Concurrency

> We want a solution that's *more correct* than letting everyone grab out a peach at once, but *more efficient* than having them all line up.

An Introduction

Building on top of previous abstractions, which provided the illusion of multiple *virtual CPUs* and a large and private *virtual memory*, we can have an abstraction for points of execution. Classical programs have just one point of execution, but we could plausibly have multiple. The result *threads*, looks like multiple processes, except that they all share the same address space.

Each thread has a program counter and a private set of registers (so switching threads involves a *context switch*, though we don't have to switch page tables). Each thread has its own stack (so multiple function executions can happen in the address space at once), sometimes called *thead-local storage*.

Why Use Threads?

- *Parallelism*: it's possible to more effectively use multi-core CPUs
- *Avoiding blocking*: overlap of IO/blocking activities and other activities becomes possible within one program

The Basic API

On Unix, threads can be created with `pthread_create`, and `pthread_join` makes the calling thread wait for another thread to terminate before it proceeds. The order that threads are run is otherwise non-deterministic, governed by some scheduling algorithm. This can lead to *data races*, where uncontrolled scheduling can produce non-deterministic results when code is run. This basically happens because operations are often not atomic meaning the program can be pre-empted in the middle of their execution, causing the end result to come out different from what's expected.

Mutual Exclusion

To solve this, we need *mutual exclusion*, to ensure that only one point of execution passes through certain *critical sections* at a time. One approach would be to have more sophisticated instructions, like `memory-add` that are processed atomically, but this breaks down first because we would have to have tons of them, and second because we'd end up with the same problem composing these operations.

Instead, we build a set of *synchronization primatives* in hardware which allow us to section off critical sections in a way that's much more flexible.

Condition Variables

Another common situation is when we have one thread that needs to wait for another to finish executing before moving forward. For example, we could have a thread that initiates some IO, and we want it to sleep until the IO is finished and be awoken then. This will involve *condition variables*.