From last time --- you didn't get to talk about the principle of "own your locks" before, so maybe *start* class with a bit of a digression there. If you feel inspired, you can talk about how *other* languages choose to expose locking primitives, beyond just C.

Other things you want to cover:

- concurrency races / how to debug them

- coroutines / event-based concurrency

- locks as containers

Ok, so it sounds like this is going to be a lecture in two parts: part 1, is fancy locks ... wait, they just did coroutines with Amit. So probably not good to start with fancy locks then, let's start with co-routines. Which means we need working coroutine code! Hmmm let's go back to david's tutorial.

Ok, we have coroutines and generators working now. So part 1, we'll show them coroutines and generators, and then part 2 we'll talk about avoiding deadlock. Which means dining philosophers, so I should restore that from last week's notes

More primitives / alternatives

So: at the end of last time, you *just* saw what continuations and co-routines are, and we talked a lot about why threads are bad, yes?

(everyone nods. Please. Nod!)

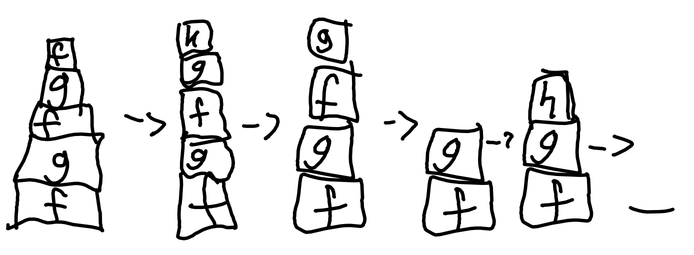
Great! So let's pick back up from there. In c++ this time, because it's a little closer to C / the syntax you're familiar with.

A *coroutine* is an abstraction of an interruptable function. In other words, its a function that can return (or yield) early *and then resume execution* at the point at which it returned. Kind of like generators in Python, though it's more flexible than that.

(let's freehand a basic co-routine on the board, in c++, remember to have your stuff up+ready ahead of time, this time)!

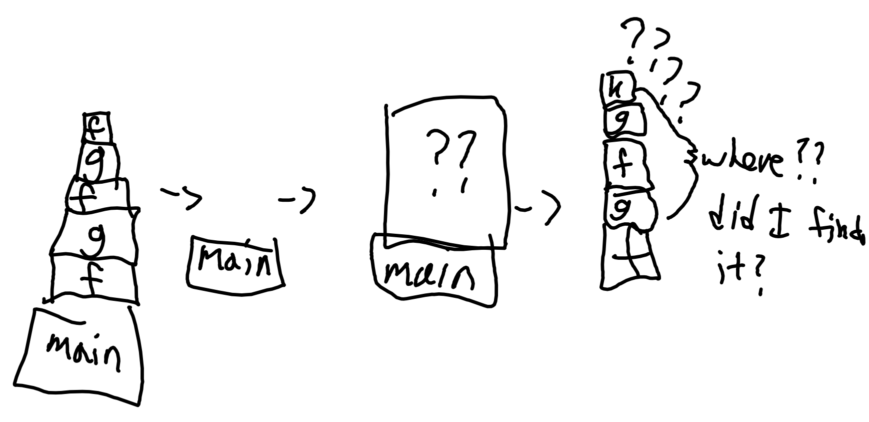
In order for a co-routine to work, it needs to do a few things differently than a normal funciton. In particular, with normal functions we have a notion of a call stack, where each new function call pushes a frame onto the call stack, and each function return pops it.

```
f(int i){
  if (i < 2) g(i);
}

g(int i) {
  f(i+1);
  h();
}

h(){}

f(0)
```

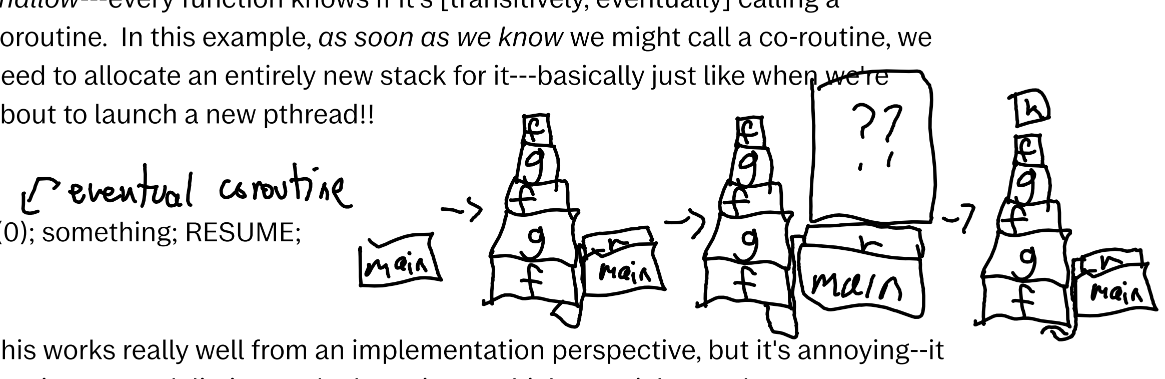


The trouble with co-routines, is that you can now *resume* a function. But when you resume it, how do you find its stack frame?

```
f(int i){
  if (i < 2) g(i);
}

g(int i) {
  f(i+1);
  YIELD;
  h();
}

h(){}

f(0); something; RESUME;
```



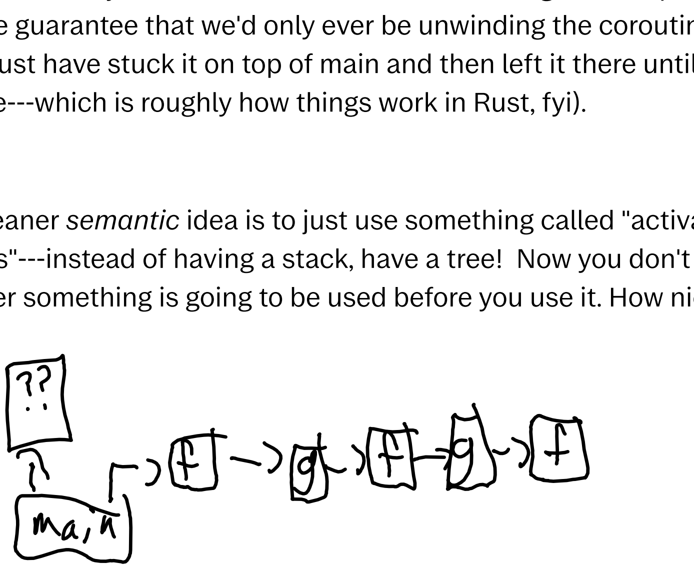
You can't with a basic stack! This means that we need to do something special. There are two schools of thought here: one option is that we make coroutines *shallow*---every function knows if it's [transitively, eventually] calling a coroutine. In this example, *as soon as we know* we might call a co-routine, we need to allocate an entirely new stack for it---basically just like when we're about to launch a new pthread!!



f(0); something; RESUME;

This works really well from an implementation perspective, but it's annoying--it requiers us to *delimit* exactly the point at which we might need to return to a co-routine. So, that can be frustrating in code. But! It doesn't really screw with the stack at all, and theres a fairly clear resource management story for it. (As to why we "really" need to do this? Both stacks can grow independently. If we had the guarantee that we'd only ever be unwinding the coroutine stack, we could just have stuck it on top of main and then left it there until after the last resume---which is roughly how things work in Rust, fyi).

The cleaner *semantic* idea is to just use something called "activation records"---instead of having a stack, have a tree! Now you don't have to know whether something is going to be used before you use it. How nice, how easy, yes?



...but it messes with the basic assumptions about how the stack works, which will screw with *everything* in your program as soon as you have *one* co-routine. Plus, it usually generates a lot of garbage. Which we don't like.

Ok! So, now we have a basic sense of how co-routines work, why do we *want* them? Because we can implement event-based concurrency, yes.... but even better, because we can implement *green threads*.

Remember when we were working through process scheduling, and we talked about the various strategies for running and scheduling processes *fairly*? At the time, I said that if we could trust the compiler to insert frequent-enough yields, or the programmer to just never try to hog the CPU, then there would be *no need* for a timer-based interrupt scheme. In the context of processes, this was fantasy; nobody knows why anybody is running any particular process. But in the context of *threads,* this suddenly makes sense! Threads run in a single process space, which (from the operating system's perspective) means that we can make *fundamentally different trust assumptions* about how threads cooperate vs. how processes cooperate. In particular, we *can* trust them to yield "enough" to enable *cooperative* multi-threading.

Let's look at an example of this using the coroutine framework in c++, with one of our classic increment examples as the basis for it.

(this is where we go to the code).

If you can get away with this, *please do.* With locks, screwing up your critical regions means you limit concurrency and add overhead. With event-based coroutines, that's not the case!

...except everything is running on a single code

which leads us to the usual principle of coroutine design:

separate threads from tasks

only have as many threads as cores (i.e. thread=core for our purposes)

run different tasks on different threads

you will *still need locking* when you have multi-core execution, but often not as much; since *you completely control the scheduler,* you can simply choose to never place two tasks that contend the same area of memory onto the same core! It takes a careful eye, but modern programming practices can really help make this less error-prone.

Speaking of modern programming practices...

Locks as containers, or, how to use a good programming language

A *lot* of what we've heard about so far during locking has been about best-practices and design patterns; always remember to grab the right lock for the right data items; alway remember to signal with a CV if you know there is a waiter; never forget to unlock after you have locked.

Doing this well can be tricky... not because it's conceptually hard, but because it's a big checklist of tasks to keep in your head!

In modern languages, *the compiler can help with that*. Because we're in C for this class, I'll only show the design-pattern equivalent of what I mean here. But in general, for *any* of these locking issues, you should be thinking to yourself:

- what is the *universal invariant* that I am enforcing / need to enforce for correctness?
- How could I write that down in a general way, so it's not too tied to my exact code?

Any time you can come up with an invariant, and express it in a general way, you're probably looking at something a modern programming language or compiler can help with.

Ok! So let's look at the most-basic one---locks as containers. A simple issue that can come up when you're handling locks is that you forget to unlock them, right? Or, that you grab the wrong lock for a protected data item? So, let's see if we can rethink the locking patterns to *ensure* that we can't forget to unlock them.

(let's rederive the container-locks macros live in class. Use the support code if you need it for correctness).

(This exercise is done live, sorry y'all)

Great! There's lots of examples of times you can do that. Next, let's move on to talking about another locking primitive that we've seen before: reader-writer locks (next page).

Ok, reader-writer locks! You can explain how reader-writer locks work, basically. Draw on the board the reader lock, and the writer lock; illustrate a schedule of several threads getting reader locks, and then *blocking those threads* until we acquire a writer-lock. We *should* have seen this before, so hopefully this is just all review!

Now, we've asked for this before, but to ask again---what are some scenarios where we might want these? (interact!)

It turns out, we can implement these with semaphores! Let's see if we can come up with *how* we can use semaphores to implement reader-writer locks.

(let's pop to the code examples we've prepared for this)

Hint: an easy way to do this involves a semaphore and a lock, but that's not the only way. Another hint: unlike a mutex, the thread that "releases" a binary semaphore doesn't *need* to be the same thread that "acquires" it!

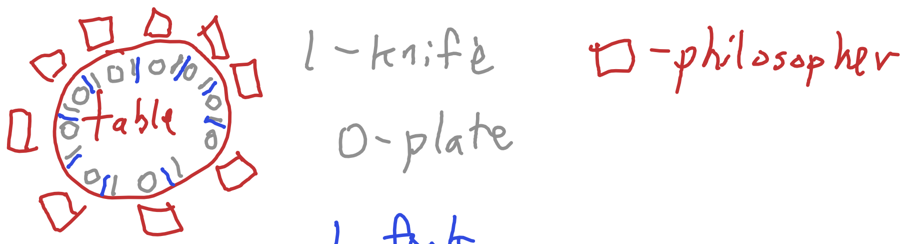
Let's do this together in code / on the board.

Hopefully this makes sense? Maybe?

(from here: *if time allows)*

Deadlock

Let's talk about Deadlock. In particular, let's use the dining philosophers to do so.



Basic algorithm: the philosopher first grabs a fork, and then grabs a knife, and then proceeds to eat!

What happens if two philosophers try to grab the same fork? What happens if they try to grab the same knife?

Is there an algorithm that we can run at *every* philosopher in order to ensure we grab the forks correctly?? (there is not. Somebody needs to be different).