Extracted from:

The Pragmatic Programmer

your journey to mastery

20th Anniversary Edition

Addison-Wesley

Boston • Columbus • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

20th Anniversary Edition

The Pragmatic Programmer

your journey to mastery

David Thomas

Andrew Hunt

The Pragmatic Programmer

your journey to mastery

20th Anniversary Edition

Dave Thomas
Andy Hunt

Addison-Wesley

Boston • Columbus • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Library of Congress Control Number: [to come from ITP]

ISBN-13: 978-0-13-595705-9
ISBN-10: 0-13-595705-2

1 19

34 Shared State is Incorrect State

You’re in your favorite diner. You finish your main course, and ask your server if there’s any apple pie left. He looks over his shoulder, sees one piece in the display case, and says yes. You order it and sigh contentedly.

Meanwhile, on the other side of the restaurant, another customer asks their server the same question. She also looks, confirms there’s a piece, and that customer orders.

One of you is going to be disappointed.

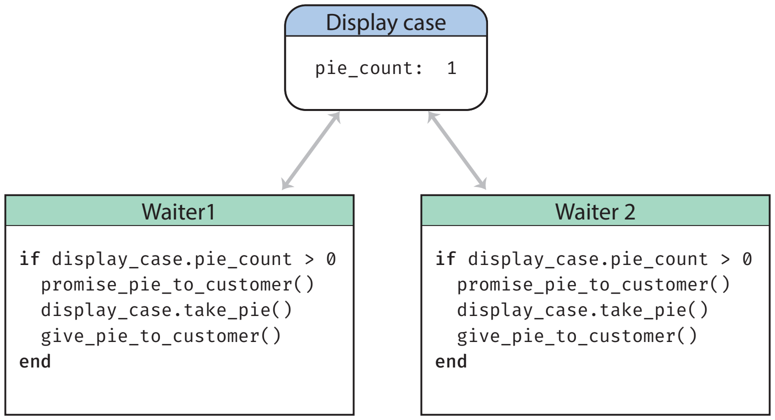
Swap the display case for a joint bank account, and turn the waitstaff into point-of-sale devices. You and your partner both decide to buy a new phone at the same time, but there’s only enough in the account for one. Someone—the bank, the store, or you—is going to be very unhappy.

The problem is the shared state. Each server in the restaurant looked into the display case without regard for the other. Each point-of-sale device looked at an account balance without regard for the other.

Tip 57 Shared State is Incorrect State

Nonatomic Updates

Let’s look at our diner example as if it were code:



Waiter1:

```
if display_case.pie_count > 0
  promise_pie_to_customer()
  display_case.take_pie()
  give_pie_to_customer()
end
```

Waiter 2:

```
if display_case.pie_count > 0
  promise_pie_to_customer()
  display_case.take_pie()
  give_pie_to_customer()
end
```

The two waiters operate concurrently (and, in real life, in parallel). Let's look at their code:

```
if display_case.pie_count > 0
  promise_pie_to_customer()
  display_case.take_pie()
  give_pie_to_customer()
end
```

Waiter 1 gets the current pie count, and finds that it is one. He promises the pie to the customer. But at that point, waiter 2 runs. She also see the pie count is one and makes the same promise to her customer. One of the two then grabs the last piece of pie, and the other waiter enters some kind of error state.

The problem here is not that two processes can write to the same memory. The problem is that neither process can guarantee that its view of that memory is consistent. Effectively, when a waiter executes `display_case.pie_count()`, they copy the value from the display case into their own memory. If the value in the display case changes, their memory (which they are using to make decisions) is now out of date.

This is all because the fetching and then updating the pie count is not an atomic operation: the underlying value can change in the middle.

So how can we make it atomic?

Semaphores and Other Forms of Mutual Exclusion

A semaphore is simply a *thing* that only one person can own at a time. You can create a semaphore and then use it to control access to some other resource. In our example, we could create a semaphore to control access to the pie case, and adopt the convention that anyone who wants to update the pie case contents can only do so if they are holding that semaphore.

Say the diner decides to fix the pie problem with a physical semaphore. They place a plastic Leprechaun on the pie case. Before any waiter can sell a pie, they have to be holding the Leprechaun in their hand. Once their order has been completed (which means delivering the pie to the table) they can return the Leprechaun to its place guarding the treasure of the pies, ready to mediate the next order.

Let's look at this in code. Classically, the operation to grab the semaphore was called *P*, and the operation to release it was called *V*.[2] Today we use terms such as *lock/unlock*, *claim/release*, and so on.

```
case_semaphore.lock()

if display_case.pie_count > 0
  promise_pie_to_customer()
  display_case.take_pie()
  give_pie_to_customer()
end

case_semaphore.unlock()
```

This code assumes that a semaphore has already been created and stored in the variable `case_semapohore`.

Let's assume both waiters execute the code at the same time. They both try to lock the semaphore, but only one succeeds. The one that gets the semaphore continues to run as normal. The one that doesn't get the semaphore is suspended until the semaphore becomes available (the waiter waits…). When the first waiter completes the order they unlock the semaphore and the second waiter continues running. They now see there's no pie in the case, and apologize to the customer.

There's are some problems with this approach. Probably the most significant is that it only works because everyone who accesses the pie case agrees on the convention of using the semaphore. If someone forgets (that is, some developer writes code that doesn't follow the convention) then we're back in chaos.

Make The Resource Transactional

The current design is poor because it delegates responsibility for protecting access to the pie case to the people who use it. Let's change it to centralize that control. To do this, we have to change the API so that waiters can check the count and also take a slice of pie in a single call.

```
slice = display_case.get_pie_if_available()
if slice
  give_pie_to_customer()
end
```

2. The names P and V come from the initial letters of Dutch words. However there is some discussion about exactly which words. The inventor of the technique, Edsgar Dijkstra, has suggested both *passering* and *prolaag* for P, and *vrijgave* and possibly *verhogen* for V.

To make this work, we need to write a method that runs as part of the display case itself:

```
def get_pie_if_available()     ####
  if @slices.size > 0             #
    update_sales_data(:pie)       #
    return @slices.shift          #
  else                            #   incorrect code!
    false                         #
  end                             #
end                            ####
```

This code illustrates a common misconception. We've moved the resource access into a central place, but our method can still be called from multiple concurrent threads, so we still need to protect it with a semaphore.

```
def get_pie_if_available()
  @case_semaphore.lock()

  if @slices.size > 0
    update_sales_data(:pie)
    return @slices.shift
  else
    false
  end

  @case_semaphore.unlock()
end
```

Even this code might not be correct. If `update_sales_data` raises an exception, the semaphore will never get unlocked, and all future access to the pie case will hang indefinitely. We need to handle this:

```
def get_pie_if_available()
  @case_semaphore.lock()

  try {
    if @slices.size > 0
      update_sales_data(:pie)
      return @slices.shift
    else
      false
    end
  }
  ensure {
    @case_semaphore.unlock()
  }
end
```

Because this is such a common mistake, many languages provide libraries that handle this for you:

```
def get_pie_if_available()
  @case_semaphore.protect() {
    if @slices.size > 0
      update_sales_data(:pie)
      return @slices.shift
    else
      false
    end
  }
end
```

Multiple Resource Transactions

Our diner just installed an ice cream freezer. If a customer orders pie *à la mode*,[3] the waiter will need to check that both pie *and* ice cream are available.

We could change the waiter code to something like:

```
slice = display_case.get_pie_if_available()
scoop = freezer.get_ice_cream_if_available()

if slice && scoop
  give_order_to_customer()
end
```

This won't work, though. What happens if we claim a slice of pie, but when we try to get a scoop of ice cream we find out there isn't any? We're now left holding some pie that we can't do anything with (because our customer *must* have ice cream). And the fact we're holding the pie means it isn't in the case, so it isn't available to some other customer who (being a purist) doesn't want ice cream with it.

We could fix this by adding a method to the case that lets us return a slice of pie. We'll need to add exception handling to ensure we don't keep resources if something fails.

```
slice = display_case.get_pie_if_available()

if slice
  try {
      scoop = freezer.get_ice_cream_if_available()
      if scoop
      try {
        give_order_to_customer()
```

3. That's with a scoop of ice cream for those of you living outside the U.S.

```
      }
      rescue {
        freezer.give_back(scoop)
      }
      end
  }
  rescue {
    display_case.give_back(slice)
  }
end
```

Again, this is less than ideal. The code is now really ugly: working out what it actually does is difficult: the business logic is buried in all the housekeeping.

Previously we fixed this by moving the resource handling code into the resource itself. Here, though, we have two resources. Should we put the code in the display case or the freezer?

We think the answer is "no" to both options. The pragmatic approach would be to say that "apple pie à la mode" is it's own resource. We'd move this code into a new module, and then the client could just say "get me apple pie with ice cream" and it either succeeds or fails.

Of course, in the real world there are likely to be many composite dishes like this, and you wouldn't want to write new modules for each. instead, you'd probably want some kind of menu item which contained references to its components, and then have a generic `get_menu_item` method that does the resource dance with each.

Non-transactional Updates

A lot of attention is given to shared memory as a source of concurrency problems, but in fact the problems can pop up *anywhere* where your application code shares mutable resources: files, databases, external services, and so on. Whenever two or more instances of your code can access some resource at the same time, you're looking at a potential problem.

Sometimes, the resource isn't all that obvious. While writing this edition of the book we updated the toolchain to do more work in parallel using threads. This caused the build to fail, but in bizarre ways and random places. A common thread through all the errors was that files or directories could not be found, even though they were really in exactly the right place.

We tracked this down to a couple of places in the code which temporarily changed the current directory. In the nonparallel version, the fact that this code restored the directory back was good enough. But in the parallel version, one thread would change the directory and then, while in that directory, another thread would be running. That thread would expect to be in the original directory, but because the current directory is shared between thread, that wasn't the case.

The nature of this problem prompts another tip:

Tip 58 Random Failures Are Often Concurrency Issues

Other Kinds of Exclusive Access

Most languages have library support for some kind of exclusive access to shared resources. They may call it mutexs (for mutual exclusion), monitors, or semaphores. These are all implemented as libraries.

However, some languages have concurrency support built into the language itself. Rust, for example, enforces the concept of data ownership; only one variable or parameter can hold a reference to any particular piece of mutable data at a time.

You could also argue that functional languages, with their tendency to make all data immutable, make concurrency simpler. However, they still face the same challenges, because at some point they are forced to step into the real, mutable world.

Doctor, It Hurts…

If you take nothing else away from this section, take this: concurrency in a shared resource environment is difficult, and managing it yourself is fraught with challenges.

Which is why we're recommending the punchline to the old joke:

> Doctor, it hurts when I do this.
>
> Then don't do it.

The next couple of sections suggest alternative ways of getting the benefits of concurrency without the pain.

Related Sections Include

- Topic 38, *Programming by Coincidence*, on page ?
- Topic 28, *Decoupling*, on page ?