

Lecture 27: Synchronization

CS 2110

December 1, 2026

Today's Learning Outcomes

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

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

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

Where We Left Off: Race Conditions

When **multiple threads** access the **same shared variable** during the **same time window**, and **at least one writes** to that variable, different *interleaving of machine instructions* can lead to different post-conditions.

```
void simulate() throws InterruptedException {
  int[] a = {0}; // shared mutable variable

  Thread t1 = new Thread(() -> a[0]++);
  Thread t2 = new Thread(() -> a[0]++);

  t1.start(); t2.start();
  t1.join();  t2.join();
  System.out.println(a[0]);
}
```

Critical Sections

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

Mutexes and Synchronized Blocks

In Java, every object can act as a **mutex** (short for *mutual exclusion mechanism*).

New Syntax: `synchronized` blocks



Coding Demo: Synchronized Blocks



Disciplined Synchronization

Synchronization only works when *every* modification to a shared variable is synchronized.

Proper synchronization is **not** enforced by the Java compiler, and it is hard to ensure with unit testing (since there is inherent non-determinism).

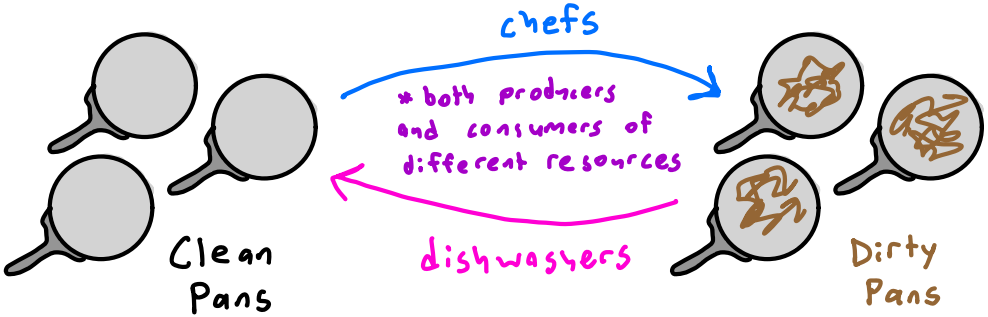
Producer-Consumer Problems

= Multi-threading problems centering on the processing of resource units.

- Some threads are **producers**, which add resource units
- Some threads are **consumers**, which remove resource units

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

Today: Toy "Kitchen Simulation"



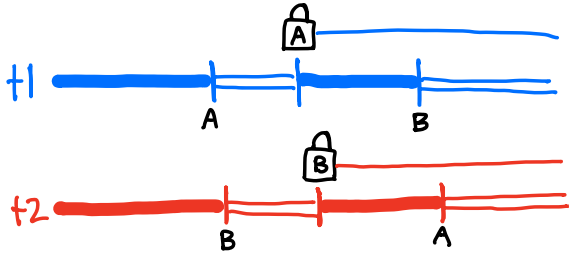


Coding Demo: Multiple Shared Resources



Deadlock

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



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 Conditions:

1.

2.

3.

Coordinating Threads

Sometimes, threads will "strategically" pause executing, often to give other threads time to do prerequisite work. We've seen 3 examples of this:

`Thread.sleep()`:

`t.join()`:

`synchronized(obj)`:

wait() and notifyAll()

When a thread realized that a prerequisite condition of a variable hasn't been met, it should voluntarily release the mutex for that variable so another thread can work toward meeting it.

Syntax:

Visualizing Synchronization



Coding Demo: Synchronization



Keys for Proper Synchronization

* Synchronization requires careful attention and discipline. Missing one rule can break everything (leading to race conditions or deadlock), and this may not be easy to detect during testing.

The Monitor Pattern

= A strategy for writing *thread-safe* (i.e., properly synchronized) classes:

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