Coroutines and Concurrency

We'll cover the following

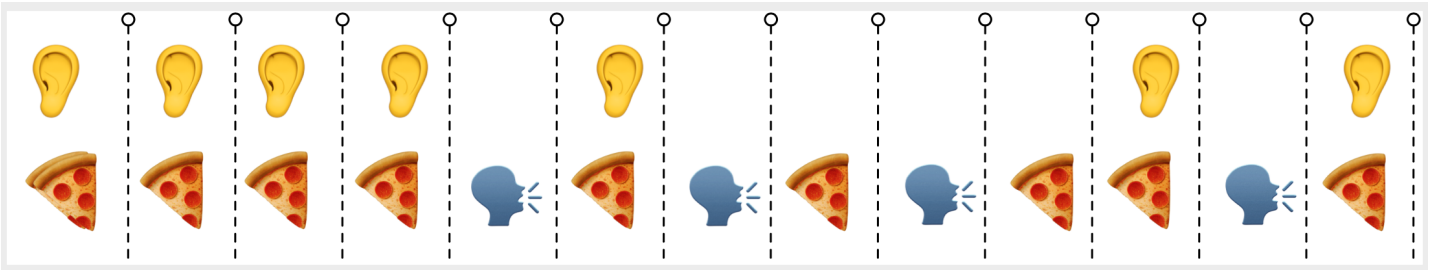
- Parallel vs. concurrent
- Coroutines as cooperating functions

Some tasks have to be executed sequentially, some may be performed in parallel, and yet others may be done concurrently. Most people put on the underwear before putting on the pants, though the order may be in reverse—even then, superwoman and superman don't put them on at the same time. These two tasks are inherently sequential; don't attempt an alternative.

Sequential execution is clear; we perform one task to completion before beginning the next. But some confusion exists among programmers between parallel and concurrent execution. Understanding the difference between them is critical, as multithreading on a multi-core processor generally is parallel, while coroutines are generally used more often for concurrent execution rather than parallel execution.

Parallel vs. concurrent

Let's understand the difference between parallel and concurrent using an example. Suppose longtime friends Sara and Priya meet for dinner, and Sara is curious about Priya's recent international trip. Priya can listen to Sara while munching on food, but she'll never speak with her mouth full, as you can see in the following figure.

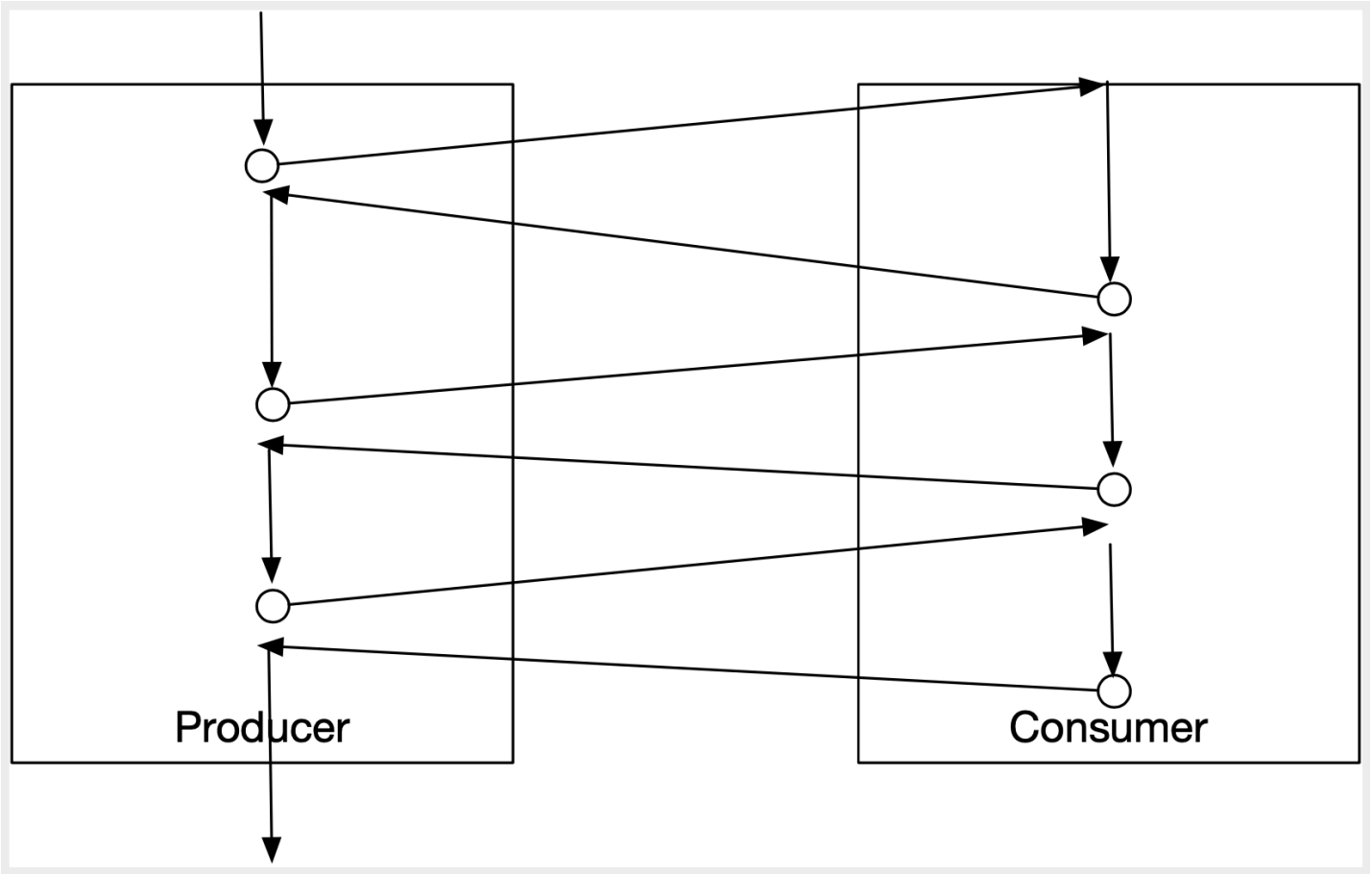


In the left part of the figure, we see Priya eating and listening in parallel. But when it's her turn to speak, she takes a break from eating to talk about her travel

experience. She's got several things to say, but doesn't want to let the pizza go cold, so her speaking is interleaved with her eating—that's concurrency. In general, we can eat and listen in parallel, but we eat and speak concurrently.

Coroutines as cooperating functions

Subroutines are more common than coroutines in general-purpose programming. Subroutines are functions that run to completion before returning to the caller. Another call to the same subroutine is as good as the first call; subroutines don't maintain any state between calls. Coroutines are also functions but behave differently than subroutines. Unlike subroutines, which have a single point of entry, coroutines have multiple points of entry. Additionally, coroutines may remember state between calls. And, a call to a coroutine can jump right into the middle of the coroutine, where it left off in a previous call. Because of all this, we can implement cooperating functions—that is, functions that work in tandem—where two functions can run concurrently, with the flow of execution switching between them, like in the example shown in the figure below.



First, the Producer coroutine calls the Consumer coroutine. After executing part of its code, the Consumer saves its current state and returns or yields to the caller

its code, the Consumer saves its current state and returns or yields to the caller. The Producer performs some more steps and calls back into the Consumer. This

time, however, instead of the call starting in the beginning, the call resumes from the point where the previous execution left off, with the state of the previous call restored.

This appears to be magic, but let's take a closer look at one of the benefits of this capability to run coroutines as cooperating functions in the next lesson—the ability to run coroutines concurrently.

QUIZ

Q

You are working on your laptop while waiting in line to get your passport. This is an example of concurrency.

Retake Quiz