Solution: locks

- Every object has a *lock* associated with it.
  - Sometimes called an intrinsic lock or monitor lock.
  - Note: separate locks for each instance!
- A lock can be owned by at most one thread.
  - Sometimes owned by no threads.
- Prevent memory inconsistencies by forcing methods to own the object's lock before running code that needs exclusive access to that object's fields.

Locks

- Locks are not objects themselves.
- Access to them is controlled through blocks of code that are declared as "synchronized."

- When a thread T1 attempts to enter a block of code that is synchronized on object x, T1 tries to acquire x's lock.
  - If x's lock is available, then T1 acquires the lock and runs the block of code.
  - If x's lock is not available (owned by another thread), then the scheduler switches to a different thread. At some point, the scheduler will switch back to T1 and try again to acquire the lock.
- When T1 leaves the synchronized block, x's lock is released.

- First kind of synch block: *synchronized method.*

- Use the word **`synchronized`** before the return type in the declaration line of a method.
- When a thread calls x.method(), the thread will try to acquire x's lock.

```
Class C {
  synchronized void methodA() { }
  synchronized void methodB() { }
}
in main:
  C x = new C(), y = new C();
```

Thread 1:	Thread 2:
x.methodA() // 1 acquires x's lock. // 1 starts running methodA // 1 finishes methodA // 1 releases x's lock	x.methodA() // 2 fails to acquire x's lock // 2 acquires x's lock // 2 starts running methodA // 2 finishes methodA // 2 releases x's lock

```
Class C {
  void synchronized methodA() { }
  void synchronized methodB() { }
}
in main:
  C x = new C(), y = new C();
```

<table>
<tr>
<td>

```
Thread 1:

x.methodA()
  // 1 acquires x's lock.
  // 1 starts running methodA
  // 1 finishes methodA
  // 1 releases x's lock
```

</td>
<td>

```
Thread 2:

x.methodB()
 // 2 fails to acquire x's lock



 // 2 acquires x's lock
 // 2 starts running methodB
 // 2 finishes methodB
 // 2 releases x's lock
```

</td>
</tr>
</table>

```
Class C {
  void synchronized methodA() { }
  void synchronized methodB() { }
}
in main:
  C x = new C(), y = new C();
```

Thread 1:	Thread 2:

`x.methodA()`
`  // 1 acquires x's lock.`
`  // 1 starts running methodA`
`  // 1 finishes methodA`
`  // 1 releases x's lock`
	`y.methodA()`
	`  // 2 acquires y's lock.`
	`  // 2 starts running methodA`
	`  // 2 finishes methodA`
	`  // 2 releases y's lock`

- If T1 owns x's lock, (presumably because T1 has already synchronized on x), T1 may enter another synchronized method of x.
- In other words, if you try to acquire a lock you already own, nothing bad happens.
  - Happens when synch blocks call other functions that have synch blocks.

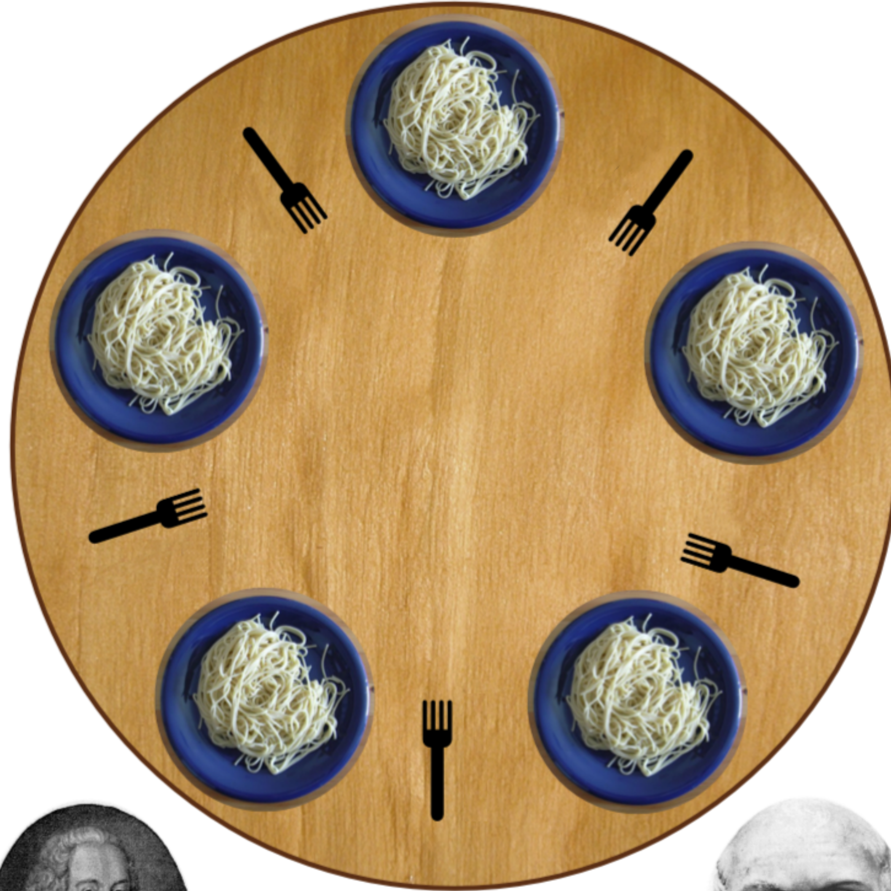
- CPU can still stop a thread T1 in the middle of a synch block and switch to a different thread T2.
- If T2 happens to need a lock owned by T1, then the scheduler will immediately switch again.

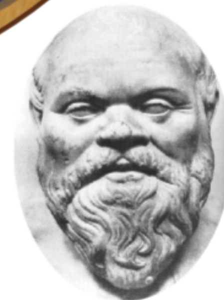
Fix bank account

- Also can have synchronized blocks (inside any method):

```
class C {
  public void method() {
    synchronized (y) {  …  }
  }
in main: C x = new C(); x.method()
```

- When a thread tries to call x.method(), the thread will try to acquire the lock for some other object y, not x.





Assume we have five Fork instances.

Inside each philosopher's run method:

```
synchronized (fork to the left) {
  synchronized (fork to the right) {
    // eat spaghetti
  }
}
```

Deadlock

Remedies

- Resource hierarchy: assign numbers to the forks; can't request a higher-numbered for before a lower-numbered fork.
- Central arbiter: Write a waiter class that manages all the forks. The waiter will never give out forks in a way that will allow deadlock.

Other issues

- Starvation
  - A thread is consistently denied access to a shared resource by other "greedy" threads.
  - Example: synch methods that take a long time to run and are called frequently.
- Livelock
  - Thread A takes some action in response to another Thread B in attempt to avoid a problem.
  - Thread B then response to A's action.
  - Back and forth: neither thread is deadlocked, but they are too busy responding to each other to get anything else done.

Coordination

- Imagine a restaurant with a chef and a waiter.
- The chef's job is to prepare food and place the food in the pickup area.
  - Apparently this area is called the "line."
- The pickup area can only hold one order at a time.
- The waiter's job is to take the food from the pickup area to the tables.

- Class PickupArea models the waiting area for an order. Holds the order number as an int.
- Class Chef is a thread that when started, will cook ten orders back to back (sleeping randomly between them) and place them in the waiting area.
- Class Waiter is a thread that when started, will pick up ten orders from the waiting area and serve them (sleeping randomly between them).

- Waiter doesn't wait for chef to cook meals before serving them.
  - The waiter might serve the same meal over and over, or sometimes will serve order 0, which means there is no meal!
- Chef doesn't wait for the pickup area to be empty before cooking the next meal.
  - The chef might cook multiple orders and put them all in the waiting area back to back, overwriting the existing order that was already there.

2 part solution

- Part A:
  - Synchronize on the pickup area so that the waiter and chef don't step on each other's toes.
- Part B:
  - Have the two threads communicate about when orders are ready.

Solution: Guarded blocks

- A guarded block is a block of code that cannot execute until a condition is true.
- Chef should not cook a new order until the pickup area is free.
- Waiter should not pickup an order unless there is one waiting in the pickup area.

In Chef.run():

```
while (pickupArea.orderNumber > 0) { }
```

In Waiter.run():

```
while (pickupArea.orderNumber == 0) { }
```

Let's try.

Busy waiting is bad, mm'kay?

- Never wait on a condition with an empty while loop.
- If a thread cannot continue until a condition is true, we need to tell the thread to wait without wasting CPU cycles.

- Every object has two methods, called wait() and notifyAll()
- Inside a synchronized block on object x, a thread may call wait() and/or notifyAll()
- x.wait() suspends the current thread until it receives a wakeup call from x.notifyAll()
- x.notifyAll() wakes up all the threads that are waiting on object x.

Most common idiom:

T1:

```
while (!condition) { x.wait(); }
```

T2:

```
condition = true; x.notifyAll();
```

Try it out

Why does this work?

- If T1 holds x's lock and calls x.wait(), then x's lock is temporarily released!
- Therefore, another thread T2 can acquire x's lock to fix the condition that T1 is waiting on.
- Busy waits and sleep()s don't release locks, so our first fix just got stuck forever waiting.

Bank account vs Restaurant

- BankAccount worked with synchronized methods only because if we try to withdraw more money than we have, the withdraw() method simply ***fails***.

Bank account vs Restaurant

- Chef & Waiter needs wait/notifyAll because:
  - We don't want the Chef to lose an order (fail) if there's already an order waiting to be picked up (aka when the Chef is ahead of the Waiter)
  - We don't want the Waiter to pick up the same order twice (fail) if there's not a new order waiting to be picked up (aka when the Waiter is ahead of the Chef).