concurrency bugs and deadlock.

Broadly speaking, concurrency bugs break down into two categories:

local-invariant bugs

In this bug, there's some sort of pattern---e.g., always hold a lock when accessing data, or always remember to notify() when filling a queue---which has not been followed somewhere in the program. The nice thing about this bug is that it's *local*---as in, you can look at just a few lines of code in which you [should have] used a concurrency primitive, and catch that you used it incorrectly. We talked a bit about this bug last time, but the key thing you can do to prevent these *local-invariant* bugs is to **make your invariants impossible to ignore.** The container-lock macro pattern is an illustration of this: we created a convention (only use locked memory within a container-lock scope), and we built out a *language abstraction* that *forces* us to remember to close our locks---thus eliminating cases where our code compiles *and* we forgot to release one of our locks.

Other invariants---e.g. when building a queue or buffer datastructure, one must always remember to signal/notify whenever one inserts (in case there are waiting readers)---are harder to directly enforce in C. The "good" software engineering thing to do here is threefold:

- create accessor functions that manipulate your datastructure
- write invariants (in comments) about what locks and condition variables your datastructure should use, and when
- audit your accessor functions to ensure that those invariants are maintained
- write invariants about when you can safely use your accessor functions (if there are any limitations on safe times to use them)

Let's go over the container-lock example from last time, and then work together to figure out how to use this container-lock, and a condition variable, to correctly manipulate a bounded-buffer / queue.

(go to the code).

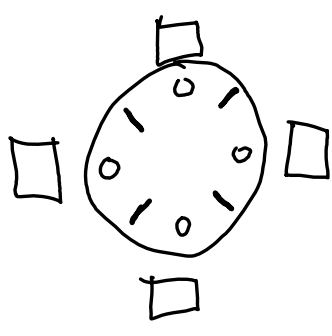
We're going to do this really, really interactively---so I have *no* code prepped, and I hope y'all are ready and paying attention

global-invariant bugs

The nice thing about local-invariant bugs is that careful design (and, if you're using a modern programming language, compiler support) can actually eliminate *almost all* of these bugs. If you know what you need to do in order to be correct, and you know that each context can be analyzed independently to assess correctness, then your debugging experience is *composable;* you don't need to know *anything* about the rest of the program to know that your particular function snippet is correct.

We really like these composable properties!

Global-invariant bugs are *not* that. The most famous example of such a bug is the dining philosophers.



remember: you have to grab *two chopsticks* to eat, because that's how chopsticks work. (The lines between each box represent chopsticks; the previous lecture notes has this example rendered in color, too).

(You can read dining philosophers online or in the book (section 31.6), but the short version is that it's possible to get in an infinite waiting cycle; we're going to do this interactively in class, so no spoilers shall be present in these notes)

The challenge: each philosopher is its own thread. Each fork is a mutex. Can we come up with a *uniform algorithm* that will always *make progress*? As in, can we show that (given a fair mutex) every philosopher who wants to eat, will eventually be able to eat?

(interactive!)

Well, we can't.

This is a famous example of a **deadlock**, which (along with its cousin **livelock**) is the most-studied class of *global invariant* violations in concurrent code. The underlying invariant that we need, is that every lock will *eventually* be released---which in turn means that if **any thread holds a lock**, then when it goes to **acquire** an additional lock, its ability to acquire this second lock **cannot depend** on the state of the locks it *already holds*.

Or, to put it more simply: if a thread holds lock A and wants lock B, then there must *never* be any threads that could hold lock B while wanting lock A.

This is an example of a *waiting cycle*; a program is deadlock-free if we can show that there are no waiting cycles.

Let's look again at the dining philosophers to show the waiting cycle on the board (do this diagramatically, on the board. The basic idea: program order within a single philosopher creates a dependency between a pair of locks; if each philosopher grabs L then R, then each left-fork will wait for each right-fork. You should give the forks numbers in order to make it easier for us to do the next part).

Eliminating deadlocks, comprehensively.

This is an example of a global invariant; can we think of any ways that we could get around it? Some property we would want to be true of our programs to provably eliminate it? Let's actually think about this one for a bit; turn to your neighbor and chat for like 2 minutes about how we can be *sure* that there are no waiting cycles here. As always, I want you to have a suggestion, an idea, an answer, or a question for me!

(we wait, and see who's reading the notes / who's feeling very clever today)

Quite possibly the simplest way to eliminate a deadlock is to just enforce a global, total order on which locks are acquired! If it's a total order, it doesn't have a cycle. Let's see what this looks like in the dining philosophers.

(you are illustrating this on the board)

(We can now assume that the philosophers can read their fok-number, in the "real world" we can just use the address of the mutex to derive this total order.)

Ok! This solution will always work. But is it always possible to acquire locks in this single, total order? Can anyone suggest why or why not??

(another round of discussion)

An answer we might expect: if your decision of whether to grab a next lock depends on something you can only read when you hold a *first* lock, then we have to respect that order.

This is why deadlock is a ***global*** *invariant*:

- (composed property) unless I know about *all* the locks in the system, and I know about how *all* the threads are going to use the locks, it is *always possible* that some unknown thread, using known (or unknown!) locks, will introduce a deadlock.
- (no local check) If we look at any individual thread acquiring locks in any order, then it's always possible that this thread "got it right;" as in, we can't really blame any single participant in a waiting-cycle as "the" thread that got it wrong.
- we can solve the deadlock issue by using *global knowledge*; we assumed that we can have a total order on *all locks*, even those locks that aren't mentioned anywhere in our compilation unit / source file. This is a strong assumption! It's also a true assumption...at runtime :/ So we only really get to see violations of it *at runtime*.

Looking again at dining philosophers, we were promised a total order. That's what we rely on.

Let's make sure we spend the time to let "global knowledge cannot be known at compile-time" sink in a bit... it's the one, really-big difference between the local-invariant patterns and this one.

Programming solutions to deadlock.

We might have time, if so we'll just do this live:

- *the oracle function*: write a library that exposes a single function "lock all", which takes any number of locks as parameters and locks all of them in "the right" order. In C-on-UNIX, we can use the address of the locks to implement this function (you can show it implemented in C++)
- *the livelocker*: write a library that, when a thread attempts to acquire a new lock, first releases all old locks and then re-acquires in the right order. **Danger:** you need to double-check any invariants that your code relied upon after this, as another thread might have come in and changed state! (note: also a livelock threat, explain if there's time)
- *the checker:* when you initialize your locks, register them with some global list of all locks in the system. Create a thread that periodically looks at those locks, to see if it can detect a cycle (it can see who's in the lock queue, and who holds a lock, for every lock). If you find a cycle, then... usually just kill the program with an easy debugging message.