Synchronization

Last time: Threads are powerful... but tricky.
⇒ need synchronization. (locks : mutexes)
⇒ guard shared mutable threads with mutex.

How to communicate between threads?

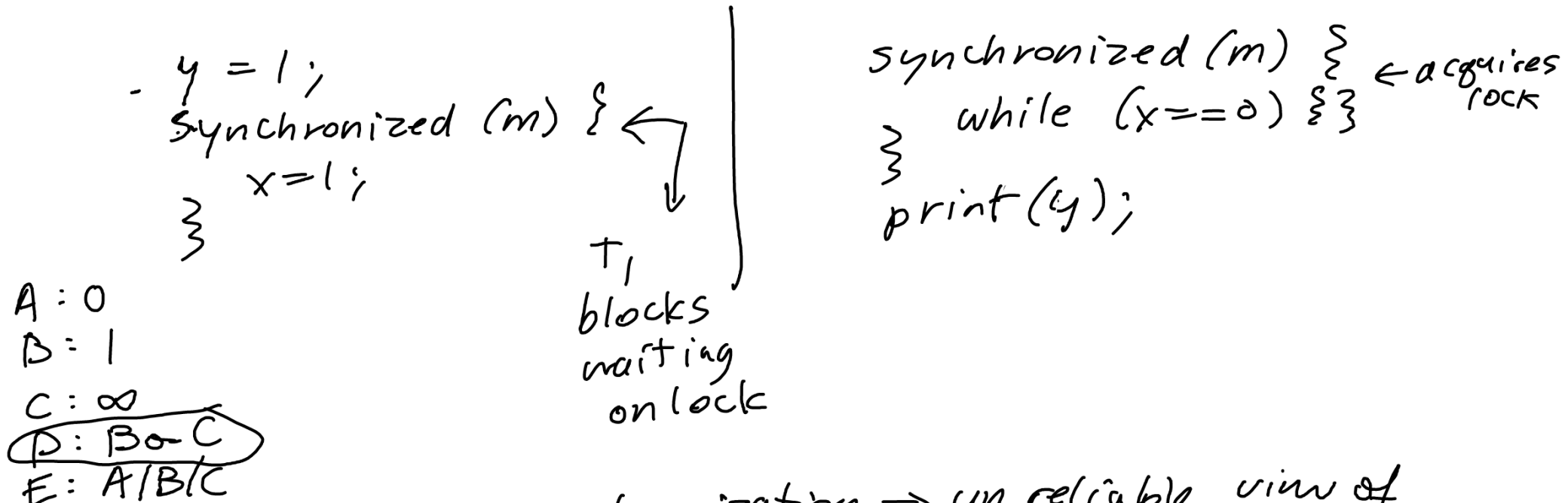
Ex/ Want thread T_2 to wait until T_1 changes x

x = y = 0 initially.

T_1	T_2
y = 1;	while (x == 0) { }
x = 1;	print(y);

A : 0
B : 1
C : nothing ← x86
D : B or C
E : A or B or C ← ARM

Updates to memory by T_1 are not guaranteed to be seen by T_2 unless synchronization dependency



A : 0
B : 1
C : ∞
D : B or C
E : A/B/C

- No synchronization ⇒ unreliable view of memory
- Need something more to communicate...

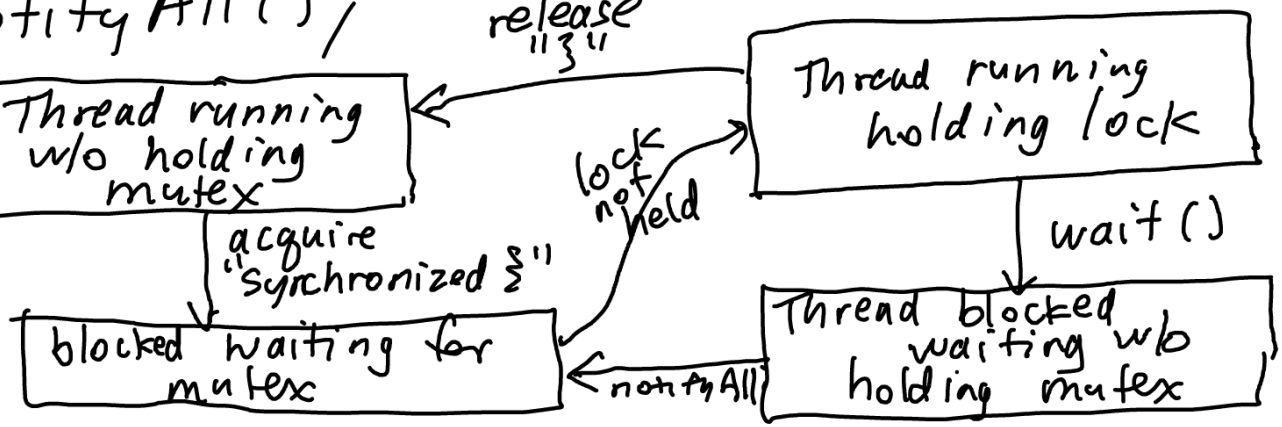
Condition Variables

- Allows thread to block until some condition is true.
- Always tied to mutex that guards shared state used by condition.
- Every Java object has mutex and condition variable.

Operations

```
/** Effect: block this thread and release mutex.
    Requires: mutex held by this thread */
void wait();
/** Effect: unblock all threads waiting on CV.
    Requires: mutex held. */
void notifyAll();
```

State diagram



To use:

wait until a condition is true? Always use a loop:

```
while (! condition) wait();
```

wakeup-waiting race

⇒ first thread to acquire mutex can invalidate condition.

⇒ rest of threads wake up w/ condition false.

Monitors

Pattern for concurrent programming.

- object state guarded by object's mutex
- all public methods "synchronized"

⇒ can't access ivars w/o getting mutex.

- use condition variables for blocking abstractions

Barriers

High-level synchronization mechanism

- spawn N threads & wait for all N to finish.
- updates <u>before</u> barrier seen by all threads <u>after</u> barrier

Java: `b = new CyclicBarrier(N);`

<u>blocking abstraction</u>

```
class Barrier {
    int awaits;
    Barrier(int N) { awaits = N; }

    synchronized void await() {
        awaits--;
        if (awaits == 0) notifyAll();
        while (awaits > 0) wait();
    }
}
```

Deadlocks

What if everything is a monitor?

Problem: all threads can block.

Ex/ monitors a, b

```
a.f() {
    ⋮
    b.g();
    ⋮
}
```

```
b.g() {
    ⋮
    a.f()   ← illegal
    ⋮
}
```

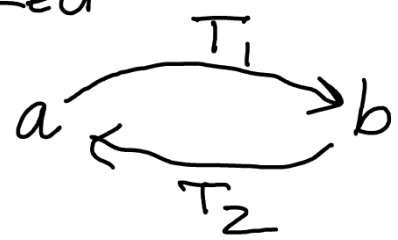
a < b

T_1: a.f() ⇝ (a locked) ⇝ b.g() ← blocks!

T_2: b.g() ⇝ (b locked) ⇝ a.f() ← blocks!

deadlock

Acquisition order graph



cycle in graph ⇒ possible deadlock

Solution: lock ordering (acyclic)

if $a < b$, cannot acquire a after b.

lock level: "highest" acquired lock in lock order

monitor precondition: lock level $\leq$ this
$\Rightarrow$ no deadlocks.