Poll Everywhere

PollEv.com/javabear text **javabear** to **22333**

Which of the following is *not* a benefit of concurrency?

- It helps speed up complex calculations on large data sets. **(A)**
- It allows us to better utilize modern computer hardware. **(B)**
- It improves the user experience in graphical applications. **(C)**
- It helps avoid bugs in complex computations. actually, more subtle bugs become possible **(D)**

Lecture 27: Synchronization

CS 2110

April 30, 2026

Today's Learning Outcomes

106. Explain the semantics of locks in Java and write code involving synchronization.

107. Describe the conditions that can cause deadlock in a concurrent program and how it can be avoided.

108. Explain how condition variables can be used to synchronize modifications of a shared resource.

Where We Left Off: Race Conditions

Multiple threads access same variables during same time window, and at least one writes to the variable.

Different interleavings of machine instructions lead to different results.

i.x++ → compiles to
- load i.x into CPU register
- add 1 to that register
- store register val to i.x

load	→ 0	load	→ 0
add	0+1=1	load	→ 0
store	1 ←	add	0+1=1
load	→ 1	Store	1 ←
add	1+1=2	add	0+1=1
store	2 ←	store	1 ←

```
class SharedInt { int x = 0; }

static void increment(SharedInt i) {  i.x++; }

public static void main(String[] args) throws ... {
   SharedInt s = new SharedInt(); // shared variable
   Thread t1 = new Thread(() -> increment(s));
   Thread t2 = new Thread(() -> increment(s));

   t1.start();  t2.start();  // start the threads
   t1.join();  t2.join();    // wait for threads to stop

   System.out.println("Final value of s.x: " + s.x);
}
```

Critical Sections

Sometimes, multiple instructions must be executed sequentially to properly update a shared variable without the possibility of another thread messing with that variable.

{ load, add, store } | Treating these as an atomic operation ensures the variable's state accurately reflects the update we wanted to perform

Atomic operations can prevent race conditions

How do we achieve this?

1. Write a single-thread application
2. Eliminate shared state: each variable can be written to by only one thread

} lose many performance benefits of concurrency

3. Synchronize: have threads cooperate to ensure they don't try to update variable at the same time

Mutexes and Synchronized Blocks

"one at a time"

In Java, every object can act as a mutex { mutual exclusion mechanism

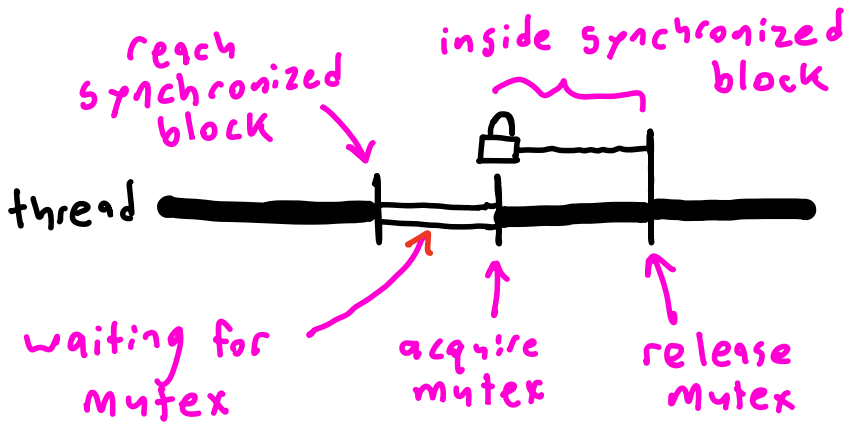
New syntax: synchronized blocks

```
synchronized (obj) {   // atomic operation
}
```

(obj) ← any object reference

{ ← must hold obj mutex while in this block

} ← release obj mutex



Since each object has only one mutex to hand to a thread, at most one synchronized block for each object can be executing at a time

* Synchronized block bodies model atomic operations!

Coding Demo: Synchronized Blocks

Poll Everywhere

PollEv.com/javabear text **javabear** to 22333

```
static void plusOne(SharedInt i) {
    synchronized(i) {
        System.out.print("Before: " + i.x + "\t" );
        i.x += 1;
        System.out.print("After: " + i.x);
    }
}

static void plusTwo(SharedInt i) {  i.x += 2; }

public static void main(String[] args) {
    SharedInt s = new SharedInt();
    new Thread(() -> plusOne(s)).start();
    new Thread(() -> plusTwo(s)).start();
}
```

Which can't be printed by this code?

Before: 0 After: 1	**(A)**
Before: 2 After: 3	**(B)**
Before: 0 After: 3	**(C)**
They all can !	**(D)**

Disciplined Synchronization

- Synchronization only works when every modification to a shared variable is synchronized
- One unsynchronized write = "I'm not waiting for the mutex"
  - reintroduces race conditions
- Proper synchronization is not enforced by the compiler and is hard to ensure with testing... requires good discipline

Also: some atomic operations have more than one piece of shared state

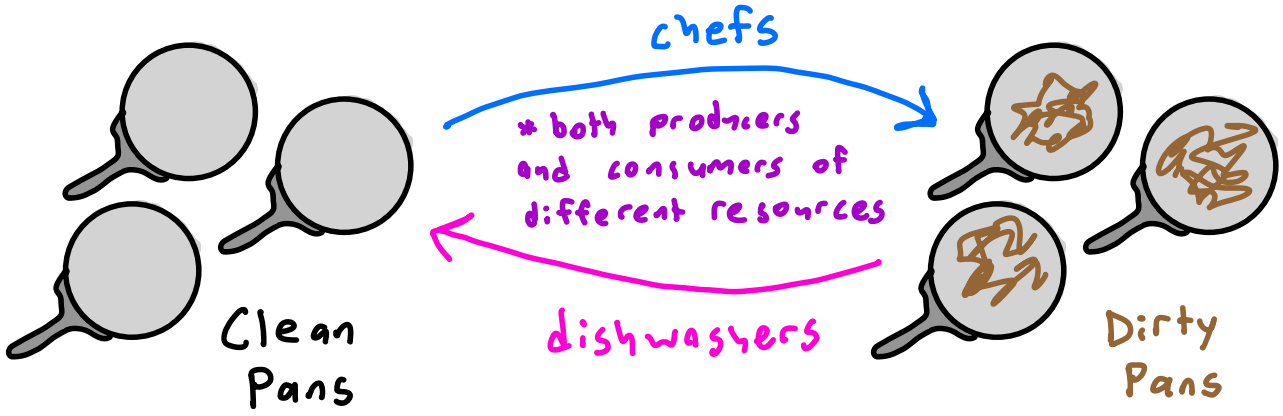
- synchronized blocks only cooperate when they're guarded by the same mutex

Producer-Consumer Problems

Multi-threading problem for resource processing
- some threads are producers that add resources
- some threads are consumers that remove resources

Ex. - order fulfillment on e-commerce sites
- job scheduling on computing servers

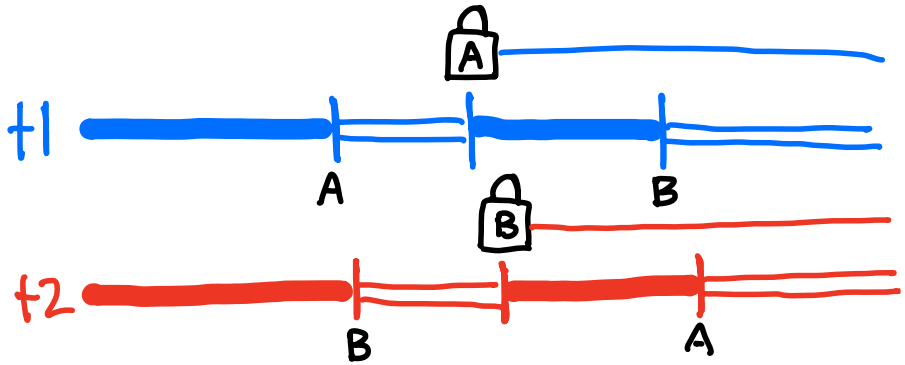
Today: Toy "Kitchen Simulation"



Coding Demo: Multiple Shared Resources

Deadlock

Concurrent code can freeze up when every thread is waiting to acquire a mutex held by another.



t1 holds A and can't release it while waiting for B

t2 holds B and can't release it while waiting for A

Three necessary ingredients:

Solutions: Eliminate one of these

1. Multiple threads want need multiple mutexes → single, coarser lock
2. Nested synchronized blocks → smaller disjoint blocks
3. Cyclic acquisition order → standardize acquisition order

Coordinating Threads

Sometimes, threads need to pause execution to give others time to do prerequisite work.

Already seen 3 examples of this:

- Thread.sleep() (static): pause for (≥) specified amount of time
- t.join(): pause this thread until thread t finishes its execution
- synchronized(obj): pause this thread until no other thread holds the mutex for obj and this thread can acquire it

One more example:

pause a thread until a specific condition is true in the program

* called a "condition variable", but IMO this name is bad because it's not a variable

Poll Everywhere

PollEv.com/javabear text **javabear** to 22333

What can go wrong in this simulation?

```
static void simulateChef(SharedInt c, SharedInt d) {
   System.out.println("The chef takes a clean pan." );
   synchronized (c) { c.x--; }    // take a clean pan
   sleep(100); // cooking time
   System.out.println("Finished cooking!" );
   synchronized (d) { d.x++; }  // pan is dirty now
}

static void simulateDishwasher(SharedInt c, SharedInt d) {
   System.out.println("The dishwasher takes a dirty pan." );
   synchronized (d) { d.x--; }    // take a dirty pan
   sleep(100); // washing time
   System.out.println("Finished cleaning!" );
   synchronized (c) { c.x++; }   // pan is clean now
}
```

A thread can reduce the # of clean / dirty pans to < 0.

Not physically realistic.

```
public static void main(String[] args) {
   SharedInt c = new SharedInt(3); // # clean pans
   SharedInt d = new SharedInt(0); // # dirty pans

   for (int i=0; i<5; i++) {
      // "chef" thread
      new Thread(() -> simulateChef(c, d)).start();
      // "dishwasher" thread
      new Thread(() -> simulateDishwasher(c, d)).start();
   }
}
```

wait() and notifyAll()

When a thread realizes a prerequisite condition hasn't been met, it should voluntarily release its mutex so another thread can take care of it.

syntax:

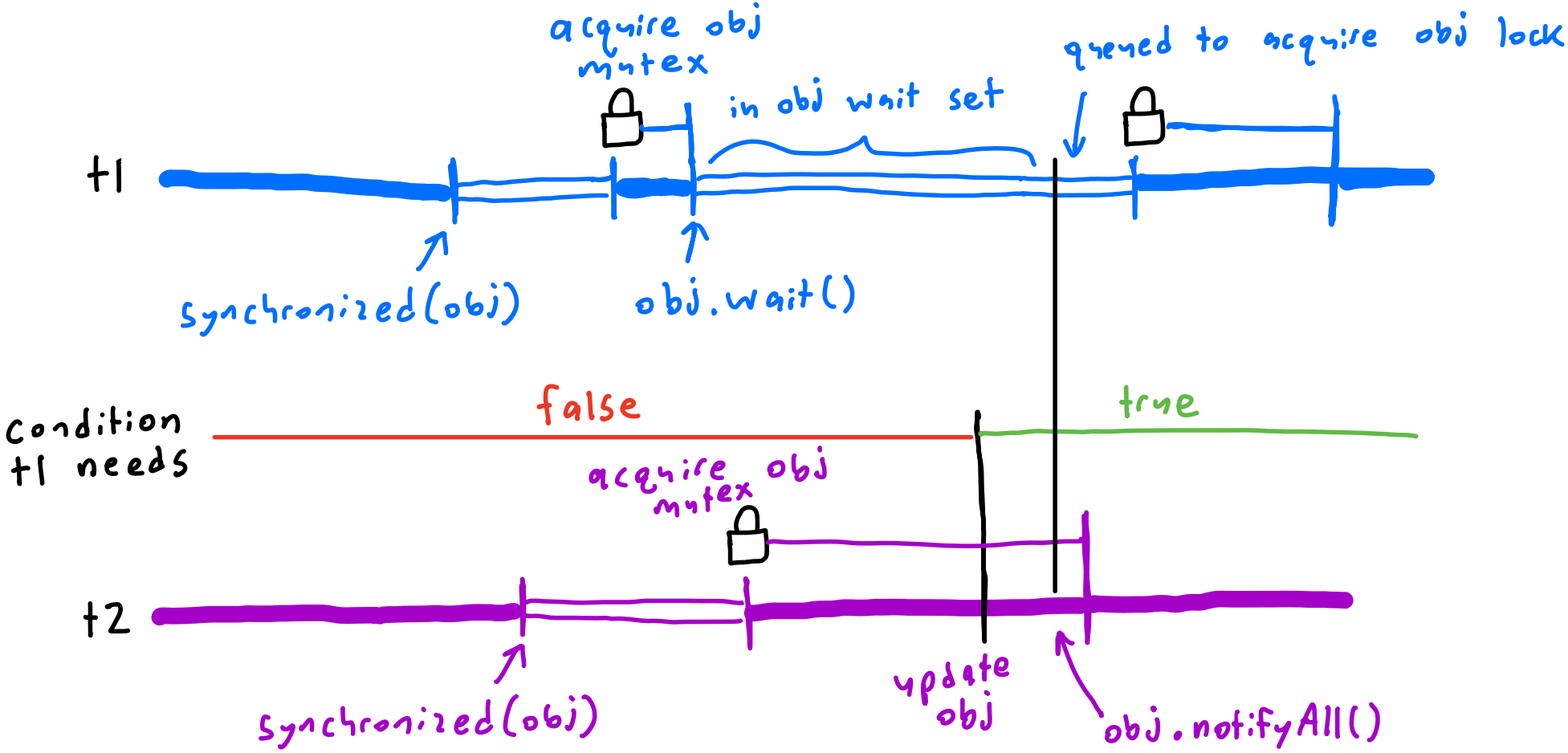
```
synchronized (obj) {
    while (!<condition involving obj>) {
        obj.wait();
    }
    <code requiring condition>
}
```

Always check condition in loop. Just because thread awoken doesn't mean condition became true.

1. drops obj mutex
2. pauses thread's execution
3. thread joins obj's wait set so it can be "woken up" later.

after other threads update state of obj, they should call obj.notifyAll() to "wake up" all threads wait()ing on obj

Visualizing Synchronization



Coding Demo: Synchronization

Keys for Proper Synchronization

- Always call wait() inside of a synchronized block for that object. Part of wait() is releasing the mutex
- Always call wait() within a loop checking for the condition
- Always call notifyAll() (not notify()) on a shared object after modifying its state

* Synchronization requires careful attention and discipline. Missing one rule can break everything and this may not be caught during testing.

The Monitor Pattern

Strategy for writing thread-safe (i.e., properly synchronized) classes.

Add the synchronized keyword to all public instance methods of the class.

- This has the effect of placing the entire method body into a block syncronized on "this"

- Very coarse locking strategy: one lock for whole object

Pro: Easy way to ensure thread-safety

Con: Lose many benefits of concurrency in large data structures.

More sophisticated synchronization with many smaller mutexes can increase performance (but increase opportunities for bugs)