Lec 7

Race Conditions
+
Locks

Reasoning about sequential code

```
⋮
int x = 42
⋮ // code that doesn't change x
print(x)
⋮
x++;
⋮
print(x)
```

Facts updated when variable modified

Reasoning about threads

initially, $x = 0$

Thread 1

```
main() {
    x = 1
    print(x)
}
```

Thread 2

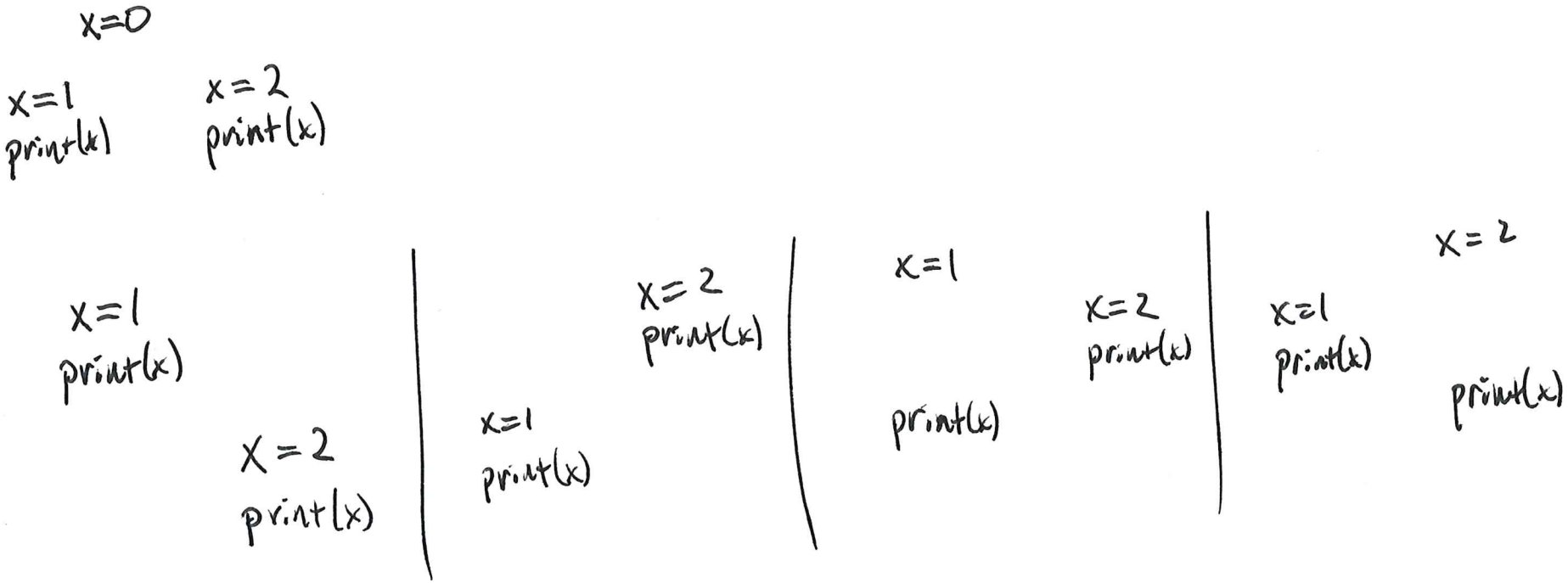
```
main() {
    x = 2
    print(x)
}
```

what are the possible outputs of this program?

1	2	1	2
2	1	1	2

Sequential Consistency (SC)

An execution of a multithreaded program is an interleaving of the operations of its threads



Reasoning about interleavings

- SC gives us one way to reason:
  enumerate all interleavings and check each one
- unfortunately there are at least 2^n interleavings of two threads with n operations
  - so this only works for tiny programs

Reasoning about interference

- a better way to think about multithreaded programs

"how can another thread change relevant state"

x = 0

Thread 1	Thread 2
⋮	⋮
print (x)	print (x)
⋮	⋮
inc x	inc x
⋮	⋮
print (x)	print (x)

- each thread knows x only increases
- "stable properties" ←

The truth about sequential consistency:

it's all a big lie

- need to know what operations are atomic
  - e.g. "x++" is more like "$\frac{t = x}{x = t+1}$"

- compilers reorder instructions for performance
  - but only guarantee equivalence for single threaded

- processors reorder instructions
  - details vary by CPU and are complicated
  - write buffering

→ **SC leads to wrong conclusions**

"DRF ⇒ SC" data-race freedom ⇒ sequential ~~consistency~~ consistency

A **race condition** is any situation where program behavior depends on thread schedule

initially $x = 0$

Thread 1	Thread 2
$x = 1$	$x = 2$

final value of x?

Too much milk

Roommate 1	Roommate 2
Look in fridge; no milk	Look in fridge; no milk
Leave for store	Leave for store
Arrive at store	Arrive at store
Buy milk	Buy milk
Arrive home; store milk	Arrive at home; store milk
	Oh no!

Safety: At most 1 person buys milk

Liveness: If milk is needed, someone eventually buys it

Non-solution 1: Leave a note

```
if (milk == 0) {
    if (note == 0) {
        note = 1
        milk ++
        note = 0
    }
}
```

R 1

```
if (milk == 0) {

if (note == 0) {
    :
```

R 2

```
if (milk == 0) {
    if (note == 0) {
        note = 1
        milk ++
        note = 0
    }
}
```

Non-solution 2: Check note after leaving note
Check milk after leaving note

R1

```
note1 = 1
if(note2 == 0) {
    if (milk == 0) {
        milk ++
    }
}
note1 = 0
```

R2

```
note2 = 1
if(note1 == 0) {
    if (milk == 0) {
        milk ++
    }
}
note2 = 0
```

Safe: each roommate decides to buy milk only if the other is not trying.

not live: both roommates leave a note, then see the other's and decide not to buy milk

Solution 3: one thread checks if other bought milk

R1

```
note1 = 1
while (note2 == 1) {}
if (milk == 0) {
    milk ++
}
note1 = 0
```

R2

```
note2 = 1
if (note1 == 0) {
    if (milk == 0) {
        milk ++
    }
}
note 2 = 0
```

Safe: as before

live: R1 waits until its unambiguous whether R2 bought milk

(see: Peterson's Algorithm)

Real Solution: use Locks!

- a lock is either Busy or Free
- a lock is initially Free
- acquire() waits until lock is Free
  then atomically sets it to Busy
- release() sets the lock to Free

```
lock.acquire()
if (milk ==0)
    milk ++
lock.release()
```