,....

- How do you think ”locks” are implemented?
- Atomic compare-and-swap (CAS)

```
AcquireLock(lock){
  while (!CAS (lock, 0, 1)) {}
}
```

```
ReleaseLock(lock) {
    lock = 0;
}
```

Data Race ?

Acceptable Concurrent Conflicting Accesses

- Implementing synchronization (such as locks) usually requires concurrent conflicting accesses to shared memory

- Innovative uses of shared memory
  - Fast reads
  - Double-checked locking
  - Lazy initialization
  - Setting dirty flag
  - …

- Need mechanisms to distinguish these from erroneous conflicts

Solution: Programmer Annotation

- Programmer explicitly annotates variables as “synchronization”
  - Java – volatile keyword
  - C++ – std::atomic<> types

Data Race

- Two accesses *conflict* if
  - they access the same memory location, and
  - at least one of them is a write
- A data race is a pair of concurrent conflicting accesses to locations not annotated as synchronization
  - Recall: "Concurrent" means there exists a sequentially consistent execution in which they happen one after the other
- Equivalent definition: a pair of conflicting accesses where one doesn't happen before the other
  - Program order
  - Synchronization order
    - Acquire/release, wait-notify, fork-join, volatile read/write

Exercise 2: Is there a data race? If so, on what variable(s)?

Initially:

```
int data = 0;
boolean flag = false;
```

T1:

```
data = 42;
flag = true;
```

T2:

```
if (flag)
  t = data;
```

Is there a data race?

Initially:

```
int data = 0;
boolean flag = false;
```

T1:

```
data = 42;
flag = true;
```

T2:

```
if (flag)
  t = data;
```

Consider regular compiler transformations/optimizations

<u>Before</u>:

```
data = 42;
flag = true;
```

<u>After</u>:

```
flag = true;
data = 42;
```

Possible behavior

Initially:

```
int data = 0;
boolean flag = false;
```

T1:

```
flag = true;
```

T2:

```
if (flag)
  t = data;
```

```
data = 42;
```

Consider regular compiler transformations/optimizations

Before:

```
if (flag)
  t = data;
```

After:

```
t2 = data;
if (flag)
  t = t2;
```

Possible behavior

Initially:

```
int data = 0;
boolean flag = false;
```

T1:

```
data = 42;
flag = true;
```

T2:

```
t2 = data;

if (flag)
  t = t2;
```

How do we fix this?

Initially:

```
int data = 0;
boolean flag = false;
```

T1:

```
data = 42;
flag = true;
```

T2:

```
if (flag)
  t = data;
```

Using "synchronized" keyword in Java

Initially:

```
int data = 0;
boolean flag = false;
```

T1:

```
data = ...;
synchronized (m) {
  flag = true;
}
```

T2:

```
boolean f;
synchronized (m) {
  f = flag;
}
if (f)
  ... = data;
```

… Implemented via locks

Initially:

```
int data = 0;
boolean flag = false;
```

T1:

```
data = ...;
acquire(m);
  flag = true;
release(m);
```

Happens-before relationship

T2:

```
boolean f;
acquire(m);
  f = flag;
release(m);
if (f)
  ... = data;
```

Using “volatile” keyword in Java

Initially:

```
int data = 0;
volatile boolean flag = false;
```

T1:

```
data = ...;
flag = true;
```

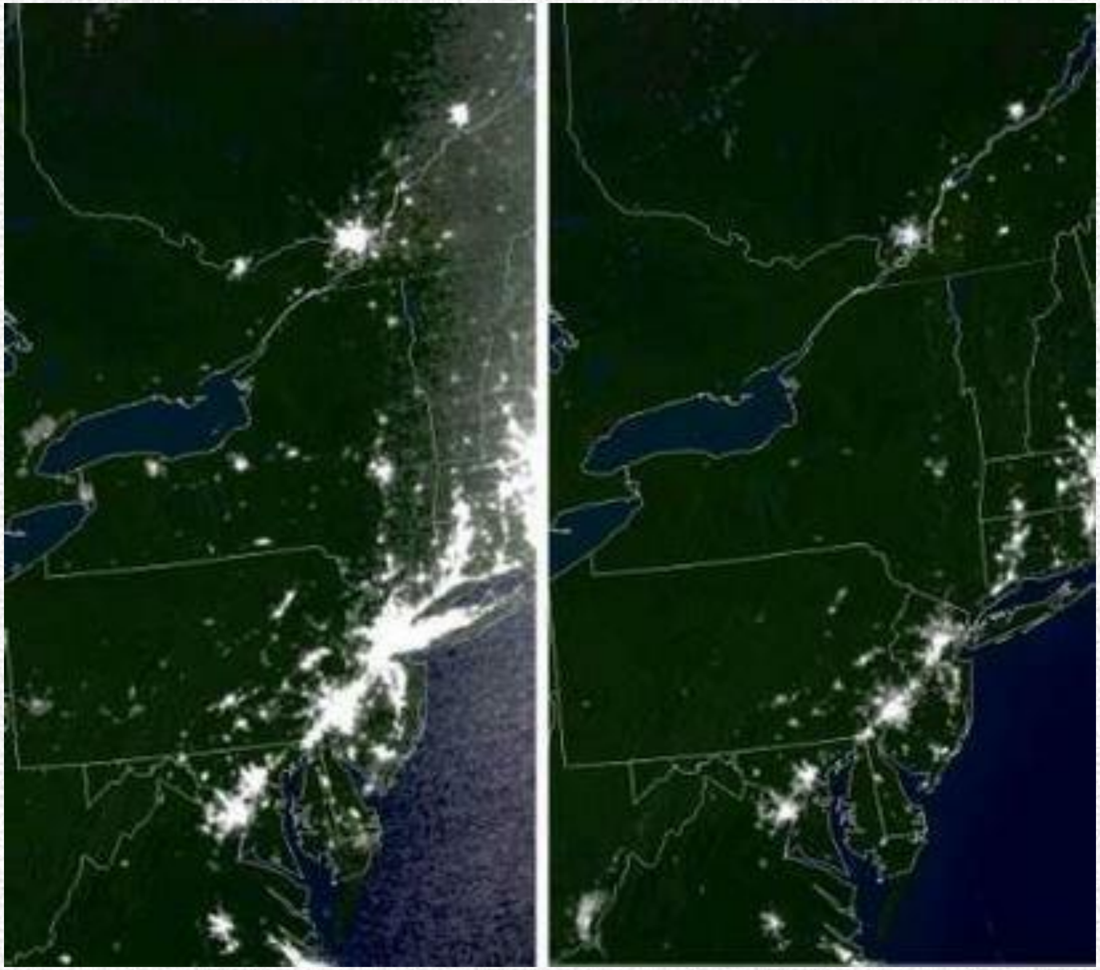
Happens-before relationship

T2:

```
if (flag)
  ... = data;
```

Data Race vs Race Conditions

- Data Races != Race Conditions
  - Confusing terminology
- Race Condition
  - Any timing error in the program
  - Due to events, device interaction, thread interleaving, …
  - Race conditions can be very bad!



Data Race vs Race Conditions

- Data Races != Race Conditions
  - Confusing terminology

- Race Condition
  - Any timing error in the program
  - Due to events, device interaction, thread interleaving, …
  - Race conditions can be very bad!

- Data races are neither sufficient nor necessary for a race condition
  - Data race is a good symptom for a race condition

DATA-RACE-FREEDOM SIMPLIFIES LANGUAGE SEMANTICS

Advantage of Eliminating All Data Races

- Defining semantics for concurrent programs becomes surprisingly easy

- In the presence of compiler and hardware optimizations

Can A Compiler Do This?

```
L1: t = X*5;
L2: u = Y;
L3: v = X*5;
```

→

```
L1: t = X*5;
L2: u = Y;
L3: v = t;
```

OK for sequential programs
if X is not modified between L1 and L3

t,u,v are local variables
X,Y are possibly shared

Can Break Sequential Consistent Semantics

Init: X = Y = 0;

```
L1: t = X*5;
L2: u = Y;
L3: v = X*5;
```

```
M1: X = 1;
M2: Y = 1;
```

u == 1 ➔ v == 5

Data Race

Init: X = Y = 0;

```
L1: t = X*5;
L2: u = Y;
L3: v = t;
```

```
M1: X = 1;
M2: Y = 1;
```

possibly u == 1 && v == 0

Can A Compiler Do This?

```
L1: t = X*5;
L2: u = Y;
L3: v = X*5;
```

```
L1: t = X*5;
L2: u = Y;
L3: v = t;
```

OK for sequential programs
if X is not modified between L1 and L3

OK for concurrent programs
if there is no data race on X or
if there is no data race on Y

t,u,v are local variables
X,Y are possibly shared

Key Observation [Adve& Hill '90]

- Many sequentially valid (compiler & hardware) transformations also preserve sequential consistency
- Provided the program is data-race free

- Forms the basis for modern C++, Java semantics

data-race-free → sequential consistency

otherwise → weak/undefined semantics