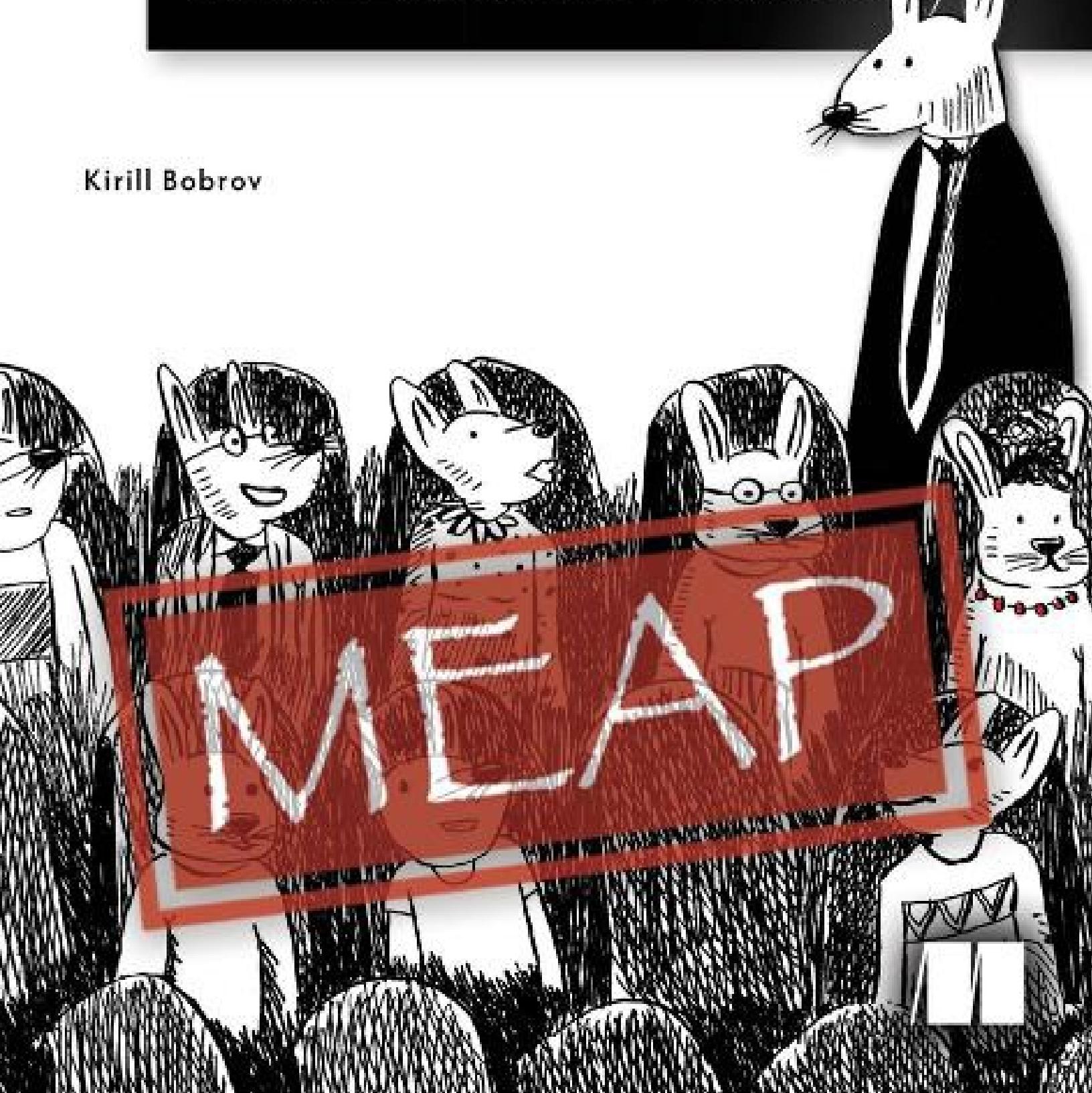


grokking

# concurrency

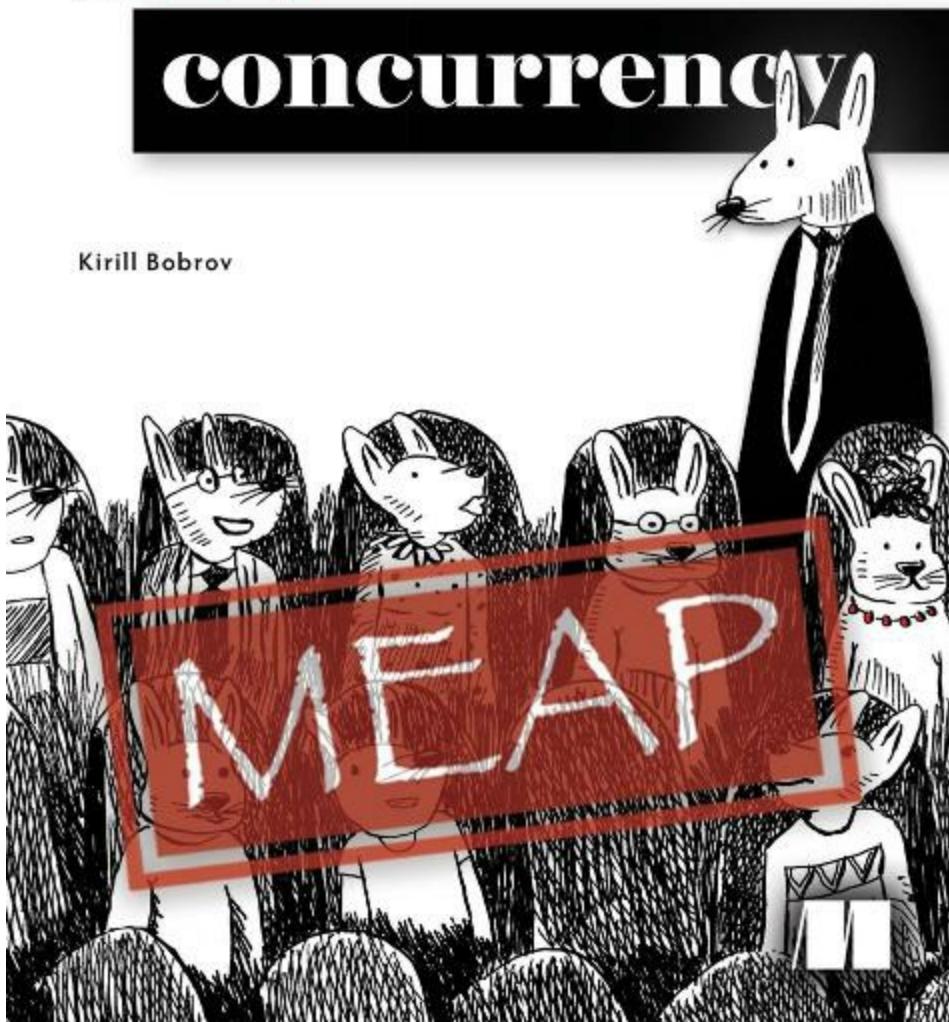
Kirill Bobrov



grokking

# concurrency

Kirill Bobrov



# Grokking Concurrency MEAP V12

1. [Copyright 2023 Manning Publications](#)
2. [welcome](#)
3. [1 Introducing concurrency](#)
4. [2 Serial and parallel execution](#)
5. [3 How computers work](#)
6. [4 Building blocks of concurrency](#)
7. [5 Inter-process communication](#)
8. [6 Multitasking](#)
9. [7 Decomposition](#)
10. [8 Solving Concurrency Problems: Race condition & synchronization](#)
11. [9 Solving Concurrency Problems: Deadlock & starvation](#)
12. [10 Non-blocking I/O](#)
13. [11 Event-based concurrency](#)
14. [12 Asynchronous communication](#)
15. [13 Writing concurrent applications](#)



MEAP Edition Manning Early Access Program Grokking Concurrency  
Version 12

Copyright 2023 Manning Publications

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://livebook.manning.com/book/grokking-concurrency/discussion>

For more information on this and other Manning titles go to

[manning.com](https://manning.com)

# welcome

Thank you for purchasing *Grokking Concurrency!*

This is a book for everyone who wants to learn the concepts behind concurrent and asynchronous programming.

Concurrency is a hot topic, and it will remain hot for the foreseeable future, because the industry is struggling with scalability and performance. Despite this, there aren't many books on the subject that provide readers with a clear introduction to the world of concurrent programming. Consider this book to be your guide to getting started in the concurrency space.

The book itself is not tied to any specific implementation in a programming language or framework, but rather the fundamental ideas and concepts behind concurrency. While describing the concepts, we will remain practical - the book contains many references to existing programming languages and tools that implement ideas behind concurrency. The book demonstrates best practices and patterns to help you implement concurrency in your systems.

Before starting this book, you should have some familiarity with the basics of computer systems and programming language concepts. Although the examples are written in Python programming language, the concepts described in the book are language independent, although their implementation in different programming languages may be different. So, knowing Python will be beneficial but is not a prerequisite.

Knowledge on the workings of operating systems is not required, as all the necessary information will be provided; however, some basic networking fundamentals are required. You don't need deep knowledge on any of these topics, and, if needed, you could definitely research them as you go.

I am not a scientist, so there are no mathematical explanations here, but more high-level and practical aspects of the problem in order to create a solid understanding of the topic. I'm also a visual learner, and I like to learn

through visual diagrams and illustrations, so illustrations are a huge if not central part to all the stories in the book.

I hope that you enjoy the book and that it will occupy an important place on your bookshelf (be it physical or digital!). This journey should be fun, but you tell me. Happy reading!

If you have any questions, comments, or suggestions, please share them in [liveBook Discussion Forum](#).

- Kirill

#### In this book

[Copyright 2023 Manning Publications welcome](#) [brief contents](#) [1 Introducing concurrency](#) [2 Serial and parallel execution](#) [3 How computers work](#) [4 Building blocks of concurrency](#) [5 Inter-process communication](#) [6 Multitasking](#) [7 Decomposition](#) [8 Solving Concurrency Problems: Race condition & synchronization](#) [9 Solving Concurrency Problems: Deadlock & starvation](#) [10 Non-blocking I/O](#) [11 Event-based concurrency](#) [12 Asynchronous communication](#) [13 Writing concurrent applications](#)

# 1 Introducing concurrency

## In this chapter

- You learn why concurrency is an important topic worth studying
- You learn how to measure the performance of the systems
- You learn that there are different layers of concurrency

## 1.1 We live in the concurrent world

Look out of the window and take a moment to observe the world around you. Do you see the world moving in a linear, sequential fashion? Or do you see a complex web of interacting, independently behaving pieces all happening at the same time?



Although people tend to think sequentially – like going through to-do lists, doing things step by step, one step at a time, the reality is that the world is much more complex than that. It is not sequential but rather concurrent. Interrelated events are happening simultaneously. From the chaotic rush of a busy supermarket to the coordinated moves of a football team, to the ever-changing flow of traffic on the road, concurrency is all around us. Just as in the natural world, your computer needs to be concurrent to be suited for modeling, simulating, and understanding complex real-world phenomena.

Concurrency in computing allows a system to deal with more than one task at a time. This could be a program, a computer, or a network of computers.

Without concurrent computing, our application would not be able to keep up with the complexity of the world around us.

As we delve deeper into the topic of concurrency, several questions may arise. First off, if you're still not convinced - why should you care about concurrency?

## 1.2 Why is concurrency important?

Concurrency is essential in software engineering. The demand for high-performance applications and concurrent systems makes it a crucial skill for software engineers to possess.

Concurrent programming is not a new concept, but it has gained significant attention in recent years. With the increasing number of cores and processors in modern computer systems, concurrent programming has become a necessary skill for writing software. Companies are looking for developers who are proficient in concurrency, as it is often the only way to solve problems where computing resources are limited, and fast performance is required...

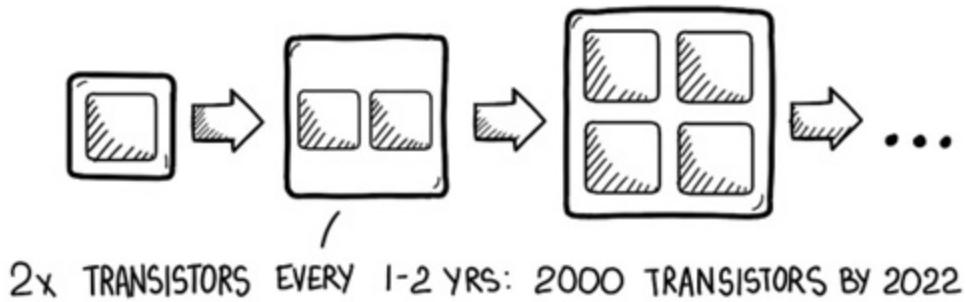
The most important advantage of concurrency, and historically the first reason to start exploring this area, is to increase system performance. Let's look at how that happened.

### 1.2.1 Increase system performance

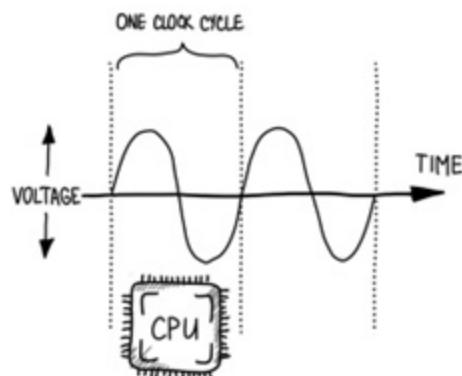
When we need to improve performance, why can't we just buy faster computers? Well, that was what people did do a few decades ago, but we found out that eventually it is no longer feasible.

In 1965, Gordon Moore, one of the founders of Intel, discovered a pattern. New processor models appeared about 2 years after their predecessors, with the number of transistors in them roughly doubling each time. He concluded that the number of transistors, and consequently the processor's clock speed would double every 24 months. This observation became known as Moore's Law. For software engineers that meant that they had to wait only 2 years for

an application to double in speed.



The problem was that around 2002, the rules changed. As the famous C++ expert Herb Sutter put it, “the free lunch was over”<sup>[1]</sup>. We discovered a fundamental relationship between the physical size of the processor and the processing speed (processor’s frequency). The time required to execute an operation depends on the circuit length and the speed of light. Simply put, we can only add so many transistors (the fundamental building block of computer circuitry) before we run out of space. Rising temperatures also play a major role. Further performance improvements could not depend on merely increasing the processor’s frequency. Thus, began what’s become known as the “multi-core crisis”.



The progress of individual processors in terms of clock speed stopped due to physical limitations. But the need to increase the performance of systems is not. And manufacturers' focus shifted to horizontal expansion in the form of multi-processors, forcing software engineers, architects, and language developers to adapt to architectures with multiple processing resources.

The most important conclusion from this history tour is that by far the most important advantage of concurrency and historically the first reason to start exploring this area is to increase system performance in a way that makes efficient use of additional processing resources. This leads us to an important point: how do we measure performance, and how can we improve it?

## Latency vs Throughput

In computing, performance can be quantified a number of ways, depending on how you look at the computer system. One way to increase the amount of work done is reduce the time it takes to perform individual tasks.

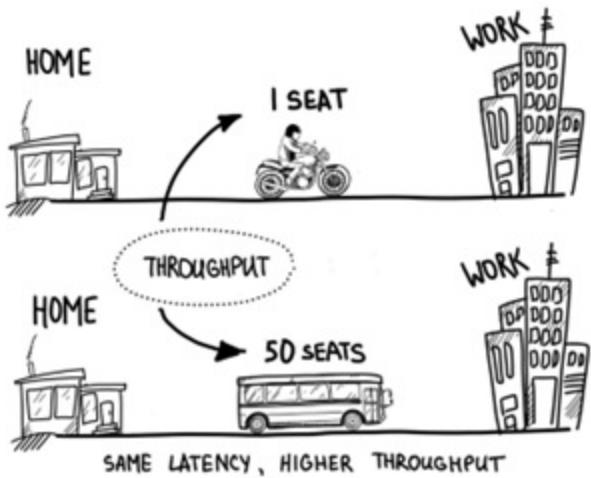
Let's say you use a motorcycle to travel between home and work and it takes an hour for a one-way trip. You care about how fast you can get to work; hence you measure the system performance by this metric. If you drive faster, you'll get to work sooner. From a computing system perspective, this scenario is called *latency*. Latency is a measure of how long a single task takes from start to finish.



Now imagine you work for a transportation department, and your job is to increase the performance of the bus system. You aren't just concerned about getting one person to the office faster. It's about increasing the number of people who can get from home to work per unit time. This scenario is called *throughput*, the number tasks a system can handle over a period of time.

It is very important to understand the difference between latency and throughput. Even if the motorcycle went twice as fast as the bus, the bus still would have 25x greater throughput (the motorcycle transports one person over the course of an hour, while the bus, even if it takes 2 hours, is

transporting 50 – which, when averaged for time, gives us 25 people per hour!). In other words, higher system throughput does not necessarily mean lower latency. When optimizing performance, an improvement in one factor (such as throughput) may lead to the worsening in another factor (such as latency).



Concurrency can help with decreasing latency. For example, a long-running task can be broken down into smaller tasks executed in parallel thus reducing the overall execution time. Concurrency can also help with increasing throughput by allowing multiple tasks to be processed simultaneously.

In addition, concurrency can also hide latency. When we are waiting for a call, waiting for the subway to take us to work, and so forth, we can just wait, or we can use our processing resources for something else. For example, we might read our emails while catching a ride on the subway. This way, we're essentially doing multiple tasks at once and hiding the delay by making production use of the waiting time. Hiding latency is key to responsive systems and is applicable to problems that involve waiting.

Therefore, using concurrency can improve system performance in three main ways:

- it can reduce latency (that is, make a unit of work faster)
- it can hide latency (that is, allow the system to accomplish something else during an operation with high latency)

- it can increase throughput (that is, make the system able to do more work).

Now that we've looked at how concurrency is applied to system performance, let's look at another application of concurrency. Early in this chapter we considered how concurrency is necessary if we want to model the complex world around us. Now we can get more specific about how concurrency can solve large or complex problems computationally.

### **1.2.2 Solving complex and large problems**

Many problems that software engineers need to solve when developing systems that deal with the real world are so complex that it is often impractical to solve them using a sequential system. Complexity can come from the size of the problem or how hard it is to understand a given piece of the systems we develop.

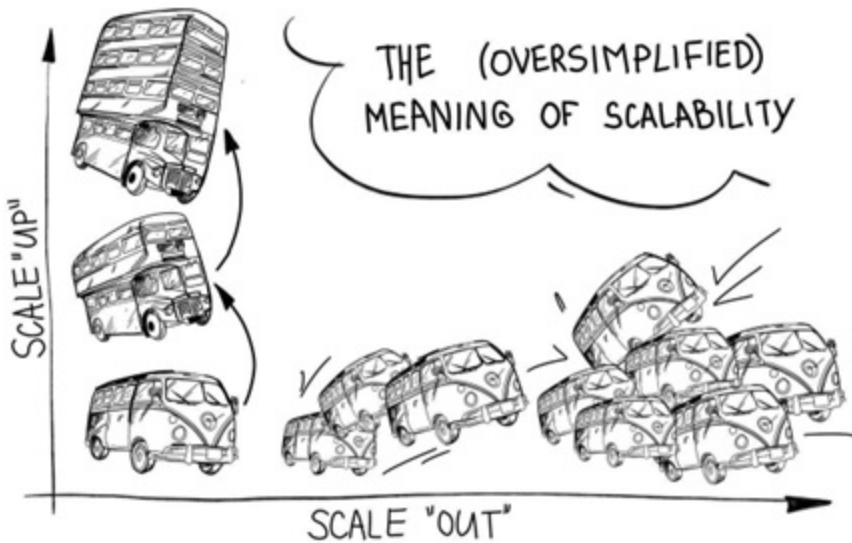
#### **Scalability**

A problem's size involves *scalability* or the characteristic of a system that can increase performance by adding more resources. Ways to increase the scalability of systems can be divided into two types: vertical and horizontal.

*Vertical scaling (scaling “up”)* increases the performance by upgrading existing processing resources by increasing the amount of memory or replacing a processor with a more powerful one. In this case, scalability is limited since it is very difficult to increase the speed of individual processors making it is very easy to hit the performance ceiling. Upgrading to more powerful processing resources is also expensive (i.e. buying a super computer), as we have to eventually pay an increasing price for smaller and smaller gains for top tier cloud instances or hardware.

Decreasing processing time associated with a particular work unit will get you so far, but ultimately, you'll need to scale out your system. *Horizontal scaling (scaling “out”)* involves increasing the program or system performance by distributing the load between the existing and new processing resources. As long as it is possible to increase the number of

processing resources, it is possible to increase system performance. In this case, scalability problems won't arise as quickly as in the case of vertical scaling.



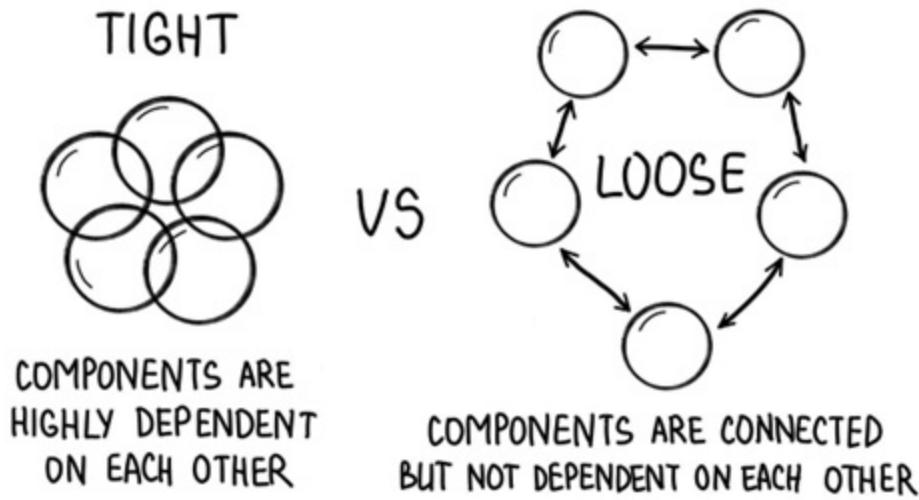
The industry decided to migrate towards a horizontally scalable approach. This trend is driven by demand for real-time systems, high volumes of data, reliability through redundancy, and improved utilization through resource sharing due to migration to cloud/SaaS environments.

Horizontal scaling requires system concurrency, and one computer may not be enough. Multiple interconnected machines, called *computing clusters*, solve data processing tasks in a reasonable time.

## Decoupling

Another aspect of large problems is complexity. The complexity of systems, unfortunately, does not decrease over time without some effort on the part of the engineers. Businesses want to make their products more powerful and functional. This inevitably increases the complexity of the code base, infrastructure, and maintenance efforts. Engineers have to find and implement different architectural approaches to simplify the systems and divide them into simpler independent communicating units.

Separation of duties is almost always welcome in software engineering. A basic engineering principle is to create loosely coupled systems, called "*Divide and conquer*". Grouping related code (*tightly coupled* components) and separating out unrelated code (*loosely coupled* components), makes the applications easier to understand and test and contain fewer bugs... at least in theory.



Another way of looking at concurrency is that it is a decoupling strategy. Dividing the functionality between modules or units of concurrency helps focus individual pieces, makes them maintainable, and reduces the overall system complexity. Software engineers decouple what gets done from when it gets done. That dramatically improves the performance, scalability, reliability, and internal structure of an application.

Concurrency is important and widely used in modern computing systems, operating systems and large distributed clusters. It helps model the real world, maximizes the efficiency of systems from users and developers' perspectives, and allows developers to solve large, complex problems.

As we explore the world of concurrency, the journey will change the way you think about computer systems and their capabilities. This book will reveal the “lay of the land” as we learn about the different layers of concurrency.

## 1.3 Layers of concurrency

As with most complex design problems, concurrency is built using multiple layers. In a layered architecture, it is important to understand that contradictory or seemingly mutually exclusive concepts can coexist at different levels *concurrently*. For example, it's possible to have concurrent execution on a sequential machine.

I like to think of concurrency layered architecture as a symphony orchestra that plays, say, Tchaikovsky:

- At the top we have the conceptual or design layer (*Application layer*): We can think of this as the composer's composition within the orchestra; within a computer system, like musical notation, algorithms tell the components of the system what should be done.



- Next, we have multitasking at runtime (i.e. *runtime system layer*): We can think of this as the musicians all playing different portions of the composition, using different instruments cooperatively. The music flow moves from one group to another, following the conductor's instructions. Within a computer system, various processes each do their part to achieve an overall purpose.



- Low-level execution (*Hardware layer*): Here we zoom in on one specific instrument: the violin. Each note produced by a violinist is the result of a string or set of one to four strings oscillating at a specific frequency determined by the length, diameter, tension, and density of the wire. Within a computer system, a single process performs tasks as dictated by instructions specific to that process.



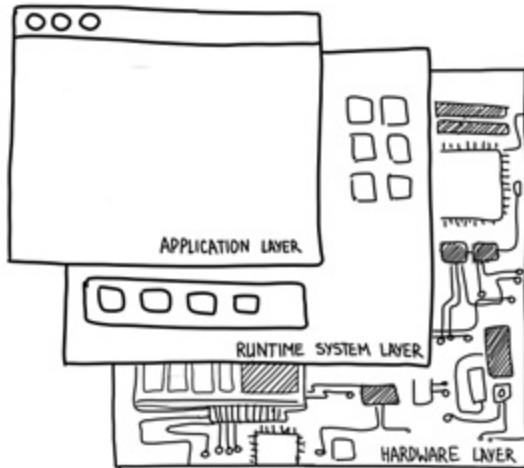
Each of those layers describe the same process at different levels, but the specifics involved are different and sometimes contradictory.

The same happens in concurrency:

- At the *hardware layer*, we directly encounter machine instructions executed by the processing resources using signals for access to

hardware peripherals. Modern architectures continue to increase in complexity. Because of this, optimizing application performance on these architectures now requires a deep understanding of the application's interactions with the hardware components.

- Moving to the *runtime system layer*, many of the shortcomings associated with programming abstractions are hidden behind mysterious system calls, device drivers and scheduling algorithms, which in themselves have a significant impact on concurrent system and therefore require a thorough understanding. This layer frequently presented by operating system which will be described in some detail in Chapter 3.
- Finally, at the *application layer*, abstractions that are closer in spirit to how the physical world operates become available. Software engineers have source code that can implement complex algorithms and represent business logic. It can also modify the execution flow using programming language features, and generally represent very abstract concepts that only a software engineer can think of.



We will use these layers extensively as a travel guide while moving up the ladder of knowledge about concurrency.

## 1.4 What you'll learn from this book



Concurrency has earned itself a reputation as a hard field. Some part of its complexity lies in the lack of written wisdom from experienced practitioners. Oral tradition instead of formal writing has left this area shrouded in mystery. Therefore, in an effort to make this area less mysterious, this book was written.

This book won't teach you everything you will ever need to know about concurrency. It will get you started and help you understand what you need to learn more about. We will explore the issues involved in concurrent programming and gain insight into the best practices needed to create concurrent and scalable applications.

Beginner and intermediate programmers will get a basic understanding of how to write concurrent systems. To get the most out of it you should have some programming experience, but you don't need to be an expert. Concrete examples will explain the key concepts in general terms, and then we demonstrate them in action using the Python programming language.

This book is organized into three parts, covering different levels of concurrency. The first part will discuss fundamental concepts and primitives of writing concurrent programs, covering knowledge from the hardware layer

to the application layer.

The second part of the book will focus on designing concurrent applications and popular concurrent patterns. It will also cover how to avoid common concurrency issues that arise when building concurrent systems.

The third part of the book will expand on our knowledge of concurrency beyond a single machine and delve into the scaling our application to multiple machines connected via a network. We will explore asynchronous communication between tasks, which is a crucial aspect in this context. Additionally, we will provide a step-by-step guide on how to write a concurrent application.

By the end of the book, you will get up to speed on concurrency, modern asynchronous and concurrent programming approaches. We will move from low-level hardware operations to a higher level of application design and translate theory into practical implementation.

All the code in the book is written in Python 3.9 programming language and tested on MacOS and Linux operating systems. The narrative is not tied to any specific programming language but references the Linux kernel subsystem. All the source code for examples can be found in the [github repository](#).



## 1.5 Recap

- A *concurrent system* is a system that can deal with many things at once.
- In the real world, many things are happening *concurrently* at any given time. If we want to model the real world, we will need concurrent programming.
- Concurrency enhances the *throughput* and performance of the system drastically by reducing or hiding *latency*, utilizing the existing resources in a more efficient way.
- We were introduced to two concepts that will be used across the book: scalability and decoupling.
  - Scalability can be vertical or horizontal. *Vertical scaling* increases program and system performance by upgrading the existing processing. *Horizontal scaling* increases performance by distributing the load between the existing and new processing resources. The industry migrated towards a horizontally scalable approach to scaling architecture, which is a prerequisite for the need for concurrency.
  - Complex problems can be decoupled into simple components and linking them together. And concurrency in a way is a *decoupling strategy* that can help us solve large and complex problems.

- A journey to an unfamiliar place usually requires a map if we want to find our way without getting lost; in this book we will navigate using layers of concurrency: *application layer*, *runtime system layer* and *hardware layer*.

[1] Herb Sutter, "The free lunch is over," blog post,  
<http://www.gotw.ca/publications/concurrency-ddj.htm>

# 2 Serial and parallel execution

## In this chapter

- You will learn the terminology on how to talk about a running program
- You learn different approaches at the lowest layer of concurrency – physical tasks execution
- You draft your first parallel program
- You learn the limitations of parallel computing approach

For thousands of years (well, not quite, but for a long time), developers have been writing programs using the simplest model of computation: the sequential model. The serial execution approach is at the core of sequential programming, and this will be our starting point in our introduction to concurrency. We will introduce different execution approaches, which lie at the low-level execution layer.

## 2.1 Review: What is a program?

The first problem with concurrency, and computer science in general, is that we're extremely bad at naming things. We sometimes use the same word to describe several distinct concepts, different words to describe the same thing, or even different words to describe different things where the meanings depending on context. And sometimes we just make words up altogether.

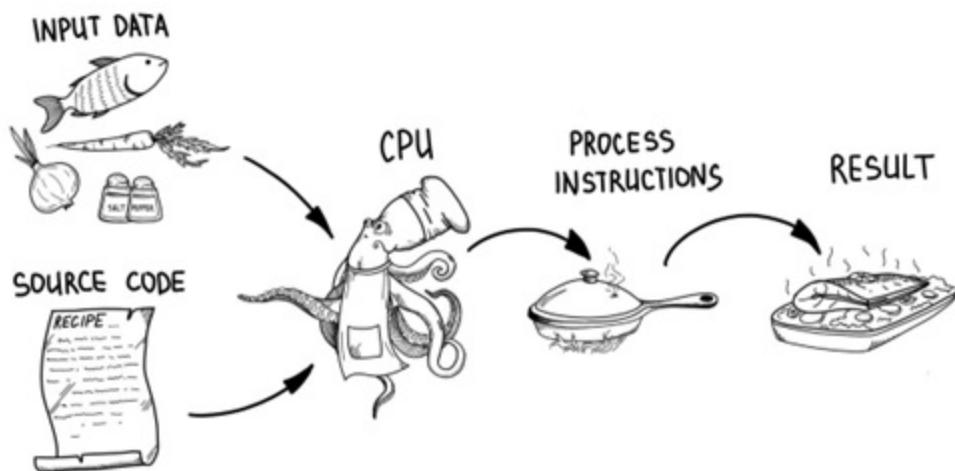
### NOTE

Did you know that CAPTCHA is a contrived acronym for "Completely Automated Public Turing test to tell Computers and Humans Apart"?

So, before we start looking at execution, it would be helpful to understand what is being executed and to establish the general terminology we will use in this book. Generally speaking, a *program* is a sequence of instructions that a computer system performs or *executes*.

Prior to execution, a program must be written first. This is done by writing *source code* using one of many programming languages. The source code can be thought of as a recipe in a cookbook – a set of steps that helps the cook to make a meal from raw ingredients. There are many components to cooking: the recipe itself, the cook, and the raw ingredients.

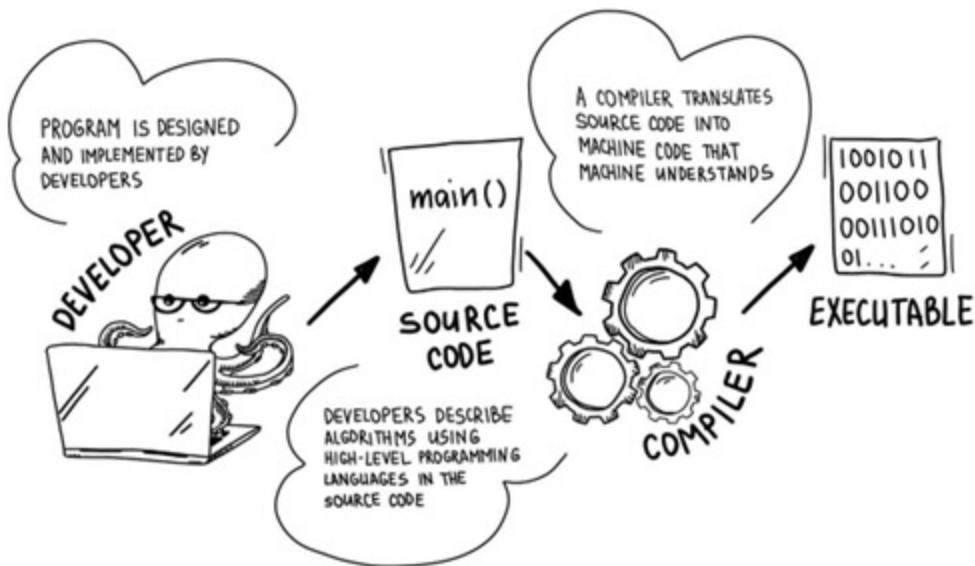
Executing a program is very similar to executing a recipe. We also have the source code of the program (the recipe), we have the chef (the processor, aka *CPU*), and the raw ingredients (the input data of the program).



The processor cannot solve a single meaningful task on its own. It can't sort things or search for objects with some specific characteristics; a processor can only do a limited number of very simple instructions. All their “intellectual” power is determined by the programs they are executing. No matter how much processing power you have, you will not accomplish anything unless that power is given direction. Turning a task into a set of steps that can be executed on a processor is what a *developer* does, not unlike the writer of a cookbook.

Developers normally describe the task they want to accomplish by using a programming language. However, the CPU cannot actually understand the source code written in a normal programming language. First, the source code has to be translated into machine code, which is the language the CPU speaks. This translation is done by special programs called *compilers*. A compiler creates a file, often called an *executable*, with a machine-level

instructions that the CPU can understand and execute.



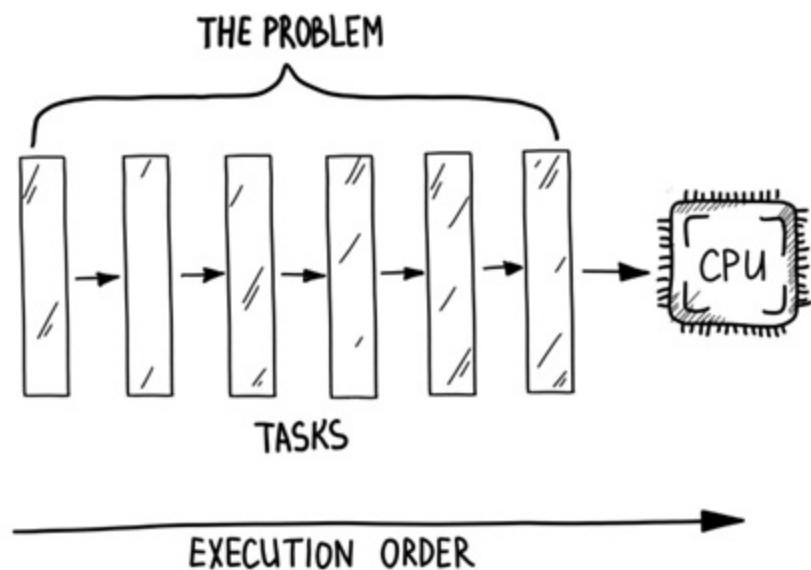
The CPU can take a few different approaches when it executes the machine code. The most fundamental approach for handling multiple instructions is *serial execution*, which is at the heart of *sequential* computing, which we'll look at now.

## 2.2 Serial execution

As stated earlier, the program is the list of instructions, and, generally, the order of that list matters. Back to our recipe example, suppose you started cooking your favorite recipe and you did all the steps that the recipe told you but in the wrong order. For example, maybe you cooked your egg before you mixed it into the flour. It's likely you would probably not be happy with the outcome. For many tasks, the order of the steps matters a lot.

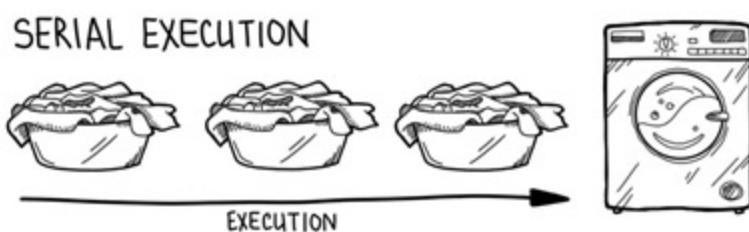
The same is true with programming. When we solve a programming problem, we first divide the problem into a series of small tasks and execute these small tasks one after another, or *serially*. Task-based programming allows us to talk about computations in a machine-independent manner and provides a framework for constructing programs modularly.

A task can be thought as a piece of work. If we are talking about CPU execution, we might call that task an *instruction*. A task can also be some sequence of operations forming an *abstraction* of a real-world model, such as writing data to a file, rotating an image, or printing a message on a screen. A task can contain a single operation or many, which we will talk more about in the next chapters, but it is a logically independent chunk of work. We will use the term “task” as a general abstraction for the unit of execution.



The *serial* execution of tasks is a sort of chain, where the first task is followed by the second one, the second is followed by the third, and so on, without overlapping time periods.

Imagine that today is laundry day and you have a pile of laundry to wash. Unfortunately, as in many apartments, you have only one washing machine and you warily remember how you once washed your favorite white T-shirt with a colored shirt. Tragic!



Chastened by that mistake, you start by washing white laundry in the washing machine, followed by washing dark laundry, then sheets, and finally towels. The minimum time in which anybody can do laundry is determined by the speed of the washing machine and the amount of laundry. Even if we have a ton of laundry to wash we still have to do it *serially*, one pile after another. Each execution is blocking the entire processing resource; it can't wash half of the white clothes then start washing dark clothes. That's not the behavior you expect from the washing machine.

## 2.3 Sequential computations

On the other hand, to describe dynamic, time-related phenomena, we use the term *sequential*. This is a conceptual property of a program or a system. It's more about how the program or system has been designed and written in the source code and not during the actual execution.



Imagine that you need to implement a Tic-Tac-Toe game.

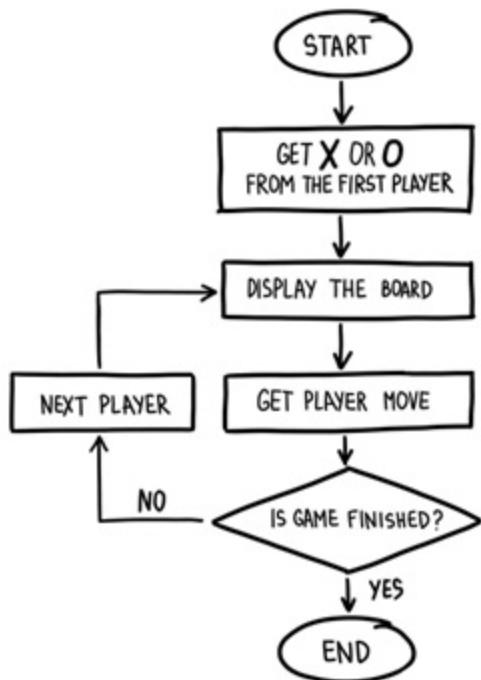
The rules of the game are simple enough: there are two players, one of the players chooses 0 and the other X to mark their respective cells. Players take turns putting their X or 0 on the board one after another. If a player gets his three marks on the board in a row, a column or one of the two diagonals, he wins. If the board is full and no player wins, the game ends in a draw.

Can you write such a game?

Let's discuss the game logic. Players take turns by typing the row number and column number in which they want to make a move. After a player makes a move, the program checks if this player has won or if there is a tie, and then switches to the other player's turn. The game proceeds in this way until one of the players wins or there is a draw. If one of the players wins, the program

displays a message saying which player won, and then the user presses any button to exit the program.

This illustration shows how a diagram of the game Tic-Tac-Toe might look like.



We see that the program has serial steps to solve the problem. Each step relies on the result from the previous step. Hence, each step is *blocking* the execution of the subsequent steps. We can only implement such a program using a sequential programming approach.

As you can see, the computational model of the program here is determined by the rules of the game – the algorithm. There is a clear dependency between the tasks that cannot be broken down in any way. We can't check the move that wasn't made yet by the player and we can't give the first player two moves in a row, since that would be cheating.

#### NOTE

In fact, there are not many tasks where the next step depends on the

completion of the previous steps. Hence it is comparatively easy to exploit concurrency in majority of the programming problems that developers facing every day. We going to talk about that in the subsequent chapters.

What tasks can you think of where serial execution is required? Hint: That means that no step cannot execute until the previous step has completed.

The opposite of sequential programming is *concurrent programming*. Concurrency is based on the idea that there are *independent* computations that can be executed in an arbitrary order with the same outcome.

### **2.3.1 Pros and cons of sequential computing**

Such a sequential computation has several important advantages, but also comes with pitfalls.

#### **Simplicity (PRO)**

Any program can be written in this paradigm. It's a very clear and predictable concept, so it's the most common one. When we think of tasks, it is very natural to consider a sequence of tasks. Cook first, then eat, and then wash the dishes is a reasonable sequence of tasks. Eating first, then washing the dishes, and then cooking is making less sense.

It is a straightforward approach, with a clear set of step-by-step instructions about what to do and when to do it. The execution guarantees that there is no need to check whether a dependent step has completed or not – the next operation will not start executing until the previous one finishes its execution.

#### **Scalability (CON)**

Scalability is the ability of a system to handle an increasing amount of work or its potential to be increased to accommodate growth. A system is considered scalable if performance improves after adding more processing resources. In the case of sequential computing, the only way to scale the system is to increase the performance of system resources used – CPU, memory, etc. That is vertical scaling, and it is limited by the performance of

available CPU on the market.

## Overhead (CON)

In the sequential computing, no communication or synchronization is required between different steps of the program execution. But there is an indirect overhead of the underutilization of available processing resources – even if we are happy with the sequential approach inside the program, we may not use all available resources of the system, leading to decreased efficiency and unnecessary costs.

Even if the system has a single one-core processor it can still be underutilized, we will go deeper into why in Chapter 6.

## 2.4 Parallel execution

If you're familiar with gardening, you may be aware that growing a tomato plant typically takes around four months. With this in mind, consider the following question: is it true or false that you can only grow three tomatoes in a year?

Clearly, the answer is false, because you can grow more than one tomato at a time.

As we've seen, in serial execution only one instruction is done at a time. Sequential programming is what most people learn first, and most programs are written that way: execution starts at the beginning of the main function and proceeds one task/function call/operation at a time serially.

When you remove the assumption that you can only do one thing at a time, you open up the possibility of working in *parallel*—just like growing more than one tomato plant. However, programs that can do things *in parallel* can be more difficult to write. Let's start with a simple analogy.

### 2.4.1 How to speed up the process of doing the laundry

Congratulations! You just won free tickets to Hawaii in the lottery. Sweet!

But there's a catch – you have a couple of hours before your plane, and you need to do four loads of laundry. Your washing machine, no matter how much you love it, cannot wash more than one load at a time, and you do not want to mix the laundry.

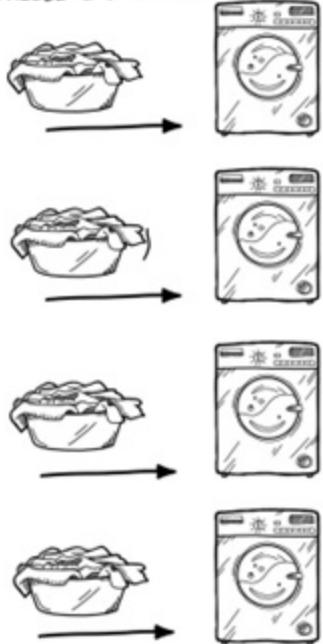


With programming, as with laundry, the time it takes for a sequential program to run is limited by the speed of the processor and how fast it can execute that series of instructions. But what if we use more than one washing machine? Because each load is independent of any other laundry load, if we have multiple machines, we can cope with the task much faster, right?

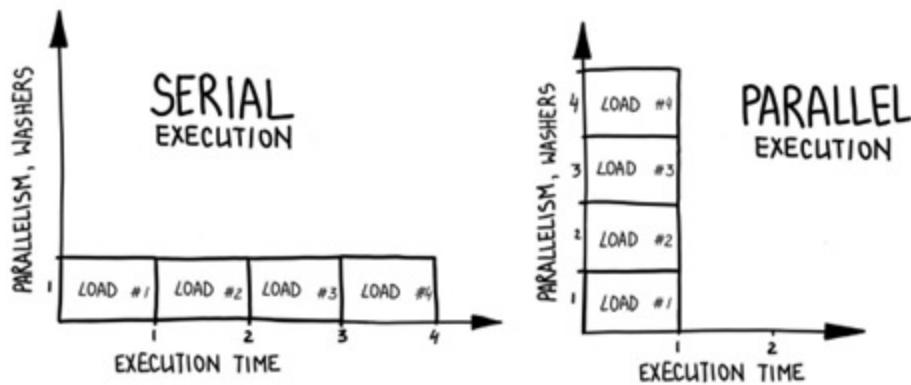
So you decide to visit the nearest local laundromat. There are a bunch of washing machines there, and you could easily wash all four of your loads in four separate machines, all at the same time. In this case, you could say that all the washing machines are working in *parallel* – more than one load is being washed at a time. Thus, you have increased the throughput by four times.

Remember the horizontal scaling we talked about in the first chapter? Here we have applied this approach.

### PARALLEL EXECUTION



*Parallel execution* means that task execution is physically simultaneous. Parallel execution is the opposite to serial execution. *Parallelism* can be measured by the number of tasks that can be executed in parallel. In our case, we have four washing machines, so the parallelism is equal to four.



Now that we know what parallel execution is, we need to understand what the requirements are for it to be possible.

## 2.5 Parallel computing requirements

Before we go deeper into parallel execution, let's first consider the requirements to achieve it: task independence, and hardware support.

### 2.5.1 Task independence

In sequential computing, all operations are accelerated by increasing the CPU clock speed. This is the simplest solution to the latency reduction problem. It does not require any special program design. All you need is a more powerful processor. Parallel computing is mainly used to decrease latency by dividing a problem into tasks that can be executed concurrently and independently of each other.

#### NOTE

Large programs often consist of many smaller ones. For example, a web server processes requests from web browsers and responds with HTML web pages. Each request is handled like a small program, and it would be ideal if such programs could run simultaneously.

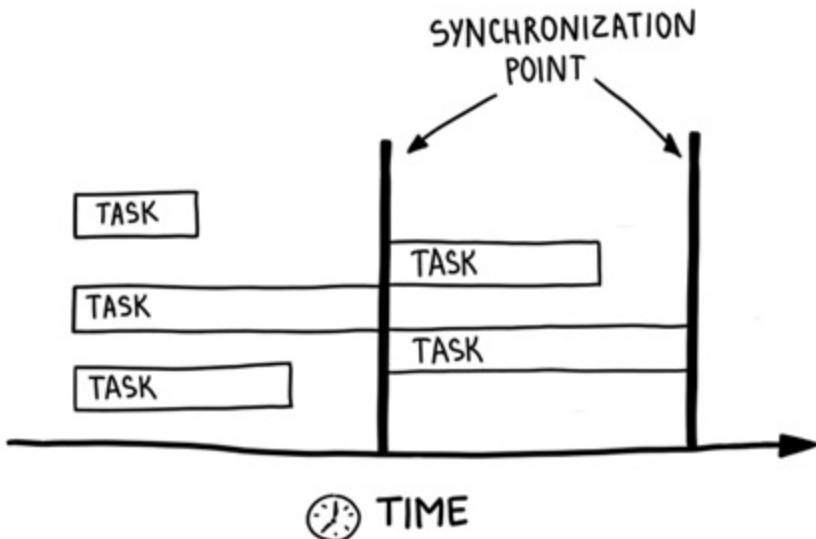
The use of parallel computing is problem-specific. To apply parallel computing to a problem, it must be possible for the problem to be decomposed into a set of independent tasks so that each processing resource can execute part of the algorithm simultaneously with the others. Independence here means that processing resources can process tasks in whatever order it likes and wherever it likes, as long, as a result is the same. Non-compliance with this requirement makes the problem non-parallelizable.

The key to understanding if a program can be executed in parallel is to analyze which tasks can be decomposed and which tasks can be executed independently. We will talk more about how to do decomposition in Chapter 7.

#### NOTE

It should be noted that in this case, the logic goes only in one direction – a program that can run in parallel can always be made sequential, but a sequential program cannot always be made parallel.

Task independence is not always possible because not every program or algorithm can be divided into independent tasks from start to finish. Some tasks can be independent, some cannot if they depend on previously executed tasks. That requires developers to *synchronize* different dependent pieces of a program to get the correct results. Synchronization means *blocking* the execution of the task waiting for dependencies. With Tic-Tac-Toe example the program execution is blocked by individual player's moves. Coordinating interdependent parallel computations via synchronization can severely limit the parallelism of the program, presenting a significant challenge in writing parallel programs compared to simple sequential programs (which will be discussed in more detail in Chapter 8).



That extra work can be worth the effort. When done right, parallel execution increases the overall throughput of a program, enabling us to break down large tasks to accomplish them faster, or accomplish more tasks in a given time frame.

Tasks that require little or no synchronization are sometimes called *embarrassingly parallel*. They can very easily be broken down into independent tasks executed in parallel. Such tasks are often found in scientific computing. For example, distributing the work of finding prime numbers can be done by allocating subsets to each processing resource.

## **NOTE**

There is no shame in having an embarrassingly parallel task! On the contrary, having an embarrassingly parallel application is cool because it is easy to program. In recent years, the term "embarrassingly parallel" has taken on a slightly different meaning. Algorithms that are embarrassingly parallel tend to have very little communication between processes, which is the key to good performance, so the term "embarrassingly parallel" usually refers to an algorithm with low communication needs. We will touch on this a little more in Chapter 5.

Thus, the extent of parallelism depends more on the problem than on the people trying to solve it.

### **2.5.2 Hardware support**

Parallel computing requires hardware support. Parallel programs need hardware with multiple processing resources. Without at least two processing resources, we cannot achieve true parallelism. We will talk about the actual hardware and how it can support multiple simultaneous operations in the next chapter.

Having all the requirements to parallel computing we are now ready to explore what that actually is.

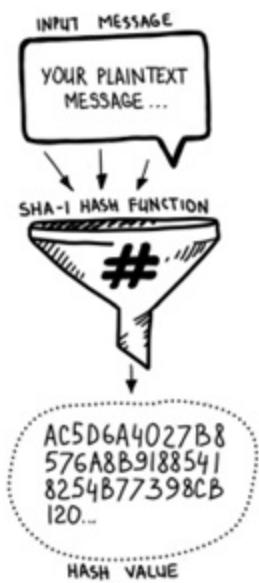
## **2.6 Parallel computing**

*Parallel computing* uses decomposition to split large or complex problems into small tasks and then utilizes parallel execution of the runtime system to solve it effectively. In the next example, let's demonstrate how parallelism can save the world.

Imagine that you are working at the FBI IT department, and, for the next mission, you must implement a program to crack the password (number combination of particular length) and access to a system that can destroy the whole world.

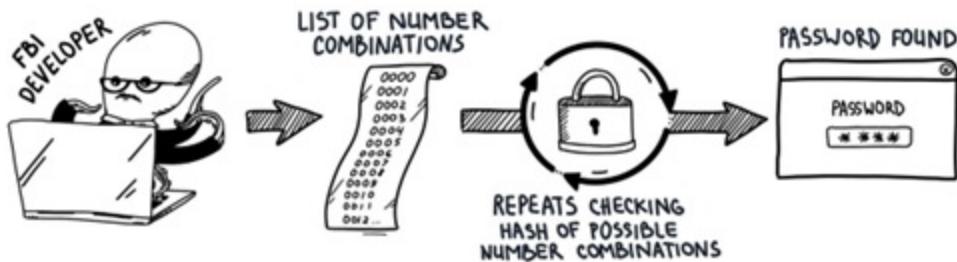


The usual approach for finding a correct password consists of repeatedly guessing the password (known as the brute-force approach), calculating its scrambled form (cryptographic hash), and comparing the resulted cryptographic hash with the one stored on the system. Let's assume that we already have a cryptographic hash of the password.



How do you implement such a program?

Brute force is understood as a general method of finding a solution to a problem that requires listing all possible combinations, iterating over this list, and checking whether each particular solution solves the problem. In our case, we need to go through a list of all possible number combinations, and check whether its cryptographic hash corresponds to the hash found on the system.



After a couple of sleepless nights, you figured out how to check the cryptographic hash, went through the whole possible number combinations, and finished implementing the program using your favorite programming language. The algorithm generates a number combination and checks the cryptographic hash. When it matches, the found password is printed out, and the program finishes; if not, it goes to the next combination and does the cycle again.

Essentially, we process all possible password combinations one by one, using sequential computing. Here we let our CPU process one task at a time and then get the next task and do it serially until all the tasks have been completed. Steps how we use serial execution to solve a problem are shown in the previous figure. Or in the code:

```
# Chapter 2/password_cracking_sequential.py
import time
import math
import hashlib
import typing as T

def get_combinations(*, length: int, min_number: int = 0) -> T.List[str]:
    combinations = []
    max_number = int(math.pow(10, length) - 1)
    for i in range(min_number, max_number + 1): #A
        str_num = str(i) #A
        zeros = "0" * (length - len(str_num)) #A
        combinations.append("".join((zeros, str_num))) #A
    return combinations

def get_crypto_hash(password: str) -> str:
    return hashlib.sha256(password.encode()).hexdigest()

def check_password(expected_crypto_hash: str,
```

```

        possible_password: str) -> bool:
actual_crypto_hash = get_crypto_hash(possible_password) #B
return expected_crypto_hash == actual_crypto_hash #B

def crack_password(crypto_hash: str, length: int) -> None:
    print("Processing number combinations sequentially")
    start_time = time.perf_counter()
    combinations = get_combinations(length=length) #C
    for combination in combinations:
        if check_password(crypto_hash, combination): #C
            print(f"PASSWORD CRACKED: {combination}") #C
            break #C

    process_time = time.perf_counter() - start_time
    print(f"PROCESS TIME: {process_time}")

if __name__ == "__main__":
    crypto_hash = "e24df920078c3dd4e7e8d2442f00e5c9ab2a231bb3918d"
    length = 8
    crack_password(crypto_hash, length)

```

Output would look similar to this:

```

Processing number combinations sequentially
PASSWORD CRACKED: 87654321
PROCESS TIME: 64.60886170799999

```

In absolute confidence of your heroism in solving the problem, you give the program to the next hero who will go on the mission – agent 008 nods his head and finishes his vodka martini.

We all know how spies don't trust anyone, right? Agent 008, in less than an hour, bursts into your office and tells you that the program is taking too long – according to his calculations, it will take an hour to process all the possible password combinations on his super device! “I have a couple of minutes before the building will burst into flames,” said he fearfully, sipping the vodka martini again – He leaves the room with a parting command: “Speed it up!”. Ouch!



How do you speed up the program like this?

The obvious way would be to increase the performance of the CPU – by increasing the clock speed of the super device we can process more passwords in the same amount of time. Unfortunately, we have already discovered that this approach has limits – there is a physical limit on how fast the CPU can be. And, after all, it's the FBI. We have the fastest processor right now. There is no way we can increase the performance of it. This is the most significant disadvantage of sequential computing. It is not easily scalable, even if we have more than one processing resource on the computer system.

Another way to make the program's execution faster is to break it down into independent parts and distribute those tasks among multiple processing resources so they can be processed simultaneously. The more processing resources and smaller tasks we have, the faster the processing goes. This is the core idea behind parallel computing, and something we'll look at in more detail in chapters 8 and 12.

Do you think we can use parallel execution here? The super device uses a top-tier CPU with a lot of cores. So, the first condition is fulfilled – we have proper hardware that can execute tasks in parallel.

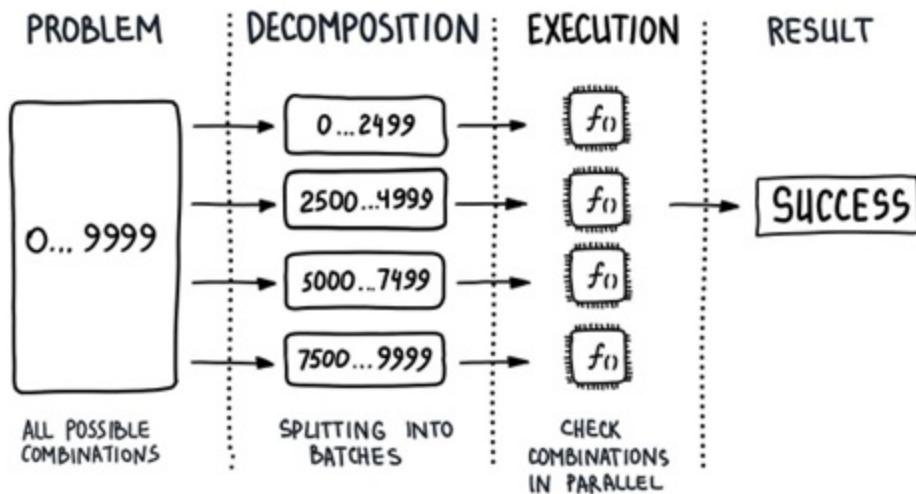
Is it possible to decompose the problem into independent tasks? Checking individual password combinations can be thought of as a task, and they are not dependent on each other – we don't need to check all previous passwords

before checking the current one. It does not matter which password is processed first as they can be executed entirely independent of each other as long as we find the right one. Great!

So, we have all the requirements for parallel computing met. We have hardware support and task independence. Let's design a final solution!

The first step for every such problem is to decompose the problem into separate tasks. As we've already discovered individual password checks can be thought as independent tasks and we can execute them in parallel. There are no dependencies here, which leads to no synchronization points, hence, it is embarrassingly parallel problem.

In the illustration below, you can see the diagram of the solution split into several steps. The first step is creating ranges of passwords (chunks) to check for each individual processing resource. Then we distribute those chunks between available processing resources. As a result, we have a set of passwords ranges that are assigned to each processing resource. The next step would be to run the actual execution.



In the code:

```
# Chapter 5/password_cracking_parallel.py
ChunkRange = T.Tuple[int, int]
```

```

def get_chunks(num_ranges: int, length: int) -> T.Iterator[ChunkR
    max_number = int(math.pow(10, length) - 1) #A
    chunk_starts = [int(max_number / num_ranges * i) for i in #A
                    range(num_ranges)] #A
    chunk_ends = [start_point - 1 for start_point in chunk_starts
                  max_number] #A
    return zip(chunk_starts, chunk_ends) #A

def crack_password_parallel(crypto_hash: str, length: int) -> Non
    num_cores = cpu_count() #B
    chunks = get_chunks(num_cores, length)

    # DO IN PARALLEL #C
    # for chunk_start, chunk_end in chunks: #C
    #     crack_chunk(crypto_hash, length, chunk_start, chunk_end)

```

As we can see, we have added new function `crack_password_parallel` that should execute `crack_password` function in parallel on multiple cores. It may look different in different programming languages, but the idea is still the same. It should create a set of parallel units and distribute the password ranges between them for parallel execution. This requires the use of pseudocode (human-readable representation of the logic of a program, written in a stylized language that mimics actual code) which we will discuss further in Chapter 4 and 5.

#### **NOTE**

Even if we use pseudocode in our example, it is actually very realistic in terms of usage. For example, Matlab language has a `parfor` construct that makes it trivial to use parallel for loops. Python language has a `joblib` package, which makes parallelism incredibly simple, using the `Parallel` class. R language has a `Parallel` library with the same functionality. The standard Scala library has parallel collections to facilitate parallel programming, sparing users the low-level parallelization details.

Because of parallel computing, Agent 008 once again saved the world with seconds to spare. Unfortunately, most of us don't have the resources of the FBI; parallel execution has its own limits and costs, which we need to consider before we apply it to our problems. We'll think about this in the next section.



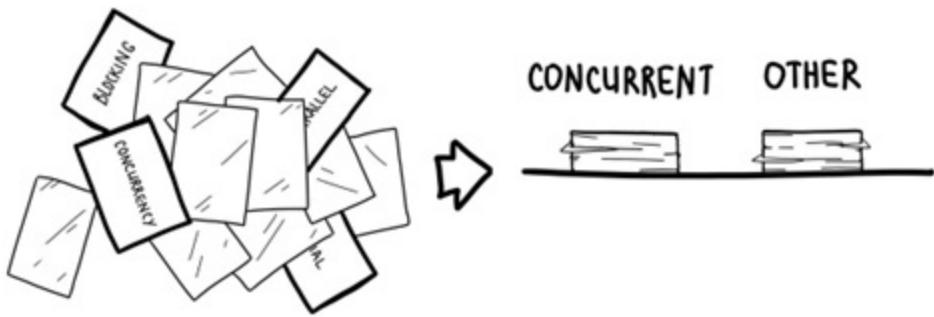
## 2.7 Amdahl's law

One mother can deliver a baby in nine months. Does this mean that nine people working together can deliver a baby in one month?

It seems that we can infinitely increase the number of processors and thus make the system run as fast as possible. But unfortunately, this is not the case. A famous observation of Gene Amdahl, known as Amdahl's law, demonstrates this clearly.

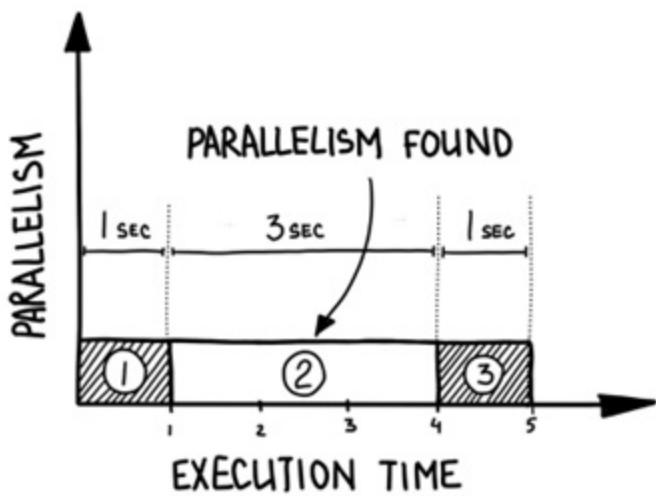
So far, we have analyzed the execution of a parallel algorithm. Although a parallel algorithm may have some sequential parts, it is common to think of execution in terms of some fully parallel parts and some fully sequential parts. Sequential parts may simply be steps that have not been parallelized, or they may be sequential in nature, as we've seen before.

Imagine you have a huge pile of index cards with definitions written on them. You want to find cards with information about concurrency and add them to a separate stack, but the cards are all mixed up. Fortunately, you have two friends with you, so you could divide up the cards, give each person a pile and tell them what to look for. Then all the friends could search their own pile of cards. Once someone finds the right card, they can announce it and put it into a separate stack.



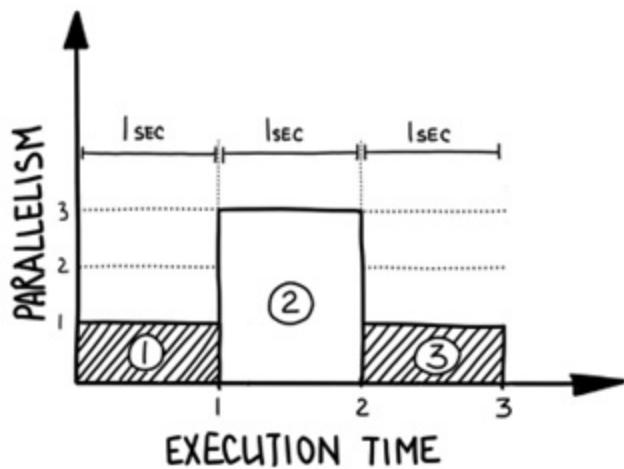
This algorithm might look like:

1. Divide pile up into stacks and hand out one stack to each person (Serial)
2. Everyone looks for the “concurrency” card (Parallel)
3. Give the card to a separate pile (Serial)

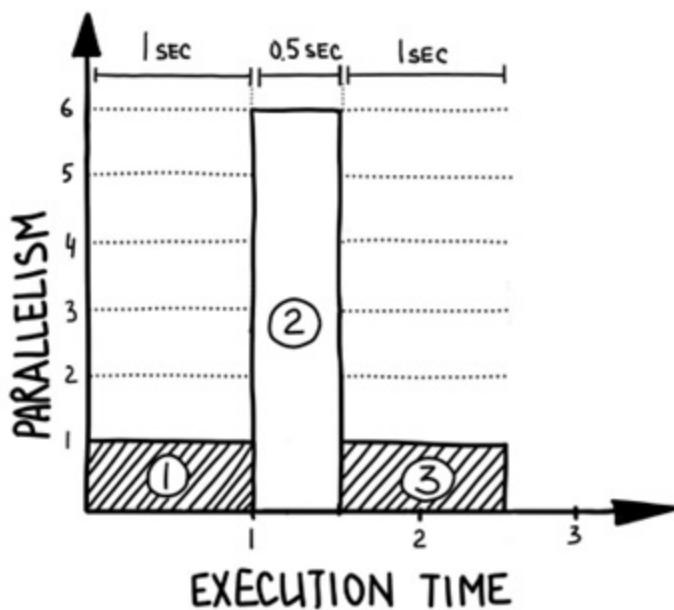


The first and last part of the above algorithm takes one second, a second part takes 3 seconds. Thus, it takes 5 seconds to execute it from the beginning to the end if you will do it yourself. The first and third parts are algorithmically sequential, and there is no way to separate them into independent tasks and use parallel execution there. But we can easily use parallel execution in the second part by dividing cards into any number of stacks, as long as we have a friend to execute this step independently. We reduced the execution time of that part to 1 second with the help of two friends. The whole program now takes only 3 seconds, which is a 40% speedup for the whole program.

Speedup here is calculated as a ratio of the time it takes to execute the program in the optimal sequential manner with a single processing resource, over the time it takes to execute in a parallel manner with a certain number of processing resources.

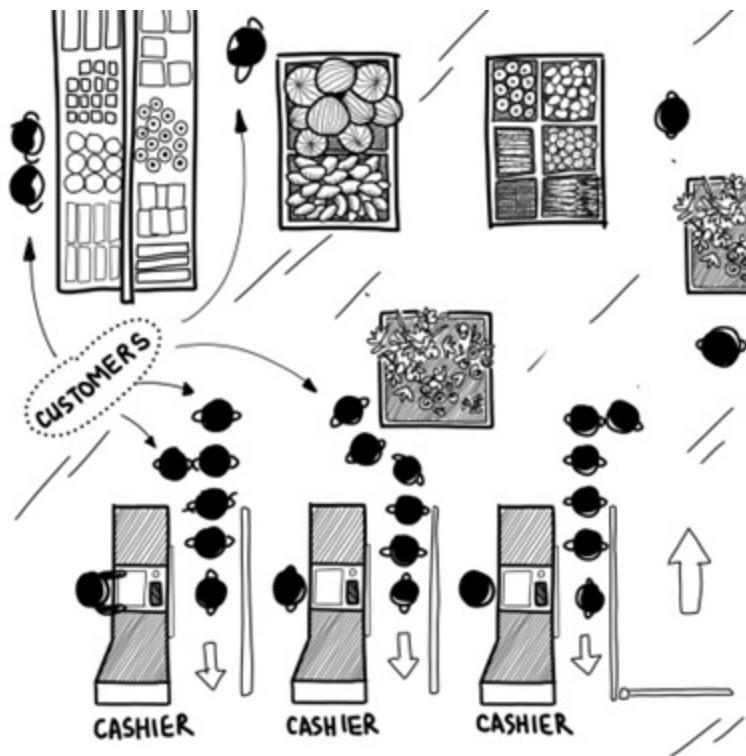


What happens if we keep increasing the number of friends? For example, suppose we added three more friends, six in total, and now the second part of the program takes only half a second to execute. Then the whole algorithm would only take 2.5 seconds to complete, which is around 50% speedup for the whole program.



Following the same logic, we can invite all the people from the city and making the parallel part of the algorithm execute instantaneously (in theory, we have a communication cost overhead, more about it in Chapter 6), we would still end up with a latency of at least 2 seconds – the serial part of the algorithm.

Your parallel program runs as fast as its slowest sequential part. An example of this phenomenon can be seen every time you go to the mall. Hundreds of people can shop at the same time, rarely disturbing each other. Then, when it comes time to pay, lines form as there are fewer cashiers than shoppers ready to leave.



The same applies to programming. Since we cannot speed up sequential parts of a program, increasing the number of resources will have no effect on their execution. This is the key to understanding Amdahl's law. The potential speed of a program using parallel computing is limited to sequential parts of the program. The law describes the maximum speedup you can expect when you add resources to the system, assuming parallel computing. Amdahl's law predicts, for our example, if we have 2/3 of a program being sequential, then no matter how many processors we have, we will never get more than 1.5x

speedup.

More formally, the law is stated using the formula:

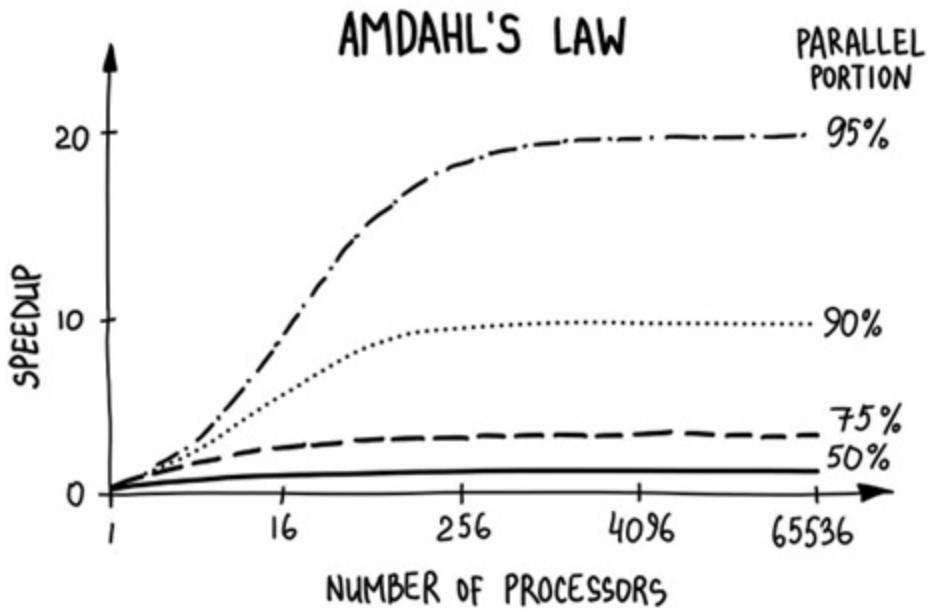
*AMDAHL'S LAW STATES THAT THEORETICAL SPEEDUP IS DETERMINED BY THE FRACTION OF CODE P THAT CAN BE PARALLELISED*

$$\text{SPEEDUP} = \frac{1}{(1-P) + P/N}$$

↑                    ↓  
SERIAL PART OF JOB = 1(100%) - PARALLEL PART      PARALLEL PART IS DIVIDED BY N WORKERS

The formula above may look harmless until you start putting in values. For example, if 33% of the program is sequential, adding a million processors can give no more than 3x speedup. You can't accelerate 1/3 of the program, so even if the rest of the program runs instantaneously, the performance gain is limited to 300%. Adding several additional processors can often provide significant acceleration, but as the number of processors grows, the advantage quickly diminishes.

The graphical representation below shows the speedup versus number of processors for different fractions of code that can be parallelized, if we do not consider algorithms or any coordination overhead.



You can also do the math the other way around – for example, with 2500 processors, what percentage of the program must be perfectly parallelizable to get a 100x acceleration? Putting the values into Amdahl's law, we get  $100 \leq 1/(S + (1 - S)/2500)$ . By getting the  $S$ , we see that only less than 1% of the program can be sequential.

To sum up, Amdahl's Law illustrates why using multiple processors for parallel computing is only really useful for programs that are highly parallelizable. Just because you can write programs to be parallel, doesn't mean you always should, because the costs and overhead associated with parallelization can sometimes outweigh the benefits. Amdahl's Law is one handy tool to estimate the benefits of parallelizing a program to determine whether or not it makes sense to do so.

## 2.8 Gustafson's law

With such disappointing results, it is tempting to abandon parallelism as a way to improve performance. You should not get completely discouraged. Parallelism does provide real acceleration of performance-critical parts of programs, but you can't speed up all parts of a program – only if it is embarrassingly parallel problem. For other tasks, there is a hard limit to the

possible gains.

But we can look at Amdahl's law from a different perspective. Our program ran in 5 seconds – what if we double the amount of work done in our parallelizable part – not 3 but 6 tasks. So, we would get 6 tasks done simultaneously, and the program would still run in 5 seconds, resulting in a total of 8 tasks done, which is 1.6x increase having just two processors. And if we continue doing this and add a couple more processors, each doing the same amount of work, we can get 11 tasks done in the same 5 seconds, 2.6x increase.

According to Amdahl's law, speedup shows how much less time it will take to execute a parallel program, assuming that the volume of the problem remains constant. However, the speedup can also be seen as an increase in the volume of the executed task in a constant time interval (throughput). Gustafson's law emerged from this assumption.

Gustafson's law gives a more optimistic perspective of parallelism limits. If we can keep increasing the amount of work, then the sequential parts have less and less effect on us, and we can see speedup in proportion to the number of processors we have.

So if you ever hear Amdahl's law cited as a reason why parallelism won't work in your case, you can go back and make the observation that Gustafson had an explanation for what to do. And that's the key to why supercomputers and distributed systems have been successful with parallelism, because we can keep increasing the volume of data.

Now we understand a lot about parallel computing, it's time to talk about how it relates to concurrency.

## 2.9 Concurrency vs parallelism

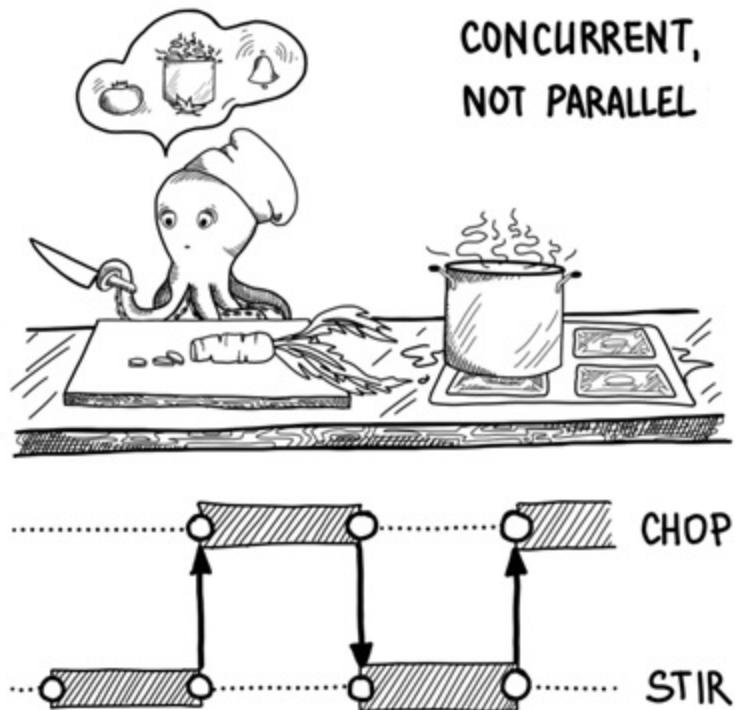
Conversational meanings of the words “parallel” and “concurrent” are mostly synonymous, which is a source of significant confusion that extends even to the computer science literature. Distinguishing between parallel and concurrent programming is very important because both pursue different

goals at different conceptual levels.

Concurrency is about multiple tasks which start, run, and complete in overlapping time periods, in no specific order. Parallelism is about multiple tasks literally run at the same time on a hardware with multiple computing resources like multi-core processor. Concurrency and parallelism are not the same thing.

Imagine that one cook is chopping salad while occasionally stirring the soup on the stove. He has to stop chopping, check the stove top, and then start chopping again, and repeat this process until everything is done.

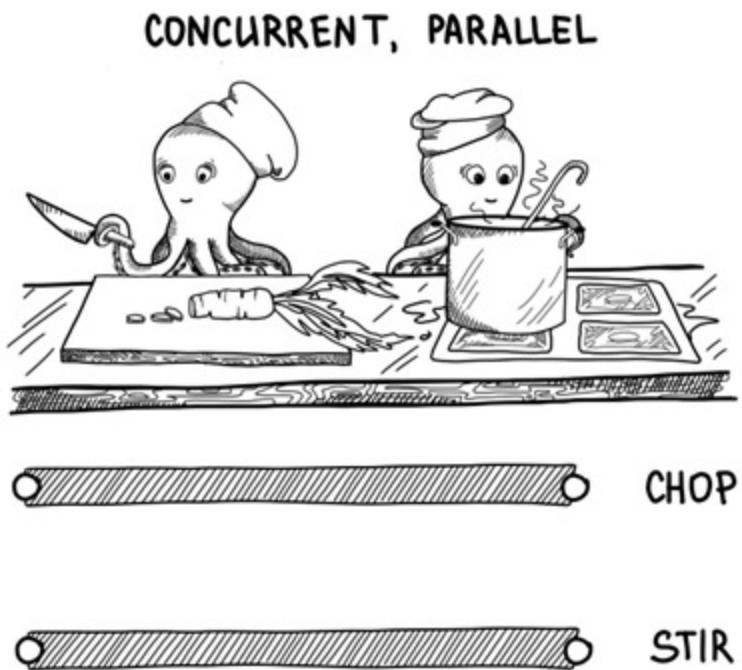
As you can see, we only have one processing resource here, the chef, and his concurrency is mostly related to logistics; without concurrency, the chef has to wait until the soup on the stove is ready to chop the salad.



Parallelism is an *implementation property*. Parallelism is literally the simultaneous physical execution of tasks at runtime, and it requires hardware with multiple computing resources. It lies on the hardware layer.

Back in the kitchen, now we have two chefs, one who can do the stirring and one who can chop the salad. We've divided the work by having another processing resource, another chef.

Parallelism is a subclass of concurrency: before you can do several tasks at once, you have to manage several tasks first.



The essence of the relationship between concurrency and parallelism is that concurrent computations can be parallelized without changing the correctness of the result, but concurrency itself does not imply parallelism. Further, parallelism does not imply concurrency; it is often possible for an optimizer to take programs with no semantic concurrency and break them down into parallel components via such techniques as pipeline processing, wide vector SIMD operations, divide and conquer, and we will be talking about some of those later in the book.

As Unix and Go programming legend Rob Pike pointed out “Concurrency is about dealing with lots of things at once. Parallelism is about doing lots of things at once.”<sup>[1]</sup> The concurrency of a program depends on the programming language and the way it is programmed, while the parallelism depends on the actual execution environment. In single-core CPU, you may

get concurrency but not parallelism. But both go beyond the traditional sequential model in which things happen one at a time.

To get a better idea about the distinction between concurrency and parallelism, consider the following points:

- An application can be concurrent but not parallel. It processes more than one task over a given period of time (i.e. “juggling” more than one task even if no two tasks are executing at the same time instant – this will be described in more detail in Chapter 6).
- An application can be parallel but not concurrent, which means that it processes multiple sub-tasks of a single task at the same time.
- An application can be neither parallel nor concurrent, which means that it processes one task at a time, sequentially, and the task is never broken into subtasks.
- An application can be both parallel and concurrent, which means that it processes multiple tasks or subtasks of a single task concurrently at the same time (executing them in parallel)

Imagine you have a program that inserts values into a hash table. In terms of spreading the insert operation between multiple cores, that's parallelism. But in terms of coordinating access to the hash table, that's concurrency. And if you still don't understand the latter, don't worry, the following chapters will explain the concept in detail.

Concurrency covers a variety of topics, including interaction between processes, sharing and competition for resources (such as memory, files, and I/O access), synchronization between multiple processes, and allocation of processor time between processes. We will see that these issues arise not only in multi-processor and distributed processing environments, but even in single-processor systems. In the next chapter we start with understanding the environment the program running in – the computer hardware and runtime system.

## 2.10 Recap

- Every problem when formulated into application is divided into series of

tasks that in the simplest case is executed serially

- A *task* can be thought of as logically independent piece of work
- *Sequential computing* means that each task in a program depends on the execution of all previous tasks in the order in which they are listed in the code
- *Serial execution* refers to a set of ordered instructions executed one at a time on one processing unit. Serial execution is required when the input to each task requires the output of a previous task
- *Parallel execution* refers to executing multiple computations at the same time. Parallel execution can be used when the tasks can be performed independently
- *Parallel computing* uses multiple processing elements simultaneously to solve a problem. This often led to significant program redesign – decomposition of the problem, creating or adapting algorithm, adding synchronization points to the program, etc
- Concurrency describes working on multiple tasks at the same time. Parallelism depends on the actual runtime environment, and it requires multiple processing resources and task independence in decomposed algorithm. The concurrency of a program depends on the programming language and the way it is programmed, while the concurrency depends on the actual execution environment.
- Amdahl's Law is one handy tool to estimate the benefits of parallelizing a program to determine whether or not it makes sense to do so
- Gustafson's Law describes how to get more work out of systems despite the limitations of Amdahl's Law

[1] Rob Pike gave a talk at Heroku's Waza conference entitled Concurrency is not parallelism: <https://go.dev/blog/waza-talk>

# 3 How computers work

## In this chapter

- You learn the details of how code is actually executed on the CPU
- You learn about functions and goals of the runtime system
- You learn how to choose the hardware suitable for your problem

Twenty years ago, a working programmer could go for years without encountering a system that had more than two processing resources at most. Today, even a mobile phone has multiple processing resources. A modern programmer's mental model needs to encompass multiple processes running on different processing resources at the same moment.

When describing concurrent algorithms, it is not necessary to know the specific programming language to implement a program. It is necessary, however, to understand the features of the computer system on which the algorithm will be executed. It is possible to construct the most effective concurrent algorithm by selecting the types of operations that most fully utilizes computer system hardware. Therefore, it is necessary to understand the potential capabilities of different hardware architectures.

Since the goal of using parallel hardware is performance, the efficiency of our code is a major issue. This, in turn, means that we need a good understanding of the underlying hardware we are programming. In this chapter we will give an overview of the parallel hardware giving us an advantage to make an informed decision when designing software.

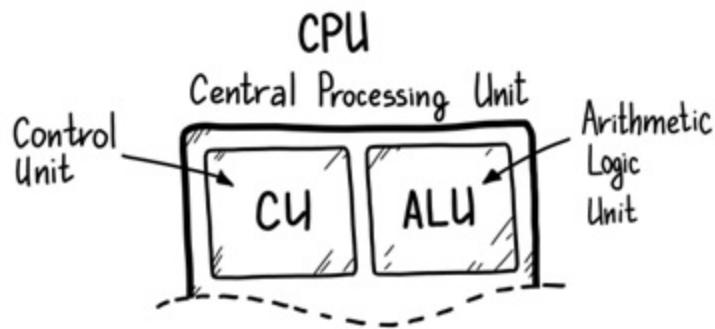
## 3.1 Processor

The term *central processing unit* (CPU) originated in the misty days of the first computers, when one massive cabinet contained the circuitry needed to interpret machine instructions and executing those instructions. The CPU also performed all operations for all connected peripheral devices like printers,

card readers, and early storage devices such as drum and disk drives.

The modern CPU has become a slightly different device, more focused on its primary task of executing machine instructions. The CPU can easily process these instructions thanks to the *Control Unit* (CU) that interprets machine instructions, and the *Arithmetic-Logic Unit* (ALU), that performs arithmetic and bitwise operations. Thanks to the CU and ALU together, the CPU processes more complex programs than a simple calculator can.

#### Components of the CPU

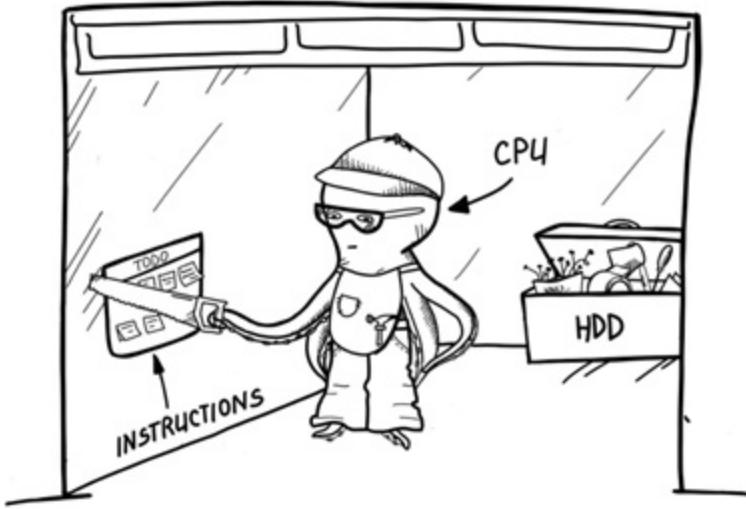


But there is another component on the CPU that plays the important role in speeding up the execution.

### 3.1.1 Cache

The cache is a temporary memory on the CPU. This chip-based feature of your computer lets you access information more quickly than from your computer's main memory.

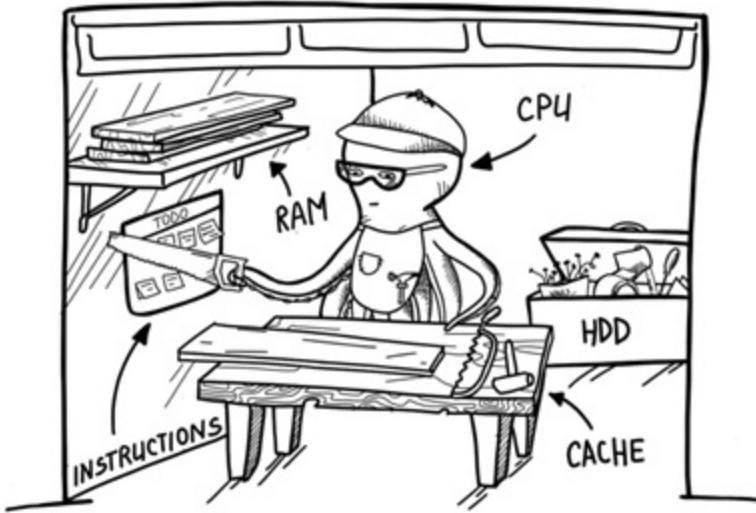
Imagine a joinery workshop where we have one joiner. The joiner (CPU) has to fulfill incoming customer requests (instructions). To make the product the customer wants, he creates nearby temporary storage for fresh wood and some resources without having to go to the warehouse where he keeps all supplies (hard disk drive or HDD).



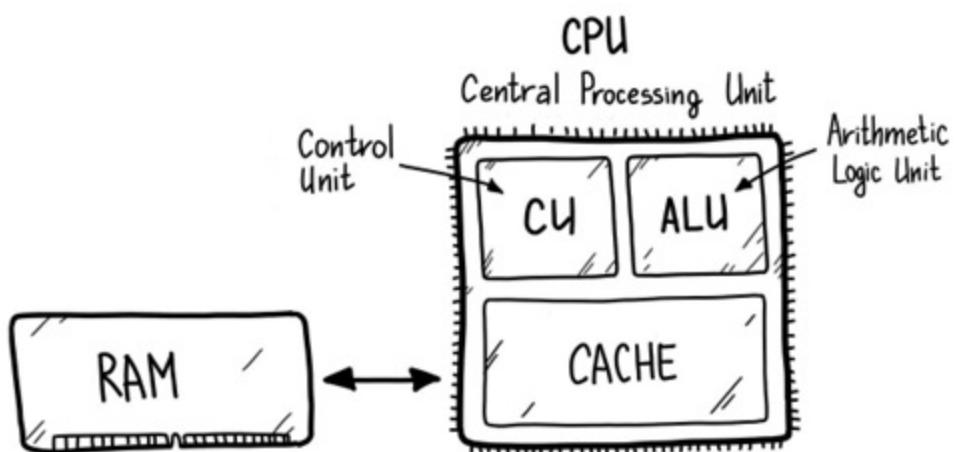
The temporary storage is the memory attached to the CPU is called random access memory (*RAM*) and is used to store data and instructions. When a program starts to run, executable files and data is copied to the RAM and stored there until the end of the program execution.

But the CPU never directly accesses RAM. The CPU's ability to perform calculations is much faster than the RAM's ability to transfer data to the CPU. Modern CPUs have one or more levels of *cache memory* to speed up access.

Going back to workshop you can think that the joiner aside from having access to temporary storage in his workshop he also always needs fast access to his tools all the time, so they should be always at hand. He stores them on his workbench where he has very fast access. You can think of the cache memory as a workbench for the processor.

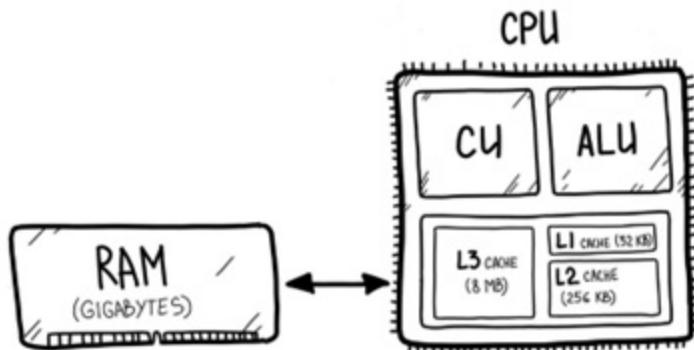


Cache memory is faster than RAM, and it is closer to the CPU because it is located on the CPU chip. Cache memory provides storage for data and instructions so that the CPU doesn't have to wait for data to be retrieved from RAM. When the processor needs data, and program instructions are also considered data, the cache controller determines if the data is in the cache and provides it to the processor. If the requested data is not in the cache, it is retrieved from RAM and moved to the cache. The cache controller analyzes the requested data, predicts what additional data will be required from RAM, and loads it into the cache.



A processor has three levels of cache: Levels 1, 2, and 3 (L1, L2, and L3).

Levels 2(L2) and 3(L3) are designed to predict what data and instructions will be needed next, moving them from RAM and closer to the processor to L1 cache so it is ready when it is needed. The bigger the level, the slower communication channel is, and bigger memory is available. The L1 cache is closest to the processor. Because of all the additional cache levels, the processor can stay busy and not waste cycles waiting for required data.



Almost all of the data access and communication adds up to the execution latency – the *communication cost*, and it's one of the biggest threats to system performance. And cache is here to mitigate or at least soften this communication costs. Let's take a look how it really affects the latency if things were scaled up into everyday units that humans can intuitively picture (this is called scaled latency):

System event	Actual latency	Scaled latency
<b>One CPU cycle</b>	0.4ns	1s
<b>L1 cache access</b>	0.9ns	2s
<b>L2 cache access</b>	2.8ns	7s

<b>L3 cache access</b>	28ns	1min
<b>Min memory access (RAM)</b>	~100ns	4min
<b>High speed SSD I/O</b>	<10 µs	7h
<b>SSD I/O</b>	50-150 µs	1.5-4d
<b>HDD I/O</b>	1-10ms	1-9months
<b>Network request San Francisco to NYC</b>	65ms	5years

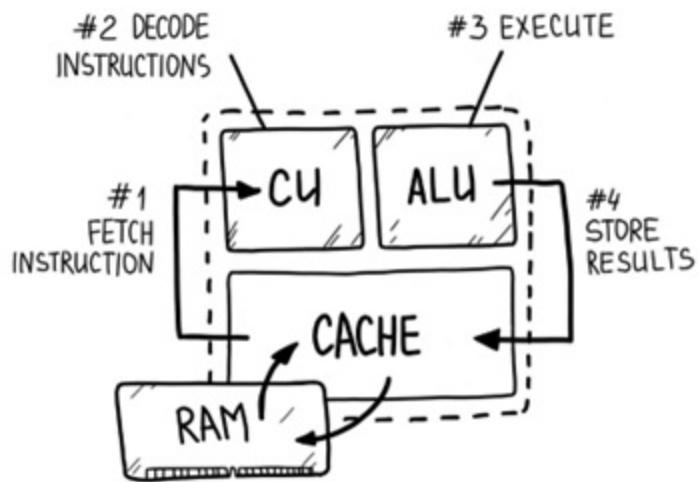
Having said that, let's take a look on the actual execution cycle.

### 3.1.2 CPU execution cycle

Back to the workshop again. Our single joiner does all the work – from communicating with customers to doing the actual woodworking. So, his work includes getting customer ideas, translating those ideas into an actual task item, executing tasks and giving the results back to the customers. The joiner spends all his time in this cycle, and that's what keeps his business running.

The CPU similarly is carrying out a continuous process of instruction execution through various stages – these sequences of stages are known as the *CPU cycle*. In their simplest form, processors operate in four different stages:

#### CPU execution cycle



1. **Fetch**. Here the Control Unit (CU) fetches the instruction from memory or cache and copies to the CPU. In this process, the CU uses various counters to understand what instruction to fetch and where to find it.
2. **Decode**. Here, the previously fetched instruction is decoded and sent for processing. There are different types of instructions that are doing different things, so depending on the type of instruction and the operation code we need to know which processing units will be sent to.
3. **Execution**. The compute instruction is then moved to ALU and started the execution.
4. **Store the result**. Once the instruction is complete, the result is written into RAM and the next instruction is starting the execution. Then the processor goes back to the first step until there are no more instructions left to fetch.

The processor spends all its time in this cycle, endlessly retrieving the next instruction, decoding, executing it and storing the result.

## 3.2 Runtime system

Working with the CPU is not a simple process. Developers have to handle everything ourselves, including the various operational tasks: controlling hardware resources, managing access to those hardware resources, managing the exact functionality that should be executed, providing isolation between programs in case of a crash, or accessing a shared resource and other

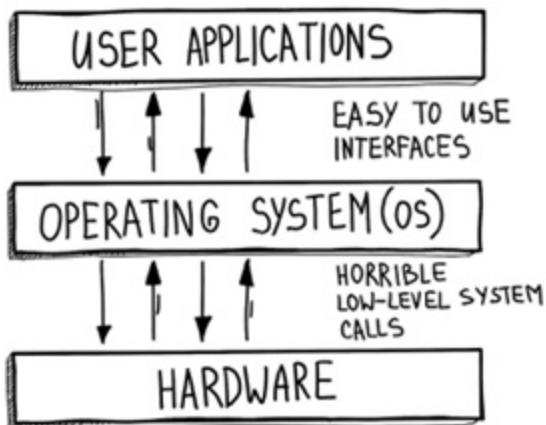
operational types of tasks.

Modern systems need to be multipurpose and hence complex. Eventually they become overgrown with many software systems related to specific management tasks. For example, file management systems, graphics management systems, task management systems, etc. These are all examples of microprogram management systems that eventually evolved into an additional level of abstraction introduced between the application and the system: the *runtime system*, the common example of which is an *Operating system*.

### 3.2.1 Operating system

Going back to the workshop again. Our joiner starts getting very strange orders from customers, like delivering wood or asking him to build a ship or a bridge or asking for products beyond the ability of his tools. He realized that they were coming to him from customers by mistake, since there were other businesses doing work they wanted. He decided to hire someone to take care of these requests and give him only the work he could really do. So, he arranged with the other business owners on this street to hire a manager to handle incoming requests. This manager would give the customer some predetermined form, determines the right business for the request and pass it on to the joiner or any other business.

Our manager is the *operating system*, also known as *OS*, a low-level system interface between the hardware component of the computer system and the developer. Those interfaces are called *system calls*. They interact with the computer hardware and provides services and utilities that user applications can use.



For example, when a program wants to write some data to a disk, it delegates that task to the operating system. The operating system gives instructions to the disk, using the disk controller that can send the right signals to the disk. The program that wants to use disk doesn't worry about what kind of disk the system has to or understand how it works. The OS handles the details and, if possible, tries to protect the hardware and other resources from improper use. This introduces overhead as the program uses OS functionality without directly communicating with the hardware. Sometimes this can be critical, and it is advantageous not to introduce a system call but to do something at the user application level. We will describe specific examples in the following chapters.

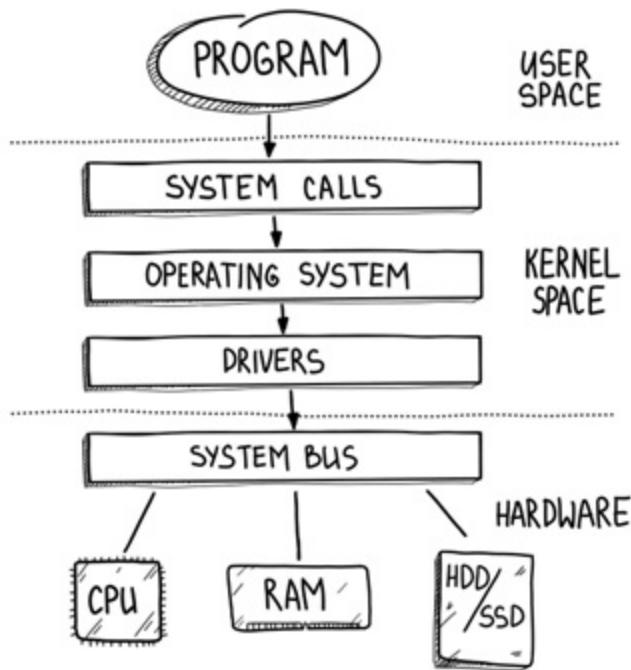
To start a program using OS, the first thing is loading the executable file and any static data (such as initialized variables) into memory. There is one last step left: starting the program from the entry point, namely `main()`. When the OS switches to `main()`, the control of the processor is transferred to the program and thus our program starts its execution under the control and protection of the OS.

All modern computer systems follow the steps described above. The process may be more sophisticated than the one described, but overall, they have the same design components.

### 3.3 Design of computer systems

If you look at the organization of a computer system, you will see one or

more processors, RAM that the processors can access, various peripheral devices (printers, card readers, hard drives, monitors, etc) and device controllers or *drivers* that allow all of those devices to communicate with the processor or RAM. To connect all those things there is a channel, the *system bus* that allows communications between CPU, RAM, and peripheral devices.



Let's now turn our attention to *user space* and *kernel space*, two distinct areas in a computer's system. User space is where user-level applications run, and kernel space is where the operating system's core functions and system calls run. The distinction is important because applications in user space cannot access or modify the underlying system, while the kernel has complete control over the system and its resources.

This internal design of a computer system remains largely the same, regardless of hardware platform specifics, such as the form factor, operating system structure, or intended use.

Having understood the general components of the design, let's move on to look at the several levels of parallel hardware that this design can represent.

## 3.4 Multiple levels of concurrent hardware

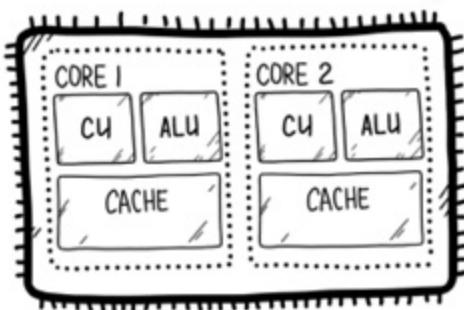
CPU are composed of a large number of circuits (ALUs) that can perform basic arithmetic operations (like addition or multiplication). Because of this, CPU can break up complex mathematical operations so that subparts of the operation run on separate arithmetic units at the same time, simultaneously. This is called *instruction-level parallelism*. Sometimes, this type of parallelism is taken to an even deeper level – *bit-level parallelism*.

Most developers rarely think about this level. The work of arranging instructions in the most convenient sequence for the processor is done by the compiler. Only a small group of engineers trying to squeeze all possible power from processor or compiler can be interested in this level.

Another simple idea for creating parallel hardware is that we can install more than one chip in a computer system replicating the processor, just like hiring the manager so all craftsmen can work together on the incoming customer requests. This is called *multi-processor*. That is what you can call any computer system with more than one processor.

A *multi-core processor* is a special kind of multi-processor with all processors on the same chip. Each core works independently, and OS perceives each core as a separate processor. There are slight differences between these two approaches, in terms of how quickly the processors can work together and how they access memory but for this book, we'll treat them as the same.

### A multi-core processor

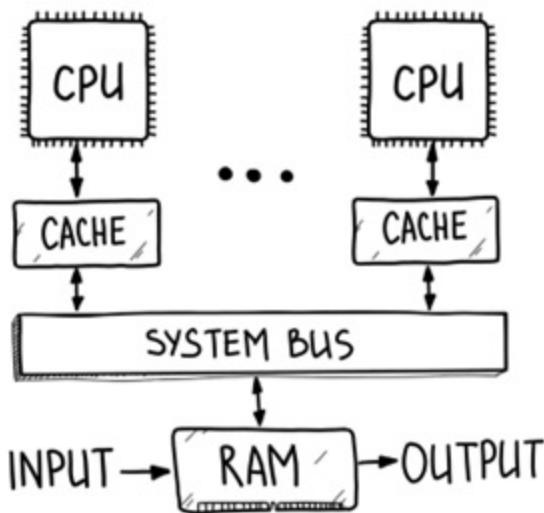


### 3.4.1 Symmetric Multiprocessing architecture

Computer memory usually operates at a much slower speed than processors do, resulting in the communication costs mentioned in Chapter 2. That's why most multi-processor systems today use *Symmetric Multiprocessing* or *SMP architecture*. SMP is a set of identical processors connected to a single, shared memory with a single address space and operate under the same operating system.

The processors in SMP architecture are linked by an interconnection network via the system bus. Although these networks are fast, if processors need to exchange data, the exchange will not be instantaneous because it must go through one or more interconnections. These communication costs are not negligible, and this is a problem that increasingly worsens the latency as the number of interacting resources and the distance between them increases. Thus, in the SMP architecture, all processors have their own private cache to reduce system bus traffic, resulting in lower latency.

**Symmetric Multiprocessing (SMP) architecture consist of multiple interconnected processors that have a shared memory**



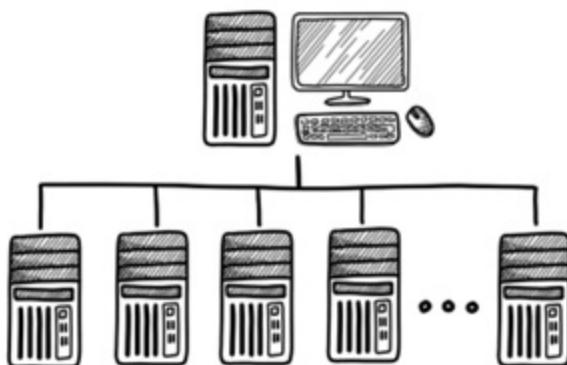
The coolest feature of an SMP is the existence of multiple processors is transparent to the end user. The OS takes care of scheduling processes on individual processors and of synchronization among those processors.

However, in such systems, increasing the number of processors connected to a common system bus makes it a bottleneck. This problem is worsened by *cache coherence* where multiple processor cores share the same memory hierarchy but have their own L1 data and instruction caches.

#### NOTE

The development of the MESI protocol in the 1980s solved the problem of cache coherence in multi-processor systems. By tracking the state of each cache line, MESI ensures that all processors have a consistent view of the data, allowing for efficient and conflict-free collaboration. Today, MESI is an essential part of modern computing.

The only way to move beyond SMPs to large, massive parallel computers is to abandon the shared memory architecture and move to a distributed memory systems called *computer clusters*. These clusters are distributed machines with their own CPUs, connected via a network. Computer clusters are very powerful parallel systems. One machine cannot share the memory with another one as each machine operates independently. If one machine makes changes to its local memory, that change is not automatically reflected in the memory of processors on other machines. Hence, clusters typically have distributed memory that brings more communication costs because now we need to communicate via network, which is much slower than transferring data between processes on the local machine.



Clusters are appropriate for “loosely coupled” problems (which do not

require frequent communication between processors but more power), while “tightly coupled” problems are more suitable for single machine systems. The advantage of clusters is high scalability. The disadvantage is the high communication costs. We will discuss distributed in detail in the later chapters but right now we focus on the types of multiprocessor architectures.

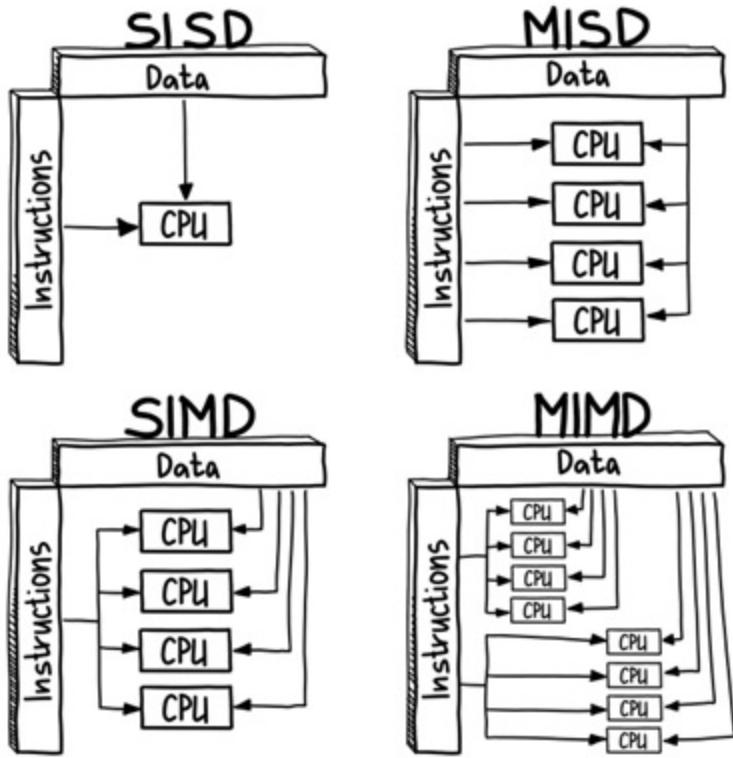
### 3.4.2 Taxonomy of parallel computers

One of the most widely used systems for classifying multiprocessor architectures is Flynn's Taxonomy. It distinguishes four classes of computer architectures based on two independent dimensions of *instructions* and *data flow*.

The first and second categories of computer architectures, Single Instruction Single Data (*SISD*) and Multiple Instruction Single Data (*MISD*), involve processing one block of data with one or multiple instructions respectively. However, as they lack parallelization, they are not relevant for concurrent systems and are mentioned here only for reference.

The third category is Single Instruction Multiple Data (*SIMD*), which features shared control units across multiple cores. With this design, only one instruction can be executed simultaneously on all available processing resources, allowing for the same operation to be performed on a large set of data elements at once. However, the instruction set in SIMD machines is limited, making them suitable for solving specific problems that require high computing power but not much versatility. A well-known example of SIMDs today are graphics processing units (GPUs).

The fourth category is Multiple Instruction Multiple Data or *MIMD*. Here, each processing resource has an independent control unit. So, it is not limited on types of instructions, and it executes different instructions independently on a separate block of data. Thus, it includes architectures with multiple cores, multiple CPUs, or even multiple machines, so different tasks can be literally executed on several different devices simultaneously.



As MIMD has a wider set of instructions, the individual processing resources are more versatile than in SIMD. That's why MIMD is the most commonly used architecture in Flynn's Taxonomy, and you'll find it in everything from multi-core PCs to distributed clusters.

### 3.4.3 CPU vs GPU

Even if you don't play video games, you can be grateful to the players, because they have spawned a class of very powerful parallel processing devices: Graphics Processing Units (aka GPU).

The CPU and GPU are very similar. They both have millions of transistors and can process a vast number of instructions per second. But how are these two important components different and when should you use one or the other?

Standard CPUs are built using the MIMD architecture. A modern CPU is powerful because engineers have implemented a wide variety of instructions

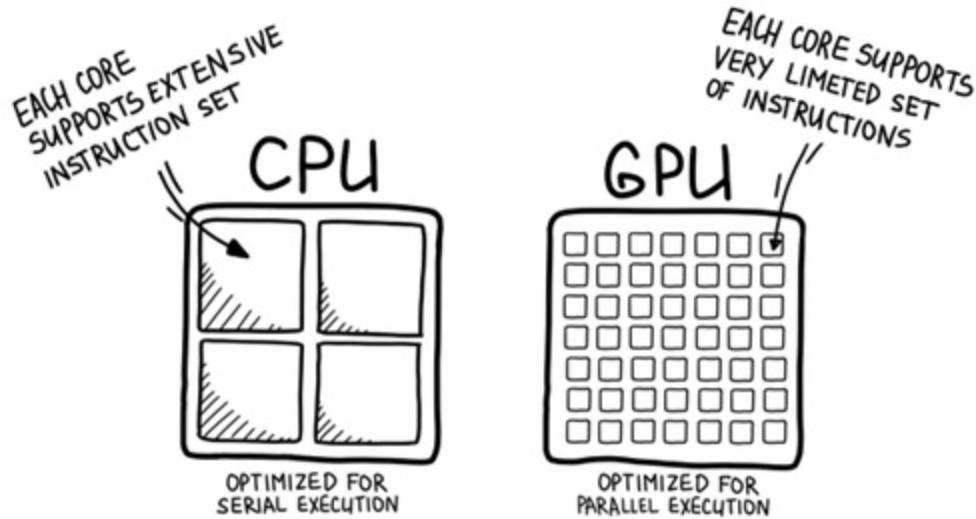
in them. And a computer system is capable of completing a task because its CPU is capable of completing that task.

The GPU is a specialized type of processor based on SIMD architecture, optimized for a very limited set of instructions. The GPU operates at a lower clock speed than CPU but has a more cores — hundreds or even thousands that run simultaneously. That means it performs a huge number of simple instructions at incredible speed due to massive parallelism.

#### NOTE

For example, Nvidia GTX 1080 graphics card has 2560 cores with 1607Mhz clock speed. Thanks to these cores, Nvidia GTX 1080 can perform 2560 instructions per clock cycle. If you want to make the picture by 1% brighter, the GPU will cope with this without any difficulty. But Intel Core i9-10940X CPU with 3.3GHz can only execute 14 instructions per clock cycle<sup>[1]</sup>.

Although individual CPU cores are faster, based on clock speed, and have extensive instruction sets, the sheer number of GPU cores and the massive parallelism they provide more than compensate for the difference in CPU core clock speed and limited instruction set. CPUs are just better suited for complex linear tasks.



GPUs are best suited for repetitive and highly parallel computational tasks

such as video and image processing, machine learning, financial simulation, and many other types of scientific computing. Operations such as matrix addition and multiplication are easily performed using the GPU because most of these operations in matrix cells are independent of each other, are similar in nature, and therefore can be parallelized.

Hardware architectures are highly variable and can affect the portability of programs between different systems as well as programs themselves can sometimes inherently accelerate differently depending on where they run. For example, many graphics programs run much better and faster on GPU resources, while ordinary programs with mixed logic make sense to run on the CPU.

In this book, we will use the term “CPU” in a general sense that covers both types of processing resources. With all the components of physical execution in mind, in the next chapter we will add a couple of easy-to-use abstractions that will represent instruction streams.

## 3.5 Recap

- Execution depends on the actual hardware. Modern hardware has multiple processing resources – multiple *cores*, *multi-processors* or *computer clusters* and they are optimized for executing programs
- Flynn's classification describes four types of architecture, based on whether the system processes single or multiple instructions at a time (SI or MI) and on whether each instruction acts on a single or multiple blocks of data (SD or MD)
- *GPU* is example of *SIMD* architecture. It's optimized for highly parallel tasks execution.
- Modern multi-processor and multi-core processors are examples of *MIMD*. They are far more complex because they're multipurpose
- Processor or CPU is the brain of the computer system but it's difficult to work with it directly – in programming, an additional level of abstraction is introduced between the application and the system: the *runtime system*
- In order to exploit parallel execution in application developer need to have a processing unit that is suitable for the problem – CPU has higher

clock frequency and wider set of instruction that can be executed in parallel, while GPU operates at a lower clock speed, and have only one instruction executed across all of the cores, but it is doing that at incredible speed due to massive parallelism

- [1] Intel® Core™ i9-10940X X-series processor specifications,  
<https://www.intel.com/content/www/us/en/products/sku/198014/intel-core-i910940x-xseries-processor-19-25m-cache-3-30-ghz/specifications.html>.

# 4 Building blocks of concurrency

## In this chapter

- You learn more about the middle layer of concurrency – the runtime system, a popular example of which is the operating system
- You learn the internals of the two basic concurrency abstractions: threads and processes
- You learn how to implement concurrent applications using threads and processes
- You learn how to choose the concurrency abstraction suitable for your problem

Concurrent programming involves breaking down applications into independent units of concurrency. In previous chapters, we referred to these units as "tasks" for organizing the flow of the application. Now, with knowledge of the hardware being used, we need to map these abstractions onto the physical devices executing the code. Thankfully, there is another layer of abstraction that can handle this task: the operating system. Its role is to apply the available hardware as efficiently as possible, but it is not a magical solution. This chapter will focus on how developers can structure their programs to aid the operating system in achieving optimal hardware utilization.

## 4.1 Concurrent programming steps

Concurrent programming is a set of abstractions that allow the developer to structure the program to generate small independent tasks and pass them to the runtime system, queuing them for execution. The runtime system orchestrates tasks to optimally utilize system resources and passes them to the execution on appropriate processing resources. The two main abstractions used in concurrent programming to accomplish this are *processes* and *threads*.

## 4.2 Processes

The informal definition of a *process* is relatively straightforward: it is a running program. The program itself is a lifeless thing. It just sits on a disk representing a set of instructions waiting to be executed. It is the operating system that takes these instructions and executes them on hardware, turning the program into something useful.

Imagine a car. A car is just a set of mechanical parts that together make up a car. Even though the car has great potential – if it just stands still, it has no value. But when someone turns the key and the engine starts running, the car starts up and moves. It evolves into the process of driving. It becomes not just a car but a trip from point A to point B, bringing value. The car facilitates the desired action.



Source code is like the car. It is just a passive sequence of instructions that operate with resource abstractions. When writing the source code, developers do not have memory to store temporary data, do not have files to read or write, and no devices they want to send signals to. Developers write code using real-world models built upon abstractions provided by programming languages and runtime environments. Actual resources must be provided at the time of the actual execution.

The abstraction provided by the OS for a running program is what we will call a process; there is no concept of a process at the level of machine instructions.

The purpose of using processes in an operating system is to isolate tasks and

allocate hardware resources for their execution. All processes in an operating system share hardware resource and are managed by the operating system. To ensure that the operating system knows the relationship between processes and resources, each process must have its own independent address space and file table. Therefore, processes are the unit for resource allocation in the operating system.

Operating systems provide the illusion of full ownership of the computer system to each process, even though there are usually multiple processes running concurrently. To maintain this illusion, operating systems take great measures to control, protect, and isolate processes from each other. This includes controlling the allocation of CPU cores and memory for each process. The main advantage of processes is the complete independence and isolation of their execution from the rest of the system, preventing accidental interference with global objects and ensuring that crashes of one program do not affect others.

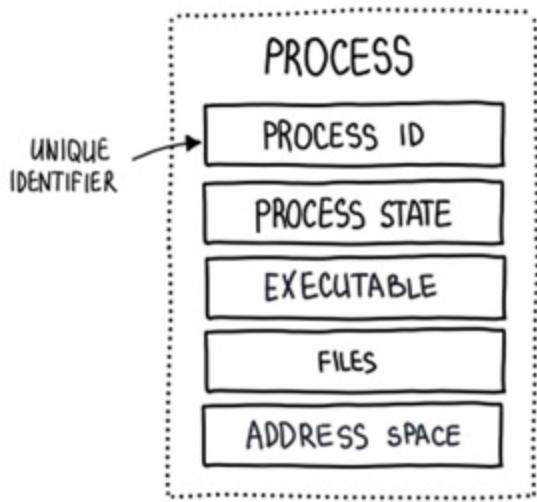
From this advantage, however, also comes a disadvantage. Processes are independent of each other by design, which makes communication between them difficult. Formally, there is almost nothing in common between processes, and any non-trivial communication between processes requires the use of other mechanisms, which are usually several orders of magnitude slower than the direct access to data. We will talk about that in details in the next chapter, but for now let's look inside the process.

### **4.2.1 Process internals**

As has been said before, a process is just a running program; at any given time, we can assemble a process by making a list of the various parts of the computer system that it accesses or modifies at runtime:

- The data the process reads or writes to is stored in memory. Thus, the memory which the process can see or access (the address space) is part of the running process.
- The executable file with all the machine instructions is part of the process.
- The process also needs an identifier: a unique name by which the

- process can be identified; it is called a Process ID or *PID*.
- Finally, programs often access disks or network resources, or other third-party devices. Such information must include a list of files currently open by the process, open network connections, and any additional information about the resources it uses.

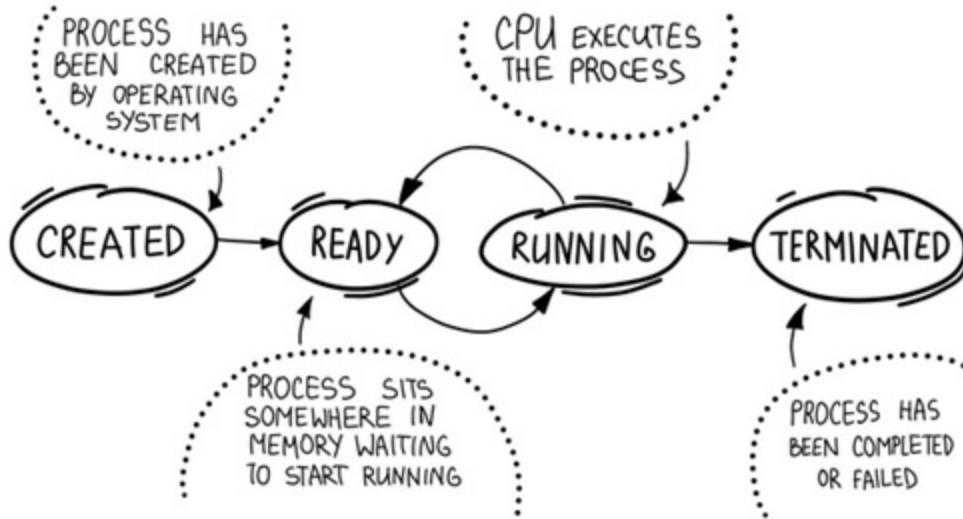


Thus, a process encapsulates many things: executable file, the set of resources used (files, connections, etc.), and the address space with internal variables. All of this is called *execution context*. Because so many things exist inside processes, starting a new process is a pretty heavy thing to do that's why they are often called "heavyweight" processes.

### 4.2.2 Process states

If you look at the process from a high level everything seems quite trivial.

At first, it seems the process doesn't exist. Then it is created and initialized, after which it exists somewhere in computer memory (*Created* state). Then when the user code starts a process, it goes to the *Ready* state – it is ready to be executed on some processor core at any moment, but it doesn't do anything yet. It needs a processing resource to start the execution. Then the OS selects the next process to be executed on the CPU from the list of processes ready for execution. After the OS chooses the process, the chosen process goes into the *Running* state.

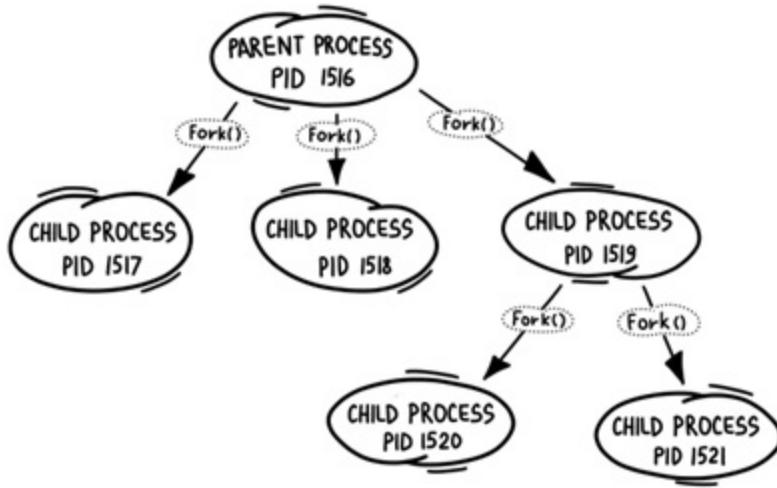


Processes are usually created by the operating system. Apart from creating processes the operating system is also responsible for the process termination. This is not the most trivial task. The operating system needs to understand that the process is finished – either the task is complete, the process failed and it's time to clean it up or the parent process is dead. Creating or terminating a process is relatively expensive, because as we have seen a process has many resources attached to it and they must be created or freed up. It takes system time and introduces additional latency.

### 4.2.3 Multiple processes

Processes can create their own processes – called *child processes* through appropriate system calls, such as `fork()` or `spawn()`; this process is called *spawning*. Child processes are independent forks of the main process with a separate memory address space, which again means that the process works independently and is isolated from others by operating systems control. It cannot directly access the data of other processes, and instructions belonging to each process will be executed in the corresponding process independently of each other and, ideally, in parallel.

Now we are moving into the territory of concurrency. Using spawning programs, execution can be decomposed into multiple processes that can be executed simultaneously on parallel hardware.



However, this is probably easier to understand in code than in theory. Below is an example of a program that makes three child processes using a forking mechanism.

```

# Chapter 4/child_processes.py
import os
from multiprocessing import Process

def run_child() -> None:
    print("Child: I am the child process")
    print(f"Child: Child's PID: {os.getpid()}")
    print(f"Child: Parent's PID: {os.getppid()}")

def start_parent(num_children: int) -> None:
    print("Parent : I am the parent process")
    print(f"Parent : Parent's PID: {os.getpid()}")
    for i in range(num_children):
        print(f"Starting Process {i}")
        Process(target=run_child).start() #A

if __name__ == "__main__":
    num_children = 3
    start_parent(num_children)

```

The code will create a parent process with three child processes that will be a copy of the parent process; the only difference would be a process ID. The execution of the parent and child process is independent of each other.

## **NOTE**

It's important to note that when forking a process, the new process starts its execution from the point where the forking occurred, and that its internal state is copied. It is not executing the script again from the beginning.

The program will output messages from parent and child processes with their respected PID, similar to the following:

```
Parent : I am the parent process
Parent : Parent's PID: 73553
Parent : Child's PID: 73554
Child: I am the child process
Child: Child's PID: 73554
Child: Parent's PID: 73553
Parent : I am the parent process
Parent : Parent's PID: 73553
Parent : Child's PID: 73555
Child: I am the child process
Child: Child's PID: 73555
Child: Parent's PID: 73553
Parent : I am the parent process
Parent : Parent's PID: 73553
Parent : Child's PID: 73556
Child: I am the child process
Child: Child's PID: 73556
Child: Parent's PID: 73553
```

Programming languages commonly have some high-level abstractions or service methods for working with processes as they are easier to maintain and follow in the program source code.

## **NOTE**

This forking/spawning approach has been implemented in “prefork” mode in several popular server technologies. Pre-forking means a server creates forks at server startup which then handle incoming requests. Nginx, Apache HTTP server, and Gunicorn work in this mode, allowing them to handle hundreds of requests. But these solutions also support other methods.

## **4.3 Threads**

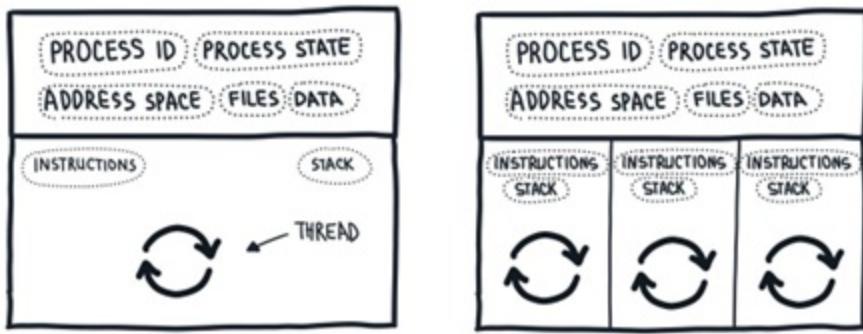
Sharing memory between processes is possible on most operating systems, but it needs additional efforts for that (we will talk about it in Chapter 5). There is another abstraction that allows us to share a bit more – *threads*.

In the end, a program is simply a set of machine instructions that must be executed one after the other in sequence. To make this happen, the operating system uses the concept of a thread. Technically, a thread is defined as an independent stream of instructions whose execution can be scheduled by the operating system.

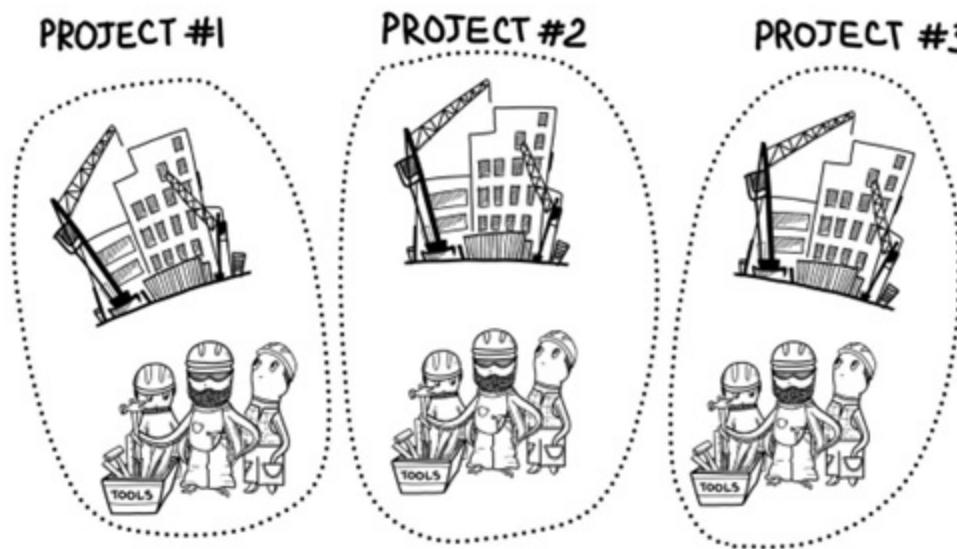
Remember we said that a process is a running program plus resources? If we try to split the program into separate components, then a process would be a container of resources (address space, files, connections, etc.) and a thread would be a dynamic part – sequence of instructions which are executed inside this container. Therefore, in the context of operating system, a process can be seen as a unit of resources, while a thread can be viewed as a unit of execution.

But threads were born from the idea that the most efficient way to share data between interacting processes is to share a common address space. Thus, threads in a single process are like processes that can easily share resources with each other and with their parent process, such as address space, files, connections, any shared data, etc.

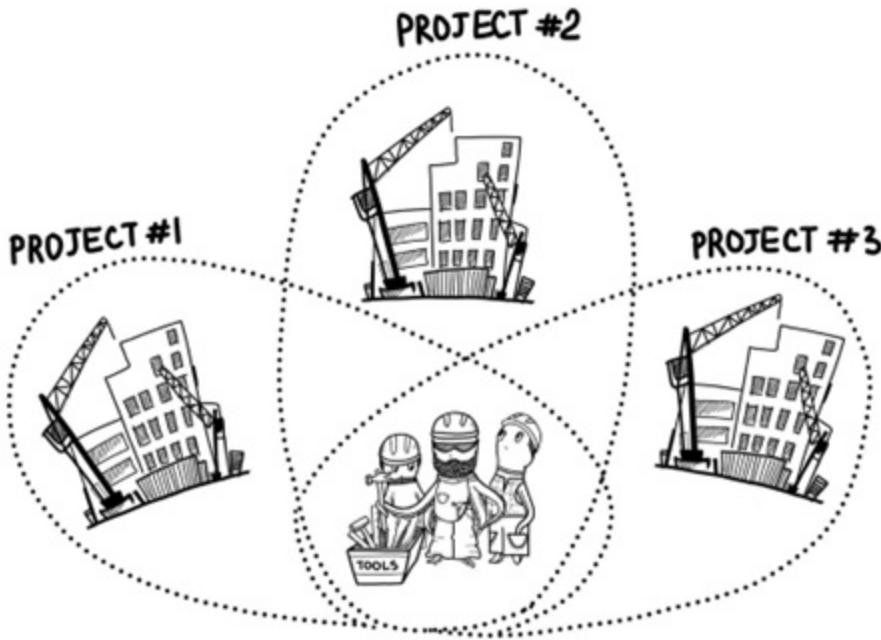
Threads also maintain their own state to allow for the safe, local, and independent execution of their instructions. Each thread is unaware of the other threads unless it is interfering with them on purpose. The operating system itself manages the threads and is able to distribute them among the available processor cores. Thus, creating a multi-threaded program can be a convenient way to run multiple tasks concurrently.



To illustrate the difference between processes and threads, let me show you an example. Imagine that you are managing the work of a construction company and you hire three construction crews to work on three different projects.



It would be a process-like job. Each construction crew (process) is dedicated to one project (task) with its own tools, project plan, resources, etc. On the other hand, to save budget, you could hire just one construction crew for all three different projects, where they could use common tools and resources, but there would be a separate list of instructions for each of the projects, similar to how threads work.



Historically, hardware vendors have implemented their own versions of threads. These implementations differed significantly from each other, making it difficult for developers to implement portable threaded applications. A standardized programming interface was needed.

For UNIX systems, this interface was defined by IEEE POSIX<sup>[1]</sup> and available as an optional library for Windows-family operating systems. Implementations that adhere to this standard are called *POSIX threads*, or *Pthreads* (also the name of the C library implementation). Most hardware manufacturers use Pthreads and so we'll talk more about this standard.

In that standard, every program you run causes the operating system to create a process, and every process has at least one thread; a process without a thread cannot exist. Each thread also maintains its own independent execution context to ensure that its instructions are executed safely and independently.

### 4.3.1 Threads features

Properly implemented, threads have advantages and disadvantages compared to processes.

## **Advantage: Less memory overhead**

The processes are completely independent, each with their own address space, with their own set of threads, with their own copies of variables which are completely independent from the same variables in the other process. Threads have much less memory overhead than the standard `fork()` function as the parent thread is not copied – threads still use the same process. Because of this, threads are also sometimes called “lightweight” processes.

Consequently, you can create more threads than processes on the same system. Creation and termination of threads is faster than processes because it takes less time for the operating system to allocate and manage thread resources. Because of this you can create threads whenever it makes sense in your application and not worry about wasting CPU time and memory.

## **Advantage: Less communication overhead**

Each process works with its own memory – they can only exchange something through process communication mechanism which we will discuss in the next chapter.

Threads use the same address space, and therefore can communicate with each other by writing and reading to the same shared address space of their parent process without any problems or overhead: anything changed by one thread is immediately available to all. Hence for widely used SMP systems it is sometimes much more convenient to use threads than processes.

## **Disadvantage: Need for synchronization**

The operating system provides complete independence of processes from each other, so if one of them crashes, other processes are not harmed. This is not the case with threads: since all threads in a process use the same shared resources, if one of them crashes or is corrupted, the others will likely be affected as well. To prevent this from happening, developers need to synchronize access to shared resources and have more control over the behavior of threads (Chapter 8).

### 4.3.2 Threads implementation

A thread-based approach is a common way to achieve concurrency in many languages. This does not mean that threads are explicitly used in programming languages. Instead, the runtime environment can map other programming language concurrency constructs to physical threads at runtime.

Programming languages usually have some higher-level abstractions for creating processes, because they are easier to maintain and keep track of in the source code of the program.

#### NOTE

Avoid using low-level threads if you can get away with it. Look at libraries that abstract away the need to use low-level threads. A general implementation of POSIX is presented in C/C++ as a library of functions. Modern languages such as Python, Java and C# (.NET) provide a set of abstractions on top of native threads that is most closely matches the design characteristic of these languages. Or similarly, the property of multiple threads can be idiomatically hidden in a language such as Go's goroutines, Scala parallel collections, Haskell's GHC, Erlang processes, OpenMP, MPI and others. These implementations are portable in any operating system that provides the runtime implementation required by these languages.

Here is an example in Python where we are creating five child threads:

```
# Chapter 4/multithreading.py
import os
import time
import threading
from threading import Thread

def cpu_waster(i: int) -> None:
    name = threading.current_thread().getName()
    print(f"{name} doing {i} work")
    time.sleep(3)

def display_threads() -> None:
    print("-" * 10)
```

```

print(f"Current process PID: {os.getpid()}")
print(f"Thread Count: {threading.active_count()}")
print("Active threads:")
for thread in threading.enumerate():
    print(thread)

def main(num_threads: int) -> None:
    display_threads() #A

    print(f"Starting {num_threads} CPU wasters...")
    for i in range(num_threads):
        thread = Thread(target=cpu_waster, args=(i,)) #B
        thread.start() #B

    display_threads() #A

if __name__ == "__main__":
    num_threads = 5
    main(num_threads)

```

Output:

```

-----
Current process PID: 35930
Thread Count: 1
Active threads:
<_MainThread(MainThread, started 8607733248)>
Starting 5 CPU wasters...
Thread-1 doing 0 work
Thread-2 doing 1 work
Thread-3 doing 2 work
Thread-4 doing 3 work
Thread-5 doing 4 work
-----
Current process PID: 35930
Thread Count: 6
Active threads:
<_MainThread(MainThread, started 8607733248)>
<Thread(Thread-1, started 12940410880)>
<Thread(Thread-2, started 12945666048)>
<Thread(Thread-3, started 12950921216)>
<Thread(Thread-4, started 12956176384)>
<Thread(Thread-5, started 12961431552)>
```

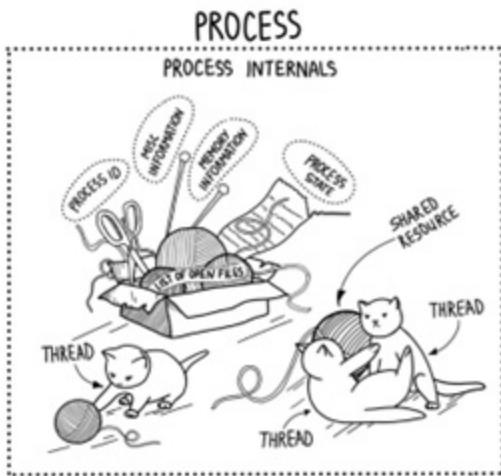
When we start our program, we create a process in which a main execution

thread is created. Note that any thread, including the main thread, can create child threads at any time (that's why we have "Thread Count: 6" output above). In our example we create five new threads and run them concurrently.

Processes and threads are the building blocks of concurrency, and we will talk a lot about them, but whether you work with threads or processes, you can think of them all as just threads because every process has at least one thread. Later in the book, we will try to use the term "task" as a generic entity if the specific implementation is not important.

Before we move on, you've probably figured out by now that implementing concurrency is not easy work. Over the course of the last four chapters, we've outlined just how difficult it can be. You might be wondering if it's the right thing for you after reading this far. But let's take a moment to encourage you.

Think of kittens in a basket of yarn. They are inquisitive, experimental and love a good time. Kittens don't loom at the knitting basket with dismay? They see it as a playground to explore, disassemble and make it their own.



A good programmer is much the same way. You've got processes, shared resources, threads, open files and data to work with, all to create a program that can solve real-world problems, automate tasks, or entertain millions of users.

So be encouraged and press on. Grab that thread and unravel it. What you do

from here on could change the world.

## 4.4 Recap

- The job of the operating system is to map execution onto the actual hardware
- A *process* is an instance of a program running within a computer system. Each process has one or more *threads* of execution, and no thread can exist outside a process.
- A thread is a unit of computation, an independent set of programming instructions designed to achieve a particular result, which the OS independently executes and manages.
- Multiple execution threads can exist within the same process and share resources, while processes are almost independent.
- Using threads makes it easy to create concurrent applications because switching between threads is easier than switching between processes. Moreover, threads use a common address space which results in faster access to shared data. But there is also a risk of data corruption, which requires some caution to control access and synchronization to shared objects.

[1] IEEE POSIX 1003.1c (1995), <https://standards.ieee.org/ieee/1003.1c/1393/>

# 5 Inter-process communication

## In this chapter

- You learn how to achieve effective task communication
- You learn how to choose communication type for your applications
- You learn a popular programming pattern for creating concurrent applications – Thread pool

We can't always guarantee that concurrent tasks running on a computer are independent. Often, communication between tasks is necessary for efficient execution. For example, if one task depends on the result of another, the application will have to know when to pause its work while it waits for the other tasks to finish.

Communication is therefore at the heart of any concurrent system. If we cannot ensure proper communication between tasks, the performance gains from concurrency are meaningless. In this chapter you will learn the concepts provided by the operating system to allow processes and threads to communicate with each other and to coordinate their work. We'll start off by looking at the different types of communication you're likely to encounter in a concurrent system.

## 5.1 Types of communication

The operating system provides mechanisms allowing processes and threads to communicate with each other. These mechanisms are called *inter-process communication*, or IPCs. Once you decide that your application will benefit from IPC, you must decide which of the available IPC methods on your system to use.

### NOTE

IPC is called inter-PROCESS communication, but that does not mean that

only the processes need to communicate. Whether you work with threads or processes you can think of them all as threads, because every process has at least one thread, so de facto communication only occurs between threads. Ignore that confusion in terminology – we will use term *task* as a general abstraction for the unit of execution.

The most popular types of IPCs are via *shared memory* or *message passing*.

### 5.1.1 Shared memory IPC

The simplest way to communicate between tasks is to use a shared memory. Shared memory allows one or more tasks to communicate through common memory that appears in all of their virtual address spaces, as if they were reading and writing to local variables that is part of their address space. So, changes made by one process or thread are instantly reflected in the others without interacting with operating system.

Imagine that you live in the same house with several friends. There's a shared kitchen with a single refrigerator for everyone's use. You can get a beer for yourself and inform your friends that they can find a six-pack on the lowest shelf. The refrigerator serves as a shared memory, used by all friends (tasks) to store beer (shared data).



Shared memory IPC can be found if two processors (or processor cores) in the same computer refer to the same physical memory location, or when threads within the same program share the same objects.

In the code it will look like this:

```
# Chapter 5/shared_ipc.py
import time
from threading import Thread, current_thread

SIZE = 5
shared_memory = [-1] * SIZE #A

class Producer(Thread):
    def run(self) -> None:
        self.name = "Producer"
        global shared_memory
        for i in range(SIZE):
            print(f"{current_thread().name}: Writing {int(i)}")
            shared_memory[i - 1] = i #B

class Consumer(Thread):
    def run(self) -> None:
        self.name = "Consumer"
        global shared_memory
        for i in range(SIZE):
            while True: #C
                line = shared_memory[i] #C
                if line == -1: #C
                    print(f"{current_thread().name}: Data not available")
                    f"Sleeping for 1 second before retrying"
                    time.sleep(1) #C
                    continue #C
                print(f"{current_thread().name}: Read: {int(line)}")
                break #C

def main() -> None:
    threads = [
        Consumer(),
        Producer(),
    ]
    for thread in threads: #D
        thread.start() #D
    for thread in threads: #E
        thread.join() #E

if __name__ == "__main__":
    main()
```

Here we have created two threads – Producer and Consumer. Producer produces data and stores it in the shared memory; Consumer consumes the data that has been stored in shared memory. Hence, they communicate with each other using a shared array.

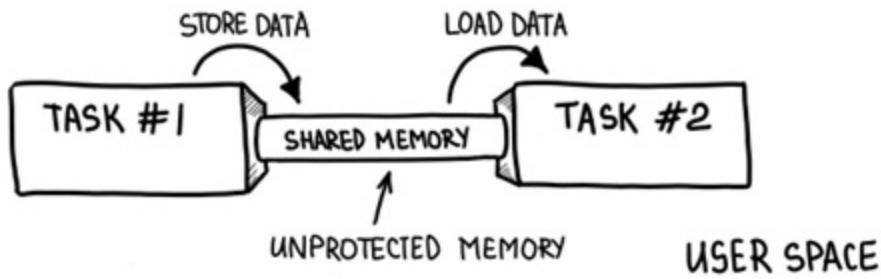
Output of that program:

```
Consumer: Data not available
Sleeping for 1 second before retrying
Producer: Writing 0
Producer: Writing 1
Producer: Writing 2
Producer: Writing 3
Producer: Writing 4
Consumer: Read: 1
Consumer: Read: 2
Consumer: Read: 3
Consumer: Read: 4
Consumer: Read: 0
```

This sharing of memory is both a blessing and a curse for the developer, as we'll see in a moment.

## **Advantages**

The blessing of the approach is the fact that it provides the fastest and the least resource intensive communication possible. Although the operating system helps with the allocation of the shared memory, it does not participate at all in the communication between the tasks. Thus, the operating system in this case is completely removed from the communication and all the overhead of working with it, resulting in higher speed and less data copying.



## Disadvantages

The “curse” of this approach is it is not necessarily the safest communication between the tasks. The operating system no longer provides the interfaces and protection of the shared memory. For instance, two friends may want to drink the last bottle of beer. That's a conflict (even war, sometimes). Similarly, tasks running the same program may want to read or update the same data structures. For that reason, its use sometimes becomes more error-prone, and developers have to redesign the code by protecting shared memory objects (we will talk more about that in Chapter 8).

Another disadvantage of this approach is that it does not scale beyond one machine. Shared memory can be used just by local tasks. This creates problems in large distributed systems where data that need to be processed can't fit into one machine, but it's a perfect fit for the SMP architecture systems.

In a SMP system, all processes or threads on the various CPUs share a unique logical address space, mapped to a physical memory. And that's why shared memory approach has been very popular for SMP systems especially using threads as they build around the shared memory idea in mind from the beginning. However, in such systems, increasing the number of processors connected to a common system bus makes it a bottleneck (Chapter 3).

### 5.1.2 Message passing IPC

Probably the most widely used type of inter-process communication mechanism today that is often supported by operating systems is *message*

*passing.*

In message passing IPC each task is identified by a unique name, and tasks interact with each other by sending and receiving messages to and from named tasks. The operating system establishes a communication channel and provides proper system calls for tasks to pass messages through this channel.

The advantage of this approach is that the operating system will manage this channel, and it provides easy-to-use interfaces to send and receive data without conflicts. On the other hand, there is a huge communication cost. To transfer any piece of information between tasks, it must be copied from the task's user space to the operating system channel through system calls (as discussed in Chapter 3), and then copied back to the address space of the receiving task.

Message passing has another advantage: it can be easily scaled beyond just one machine to distributed systems, but there more to it so let's skip it for now.

#### **NOTE**

There are a lot of programming languages that choose to use only message passing IPC. The Go language philosophy is to share memory through communication. Here's the idea in a slogan from the Go language documentation: "Do not communicate by sharing memory; instead, share memory by communicating." Another example is Erlang, where processes don't share any data, and they communicate with each other exclusively by message passing.

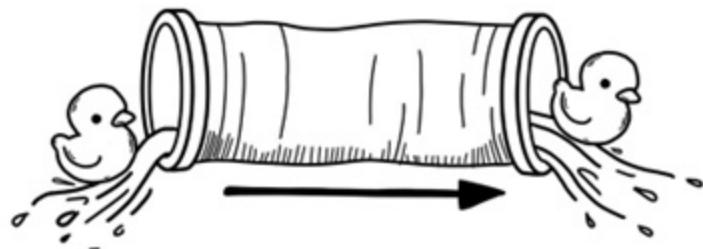
There are a lot of technologies to implement the message passing approach; we will cover some of the most common ones in modern operating systems – pipes, sockets and message queues – in the sections below.

## **Pipes**

This is probably the simplest form of IPC. A pipe is a simple synchronized way of transferring information from one task to another. As the name

implies, a pipe defines a one-way flow of data between tasks – data is written to one end and read from the other. A pipe allows for data flow in one direction; when bidirectional communication is needed, two pipes need to be created.

You can imagine a pipe in IPC as being an actual water pipe. If you put something like a rubber duck into a stream, it will travel downstream to the end of the waterway. The writer end is the upstream location where you put rubber ducks into the pipe, and the reader end is where the rubber duck ends up downstream.



One part of your code calls methods on the writer end with the data you want to send, and another part reads incoming data. A pipe is a temporary object that can be used just by two tasks and will be closed if either the sender or receiver half is dropped.

#### NOTE

Channel is a popular data type in Go which provides synchronization and communication between Go concurrency primitive, or goroutines. Channels can be thought of as pipes that are used by goroutines to communicate.

Pipes come in two kinds: *unnamed* and *named*. *Unnamed pipes* can only be used by related tasks (i.e. child-parent or sibling processes or threads in the same process) because related tasks share file descriptors. Unnamed pipes disappear after the tasks finish using them.

Since a pipe is essentially a file descriptor (in UNIX systems), pipe operations are actually very similar to file operations but have no connection

whatsoever to the filesystem. When the writers want to write data to a pipe, they use `write()` OS system call on the pipe. To retrieve data from a pipe, the `read()` system call is used. `read()` handles pipes in the same way it handles files, but it will be blocked until there is no data to be read. The pipes may be implemented differently in different systems.

By creating a pipe in the main thread, and then passing file descriptors to the child threads, we can pass data from one thread to another through the pipe. This is exactly how a standard pipe works. Let's look at the code:

```
# Chapter 5/pipe.py
from threading import Thread, current_thread
from multiprocessing import Pipe
from multiprocessing.connection import Connection

class Writer(Thread):
    def __init__(self, conn: Connection):
        super().__init__()
        self.conn = conn
        self.name = "Writer"

    def run(self) -> None:
        print(f"{current_thread().name}: Sending rubber duck...")
        self.conn.send("Rubber duck") #A

class Reader(Thread):
    def __init__(self, conn: Connection):
        super().__init__()
        self.conn = conn
        self.name = "Reader"

    def run(self) -> None:
        print(f"{current_thread().name}: Reading...")
        msg = self.conn.recv() #B
        print(f"{current_thread().name}: Received: {msg}")

def main() -> None:
    reader_conn, writer_conn = Pipe() #C
    reader = Reader(reader_conn)
    writer = Writer(writer_conn)

    threads = [
```

```

        writer,
        reader
    ]
    for thread in threads:
        thread.start()

    for thread in threads:
        thread.join()

if __name__ == "__main__":
    main()

```

Here we create an unnamed pipe with two threads. The writing thread writes a message to the reader through the pipe.

Output of the program:

```

Writer: Sending rubber duck...
Reader: Reading...
Reader: Received: Rubber duck

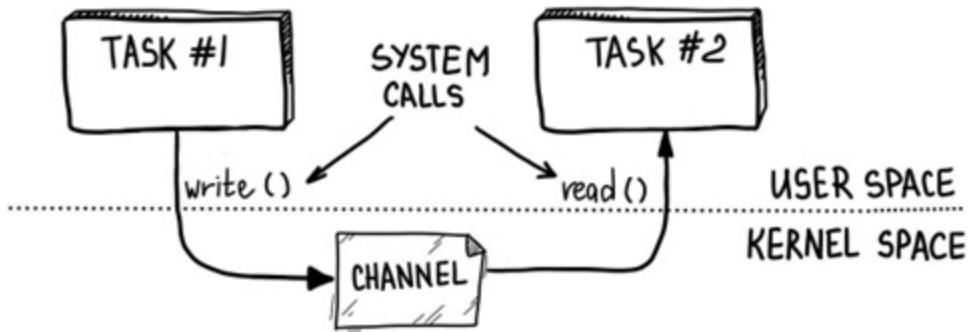
```

#### **NOTE**

`pipe()` and `fork()` make up the famous functionality behind the pipe operator ("|") in `ls | more` in popular Unix shell and command language Bash.

*Named pipes* allow the transfer of data between tasks according to the FIFO (First In First Out) principle, which means the request is processed in the order it arrives. Because of that they are also often referred as *FIFOs*.

Unlike unnamed pipes, FIFOs are not temporary objects; they are entities in the file system and can be freely used by unrelated tasks if they have appropriate permissions to access them. Using named pipes allows tasks to interact even if they don't know which tasks are on the other end of the pipe, even over network. Otherwise, FIFOs are treated exactly like unnamed pipes and use the same system calls.



Because of this unidirectional nature of pipes, probably the best use of pipes is to transfer data from producer programs to consumer programs. For other uses it is rather limited, and there are often other IPC methods that work better.

## Message Queues

Another popular message passing IPC implementation is *message queue*. Similar to named pipes, message queues keep data organized, by using the same FIFO principle, which is why they have “queue” in the name, but they also support multiple tasks to write or read messages.

Message queues provide a powerful means of decoupling tasks in a system, allowing producers and consumers to interact with the queue instead of directly with each other. That gives a lot of freedom for developers to control the execution. For example, workers can put messages back to the message queue if they have not been processed for some reason.

Here how they look like in code:

```
# Chapter 5/message_queue.py
import time
from queue import Queue
from threading import Thread, current_thread

class Worker(Thread):
    def __init__(self, queue: Queue, id: int):
        super().__init__(name=str(id))
        self.queue = queue
```

```

def run(self) -> None:
    while not self.queue.empty():
        item = self.queue.get() #A
        print(f"Thread {current_thread().name}: "
              f"processing item {item} from the queue")
        time.sleep(2)

def main(thread_num: int) -> None:
    q = Queue() #B
    for i in range(10): #B
        q.put(i) #B

    threads = []
    for i in range(thread_num):
        thread = Worker(q, i + 1)
        thread.start()
        threads.append(thread)

    for thread in threads:
        thread.join()

if __name__ == "__main__":
    thread_num = 4
    main(thread_num)

```

Here we have created a message queue and placed ten messages in it for our four children to process. Our threads will process all the messages in the queue until it is empty. Note that the queue is not just a single thread interaction point but also holds the messages until they are processed – creating a loosely coupled system.

The output of the program we just ran looks like this:

```

Thread 1 : processing item 0 from the queue
Thread 2 : processing item 1 from the queue
Thread 3 : processing item 2 from the queue
Thread 4 : processing item 3 from the queue
Thread 1 : processing item 4 from the queue
Thread 2 : processing item 5 from the queue
Thread 3 : processing item 6 from the queue
Thread 4 : processing item 7 from the queue
Thread 1 : processing item 8 from the queue
Thread 3 : processing item 9 from the queue

```

As we've seen, message queues are really used to implement loosely coupled

systems. They are used everywhere: in operating systems to schedule processes and in routers as buffers to store packets before they are processed. Even cloud applications consisting of microservices use message queues to communicate. Also, message queues are widely used for asynchronous processing. We will get to the practical use of queues at the end of this chapter, but for now let's stop here and move on to a discussion of UDS sockets.

## **UDS sockets**

*Sockets* can be used to communicate in a wide variety of domains and in this chapter, we will be talking about Unix domain sockets (UDS sockets) used between threads on the same system. We will talk about network and network sockets, other common domains sockets, in Chapter 10.

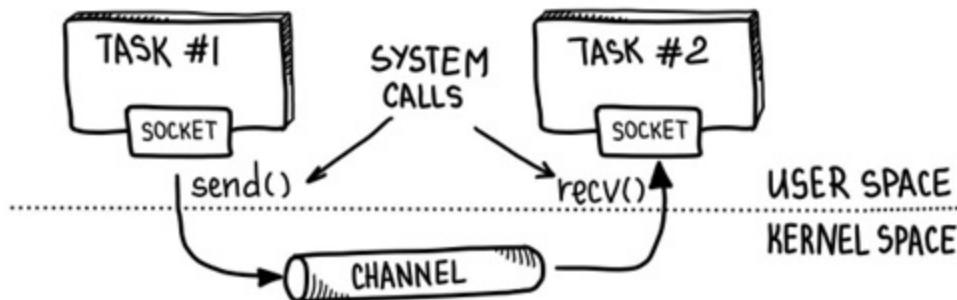
We can create a two-way, FIFO communications via sockets implementing message-passing inter-process communication. In this IPC, one thread can write information to the socket and a second thread can read information from the socket. A socket is an object that represents the end point of that connection. Threads from both ends have their own socket, which is connected to another socket. So, to send information from one thread to another, you write it to the output stream of one socket and read it from the input stream of the other socket.

Speaking of sending messages between two entities, let's take a moment to imagine sending a Christmas card to your mom. You need to write some sweet holiday wishes on the card, put the address and your mum's name on it. Then you need to drop it in your local mailbox. You've done your part. Now the postal service will do the rest. They will send it to your mom's local post office and the mailman will deliver your card to your mom's door, seeing the happy look on her face.



In the case of a Christmas card, we first put the sender and recipient address on the card. With sockets you also have to establish a connection first. And then the message exchange starts.

The Sender thread puts the information it wants to send in the message and then sends it explicitly over that dedicated channel to the Receiver thread, and the receiver thread then reads it. We need at least two primitives: `send(message, destination)` and `receive(message)`. The threads in this exchange can be executed either on the same machine or on different machines connected to each other by a network.



In the code:

```
# Chapter 5/sockets.py
import socket
import os.path
import time
from threading import Thread, current_thread

SOCK_FILE = "./mailbox" #A
BUFFER_SIZE = 1024 #B
```

```

class Sender(Thread):
    def run(self) -> None:
        self.name = "Sender"
        client = socket.socket(socket.AF_UNIX, socket.SOCK_STREAM)
        client.connect(SOCK_FILE) #D

        messages = ["Hello", " ", "world!"]
        for msg in messages: #E
            print(f"{current_thread().name}: Send: '{msg}'") #E
            client.sendall(str.encode(msg)) #E

        client.close()

class Receiver(Thread):
    def run(self) -> None:
        self.name = "Receiver"
        server = socket.socket(socket.AF_UNIX, socket.SOCK_STREAM)
        server.bind(SOCK_FILE) #G
        server.listen() #H

        print(f"{current_thread().name}: Listening of incoming me")
        conn, addr = server.accept() #I

        while True: #J
            data = conn.recv(BUFFER_SIZE) #J
            if not data: #J
                break #J
            message = data.decode() #J
            print(f"{current_thread().name}: Received: '{message}")

        server.close()

def main() -> None:
    if os.path.exists(SOCK_FILE):
        os.remove(SOCK_FILE)

    receiver = Receiver()
    receiver.start()
    time.sleep(1)
    sender = Sender()
    sender.start()

    for thread in [receiver, sender]:
        thread.join()

```

```
os.remove(SOCK_FILE)

if __name__ == "__main__":
    main()
```

Here we have created two threads, `Sender` and `Receiver`, each has its own socket. The only difference between them is the `Receiver` is in listening mode, waiting for incoming senders to send their messages.

Output:

```
Receiver: Listening of incoming messages...
Sender: Send: 'Hello'
Sender: Send: ''
Receiver: Received: 'Hello'
Receiver: Received: ''
Sender: Send: 'world!'
Receiver: Received: 'world!'
```

This is probably the simplest and best-known way to implement IPC, but it is also costly because it requires serialization, which in turn requires the developer to think about what data actually needs to be transmitted. On the bright side, sockets are generally more flexible and can be extended to network sockets if needed with almost no changes and easily scale your program to multiple machines. We will talk more about that in the Part 3 of the book.

#### NOTE

This is not a complete list of IPCs, only the most popular and the ones we will need later on. For example, *signals* are one of the oldest methods of inter-process communication, and there are unique things, such as *mailslots*<sup>[1]</sup>, which are only available in Windows.

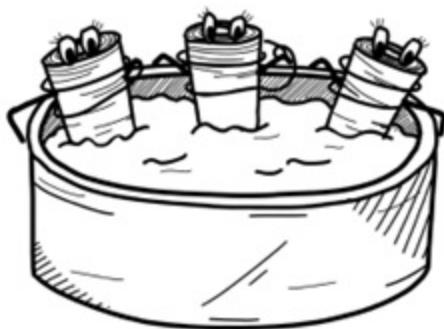
Having discussed IPCs we have covered our concurrency fundamentals and now we are ready to start with our first concurrency pattern – *the Thread pool*.

## 5.2 Thread pool pattern

Developing software using threads can be a daunting task. Not only are threads a low-level concurrency construct that requires manual management, but the synchronization mechanisms that are typically employed with threads can complicate software design without necessarily improving performance. Moreover, since the optimal number of threads for an application can vary dynamically based on the current system load and hardware configuration, creating a robust thread management solution is exceedingly challenging.

Despite the challenges, most concurrent applications actively use multiple threads. However, this does not mean that threads are explicit programming language entities. Instead, the runtime environment can map other programming language concurrency constructs to actual threads at runtime. One of the commonly implemented and widely used patterns in various frameworks and programming languages is *Thread pool*.

As the name implies, a thread pool consists of creating a small collection of long-running worker threads created at program startup, then putting them into a pool (a container). When a task needs to be executed, the pool takes one of the pre-created threads and executes it; the developer does not need to create them himself. Sending tasks to the thread pool is similar to adding them to the to-do list for worker threads.



Reusing threads with a thread pool eliminates the overhead associated with creating new threads and provide protection against the unexpected failure of the task, such as raising an exception, without impacting the worker thread itself. It becomes a real advantage when the time required to perform a task is less than the time required to create a new thread.

## **NOTE**

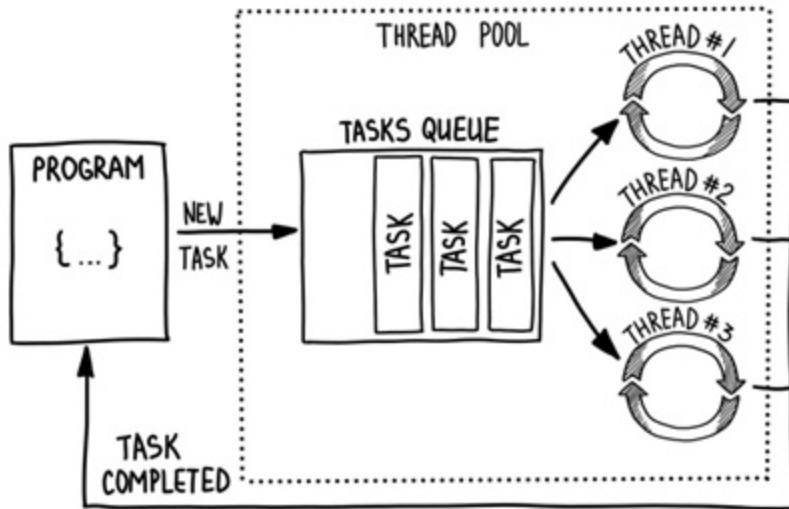
Thread pool takes care of creating, managing, and scheduling worker threads, which can become a complex and costly task if not handled carefully. Thread pools come in different types, with different scheduling and execution techniques, with a fixed number of threads or the ability to dynamically change the size of the pool depending on the workload.

Suppose we have a large set of tasks to process using multiple threads, such as cracking passwords, as described in the second chapter. By dividing the possible passwords into smaller chunks and assigning them to separate threads, we can achieve concurrency in our processing. In this scenario, we need a main thread that generates tasks for the worker threads running in the background.

To facilitate communication between the main thread and the worker threads running in the background, we need a storage mechanism that can act as a link between them. This storage should prioritize processing tasks in the order they are received. Moreover, any free worker thread should be able to pick up and process the next available task from this storage.

How to build such communication between threads?

Yes, message queues are a means of communication between threads within a pool. A queue logically consists of a list of tasks. Threads in the pool retrieve tasks from message queue and process them concurrently.



The implementation in different programming languages may differ. Below is an example of a thread pool implementation in Python:

```
# Chapter 5/thread_pool.py
import time
import queue
import typing as T
from threading import Thread, current_thread

Callback = T.Callable[..., None]
Task = T.Tuple[Callback, T.Any, T.Any]
TaskQueue = queue.Queue

class Worker(Thread):
    def __init__(self, tasks: queue.Queue[Task]):
        super().__init__()
        self.tasks = tasks

    def run(self) -> None:
        while True: #A
            func, args, kargs = self.tasks.get() #A
            try: #A
                func(*args, **kargs) #A
            except Exception as e: #A
                print(e) #A
            self.tasks.task_done() #A

class ThreadPool:
```

```

def __init__(self, num_threads: int):
    self.tasks: TaskQueue = queue.Queue(num_threads) #B
    self.num_threads = num_threads

    for _ in range(self.num_threads): #C
        worker = Worker(self.tasks) #C
        worker.setDaemon(True) #C
        worker.start() #C

def submit(self, func: Callback, *args, **kargs) -> None:
    self.tasks.put((func, args, kargs))

def wait_completion(self) -> None:
    self.tasks.join() #D

def cpu_waster(i: int) -> None:
    name = current_thread().getName()
    print(f"{name} doing {i} work")
    time.sleep(3)

def main() -> None:
    pool = ThreadPool(num_threads=5) #E
    for i in range(20): #F
        pool.submit(cpu_waster, i) #F

    print("All work requests sent")
    pool.wait_completion()
    print("All work completed")

if __name__ == "__main__":
    main()

```

When we create this thread pool, it automatically creates a number of threads and a message queue where incoming tasks are stored. Next, in the main thread we add a lot of tasks for the pool to process and wait for them to finish.

When a new task arrives, available thread wakes up, executes the task, and returns back to the Ready state. This avoids the relatively costly creation and termination of a thread for each task in progress and takes thread management out of the control of the developer, passing it to a library or operating system that is better suited to optimizing program execution.

## NOTE

Check Chapter 5/library\_thread\_pool.py for the Python libraries implementation of Thread pool pattern.

Thread pooling is a good default choice for most concurrent applications, but there are some scenarios where it makes sense to create and manage your own threads instead of using thread pool:

- You want to control different priorities of the threads
- You have tasks that cause the thread to block for a long time. Most thread pool implementations have a maximum number of threads, so a large number of blocked threads might prevent tasks from starting in the thread pool.
- You need to have a static identifier associated with the thread
- You want to dedicate a whole thread to one specific task

As promised, let's dive into the implementation of communication concepts and summarize our knowledge of concurrent application execution along the way.

### 5.2.1 Cracking passwords revisited

Now that we have acquired new knowledge, let us proceed with the unfinished implementation from Chapter 2 of the password cracking program using pools and processes (there is a Python limitation on using threads<sup>[2]</sup>, but it should not matter for other languages):

```
# Chapter 5/password_cracking_parallel.py
def crack_chunk(crypto_hash: str, length: int, chunk_start: int,
                 chunk_end: int) -> T.Union[str, None]:
    print(f"Processing {chunk_start} to {chunk_end}")
    combinations = get_combinations(length=length, min_number=chu
                                    max_number=chunk_end)
    for combination in combinations:
        if check_password(crypto_hash, combination):
            return combination #A
    return #B
```

```

def crack_password_parallel(crypto_hash: str, length: int) -> Non
    num_cores = os.cpu_count() #C
    print("Processing number combinations concurrently")
    start_time = time.perf_counter()

    with Pool() as pool: #D
        arguments = ((crypto_hash, length, chunk_start, chunk_end
                      chunk_start, chunk_end in #D
                      get_chunks(num_cores, length)) #D
        results = pool.starmap(crack_chunk, arguments) #D
        print("Waiting for chunks to finish")
        pool.close() #E
        pool.join() #F

    result = [res for res in results if res]
    print(f"PASSWORD CRACKED: {result[0]}")
    process_time = time.perf_counter() - start_time
    print(f"PROCESS TIME: {process_time}")

if __name__ == "__main__":
    crypto_hash = "e24df920078c3dd4e7e8d2442f00e5c9ab2a231bb3918d
    length = 8
    crack_password_parallel(crypto_hash, length)

```

Here the main thread creates number of worker threads equals to the number of available CPU cores using thread pool pattern. Each of the worker threads does the same thing as in the original version from the original chapter and we process all of the password chunks concurrently.

Output would look similar to this:

```

Processing number combinations concurrently
Chunk submitted checking 0 to 12499998
Chunk submitted checking 12499999 to 24999998
Chunk submitted checking 24999999 to 37499998
Chunk submitted checking 37499999 to 49999998
Chunk submitted checking 49999999 to 62499998
Chunk submitted checking 62499999 to 74999998
Chunk submitted checking 74999999 to 87499998
Chunk submitted checking 87499999 to 99999999
Waiting for chunks to finish
PASSWORD CRACKED: 87654321
PROCESS TIME: 17.183910416

```

We get more than 3x speedup from our original sequential implementation.  
Great job!



Now we can implement a lot of things with the use of parallel hardware, but sometimes parallel hardware is a luxury as you have to use only one core. That is not a reason to give up concurrency because this is where concurrency beats parallelism. More in the next chapter.

## 5.3 Recap

- The mechanism by which threads and processes synchronize themselves and exchange data is called *inter-process communication* or *IPC*.
- Each of the IPC mechanisms has its advantages and disadvantages; each is the optimal solution for a particular problem.
- *Shared memory* mechanism used when threads or processes need to efficiently exchange large amounts of data but have a problem with synchronizing access to the data.
- *Pipes* provide an efficient way to implement synchronous communication between producer/consumer processes. Named pipes provide a simple interface for transferring data between two processes, whether they are on the same computer or on a network.
- *Message queue* between processes or threads is a way of asynchronously exchanging data. Message queues are really used to implement weakly coupled systems.
- *Sockets* is a two-way communication channel which can use networking

capabilities. Here data communication will be taking place through the sockets interface, instead of through the file interface. In most cases, sockets typically provide the best combination of performance, scalability, and ease of use.

- *Thread pool* is a collection of worker threads that efficiently execute incoming tasks on behalf of the main thread of the program. Worker threads in a thread pool are designed to be re-used once the task is completed and provide protection against the unexpected failure of the task, such as raising an exception, without impacting the worker thread itself.

[1] Microsoft documentation, “Mailslots”, <https://learn.microsoft.com/en-us/windows/win32/ipc/mailslots>

[2] Python documentation, “Thread State and the Global Interpreter Lock”, <https://docs.python.org/3/c-api/init.html#thread-state-and-the-global-interpreter-lock>

# 6 Multitasking

## In this chapter

- You learn how to identify and analyze possible bottlenecks in your application
- You will learn how to run multiple tasks concurrently in the absence of parallel hardware
- You learn about the preemptive multitasking technique and how to apply it to solve the I/O-bound problems, its pros and cons

Do you ever stop to marvel at the sheer multitasking ability of your computer? It's truly incredible how it can handle multiple applications running at the same time, all while you continue to work on a text editor without a hitch. It's a feat that we often take for granted, but it's a testament to the impressive capabilities of modern computing.

But have you ever stopped to wonder how your computer accomplishes all of this? How does it manage to juggle so many tasks at once? Even more interestingly, what types of tasks are being handled and how are they classified?

In this chapter, we'll take a deeper dive into the concept of concurrency and explore the fascinating world of multitasking. By introducing multitasking into the runtime layer, we'll gain a better understanding of how our machines can handle a variety of tasks simultaneously. But before we delve into the intricacies of multitasking, we'll first take a closer look at the different types of tasks that our computers are capable of handling.

## 6.1 CPU-bound and I/O-bound applications

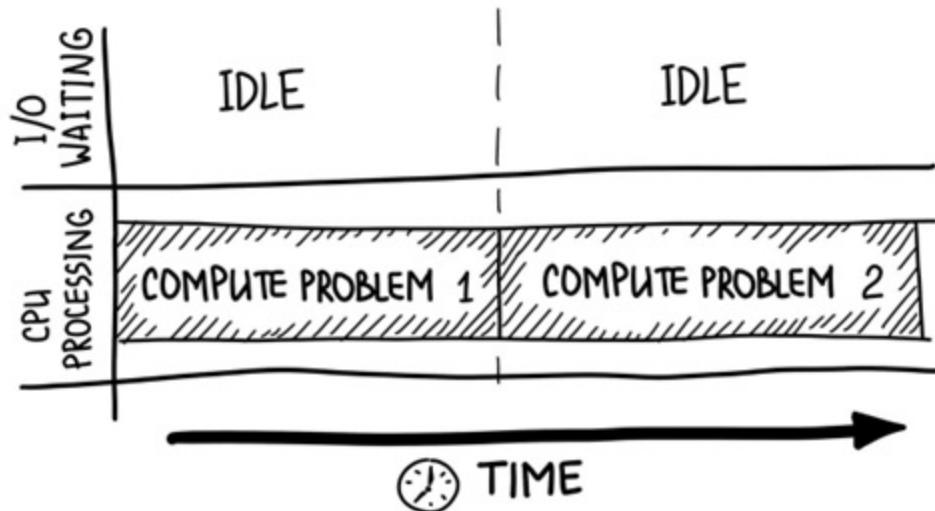
Applications consist of numerical, arithmetic, and logical operations. They also read from a keyboard, hard drive or network card; and produce output in the form of writing files, printing to “high-speed” printers or sending signals

to the display. While the first three of the load requires intensive CPU work, the rest needs to communicate with some type of device by sending and receiving signals. Most of the time that does not require CPU at all as there is nothing to compute, just wait for the response from the device. Such operations are also known as *input-output operations* or I/O. Consequently, it does not always make sense to give some tasks the use of the CPU. First, you need to understand the type of load.

An application is considered *bound* by something when the required resource for its work is the bottleneck for achieving increased performance. There are two main types of operations: *CPU-bound* and *I/O-bound*.

### 6.1.1 CPU-bound

So far, we've mostly been talking about CPU-bound applications. An application is bound by the CPU if it would run faster if the CPU was faster, i.e., it spends most of its time just using the CPU doing some kind of computation.



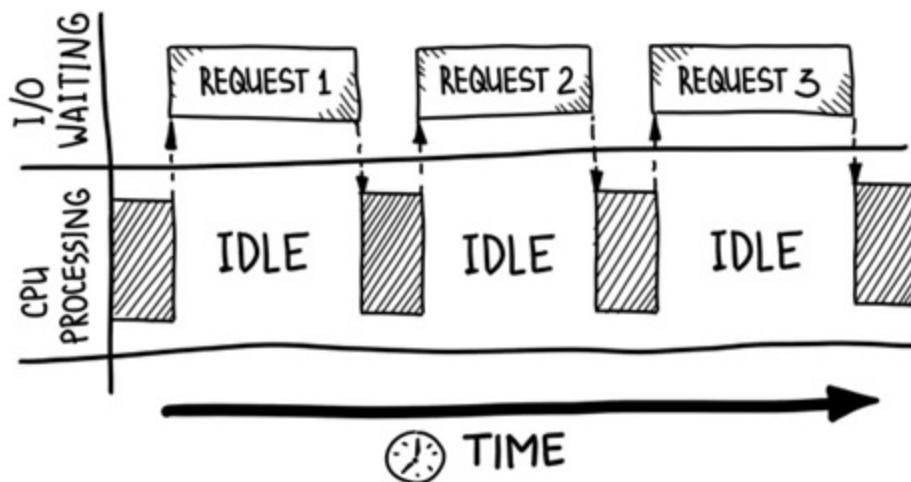
Some examples of CPU-bound operations:

- Mathematical operations like addition, subtraction, division, matrix multiplication, etc
- Encryption and decryption algorithms involve a lot of computationally intensive operations like prime factorization, computing cryptographic

- functions, etc
- Image processing, video processing, etc
- Executing algorithms like binary search, sorting, etc

### 6.1.2 I/O-bound

An application is bound by I/O if it would run faster if the I/O subsystem was faster. The kind of I/O subsystem can vary here, but you can associate it with reading from disk or getting user input or waiting for network response. An application that goes through a huge file looking for search term can become I/O bound because the bottleneck is in reading a lot of data from the disk.



Idle sections on the illustration represent periods of time when a particular task is pending and thus cannot advance. A common reason is waiting for I/O to be performed. But to perform various I/O operations CPU often does nothing but wait for data to be transferred to or from an external device, and CPU time is expensive. Examples of I/O-bound operations are:

- Most graphical user interface (GUI) applications are I/O-bound, even if they never read from or write to the disk because they spend most of their time waiting on user interaction via the keyboard or mouse
- Processes that spend most of their time in doing disk I/O or network I/O, like databases or web servers

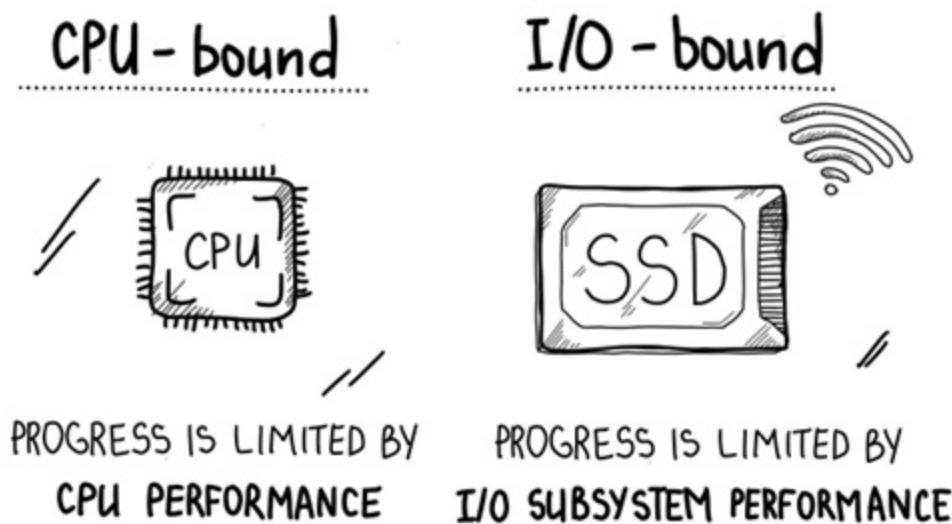
### 6.1.3 Identifying bottlenecks

When determining your application bottleneck, you should think about which resource you need to improve so your application performs better. That leads you directly to the source of what resource your application is hanging onto. Often CPU and I/O operations are highlighted as the most important ones.

#### NOTE

Of course, this isn't just about I/O-only work and CPU-only work. We can also think about memory and cache work, but for majority of developers and for the purposes of this book it is enough to consider the difference between CPU and I/O.

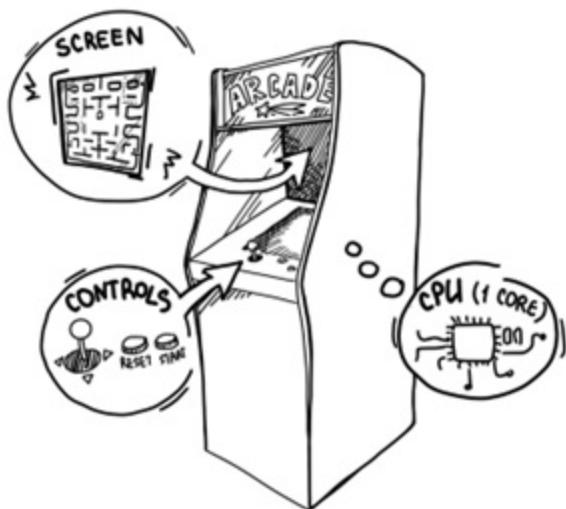
Imagine two programs. The first program performs the multiplication of two huge matrices and returns the answer. The second program writes a huge amount of information from the network to a file on a disk. It is clear that these programs will not be equally accelerated by the increased CPU clock speed or increased number of cores. What does it matter how many cores there are if most of the time they are waiting for the next batch of data to be transferred to disk? One core or a thousand, we will not get a performance increase with an I/O-bound load. But if we have a CPU-bound load, there is a chance to get a boost when we parallelize our program so it can be able to utilize multiple cores.



## 6.2 The need for multitasking

Applications naturally become more and more I/O-bound. This is because CPU speed has historically increased allowing more instructions to be executed in a given time, while data transfer speed has not increased that much. Therefore, the limiting factor in programs are often I/O-bound operations that block the CPU. But they can be identified and executed in background like most modern runtime systems do.

Imagine that your friend Alan found an old arcade machine in his parents' attic. It had an old single-core processor, a big pixel screen, and a joystick. He approached you, his only developer friend, and asked you to implement a Pac-Man-like game for the machine.



The game is interactive and needs the gamer's input to move the character in the game. At the same time, the world inside the game is dynamic. Ghosts need to move at the same time as the gamer controls character in the game. And the gamer should see how the world is changing as well as how their character is moving.



Your first steps were to create some game functionality divided between three functions:

- `get_user_input()` – this function basically gets the input from the controllers and save it in the game internal state. It is an I/O-bound operation.
- `compute_game_world()` – this function computes game world according to the game rules, gamer's input and game internal state. It is a CPU-bound operation.
- `render_next_screen()` – this function will get the game internal state and render the game world into the screen. It is an I/O-bound operation.

Given those three functions, you can see that we have a problem – many things should be happening simultaneously for the gamer, but we only have an old one-core CPU.

*How can we solve this problem?*

Let's start with trying to create a parallel program using one of the OS abstractions. We utilize threads for this problem, so we have one process and three threads. Using threads are beneficial as we need to share data between the tasks, and easier as they can share the same process address space. So, our program will look like the following:

```
# Chapter 6/arcade_machine.py
import typing as T
from threading import Thread, Event
```

```

from pacman import get_user_input, compute_game_world, render_next_screen

processor_free = Event() #A
processor_free.set() #A

class Task(Thread):
    def __init__(self, func: T.Callable[..., None]): #B
        super().__init__()
        self.func = func

    def run(self) -> None:
        while True:
            processor_free.wait() #A
            processor_free.clear() #A
            self.func() #B

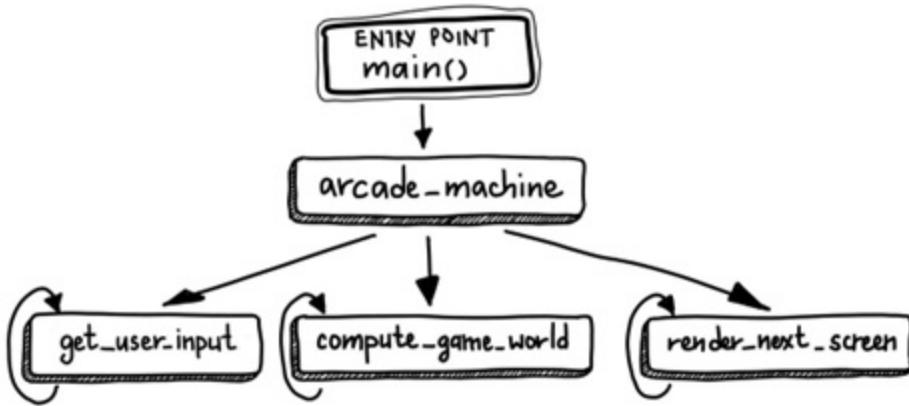
def arcade_machine() -> None:
    get_user_input_task = Task(get_user_input) #C
    compute_game_world_task = Task(compute_game_world) #C
    render_next_screen_task = Task(render_next_screen) #C

    get_user_input_task.start() #C
    compute_game_world_task.start() #C
    render_next_screen_task.start() #C

if __name__ == "__main__":
    arcade_machine()

```

Here, we initialize three threads corresponding to one of the three functions. Each function inside a thread runs in its own endless loop (assuming we don't stop a thread after a single execution), so that our threads are always kept working as a gamer continues playing.



Unfortunately, if we start this program, it will get stuck on the first thread, asking user input in the infinite loop and will do nothing else as our CPU has room just for one thread. So, we cannot really utilize parallelism here, as it requires us to have the proper hardware for that. Do not worry. We still can utilize concurrency with multitasking!

Before we apply multitasking to our arcade problem, we need to learn its fundamentals. Let's put our problem to one side for now, as we learn more about multitasking in the next section.

## 6.3 Multitasking from a bird's eye view

In today's world, multitasking is everywhere. We multitask as we listen to music while walking, or taking a call while cooking, or eating while reading a book.

*Multitasking* is the concept of performing multiple tasks over a period of time by executing them concurrently. Multitasking can be compared to the plate spinner in a circus who juggles multiple plates while they are spinning on sticks. He rushes from plate to plate, trying to keep them spinning so they wouldn't fall off the sticks.



In a “true” multitasking system, operations run in parallel, but parallel execution requires appropriate hardware support. However, the appearance of multitasking can be achieved even on older processors using several tricks.

## 6.4 Preemptive multitasking

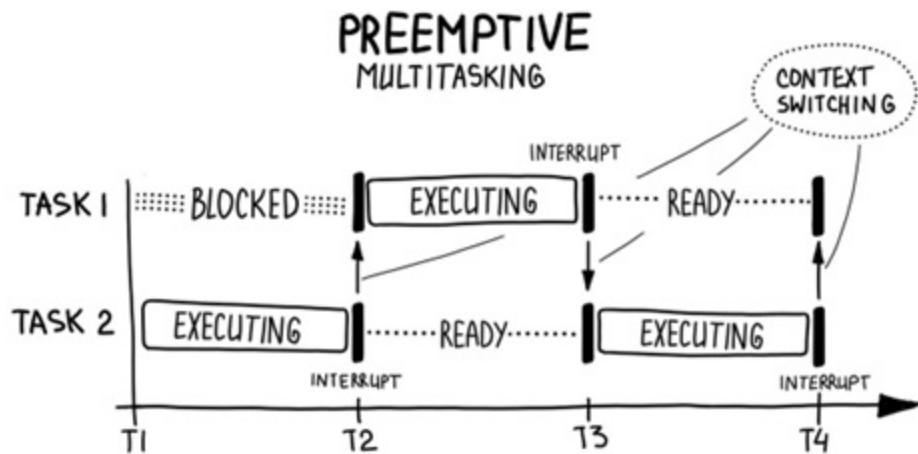
The main task of the OS is resource management, and one of the most important resources for it to manage is the CPU. The OS must be able to allow every program to be executed on the CPU. This means that it should be able to run a task for a while, but then “park” it and allow another task to run. The problem is that most applications are not written to be attentive to other running applications. So, the OS needs a way to suspend the execution of an application without it being involved.

The idea behind *preemptive multitasking* is to define a period of time a single task is allowed to run. This period is also known as *time slice* because OS tries to guarantee a slice of CPU time for each running task. And that’s why this scheduling technique is called the *time-sharing* policy<sup>[1]</sup>. The CPU will execute the task in the Ready state during this time slice if it does not perform any blocking operations.

When the time slice expires, the scheduler *interrupts* the task (*preempts*) and allows another task to run in its place while the first task waits its turn again. An interrupt is a signal to the CPU to stop the task in order to resume it at a later time. There are three types of interrupts: hardware interrupts with a

special interrupt controller (e.g. pressing a keyboard button, completing a write to a file); and software interrupts (e.g. system calls) caused by the application itself, errors and timer interrupts.

Imagine a processor allocates a small amount of time to each of the running tasks and then quickly switches between them, allowing each task to execute in the interleaved fashion. By switching quickly and passing control to the tasks in the queue, the OS creates the illusion of multitasking, although only one task is executing at any given time. The diagram below shows the progress of the three tasks as a function of time. Time moves from left to right, and the lines indicate which task is in progress at any given time. The illustration shows the perceived simultaneous execution model.



Most operating systems developed in the last decade provide preemptive multitasking (we contrast it with the cooperative multitasking in Chapter 12). If you work on Linux, MacOS, or Windows, you work on an OS that has a preemptive multitasking.

To better understand how multitasking can be implemented, we will go back to our arcade machine example.

#### 6.4.1 Arcade machine with preemptive multitasking

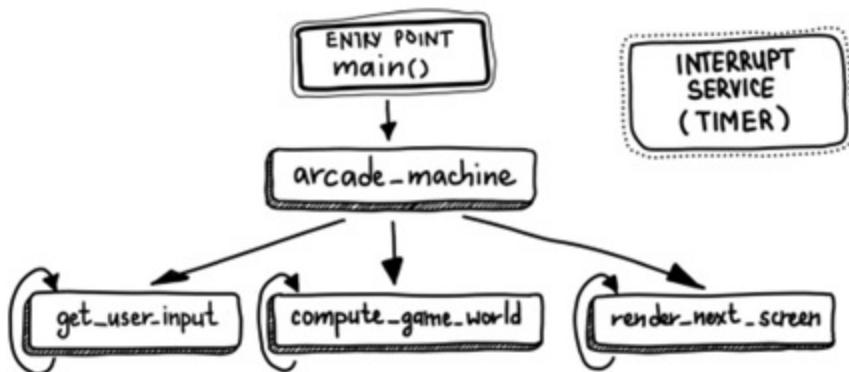
We have two I/O-bound operations that are waiting for some event to occur, hence blocking the CPU. For example, the `get_user_input_task` thread is

waiting for the gamer to press a button on controller.

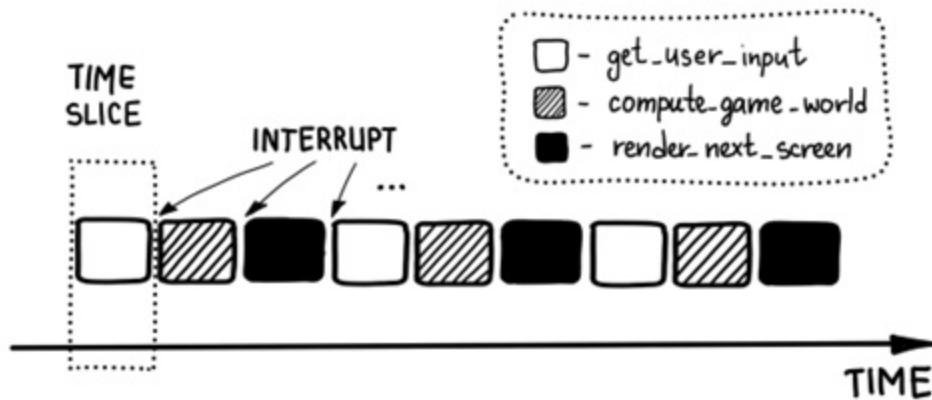
We have an old one-core CPU on arcade machine, but it's still very fast compared to human reflexes. It takes an unimaginable amount of time for the CPU until a person moves their finger over the button and presses it. The fastest possible conscious human reactions take around 0.15 seconds, if we have a 2 GHz processor it can execute 300 million cycles in the same amount of time. Which can roughly be thought of as the number of instructions. While we wait for human input (gamer pressing the button) we waste CPU computation resources as the CPU core just waits and does nothing. We can utilize this unoccupied CPU time by passing control to computational tasks during idle time.

Essentially, we need to implement part of the operating system. This can be done via preemptive multitasking – giving each thread a CPU time to run, and then passing processor to the next thread. We will use just simple *time-sharing policy* and divide all available CPU time into equal time slices.

This is where the timer comes to the rescue. Timers tick at regular intervals and can be set to interrupt after a certain number of ticks. This interrupt pauses current thread, allowing us to let another thread use the processor. So, the diagram of our program will look like this:



Implementing the runtime system divides the processor time between the threads into time slices to give the impression they are running concurrently:



In the code it looks like the following:

```
# Chapter 6/arcade_machine_multitasking.py
import typing as T
from threading import Thread, Timer, Event

from pacman import get_user_input, compute_game_world, render_nex
processor_free = Event()
processor_free.set()
TIME_SLICE = 0.5 #A

class Task(Thread):
    def __init__(self, func: T.Callable[..., None]):
        super().__init__()
        self.func = func

    def run(self) -> None:
        while True:
            processor_free.wait()
            processor_free.clear()
            self.func()

class InterruptService(Timer):
    def __init__(self):
        super().__init__(TIME_SLICE, lambda: None)

    def run(self):
        while not self.finished.wait(self.interval): #B
            print("Tick!") #B
            processor_free.set() #B
```

```

def arcade_machine() -> None:
    get_user_input_task = Task(get_user_input)
    compute_game_world_task = Task(compute_game_world)
    render_next_screen_task = Task(render_next_screen)

    InterruptService().start()
    get_user_input_task.start()
    compute_game_world_task.start()
    render_next_screen_task.start()

if __name__ == "__main__":
    arcade_machine()

```

We have implemented multitasking by putting threads into one infinite control loop, where we can, in an interleaved manner, provide each thread with a CPU time slice. If the interleaving happens fast enough (say 10 milliseconds), gamer get the impression of simultaneous execution. It seems to the gamer that all the attention of the game is devoted only to him, when in fact the processor and the computer system as a whole may be working on a completely different task at the moment. The gamer gets the impression of parallel execution because of the extremely fast switching between threads.

So physically we still have serial execution of tasks because we have limits on processing resources, but conceptually all three of our threads is “in progress”, making them run concurrently.

Concurrent computations have overlapping lifetimes. As we’ve seen with proper hardware, we can achieve true parallelism with physically simultaneous tasks execution, while multitasking helps us abstract away overlapping execution to the runtime system. Thus, true parallelism is essentially an implementation detail of the execution, while multitasking is part of the computational model.

There is one pitfall here that we have missed, so let’s step back a bit.

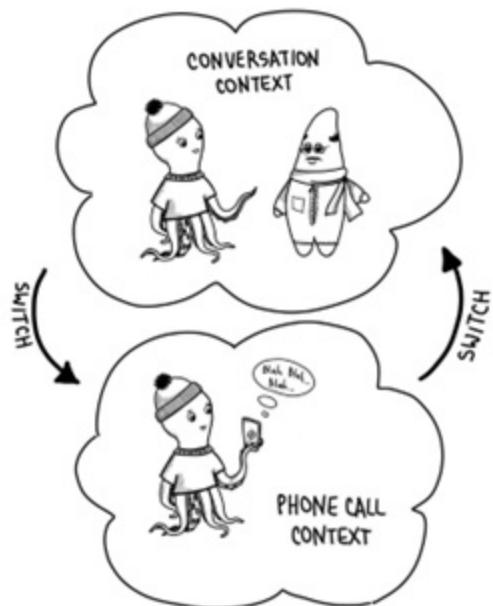
### 6.4.2 Context switching

The *execution context* of a task contains the code that's currently running

(namely the instruction pointer), and everything that aids in its execution on a CPU core (CPU flags, key registers, variables, open files, connections, etc); it must be loaded back into the processor before the code resumes execution. Consequently, *context switching* is a physical act of swapping from one task's context to another without losing the data so that it can then be recovered back to the same moment it was switched. The task that was selected from the Ready queue moves into a Running state.



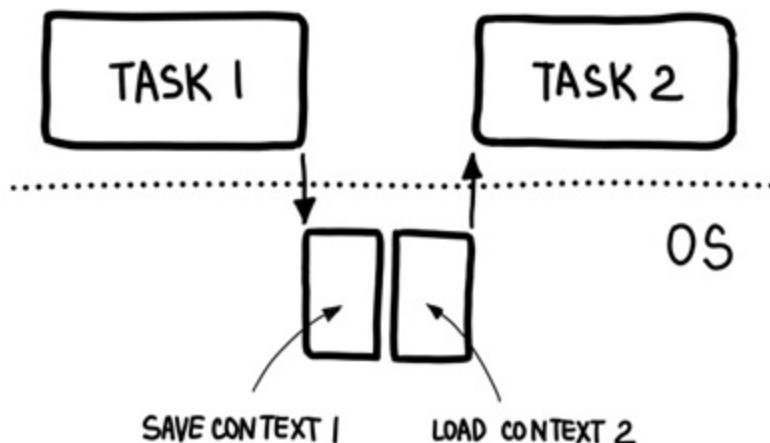
Imagine you're having an engaging conversation with a friend, but then your phone starts ringing and you get distracted. You say “Wait a minute” to your friend and pick up the phone. Now you are entering a new conversation, a new context. When it's clear to you who's calling and what they want, you can focus on their request. After the phone call is over, you go back to the initial conversation. There are times when you forget the context where you left off, but once your friend reminds you of what you were talking about, you can continue. It happens quickly, but not instantly.



Like you, the processor needs to find the context in which the task was and

reconstruct it. From the point of view of the task, everything around him is in the same state as it was before. It doesn't make any difference whether he started just now or 25 minutes ago. Context switching is a procedure performed by OS, and it's one of the key mechanisms that provides the OS with multitasking feature.

Context switches are considered costly because they require system resources. Switching from one task to another requires certain actions. First the context of a running task must be saved somewhere; then the new task starts. If the new task was previously in progress, it will also have a stored context, which must be pre-loaded before it can continue execution. When the new task completes, the scheduler saves its final context and restores the context of the preempted task. The preempted task resumes execution as if nothing had happened to it at all (except for the time shift).



The overhead associated with saving and restoring state when switching contexts seems to have a negative impact on program performance as the application loses the ability to execute instructions when switching contexts. It all depends on the type of operations your program is performing.

#### NOTE

The amount of latency incurred during context switching depends on various factors, but let's take a reasonable ~800 to ~1300 nanoseconds per context

switch (the numbers I get using [LMbench](#) on my laptop). Given that the hardware should be able to reasonably execute on average 12 instructions per nanosecond per core, context switching could cost roughly ~9k to ~15k executed instructions.

Be careful when using multiple tasks in an application because system performance can degrade if too many tasks are running. The system will waste a lot of usable time in the context-switching loop.

Understanding what multitasking is, let's integrate it into the runtime environment and combine all the other concurrency concepts.

## 6.5 Multitasking environment

In the early days of computers people didn't think on doing more than one task at the same time in a single machine because the OS and the applications where not designed for multitasking. So you had to exit one application and open a new one every time.

Then the ability to perform multiple tasks concurrently has become one of the most important requirements for runtime systems. This requirement is addressed by multitasking. Although real parallel processing is not achieved, and even though there is some overhead associated with switching between tasks, interleaved execution provides significant advantages in processing efficiency and program structuring.

For the user, the advantage of multitasking system is the ability to have multiple applications running at the same time. For example, a user can edit a document in one application while watching movie in another.

For the developer, the advantage of multitasking is the ability to create applications using more than one process and to create processes using more than one execution thread. For example, a process might have a user interface thread that handles user interaction (keyboard and mouse input) and worker threads that perform computational tasks while the user interface thread waits for user input.

Delegating the scheduling and coordination of tasks to a runtime system simplifies the development process while allowing flexibility to adapt transparently to different hardware or software architectures. Using different runtime environment (computer OS or IoT runtime environment or Manufacturing Operating System, etc.) allows developers to optimize for different purposes. For example, minimizing power consumption may require a very different schedulers than maximizing throughput.

#### **NOTE**

In the 1960s and 1970s, the development of multitasking operating systems such as IBM's OS/360 and Unix allowed multiple programs to run on a single computer but required more memory than was physically available. To solve this problem, virtual memory was developed, a technique that temporarily transfers data from RAM to disk storage, allowing computers to use more memory than they have. This development enabled computers to run more programs simultaneously, and virtual memory remains an essential component of modern operating systems.

### **6.5.1 Multitasking operating system**

Multitasking in multi-processor environments can be supplemented by distributing different tasks to the available CPU cores.

The CPU does not know anything about processes or threads at all. CPU's job is just to execute machine instructions. Thus, from the CPU's point of view there is only one execution thread: serial execution of all incoming machine instructions from OS. To make that happen, the operating system uses thread and process abstractions. And the task of the operating system when there are multiple running threads for a single processor core is to somehow juggle the tasks, simulating for the user the parallel execution, but, in fact, making them run concurrently.

Multitasking is a runtime system-level feature, there is no concept of multitasking at the hardware level. However, implementing multitasking is not without its challenges, and often requires the runtime system to have strong task isolation and an efficient task scheduler.

## 6.5.2 Task isolation

By definition of multitasking, there are multiple tasks in the OS. You may have already guessed that we are now going to refer to processes and threads abstractions provided by OS but if you are creating your own runtime system things can be different.

There are two ways to create multiple tasks:

- as a single process with multiple threads; or
- as multiple processes, each having one or more threads.

As previously discussed, each of these options has its pros and cons, but they all provide, to a greater or lesser extent, isolation of the execution of tasks. The operating system, in turn, takes care of how these abstractions are mapped to the physical threads of the computer system and how they are executed on the hardware.

The operating system abstracts from how the hardware actually works, and even if the system has only one core, the OS gives the developer the illusion that it is not. So even if the system cannot use parallelism, developers can still use concurrent programming and take advantage of the operating system's multitasking. A program divided in this way can be written as if the processor is at its full disposal.

It is generally more efficient to implement multitasking by creating a single multithreaded process rather than multiple processes for the following reasons:

- The system can perform context switching faster for threads than for processes because a process has more overhead than a thread (a process context is larger than a thread context).
- All process threads share address space and can access global process variables, which simplifies communication between threads.

## 6.5.3 Task scheduling

The scheduler is the core of multitasking operating systems. The scheduler

chooses from all the tasks in the Ready state the one which should be executed next.

The idea behind scheduling the execution is quite simple. Something should always be running to make better use of processor time. If there are more tasks than processors in the system, which is common, some tasks will not run at all times but wait in the Ready state. The choice of which task should be executed at the next moment is the fundamental decision of the scheduler from the information about the tasks that are ready to run.

Since the scheduler allocates a limited resource (CPU time), the logic it follows is based on balancing conflicting goals and priorities. Typical goals are to maximize throughput (number of actions per time) or fairness (prioritizing or aligning computation) or to minimize response time (time to complete action) or minimize delay (making it react faster).

The scheduler can forcefully take control away from a task (e.g. by timer or when a task with a higher priority emerges) or just wait until the task gives control explicitly (by calling some system procedure) or implicitly (when it finishes) to the scheduler. This means the scheduler is unpredictable when it comes to what task will be selected for execution at any given time. Thus, developer should never write a program based on previously seen behavior because it is not guaranteed to happen every time. You must control the synchronization and coordination of tasks if you want to achieve determinism in your application. We'll talk about this in the next chapters.

Most importantly, the scheduler opens the door to implementing new methods of improving system performance without changing the program. Of course, introducing an additional layer between the application and the operating system increases execution overhead. For this approach to work, the performance benefits that the runtime environment can provide must exceed the runtime management overhead.

#### NOTE

We focused on the operating system in this chapter, but there are other runtime environments that implement the same multitasking concepts. For example, it is used with await in languages, such as JavaScript or Python, that

feature a single-threaded event-loop in their runtime. One of the most efficient JavaScript execution engines on the market, V8 and Go programming language, known for its scalability and small memory footprints, does its own multitasking, lying on top of the OS (at the user level). We will touch this topic in Chapter 12 when we talk about cooperative multitasking and asynchronous communication.

## 6.6 Recap

- There are two types of bottlenecks in program based on the resource that is used the most by the program: CPU-bound and I/O-bound
  - *CPU-bound* operation mostly requires processor resources to finish its computation. In this case the limitation is the speed at which the system can compute something
  - *I/O-bound* operation that mostly does I/O and it doesn't depend on your computation resources, e.g. waiting for disk operation to finish or external service to answer your request. In this case the limit is the speed of the hardware, such as how fast a disk can read data or how fast a network can transmit it.
- *Context switching* is a physical act of swapping from one task's context to another so that it can then be recovered back to the same moment it was switched later. Context switching is a procedure handled by the OS, and it's one of the key mechanisms that provides multitasking to OS.
  - Context switching is not free, so be careful when using multiple tasks in an application, because system performance can degrade if too many tasks are running – the system will waste a lot of usable time in the context-switching loop
- The ability to perform multiple tasks simultaneously is a critical requirement for runtime systems. It is solved by *multitasking*. This mechanism controls the interleaving and alternating execution of tasks. By constantly switching tasks, the system can maintain the illusion of simultaneous execution of tasks, although in fact the tasks are not executed in parallel.
- *Multitasking* is the concept of performing multiple tasks over a period of time by executing them concurrently. Multitasking is a runtime system-level feature, there is no concept of multitasking at the hardware level
  - In *preemptive multitasking* the scheduler prioritizes tasks and

- forces the tasks to pass the control to other tasks
- It is generally more efficient to implement multitasking by creating a single multithreaded process rather than multiple processes.
- It is important for the runtime system scheduler to distinguish between I/O-bound and CPU-bound tasks to ensure optimal use of system resources.

[1] If you're interested in learning more, here is a good video on the topic:  
<https://youtu.be/Q07PhW5sCEk>

# 7 Decomposition

## In this chapter

- You learn decomposition techniques, which involve efficiently breaking down programming problems into separate, independent tasks.
- You learn popular concurrency patterns for creating concurrent applications – Pipeline pattern, Map pattern, Fork/Join pattern and Map/Reduce pattern
- You learn how to choose the granularity of your applications.
- You will learn how to use agglomeration as a means of reducing communication overhead and increasing system performance

Previously we learned that concurrent programming implies decomposing a problem into independent units of concurrency, or tasks. Deciding how to decompose a problem into concurrent tasks is one of the more difficult, but also important steps. Automatically decomposing programs using a concurrent programming approach is a difficult research topic, thus, so in most cases, decomposition falls on the shoulders of the developers.

In this chapter, we will discuss methods and popular programming patterns for designing concurrent applications. We will talk about the application level of concurrency – where we focus on where we can find independence of the tasks and how to structure and design a program rather than how it will actually be executed, although we will touch on this part as well.

## 7.1 Dependency analysis

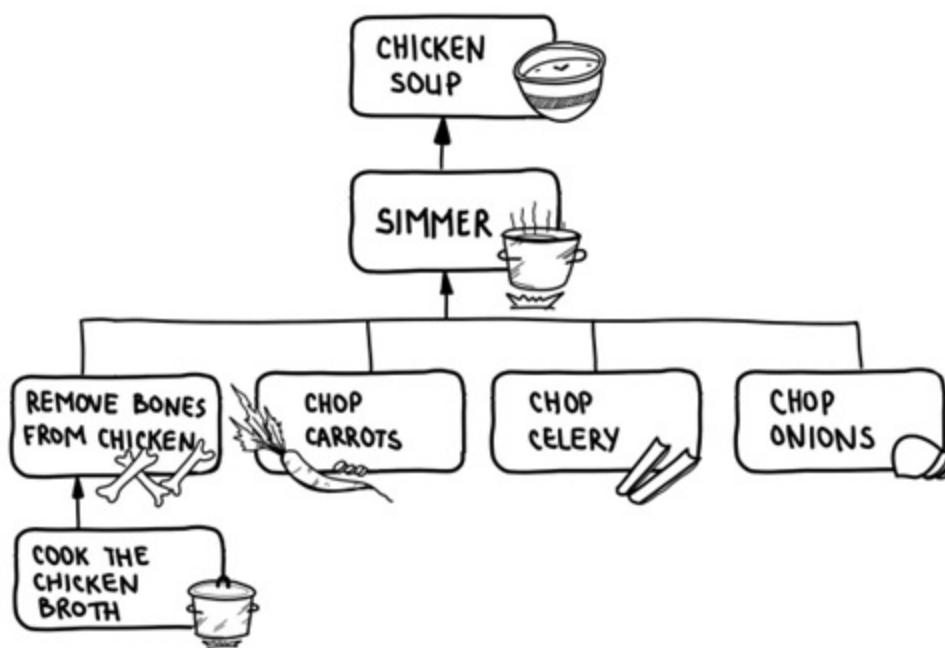
Decomposition of a problem into concurrent tasks is one of the first necessary steps in writing concurrent applications and is the key to concurrent programming. When you decide to decompose a programming problem into tasks, don't forget that tasks can have dependencies on other tasks. Therefore, the first step to decompose the problem is to find the dependencies of all its constituent tasks and identify the ones that are

independent. One method to help model how the tasks in a program relate to each other is to build the *task dependency graph*.

### 7.1.1 Task Dependency Graph

Dependency graphs help describe relationships between tasks.

Consider the steps for cooking a very simple chicken soup. To make chicken soup, you need to boil broth from chicken, then remove bones, chop carrots, chop celery, mix these ingredients together in the soup, and cook until tender. We can't start to simmer until we've done all four of the preceding tasks. Each of these steps represents a task, and going backwards from the result through each task dependencies we can build a dependency graph that would look like this:



There are several variations and ways of drawing these types of computational graphs, but their general purpose is to provide an abstract representation of the program. They help visualize relationships and dependencies between tasks. Each node represents a task and edge represents dependence.

Dependency graphs can also be used to get an idea of how concurrent a

program can be. The fact that there are no direct edges between the tasks of making broth and chopping vegetables indicates that concurrency is possible at this point. Thus, if this program can be implemented using threads, we would be able to create four separate threads; one for making broth and three others for chopping vegetables. All of them can be executed simultaneously. The same concept is used in runtime systems when scheduling individual tasks.

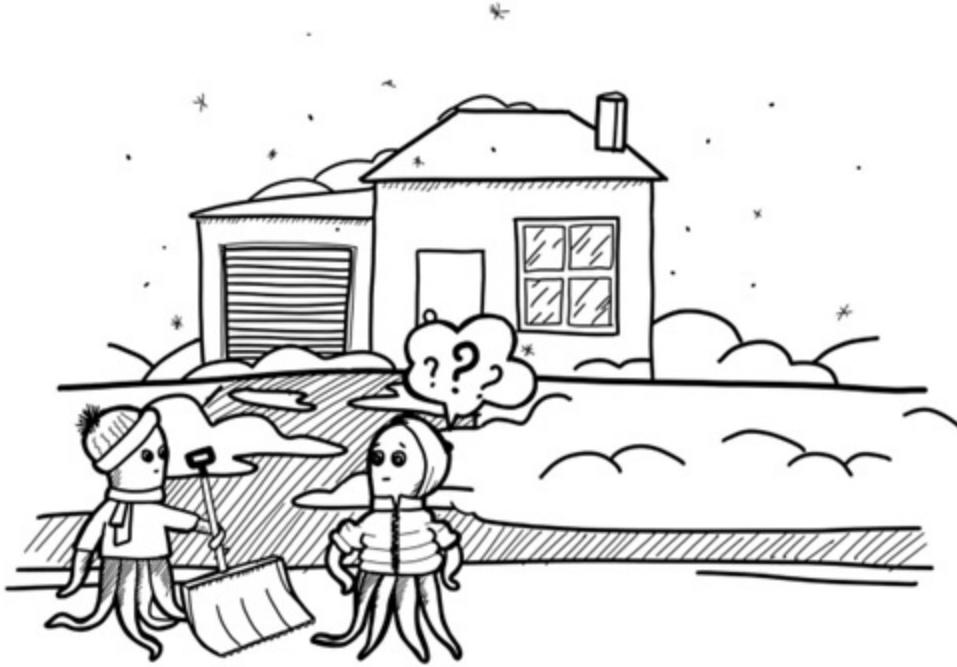
Building a dependency graph would be the first step towards program or system design. It helps to identify portions of work that can be performed concurrently. For now, we are ignoring problems of practical implementation, such as the number of processors or cores that can be used. All attention is focused on the possible concurrency of the original problem. Having said that about the dependency graph, let's look at it from different angles.

The two types of dependencies in code are control dependency and data dependency. The corresponding ways of dividing a problem into smaller tasks are *task decomposition* and *data decomposition*.

## 7.2 Task decomposition

Task decomposition answers the question: “How can a problem be decomposed into independent functionality that can be executed concurrently?” or to put it in plain English: “How can we split a problem into a bunch of tasks that we can perform all at once?”

Let's imagine there has been a major snowfall. You want to clear the area around the house by shoveling snow and scattering salt. Your friend comes over to help you finish the job faster, but you only have one shovel. So while one shovels, the other is waits to take a turn. Remembering while this process can make sense, having only one resource (the shovel) does not speed up the work, it only slows it down. Overhead on context switching makes this process very inefficient as it just interrupts the actual shoveling process all the time.



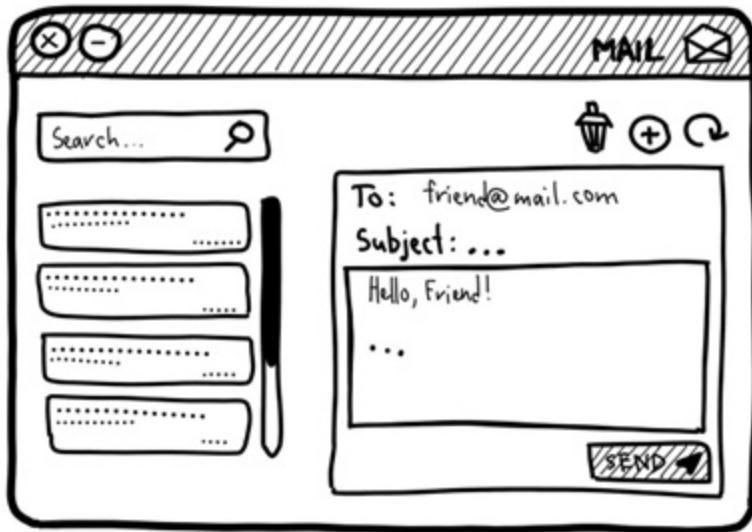
Having the same goal of clearing the area around the house, you decide to give your friend another subtask. While you clear the snow with the only shovel you have, let him scatter salt. By eliminating the wait for the shovel, you make the job more efficient. This is essentially what task decomposition (also known as task parallelism) gives you.



This is an obvious example to demonstrate breaking down the problem into tasks by functionality. But task decomposition often can be far from obvious, are often complex, and can be very subjective.

Task decomposition implies decomposition of the application into functionally independent tasks based on application functionality. Such decomposition is possible when the problem to be solved naturally consists of different types of tasks, each of which can be solved independently.

For example, an email management application would have a lot of functional requirements: a user interface; a way to reliably receive new emails; the ability for the user to write and send emails; and searching through emails should all be in standard features.

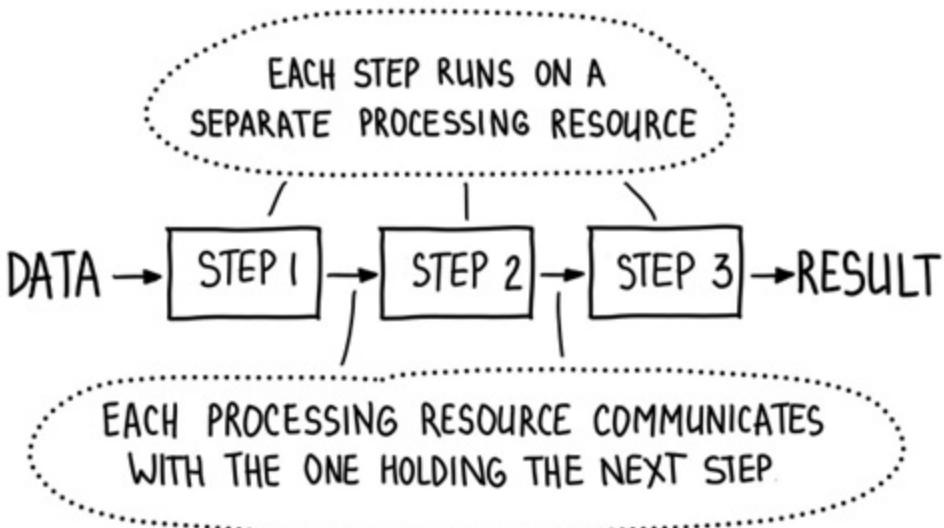


The tasks of finding the email and UI that lists those emails depend on the same data, but are completely independent of each other, so they can be split into two tasks and executed independently of each other. The same also applies to sending and receiving emails. For example, we can use different processors, each working with the same data but doing its job in its own task concurrently.

As we have seen, the functionality of the different tasks in task decomposition is very diverse with a wide range of operations used, and therefore it can only be used on MIMD/MISD systems.

### 7.2.1 Pipeline pattern

The most common pattern in task decomposition is the so-called *pipeline processing*. The essence of pipeline processing is to decompose the algorithm into several separate consecutive steps. Pipeline steps can then be distributed among the different cores. Each core is like one worker on an assembly line; having completed its work, it passes the result to the next one, at the same time accepting a new portion of data. Hence, cores can execute multiple chunks of data simultaneously, starting new computations while others are still running.

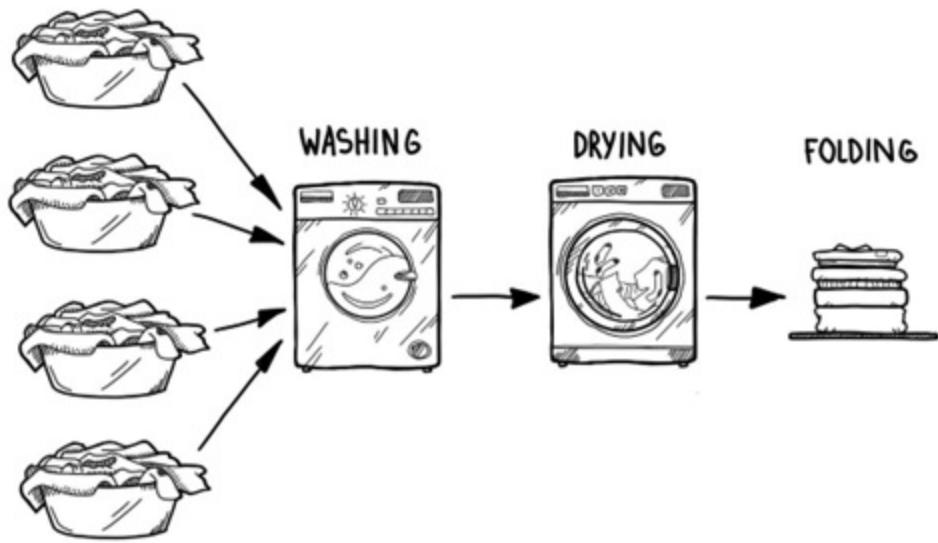


#### NOTE

Remember we talked about the infinite CPU execution cycle? The execution of a single instruction includes passing through the fetch instruction, decoding, execution, and store the results steps. In modern processors stages are designed in such a way that instructions can be executed using pipeline processing at such a low level.

Remember when we did the laundry in a hurry in Chapter 2? Let's bring it closer to reality – in addition to the laundry itself, which takes a decent amount of time, we also need to dry the laundry after washing and then fold it – we don't want to wear crumpled laundry to Hawaii, do we?

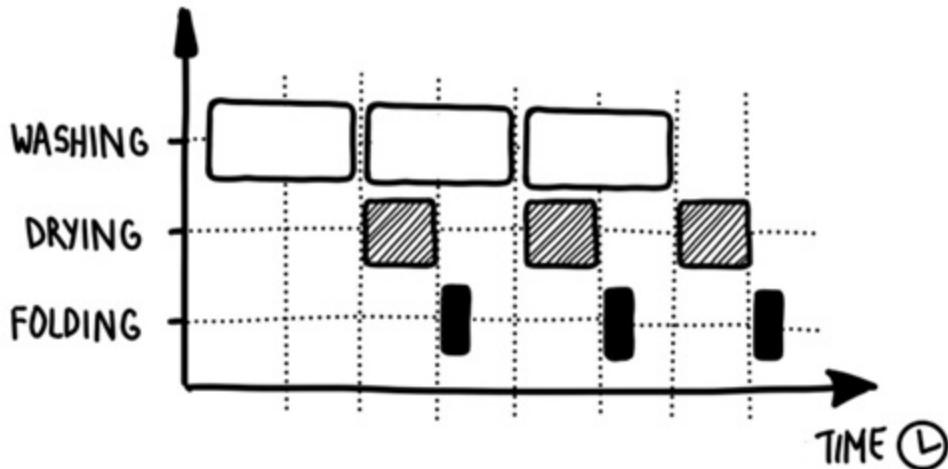
If you're not using pipeline processing, washing, drying, and folding four loads of laundry with one washer and dryer would look like this:



With this approach, we see that our resources (washer, dryer) are not fully utilized – there are times when they are idle while some other action is performed.

Using pipeline processing ensures that we are constantly using the washer and dryer without wasting any time – we divide three steps of the washing process into three different workers: washer, dryer and folder (the latter is probably you). Each worker has a lock on a common resource.

The first batch of laundry is ready to be washed and placed in the washer. When the laundry is washed and removed from the washer, it is transferred to the next stage of our pipeline, the dryer. While the first batch of laundry is drying in the dryer, the second batch can begin washing because the washing machine is idle at this point.



Concurrency occurs when a second load, going through our pipeline, ends up there at the same time as the first load. The coincidence of previously separated operations definitely has a positive effect on processing speed.

#### NOTE

One of the most popular patterns in the big data world is ETL (Extract Transform Load) – a popular paradigm for collecting and processing data from various sources that implements a pipeline pattern. Using ETL tools we Extract data from the source(s), Transform it into structured information which we Load into the target data warehouse or other target system.

In order to implement this type of functionality in our code, we need two things: a way to create independently running tasks and a way for tasks to communicate with each other. This is where threads and queues come to the rescue.

Let's see what that looks like in the code:

```
# Chapter 7/pipeline.py
import time
from queue import Queue
from threading import Thread

washload = str

class Washer(Thread):
```

```

def __init__(self, in_queue: Queue[Washload], out_queue: Queue[Washload]):
    super().__init__()
    self.in_queue = in_queue
    self.out_queue = out_queue

def run(self) -> None:
    while True:
        washload = self.in_queue.get() #A
        print(f"Washer: washing {washload}...")
        time.sleep(4) #B
        self.out_queue.put(f'{washload}') #C
        self.in_queue.task_done()

class Dryer(Thread):
    def __init__(self, in_queue: Queue[Washload], out_queue: Queue[Washload]):
        super().__init__()
        self.in_queue = in_queue
        self.out_queue = out_queue

    def run(self) -> None:
        while True:
            washload = self.in_queue.get() #A
            print(f"Dryer: drying {washload}...")
            time.sleep(2) #B
            self.out_queue.put(f'{washload}') #C
            self.in_queue.task_done()

class Folder(Thread):
    def __init__(self, in_queue: Queue[Washload]):
        super().__init__()
        self.in_queue = in_queue

    def run(self) -> None:
        while True:
            washload = self.in_queue.get() #A
            print(f"Folder: folding {washload}...")
            time.sleep(1) #B
            print(f"Folder: {washload} done!")
            self.in_queue.task_done() #C

class Pipeline:
    def assemble_laundry_for_washing(self) -> Queue[Washload]:
        washload_count = 8
        washloads_in: Queue[Washload] = Queue(washload_count)
        for washload_num in range(washload_count):
            washloads_in.put(f'Washload #{washload_num}')
        return washloads_in

```

```

def run_concurrently(self) -> None:
    to_be_washed = self.assemble_laundry_for_washing()
    to_be_dried: Queue[Washload] = Queue()
    to_be_folded: Queue[Washload] = Queue()

    Washer(to_be_washed, to_be_dried).start() #D
    Dryer(to_be_dried, to_be_folded).start() #D
    Folder(to_be_folded).start() #D

    to_be_washed.join() #E
    to_be_dried.join() #E
    to_be_folded.join() #E
    print("All done!")

if __name__ == "__main__":
    pipeline = Pipeline()
    pipeline.run_concurrently()

```

We have implemented three main classes: `Washer`, `Dryer` and `Folder`. In this program, each of our functions runs on separate threads concurrently.

The output look like this:

```

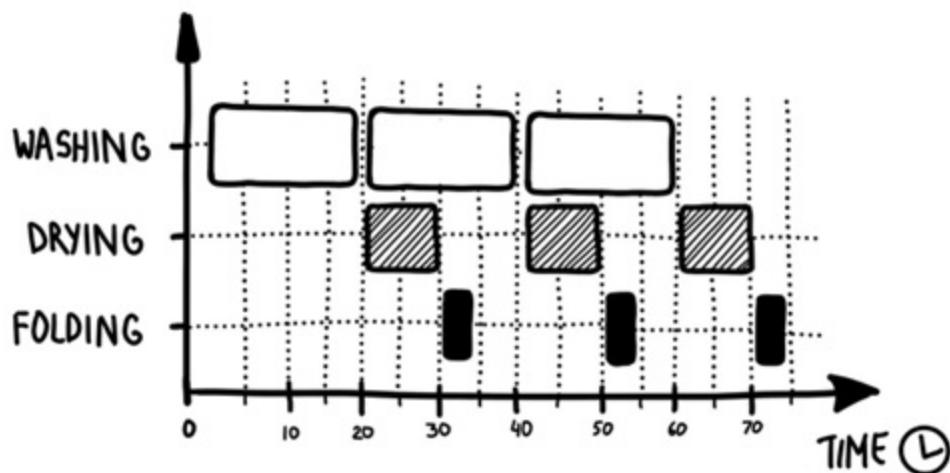
Washer: washing Washload #0...
Washer: washing Washload #1...
Dryer: drying Washload #0...
Folder: folding Washload #0...
Folder: Washload #0 done!
Washer: washing Washload #2...
Dryer: drying Washload #1...
Folder: folding Washload #1...
Folder: Washload #1 done!
Washer: washing Washload #3...
Dryer: drying Washload #2...
Folder: folding Washload #2...
Folder: Washload #2 done!
Dryer: drying Washload #3...
Folder: folding Washload #3...
Folder: Washload #3 done!
All done!

```

Since more loads can be washed at the same time, a pipeline pattern provides more efficiency than washing one load at a time. Suppose it takes three steps to wash the clothes, which take 20, 10 and 5 minutes, respectively. Then, if

all three steps were performed in sequence, you would wash one load of laundry every 35 minutes.

Using a pipeline pattern, you wash the first load in 35 minutes, and then every subsequent load every 20 minutes. Because as soon as the first load finishes washing, the second load goes into the washing phase while the first load dries. Thus, the first load leaves the pipeline 35 minutes after the start of washing, the second load after 55 minutes, the third load after 75 minutes, and so on.



It would seem that pipeline processing can be successfully replaced by simple parallelism. But even in our example, to maintain parallelism, we would need to have four washer and four dryers in the house. I would say that this is simply impossible, if only because of the cost of all this equipment and available space.

The pipeline allows you to limit the number of threads, such as in the thread pool, needed for a particular pipeline step if there are a limited number of shared resources, rather than “wasting” threads that would otherwise be left idle. This is why pipelining is most useful when the number of shared resources is limited.

#### NOTE

For example, file systems can usually handle a limited number of concurrent

read/write requests before they become overloaded. This puts an upper bound on the number of threads that give a concurrency benefit to this step.

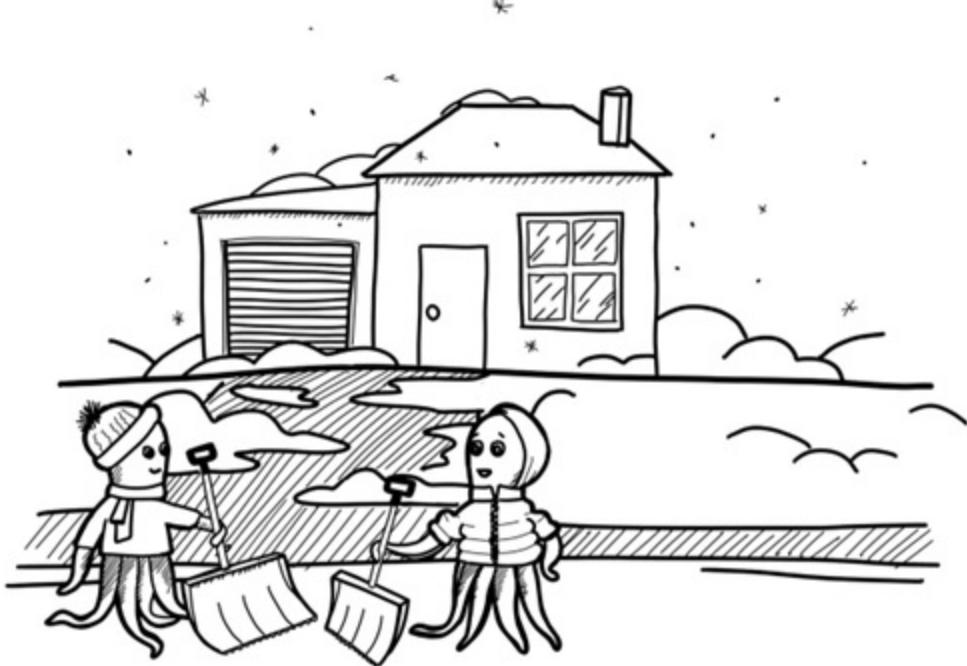
Pipeline processing is often combined with other decomposition approaches such as data decomposition. Speaking of data decomposition...

## 7.3 Data decomposition

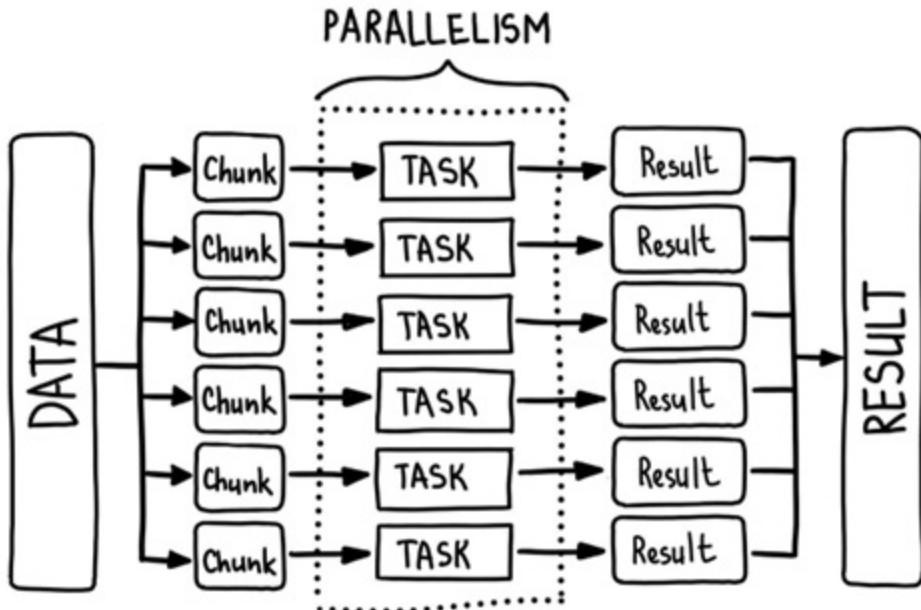
Another commonly used concurrent programming model, *data decomposition*, allows developers to exploit the concurrency that occurs when the same operation is applied to multiple elements of a collection, such as multiplying all the elements of an array by two or increasing the taxes of all citizens with salaries greater than the tax bracket. Each task performs the same set of instructions, but with its own chunk of data.

Therefore, data decomposition answers the question, “How do you decompose task data into chunks that can be processed independently of each other?” Thus, data decomposition is based on the data and not on the type of task.

Let's go back to our shovel problem. We had only one shovel, and the goal is to clear the area around the house of snow. But if we have not one but two shovels, we can divide the area (data) into two zones (chunks of data) and clean them in parallel, using the independence of operations on the different data.



Data decomposition is achieved by dividing the data into *chunks*. Since each operation on each chunk of data can be treated as an independent task, and therefore the resulting concurrent program consists of a sequence of such operations. We already used data decomposition in Chapter 3, in the password cracking example. The possible passwords (data) were divided into independent groups (parts of the task), which were evenly processed on different computing resources.



#### NOTE

Although in this chapter we talk about concurrency at the application layer, it should be noted that data decomposition depends more on the actual parallelism at the hardware layer, because without it, there is little point in using this method.

Data decomposition can be achieved in a distributed system, by dividing the work between several computers or in one computer between different processor cores. Regardless of the amount of input data coming in, we can always horizontally scale resources to increase system performance as it performs the same steps on all available computing resources simultaneously. If this sounds familiar, you are right. It is similar to the SIMD architecture, and this type of architecture is best suited for this category of tasks.

### 7.3.1 Loop-level parallelism

The main candidate for using data decomposition is a program that has an operation that can be executed independently for each chunk of data. In general, loops in any form (for loop, while loop, and for-each loop) often fit this category perfectly and that's why it's also referred to as a special term – *loop-level parallelism*. Loop-level parallelism is an approach often used to

extract concurrent tasks from loops. It can even be used automatically by some compilers that can automatically translate sequential parts of a program into semantically equivalent concurrent code.

Imagine you have the task of creating an application that searches a computer for files that contain some search term. The user enters a directory path and a text string to search, and the program outputs the names of the files containing the search term.

How would you implement such functionality?

If we implement the program in a simple sequential form without using concurrency, it will be a simple for loop:

```
# Chapter 7/find_files/find_files_sequential.py
import os
import time
import typing as T

def search_file(file_location: str, search_string: str) -> bool:
    with open(file_location, "r", encoding="utf8") as file:
        return search_string in file.read()

def search_files_sequentially(file_locations: T.List[str],
                               search_string: str) -> None:
    result = search_file(file_name, search_string)
    if result:
        print(f"Found word in file: '{file_name}'")

if __name__ == "__main__":
    file_locations = list( #A
        glob.glob(f"{os.path.abspath(os.getcwd())}/books/*.txt"))
    search_string = input("What word are you trying to find?: ")

    start_time = time.perf_counter()
    search_files_sequentially(file_locations, search_string)
    process_time = time.perf_counter() - start_time
    print(f"PROCESS TIME: {process_time}")
```

To use this script, enter the directory to search for files when prompted and the word you're searching for. The script will search for the word in all files in the specified directory and print the name of any file that contains the word. Output:

```
What word are you trying to find?: brillig
Found string in file: `Through the Looking-Glass.txt`
PROCESS TIME: 0.75120013574
```

Looking at this code, we see that inside the for loop we do the same actions on different data (files) at each iteration independently of each other – we don't need to finish processing file  $N$  in order to process file  $N + 1$ . So why can't we separate these chunks of data and start processing them in multiple threads instead?

Of course, we can:

```
# Chapter 7/find_files/find_files_concurrent.py
import os
import time
import typing as T
from multiprocessing.pool import ThreadPool

def search_file(file_location: str, search_string: str) -> bool:
    with open(file_location, "r", encoding="utf8") as file:
        return search_string in file.read()

def search_files_concurrently(file_locations: T.List[str],
                               search_string: str) -> None:
    with ThreadPool() as pool: #A
        results = pool.starmap(search_file, #A
                               ((file_location, search_string) for
                                file_location in file_locations))
    for result, file_name in zip(results, file_locations):
        if result:
            print(f"Found string in file: `{file_name}`")

if __name__ == "__main__":
    file_locations = list(
        glob.glob(f"{os.path.abspath(os.getcwd())}/books/*.txt"))
    search_string = input("What word are you trying to find?: ")

    start_time = time.perf_counter()
    search_files_concurrently(file_locations, search_string)
    process_time = time.perf_counter() - start_time
    print(f"PROCESS TIME: {process_time}")
```

This code searches for a specified word in all files in a given directory and its subdirectories using multiple threads. Sample output:

```
Search in which directory?: /Users/kirill/books/
What word are you trying to find?: brillig
Found string in file: `Through the Looking-Glass.txt`
PROCESS TIME: 0.04880058398703113
```

## NOTE

In this example, we want to use all available CPU cores to process multiple files simultaneously. But keep in mind that getting files from the hard disk is an I/O operation, so the data will not be in memory when we start the execution, so we may not get chunks of data processed simultaneously (even with parallel hardware). However, using loop-level parallelism, at least the program can start a useful execution as soon as at least one of the data chunks is read. The execution system can even be single-threaded, it still helps multitasking, where the work completes as soon as it can be executed.

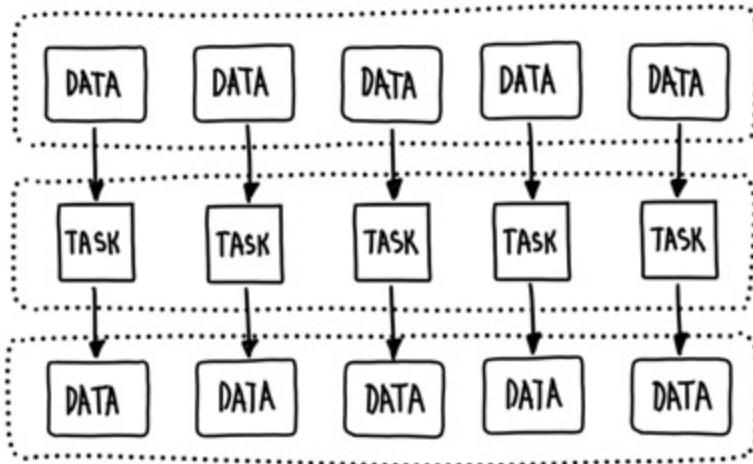
In the above code, threads do the same work, but on different iterations and hence different pieces of data –  $N$  threads can each work on  $1/N$  pieces of data concurrently.

## Map pattern

We have just implemented a new programming pattern – *map pattern*. The idea is based on the technique of functional programming languages. It is used in cases where a single operation is applied to all elements of a collection, but each individual processing is autonomous and have no side effects (they don't change the program state but only convert input data into output data).

It is used to solve embarrassingly parallel tasks: tasks that can be decomposed into independent subtasks that do not require communication/synchronization. Those subtasks are executed on one or more processes, threads, SIMD-tracks or on multiple computers.

## MAP PATTERN



Loops take up a significant portion of execution time in many programs, especially in science or analytical systems, and they can take many forms. To understand whether your problem fits this pattern, you need to do an analysis at or close to the source code level. It is important to understand the dependencies between different iterations of a loop – whether data from the previous iteration is used in subsequent iterations.

### NOTE

There are a lot of libraries and frameworks in the wild that use loop-level parallelism. Open Multi-Processing or OpenMP uses loop-level parallelism for multicore processor architectures. Nvidia's CUDA library provides loop-level parallelism for GPU architectures. Pattern map is widely implemented in majority of modern programming languages, say Scala, Java, Kotlin, Python, Haskell, etc.

As you can see data decomposition is widely used but there is another pattern that is used even wider.

### 7.3.2 Fork/Join pattern

Unfortunately, the application is likely to have sequential parts (those that are not independent and must be executed in a particular order) and concurrent

parts (which can be executed out of order or even in parallel). There is another common concurrency pattern for those types of applications.

Imagine that you're responsible for organizing a vote-counting process for the local mayor elections (please note that this is a fictional scenario, so take it with a grain of salt). The work is simple – just go through the ballots and count the number of votes for one candidate or another.

As it's your first election, you didn't put much thought into organizing the process and decided to do it all by yourself after the polls closed on Election Day. So, you've gone through the pile of blanks sequentially one after another. It took your whole day to finish, but you've made it till the end.

The sequential solution will look similar to the following:

```
# Chapter 7/count_votes/count_votes_sequential.py
import typing as T
import random

Summary = T.Mapping[int, int]

def process_votes(pile: T.List[int]) -> Summary:
    summary = {}
    for vote in pile:
        if vote in summary:
            summary[vote] += 1
        else:
            summary[vote] = 1
    return summary

if __name__ == "__main__":
    num_candidates = 3 #A
    num_voters = 100000 #A
    pile = [random.randint(1, num_candidates) for _ in range(num_
counts = process_votes(pile)
print(f"Total number of votes: {counts}")
```

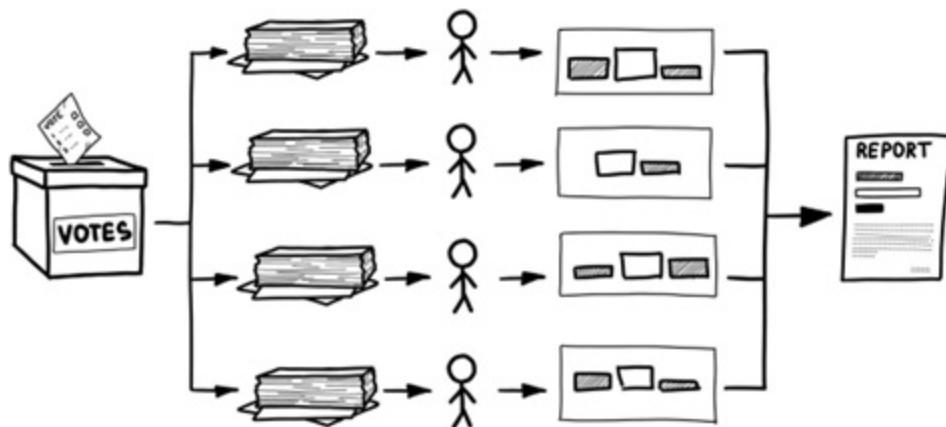
The function takes an array of votes as an argument, each element of which represents a vote for a particular candidate and returns an associative array of the number of votes for each candidate.

As a result of this campaign, you got a promotion to Election Day organizer

of the vote-counting process, not for the local election, but for presidential elections! And as a huge number of votes from different states come your way, you have realized that it's not feasible to use the same sequential approach again.

How would you organize the process in a way that is possible to process a huge number of votes in a limited amount of time?

The obvious way to process the big pile of votes is to split it into several smaller piles and give each pile to a separate staff member to process it in parallel. By distributing the work among multiple people or even groups of people you can easily speed up the process. But that's not it. We need to produce a report with total number of votes for each candidate and not piles of summaries – they should be somehow merged together. So, you decided to organize the process of splitting the votes at the beginning, distribute that work among your staff members and combine their individual results yourself when they are done.



So, to exploit parallel execution, you have decided to hire more staff members whose job will be to count votes. Suppose we have hired 4 staff members then we could do the following:

- Use the first staff member to sum the first 1/4 of the blanks
- Use the second staff member to sum the second 1/4 of the blanks
- Use the third staff member to sum the third 1/4 of the blanks
- Use the fourth staff member to sum the fourth 1/4 of the blanks
- Then you get all 4 results and combine them, returning the answer

The first four steps can be executed in parallel, but the last step will still be sequential as it depends on the results from previous steps.

Before going into the actual code, think of how you would solve the problem yourself.

```
# Chapter 7/count_votes/count_votes_concurrent.py
import typing as T
import random
from multiprocessing.pool import ThreadPool

Summary = T.Mapping[int, int]

def process_votes(pile: T.List[int], worker_count: int = 4) -> Summary:
    vote_count = len(pile) #A
    vote_per_worker = vote_count // worker_count #A
    vote_piles = [pile[i * vote_per_worker:(i + 1) * vote_per_worker] #A
                  for i in range(worker_count)] #A

    with ThreadPool(worker_count) as pool: #A
        worker_summaries = pool.map(process_pile, vote_piles) #A

    total_summary = {} #B
    for worker_summary in worker_summaries: #B
        print(f"Votes from stuff member: {worker_summary}") #B
        for candidate, count in worker_summary.items(): #B
            if candidate in total_summary: #B
                total_summary[candidate] += count #B
            else: #B
                total_summary[candidate] = count #B

    return total_summary #B

def process_pile(pile: T.List[int]) -> Summary:
    summary = {}
    for vote in pile:
        if vote in summary:
            summary[vote] += 1
        else:
            summary[vote] = 1
    return summary

if __name__ == "__main__":
    num_candidates = 3
    num_voters = 100000
    pile = [random.randint(1, num_candidates) for _ in range(num_
```

```

counts = process_votes(pile)
print(f"Total number of votes: {counts}")

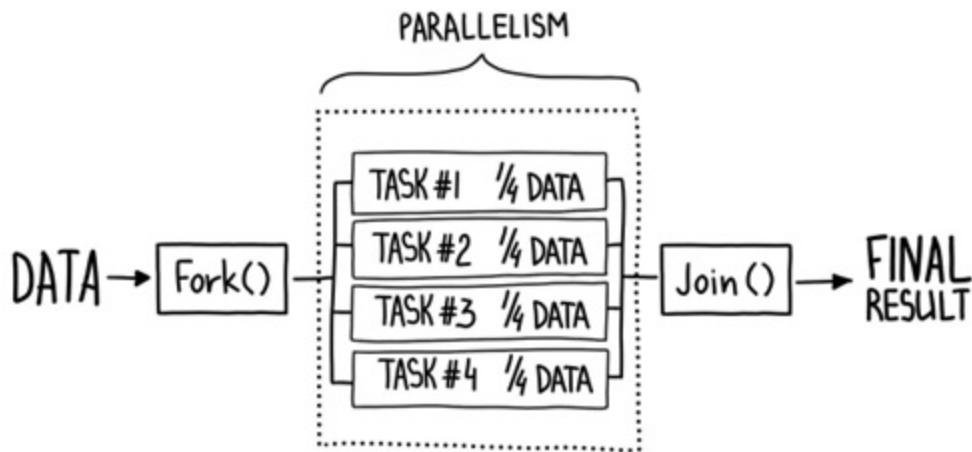
```

This example utilizes a very popular programming pattern for creating concurrent applications. It's called the *fork/join pattern*.

The idea is as follows. We split the data into multiple smaller chunks and process them as independent tasks. In the example above it will be smaller piles of votes divided by some criteria between staff members. This step is called *fork*. As in the loop-level parallelism it also can be scaled horizontally by adding more processing resources.

Then we go through the process of combining the results of individual tasks, until the solution of the original topmost problem is obtained. In the example above we need to aggregate the final election results for each candidate's votes from results of each of our staff member summaries. You can think of it as a synchronization point – on that step we are basically just waiting for all the dependent tasks to be completed before really calculating the final result. This step is called *join*.

Combining two steps together we get the fork/join pattern. The fork/join pattern we studied is one of the most popular nowadays. Many concurrent systems and libraries are written in this style.

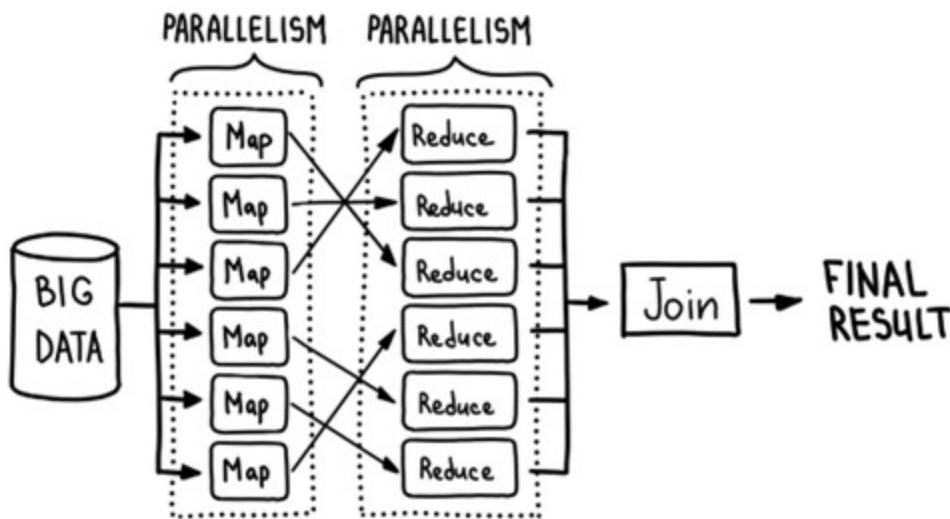


### 7.3.3 Map/Reduce pattern

*Map/reduce* is another concurrency pattern, closely related to fork/join. The

idea of the map step is literally the same as map pattern: one function maps all inputs to get new results (e.g., “multiply by 2”). The reduce step performs an aggregation (e.g. “sum up individual votes” or “take minimum value”). Map and reduce are typically performed in sequence, with map producing intermediate results that are then processed by reduce step.

In map/reduce as in fork/join, a set of input data is processed in parallel by multiple processing resources. The results are then combined until a single response is obtained. Although structurally identical, the type of work performed reflects a slightly different philosophy. The map and reduce steps are more independent than the standard fork/join as it can scale beyond a single computer, utilizing the fleet of machine for performing a single operation on a big volume of data. Another distinction from fork/join pattern is that map step sometimes can be done without reduction, and vice versa.



In fact, this is one of the key concepts behind Google's MapReduce framework of the same name and Yahoo's open-source variant of Apache Hadoop. In these systems, the developer simply writes operations that describe how to map and reduce data. The system then does all the work, often using hundreds or thousands of computers to process gigabytes or terabytes of data. Developer just need to wrap the necessary logic into the computing primitives provided by the framework, leaving everything else to the runtime system.

#### NOTE

There is another currently popular framework that largely inspired by MapReduce's model – Apache Spark. It is a framework that uses functional programming and pipeline processing to provide such support. Instead of writing data to disk for each job as MapReduce does Spark can cache the results across jobs. Moreover, Spark is the underlying framework many very different systems are built upon e.g. Spark SQL & DataFrames, GraphX, Streaming Spark. That makes it easy to mix and match the use of these systems all in the same application. These features make Spark the best fit for iterative jobs and interactive analytics and also help it to provide better performance.

Data decomposition and task decomposition are not mutually exclusive and can be implemented simultaneously by combining them together for the same application. In this way, applications get the maximum boost from the use of concurrency.

## 7.4 Granularity

In the previous voting example, we have made two rather questionable assumptions:

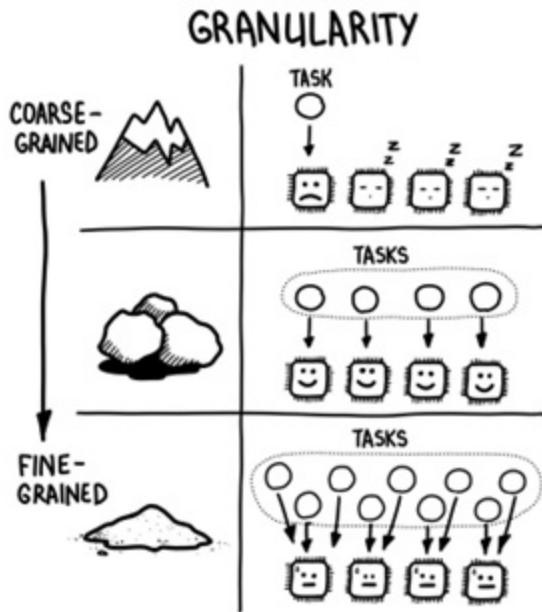
- We assumed that we will have exactly 4 processing resources – our staff members – and that each processing resource gets about the same amount of work. However, limiting the number of processing resources used doesn't make sense. We want concurrent applications to efficiently use all the processing resources available to them. Constantly using exactly 4 threads is not the best approach. If a program runs on a system with 3 cores, it will take a bit longer than if it were evenly distributed among 3 threads. On the contrary, if we have a system with 8 cores, 4 of them will be idle.
- We assumed that every processing resource is exclusively available for our application at runtime. However, our application is not the only one in the system and some processing resources may be needed by other applications or the system itself.

Putting these assumptions aside, the problem arises: How can we use all available resources on the system to perform tasks as efficiently as possible?

Ideally, the number of tasks in decomposed problem should be at least as large as the number of available processing resources, and preferably larger, to provide greater flexibility for runtime system.

The number and size of tasks into which a problem is decomposed determine the *granularity* of the decomposition. Granularity is usually measured by the number of instructions executed in a particular task. For example, in our problem above, dividing the work into 8 threads instead of 4 makes the program *finer-grained* and in this case more flexible. It can be executed on more computing resources, if available. If the system has only 4 cores available, all threads won't execute at the same time because a core can physically execute only one thread at a time. But that's okay as the runtime system will keep track of which threads are waiting their turn, and make sure that all the cores are busy. For example, the scheduler may decide that the first 4 threads will start running in parallel and when they are finished the remaining 4 will be executed. If there are 8 cores available in the system, the system can execute all tasks in parallel.

With *coarse-grained* approach the program is split into larger tasks. As a consequence, a large amount of computation falls on the processors. This can lead to load imbalance, with some tasks processing most of the data and others idle, which limits concurrency in the program. But the advantage of this type of granularity is the lower communication and coordination overhead.



#### 7.4.1 Impact of granularity on performance

When fine granularity is used, the program is broken up into a large number of small tasks. Using fine granularity leads to more parallelism and therefore increases the performance of the system since these tasks are evenly distributed among several processors, so the amount of work associated with a concurrent task is small and executed very quickly.

But creating a large number of small tasks has a downside: by increasing the number of tasks that need to communicate, we significantly increase the cost of communication. To communicate, tasks have to stop computation to send and receive messages. In addition to communication costs, we may have to look at the cost of creating tasks. As we've said before, there are some overhead costs associated with creating threads and processes. If we increase the number of tasks to, say, 1,000,000, this will significantly increase the load on the operating system scheduler and significantly reduce the system performance.

Thus, optimal performance is achieved between the two extremes, fine-grained and coarse-grained.

Many algorithms developed using task decomposition have a fixed number of tasks of the same size and structured local and global connectivity. In such

cases, efficient mapping is straightforward. We map tasks in a way that minimizes inter-processor communication; we can also combine tasks mapped to a single processor, if this has not already been done, to get coarse-grained tasks, one per processor. This process of grouping up tasks is called *agglomeration* (more about it in Chapter 13).

In talking about data decomposition, our effort should be to define as many smaller tasks as possible. This is useful because it forces us to consider a wide range of possibilities for parallel execution. If necessary, tasks are merged into larger tasks – an agglomeration process occurs to improve performance or reduce communication.

In more complex algorithms based on task decomposition, with variable workloads per task and/or unstructured communication schemes, effective agglomeration and matching strategies may not be obvious to the developer. Consequently, we can use load-balancing algorithms that seek to identify efficient agglomeration and mapping strategies, usually using heuristics.

## 7.5 Recap

- There is no magic formula on how to decompose the programming problem. One tool that can help is two visualize the dependencies of the tasks in algorithm by building a *task dependency graph* and find independent tasks in it.
- If you have clear functional components of the application, it may be advantageous to decompose that application into functionally independent tasks using *task decomposition* and use MIMD/MISD systems for the execution. Task decomposition answers the questions: “How can a problem be decomposed into tasks that can execute concurrently?”
  - *Pipeline processing* is one of the popular task decomposition patterns that can help increase throughput of the system when the number of shared resources is limited, and it can be used together with other decomposition approaches.
- If your application has steps that can be performed independently on different data chunks it may be advantageous to utilize *data decomposition* and use SIMD systems for the execution. Data

decomposition answers the question: “How can a problem's data be decomposed into units that can be operated on relatively independently?”

- *Map pattern, fork/join pattern* and *map/reduce pattern* are popular data decomposition patterns that used a lot in many popular libraries and frameworks
- Task number and size determine the *granularity* of the system. Ideally, the number of tasks into which a problem is decomposed should be at least the number of available processing resources, and preferably more, to provide more flexibility for the runtime system

# 8 Solving Concurrency Problems: Race condition & synchronization

## In this chapter

- You learn how to identify and solve one of the most common concurrency problems – race condition
- You learn to share resources between tasks safely and reliably using synchronization primitives

In sequential programs, code execution follows a happy path of predictability and determinism, looking at it and understanding what it does is as easy as understanding how each function works, given the current state of the program. But in concurrent program, the state of the program changes during the execution. External circumstances, such as the operating system scheduler, cache coherency, or platform compilers, can affect the order of execution and resources the program access. In addition, concurrent tasks conflict with each other when they compete for the same resources, such as CPU, shared variables, or files, which often cannot be controlled by the OS. This can all affect the result of the program.

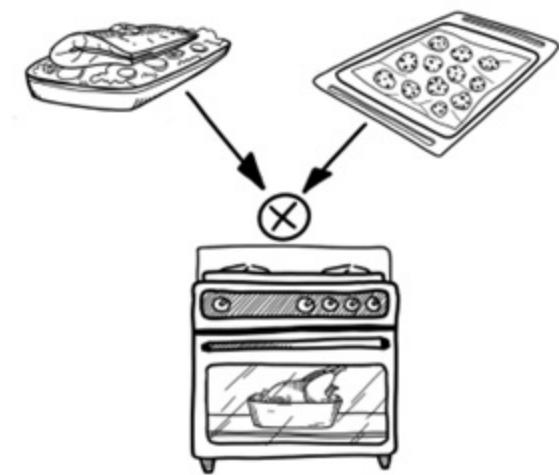
The importance of concurrency control was made clear in 2012 when a major brokerage firm experienced a glitch in their trading system<sup>[1]</sup>. The system was designed to execute trades in a specific order, but due to a race condition, some trades were executed out of order, causing chaos in the market. This led to a significant loss for the firm and disrupted trading for several hours. This incident serves as a reminder that concurrency problems can occur in any software system that involves concurrent execution of multiple tasks.

Hence, we cannot simply rely on the runtime system to manage and coordinate our program's tasks and shared resources, since the detailed requirements and program flow may not be obvious to the it. In this chapter, we will learn how to write code that provides synchronized access to shared resources, look at common concurrency problems one by one and discuss

possible solutions and popular concurrency patterns.

## 8.1 Shared resources

Let's go back to our recipe example. Often the recipe has several steps that can be done at the same time if there are several cooks in the kitchen. But if there is only one oven, you cannot cook turkey along with any other dish at different temperatures at the same time. In our example here, the oven is a shared resource.



In short, multiple cooks provide an opportunity to increase efficiency, but they also make the cooking process much more difficult. Because of required *communication* and *coordination*.

It is the same with programming, the operating system runs tasks concurrently and they also depend on limited resources. These tasks operate independently, often unaware of each other's existence and actions. Consequently, conflicts may arise when they attempt to utilize shared resources during runtime. To prevent such conflicts, it is essential for each task to leave the state of any resource it employs unaffected. For instance, consider a scenario where two tasks concurrently attempt to use a printer. Without proper control over printer access, an error could arise, leading the application (or even whole system) into an unknown and potentially invalid state.

### **8.1.1 Thread safety**

A function or operation is *thread safe* if it behaves correctly when accessed from multiple tasks, regardless of how those tasks are scheduled or interleaved by the execution environment.

When it comes to thread safety, good application design is the best protection a developer could have. Avoiding resource sharing and minimizing communication between tasks makes it less likely that these tasks will mess with each other. However, it is not always possible to create an application that is not using shared resources.

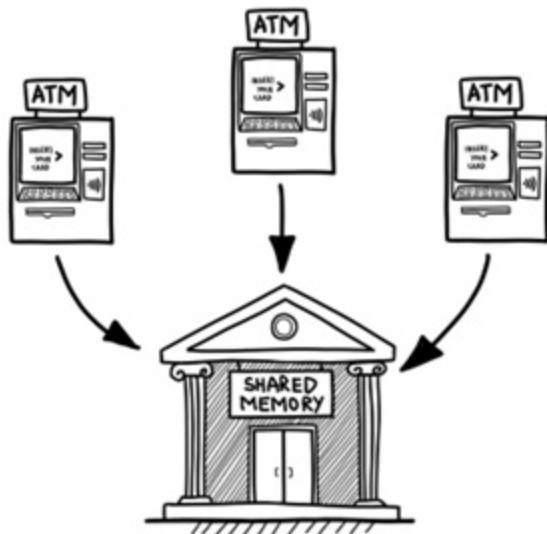
#### **NOTE**

It's easy to provide thread safety by using immutable objects and pure functions. Since they cannot change state, they cannot be corrupted by thread interference or observed in an inconsistent state. Immutability can be provided by the programming language, or application, so that you don't mutate data while multiple threads are using it. These methods will not be covered in this book.

To understand what thread safety is, let's first understand what thread unsafety is, starting, as always, with an example.

## **8.2 Race condition**

Imagine we are writing banking software where there is an object for each bank account. Different tasks (e.g., imagine a teller or an ATM) can deposit or withdraw funds from the same account. Suppose the bank has ATMs that use a shared memory approach, so that all ATMs can read and write the same account objects.



As an example, suppose the bank account class has methods for depositing and withdrawing money from it.

```
# Chapter 8/race_condition/unsynced_bank_account.py
from bank_account import BankAccount

class UnsyncedBankAccount(BankAccount):
    def deposit(self, amount: float) -> None:
        if amount > 0:
            self.balance += amount
        else:
            raise ValueError("You can't deposit negative amount o

    def withdraw(self, amount: float) -> None:
        if 0 < amount <= self.balance:
            self.balance -= amount
        else:
            raise ValueError("Account does not have sufficient fu
```

Here we have a class that implements a bank account with an internal variable `balance` representing the amount of money in the account, and two methods `deposit()` and `withdraw()`, each of which respectively increases or decreases the balance.

Imagine that we have a bunch of ATMs that executes the same transactions concurrently, as we usually assume in the real world. Here's how it would look like in the code:

```

# Chapter 8/race_condition/race_condition.py
import sys
import time
from threading import Thread
import typing as T

from bank_account import BankAccount
from unsynced_bank_account import UnsyncedBankAccount

THREAD_DELAY = 1e-16

class ATM(Thread):
    def __init__(self, bank_account: BankAccount):
        super().__init__()
        self.bank_account = bank_account

    def transaction(self) -> None:
        self.bank_account.deposit(10) #A
        time.sleep(0.001) #A
        self.bank_account.withdraw(10) #A

    def run(self) -> None:
        self.transaction()

def test_atms(account: BankAccount, atm_number: int = 1000) -> No
atms: T.List[ATM] = []
for _ in range(atm_number): #B
    atm = ATM(account) #B
    atms.append(atm) #B
    atm.start() #B

    for atm in atms: #C
        atm.join() #C

if __name__ == "__main__":
    atm_number = 1000
    sys.setswitchinterval(THREAD_DELAY) #D

    account = UnsyncedBankAccount()
    test_atms(account, atm_number=atm_number)

    print("Balance of unsynced account after concurrent transacti
print(f"Actual: {account.balance}\nExpected: 0")

```

We've implemented an ATM as a thread that simply call to deposit method followed by a call to withdraw method with the same amount of money (say \$10). And we run 1000 of them concurrently. So the account balance should

remain the same as we add and remove the same amount of money – we expect it to be zero at the end of the program, right?

But if we run this code, we often find that the balance at the end of the program is different:

Balance of unsynced account after concurrent transactions:

Actual: 380

Expected: 0

How is that possible?

Let's zoom in into how the `deposit` method breaks down into low-level instructions:

get balance (balance=0)

add 10

write back the result (balance=10)

Similar with `withdraw` method:

get balance (balance=10)

remove 10

write back the result (balance=0)

Suppose that two ATMs, let's call them A and B, are concurrently depositing

to a single bank account. In many scenarios, running the two method calls concurrently may not cause problems:

A get balance (balance=0)	
A add 10	
A write back the result (balance=10)	
	B get balance (balance=10)
	B add 10
	B write back the result (balance=20)

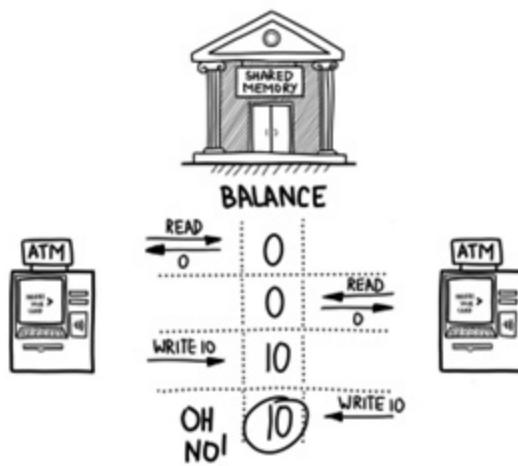
This looks great – we ended up with the correct balance \$20, so both A and B correctly executed their transactions.

But when A and B are executed concurrently, these low-level instructions can interleave with each other, for example like this:

A get balance (balance=0)	
	B get balance (balance=0)
A add 10	

	B add 10
A write back the result (balance=10)	
	B write back the result (balance=10)

In the example above, A and B simultaneously read the balance, calculate different final balances, and then save the new balance, which does not take into account the contribution of the other ATM, one of the deposits will be lost. The balance is now \$10, \$10 deposit has been lost!



The two threads run simultaneously on different processor cores, or the OS scheduler stops one thread and starts the other at any time, switching between them any number of times. If more than one call to the deposit method is executed concurrently, then the balance may end up in an incorrect state. If one thread deposits and another thread withdraws, the exception thrown by the withdrawing thread may depend on the order of operations.

This is an example of a *race condition*. In case of a race condition, tasks access shared resources or common variables that can be used concurrently by other tasks, which leads to the fact that the correctness of the program depends on the relative timing of concurrent operations. When this happens,

we say “One task is in a race with the other tasks.”



There are many reasons for a race condition. Compilers usually perform various optimizations for faster code execution without changing the semantics of the code. If we force compilers to never do interleaving and other code optimizations, it would be very difficult for compilers to be efficient. Similarly, in hardware, there is no single shared memory containing a single copy of all the data in a program. Instead, there are various caches and buffers allowing the processor to access one memory faster than another, as we saw in Chapter 3. As a result, the hardware has to keep track of different copies of the data and move them around. In doing so, memory operations can become "visible" to other threads in a different order than they occurred in the program. As with compilers, requiring the hardware to run all read and write operations in the order they occur is considered too burdensome from a performance standpoint. All these optimizations and reordering is completely hidden from developers, and you never have to worry about it if you just avoid race conditions.

Errors caused by race conditions are very hard to reproduce and isolate. They are a kind of “*heisenbug*”, i.e. a program error that disappears or changes its behavior when you try to investigate it. Because race condition is a semantic bug, it can only be detected at runtime and is difficult to understand just by looking at the code without running the program. So, unfortunately, there is no universal way to detect race conditions. Sometimes placing sleep operators in different code places can help you detect potential race conditions by changing the timing and therefore the order of threads.

#### NOTE

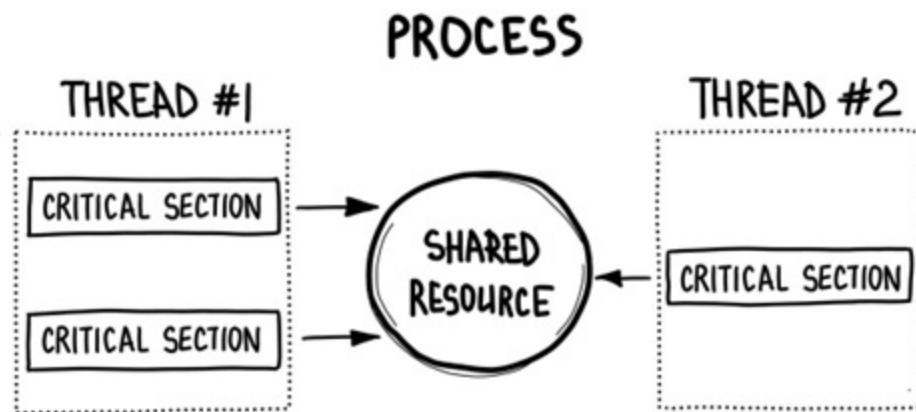
Make sure that the libraries you are using are thread-safe; if they are not, you will have to synchronize library calls. Global variables hidden in a library can prevent even this from happening if the code is not reentrant; if it is, you will have to discard it.

As a result, we need mechanisms to provide synchronized access that prevents multiple tasks from alternating their operations in a way that leads to incorrect results and provides thread security.

## 8.3 Synchronization

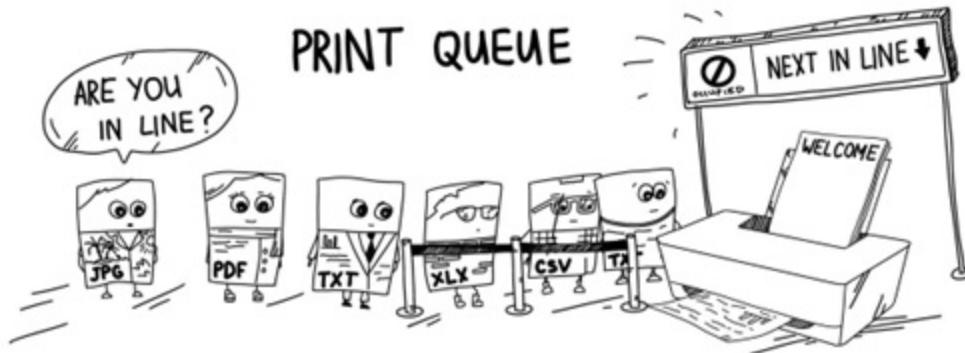
*Synchronization* is one solution to these problems. Synchronization is a mechanism that controls access to shared resources between multiple tasks. This is especially important when multiple tasks require access to resources that cannot be accessed simultaneously. The right synchronization mechanism ensures exclusivity and orderly access to a resource across tasks. In Chapters 2 and 6 we talked about coordination by synchronizing execution points and waiting for dependencies. Developers can also use synchronization to protect a *critical section* of code.

A critical section is a section of code that can be executed simultaneously by multiple tasks and has access to shared resources. For example, inside the critical section developer manipulate particular data structure or use a resource that supports no more than one client at a time like printer.



We cannot simply rely on the OS to understand and enforce this restriction,

since the detailed requirements may not be obvious to the operating system scheduler. For example, in the case of the printer, we want any individual process to have control over the printer as long as it prints the entire file. Otherwise, lines from competing processes will alternate. There must be some kind of mutual exclusion mechanism within the critical section that allows only one task to perform a printing operation at a time.



However, processors have instructions that can be used to implement synchronization. These instructions enable the temporary disabling of interrupts within specific sections of code, ensuring that they cannot be interrupted. This feature proves valuable when safeguarding critical code sections that require uninterrupted execution. While these synchronization instructions find frequent application among compiler and operating system developers, they are also abstracted as library functions in various programming languages. As a result, programmers can utilize these language-specific functions to shield critical code segments, even without directly manipulating the underlying processor-level instructions.

A popular primitive for synchronization called lock, it controls access to critical sections. There are different types of locks with different behavior and semantics.

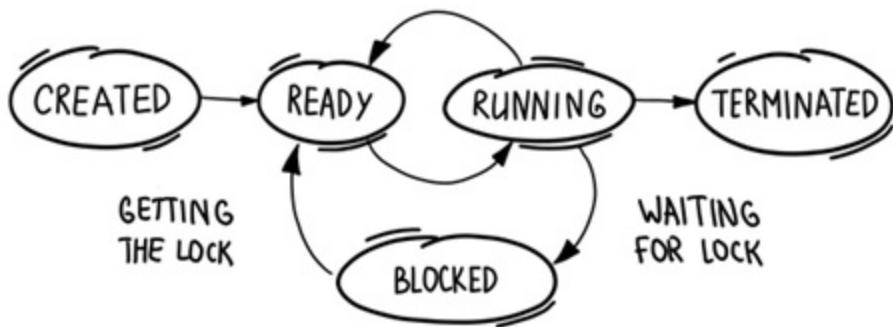
### 8.3.1 Mutual exclusion

The idea behind locks is that a task hangs a “Do not disturb” sign on the resource it is working with before the operation begins and does not remove it until the operation is complete (holding the lock). All other tasks will check for hanging “Do not disturb” sign before trying to hang the sign and perform

the operation themselves. If there is such a sign, the task will be blocked and will wait until the sign is removed to make sure that only it performs the operation, thus avoiding conflicting operations.



We have just introduced another state in which a process or thread can be in – the *Blocked* state. The following illustration shows the thread lifecycle (same applies to process) from creation, readiness and then running, to possible blocking and finally to completion or termination.



To be able to work with the shared resource, a task must first get a lock on it. If another thread already holds the lock, the first thread must wait until the lock is released before it can acquire it, entering a Blocked state until then. This technique is called mutual exclusion, or *mutex* for short, because it ensures that only one task has exclusive access to the shared resource at any given time. There is a concurrency abstraction of the same name in many programming languages and OSs.

Only two states are possible for a mutex – locked and unlocked. The primitive is created in the unlocked state and contains two methods –

acquire and release. The acquire method locks the mutex and block execution until the release method unlocks it. The release method is used to unlock the mutex and can only be called in the locked state. When the release method is called, the mutex is set to the unlocked state and control is immediately returned to the calling thread.

Let's use mutex to solve our money problem. In order for mutex to protect the internal balance variable, blocks of code that work with that variable - critical sections of our program - must be wrapped with calls to the acquire and release methods:

```
# Chapter 8/race_condition/synced_bank_account.py
from threading import Lock
from unsynced_bank_account import UnsyncedBankAccount

class SyncedBankAccount(UnsyncedBankAccount):
    def __init__(self, balance: float = 0):
        super().__init__(balance)
        self.mutex = Lock()

    def deposit(self, amount: float) -> None:
        self.mutex.acquire() #A
        super().deposit(amount)
        self.mutex.release() #B

    def withdraw(self, amount: float) -> None:
        self.mutex.acquire() #A
        super().withdraw(amount)
        self.mutex.release() #B
```

Here we have added mutex to our two methods, so that only one operation of the same type will be performed at a time. This ensures that there is no race condition: deposit() and withdraw(), which read or write the balance, do so while holding the lock. If a thread tries to get a lock that currently belongs to another thread, it will be blocked until the other thread releases the lock. Hence no more than one thread can own a mutex at a time. Therefore, there cannot be simultaneous reading/writing or writing/writing which we see as a result of the execution:

```
Balance of synced account after concurrent transactions:
Actual: 0
Expected: 0
```

Synchronization is only effective when it is used consistently by all threads in the application. If you create a mutex to restrict access to shared resource, all of the threads must receive the same mutex before attempting to manipulate the resource. Failing to do so would compromise the protection provided by the mutex, leading to potential errors.

### 8.3.2 Semaphores

*Semaphore* is another synchronization mechanism that can be used to control access to shared resources, very similar to the mutex. But unlike the mutex, the semaphore can allow several tasks to access the resource at the same time, hence it can be locked and unlocked by multiple tasks while mutex can be locked and unlocked by the same task.

Internally, semaphore holds a counter, which keeps track of how many times it has been acquired or released. As long as the value of the semaphore counter is positive, any task can acquire the semaphore, thus decreasing the value of the counter. If the counter reaches zero, tasks attempting to acquire the semaphore will be blocked and will wait until it becomes available (the counter became positive). When a task finish using a shared resource, it will release the semaphore, which increases the value of the counter. And if there are other threads waiting to acquire the semaphore, they will be told to wake up and do so.

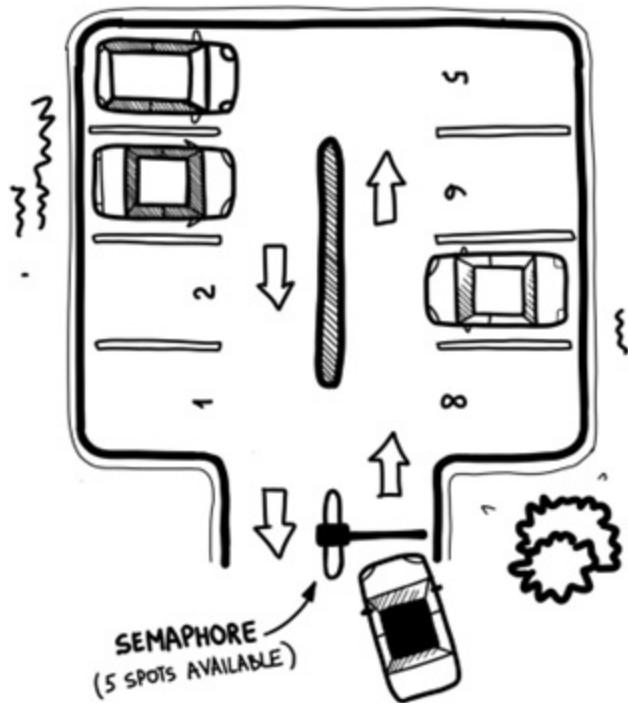
In essence, a mutex can be viewed as a specialized type of semaphore called a binary semaphore. In the case of a mutex, the internal counter can only have two possible values: 0 or 1.

#### Note

The term "semaphore" was coined by computer scientist Edsger Dijkstra in the 1960s, who used the term to describe a synchronization primitive that can be used to signal between threads. The term "semaphore" comes from the use of flags and signal lamps on ships to communicate between them. Later, Dijkstra himself acknowledged that the term "semaphore" was not the best choice for the synchronization primitive he had described, as it was a more general concept that could be used for other purposes beyond signaling.



Let's simulate a public parking garage with a certain number of parking spaces and two entrances using a semaphore. In our parking garage, we have cars that want to enter and leave the garage. A new car cannot enter if it is not guaranteed a parking space, but the car can always leave the garage when it needs to.



To enter the garage, a car needs to get a parking ticket, which corresponds to acquiring a semaphore. If there are available parking spots, the car is assigned one and the semaphore count is decreased. However, when the garage reaches full capacity, the semaphore count drops to zero, thereby preventing any additional cars from entering. Only when the car that currently holds the semaphore releases it, typically after leaving the garage, can another car acquire the semaphore and enter the garage.

```
# Chapter 8/semaphore.py
```

```

import typing as T
import time
import random
from threading import Thread, Semaphore, Lock

TOTAL_SPOTS = 3

class Garage:

    def __init__(self) -> None:
        self.semaphore = Semaphore(TOTAL_SPOTS) #A
        self.cars_lock = Lock() #B
        self.parked_cars: T.List[str] = []

    def count_parked_cars(self) -> int:
        return len(self.parked_cars)

    def enter(self, car_name: str) -> None:
        self.semaphore.acquire() #A
        self.cars_lock.acquire() #B
        self.parked_cars.append(car_name)
        print(f"{car_name} parked")
        self.cars_lock.release() #B

    def exit(self, car_name: str) -> None:
        self.cars_lock.acquire()#B
        self.parked_cars.remove(car_name)
        print(f"{car_name} leaving")
        self.semaphore.release() #C
        self.cars_lock.release()#B

```

In the above code, we have used both mutex and semaphore! Although they are very similar in their properties, in our code they are used for different purposes. We use mutex to coordinate access to an internal variable, which is a list of parked cars. Semaphore is used to coordinate the entry and exit methods of the parking garage to limit the number of cars by available parking spots. In our case, it's only three spots.

If the semaphore is unavailable (because its value is zero), the car will wait until a parking space is available, and the semaphore is released. As soon as the car thread acquires the semaphore, it prints the message that it is parked and then goes to sleep for a random period of time. The car thread then prints the message that it is leaving and releases the semaphore, increasing its value so that another waiting thread can acquire it.

Let's simulate a busy day of such parking garage:

```
# Chapter 8/semaphore.py
def park_car(garage: Garage, car_name: str) -> None:
    garage.enter(car_name) #A
    time.sleep(random.uniform(1, 2)) #A
    garage.exit(car_name) #A

def test_garage(garage: Garage, number_of_cars: int = 10) -> None
    threads = []
    for car_num in range(number_of_cars): #B
        t = Thread(target=park_car, #B
                    args=(garage, f"Car #{car_num}")) #B
        threads.append(t) #B
        t.start() #B

    for thread in threads:
        thread.join()

if __name__ == "__main__":
    number_of_cars = 10
    garage = Garage()
    test_garage(garage, number_of_cars) #C

    print("Number of parked car after a busy day:")
    print(f"Actual: {garage.count_parked_cars()}\nExpected: 0")
```

Just as with mutex we get the expected result:

```
Car #0 parked
Car #1 parked
Car #2 parked
Car #0 leaving
Car #3 parked
Car #1 leaving
Car #4 parked
Car #2 leaving
Car #5 parked
Car #4 leaving
Car #6 parked
Car #5 leaving
Car #7 parked
Car #3 leaving
Car #8 parked
Car #7 leaving
Car #9 parked
Car #6 leaving
```

```
Car #8 leaving
Car #9 leaving
Number of parked car after a busy day:
Actual: 0
Expected: 0
```

Another way to solve the synchronization problems would be to create more powerful operations that will be executed in one step and thus eliminate the possibility of undesired interrupts. Such operations exist and are called atomic operations.

### 8.3.3 Atomic operations

*Atomic operations* are the simplest form of synchronization that works with primitive data types. Atomic means that no other thread will be able to see the operation in a partially completed state.

For certain simple operations, such as incrementing a counter variable, atomic operations can offer significant performance benefits compared to traditional locking mechanisms. Instead of acquiring a lock, modifying the variable, and then releasing the lock, atomic operations provide a more streamlined approach. Consider an example using assembly code:

```
add 0x9082a1b, $0x1
```

Here the assembly instruction adds the value 1 to the memory location specified by the address 0x9082a1b. The hardware guarantees that this operation executes atomically, without any interruption. When an interruption occurs, the operation either does not execute at all or executes to the end; there is no intermediate state.

The advantage of atomic operations is that they do not block competing tasks. This can potentially maximize concurrency and minimize synchronization costs. But these operations depend on special hardware instructions, and with good communication between hardware and software, guarantees of atomicity at the hardware level can be extended to the software level.

#### NOTE

Most programming languages provide some atomic data structures, but you must be very careful because not all data structures are. For example, some Java collections are thread-safe and in addition in Java there are several non-blocking atomic data structures, such as `AtomicBoolean`, `AtomicInteger`, `AtomicLong` and `AtomicReference`. Another example, in C++, the standard library provides atomic types such as `std::atomic_int` and `std::atomic_bool`.

But not all operations are atomic so one should not rely on them. When writing concurrent applications, there is a long tradition of pretending that we don't know anything more than what the programming language standards tell us. When atomic operations are not available, use locks.

With this knowledge of synchronization in mind, let's look at some other popular concurrency problems.

## 8.4 Recap

- When using *shared resources*, typical for concurrent programs, the developer must be careful to avoid concurrent accesses to shared resource, since any task can be interrupted mid-execution. Those issues can lead to unexpected behavior and subtle bugs that don't show up until much later.
- A *critical section* is a section of code that can be executed concurrently by multiple tasks and has access to shared resources. To ensure exclusive use of critical sections some synchronization mechanism is required.
- The simplest method to prevent unexpected inside the critical section is using *atomic operations*. “Atomic” means that no other thread will be able to see the operation in a partially completed state. But these operations rely on the environment (hardware and runtime environment support).
- Another method of synchronization and the most common one is using *locks*. A lock is an abstract concept. The basic premise is that a lock protects access to a shared resource. If you own a lock, then you can access the protected shared resource. If you do not own the lock, then you cannot access the shared resource.

- Tasks may require mutually exclusive operations, which can be protected by mutually exclusive locks or *mutexes* to prevent reading shared data in one task and updating it in another.
- *Semaphore* is another lock that can be used to control access to shared resources, very similar to the mutex. But unlike the mutex, the semaphore can allow several tasks to access the resource at the same time, hence it can be locked and unlocked by multiple tasks while mutex can be locked and unlocked by the same task.
- Synchronization is expensive. Therefore, if possible, try to design without synchronization of any type.
- When two tasks access and manipulate the shared resource concurrently, and the resulting execution outcome depends on the order in which processes access the resource; this is called a *race condition*. When this happens, we say “One thread is in a race with the others”. It can be avoided by properly synchronizing threads in critical sections using techniques such as locks, atomic operations, or switching to a message passing IPC.

[\[1\]](#) The brokerage firm, Knight Capital Group, developed a high-frequency trading system to execute trades quickly. In August 2012, they deployed a new software upgrade intended to improve the system's speed and efficiency. Due to a race condition, some trades were executed out of order, causing the system to buy high and sell low, leading to a loss of over \$460 million in just 45 minutes. The incident affected 150 stocks and resulted in an SEC investigation and a \$12 million fine for Knight Capital.

# 9 Solving Concurrency Problems: Deadlock & starvation

## In this chapter

- You learn how to identify and solve common concurrency problems (deadlock, livelock, starvation)
- You learn popular concurrency design patterns: producer-consumer pattern and readers-writers pattern

In the previous chapter, we explored the challenges that arise in concurrent programming, such as race conditions, and the synchronization primitives used to address them. In this chapter, we will now focus on another set of common concurrency problems: deadlock, livelock, and starvation.

These issues can lead to extremely serious consequences, given that concurrency is used in all sorts of technology that we quite literally entrust our lives to. Two Boeing 737 Max airplanes crashed in 2018 and 2019 due to a software error caused by a concurrency problem. The airplanes' Maneuvering Characteristics Augmentation System (MCAS) was designed to prevent the airplane from stalling, but a race condition caused it to malfunction, leading to several fatal crashes that killed a total of 347 people. A decade earlier, Toyota vehicles experienced sudden, unintended acceleration issues in 2009 and 2010. The issue was linked to a software error that caused a concurrency problem in the electronic throttle control system. The error caused the throttle to open unexpectedly, leading to several accidents and fatalities.

In this chapter, we will explore how to identify and solve these common concurrency problems, providing you with the knowledge and tools to address them effectively. By the end of this chapter, you will have a comprehensive understanding of common concurrency problems and popular concurrency patterns, including the producer-consumer and readers-writers patterns, enabling you to implement appropriate solutions to avoid potential

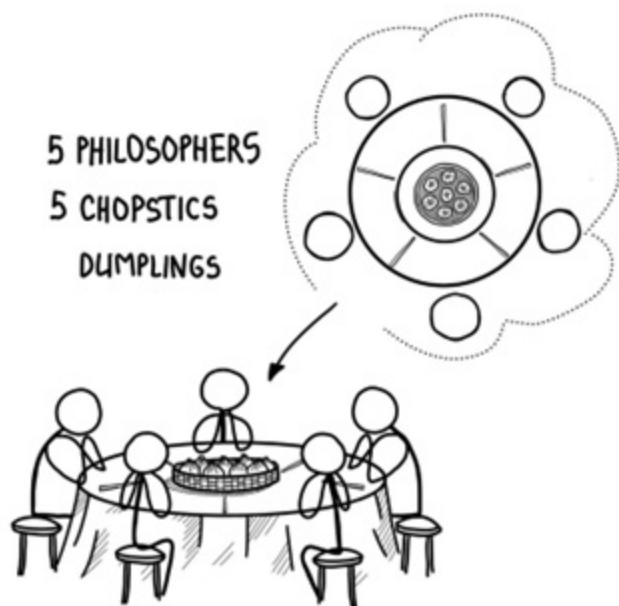
disasters.

## 9.1 Dining Philosophers

Locks (mutexes and semaphores) are super tricky to use. Incorrect use of locks can break an application when the locks acquired are not released or the locks that need to be acquired never become available. A classic example used to illustrate synchronization problems, when several tasks compete for locks, is the problem of philosophers having lunch, which was formulated by famous computer scientist Edsger Dijkstra in 1965. This example is a standard test case for evaluating synchronization approaches.

Five silent philosophers sit at a round table with a plate full of dumplings. Between each pair of neighboring philosophers lies a chopstick. And the philosophers do what philosophers do best – think and eat.

Only one philosopher can hold the chopstick, so the philosopher can only use the chopstick if no other philosopher is using it. After the philosopher has finished eating, he must put both chopsticks so that they are available to the others. The philosopher can only take the chopstick to his right or left and only when they are available and cannot start eating without taking both chopsticks.



The problem is how to design a ritual (algorithm) so that every philosopher can forever keep alternating between eating and thinking, assuming that no philosopher can know when others want to eat or think – making it a concurrent system.

The act of taking dumplings from the plate is a critical section, so we developed a mutual exclusion process to protect it, using two chopsticks as mutexes. Thus, when a philosopher wants to bite into a dumpling, he will first take the chopstick on the left, if there is one available, and put a lock on it. Then he will take the right chopstick, if there is one available, and put the lock on it as well. Now he has both chopsticks – he is in the critical section, so he bites off the dumpling. Then he puts the right chopstick to unlock it, and then the left chopstick. Finally, because he is a philosopher, he will go back to philosophizing.

In the code it will look like this:

```
# Chapter 9/deadlock/deadlock.py
import time
from threading import Thread

from lock_with_name import LockWithName

THREAD_DELAY = 0.1
dumplings = 20

class Philosopher(Thread):
    def __init__(self, name: str, left_chopstick: LockWithName,
                 right_chopstick: LockWithName):
        super().__init__()
        self.name = name
        self.left_chopstick = left_chopstick #A
        self.right_chopstick = right_chopstick #A

    def run(self) -> None:
        global dumplings

        while dumplings > 0: #B
            self.left_chopstick.acquire() #C
            print(f"{self.left_chopstick.name} grabbed by {self.n
                  f"now needs {self.right_chopstick.name}}")
            self.right_chopstick.acquire() #D
            print(f"{self.right_chopstick.name} grabbed by {self.
```

```

dumplings -= 1 #E
print(f"{self.name} eats a dumpling. "
      f"Dumplings left: {dumplings}")
self.right_chopstick.release() #F
print(f"{self.right_chopstick.name} released by {self."
self.left_chopstick.release() #G
print(f"{self.left_chopstick.name} released by {self."
print(f"{self.name} is thinking...")
time.sleep(0.1)

```

In the above code, the `Philosopher` thread represents a single philosopher, which contains the name of the philosopher and two mutexes named `left_chopstick` and `right_chopstick` to specify the order in which the philosopher will acquire them.

We also have a shared variable `dumplings` to represent the remaining amount of dumplings on the shared plate. The while loop will make philosophers keep taking dumplings as long as there is some left on the plate. As part of the loop, a philosopher will take and acquire a lock on his left chopstick, then on his right chopstick. Then, if there are still dumplings left on the plate, he will take a piece, decreasing the `dumplings` variable, and display a message about how many dumplings left.

As philosophers, they will keep alternating between eating and thinking, but because they operate as concurrent tasks, neither of them knows when the other wants to eat or think, and this can lead to problems. Let's look at some of the problems that might arise when running this code, as well as possible solutions.

## 9.2 Deadlock

To simplify explanations in this section, we decreased the number of philosophers to two, with preserving the original algorithm.

```

# Chapter 9/deadlock/deadlock.py
if __name__ == "__main__":
    chopstick_a = LockWithName("chopstick_a")
    chopstick_b = LockWithName("chopstick_b")

    philosopher_1 = Philosopher("Philosopher #1", chopstick_a, ch
    philosopher_2 = Philosopher("Philosopher #2", chopstick_b, ch

```

```
philosopher_1.start()
philosopher_2.start()
```

When we run this program, we see similar to the following:

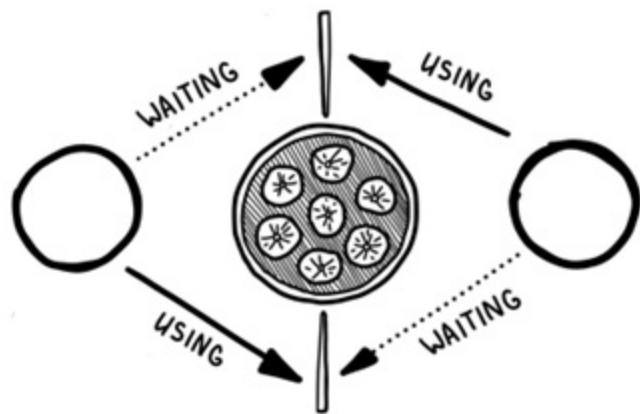
```
Philosopher #1 eat a dumpling. Dumplings left: 19
Philosopher #1 eat a dumpling. Dumplings left: 18
Philosopher #2 eat a dumpling. Dumplings left: 17
...
Philosopher #2 eat a dumpling. Dumplings left: 9
```

The program doesn't finish – it's stuck, and the dumplings are still on the plate, what's going on?

If the first philosopher gets hungry and takes the chopstick A. And the second philosopher at the same time will also get hungry and take the chopstick B. They both got one of the two locks they needed, but they are both stuck waiting for the other thread to release the remaining lock.

This is an example of a situation called *deadlock*. During deadlock several tasks are waiting for resources occupied by each other, and none of them can continue execution. The program is now stuck in this state forever, so it is necessary to manually terminate its execution. Running the same program again will result in a deadlock after a different number of dumplings. The exact number at which the philosophers get stuck will depend on how the tasks are scheduled by the system.

# DEADLOCK



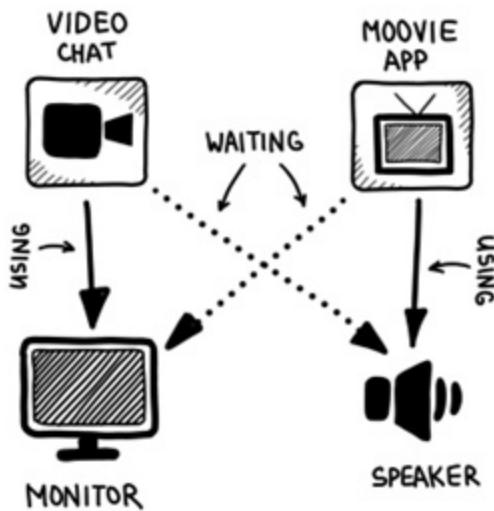
As with the race condition, you may be lucky enough to never run into this problem in your application. However, even if the potential for deadlocks exists, they should definitely be avoided. Every time a task tries to get more than one lock at a time, there is a possibility of a deadlock. Avoiding deadlocks is a common problem for concurrent programs that use mutual exclusion mechanisms to protect critical sections of code.

## NOTE

Never assume a specific order of execution. When there are multiple threads, as we have seen, the execution order is nondeterministic. If you care about the execution order of one thread relative to another, you will have to apply synchronization. But for best performance, you want to avoid synchronization as much as possible. In particular, you want highly detailed tasks that do not require synchronization; this will allow your cores to work as fast as possible on each task assigned to them.

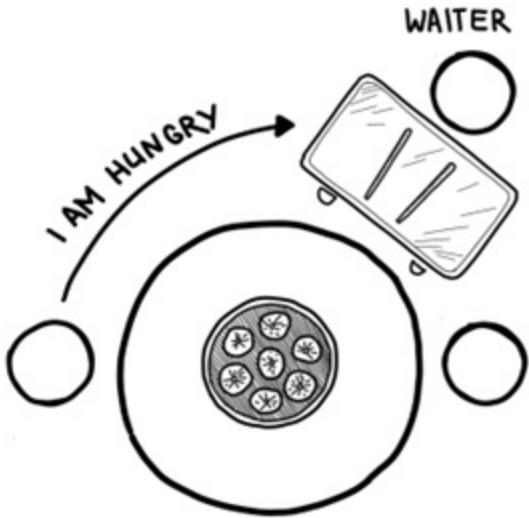
To show a more realistic example for a moment (it's not every day we feed philosophers dumplings), let's imagine a real system – your home computer with two applications installed, such as video chat (like Zoom or Skype) and a movie watching application (like Netflix or YouTube). The two programs serve different functions – one lets you chat with co-workers or friends, and the other lets you watch cool movies, but both of them access the same subsystems of your computer – like screen and audio. Imagine when they

both want access to the screen and audio. They make their requests at the same time, and the OS gives the screen to the movie app and the audio to the video chat. Both programs will block the resource they have and then wait for the remaining resource to become available. As they wait for each other, they will wait forever, just like our poor philosophers! The deadlock is permanent unless the OS takes some drastic action, such as killing one or more processes or forcing one or more processes to backtrack.



### 9.2.1 Arbitrator solution

To avoid a deadlock, we can make sure that each philosopher can either take both chopsticks or none, the easiest way to achieve this is to introduce an *arbitrator* – someone in charge of the chopsticks, such as a waiter. To take a chopstick, the philosopher must ask the waiter for permission to grab it first. The waiter only gives permission to one philosopher at a time until he takes both chopsticks. Putting down the chopstick is allowed at all times.



The waiter can be implemented with another lock:

```
# Chapter 9/deadlock/deadlock_arbitrator.py
import time
from threading import Thread, Lock

from lock_with_name import LockWithName

THREAD_DELAY = 0.1
dumplings = 20

class Waiter:
    def __init__(self) -> None:
        self.mutex = Lock()

    def ask_for_chopsticks(self, left_chopstick: LockWithName,
                           right_chopstick: LockWithName) -> None
        with self.mutex: #A
            left_chopstick.acquire() #B
            print(f"{left_chopstick.name} grabbed")
            right_chopstick.acquire() #B
            print(f"{right_chopstick.name} grabbed")

    def release_chopsticks(self, left_chopstick: LockWithName,
                           right_chopstick: LockWithName) -> None
        right_chopstick.release() #B
        print(f"{right_chopstick.name} released")
        left_chopstick.release() #B
        print(f"{left_chopstick.name} released\n")
```

And we can use it as a lock:

```
# Chapter 9/deadlock/deadlock_arbitrator.py
class Philosopher(Thread):
    def __init__(self, name: str, waiter: Waiter,
                 left_chopstick: LockWithName,
                 right_chopstick: LockWithName):
        super().__init__()
        self.name = name
        self.left_chopstick = left_chopstick
        self.right_chopstick = right_chopstick
        self.waiter = waiter

    def run(self) -> None:
        global dumplings

        while dumplings > 0:
            print(f"{self.name} asks waiter for chopsticks")
            self.waiter.ask_for_chopsticks( #A
                self.left_chopstick, self.right_chopstick) #A

            dumplings -= 1
            print(f"{self.name} eats a dumpling. "
                  f"Dumplings left: {dumplings}")
            print(f"{self.name} returns chopsticks to waiter")
            self.waiter.release_chopsticks( #B
                self.left_chopstick, self.right_chopstick) #B
            time.sleep(0.1)

if __name__ == "__main__":
    chopstick_a = LockWithName("chopstick_a")
    chopstick_b = LockWithName("chopstick_b")

    waiter = Waiter()
    philosopher_1 = Philosopher("Philosopher #1", waiter, chopstick_b)
    philosopher_2 = Philosopher("Philosopher #2", waiter, chopstick_a)

    philosopher_1.start()
    philosopher_2.start()
```

Because of introducing a new central entity – the waiter, this approach can lead to limited concurrency: if a philosopher eats and one of his neighbors' requests chopsticks, all other philosophers must wait until this request is fulfilled, even if chopsticks are still available to them. In a real computer

system, the arbitrator will do much the same thing, controlling access by the worker threads to ensure that access is performed in an orderly fashion.

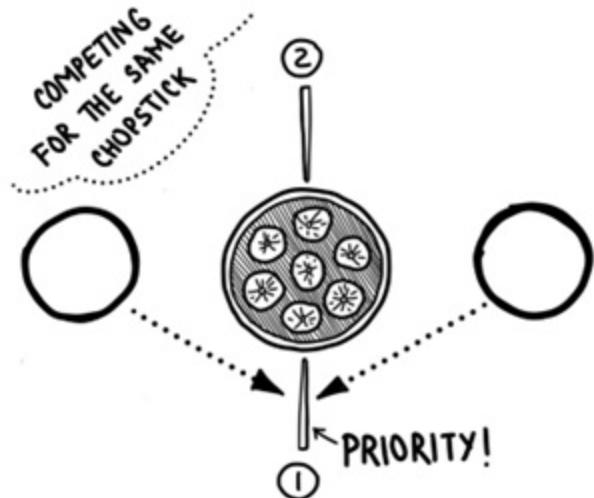
The above solution reduces concurrency – we can do better than that.

### 9.2.2 Resource hierarchy solution

What if we set priorities in these locks so that philosophers try to take the same chopstick first? This way they won't have a deadlock problem, because they will be competing for the same first lock.

All philosophers must agree that of the two chopsticks they plan to use, the chopstick with the highest priority will always be taken first. In our case, if both philosophers take the chopstick with the highest priority simultaneously, only the chopstick with the lowest priority will remain on the table, so the second philosopher cannot take any chopstick. Moreover, only one philosopher will have access to the chopstick with the lowest priority, so he can start eating with two chopsticks. Genius!

Let's set priorities for our chopsticks. We'll say that chopstick A has the highest priority, chopstick B the second highest. And each philosopher should always get the chopstick with the highest priority first.



In our code, Philosopher #2 creates a problem because he acquires a chopstick B before A. To fix that, we will change their order without

changing any other code. First, we acquire the A chopstick and then the B.

```
# Chapter 9/deadlock/deadlock_hierarchy.py
from lock_with_name import LockWithName

from deadlock import Philosopher

if __name__ == "__main__":
    chopstick_a = LockWithName("chopstick_a")
    chopstick_b = LockWithName("chopstick_b")

    philosopher_1 = Philosopher("Philosopher #1", chopstick_a, ch
    philosopher_2 = Philosopher("Philosopher #2", chopstick_a, ch

    philosopher_1.start()
    philosopher_2.start()
```

Now, when we run the program after making this change, it runs to the end without any deadlocks.

#### NOTE

Ordering locks is not always possible if a task does not know all the locks it needs to acquire beforehand. Deadlock avoidance mechanisms like resource allocation graphs (RAG) or lock hierarchies can be used to prevent deadlocks. RAG helps detect and prevent cycles in the relationships between processes and resources. Higher-level synchronization primitives in some programming languages and frameworks simplify lock management. However, careful design and testing are still necessary as these techniques do not guarantee complete elimination of deadlocks.

Another method of preventing deadlocks is to set a timeout on blocking attempts. If the task cannot successfully get all the locks it needs within a certain amount of time, we force the thread to release all the locks that it currently holds. But that may cause another problem – livelock.

## 9.3 Livelock

*Livelock* is similar to deadlock and occurs when two tasks are competing for the same set of resources, but in livelock, a task gives up its first lock in an

attempt to get a second lock. After getting the second lock, it comes back and tries to get the first lock again. The task is now in the same blocked state because it spends all of its time releasing one lock and trying to get another, instead of doing the actual work.

Imagine you are making a phone call, but the person on the other end is also trying to call you. You both hang up and try again at the same time, which again creates the same situation. In the end, no one can get through to anyone.



Livelock is when tasks that are actively performing concurrent tasks, but these tasks have no effect on moving the state of the program forward. It is similar to deadlock, but the difference is that the tasks become “polite” and let others do their work first.

Let's imagine that our philosophers have become a little more polite than they were – they can give up a chopstick if they can't get both:

```
# Chapter 9/livelock.py
import time
from threading import Thread

from deadlock.lock_with_name import LockWithName

dumplings = 20

class Philosopher(Thread):
    def __init__(self, name: str, left_chopstick: LockWithName,
```

```

        right_chopstick: LockWithName):
super().__init__()
self.name = name
self.left_chopstick = left_chopstick
self.right_chopstick = right_chopstick

def run(self) -> None:
    global dumplings

    while dumplings > 0:
        self.left_chopstick.acquire() #A
        print(f"{self.left_chopstick.name} chopstick "
              f"grabbed by {self.name}")
        if self.right_chopstick.locked(): #B
            print(f"{self.name} cannot get the " #B
                  f"{self.right_chopstick.name} chopstick, "
                  f"politely concedes...") #B
        else: #B
            self.right_chopstick.acquire() #B
            print(f"{self.right_chopstick.name} chopstick " #
                  f"grabbed by {self.name}") #B
            dumplings -= 1 #B
            print(f"{self.name} eat a dumpling. Dumplings " #
                  f"left: {dumplings}") #B
            time.sleep(1) #B
            self.right_chopstick.release() #B
            self.left_chopstick.release() #B

if __name__ == "__main__":
    chopstick_a = LockWithName("chopstick_a")
    chopstick_b = LockWithName("chopstick_b")

    philosopher_1 = Philosopher("Philosopher #1", chopstick_a, ch
    philosopher_2 = Philosopher("Philosopher #2", chopstick_b, ch

    philosopher_1.start()
    philosopher_2.start()

```

Unfortunately, these nice people are not destined to eat:

```

chopstick_a chopstick grabbed by Philosopher # 1
Philosopher # 1 cannot get the chopstick_b chopstick, politely co
chopstick_b chopstick grabbed by Philosopher # 2
Philosopher # 2 cannot get the chopstick_a chopstick, politely co
chopstick_b chopstick grabbed by Philosopher # 2
chopstick_a chopstick grabbed by Philosopher # 1
Philosopher # 2 cannot get the chopstick_a chopstick, politely co

```

```
Philosopher # 1 cannot get the chopstick_b chopstick, politely co
chopstick_b chopstick grabbed by Philosopher # 2
chopstick_a chopstick grabbed by Philosopher # 1
Philosopher # 2 cannot get the chopstick_a chopstick, politely co
Philosopher # 1 cannot get the chopstick_b chopstick, politely co
```

In addition to zero work done, this approach can lead to system overloads with frequent context switching, which reduces the overall system performance. In addition, the OS scheduler has no way to implement fairness because it does not know which task has been waiting the longest for the shared resource.

To avoid this type of locking, order the locking sequence hierarchically as we did with resolving a deadlock. This way, only one process can block both locks successfully.

#### NOTE

Detecting and resolving livelock is often more challenging than deadlock because livelock scenarios involve complex and dynamic interactions among multiple entities, making them harder to identify and resolve.

Livelock is a subset of a broader set of problems called *Starvation*.

## 9.4 Starvation

Let's add a local variable to keep track of how many dumplings each Philosopher thread ate.

```
# Chapter 9/starvation.py
from threading import Thread

from deadlock.lock_with_name import LockWithName

dumplings = 1000

class Philosopher(Thread):
    def __init__(self, name: str, left_chopstick: LockWithName,
                 right_chopstick: LockWithName):
        super().__init__()
        self.name = name
```

```

        self.left_chopstick = left_chopstick
        self.right_chopstick = right_chopstick

    def run(self) -> None:
        global dumplings

        dumplings_eaten = 0 #A
        while dumplings > 0:
            self.left_chopstick.acquire()
            self.right_chopstick.acquire()
            if dumplings > 0:
                dumplings -= 1
                dumplings_eaten += 1 #A
                time.sleep(1e-16)
            self.right_chopstick.release()
            self.left_chopstick.release()
        print(f"{self.name} took {dumplings_eaten} pieces")

if __name__ == "__main__":
    chopstick_a = LockWithName("chopstick_a")
    chopstick_b = LockWithName("chopstick_b")

    threads = []
    for i in range(10):
        threads.append(
            Philosopher(f"Philosopher #{i}", chopstick_a, chopstick_b))

    for thread in threads:
        thread.start()

    for thread in threads:
        thread.join()

```

We've named the variable `dumplings_eaten` and initialized it with zero. We will increment it each time a philosopher eats a dumpling. When the program finishes, we see that each of the philosophers have eaten a different number of dumplings, and that's not really fair:

```

Philosopher #1 took 417 pieces
Philosopher #9 took 0 pieces
Philosopher #6 took 0 pieces
Philosopher #7 took 0 pieces
Philosopher #5 took 0 pieces
Philosopher #0 took 4 pieces
Philosopher #2 took 3 pieces
Philosopher #8 took 268 pieces

```

Philosopher #3 took 308 pieces  
Philosopher #4 took 0 pieces

Philosopher #1 took a lot more dumplings than Philosopher #8. He got more than 400 pieces. Seem like Philosopher #8 is sometimes slow to take a chopstick, and Philosopher #1 thinks quickly and takes the chopstick back, while Philosopher #8 is again stuck waiting. Some philosophers never take both chopsticks. If it happens once in a while, it's probably okay, but if it happens regularly then the thread will *starve*.

*Starvation* is exactly what it sounds like, a thread is quite literally “starved” never gaining access to required resources and in turn no progress is made. If another greedy task often holds a lock on a shared resource, the starving task will not get a chance to execute.

#### NOTE

Starvation is one of the basic ideas behind the most famous attacks against online services, Denial Of Service Attacks (DOS). In this attack, the attacker tries to deplete all the resources the server has. The service starts to run out of available resources (storage, memory, or computing resources), crashes, and cannot provide its services.

Starvation is usually caused by an oversimplified scheduling algorithm. The scheduling algorithm as we learned from Chapter 6 is part of the runtime system, it should distribute resources equally among all the tasks; that is, the scheduler should distribute resources in such a way that no task is constantly blocked from accessing the resources it needs to complete its work. The treatment of different task priorities depends on the operating system, but usually tasks with a higher priority are scheduled to run more often, and this can cause low-priority tasks to starve. Another thing that can lead to starvation is too many tasks in the system, where it takes a long time before a task starts the execution.

A possible solution to starvation is to use a scheduling algorithm with priority queuing, which also uses the aging technique. *Aging* is a technique of gradually increasing the priority of threads that have been waiting in the system for a long time. Eventually, the thread will reach a high enough

priority that it will be scheduled to access resources/processor and terminate appropriately. We will not discuss this concept in detail as it is very specific topic, if you're interesting to know more please check Andrew Tanenbaum book, "Modern Operating Systems," but don't feel limited to one book; by all means see what is out there.

With all this knowledge of synchronization in mind, let's look at some concurrency design problems.

## 9.5 Designing synchronization

When designing systems, it is useful to be able to relate the problem at hand to known problems. Several problems have gained importance in the literature and are often found in real-world scenarios. The first of these problems is the *producer-consumer problem*.

### 9.5.1 Producer–consumer problem

There are one or more producers who generate items and put them into a buffer. There are also consumers who take items from the same buffer, processing them one at a time. A single producer can generate items and store them in the buffer at its own pace. The consumer acts in a similar way, but he must make sure that he does not read from an empty buffer. Thus, the system must be constrained to prevent conflicting operations with the buffer. Breaking this down, we need to make sure that the producer will not try to add data to the buffer if it is full, and the consumer will not access data from an empty buffer.

Concurrency programming is already at your fingertips, so try to solve it yourself before you go any further.

Basic implementation will look like this:

```
# Chapter 9/producer_consumer.py
import time
from threading import Thread, Semaphore, Lock
SIZE = 5
```

```

BUFFER = ["" for i in range(SIZE)] #A
producer_idx: int = 0

mutex = Lock()
empty = Semaphore(SIZE)
full = Semaphore(0)

class Producer(Thread):
    def __init__(self, name: str, maximum_items: int = 5):
        super().__init__()
        self.counter = 0
        self.name = name
        self.maximum_items = maximum_items

    def next_index(self, index: int) -> int:
        return (index + 1) % SIZE

    def run(self) -> None:
        global producer_idx
        while self.counter < self.maximum_items:
            empty.acquire() #B
            mutex.acquire() #C
            self.counter += 1
            BUFFER[producer_idx] = f"{self.name}-{self.counter}"
            print(f"{self.name} produced: "
                  f"'{BUFFER[producer_idx]}' into slot {producer_idx}")
            producer_idx = self.next_index(producer_idx)
            mutex.release() #C
            full.release() #D
            time.sleep(1)

class Consumer(Thread):
    def __init__(self, name: str, maximum_items: int = 10):
        super().__init__()
        self.name = name
        self.idx = 0
        self.counter = 0
        self.maximum_items = maximum_items

    def next_index(self) -> int:
        return (self.idx + 1) % SIZE #E

    def run(self) -> None:
        while self.counter < self.maximum_items:
            full.acquire() #F
            mutex.acquire() #C

```

```

        item = BUFFER[self.idx]
        print(f"{self.name} consumed item: "
              f"'{item}' from slot {self.idx}")
        self.idx = self.next_index()
        self.counter += 1
        mutex.release() #C
        empty.release() #G
        time.sleep(2)

if __name__ == "__main__":
    threads = [
        Producer("Spongebob"),
        Producer("Patrick"),
        Consumer("Squidward")
    ]

    for thread in threads:
        thread.start()

    for thread in threads:
        thread.join()

```

Let's analyze the code above. We used three synchronizations:

- **full**: The semaphore keeps track of the space the Producer fills. At the beginning of the program, it is initialized as locked (counter equals to zero), because at the beginning the buffer is completely empty: producers haven't had time to fill it yet.
- **empty**: The semaphore tracks empty slots in the buffer. Initially it is set to maximum value (**SIZE** in the code), because at the beginning the buffer is completely empty.
- **mutex**: The **mutex** is used for mutual exclusion, so that only one thread can access the shared resource – the buffer – at a time.

The producer can insert a buffer at any time. Being in a critical section the producer adds an element to the buffer and increases the buffer index used for all producers, access to the critical section is controlled by the **mutex**. But before putting data into the buffer the producer will try to get an **empty** semaphore and decrease its value by 1. If the value of this semaphore is already 0, it means that the buffer is full and it will block all producers until the buffer will have available space (**empty** semaphore is greater than 0). The producer will release the **full** semaphore after adding one element to it.

On the other hand, consumer will try to get a `full` semaphore before consuming data from the buffer. If the value of this semaphore is already 0 this means that the buffer is empty, and our `full` semaphore will block any consumer until the value of the `full` semaphore is greater than 0. Then it will take the element from the buffer and work with it in its critical section. After the consumer has processed all the data from the buffer it will release the `empty` semaphore, increasing its value by 1 to let producers know that there is free space for a new element.

If the producer is ahead of the consumer, which is the usual situation, the consumer will rarely block on the `empty` semaphore, because the buffer will usually not be empty. Consequently, both the producer and the consumer work without issues with shared buffer.

#### NOTE

The same problem arises in the implementation of pipe IPC in Linux. Each pipe has its own pipe buffer which is guarded by semaphores.

In the next section, we will look at another classic problem: the *Readers-writer problem*.

### 9.5.2 Readers–writer problem

Not all operations are born equal. Simultaneous reading of the same data by any number of tasks will not cause any concurrency problems if the data being accessed does not change. The data can be a file, a block of memory or even a CPU register. It is perfectly possible to allow multiple simultaneous reads of data, as long as anyone writing the data does so exclusively, i.e., as long as there are no simultaneous writers.

For example, suppose the shared data is the library catalog. Regular library users read the catalog to find a book they are interested in. One or more librarians may update the catalog. In a general case, each access to the catalog would be treated as a critical section, and users would be forced to take turns reading the catalog. This would clearly lead to unbearable delays. At the same time, it is important to prevent librarians from interfering with

each other, and it is necessary to prevent reading while writing to prevent access to conflicting information.

If we generalize it, we can say that there are a number of tasks which only read the data (readers – library users) and a number of tasks which only write the data (writers – librarians):

- Any number of readers can read shared data at the same time
- Only one writer can write to the shared data at a time
- If a writer writes to a shared data, no reader can read it

This way, we still prevent any race conditions or bad interleaving due to read/write or write/write errors.

Thus, readers are tasks that must not exclude each other, and writers are tasks that must exclude all other tasks, both readers and writers. In this way we achieve an efficient solution to the problem instead of simply mutually excluding a shared resource for any operation.

In various libraries or programming languages you will often find a Readers-writer lock (aka RWLock) which solves such problems. This type of lock is usually used in large operations and can greatly improve performance if the protected data structure is read often and changed only occasionally. Since there is no such thing in Python, let's implement it ourselves:

```
# Chapter 9/reader_writer/rwlock.py
from threading import Lock

class RWLock:
    def __init__(self) -> None:
        self.readers = 0
        self.read_lock = Lock()
        self.write_lock = Lock()

    def acquire_read(self) -> None: #A
        self.read_lock.acquire() #A
        self.readers += 1 #A
        if self.readers == 1: #A
            self.write_lock.acquire() #A
        self.read_lock.release() #A

    def release_read(self) -> None: #B
```

```

        assert self.readers >= 1 #B
        self.read_lock.acquire() #B
        self.readers -= 1 #B
        if self.readers == 0: #B
            self.write_lock.release() #B
        self.read_lock.release() #B

    def acquire_write(self) -> None: #C
        self.write_lock.acquire() #C

    def release_write(self) -> None: #D
        self.write_lock.release() #D

```

During normal operation, a lock can be accessed by several readers at the same time. However, when some thread wants to update the shared data, it locks until all readers release the lock, after which the writer gets the lock and updates the shared data. While a thread is updating the shared data, new reader threads are blocked until the writer thread is finished.

Example of the reader and writer threads implementation:

```

# Chapter 9/reader_writer/reader_writer.py
import time
import random
from threading import Thread

from rwlock import RWLock

counter = 0 #A
lock = RWLock()

class User(Thread):
    def __init__(self, idx: int):
        super().__init__()
        self.idx = idx

    def run(self) -> None:
        while True:
            lock.acquire_read()
            print(f"User {self.idx} reading: {counter}")
            time.sleep(random.randrange(1, 3))
            lock.release_read()
            time.sleep(0.5)

class Librarian(Thread):
    def run(self) -> None:

```

```

global counter
while True:
    lock.acquire_write()
    print("Librarian writing...")
    counter += 1
    print(f"New value: {counter}")
    time.sleep(random.randrange(1, 3))
    lock.release_write()

if __name__ == "__main__":
    threads = [
        User(0),
        User(1),
        Librarian()
    ]

    for thread in threads:
        thread.start()

    for thread in threads:
        thread.join()

```

Here we have two user threads that read shared memory and one librarian thread that changes it. The output:

```

User 0 reading: 0
User 1 reading: 0
Librarian writing...
New value: 1
User 0 reading: 1
User 1 reading: 1
Librarian writing...
New value: 2
User 0 reading: 2
User 1 reading: 2
User 0 reading: 2
User 1 reading: 2
User 0 reading: 2
User 1 reading: 2
User 0 reading: 2
Librarian writing...
New value: 3

```

The output will show us that no user reads while librarian is writing, and no librarian writes while any of the users are still reading the shared memory.

## 9.6 Summary

That was a long chapter! Let's go over the main points we've gone over in this chapter.

When it comes to thread safety, good design is the best protection a developer could have. Avoiding shared resources and minimizing communication between tasks makes it less likely that these tasks will mess with each other. However, it is not always possible to create an application that is completely not using shared resources. In that case proper synchronization is required.

Synchronization helps to ensure that the code is correct, but it comes at the expense of performance. The use of locks introduces delays even in non-conflicting cases. In order for a task to access shared data, it must first obtain a lock associated with that data. To get the lock, synchronize it between tasks, and monitor the shared objects, the processor has to do a bunch of work hidden from developer. Locks and atomic operations usually involve memory barriers and kernel-level synchronization to ensure proper code protection. If multiple tasks are trying to get the same lock, the overhead increases even more. Global locks can also become scalability inhibitors.

Therefore, if possible, try to design without synchronization of any type. In case of communication, instead of shared memory you may consider using message passing IPC – in that case you can avoid sharing the memory between different tasks so each task would have its own copy of data, he can work with safely. You can do this with algorithmic improvements, good design models, proper data structures, or synchronization-independent classes.

## 9.7 Recap

- Concurrency is not an easy concept, and when developers implement concurrency in their applications, they may encounter a variety of problems. A few of the most common problems they might encounter are:
  - Careless use of synchronization primitives can lead to *deadlocks*.

During deadlock several tasks are waiting for resources occupied by each other, and none of them can continue execution.

- A similar situation to deadlock occurs in *livelock*, another frequent concurrency implementation problem. Livelock is a situation where a request for an exclusive lock is repeatedly rejected because there are multiple overlapping locks that keep interfering with each other. Tasks keep running, but don't complete their work.
- An application thread can also experience *starvation*, where it never gets CPU time or access to shared resources because other "greedy" threads hog the resources. Tasks are "starved", never receiving resources, and, in turn, the work in them does not get done. Starvation can be caused by errors in the scheduling algorithm or usage of synchronization.
- Concurrency is not a new field and hence there a lot of common design problems that people already solved in the past multiple times and they become some sort of best practices or design patterns that need to be learned. Some of the most known once are *producer-consumer problem* and *readers-writers problem*. They can be solved most efficiently with the use of semaphores and mutexes.

# 10 Non-blocking I/O

## In this chapter

- You extend your knowledge of message passing IPC from a single computer to a distributed network of computers
- You meet a typical and very popular example of an I/O-bound application, the client-server application
- You learn that the use of multiple threads or processes has its limits and its problems when it comes to I/O operations
- You get to know non-blocking operations and how they can help us hide I/O-bound operations

As processor speeds have historically increased, allowing for the execution of more operations in a given time, I/O speeds have struggled to keep up.

Applications today heavily rely on I/O rather than CPU operations, resulting in longer durations for tasks such as writing to the hard disk or reading from the network compared to CPU operations. Consequently, the processor remains idle while waiting for the completion of I/O, preventing the application from performing any other tasks. This limitation creates a significant bottleneck for high-performance applications.

In this chapter, we will explore a potential solution to this problem by delving into the message passing inter-process communication approach. We will leverage our existing understanding of the thread-based model and examine its application in high-load I/O scenarios, with a particular focus on its popular use in web server development. Web servers serve as an excellent example for demonstrating how asynchronous programming functions and the underlying concepts that empower developers to fully utilize concurrency for such tasks. We will further enhance this approach in subsequent chapters.

## 10.1 Distributed world

Concurrency has long gone beyond a single computer. The Internet and the

World Wide Web have become the backbone of our modern life. And modern technology makes it possible to connect hundreds and thousands of distributed computers. This has led to the emergence of distributed systems and distributed computing. Tasks in such systems can run on the same computer or on different computers in the same local network, or geographically distant from each other. All of that is based on a set of different, interrelated technologies, the most important being message passing IPC (Chapter 5).

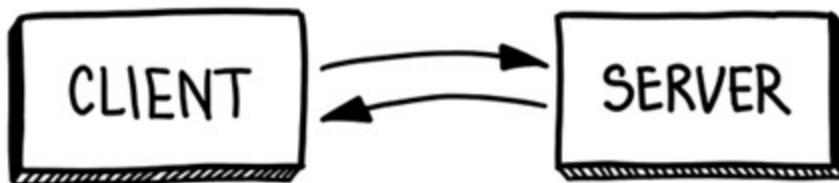
In this context, the component is a task on a single machine, the resources are all the hardware components of the computer and the individual functions that are delegated to a given computational node. Data is stored in the memory of the application process, and communication between nodes occurs through specialized protocols over the network.

The most common design for communication between such nodes is the *client-server model*.

## 10.2 Client-server model

In this model, there are two kinds of processes: *clients* and *servers*.

Server applications provide services to some client applications. The client initiates the communication by connecting to the server. Then a client can request a service by sending a message to the server. The server repeatedly receives service requests from clients, performs the service and (if necessary) returns a completion message to the client. Finally, the client disconnects.



Many network applications work this way: web browsers are clients for web

servers, an e-mail program is a client for an e-mail server, and so on.

In order to communicate, client and server can do so via network sockets.

### 10.2.1 Network sockets

We already have talked about the concept of sockets when we were talking about message passing IPC in Chapter 5, but in this case, we are talking about different type of sockets: *network sockets*.

Network sockets are the same as Unix Domain Sockets, but they are used to sending messages over a network. A network can be a logical network, the local network of a computer, or it can be a network physically connected to an outside network with its own connections to other networks. The obvious example is the Internet.

There are different types of network sockets, but in this chapter, we will focus on TCP/IP sockets. This kind of socket provides a guarantee of data delivery and is therefore the most popular. With TCP/IP sockets, a connection is established. This means that two processes must agree before information can be sent between them. This connection is maintained by both processes throughout the communication session.



The network socket is an abstraction used by the operating system to communicate with the network. For developers it represents the end point of this connection. This socket takes care of reading and writing data to/from the network, and then sends data to the network. Every socket contains two important things: IP address and port.

## **IP address**

Each device (host) connected to the network has a unique identifier. This unique identifier is represented as an IP address. IP addresses (version 4) have a common format, a set of four numbers separated by dots, such as 8.8.8.8.

Using the IP address, we can connect a socket to a specific host anywhere on the network including printers, cash registers, refrigerators, as well as servers, mainframes, PCs, etc.

In many ways, IP addresses are similar to the mailing address of a house on a street. A street may have a name, such as 5th Avenue, and there may be several houses on it. Each house has a unique number; thus, 175 5th Avenue is uniquely different from 350 5th Avenue by house number.

## **Port**

In order to accommodate multiple server applications on a single machine that clients wish to connect to, a mechanism is required to route traffic from the same network interface to different applications. This is achieved through the use of multiple *ports* on each machine

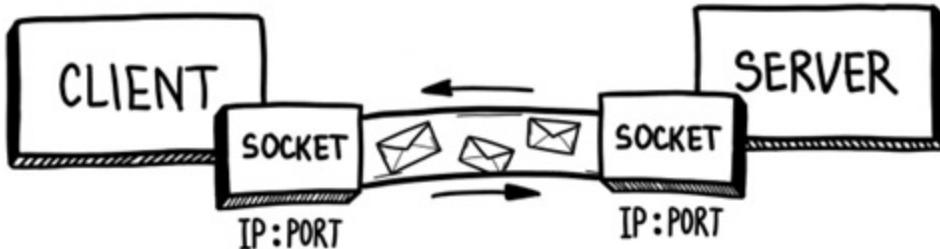
Each port serves as an entry point for a specific application, actively listening for incoming requests. The server process is bound to a particular port and remains in a listening state, ready to handle client connections. Clients, in turn, need to be aware of the port number on which the server is listening in order to establish a connection.

Certain well-known ports are reserved for system-level processes and serve as standard ports for specific services. These reserved ports provide a consistent and recognizable means for clients to connect to the corresponding services.

Think of it as an office in a business center. Each business has its own facility where they provide services.

Both client and server have their own socket connected to the other socket. The server socket listens on a specific port, and the client socket connects to the server socket on that port. Once the connection is established, data exchange begins. This is similar to a business center where business A has its own office, and clients connect to that office to receive services.

The sender process puts the information it needs into a message, then sends it explicitly over the network to the receiver socket, and the receiver process then reads it as we described on UDS sockets. The processes in this exchange can either be executed on the same machine or on different machines connected to each other by a network.



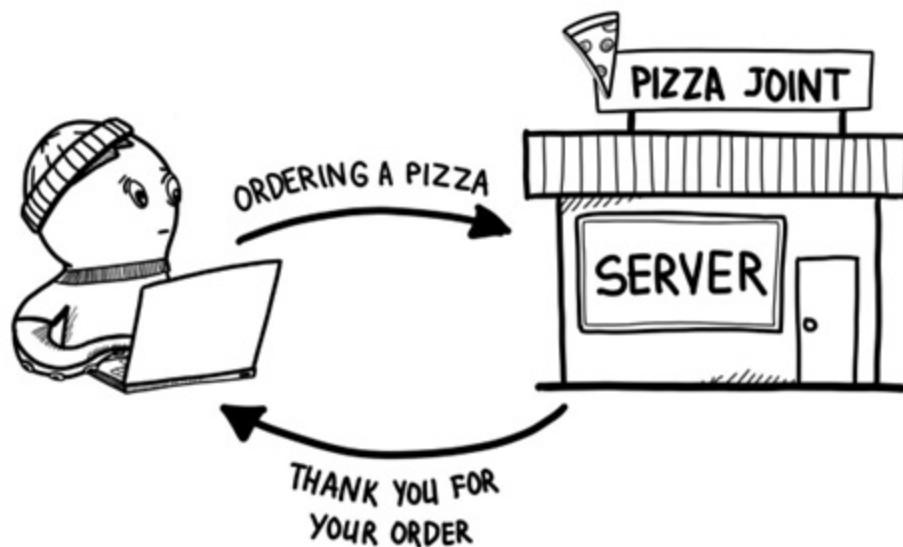
We will use the server implementation as a good exercise and a concrete example to help us understand how concurrency has evolved with the new challenges it presents.

We will not go too deeper into the network model and protocol stack for this communication. Networking and sockets are big topics and there are literally volumes written about them. If you are new to sockets or networking, it is perfectly normal to feel overwhelmed by all the terms and details. If you want to dive deep into the details, you can find more about the topic in the book “Modern Operating Systems” by Andrew Tanenbaum<sup>[1]</sup>.

Now that you have a basic understanding of network sockets and client-server communication, you are ready to build your first server. We will start with the simplest, sequential version of and later modify this implementation to see how and why the transition from concurrency to asynchrony has happened.

## 10.3 Pizza ordering service

In the 1980s, the Santa Cruz Operation (which did more to create the internet than AL Gore) ordered a lot of pizza for developers from a particular pizza parlor in downtown Santa Cruz, California. The process of ordering on the phone took too long so the developers created the world's first commerce app where they could order and pay for the pizza from their terminals to another terminal, they set up at the pizza parlor. Now, that was back in the era of dumb terminals connected via a wide area network, rather than personal computers. Today, the process is a bit more complicated. Let's take a moment to replicate that effort with more modern technology. We are going to implement a pizza ordering service for your local pizza joint. It will be a server that will accept pizza orders from clients and respond with "Thank you for ordering" message.



To make a server application, it must provide a server socket for clients to connect to. This is done by binding the server socket to an IP address and port on the server machine. The server application must then listen for incoming connections:

```
# Chapter 10/pizza_server.py
from socket import socket, create_server
```

```

BUFFER_SIZE = 1024 #A
ADDRESS = ("127.0.0.1", 12345) #B

class Server:
    def __init__(self) -> None:
        try:
            print(f"Starting up at: {ADDRESS}")
            self.server_socket: socket = create_server(ADDRESS) #C
        except OSError:
            self.server_socket.close()
            print("\nServer stopped.")

    def accept(self) -> socket:
        conn, client_address = self.server_socket.accept() #D
        print(f"Connected to {client_address}")
        return conn

    def serve(self, conn: socket) -> None:
        try:
            while True:
                data = conn.recv(BUFFER_SIZE) #E
                if not data: #E
                    break #E
                try:
                    order = int(data.decode())
                    response = f"Thank you for ordering {order} pizzas"
                except ValueError:
                    response = "Wrong number of pizzas, please try again"
                print(f"Sending message to {conn.getpeername()}") #F
                conn.send(response.encode())
        finally:
            print(f"Connection with {conn.getpeername()} has been closed") #G

    def start(self) -> None:
        print("Server listening for incoming connections")
        try: #H
            while True: #H
                conn = self.accept() #H
                self.serve(conn) #H
        finally: #H
            self.server_socket.close() #H
            print("\nServer stopped.") #H

if __name__ == "__main__":
    server = Server()
    server.start()

```

Here we use the local computer address `127.0.0.1` (our local machine) and port `12345`. We have bound our socket to this host and port with `create_server` call. It allows the server to accept incoming client connections. The `accept` method waits for client connection and the server will wait at this point until it receives an incoming client connection.

When the client connects, it will return a new socket object representing the connection and client address. When this happens, the server socket will create a new socket that will be used to communicate with the client. That's it. Now we have established a connection with the client, and we can communicate with it. The server is ready.

Now to start the server:

```
$ python pizza_server.py
```

When you run the above command, your terminal hangs. This is because the server is blocked on the `accept` call – it is waiting for a new client to connect.

We will be using [netcat](#) as a client (alternatively you can use Chapter 10/`pizza_client.py`). To run the client, open another terminal window and start the client.

On Unix/MacOS:

```
$ nc 127.0.0.1 12345
```

#### NOTE

In Windows use [ncat](#) on Windows: \$ ncat 127.0.0.1 12345

Once it's running, you can start typing messages – pizza orders. If the server works, you'll see responds from the server:

```
$ nc 127.0.0.1 12345  
10  
Thank you for ordering 10 pizzas!
```

And the server output:

```
Starting up on: 127.0.0.1:12345
Server listening for incoming connections
Connected to ('127.0.0.1', 52856)
Sending message to ('127.0.0.1', 52856)
Connection with ('127.0.0.1', 52856) has been closed
```

The server listens for incoming connections and when a client connects, the server communicates with it until the connection is closed (close the client for closing the connection). It then continues to listen for new connections. Take a moment to study this code.

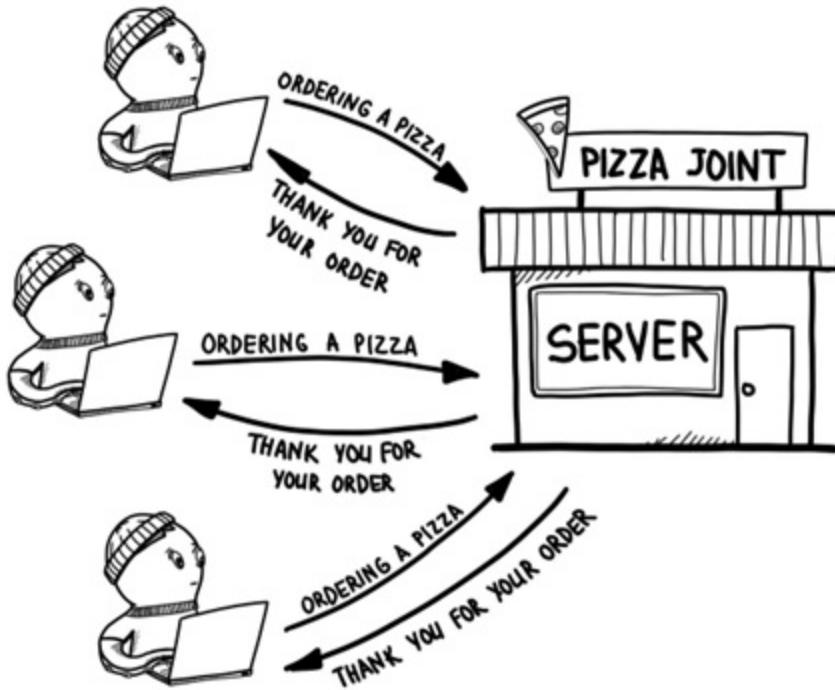
Our server is working – client now can order pizzas with it! But there is a problem in our implementation that we've missed.

### 10.3.1 A need for concurrency

Similar to the Santa Cruz Operation, this version of the server is not concurrent. When several clients try to connect to the server at about the same time, one client connects and occupies the server while other clients wait for the current client to disconnect. In the above code, the server would essentially be blocked by a single client connection!

Try it yourself. Try running a new client in a separate terminal. You will notice that the second client's connection remains pending until the first client terminates its connection. This lack of concurrency hampers the server's ability to handle multiple client connections concurrently.

In a real web application, however, concurrency is unavoidable: multiple clients and multiple servers are networked together, simultaneously sending and receiving messages and waiting for timely responses. Thus, a web application is also inherently a concurrent system requiring concurrent approaches. Consequently, concurrency is not only a feature of the web architecture, but also a necessary and decisive principle for implementing large-scale web applications to maximize the use of hardware.



### 10.3.2 Threaded pizza server

One standard solution is to use threads or processes. As we have discussed earlier, threads are generally more lightweight so we will use them for the implementation:

```
# Chapter 10/threaded_pizza_server.py
from socket import socket, create_server
from threading import Thread

BUFFER_SIZE = 1024
ADDRESS = ("127.0.0.1", 12345)

class Handler(Thread):
    def __init__(self, conn: socket):
        super().__init__()
        self.conn = conn

    def run(self) -> None:
        print(f"Connected to {self.conn.getpeername()}")
        try:
            while True:
                data = self.conn.recv(BUFFER_SIZE)
                if not data:

```

```

        break
    try:
        order = int(data.decode())
        response = f"Thank you for ordering {order} pizzas"
    except ValueError:
        response = "Wrong number of pizzas, please try again"
    print(f"Sending message to {self.conn.getpeername()}")
    self.conn.send(response.encode())
finally:
    print(f"Connection with {self.conn.getpeername()} "
          f"has been closed")
    self.conn.close()

class Server:
    def __init__(self) -> None:
        try:
            print(f"Starting up at: {ADDRESS}")
            self.server_socket = create_server(ADDRESS)
        except OSError:
            self.server_socket.close()
            print("\nServer stopped.")

    def start(self) -> None:
        print("Server listening for incoming connections")
        try:
            while True: #A
                conn, address = self.server_socket.accept() #A
                print(f"Client connection request from {address}")
                thread = Handler(conn) #A
                thread.start() #A
        finally:
            self.server_socket.close()
            print("\nServer stopped.")

if __name__ == "__main__":
    server = Server()
    server.start()

```

In the above implementation the main thread contains a listening server socket that accepts incoming connections from clients. Each client connecting to the server is handled in a separate thread. The server creates another thread that communicates with the client. The rest of the code remains unchanged.

Concurrency is achieved by using multiple threads. The operating system overlaps multiple threads with preemptive scheduling. We already know this

approach; it leads to a simple programming model because all the threads needed to process the requests can be written consistently. Moreover, it provides a simple abstraction, freeing the developer from low-level scheduling details. Instead, the developer can rely on the operating system and execution environment.

#### **NOTE**

This approach is used in many technologies, such as the module of the popular Apache web server, MPM prefork; servlets in Jakarta EE (< version 3); Spring Framework (< version 5); Ruby on Rails Phusion Passenger; Python Flask and many others.

The threaded server described above seems to solve the problem of serving multiple clients perfectly, but it comes at a price.

### **10.3.3 C10k problem**

A modern server application processes hundreds, thousands, or even tens of thousands of client requests (threads) concurrently and waiting for timely responses.

Although threads are relatively cheap to create and manage, the operating system spends a significant amount of time, precious RAM space, and other resources managing them. For small tasks such as processing single requests, the overhead associated with thread management may outweigh the benefits of concurrent execution.

#### **NOTE**

Many operating systems also have trouble handling more than a few thousand threads, usually much less. You can try your own machine with the code from Chapter 10/thread\_cost.py.

The OS is constantly shares CPU time with all threads, regardless of whether a thread is ready to continue the execution or not. For example, a thread may be waiting for data on a socket, but the OS scheduler may switch to that

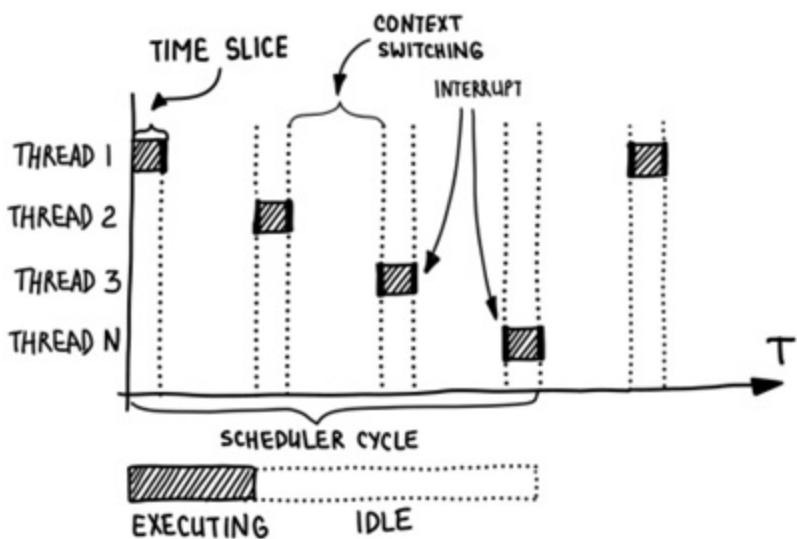
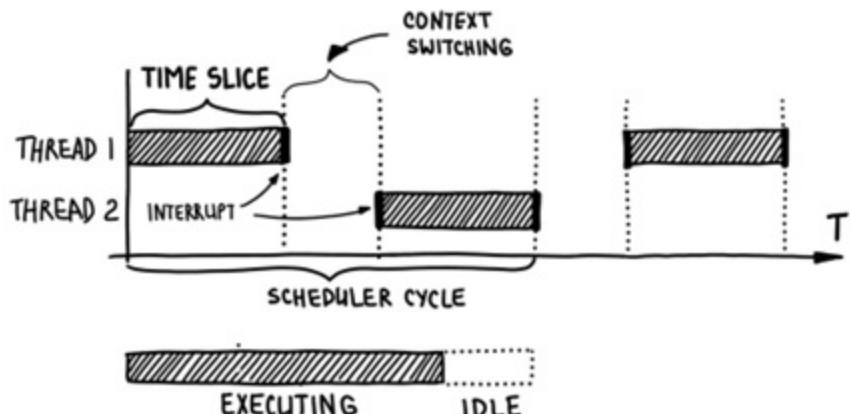
thread and back a thousand times before any useful work is done. Responding to thousands of connection requests simultaneously, using multiple threads or processes takes up significant amounts of system resources, reducing responsiveness.

Recall the preemptive scheduler from Chapter 6, which pulls up the CPU core for a thread. This may require a short wait time if the machine is heavily loaded. After that, the thread usually uses the time allocated to it and returns to the Ready state to wait for new portions of CPU time.

Now imagine if you define the scheduler period as 10 milliseconds and you have 2 threads, each thread gets 5 milliseconds separately. If you have 5 threads, each thread will get 2 milliseconds. But what happens if you have 1000 threads? Give each thread a time slice of 10 microseconds? In that case, it will spend a lot of time switching contexts and won't be able to do any real work.

You need to limit the length of the time slice. In the latter scenario, if the minimum time slice is 2 milliseconds and there are 1000 threads, the scheduler cycle needs to be increased to 2 seconds. If there are 10,000 threads, the scheduler cycle is 20 seconds.

In this simple example, if each thread uses its full time slice, it will take 20 seconds for all threads to run concurrently. That's too long.



Context-switching threads takes up precious CPU time. The more threads we have, the more time we spend switching instead of doing actual work. Thus, the overhead of starting and stopping threads would become quite high.

With a very high level of concurrency (say 10,000 threads, if you can configure the OS to create that many threads), it is possible to have an impact on throughput due to the frequent context switching overhead. This is a scalability problem, specifically named *the C10k problem*<sup>[2]</sup>, which prevents servers from handling more than 10,000 simultaneous connections.

#### NOTE

As technology has evolved from that time, this problem has been extended to

C10m, that is, how to support 10 million simultaneous connections or handle 1 million connections per second.

With threads, unfortunately, we could not solve the C10k problem; to solve it, we need to change our approach. But first, let's understand why we needed threads in the first place – we need them to handle blocking operations.

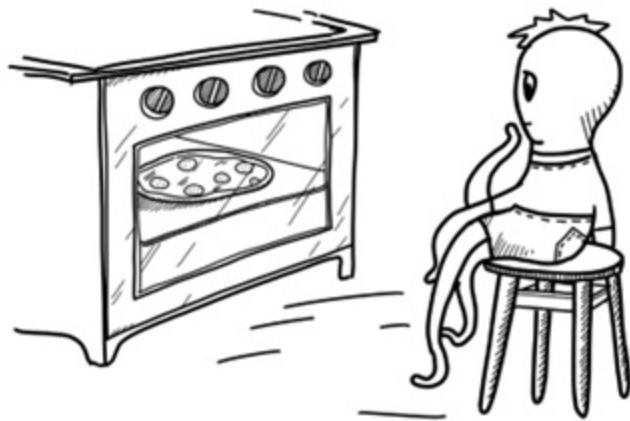
## 10.4 Blocking I/O

When you wait for data from I/O, you get a delayed response. This delay can be small when requesting a file on a hard disk and longer when requesting data from the network, because the data has to travel a greater distance to the calling party. For example, a file stored on a hard disk must reach the CPU through SATA cables and motherboard buses; data from a network resource located on a remote server must travel through miles of network cables, routers, and eventually the network interface card (NIC) in your computer to the CPU. This means that the application is *blocked* until the I/O system call is complete. The calling application is in a state where it is not using the CPU and is just waiting for a response, so it is very inefficient from a processing standpoint. And the more I/O operations, the more we run into the same problem discussed – the processor becomes just idle and doesn't do any real work.

### NOTE

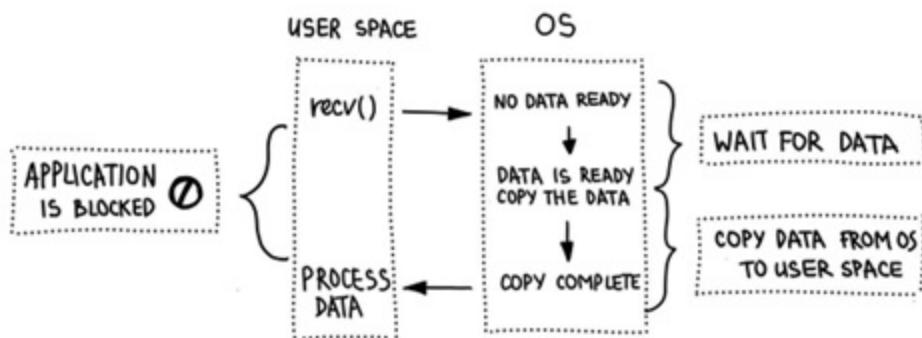
Any input/output operation is inherently a sequential operation – sending a signal and waiting for a response. Nothing is concurrent in this process, so Amdahl's Law (discussed in Chapter 2) is in full force here.

Instead of ordering, you decided to make pizza at home. To cook it, you place sauce, cheese, pepperoni, and olives on the dough (ah, pineapple on pizza is forbidden in our house). The pizza is ready to go in the oven! You put it in the oven and wait for the cheese to melt and the dough to brown. Basically, nothing else is required of you, from this point on, the oven takes care of the cooking itself. All you have to do is wait for the right moment to take it out of the oven. So, you put a chair in front of the oven, sit down and watch the pizza carefully, so you don't miss the moment when it starts to burn.



In this approach, you can't do anything else, since most of your time was spent waiting in front of the oven. It is a “synchronized” task, you are “in sync” with the oven. You have to wait and be there till the very moment the oven finishes with the pizza.

Similarly, traditional send and recv socket calls are blocking in nature. If there is no message to receive, recv system calls will block the program until it receives the data. It just put a chair, sit down and wait for the client to send the data.

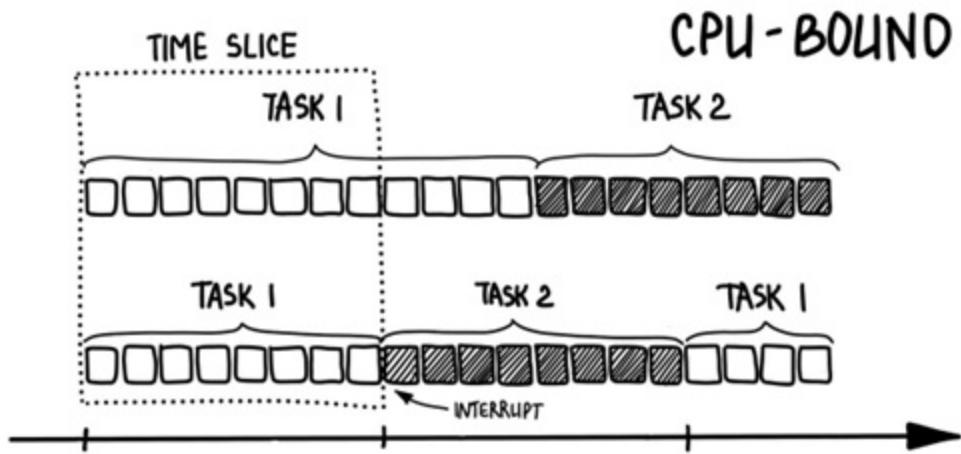


In fact, unless otherwise specified, almost all I/O interfaces (including network socket interfaces) are blocking. For conventional desktop applications, I/O-bound operations is often an occasional task. For our web servers, I/O is the primary task, and it turns out that the server doesn't use the CPU while it's waiting for a response from the client. This communication is very inefficient because it is blocking.

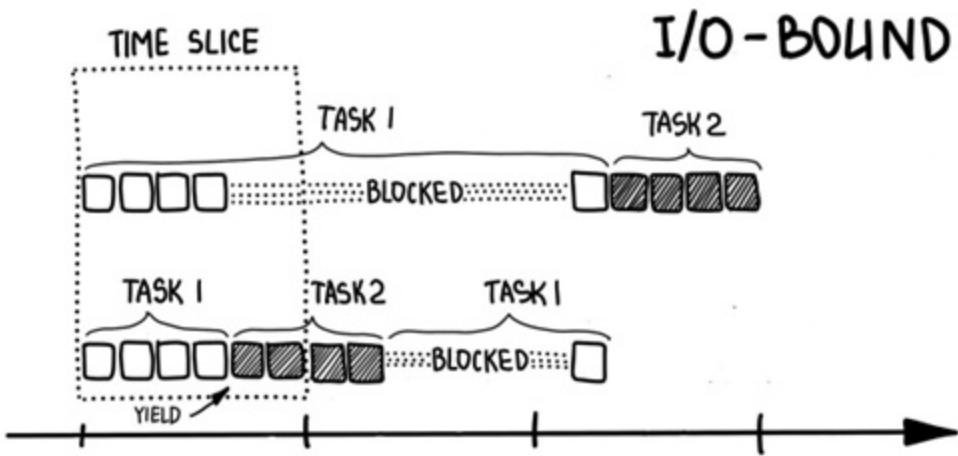
### 10.4.1 OS Optimization

Why would we use the CPU to just sit and wait for a request? When a task is blocked, the OS puts it in a *Blocked* state until the I/O operation completes. To make efficient use of physical resources, the OS immediately “parks” the blocked task, removing it from the CPU core and stores it in the system, while another Ready task is allocated CPU time. As soon as the I/O is complete, the task exits the Blocked state and goes to the Ready state and possibly to the Running state, if the OS scheduler decides so.

If a program is CPU-bound, context switching of itself will become a performance nightmare as we've seen above. Since a computational task always has something to do. It doesn't need to wait for anything, context switching stops that useful work from getting done.



If your program has a lot of I/O-bound operations, context switching will be an advantage. As soon as a task goes into a Blocked state, another task that is in a Ready state takes its place. This allows the processor to stay busy if there is work (tasks in Ready state) that needs to be done. This situation is fundamentally different from what happens with CPU-bound tasks.



So if a function is blocked (for whatever reason), it can delay other tasks. And the overall progress of the entire system may suffer. If a function is blocked because it's doing some CPU task, there's not much we can do. But if it is blocked because of I/O, we know that the CPU is idle and can be used to do another task that needs the CPU.

Blocking occurs in all concurrent programs, not only in I/O (interaction in and out of a process, possibly over a network, or writing/reading to/from a file, or with a user at the command line or GUI, etc.). Concurrent modules do not work synchronously like sequential programs. They usually have to wait for each other when coordinated actions are required.

Now imagine that we can create an operation which will not be blocked.

## 10.5 Non-blocking I/O

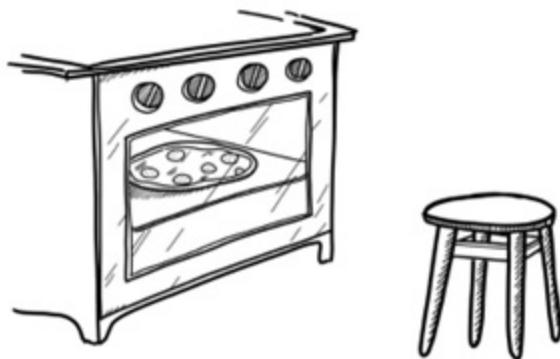
Recalling Chapter 6, it is possible to achieve concurrency without any parallelism. This can be handy when we are dealing with a large number of I/O-bound tasks. We can abandon our thread-based concurrency to achieve more scalability, avoiding the C10k problem with *non-blocking I/O*.

The idea of non-blocking I/O is to request an I/O operation and not wait for a response so we can move on to other tasks. For example, with non-blocking read we can request data over a network socket while the execution thread is

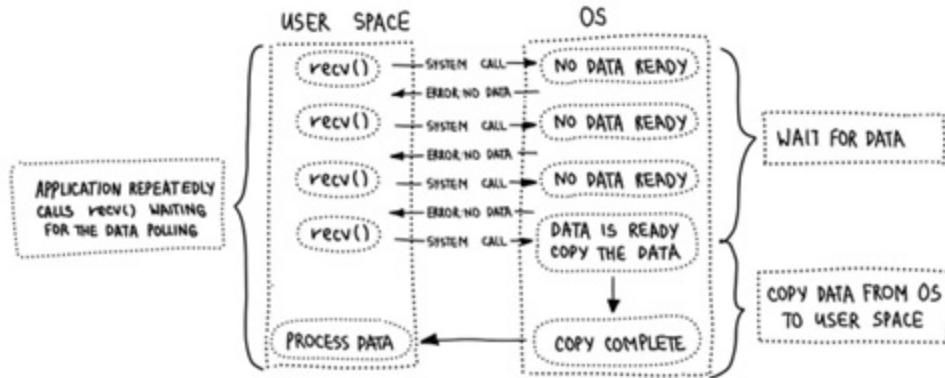
doing other things (such as working with another connection) until the data is placed in buffers ready to be consumed. The disadvantage is that we need to periodically ask if the data is ready to be read.

Since one of the problems with previous implementations was that each thread had to block and wait for the I/O to return with data, let's use another socket access mechanism: *non-blocking sockets*. All the blocking socket calls can be put into non-blocking mode.

Going back to the cooking pizza analogy. This time we're not going to be constantly monitoring the pizza, but instead we're just going to periodically go over to the oven and “asking” it whether it's ready – turn on the light in the oven and check if the pizza is ready.



The same with sockets – by putting the socket in non-blocking mode, you can effectively poll it. The consequence of non-blocking is that the I/O command cannot be executed immediately – if you try to read data from a non-blocking socket and there is no data, it will return an error (depending on the implementation, it may return special values, such as `EWOULDBLOCK` or `EAGAIN`). The simplest non-blocking approach would be creating an infinite loop by repeatedly calling I/O operations on the same socket. If any of the I/O operations are marked as complete, we process them. This approach is called *busy-waiting*.



In Python's non-blocking implementation, when calling `send`, `recv`, or `accept`, if the device has no data to read, it will raise a `BlockingIOError` exception instead of actually blocking the execution, indicating that it should have been blocked here, and the caller should try to repeat operation again in the future.

We can also remove the creation of new threads; they don't give us any particular advantage in that case. On the contrary, they only consume more RAM and waste time by context switching.

An example implementation:

```
# Chapter 10/pizza_busy_wait.py
import typing as T
from socket import socket, create_server

BUFFER_SIZE = 1024
ADDRESS = ("127.0.0.1", 12345)

class Server:
    clients: T.Set[socket] = set()

    def __init__(self) -> None:
        try:
            print(f"Starting up at: {ADDRESS}")
            self.server_socket = create_server(ADDRESS)
            self.server_socket.setblocking(False) #A
        except OSError:
            self.server_socket.close()
            print("\nServer stopped.")

    def accept(self) -> None:
```

```

try:
    conn, address = self.server_socket.accept()
    print(f"Connected to {address}")
    conn.setblocking(False) #A
    self.clients.add(conn)
except BlockingIOError: #B
    pass #B

def serve(self, conn: socket) -> None:
    try:
        while True:
            data = conn.recv(BUFFER_SIZE)
            if not data:
                break
            try:
                order = int(data.decode())
                response = f"Thank you for ordering {order} p"
            except ValueError:
                response = "Wrong number of pizzas, please tr"
                print(f"Sending message to {conn.getpeername()}") 
                conn.send(response.encode())
    except BlockingIOError: #B
        pass #B

def start(self) -> None:
    print("Server listening for incoming connections")
    try:
        while True:
            self.accept()
            for conn in self.clients.copy():
                self.serve(conn)
    finally:
        self.server_socket.close()
        print("\nServer stopped.")

if __name__ == "__main__":
    server = Server()
    server.start()

```

In the above server implementation, we make the socket non-blocking by calling `setblocking(False)`, so the server application will never wait for the operation to complete. Then, for each non-blocking socket, we try to perform accept, read and send operations in an endless while loop – *polling loop*. The polling loop should keep trying to perform them again and again because they are no longer blocking – during `send()`, we don't know if the socket is

ready – we have to keep trying until the attempt is successful. The same is true for the other calls. Thus `send`, `recv` and `accept` calls can pass control back to the main thread without doing anything at all.

#### NOTE

It is a common misconception that non-blocking I/O results in faster I/O operations. Although non-blocking I/O does not block the task, it does not necessarily execute faster. Instead, it enables the application to perform other tasks while waiting for I/O operations to complete. This allows for better utilization of processing time and efficient handling of multiple connections, ultimately enhancing overall performance. Nonetheless, the speed of the I/O operation is primarily determined by hardware and network performance characteristics, and non-blocking I/O does not affect these factors.

Since there is no blocking I/O, multiple I/O operations overlap, even if a single thread is used. This creates the illusion of parallelism because multiple tasks run concurrently (similar to how it was done in Chapter 6).

Using non-blocking I/O in the right situation will hide latency, improve throughput, and/or responsiveness of your application. It will also allow you to work with a single thread, potentially saving you from synchronization issues between threads, costs on thread management and system resources associated with this.

## 10.6 Recap

- We've met a typical and very popular example of an I/O-bound application, the *client-server application*.
- In client-server applications interacting via message passing IPC, concurrency is unavoidable: multiple clients and servers are networked together, simultaneously sending and receiving messages and waiting for timely responses.
- When I/O-bound code runs in a program, quite often the processor spends a lot of time doing nothing at all because the only thing currently running is waiting for I/O to complete.
- *Blocking* interfaces do all their work before returning to the calling

party; *non-blocking* interfaces start some work but return immediately, thus allowing other work to be done. In workloads with more I/O work than CPU work, the efficiency gains from non-blocking I/O are much higher, as one would expect.

- OS threads (and especially processes) are suitable for a small number of long running tasks, since the use of a large number of threads is limited by increasing performance degradation due to constant context switching and memory consumption due to the size of the thread stack. One simple approach to overcome the costly creation of threads or processes is to use the busy-waiting approach, where with a single thread we can concurrently process multiple client requests using non-blocking operations.

[1] Tanenbaum, Andrew S. Modern Operating Systems. 4th ed., Pearson Education, 2015.

[2] Daniel Kegel, “The C10K problem”, <http://www.kegel.com/c10k.html>

# 11 Event-based concurrency

## In this chapter

- You learn how to overcome difficulties of the inefficient busy-waiting approach in the last chapter
- You extend your knowledge on synchronization in message passing IPC
- You learn a different style of concurrency, event-based concurrency
- You learn a new design pattern – Reactor pattern

Concurrency is a critical aspect of modern software development, allowing applications to perform multiple tasks simultaneously and maximize hardware utilization. While traditional thread/process-based concurrency is a well-known technique, it is not always the best approach for every application. In fact, for high-load I/O-bound applications, event-based concurrency is often a more effective solution.

Event-based concurrency involves organizing an application around events or messages, rather than threads or processes. When an event occurs, the application responds by invoking a handler function, which performs the necessary processing. This approach has several advantages over traditional concurrency models, including lower resource usage, better scalability, and improved responsiveness.

Real-world examples of event-based concurrency can be found in many high-performance applications, such as web servers, messaging systems, and gaming platforms. For instance, a web server can use event-based concurrency to handle a large number of simultaneous connections with minimal resource consumption, while a messaging system can use it to efficiently process a high volume of messages.

In this chapter, we will explore event-based concurrency in more detail, comparing it with traditional thread/process-based concurrency and discussing its most popular use in client-server applications. We will examine the benefits and drawbacks of event-based concurrency and discuss how to

design and implement event-driven applications effectively. By the end of this chapter, you will have a solid understanding of event-based concurrency and its applications, allowing you to choose the right approach for your own projects.

## 11.1 Events

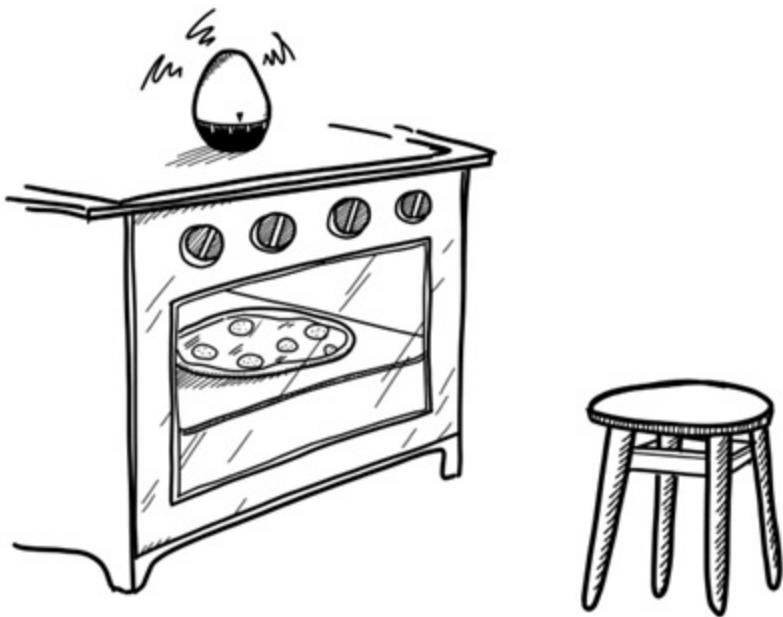
Looking back to our pizza cooking analogy, we can see that using the busy-waiting approach for cooking the pizza was not very efficient and pretty tedious.

This approach requires constantly polling all sockets, regardless of their state. If we have 10,000 sockets, and only the last socket is ready to send/receive data, we might go through them all, only to find a message waiting impatiently on the last one. The CPU is constantly running as we poll each socket to check its status. This means that 99% of our CPU time will be spent polling rather than executing other CPU-bound tasks, which is very inefficient.

We need an efficient mechanism. We need *event-based concurrency*.

What we want to know is when will the pizza be ready, right? Why don't we just set a timer to notify us when the pizza is cooked? That way we can do something else while waiting for the event. When the timer notifies us that the pizza is ready, then process the event and eat that hot, fresh pizza.

Event-based concurrency focuses on *events*. We simply wait for something to happen, i.e. an event. These could be I/O events such as data ready to consume or a socket ready for writing or any other event, such as the timer trigger. When it happens, we check what type of event it is, and do a small amount of work on processing that event (which could include executing I/O requests, scheduling other events, etc.).



#### NOTE

User interfaces are almost always designed as event-driven programs because their purpose is to respond to user actions. For example, JavaScript has historically been used to interact with the DOM and interact with the user in the browser, so an event-driven programming model was natural for this language. But this style has also become popular in some modern systems, including server-side frameworks such as Node.js. Another example of event-based concurrency is the React.js library, which is commonly used for building user interfaces. React.js uses a virtual DOM and event handlers to update the user interface in response to user input or other events, rather than updating the DOM directly. This approach allows React.js to minimize the number of DOM updates and improve performance by batching updates together.

## 11.2 Callbacks

In an event-driven program, we need to specify the code that will be run when each of the different events occurs. These are called *callbacks*.

Callback translates as “Call me back”. Indeed, the principle of callbacks is similar to the order of a phone callback. Imagine that you call an operator to

order a pizza, but an answering machine responds with a pleasant voice asking us to stay on the line until the operator is free or offers to order a callback. When the operator is free, he calls back and takes the order. Instead of waiting for an operator to answer, we can order a callback and do other things. Once the callback happens, we can do what we set out to do and then order a pizza.



#### NOTE

Callback-based code often makes the control flow less obvious and more difficult to debug. We no longer have clean and readable code. We used to be able to read the code sequentially, but now we need to spread the logic across multiple callbacks. This chain of operations in the code can cause a series of nested callbacks, also known as "[callback hell](#)".

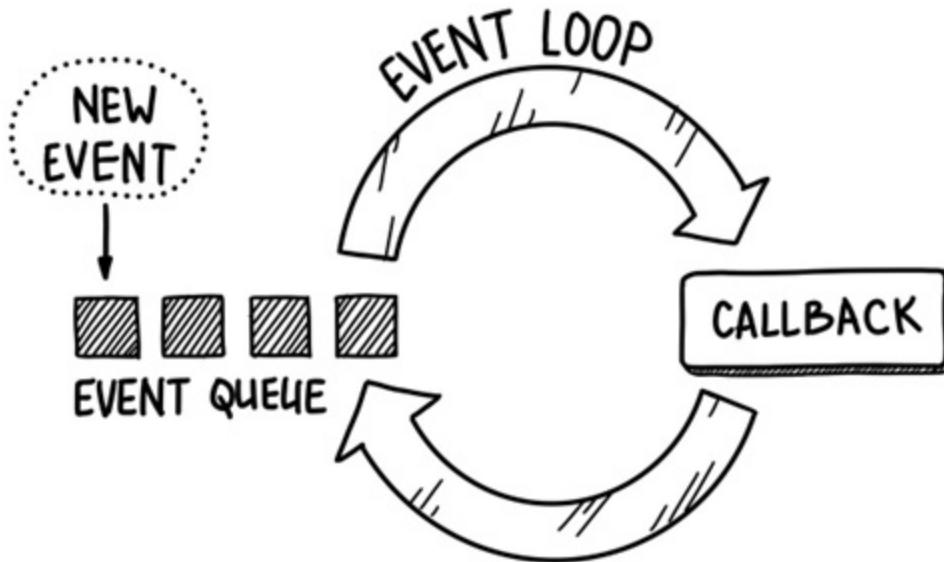
So now we have events, and we have callbacks. How do we make them work together? Event-based concurrency depends on an event loop.

### 11.3 Event loop

Combining different events and callbacks to these events means introducing a controlling entity that tracks different events and runs their appropriate callbacks. Such an entity is usually called an *event loop*.

Instead of polling for events as we did in the busy-waiting implementation, events are queued as they arrive in an *event queue*. Then event loop that waits for any event to happen and continuously retrieves them from the queue and invokes the appropriate callback.

The following diagram shows an example of a typical flow that an event-oriented program typically executes. The event loop continuously fetches events from the event queue and then makes appropriate callbacks. Although this diagram shows only one specific event mapped to one callback, it should be noted that in some event-driven applications, the number of events and callbacks could theoretically be infinite.



Essentially, all the event loop does is wait for events to occur, and then map each event to a callback that we have registered in advance and run this callback.

#### NOTE

The event loop is the heart and soul of JavaScript. In JavaScript, we are not allowed to create new threads. Instead, concurrency in JavaScript is achieved through the mechanism of event loops. This is how JavaScript is able to bridge the gap between multithreading and concurrency, making JavaScript a

serious contender in an arena filled with concurrent languages such as Java, Go, Python, Rust, and so on. Many GUI toolkits, such as Java Swing, also have an event loop.

Let's try to implement the same idea in code:

```
# Chapter 11/event_loop.py
from collections import deque
from time import sleep
import typing as T

class Event: #A
    def __init__(self, name: str, action: T.Callable[..., None],
                 next_event: T.Optional[Event] = None) -> None: #
        self.name = name #A
        self._action = action #A
        self._next_event = next_event #A

    def execute_action(self) -> None:
        self._action(self)
        if self._next_event:
            event_loop.register_event(self._next_event)

class EventLoop:
    def __init__(self) -> None:
        self._events: deque[Event] = deque() #B

    def register_event(self, event: Event) -> None:
        self._events.append(event) #C

    def run_forever(self) -> None:
        print(f"Queue running with {len(self._events)} events")
        while True: #D
            try: #D
                event = self._events.popleft() #D
            except IndexError: #D
                continue #D
            event.execute_action() #D

    def knock(event: Event) -> None:
        print(event.name)
        sleep(1)

    def who(event: Event) -> None:
        print(event.name)
        sleep(1)
```

```

if __name__ == "__main__":
    event_loop = EventLoop()
    replying = Event("Who's there?", who)
    knocking = Event("Knock-knock", knock, replying)
    for _ in range(2): #E
        event_loop.register_event(knocking) #E
    event_loop.run_forever() #F

```

Here, we have created an event loop and registered two events in it: knock and who (note that the knock event can produce who event). Then we manually put two “knock” events as like they just happened and started the infinite execution of the event loop. We see that event loop executes them one after the other:

```

Queue running with 2 events
Knock-knock
Knock-knock
Who's there?
Who's there?

```

Hence ultimately, the flow of the application depends on events. But how can the server know which event it should process next?

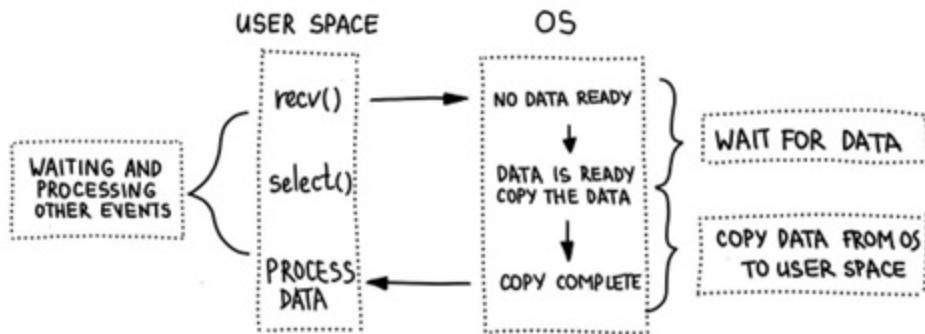
## 11.4 I/O multiplexing

Modern operating systems typically include subsystems for event notification, commonly referred to as *I/O multiplexing*. These subsystems collect and queue I/O events from monitored resources and block them until the user application is available to process them. This allows the user application to perform a simple check for incoming I/O events that require attention.

Using I/O multiplexing we don't need to keep track of all the socket events as we did in the previous chapter using busy-waiting approach. We can rely on the operating system to tell us what events happened on which sockets. The application can ask the OS to monitor the socket and queue the events until the data is ready. The application can check for events at any time it likes, perhaps doing something else in the meantime. This mechanism is provided by the granddad of system calls: select.

When you use `select` system call, you don't make any socket calls on a given socket until the `select` tells you that something has happened on that socket; for example, data has arrived and is ready to be read. However, the biggest advantage of using I/O multiplexing is that we can process multiple socket I/O requests concurrently using the same thread. We can register multiple sockets and wait for incoming events from all of them.

If there are sockets ready at the time `select` is called, it is returning control back to event loop immediately. Otherwise, it is blocked until some of the registered sockets are ready. When a new read event arrives or a socket is available for writing, `select` returns new events, places these new events in the event queue and returns control back to the event loop. This way, the application can receive new requests while it is processing a previous request. This ensures that the processing of the previous request is not blocked, but control can be quickly returned to the event loop to process the new request.



#### NOTE

Many operating systems provide a more efficient interface for event notification: POSIX provides `poll`, Linux has `epoll`, FreeBSD and MacOS use `kqueue`, Windows has `IOCP`, Solaris has `/dev/poll`... In any case, these basic primitives allow us to build a non-blocking event loop that simply checks incoming packets, reads socket messages and responds as needed.

By using I/O multiplexing, we concurrently perform several I/O operations with different sockets using the same execution thread without constantly polling for incoming events. Instead, the operating system handles the management of incoming events, notifying the application only when necessary. It is still blocking by `select` system call, but it does not actually

waste time waiting uselessly for data to arrive and wasting CPU time for in the constant event polling loop like we see in busy-waiting approach.

## 11.5 Event-driven pizza server

We are now ready to implement a single-threaded concurrent version of the pizza server using I/O multiplexing!

The core of the program is again the event loop, an infinite loop that, at each iteration, gets ready to read/write sockets from the `select` system call and invokes the corresponding registered callbacks:

```
# Chapter 11/pizza_reactor.py
class EventLoop:
    def __init__(self) -> None:
        self.writers = {} #A
        self.readers = {} #A

    def register_event(self, source: socket, event: Mask,
                      action: Action) -> None:
        key = source.fileno() #B
        if event & select.POLLIN: #C
            self.readers[key] = (source, event, action) #C
        elif event & select.POLLOUT: #D
            self.writers[key] = (source, event, action) #D

    def unregister_event(self, source: socket) -> None:
        key = source.fileno() #E
        if self.readers.get(key): #E
            del self.readers[key] #E
        if self.writers.get(key): #E
            del self.writers[key] #E

    def run_forever(self) -> None:
        while True: #F
            readers, writers, _ = select.select( #F
                self.readers, self.writers, []) #F
            for reader in readers: #G
                source, event, action = self.readers.pop(reader)
                action(source) #G

            for writer in writers: #H
                source, event, action = self.writers.pop(writer)
                action, msg = action #H
```

```
        action(source, msg) #H
```

Inside `run_forever` method of the event loop we call the `select` system call, waiting for it to tell us when clients have new events to process. This is a blocking operation, and in this case that means that the event loop will not run inefficiently. It will wait for at least one event to happen. `select` will tell us when the socket is ready to read/write, and we will call the corresponding callback.

We need to encapsulate sending and receiving data into independent functions – callbacks, for each of the expected event types – `_on_accept()`, `_on_read()`, `_on_write()`. And then let the operating system monitor the state of the client sockets instead of our application. All we need to do is register all client sockets with all expected events with corresponding callbacks. That's exactly what we do inside the `Server` class:

```
# Chapter 11/pizza_reactor.py
class Server:
    def __init__(self, event_loop: EventLoop) -> None:
        self.event_loop = event_loop
        try:
            print(f"Starting up at: {ADDRESS}")
            self.server_socket = create_server(ADDRESS)
            self.server_socket.setblocking(False)
        except OSError:
            self.server_socket.close()
            print("\nServer stopped.")

    def _on_accept(self, _: socket) -> None: #A
        try: #A
            conn, client_address = self.server_socket.accept() #A
        except BlockingIOError: #A
            return #A
        conn.setblocking(False) #A
        print(f"Connected to {client_address}") #A
        self.event_loop.register_event(conn, select.POLLIN, self._on_read)

    def _on_read(self, conn: socket) -> None: #B
        try: #B
            data = conn.recv(BUFFER_SIZE) #B
        except BlockingIOError: #B
            return #B
        if not data: #B
            self.event_loop.unregister_event(conn) #B
```

```

        print(f"Connection with {conn.getpeername()} has been
        conn.close() #B
        return #B
    message = data.decode().strip() #B
    self.event_loop.register_event(conn, select.POLLOUT, #B
                                    (self._on_write, message))

def _on_write(self, conn: socket, message: bytes) -> None: #C
    try: #C
        order = int(message) #C
        response = f"Thank you for ordering {order} pizzas!\n"
    except ValueError: #C
        response = "Wrong number of pizzas, please try again\
print(f"Sending message to {conn.getpeername()}") #C
    try: #C
        conn.send(response.encode()) #C
    except BlockingIOError: #C
        return #C
    self.event_loop.register_event(conn, select.POLLIN, self.

def start(self) -> None: #D
    print("Server listening for incoming connections")#D
    self.event_loop.register_event(self.server_socket, select
                                    self._on_accept) #D

if __name__ == "__main__":
    event_loop = EventLoop()
    Server(event_loop= event_loop).start()
    event_loop.run_forever()

```

In this implementation, we start by creating a server socket, similar to the previous approach. However, instead of a monolithic code, we represent the application as a set of callbacks, each of which handles a specific type of event request. Once these components are set up, we initiate the server to listen for incoming connections.

The core of the implementation lies in the event loop, which handles events from the event queue and invokes the corresponding event handlers. The event loop transfers control to the appropriate callback function and then resumes control once the callback has finished executing. This process continues as long as there are pending events in the event queue. When all events have been processed, the event loop returns control to the select function, which becomes blocked again, waiting for new operations to complete.

By implementing this event-driven architecture and utilizing the event loop, we have successfully addressed the challenge of handling multiple clients concurrently by running the event processing loop within a single thread. Hooray!

## 11.6 Reactor pattern

This use of the event loop, which waits for events to happen and then processes them, is so common that it has achieved the status of a design pattern: *the Reactor pattern*. By executing a single-threaded event loop using I/O multiplexing to handle non-blocking I/O and employing appropriate callbacks, you effectively employ the Reactor pattern.

The Reactor pattern handles incoming requests that arrive in the application from one or more clients. Here, an application is represented by callbacks, each of which is responsible for processing event-specific requests. For the Reactor pattern you need several components: event sources, event handlers, synchronous event demultiplexer and Reactor structure.

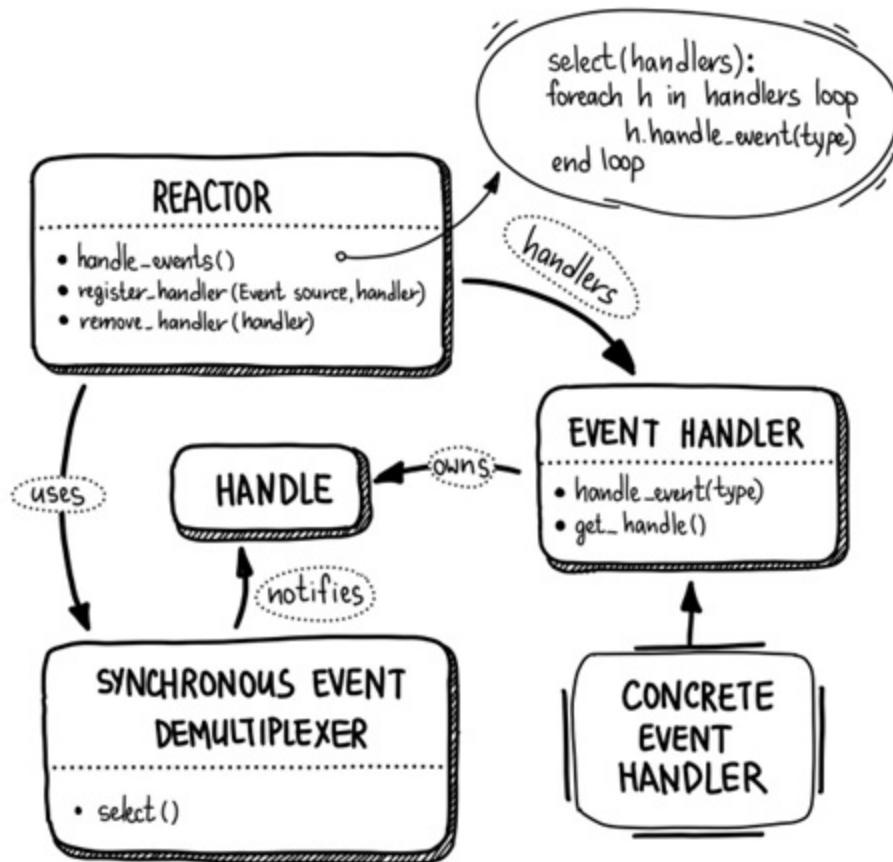
**Event sources** are entities that generate events, such as files, sockets, timers, or synchronization objects. In our pizza server code, we have two event sources: the server socket and the client sockets.

**Event handlers**, which are essentially callback functions, are responsible for processing requests from specific event sources. In our code, we had three types of event handlers:

- `_on_accept`: Handles the server socket and accepts a new connection.
- `_on_read`: Responsible for reading a new message from the client connection.
- `_on_write`: Handles writing messages to the client connection.

**Synchronous event demultiplexer** is a fancy name for getting events from event notification mechanism provided by the operating system, such as `select` or any of its flavors. It waits for specific events to occur on a set of handles.

**Reactor aka event loop** is the one who runs the show. It registers callbacks for specific events and responds to events by passing the work to the appropriate registered event handler or callback. In our code, the EventLoop class serves as the reactor, as it waits for events and “reacts” to them. When the select call returns a list of resources ready for I/O operations, the reactor calls the corresponding registered callbacks.



In summary, an application following the Reactor pattern registers event sources and event types it is interested in. For each event, it provides a corresponding event handler, which is a callback. The synchronous event demultiplexer waits for events and notifies the reactor, which then invokes the appropriate event handler to handle the event.

#### NOTE

A lot of popular core libraries and frameworks have been built on the ideas

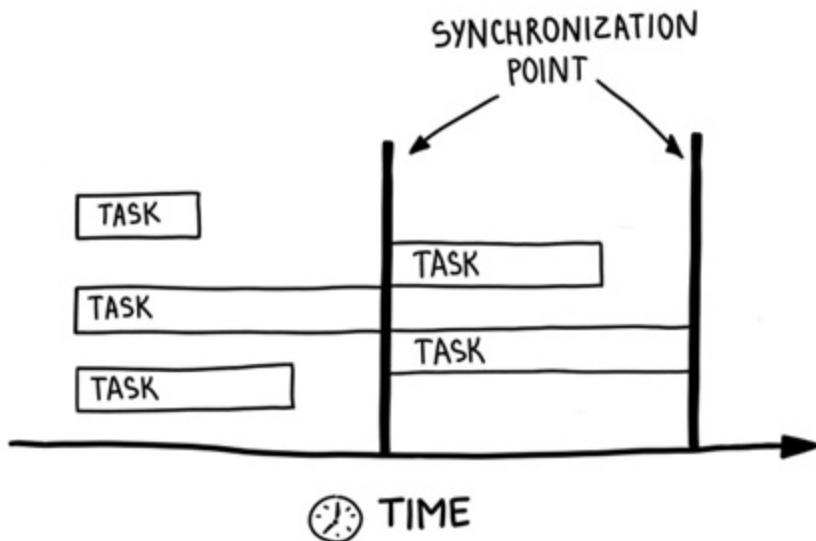
we outline here. Libevent (a widely used and long standing cross-platform event library), libuv (abstraction layer on top of libeio, libev, c-ares, iocp) is implementing low-level I/O mechanism in Node.js, Java NIO, Nginx and Vert.x use non-blocking models with event loop implementation to achieve high level of concurrency.

The Reactor pattern allows for an event-driven model of concurrency, avoiding the overhead of creating and managing system threads, context switching, and complexities associated with shared memory and locks in traditional thread-based models. By utilizing events for concurrency, resource consumption is significantly reduced, as only one execution thread is employed. However, it requires a different programming style that involves callbacks and handling events that occur at a later time.

To sum it up we can say that the Reactor pattern targets synchronous processing of the events, but asynchronous I/O processing and relies on the OS event notification system. As we've touched on the concept of synchronization let's talk about it.

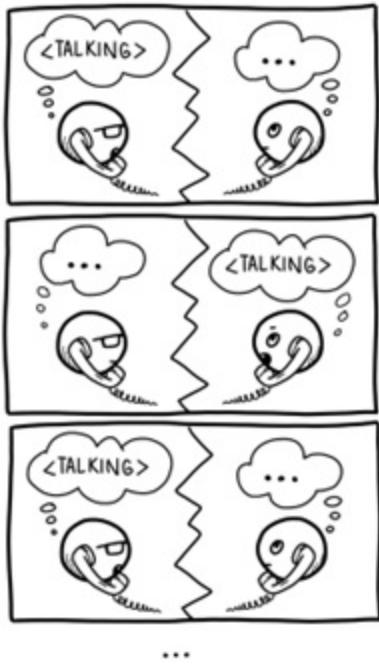
## 11.7 Synchronization in the message passing

Synchronization in message passing refers to the coordination and sequencing of tasks that rely on a specific order of execution. When tasks are synchronized, they run in order, and subsequent tasks must wait for the completion of preceding tasks before proceeding. It's important to note that synchronization refers to the start and end points of tasks, rather than their actual execution.



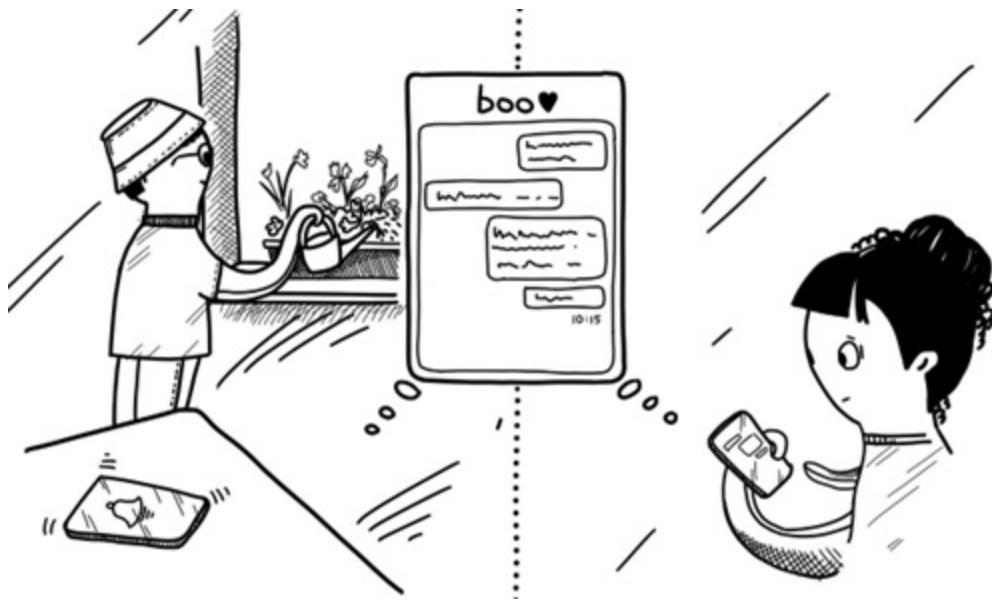
Synchronous communication requires both parties to be ready to exchange data at the same time, creating an explicit synchronization point for both tasks. This approach blocks program execution until the communication is completed, leaving system resources idle. In contrast, asynchronous communication occurs when the caller initiates a task and does not wait for it to complete but moves on. Asynchronous communication does not require synchronization when sending and receiving, and the sender is not blocked until the receiver is ready. The calling application accesses the results asynchronously and can check for events at any convenient time. This approach allows the processor to spend time processing other tasks instead of waiting.

To illustrate the difference between synchronous and asynchronous communication, think of how different people use mobile phones. During a call, while one person is talking, the other person is listening. When the first person finishes talking, the second person usually answers right away. Until the second person answers, the first person continues to wait for an answer. This means that the second person cannot continue until the first person has finished.



In this example, the first person's end point is synchronized with the second person's start point. However, while this provides immediate satisfaction to both participants, it takes longer to conclude a conversation, because the average person can consume 10 times more information while reading over listening. That's one reason why texting has become so popular among younger people.

Text messaging represents an asynchronous method of communication. One person can send a message, and the recipient can reply at their convenience. In the meantime, the sender can perform other tasks while waiting for a response.



In programming, asynchronous communication occurs when the caller initiates a task and does not wait for it to complete but moves on (like an inattentive partner). It does not require synchronization when sending and receiving, i.e., the sender is not blocked until the receiver is ready. If it cares about the results (or girlfriend) provided by such a task, it must have a way to get them (by providing a callback or in any other way). Regardless of which method is used, we say that the calling application accesses the results *asynchronously*. The application can check for events at any convenient time, perhaps by running other tasks in the meantime (or coming up with the answer to a loaded question like “How much do you love me?”). This is an asynchronous process, since the application expresses interest at one point and uses data at another point.

Asynchronous tasks don't have synchronized start and end points. Instead of waiting, the CPU time spent in synchronous communication is utilized for processing other tasks. Thus, the processor is never left idle when there is work to be done.

All asynchronous I/O operations boil down to the same pattern. It's not about how the code is executed, but where the waiting occurs. Multiple I/O operations can combine their waiting efforts so that the waiting occurs at the same place in the code. When an event occurs, the asynchronous system must resume the part of the code that was waiting for that event.

Asynchronous messaging decouples the communication between entities and allows senders to send messages without waiting for recipients. In particular, no synchronization is required for messaging between senders and receivers, and both entities can work independently of each other. When there are multiple recipients, the advantage of asynchronous messaging becomes even more apparent. It would be very inefficient to wait until all recipients of a message are ready to communicate simultaneously, or even to send a message synchronously to one recipient at a time.

The terminology blocking and synchronous, non-blocking and asynchronous are often used interchangeably. But even though they describe similar concepts they are different – they are used at different levels with different meanings. We distinguish them, at least to describe I/O operations.

## 11.8 I/O models

- **Blocking vs Non-blocking.** Using these properties, an application can tell the operating system how to access the device. When using blocking mode, the I/O operation does not return to the caller until the operation completes. In non-blocking mode, all calls are returned immediately, but only show the state of the operation. Thus, it may take several calls to make sure that the operation has been successfully complete.
- **Synchronous vs Asynchronous.** These properties are used to describe the high-level flow of control during an I/O operation. A synchronous call retains control in the sense that it does not return until the operation completes thus making a synchronization point. An asynchronous call returns immediately, allowing further operations to be performed.

Combining these properties gives four different models of I/O operations. Each of them has different uses that are advantageous for certain applications.

### 11.8.1 Synchronous blocking model

This is the most common model of operation for many typical applications. In this model, an application in the user space makes a system call that causes the application to block. This means that the application is blocked until the system call (data transfer or error) completes.

### **11.8.2 Synchronous non-blocking model**

In this model, the application accesses the I/O device in non-blocking mode. This causes the OS to immediately return the I/O call. Normally, the device is not yet ready, and the response to the call indicates that the call should be repeated later. By doing so, application code often implements busy-waiting behavior, which is highly inefficient. Once the I/O operations are complete and the data is available in the user space, the application can continue to work and use the data.

### **11.8.3 Asynchronous blocking model**

An example of this model would be a Reactor pattern. Surprisingly, the asynchronous blocking model still uses non-blocking mode for I/O operations. However, instead of busy-wait, a special blocking system call, `select`, is used to notify I/O status. However, it blocks just the notification, not the I/O call. If this notification mechanism is reliable and performant, it is a good model for highly performant I/O.

### **11.8.4 Asynchronous non-blocking model**

Finally, the asynchronous non-blocking I/O model is a model in which I/O request is returned immediately, indicating that the operation was successfully initiated. The application performs other operations while the background operation completes. When the response arrives, a signal or callback can be generated to complete the I/O operation.

An interesting feature of this model is that there is no blocking or waiting at the user level. The whole operation is moved elsewhere (to the operating system or to a device). This allows the application to take advantage of the extra processor time while I/O operations take place in the background. Not surprisingly, this model also provides good performance with highly parallel I/O.

These models describe I/O operations in operating systems at a low level only. From a more abstract developer's point of view, application framework can provide I/O access using synchronous blocking through background

threads, but provide an asynchronous interface for developers using callbacks, and vice versa.

#### NOTE

Asynchronous I/O (AIO) in Linux is a relatively recent addition to the Linux kernel. The basic idea behind AIO is to allow a process to initiate a series of I/O operations without having to block or wait for any operation to complete. Later, or after receiving an I/O completion notification, the process can retrieve the I/O results. You get a notification that the socket can be read or written without a lock. You then perform an I/O operation that is not blocked. Windows uses the completion notification model (hence I/O Completion Ports or IOCP).

## 11.9 Recap

- We learned a different style of achieving concurrency – *event-based concurrency*. It is more suitable for high-load I/O applications because it provides better scalability with higher concurrency. Such applications require less memory, even if they handle thousands of simultaneous connections.
- *Synchronous* communication refers to tasks that run in order and depend on that order. It blocks program execution for the duration of data exchange, leaving system resources idle. In synchronous communication, both parties must be ready to exchange data at the same time, and the application is blocked until the communication is completed.
- *Asynchronous* communication occurs when the caller initiates a task and does not wait for it to complete but moves on. It does not require synchronization when sending and receiving, and the sender is not blocked until the receiver is ready. Asynchronous communication allows the CPU time spent waiting in synchronous communication to be spent processing other tasks instead. The application can check for events at any convenient time, and asynchronous tasks do not have synchronized start and end points.
- The *Reactor pattern* is the most popular pattern for implementing an event-based concurrency for handling I/O-bound applications. Simply

put, it uses a single-threaded event loop, non-blocking events, and sends those events to the appropriate callbacks.

# 12 Asynchronous communication

## In this chapter

- You learn about asynchronous communication and understand when you should use an asynchronous model
- You learn the difference between preemptive and cooperative multitasking
- You learn how to implement asynchronous system using cooperative multitasking via coroutines and futures
- You learn how to combine event-based concurrency and concurrency primitives to implement an asynchronous system that can efficiently run both I/O and CPU tasks

People are impatient by nature and want systems to respond immediately. But this is not always necessary. In many programming cases, we can postpone processing or move it elsewhere so that it happens *asynchronously*. When we do this, we reduce the latency constraints on systems that actually have to run in real time. Part of the goal of moving to asynchronous operations is to reduce the workload, but it's not always a simple step.

For example, there is a very popular steak house in San Jose, California called Henry's Hi-Life, that has been an institution in the city since 1950. It is very popular with limited space, so they've developed an innovative, asynchronous method for moving patrons through quickly without making them feel rushed.

Diners enter through a small dive bar in front with the hostess at a podium in the back of the bar. The diner tells the hostess how many are in your party, and she hands them the number of menus needed. Patrons can then grab a drink at the bar while they make their choices, which is written on a checklist with special needs and then handed to the hostess. The order is taken directly to the kitchen and as soon as it is ready, the hostess guides the diners to their table and before they can put napkins in their laps the food is delivered piping hot (no microwaves allowed).

This reduces the latency constraints on the kitchen and improves the overall dining experience for customers and maximizes revenue for the restaurant. Implementing asynchronous systems can improve the performance and scalability of a system, even in scenarios where people are accustomed to immediate service.

In this chapter we will learn how to implement asynchronous systems by inheriting the "event loop plus callback" mode described in the previous chapter and turn it into its own implementation. Here we will take an in-depth look at coroutines and futures, popular abstractions for implementing asynchronous calls. We'll look at when to use asynchronous model and its examples to help you better understand this computer science term and understand what scenarios they are useful for.

## 12.1 A need for asynchrony

At first glance, the event-based approach to programming seems like a great solution. With a simple event loop, events are handled as they occur. However, there is a significant problem that arises when an event requires a system call that could potentially be blocked, such as a CPU-bound operation. This issue is compounded by the fact that instead of a single, cohesive codebase, the application is represented as a collection of callbacks, each responsible for a specific type of event request. This approach sacrifices readability and maintainability.

When it comes to servers that use threads or processes, this problem is easily solved. While one thread is busy with a blocking operation, other threads can run in parallel, enabling the server to continue functioning. The operating system handles the scheduling of threads on available CPU cores.

However, in the event-based approach, there is only a main thread with an event loop that listens for events. This means that no operation should block execution, as doing so would result in the entire system being blocked. As a result, asynchronous programming techniques must be used to ensure that operations do not block the event loop and that the system remains responsive.

## 12.2 Asynchronous procedure call

By default, in the majority of programming languages, when a method is invoked, it is executed synchronously. This means that the code runs sequentially, and control is not returned to the environment until the entire method completes. However, this can become problematic when the method takes a long time to execute, such as network call or long-running computation. In such cases, the calling thread becomes blocked until the method finishes. When this is undesirable, you can start a worker thread and call the method from it but in most cases it is not worth of additional thread with its complexities and overhead.

Let's imagine a very inefficient example. You arrive at the front desk at a hospital to check in for a procedure. If this was done with synchronous communication, the receptionist would require you to stand at the counter for as long as you need to fill out multiple forms while the receptionist just sits and waits for you. You're in the way of her or him serving other patients. The only way to scale in this approach is to hire more receptionists, and make room for additional receptionist windows. It is costly and not really efficient as most of the time they will be doing nothing. Thankfully, that is not the way it is done.



Generally, medical facilities are an asynchronous system. When you walk up

to a counter and find out you need to fill out additional forms, the receptionist hands you forms, a clipboard, and a pen and tells you to come back when you've finished. You sit down to fill out the forms while the receptionist helps the next person in line. You do not block a receptionist serving others. When you finish, you go back to the line and wait to talk to the receptionist again. If you've done something wrong or need to fill out another form, he or she will give you a new form or tell you what needs fixing, and you repeat the process: sit, do your work, and then get back in line. This system already scales well. If the queue starts to get too long, you can of course add another receptionist, making it even more scalable.

A sequential programming model can be extended to support concurrency by overloading synchronous calls with asynchronous semantics. The call does not create a synchronization point, but instead the runtime scheduler passes the results to the handler later or asynchronously. A synchronous call with asynchronous semantics added is called an *asynchronous call* or *asynchronous procedure call* (APC).

APC augmenting a potentially long running ("synchronous") method with an "asynchronous" version that returns immediately, and with additional methods that make it easy to get a completion notification or wait for completion at a later time.

Several software constructs and operations that make up the asynchronous structure have emerged in the programming world. Perhaps some of the most widely used of them is the usage of cooperative multitasking.

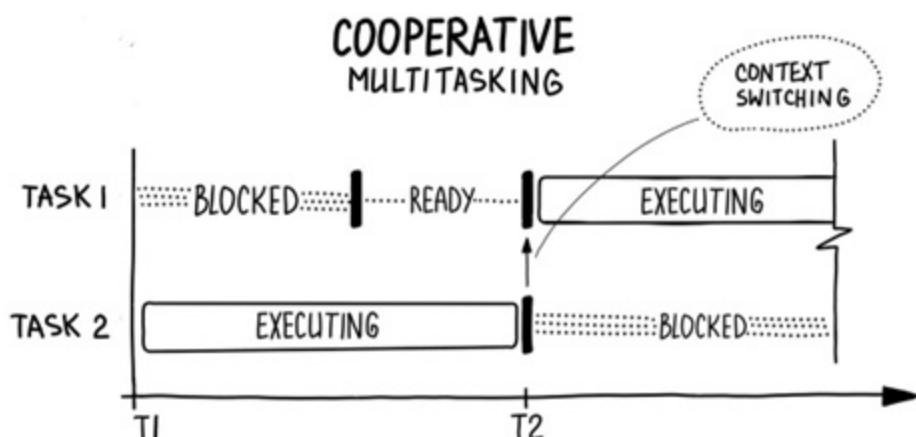
## 12.3 Cooperative multitasking

According to the Oxford Dictionary, *a-syn-chro-nous* means "of or requiring a form of computer control timing protocol in which a particular operation begins upon receiving an indication (signal) that the preceding operation has been completed". It is clear from the definition that the main problem is not how and where operations actually take place, but how to restart this or that part of the code while waiting for some event.

Up to this point, when we talked about threads, threads that relate one-to-one

to system-level threads are managed by the OS itself. But we can also have logical threads at the user or application level, these are threads that are managed by developers. The operating system knows nothing about user-level thread. It treats applications utilizing user-level threads as if they were single-threaded processes. User-level threads usually form the simplest form of multitasking, *cooperative multitasking* also known as *non-preemptive multitasking*.

In cooperative multitasking the OS never initiates context switching. Instead, each task decides when to hand over control to the scheduler, allowing other tasks to run by explicitly saying to the scheduler, “I’m pausing my work for a while; please keep running other tasks”. The scheduler’s job is only to assign tasks to available processing resources.



Thus, we have only one worker thread, and no other thread can replace the current running thread. The system is called cooperative multitasking because to be successful, the developer and the runtime environment work together in harmony to make the most of the available processing resources.

#### **NOTE**

This very simple approach has found its way into all versions of the Mac OS up to Mac OS X, also Windows up to Windows 95 and Windows NT.

Since there is only one execution thread, but multiple tasks need to be

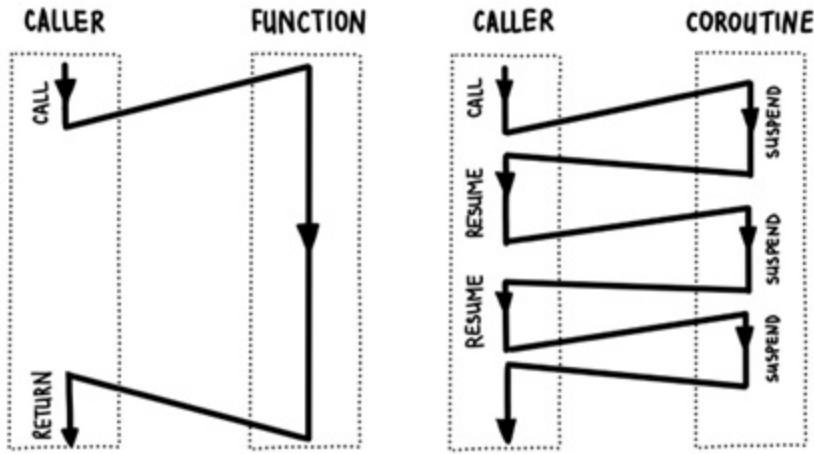
completed, there is the problem of resource sharing. In this case, the resource that needs to be shared is the thread management. But the cooperative scheduler cannot take control away from the executing task unless the task gives it by itself.

### **12.3.1 Coroutines (user-level threads)**

In our threaded server implementation (Chapter 10), the OS threads did not impose control-transfer responsibility on us, but they do provide concurrency even if we have only one processor core. The key here was the OS's ability to pause and resume thread execution using preemptive multitasking (Chapter 6). If we could have functions capable of pausing and resuming execution like OS threads, we could write concurrent single-threaded code. Guess what? We could do it with *coroutines*!

Coroutines are a programming construct that allow for cooperative multitasking, where a single thread of execution can be paused and resumed at specific points in the code. This approach offers several advantages, including the ability to write more efficient and flexible code capable of handling asynchronous tasks without explicit threading.

The key difference between coroutines and OS threads is that coroutine switching is cooperative rather than preemptive. This means that the developer, along with the programming language and its execution environment, has control over when the switch between coroutines occurs. At the right moment, a coroutine can be paused, allowing another task to start executing instead.



Coroutines are particularly useful in scenarios where certain operations are expected to block for a significant duration, such as network requests. Instead of involving the system scheduler, coroutines allow immediate switching to another task. This cooperative nature of coroutines enables developers to write code that is more elegant, readable, and reusable.

#### NOTE

The core idea of coroutines came out of work called *continuations*. Continuations can be thought of as a snapshot of the program's execution context at a specific point in time, including the current call stack, local variables, and other relevant information. By capturing this information, continuations enable a program to save its execution state and resume it later, potentially on a different thread or even a different machine.

To illustrate the usefulness of coroutines, let's consider an example of generating the Fibonacci sequence. The following Python code showcases an elegant and readable implementation using coroutines, highlighting the concept's benefits in terms of elegance, readability, and code reuse:

```
# Chapter 12/coroutine.py
from collections import deque
import typing as T

Coroutine = T.Generator[None, None, int]
```

```

class EventLoop:
    def __init__(self) -> None:
        self.tasks: T.Deque[Coroutine] = deque()

    def add_coroutine(self, task: Coroutine) -> None: #A
        self.tasks.append(task) #A

    def run_coroutine(self, task: Coroutine) -> None:
        try:
            task.send(None) #B
            self.add_coroutine(task)
        except StopIteration: #C
            print("Task completed")

    def run_forever(self) -> None: #D
        while self.tasks: #D
            print("Event loop cycle.") #D
            self.run_coroutine(self.tasks.popleft()) #D

    def fibonacci(n: int) -> Coroutine:
        a, b = 0, 1
        for i in range(n):
            a, b = b, a + b
            print(f"Fibonacci({i}): {a}")
            yield #E
        return a #F

if __name__ == "__main__":
    event_loop = EventLoop()
    event_loop.add_coroutine(fibonacci(5)) #G
    event_loop.run_forever()

```

The output of the program:

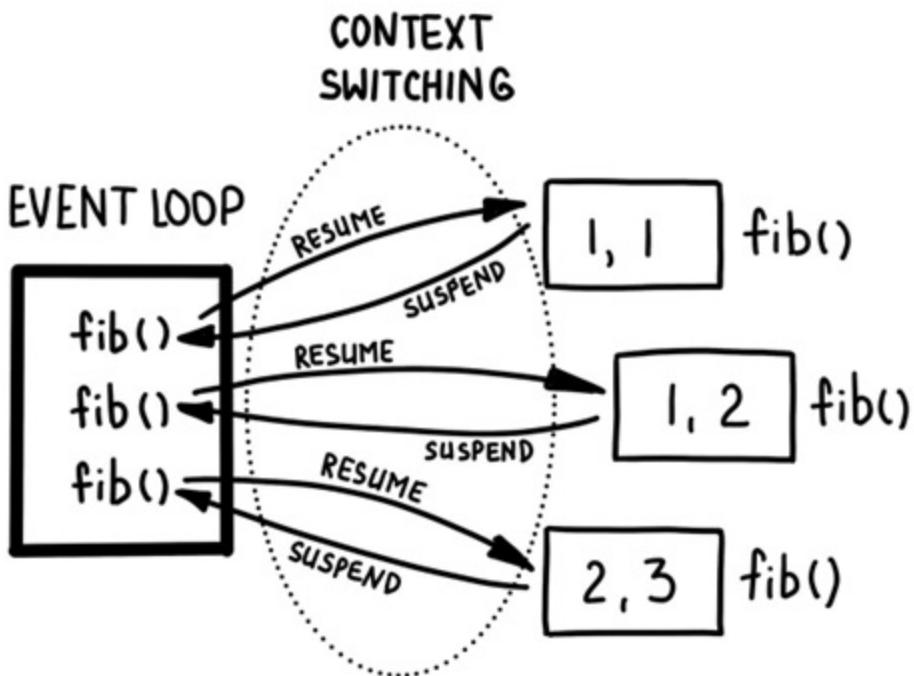
```

Event loop cycle.
Fibonacci(0): 1
Event loop cycle.
Fibonacci(1): 1
Event loop cycle.
Fibonacci(2): 2
Event loop cycle.
Fibonacci(3): 3
Event loop cycle.
Fibonacci(4): 5
Event loop cycle.
Task completed

```

In this code, we introduce a simple event loop and a coroutine. We call the coroutine just like a regular function, but it executes instructions until it reaches a "pause point" marked by the `yield` instruction. This special instruction temporarily halts the current function execution, returns control to the caller, and preserves the current instruction stack and pointer in memory, effectively saving the execution context. Consequently, the event loop remains unblocked by a single task and proceeds to execute the next task while waiting for the awaited event to occur. Once the event is complete, the event loop resumes execution from the exact line it paused on.

Over time, the main thread can invoke the same coroutine again, and it will not start execution from the beginning but from the last pause point. Therefore, a coroutine is a partially executed function that, under appropriate conditions, can be resumed at some point in the future until it completes.



In our code example, the `fibonacci` coroutine pauses and returns control to the event loop, which then awaits and pauses with a resume marker. Upon resumption, the `fibonacci` coroutine yields the result, and the event loop then resumes execution, passing the yielded value to the appropriate destination.

By employing coroutines and an event loop, we achieve cooperative multitasking, where tasks can be efficiently scheduled and executed without relying on multiple threads or processes. Coroutines allow us to write concurrent code with improved control flow, making it easier to handle asynchronous tasks and optimize resource utilization.

#### NOTE

*Fibers*, *light threads* and *green threads* are other names for coroutines or coroutine-like concepts. Sometimes they may look (usually on purpose) like operating system threads, but they don't run like real threads, instead they run coroutines. Depending on the language or implementation, there may be more specific technical features or differences between these concepts: Python (generator-based and native coroutines), Scala (coroutines), Go (goroutines), Erlang (Erlang processes), Elixir (Elixir processes), Haskell GHC (Haskell threads) and many others.

### 12.3.2 Cooperative multitasking benefits

Cooperative multitasking offers several advantages over preemptive multitasking, making it a desirable approach in certain scenarios.

#### Uses less resources

User-level threads are less resource intensive. When the operating system needs to switch between threads or processes, context switching occurs. System threads are relatively heavyweight and context switching between system threads results in significant overhead. In contrast, user-level threads are lighter in both aspects. With cooperative scheduling, because tasks maintain their own lifecycles, the scheduler does not need to monitor the state of each task, so task switching is cheaper: switching tasks is not much more expensive than calling a function. This makes it possible to create millions of coroutines without significant management overhead. Applications with this approach usually boast scalability even while being single-threaded (in the OS sense).

#### Avoids blocking shared resources

With cooperative multitasking, tasks can switch between themselves at specific points in the code, mitigating the issues of blocking shared resources. By carefully choosing these switch points, we ensure that tasks never interrupt each other in the middle of critical sections.

## Higher efficiency

Context switching is more efficient because the task itself knows when it is best to pause and pass control to another task. But this requires the task to be tremendously aware that it is not working alone – there are other tasks waiting, and it decides for itself when to hand over control. In fact, you only need one centralized sequence of operations to lose everything (see mall example in Chapter 2).

The scheduler can't make global decisions about how long tasks should run. That's why in cooperative multitasking it's important not to run long operations, and if you do, to periodically return the control. When multiple programs do very small chunks of work and voluntarily switch between each other, we can achieve a level of concurrency that no scheduler can achieve. Now you can have thousands of coroutines working together, as opposed to dozens of threads.

But preemptive multitasking and cooperative multitasking are not mutually exclusive; they are often used in the same systems at different levels of abstraction; for example, cooperative computation may be periodically preempted to provide a fairer distribution of CPU time.

## 12.4 Future objects

Imagine you went to a burger joint. There you place an order for fancy burgers for lunch. The cashier says something to the cook in the kitchen to let him know that he has to make your burger. The cashier gives you your order number, a promise that your burgers will be cooked, and you'll get them sometime in the future when the cook is done. While you wait, you go to pick a table, you sit and mind your own business. But if there is no callback method, how do we know when the burger is ready. In other words, how do we get the result of an asynchronous call?

As the return value of an asynchronous call, we can make an object that guarantees a future result (expected result or error). This object is returned as a "promise" of a future result; a placeholder object for the result that is initially unknown because the computation of its value is not yet complete. Once the actual result is obtained, we can put it inside the placeholder objects. Such objects are called *Future objects*.

Future object can be thought of as a result that will eventually become available. Future objects also act as a synchronization mechanism because they allow us to send independent computations but be synchronized with the source control and eventually return the result.

#### **NOTE**

*Future, promise, delay, or deferred* generally refer to roughly the same synchronization mechanism in different programming languages, where the object acts as a proxy for a yet unknown result. When a result becomes available, another code is executed. Over the years, these terms have come to mean slightly different meanings in different languages and ecosystems.

Back to our burger order. From time to time, you check the number listed on the counter to see if your order is ready. At some point, it's finally ready for pick up. You walk over to the counter, grab your burgers, go back to the table and enjoy your meal.



In code it looks like this:

```
# Chapter 12/future_burger.py
from __future__ import annotations

import typing as T
from collections import deque
from random import randint

Result = T.Any
Burger = Result
Coroutine = T.Callable[[], 'Future']

class Future:
    def __init__(self) -> None:
        self.done = False
        self.coroutine = None
        self.result = None

    def set_coroutine(self, coroutine: Coroutine) -> None: #A
        self.coroutine = coroutine #A

    def set_result(self, result: Result) -> None: #B
```

```

        self.done = True #B
        self.result = result #B

    def __iter__(self) -> Future:
        return self

    def __next__(self) -> Result: #C
        if not self.done: #C
            raise StopIteration #C
        return self.result #C

class EventLoop:
    def __init__(self) -> None:
        self.tasks: T.Deque[Coroutine] = deque()

    def add_coroutine(self, coroutine: Coroutine) -> None:
        self.tasks.append(coroutine)

    def run_coroutine(self, task: T.Callable) -> None:
        future = task() #D
        future.set_coroutine(task) #D
        try: #D
            next(future) #D
            if not future.done: #D
                future.set_coroutine(task) #D
                self.add_coroutine(task) #D
        except StopIteration: #D
            return #D

    def run_forever(self) -> None:
        while self.tasks:
            self.run_coroutine(self.tasks.popleft())

    def cook(on_done: T.Callable[[Burger], None]) -> None: #E
        burger: str = f"Burger #{randint(1, 10)}" #E
        print(f"{burger} is cooked!") #E
        on_done(burger) #E

    def cashier(burger: Burger, on_done: T.Callable[[Burger], None]):
        print("Burger is ready for pick up!") #F
        on_done(burger) #F

    def order_burger() -> Future:
        order = Future() #G

        def on_cook_done(burger: Burger) -> None:
            cashier(burger, on_cashier_done)

```

```

def on_cashier_done(burger: Burger) -> None:
    print(f"{burger}? That's me! Mmmmmm!")
    order.set_result(burger)

    cook(on_cook_done) #H
    return order

if __name__ == "__main__":
    event_loop = EventLoop()
    event_loop.add_coroutine(order_burger)
    event_loop.run_forever()

```

The program consists of calling the `cook` coroutine, in which the chef cooks the burger and then passes the result to the second coroutine – `cashier`, who will inform you that the burger is ready. Each coroutine returns a future object and returns control back to the main function. The function pauses until the value is ready and then resumes and completes its operation. This is what makes coroutines asynchronous.

The future object describes the idea of separating the computation and its final result by providing a proxy entity that returns the result as soon as it becomes available. The future object has a `result` property that is used to store future execution results. There is also a `set_result` method, which is used to set the result after the value is bound to the result.

While waiting until the future object is filled with the result, we can perform other computations. It provides a very simple way to call operation that takes a long time to execute or can be delayed because of costly operations such as I/O, which can slow down other elements of the program.

#### **NOTE**

There is also related scatter-gather I/O. It is a method of input/output in computing. It involves using a single procedure call to efficiently read data from multiple buffers and write it to a single data stream, or vice versa. This technique offers benefits such as improved efficiency and convenience. For example, this pattern is particularly useful for running multiple independent web requests concurrently. By scattering the requests as background tasks and gathering the results through proxy entities, it enables the concurrent

processing of operations, similar to how the `promise.all()` works in JavaScript. With `promise.all()`, you can pass an array of promises and it will wait for all of them to resolve before returning the results as an array.

If we combine future objects with the concept of coroutines – functions whose execution can be paused and then resumed – we can write asynchronous code which is very close to sequential code in its form.

## 12.5 Cooperative pizza server

Back in Chapter 10 we talked about the first e-commerce app developed by the Santa Cruz Operation to order pizza for developers back in the 1980s. That was a very simple synchronous approach but limited in scope because of a lack of computing resources. Since then programmers have learned how to run coroutines and have created future implementation, giving us all the building blocks we need to create an asynchronous server via cooperative multitasking.

### 12.5.1 Event loop

Now let's take a look at our main component – the event loop.

```
# Chapter 12/asynchronous_pizza/event_loop.py
from collections import deque
import typing as T
import socket
import select

from future import Future

Action = T.Callable[[socket.socket, T.Any], Future]
Coroutine = T.Generator[T.Any, T.Any, T.Any]
Mask = int

class EventLoop:
    def __init__(self):
        self._numtasks = 0
        self._ready = deque()
        self._read_waiting = {}
        self._write_waiting = {}
```

```

def register_event(self, source: socket.socket, event: Mask,
                   task: Action) -> None:
    key = source.fileno()
    if event & select.POLLIN:
        self._read_waiting[key] = (future, task)
    elif event & select.POLLOUT:
        self._write_waiting[key] = (future, task)

def add_coroutine(self, task: Coroutine) -> None:
    self._ready.append((task, None))
    self._numtasks += 1

def add_ready(self, task: Coroutine, msg=None):
    self._ready.append((task, msg))

def run_coroutine(self, task: Coroutine, msg) -> None:
    try:
        future = task.send(msg)
        future.coroutine(self, task)
    except StopIteration:
        self._numtasks -= 1

def run_forever(self) -> None:
    while self._numtasks:
        if not self._ready: #A
            readers, writers, _ = select.select( #A
                self._read_waiting, self._write_waiting, [])
            for reader in readers: #A
                future, task = self._read_waiting.pop(reader)
                future.coroutine(self, task) #A

            for writer in writers: #A
                future, task = self._write_waiting.pop(writer)
                future.coroutine(self, task) #A

        task, msg = self._ready.popleft() #A
        self.run_coroutine(task, msg) #A

```

In addition to the same event notification loop in our main entry point method `run_forever`, we run the `run_coroutine` method for all coroutines that are ready to run. As soon as all the tasks are done (returns future and give control back or return the result), we remove all completed tasks from the task queue. If there are no ready tasks, we call `select` as before, blocking the event loop until some event happens on the client sockets we have registered. As soon as they happen, we run the appropriate callbacks and start a new iteration of the

loop.

As we stated earlier, a cooperative scheduler cannot take control away from the executing task, as the event loop cannot interrupt a running coroutine. A running task will run until it passes control. The event loop serves to select the next task and keeps track of the blocked tasks that cannot run until I/O is complete, but it only when none of them are currently running.

To implement a cooperative server, we will need to implement coroutines for each of the server socket methods (`accept`, `send` and `recv`). There we will create a future object and return it to the event loop. We will put the result into future when the desired event has completed. To make it easier to operate let's put the asynchronous socket implementation into separate class.

```
# Chapter 12/asynchronous_pizza/async_socket.py
from __future__ import annotations

import select
import typing as T
import socket

from future import Future

Data = bytes

class AsyncSocket:
    def __init__(self, sock: socket.socket):
        self._sock = sock
        self._sock.setblocking(False)

    def recv(self, bufsize: int) -> Future:
        future = Future()

        def handle_yield(loop, task) -> None:
            try:
                data = self._sock.recv(bufsize)
                loop.add_ready(task, data)
            except BlockingIOError:
                loop.register_event(self._sock, select.POLLIN, fu

        future.set_coroutine(handle_yield)
        return future

    def send(self, data: Data) -> Future:
```

```

future = Future()

def handle_yield(loop, task):
    try:
        nsent = self._sock.send(data)
        loop.add_ready(task, nsent)
    except BlockingIOError:
        loop.register_event(self._sock, select.POLLOUT, f

future.set_coroutine(handle_yield)
return future

def accept(self) -> Future:
    future = Future()

    def handle_yield(loop, task):
        try:
            r = self._sock.accept()
            loop.add_ready(task, r)
        except BlockingIOError:
            loop.register_event(self._sock, select.POLLIN, fu

future.set_coroutine(handle_yield)
return future

def close(self) -> Future:
    future = Future()

    def handle_yield(*args):
        self._sock.close()

future.set_coroutine(handle_yield)
return future

def __getattr__(self, name: str) -> T.Any:
    return getattr(self._sock, name)

```

We made our server socket non-blocking, and, in each method, we execute the corresponding operation without waiting for it to complete. We simply give control and a future object in which we will write the result of the operation later.

We have now prepared a generic boilerplate and are ready to create our cooperative server application.

## 12.5.2 Cooperative pizza server implementation

Let's implement our asynchronous server with cooperative multitasking:

```
# Chapter 12/asynchronous_pizza/cooperative_pizza_server.py
import socket

from async_socket import AsyncSocket
from event_loop import EventLoop

BUFFER_SIZE = 1024
ADDRESS = ("127.0.0.1", 12345)

class Server:
    def __init__(self, event_loop: EventLoop):
        self.event_loop = event_loop
        print(f"Starting up on: {ADDRESS}")
        self.server_socket = AsyncSocket(socket.create_server(ADD

    def start(self):
        print("Server listening for incoming connections")
        try:
            while True:
                conn, address = yield self.server_socket.accept()
                print(f"Connected to {address}")
                self.event_loop.add_coroutine(
                    self.serve(AsyncSocket(conn)))
        except Exception:
            self.server_socket.close()
            print("\nServer stopped.")

    def serve(self, conn: AsyncSocket):
        while True:
            data = yield conn.recv(BUFFER_SIZE) #B
            if not data:
                break

            try:
                order = int(data.decode())
                response = f"Thank you for ordering {order} pizza"
            except ValueError:
                response = "Wrong number of pizzas, please try ag

            print(f"Sending message to {conn.getpeername()}") #D
            yield conn.send(response.encode()) #C
print(f"Connection with {conn.getpeername()} has been clo
```

```

conn.close()

if __name__ == "__main__":
    event_loop = EventLoop()
    server = Server(event_loop=event_loop)
    event_loop.add_coroutine(server.start())
    event_loop.run_forever()

```

In our implementation, we follow a similar approach as in previous versions by creating an event loop and assigning our server function to it for execution. Once the event loop starts running, we run clients and submit orders to the server.

However, in our cooperative multitasking approach, we don't rely on threads or processes that require control transfer, as all execution occurs within a single thread. Instead, we manage multiple tasks by transferring control to a central function responsible for coordinating these tasks—the event loop.

To sum up, cooperative multitasking provides a significant reduction in CPU and memory overhead, especially for workloads with a large number of I/O-related tasks such as servers and databases. All other things being equal, you can have orders of magnitude more tasks than OS threads because it uses a one expensive thread to handle a large number of cheap tasks.

## 12.6 Asynchronous pizza joint

For the last two chapters, you've probably been thinking, "What kind of pizza joint is this if we're not actually making pizza, but just saying 'Thanks for ordering'?" Time to put on an apron and start the oven after all!

As you can imagine, making pizza is quite a long process. Let's use this kitchen class to simulate the cooking process:

```

# Chapter 12/asynchronous_pizza/asynchronous_pizza_joint.py
class Kitchen:
    @staticmethod
    def cook_pizza(n):
        print(f"Started cooking {n} pizzas")
        time.sleep(n) #A

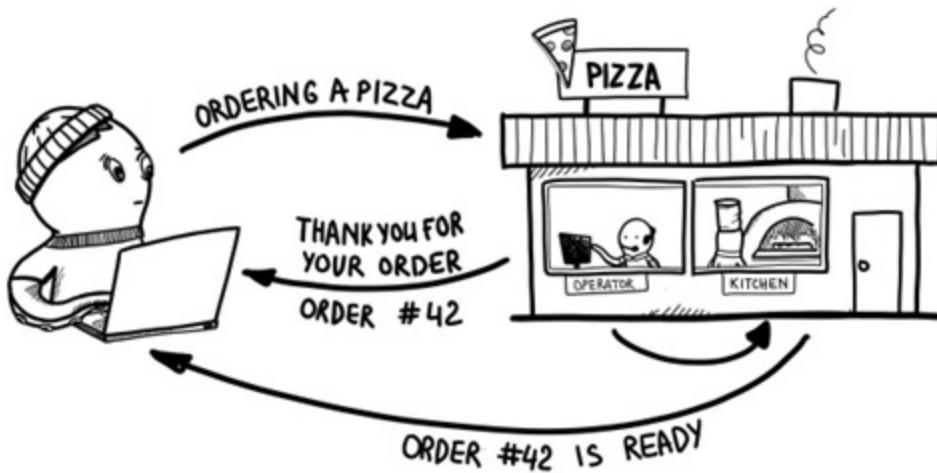
```

```
print(f"Fresh {n} pizzas are ready!")
```

If we run it in our cooperative server implementation, our server will be busy making pizza for one customer for a long time and only then serving other customers! There's a blocking call lurking in a dark corner of our "beautiful" asynchronous system! Bummer!

But we still want to continue getting the orders from customers while making pizza in background. The oven and the order server should not be blocked by each other. Yes, we are going back to basics – to threads but this time we will be using concurrency with asynchronous communication – how exciting!

The idea is to create an asynchronous method that returns a future that encapsulates a long operation that will complete at some point in the future. Once the job is sent, the future object is returned, and the caller's execution thread can continue working, separated from the new computation.



For the implementation we will be using the same approach to event notification, we can return a future object that promises that the result will arrive sometime in the future:

```
# Chapter 12/asynchronous_pizza/event_loop_with_pool.py
import socket
from collections import deque
from multiprocessing.pool import ThreadPool
import typing as T
import select
```

```
from future import Future

Data = bytes
Action = T.Callable[[socket, T.Any], None]
Mask = int

BUFFER_SIZE = 1024

class Executor:
    def __init__(self):
        self.pool = ThreadPool() #A

    def execute(self, func, *args):
        future_notify, future_event = socket.socketpair() #B
        future_event.setblocking(False) #B

        def _execute():
            result = func(*args)
            future_notify.send(result.encode())

        self.pool.apply_async(_execute) #C
        return future_event

class EventLoop:
    def __init__(self):
        self._numtasks = 0
        self._ready = deque()
        self._read_waiting = {}
        self._write_waiting = {}
        self.executor = Executor()

    def register_event(self, source: socket.socket, event: Mask,
                      task: Action) -> None:
        key = source.fileno()
        if event & select.POLLIN:
            self._read_waiting[key] = (future, task)
        elif event & select.POLLOUT:
            self._write_waiting[key] = (future, task)

    def add_coroutine(self, task: T.Generator) -> None:
        self._ready.append((task, None))
        self._numtasks += 1

    def add_ready(self, task: T.Generator, msg=None):
        self._ready.append((task, msg))

    def run_coroutine(self, task: T.Generator, msg) -> None:
```

```

        try:
            future = task.send(msg)
            future.coroutine(self, task)
        except StopIteration:
            self._numtasks -= 1

    def run_in_executor(self, func, *args) -> Future:
        future_event = self.executor.execute(func, *args)
        future = Future()

    def handle_yield(loop, task): #D
        try:
            data = future_event.recv(BUFFER_SIZE) #D
            loop.add_ready(task, data) #D
        except BlockingIOError: #D
            loop.register_event( #D
                future_event, select.POLLIN, future, task) #D

        future.set_coroutine(handle_yield)
        return future

    def run_forever(self) -> None:
        while self._numtasks:
            if not self._ready:
                readers, writers, _ = select.select(
                    self._read_waiting, self._write_waiting, [])
                for reader in readers:
                    future, task = self._read_waiting.pop(reader)
                    future.coroutine(self, task)

                for writer in writers:
                    future, task = self._write_waiting.pop(writer)
                    future.coroutine(self, task)

            task, msg = self._ready.popleft()
            self.run_coroutine(task, msg)

```

Here we are combining the thread pool with event loop. Once we get some CPU-heavy tasks we can run it inside the thread pool and return future object. Once the task is done, one of the execution threads will send a notification that it's ready and we can set the result of the future object.

Finally, our pizza joint server would look like this:

```
# Chapter 11/asynchronous_pizza_joint.py
import socket
```

```

import time

from async_socket import AsyncSocket
from event_loop_with_pool import EventLoop

BUFFER_SIZE = 1024
ADDRESS = ("127.0.0.1", 12345)

class Server:
    def __init__(self, event_loop: EventLoop):
        self.event_loop = event_loop
        print(f"Starting up on: {ADDRESS}")
        self.server_socket = AsyncSocket(socket.create_server(ADD

    def start(self):
        print("Server listening for incoming connections")
        try:
            while True:
                conn, address = yield self.server_socket.accept()
                print(f"Connected to {address}")
                self.event_loop.add_coroutine(
                    self.serve(AsyncSocket(conn)))
        except Exception:
            self.server_socket.close()
            print("\nServer stopped.")

    def serve(self, conn: AsyncSocket):
        while True:
            data = yield conn.recv(BUFFER_SIZE)
            if not data:
                break

            try:
                order = int(data.decode())
                response = f"Thank you for ordering {order} pizza"
                print(f"Sending message to {conn.getpeername()}")
                yield conn.send(response.encode())
                yield self.event_loop.run_in_executor( #A
                    Kitchen.cook_pizza, order) #A
                response = f"You order on {order} pizzas is ready"
            except ValueError:
                response = "Wrong number of pizzas, please try ag

            print(f"Sending message to {conn.getpeername()}")
            yield conn.send(response.encode())
        print(f"Connection with {conn.getpeername()} has been clo
        conn.close()

```

```
if __name__ == "__main__":
    event_loop = EventLoop()
    server = Server(event_loop=event_loop)
    event_loop.add_coroutine(server.start())
    event_loop.run_forever()
```

Although this implementation is not yet suitable for production use due to various limitations, such as insufficient exception handling and the restriction that only socket events can trigger event loop iteration, it provides a glimpse into the mechanics of concurrency using asynchronous calls. By leveraging our current knowledge, we can utilize hardware resources more efficiently, leading to increased performance. This example serves as a foundation for building next-generation asynchronous frameworks in any programming language of your choosing. By employing similar principles and techniques, you can develop more robust and scalable systems capable of handling a multitude of tasks concurrently.

#### NOTE

JavaScript is single-threaded, so the only way to achieve multithreading is to run multiple instances of the JavaScript engine. But then how to communicate between these instances? This is where Web Workers comes in. Web Workers allows to execute script operations in a background thread separate from the main execution thread of the JavaScript application. Web workers allow tasks to run in a separate thread in the background, isolated from the main thread of the web application. This multithreading capability is provided by the browser container, so not all browsers support web workers yet. Node.js is another container for the JavaScript engine, which provides multithreading with the operating system.

### 12.6.1 Asyncio library

The asynchronous programming is complex, but many complexities can be covered by the asynchronous libraries and frameworks. As an example, we can take a look at the exact same logic but with the usage of built-in into Python asyncio library:

```
# Chapter 12/asynchronous_pizza/aio.py
import asyncio
```

```

import socket

from asynchronous_pizza_joint import Kitchen

BUFFER_SIZE = 1024
ADDRESS = ("127.0.0.1", 12345)

class Server:
    def __init__(self, event_loop: asyncio.AbstractEventLoop) ->
        self.event_loop = event_loop
        print(f"Starting up at: {ADDRESS}")
        self.server_socket = socket.create_server(ADDRESS)
        self.server_socket.setblocking(False)

    async def start(self) -> None: #A
        print("Server listening for incoming connections")
        try:
            while True:
                conn, client_address = await self.event_loop.sock_
                    self.server_socket)
                self.event_loop.create_task(self.serve(conn))
        except Exception:
            self.server_socket.close()
            print("\nServer stopped.")

    async def serve(self, conn) -> None: #A
        while True:
            data = await self.event_loop.sock_recv(conn, BUFFER_S
            if not data:
                break
            try:
                order = int(data.decode())
                response = f"Thank you for ordering {order} pizza
                print(f"Sending message to {conn.getpeername()}")
                await self.event_loop.sock_sendall( #B
                    conn, f"{response}".encode())
                await self.event_loop.run_in_executor( #B
                    None, Kitchen.cook_pizza, order)
                response = f"Your order of {order} pizzas is ready"
            except ValueError:
                response = "Wrong number of pizzas, please try again"

                print(f"Sending message to {conn.getpeername()}")
                await self.event_loop.sock_sendall(conn, response.en
            print(f"Connection with {conn.getpeername()} has been closed")
            conn.close()

```

```
if __name__ == "__main__":
    event_loop = asyncio.get_event_loop()
    server = Server(event_loop=event_loop)
    event_loop.create_task(server.start())
    event_loop.run_forever()
```

The application code is greatly simplified – all the boilerplate code is now gone. Everything from sockets to the event loop and concurrency is now hidden under the library calls and managed by the library developers.

#### NOTE

This does not mean that `async/await` is the only correct approach to communication in concurrent systems. Take as an example the communicating sequential processes model (or CSP for short) implemented in Go and Clojure, or the actor model implemented in Erlang and Akka. However, `async/await` seems to be the best model we have in Python today.

This was definitely not easy code so let's step back a little bit and talk about asynchronous model in general.

## 12.7 Conclusions on the asynchronous model

Asynchronous operations, in general, do not wait for results to be completed. Instead, they delegate tasks to other locations, such as devices, threads, processes, or external systems, that can handle them independently. This allows the program to continue executing other tasks without waiting, and it receives a notification when a task finishes or encounters an error.

It's important to note that asynchrony is a characteristic of an operation call or communication, not tied to a specific implementation. Various asynchronous mechanisms exist, but they all adhere to the same underlying model. They differ in how they structure code to enable pausing when a blocking operation is requested and resuming once the operation is complete. This flexibility allows developers to choose the most suitable approach for their specific requirements and programming environment.

### 12.7.1 When should you use an asynchronous model?

Asynchronous communication is a very powerful tool for optimizing a heavily loaded system with frequent blocking system calls. But like any complex technology, it cannot be used just because it exists. Asynchrony adds complexity and make the code less maintainable. Compared to the synchronous model, the asynchronous model works best when:

- There are a large number of tasks. In that case, there's probably always at least one task that can move forward. This often results in faster response times and improved overall performance, which can be good for the end users of the system.
- Asynchrony is appropriate if the application spends most of its time doing I/O rather than processing it. For example, you have a lot of slow requests – web sockets, long polling, or you have slow external synchronous backends for which you don't know when requests will run out.
- Tasks are largely independent of each other, so there's no need for inter-task communication (and therefore no need to wait for one task to run from another).

These conditions almost perfectly characterize a typical busy server (such as a web server) in a client-server system (so the pizza examples make perfect sense). In server-side programs, asynchronous communication allows you to efficiently handle massive concurrent I/O operations, intelligently utilizing resources during their downtime and avoiding the creation of new resources. Server-side implementation is a prime candidate for the asynchronous model, which is why Python's `asyncio` and JavaScript's `Node.js` along with other asynchronous libraries have become so popular in recent years. Front-end and UI applications can also benefit from it because it enhances the flow of an application, particularly in high-volume independent I/O tasks.

## 12.8 Recap

- Asynchronous communication is a software development method that enables a single process to continue running without being blocked by time-consuming tasks, such as I/O operations or network requests. Instead of waiting for a task to finish before moving on to the next one, an asynchronous program can execute other code while the task is being

performed in the background. This approach optimizes system resources, resulting in improved program performance and responsiveness.

- Asynchrony is a property of an operation call or communication, not of a specific implementation. Asynchronous model allows efficient handling of massive concurrent I/O operations, optimizing resource utilization, reducing system delay, increasing scalability and system throughput. But without good libraries and frameworks asynchronous programs may be difficult to write and debug.
- *Cooperative multitasking* is one method used to implement asynchronous systems. It allows multiple tasks to share processing time and CPU resources. In cooperative multitasking, tasks must cooperate by yielding control back to the system once they complete a portion of their work.
  - Compared to preemptive multitasking, cooperative multitasking offers several advantages. User-level threads, which are used in cooperative multitasking, are less resource-intensive than system threads. This allows for the creation of a large number of coroutines without significant management overhead. However, it is crucial for tasks to be aware that they are not working alone and must decide when to hand over control to other tasks.
  - Cooperative multitasking significantly reduces CPU and memory overhead, particularly for workloads involving numerous I/O-related tasks like servers and databases. By utilizing a small number of threads to handle a large number of tasks, cooperative multitasking allows for a more efficient utilization of hardware resources. This, combined with asynchronous communication, leads to better resource utilization.
- Popular abstractions for implementing asynchronous calls are coroutines and futures. *Coroutine* is a function that is partially executed and paused, and under appropriate conditions will be resumed at some point in the future until its execution is complete. A *future* is a "promise" of a future result – proxy object for the result, which is initially unknown, usually because the computation of its value is not yet complete.

# 13 Writing concurrent applications

## In this chapter

- You will learn a framework for designing concurrent systems illustrated by two sample problems
- You will learn how to connect all the pieces of knowledge we learned so far together

Throughout this book, we have examined various approaches for implementing concurrent applications, as well as the concepts and problems associated with them. Now, it's time to take that knowledge and apply it to real-world scenarios.

In this chapter, we will focus on the practical application of concurrent programming by introducing a methodical approach to designing concurrent systems. We will also illustrate this approach through the examination of sample problems. By the end of this chapter, you will have the knowledge and skills needed to methodically design a simple concurrent system, as well as the ability to recognize and address any potential flaws that reduce efficiency or scalability. But before we begin, let's take a moment to review the key concepts and principles we've previously covered on the topic of concurrency.

## 13.1 So, what is concurrency?



Concurrency is a big, sometimes dizzying puzzle. Early in the history of computer, programs were written for *sequential* computations. To solve a problem in this tradition, an algorithm is constructed and implemented as a sequential stream of instructions. These instructions are executed on the CPU of a single computer. This is the simplest style of programming and straightforward execution model. Each task is executed in turn with one task being completed before the other. If the tasks are always executed in a certain order, then when the subsequent task started the execution, it can be assumed that all previous tasks have completed without errors, and all their results are available for use; a certain simplification of logic.

*Concurrent* programming means splitting a program into tasks, run them in any order, with the same result. That makes concurrency is a challenging area of software development. Decades of research and practice have led to a wide variety of concurrency models with different goals. These models are primarily designed to optimize performance, efficiency, correctness, and usability aspects. Depending on the context, there are different terms for concurrency units, such as tasks, coroutines, processes, or threads.

The processing elements can vary and include resources such as a single computer with multiple processors, multiple computers connected via network, specialized hardware, or any combination of the above. Execution process is controlled by the runtime system (operating system), in a system

with multiple processors or multiple cores, it can run in *parallel*, or *multitask* on a single processor core. The point is the execution details are handled by the runtime system, and the developer simply thinks in terms of independent tasks that can execute concurrently.

Now that you have a way to safely run tasks concurrently, you also need to find a way to coordinate them with shared resources. This is where concurrency causes problems. Tasks using old data may make inconsistent updates; systems can *deadlock*; data in different systems may never converge to consistent values, etc. The order tasks access shared resources is not fully controlled by the developer, but by how tasks are allocated to processors. That is, when each task executes and for how long is decided automatically by the implementation of the programming language in the operating system. As a result, concurrency errors are very difficult to reproduce but can be avoided by implementing proper design practices in your application, minimizing task communication, and employing effective synchronization techniques.

Now that you've got a way to safely coordinate tasks, often they also need to communicate with one another. Communication between tasks can be *synchronous* or *asynchronous*. A synchronous call retains control in the sense that it does not return until the operation completes thus making a *synchronization point*. An asynchronous call asks for something to happen, and then get notified when it does, releasing resources to do other stuff in the meantime. In the asynchronous model, a task will run until it explicitly passes control to other tasks. Note that you can mix asynchronous and concurrent models and use both in the same system.

Now let's learn a methodology that will help us create concurrent programs – Foster's Methodology.

## 13.2 Foster's Methodology

In 1995, Ian Foster proposed a set of steps for designing concurrent systems, known as *Foster's design methodology*<sup>[1]</sup>. It is a four-step design process, let's walk through the steps with an abstract approach. We will follow with some examples afterwards.

Suppose you are planning a road trip with your friends. Your task is to ensure that the trip is enjoyable and that all necessary arrangements are made.

**Partitioning** — It is possible to partition the road trip into smaller tasks such as planning the route, booking accommodation, and researching places to visit. This allows for better organization and ensures that all necessary tasks are completed.

Applying that to concurrency, you begin with identifying portions of work that can be performed concurrently. We decompose the problem into many tasks. This decomposition is achieved using data or task decomposition approaches (Chapter 7). Practical issues such as the number of processors in the target computer are ignored, and attention is focused on recognizing opportunities for independent execution.

**Communication** — In the road trip, you need to communicate with everyone involved to obtain the necessary data to execute the tasks. You can create a group chat or email thread to discuss everyone's preferences for the route, accommodation, and places to visit.

Likewise, we organize the communications necessary to obtain the data needed to execute the task. The communication required to coordinate task execution is determined, and appropriate communication structures and algorithms are defined.

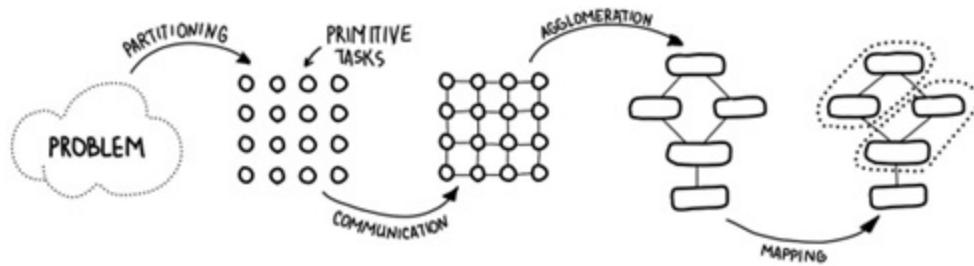
**Agglomeration** — Agglomeration refers to the process of establishing responsibility areas by dividing tasks and responsibilities into specific domains. Tasks are grouped together based on their similarity or relatedness, such as booking accommodation and researching places to visit. This allows for easier communication and coordination between team members, and simplifies the planning process, as each person is responsible for a specific area.

The tasks and communication structures defined in the first two stages of our design are evaluated with respect to performance requirements and implementation costs. It may involve the process of grouping tasks into larger tasks to reduce communication or simplify implementation, while maintaining flexibility if possible.

**Mapping** — Finally, you need to assign the tasks to the members of your road trip. For example, one person can be assigned to navigate and drive the car, while someone else can be responsible for booking accommodation and admission tickets. The goal is to minimize overall execution time and ensure that everyone has a role to play in making the road trip a success.

When we assign tasks to physical processors, usually with the goal of minimizing overall execution time. Load balancing or task scheduling techniques can be used to improve the quality of the mapping. Each task is assigned to a processor in a manner that attempts to satisfy the competing goals of maximizing processor utilization and minimizing communication costs. Mapping can be specified statically or determined at runtime by load-balancing algorithms.

### Foster's methodology



### NOTE

A common mistake in designing concurrent systems is to choose the specific mechanisms to be used for concurrency too early in the design process. Each mechanism carries certain advantages and disadvantages and choosing the "best" mechanism for a particular use-case is often determined by subtle compromises and concessions. The earlier a mechanism is chosen, the less information on which to base a choice.

Thus, design methodology machine-independent aspects, such as task independence, are considered early on, and machine-specific aspects of the design are deferred until the end of the design process. In the first two stages, we focus on concurrency and scalability and seek to find algorithms with these qualities. In the third and fourth stages, the focus shifts to efficiency

and performance. Implementation of a concurrent program is seen as the final step to ensure effective implementation of the intended algorithm, perhaps with machine or algorithm specific features in mind.

In the rest of this chapter, we will deep dive into these steps with a few examples to illustrate their application.

### 13.3 Matrix multiplication

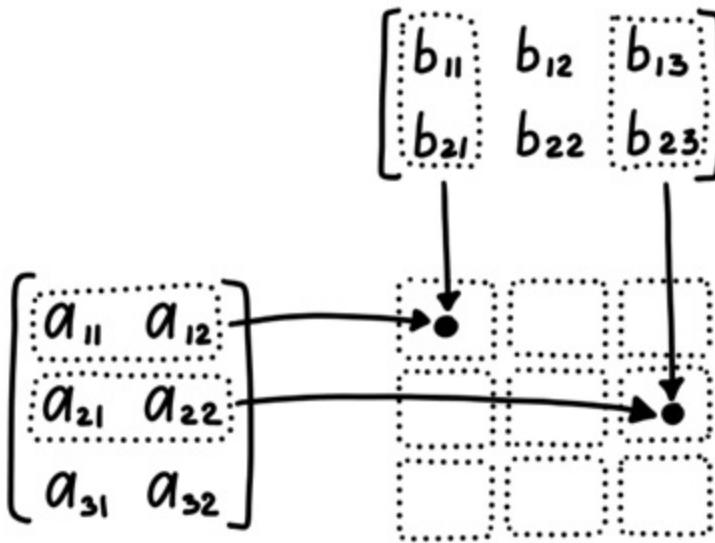
Consider using Foster's methodology on the example of matrix multiplication.

Each matrix is represented as a two-dimensional array of arrays. Two matrices can be multiplied together if the number of columns in the first matrix, A, equals the number of rows in the second matrix, B.

$$\begin{array}{c} k \\ \times \\ \boxed{\begin{array}{|c|c|c|}\hline & & \\ \hline & & \\ \hline & & \\ \hline \end{array}} \\ k \end{array} = \boxed{\begin{array}{|c|c|c|}\hline & & \\ \hline & & \\ \hline & & \\ \hline \end{array}} \\ n \quad m \\ A \times B = C$$

The product of A by B, which we will call matrix C, will have dimensions based on the number of rows in A and the number of columns in B. Each element in matrix C is the product of the corresponding row in A and the column in B.

So, for example, the element  $c_{2,3}$  is the product of the second row from matrix A and the first column from B. Written as a formula,  $c_{2,3} = a_{2,1} * b_{1,3} + a_{2,2} * b_{2,3}$ .



To give you something to compare it to, let's first make an example of the sequential algorithm.

```
# Chapter 13/matmul/matmul_sequential.py
import random
from typing import List

Row = List[int]
Matrix = List[Row]

def matrix_multiply(matrix_a: Matrix, matrix_b: Matrix) -> Matrix
    num_rows_a = len(matrix_a)
    num_cols_a = len(matrix_a[0])
    num_rows_b = len(matrix_b)
    num_cols_b = len(matrix_b[0])
    if num_cols_a != num_rows_b: #A
        raise ArithmeticError( #A
            f"Invalid dimensions; Cannot multiply " #A
            f"\{num_rows_a}\x\{num_cols_a}\x\{num_rows_b}\x\{num_cols_b}" )
    #A
    solution_matrix = [[0] * num_cols_b for _ in range(num_rows_a)]
    for i in range(num_rows_a): #C
        for j in range(num_cols_b): #D
            for k in range(num_cols_a): #E
                solution_matrix[i][j] += matrix_a[i][k] * matrix_
    return solution_matrix

if __name__ == "__main__":
    cols = 3
```

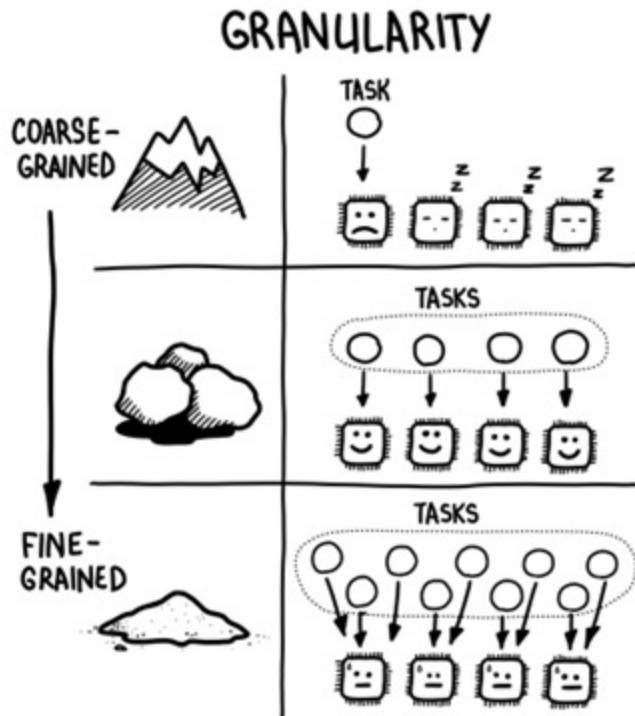
```
rows = 2
A = [[random.randint(0, 10) for i in range(cols)] for j in range(rows)]
print(f"matrix A: {A}")
B = [[random.randint(0, 10) for i in range(rows)] for j in range(cols)]
print(f"matrix B: {B}")
C = matrix_multiply(A, B)
print(f"matrix C: {C}")
```

Here, we implemented a sequential version of matrix multiplication that takes two matrixes A and B and produce the result of multiplication, matrix C. The function uses a set of nested `for` loops that iterates over rows in A and columns in B. And then uses the third `for` loop to sum the products of the elements from row A and column B. In this way, the program fills the result matrix C with values.

The goal is to design and build a concurrent program that calculates the product of two matrices, a common mathematical problem that can greatly benefit from use of concurrency.

### 13.3.1 Partitioning

The first step of the Foster's methodology, partitioning, is designed to identify opportunities for concurrency. Consequently, the focus is on identifying a large number of small tasks in order to obtain a fine-grained decomposition of the problem (Chapter 7). Just as fine sand is easier to pile than a pile of bricks, so fine-grained decomposition provides the greatest flexibility in terms of potential concurrent algorithms.



## The goal

The goal of partitioning is to discover as granular tasks as possible. This step is the only way to do this; other steps usually reduce the amount of concurrency, so the goal of this step is to find all of it. At this initial stage, we are not concerned with practical issues such as the number of processor cores and the type of target machine is ignored, and attention is focused on recognizing opportunities for parallel execution.

### NOTE

Partitioning step must produce at least an order of magnitude more tasks than the processors in the target machine. Otherwise, you will have fewer options in the later stages of the design.

## Data vs task decomposition

When we implement concurrent algorithm, we assume that it will be executed by multiple processing units. To do this, we need to isolate sets of operations

in the algorithm that can be executed independently of each other, or “decompose” it. Two types of decomposition exist – data decomposition and task decomposition (Chapter 7).

If the algorithm is used to process large amounts of data, then we can try to divide the data into parts, each of which allows independent processing by a separate processing unit. This is a *data decomposition*. Another approach involves dividing calculations by their functionality. This is *task decomposition*.

#### **NOTE**

Decomposition is not always possible. There are algorithms that do not allow the participation of several executors in their implementation. To speed up those algorithms there is still vertical scaling, but it has physical limitations (Chapter 1).

Remember that data and task decomposition are complementary ways of approaching a problem, and it's natural to use a combination of the two. Developers usually start with data decomposition because it is the basis for many concurrent algorithms. But sometimes using a task decomposition can provide a different perspective on problems. Task decomposition may reveal problems or opportunities for better optimization that an inexperienced programmer may miss by just looking at just the data.

#### **Example**

Now going back to our matrix multiplication example. We have a program in front of us, and we can start thinking about how do decompose it, and also where the dependencies are. What parts of the program we can run independently?

As it is clear from the definition of matrix multiplication, all elements of the matrix C may be computed independently. As a result, a possible approach for partitioning the matrix multiplication is to define the basic computational subtask as the problem of computing a single element of the result matrix C. The total number of subtasks in that case appears to be equal to  $n * m$

(according to the number of elements of the matrix C).

The level of concurrency achieved using this approach may seem excessive – the number of the subtasks may greatly exceed the number of the available processor cores. But that is fine at this stage, we have a follow up stage (agglomeration) where we will aggregate the computations for our specific needs.

### 13.3.2 Communication

The next step in our design process is to establish communication, which involves figuring out how to coordinate execution and set up communication channel between tasks.

#### The goal

When all computations are a single sequential program, all data is available to all parts of the program. When a computation is divided into independent tasks that may run in separate processors or even in separate processor cores, some of the data needed by a task may reside in its local memory and some in the memory of other tasks. In either case, these tasks need to exchange data with each other. Organizing this communication in an efficient way can be a challenge. Even simple decomposition can have complex communication structures. We want to minimize this overhead in our program, so it is important to define it.

#### NOTE

As we said before, the best way to implement concurrency is to reduce communication and interdependencies between concurrent tasks. If each task works with its own dataset, it does not need to protect that data with locks. Even in situations where two tasks share a dataset, you might consider splitting that dataset or giving each task its own copy. Of course, there are also costs associated with copying datasets, so you need to weigh those costs against the costs of synchronization before making a decision.

#### Example

Our concurrent algorithm at this stage is formulated as a set of tasks where each task calculates the value of an element of matrix C and expects a single row of matrix A and a single column of matrix B to be the input.

In the agglomeration stage we may consider combining the tasks to calculate not just one element of matrix C but the whole matrix row. In that case, a row of the matrix A and all the columns of the matrix B must be available for carrying out all the necessary computations of the tasks. The simple solution to that is duplicating the matrix B in all the tasks, but it may be unacceptable because of sizeable memory expenses needed for data storage.

Another option is to use shared memory all the time, as the algorithm only uses A and B for read access and elements of matrix C will be executed independently.

On the later stages we consider those options and think about the best solution for the use case at hand.

### **13.3.3 Agglomeration**

In the first two stages of the design process, computation is broken down to maximize concurrency, and communication between tasks is introduced so that the tasks have the data they need. The resulting algorithm is still an abstraction, since it is not designed to run on any particular computer. The design obtained at this point probably doesn't map well onto a real machine. If the number of tasks greatly exceeds the number of processors, the overhead will be strongly affected by how the tasks are assigned to the processors.

This third step, agglomeration, revisits the decisions made in the partitioning and communication steps.

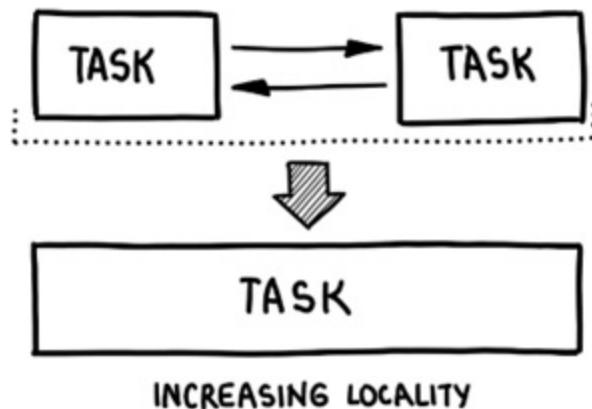
#### **The goal**

The goal of this step is to improve performance and simplify development efforts often by combining groups of tasks into larger tasks. Very often the goals are contradictory, and compromises have to be made.

In some cases, combining tasks with very different execution times can lead to performance issues. For example, if a long-running task is combined with many short-running tasks, the short-running tasks may have to wait a long time for the long-running task to complete. On the other hand, separating the tasks may simplify the design, but may result in lower performance. In such cases, a compromise may be necessary between the benefits of simplicity and performance.

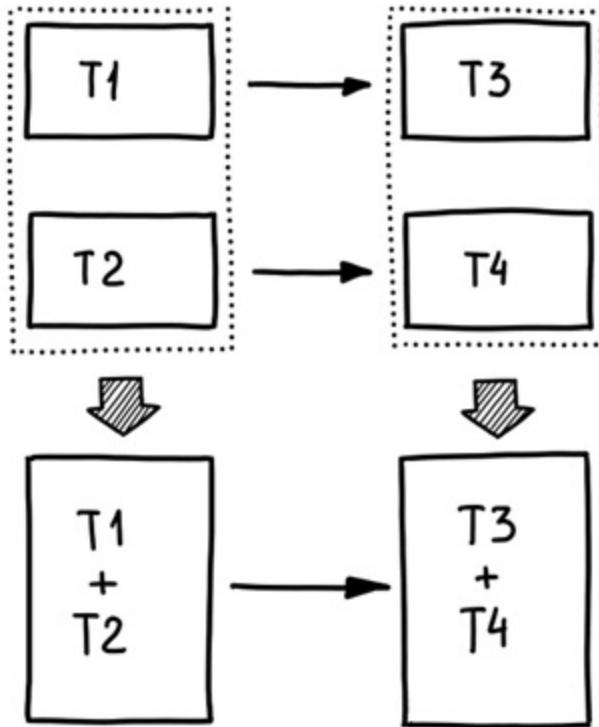
Let's consider the snow-shoveling example again from Chapter 7. Shoveling the snow is harder and slower than scattering salt, so in coming up with a plan of attack, the worker with the salt bag may want to give the shoveler a head start and then start scattering salt. When he catches up to the shoveler, then they switch jobs, giving the shoveler a breather as he takes the bag of salt and waits for the second worker to get a head start. They continue this pattern until all jobs are done. This reduces communication between the workers and improves overall performance.

Reducing communication overhead is one way to improve performance. When two tasks exchanging data with each other are combined into one task, the data communication becomes part of one task, and that communication and overhead are removed. This is called *increasing locality*.



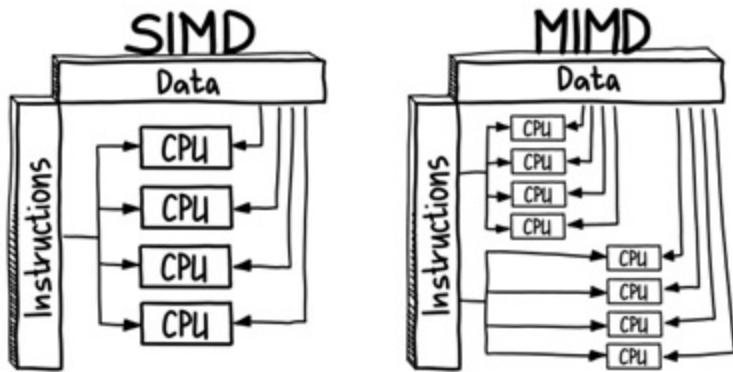
Another way to reduce communication overhead is to combine groups of tasks that all send data and groups of tasks that all receive data from each other. In other words, suppose task T1 sends data to task T3 and T2 sends T4. If we merge T1 and T2 into one task T1, and T3 and T4 into one task T3, the communication overhead will decrease. The transmission time is not reduced,

but we halve the total waiting time. Remember that when a task is waiting for data, it cannot compute, so the time spent waiting is time lost.



## Example

When we partitioned our matrixes earlier, we used a fine-grained approach. Each element of the resulting matrix needs to be calculated. We divided the multiplication task into separate tasks according to the number of elements in the resulting matrix, one for each matrix element. In evaluating the communication, we determined that each subtask should have a row of matrix A and a column of matrix B. For the SIMD computer (Chapter 3) it may be great we can share A and B matrixes between threads and on this type of machine the solution will work better if we use a large number of threads, a natural choice is for each thread to compute one element of the result.



But if we have just an ordinary hardware – MIMD computer (Chapter 3), the number of tasks would be more than the number of processors. If the size of matrixes  $n$  turns out to be larger than the number of processors, the tasks can be aggregated by combining several neighboring rows and columns of multiplied matrices into one subtask. In this case the original matrix  $A$  is split into a number of horizontal strips, and matrix  $B$  is represented as a set of vertical strips. Band size should be equal to  $d=n/p$  (provided that  $n$  is a multiple of  $p$ ), because this will ensure equal distribution of computational load among processors. This will reduce the amount of communication between these tasks since everything else is handled locally within the task.

#### NOTE

Too much agglomeration is not good either. It's easy to make a short-sighted decision that could limit the scalability of the program. A well-designed parallel program must adapt to changes in the number of processors. Try not to put unnecessary strict limits on the number of tasks in the program. You should design your system to take advantage of more cores as they appear. Make the number of cores an input variable and design based on it.

### 13.3.4 Mapping

The last step in Foster's methodology is the process of assigning each task to a processing unit. Of course, this problem does not arise on single-processor computers or on shared-memory computers whose operating systems provide automatic task scheduling. If we are just writing programs to run on a desktop computer, as in the examples we've shown you throughout the book,

then scheduling isn't something we need to think about. Scheduling does become a factor if we're using a distributed system or specialized hardware with lots of processors for large-scale tasks. We will touch that aspect in the next example.

## The goal

The goal of mapping the algorithm is twofold: to minimize the overall program execution time and optimize resource utilization.

There are two basic strategies for achieving that: Place tasks that can run in parallel on different processors to increase overall concurrency, or focus on placing tasks that often interact with each other on the same processor to increase locality by keeping them close to each other. In some situations, you can use both of these approaches, but more often than not they will conflict with each other, which means you'll have to make design tradeoffs.

Designing a good mapping algorithm depends heavily on both the structure of the program and the hardware on which it runs, and this is unfortunately beyond the scope of this book.

## Example

In our example of matrix multiplication, we delegate the mapping and scheduling of the tasks to the operating system, so it's not our concern.

### 13.3.5 Implementation

There are a few steps left in the design process. First, we need to do some simple performance analyses to choose between alternative algorithms and check that our design meets our requirements and performance goals. We also need to think hard about the cost of implementing our algorithm, the reusability of existing code when we implement it, and how it all fits into the larger systems of which they may become a part. These questions are specific to the use case at hand, and real-world systems will likely bring more complications that need to be considered of a case-by-case basis. Such considerations are also beyond the scope of this book.

The example implementation of the concurrent matrix multiplication:

```
# Chapter 13/matmul/matmul_concurrent.py
from typing import List
import random
from multiprocessing import Pool

Row = List[int]
Column = List[int]
Matrix = List[Row]

def matrix_multiply(matrix_a: Matrix, matrix_b: Matrix) -> Matrix:
    num_rows_a = len(matrix_a)
    num_cols_a = len(matrix_a[0])
    num_rows_b = len(matrix_b)
    num_cols_b = len(matrix_b[0])
    if num_cols_a != num_rows_b:
        raise ArithmeticError(
            f"Invalid dimensions; Cannot multiply "
            f"{num_rows_a}x{num_cols_a}*{num_rows_b}x{num_cols_b}"
        )

    pool = Pool() #A
    results = pool.map( #B
        process_row, #B
        [(matrix_a, matrix_b, i) for i in range(num_rows_a)]) #B
    pool.close()
    pool.join() #C
    return results

def process_row(args: tuple) -> Column: #D
    matrix_a, matrix_b, row_idx = args #D
    num_cols_a = len(matrix_a[0]) #D
    num_cols_b = len(matrix_b[0]) #D

    result_col = [0] * num_cols_b #D
    for j in range(num_cols_b): #D
        for k in range(num_cols_a): #D
            result_col[j] += matrix_a[row_idx][k] * matrix_b[k][j]
    return result_col #D

if __name__ == "__main__":
    cols = 4
    rows = 2
    A = [[random.randint(0, 10) for i in range(cols)] for j in range(rows)]
    print(f"matrix A: {A}")
    B = [[random.randint(0, 10) for i in range(rows)] for j in range(cols)]
    print(f"matrix B: {B}")
    C = matrix_multiply(A, B)
    print(f"matrix C: {C}")
```

```
print(f"matrix B: {B}")
C = matrix_multiply(A, B)
print(f"matrix C: {C}")
```

This program defines a function `matrix_multiply` that takes in two matrices and calculates their product concurrently. It uses process pool to break down the calculation into smaller tasks of calculating individual columns of the solution matrix concurrently. The program collects the results of these tasks and stores them in the result matrix.

This is all cool and all, but all those math problems are already solved by a lot of frameworks and libraries. Let's tackle another problem, a bit more realistic, some big data engineering courses consider it as a "Hello world" application but we will try to do it purely on Python.

## 13.4 Distributed word count

The distributed word-count problem is a classic example of a big data problem that can be solved using distributed computing. The objective is to count the occurrences of each word in a large dataset, typically a text file or a collection of text files. While seemingly simple, this task can become time-consuming and resource-intensive when dealing with massive datasets.

To illustrate the significance of this challenge, consider the infamous incident that occurred during the reprint of the King James Bible in 1631. The printing process involved placing each letter, a total of 3,116,480, carefully in the lower platen of the printing press to create all 783,137 words in the Bible. However, a mistake was made, and the word "not" was omitted from a well-known verse. The resulting work became known as "The Wicked Bible" because in the Ten Commandments, it said, "Thou shall commit adultery." If the printers had a way to automate the counting of all the words and letters that were supposed to be in the final product, the crucial mistake might have been avoided. This incident underscores the importance of accurate and efficient word-counting processes, especially when dealing with large datasets.

99 LITTLE BUGS IN THE CODE...  
TAKE ONE DOWN, PATCH IT AROUND  
127 LITTLE BUGS IN THE CODE...



### OCCURRENCES OF EACH WORD

```
{ "LITTLE": 2, "BUGS": 2, "IN": 2, "THE": 2, "CODE": 2,  
  "TAKE": 1, "ONE": 1, "DOWN": 1, "PATCH": 1,  
  "IT": 1, "AROUND": 1 }
```

As a starting point let's create a simple sequential program first:

```
# Chapter 13/wordcount/wordcount_seq.py  
import re  
import os  
import glob  
import typing as T  
  
Occurrences = T.Dict[str, int]  
  
ENCODING = "ISO-8859-1"  
  
def wordcount(filenames: T.List[str]) -> Occurrences:  
    word_counts = {}  
    for filename in filenames: #A  
        print(f"Calculating {filename}")  
        with open(filename, "r", encoding=ENCODING) as file:  
            for line in file: #B  
                words = re.split("\w+", line) #C  
                for word in words: #D  
                    word = word.lower() #E  
                    if word != "": #F  
                        word_counts[word] = 1 + word_counts.get(w  
    return word_counts  
  
if __name__ == "__main__":  
    data = list(
```

```
glob.glob(f"{os.path.abspath(os.getcwd())}/input_files/*.  
result = wordcount(data)  
print(result)
```

For each file our application will read the text, divide it into words ignoring punctuation and capitalization, and add it to the total count of each word to the dictionary. From each word, it creates a key-value pair (`word, 1`). That is, the word is treated as a key, and the associated value of 1 means that we have seen that word once.

The goal here is to design and build a concurrent program that calculates the number of occurrences of each word in each document, having gigabytes of files and a distributed computer cluster. Let's run through the four stages again, this time using our new problem.

#### NOTE

Word count is a problem that has been used to demonstrate several generations of distributed data engines. It was introduced in MapReduce and then in many others, including Pig, Hive, and Spark.

### 13.4.1 Partitioning

In order to create a solution that associates each word with its frequency in a dataset, we must tackle two main challenges: breaking down the text files into individual words and counting the number of occurrences of each word. The second task depends on completion of the first as we cannot begin counting word occurrences until the text has been divided into individual words. This situation is a prime example of task decomposition, where we can break down the problem into smaller tasks based on their functionality.

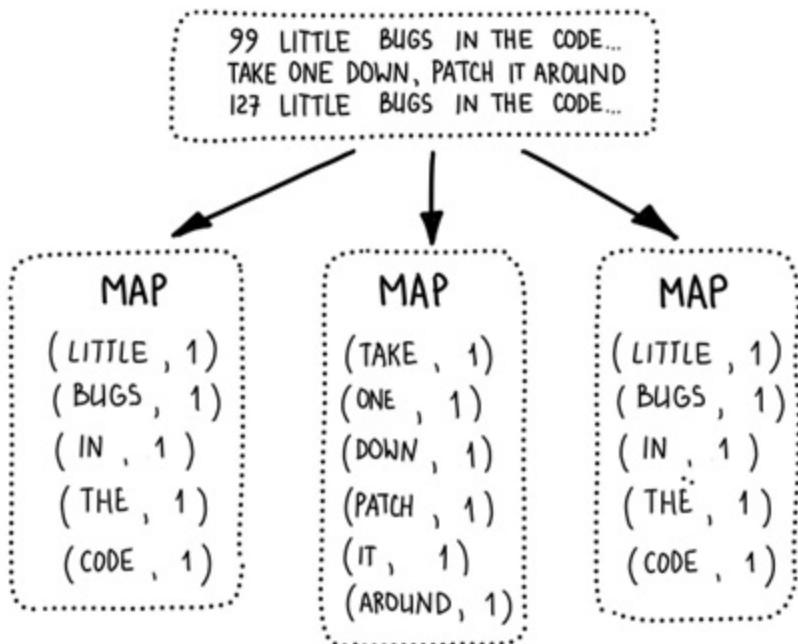
In this approach, the focus is primarily on the type of the task to be performed rather than on the data needed for the computation.

This also looks like a great example of applying the Map/Reduce pattern that we learn from Chapter 7. We can express the computation into two steps or phases: *Map* and *Reduce*.

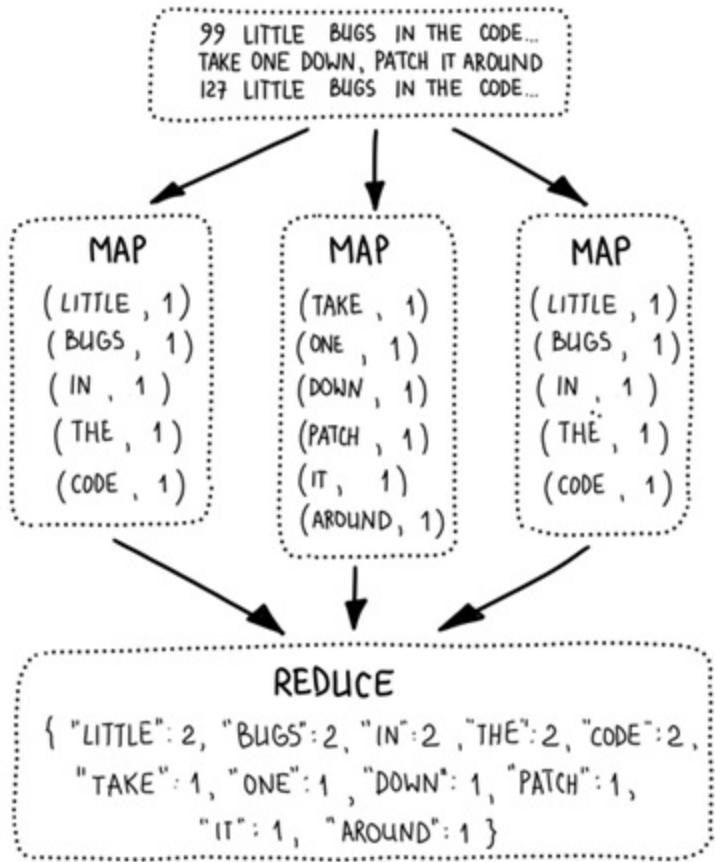


Here the Map phase plays a role in reading the text files and splitting them into word pairs. We can achieve maximum concurrency (what we are looking for in this step) in the Map phase by splitting the input data into multiple chunks. For  $M$  workers, we want to have  $M$  chunks so that each worker has something to work on. The number of workers depends mainly on the number of machines at our disposal.

No matter how complex the data you are trying to process, the Map phase produces events consisting of a key and a value. The key is very important in the next Reduce phase.



Reducer task takes the output from the Map task, a list of key-value pairs, and combines all of the values for each unique key. For example, if the mapper task output is `[("the", 1), ("take", 1), ("the", 1)]`, the reducer task would combine the values for the key "the" to produce the output `[("the", 2)]`. This is known as "reducing" or "aggregating" the data. The Reducer output is a list of unique keys and their associated total count.



We can build the algorithm differently here by creating multiple Reduce tasks and each of them assigning some list of words that it will handle up. It's up to the next steps to decide the best way to implement.

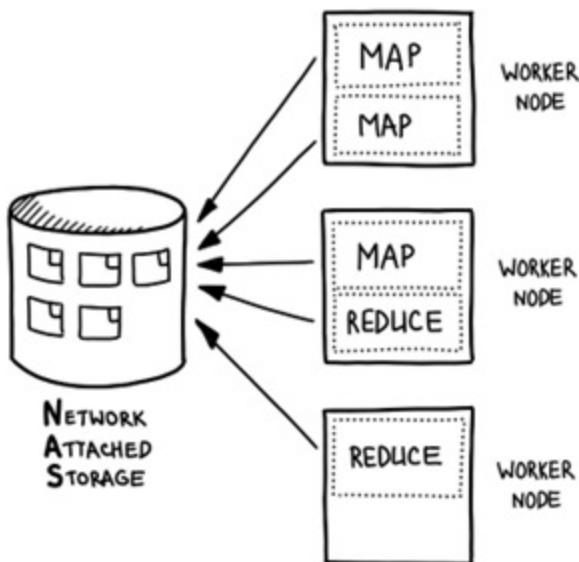
There is no predicting which worker will get which file to read. It can be any file, in any order. This gives our program ample horizontal scaling capability. Just add more worker nodes and we can read more files at the same time. If we had infinite hardware, we could potentially read each file in parallel, reducing data read time to the length of the longest text.

### 13.4.2 Communication

All worker nodes of our cluster are assigned chunks of data to read. In our word count example, imagine that we are reading a gigantic number of text files, such as a complete collection of books, where each book is a separate file.

To store and distribute this text data we can use *NAS* (*Network Attached Storage*). NAS can be described as a combination of a large storage drive and a special hardware platform that allows us to connect this storage drive to the local computer network. This way we do not need to worry about complex communication protocols and each node in the cluster can access files as if they were on a local disk.

The Map and Reduce tasks are expected to run on completely arbitrary machines of our cluster, without any common context. It can be the same machine or completely different machines. This means that all the data that Map phase outputs must be transferred to the Reduce phase and possibly written to disk if they are too big to fit in memory (which is often the case) and we can have several options for that in our case. The first one is IPC with message passing (Chapter 5). Aside from message passing IPC we can also use shared data to store intermediate data Map tasks, and Reducer tasks can use this shared NAS volume. That is what we will do.



Another factor to consider is whether communications will be synchronous or asynchronous.

In synchronous communications all of the tasks involved must wait until the entire communication process is complete before they can continue to do other work. This can potentially cause tasks to spend a lot of time waiting for

data exchange instead of doing useful work.

In asynchronous communications, on the other hand, once a task sends an asynchronous message it can immediately get to other work regardless of when the receiving task receives the message. Also consider the amount of processing overhead that a particular communication strategy entails. After all, CPU cycles spent sending and receiving data are cycles not spent processing them.

For our problem it is beneficial to use asynchronous communication as we have long-running tasks that do not require to block the execution and we will have a lot of communication between them.

### **13.4.3 Agglomeration**

Right now, each of our Map task yields word pairs (`word, 1`). A very easy way to speed things up is to pre-aggregate those pairs locally on each Map task, before the Map phase ends and the Reduce phase begins. This step, known as *Combine* is very similar in nature to Reduce. It takes an arbitrary list of intermediate key/value pairs grouped by key, performs a value aggregation operation (if possible), and outputs fewer key/value pairs. In other words, it can opportunistically pre-aggregate some of the intermediate values to reduce the communication overhead between Map tasks and Reduce tasks.

Also, going back to our previous thoughts about the number of Reduce tasks to simplify the algorithm, we will use only one Reduce task – we will agglomerate all the Reduce tasks into one big Reduce task. This won't be as much data to compute since we just added the Combine task, so that should be fine.

### **13.4.4 Mapping**

After the agglomeration phase, we are in the state of a composer who has prepared everything to perform his symphony. But the beautiful sound of the orchestra is only possible with a conductor who coordinates the individual musicians and brings his own style to the performance. Yes, we are talking

about scheduling our tasks on the actual processing resources that we have.



The most important (and complex) aspect of the task scheduling algorithm is the strategy used to distribute tasks among workers. Typically, the strategy chosen is a compromise between the conflicting demands of independent work (to reduce communication costs) and global knowledge of the state of computation (to improve load balance).

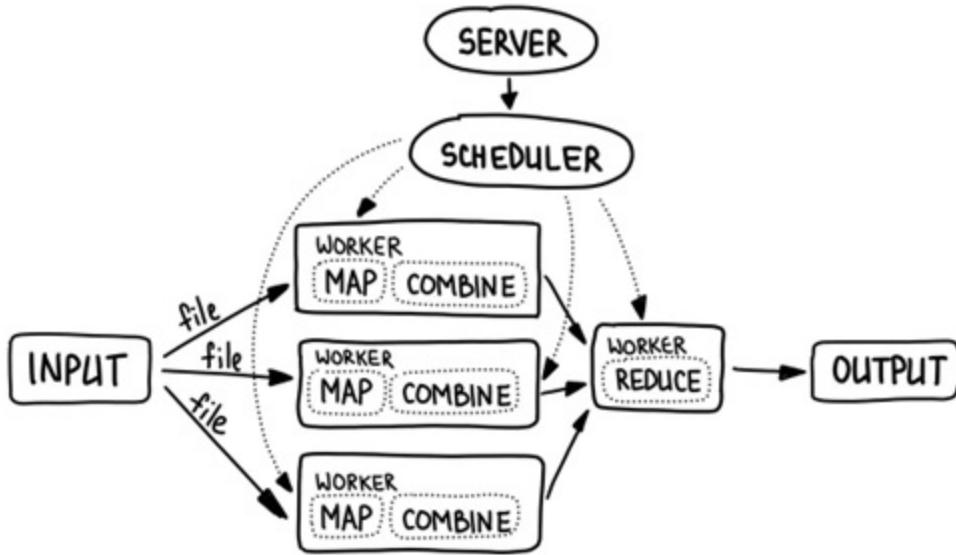
We will implement the simplest idea – a central scheduler. The central scheduler is responsible for sending tasks to workers, tracking progress, and returning results. It selects idle workers and assigns them either a Map task or a Reduce task. When all workers have finished their Map task, the scheduler notifies them to start the Reduce task (in our case it will be only one worker).

Each worker repeatedly requests and completes the task from the scheduler and returns the results of the work to him. The efficiency of this strategy depends on the number of workers and the relative costs of receiving and completing the tasks. We will use a somewhat complex strategy of dynamic task allocation because we do not know in advance the number of file and their sizes. Therefore, we cannot guarantee optimal task allocation before the job starts.

### 13.4.5 Implementation

The diagram below provides an overview of the entire program in action. The Server initiates the execution and creates a central scheduler. Each Map

worker is assigned a file for processing. If there are more files than workers, a worker will be assigned another file once it has finished processing. Before completing the Map task, Combine tasks are triggered to aggregate the output of the Map tasks, thereby reducing communication overhead. Once the Map phase is finished, the Scheduler will commence the Reduce phase, in which all the Map outputs will be combined into a single output.



Our main server functionality:

```

# Chapter 13/wordcount/server.py
import os
import glob
import asyncio

from scheduler import Scheduler
from protocol import Protocol, HOST, PORT, FileWithId

class Server(Protocol):
    def __init__(self, scheduler: Scheduler) -> None:
        super().__init__()
        self.scheduler = scheduler

    def connection_made(self, transport: asyncio.Transport) -> No
        peername = transport.get_extra_info("peername") #A
        print(f"New worker connection from {peername}") #A
        self.transport = transport #A
  
```

```

        self.start_new_task() #A

def start_new_task(self) -> None: #B
    command, data = self.scheduler.get_next_task() #B
    self.send_command(command=command, data=data) #B

def process_command(self, command: bytes,
                    data: FileWithId = None) -> None:
    if command == b"mapdone":
        self.scheduler.map_done(data)
        self.start_new_task()
    elif command == b"reducedone":
        self.scheduler.reduce_done()
        self.start_new_task()
    else:
        print(f"Unknown command received: {command}")

def main():
    event_loop = asyncio.get_event_loop() #C
    current_path = os.path.abspath(os.getcwd())
    file_locations = list( #D
        glob.glob(f"{current_path}/input_files/*.txt")) #D
    scheduler = Scheduler(file_locations) #E
    server = event_loop.create_server(lambda: Server(scheduler),
    server = event_loop.run_until_complete(server) #G
    print(f"Serving on {server.sockets[0].getsockname()}") #H
    try: #H
        event_loop.run_forever() #H
    finally: #H
        server.close() #H
        event_loop.run_until_complete(server.wait_closed()) #H
        event_loop.close()#H

if __name__ == "__main__":
    main()

```

This is Server – our main execution process that is responsible for communication with every worker process, it is also calling Scheduler for getting the next task for each worker and coordinate the Map and Reduce phases.

Our worker functionality:

```
# Chapter 13/wordcount/worker.py
import re
```

```

import os
import json
import asyncio
import typing as T
from uuid import uuid4

from protocol import Protocol, HOST, PORT, FileWithId, \
    Occurrences

ENCODING = "ISO-8859-1"
RESULT_FILENAME = "result.json"

class Worker(Protocol):
    def connection_lost(self, exc): #A
        print("The server closed the connection") #A
        asyncio.get_running_loop().stop() #A

    def process_command(self, command: bytes, data: T.Any) -> Non
        if command == b"map":
            self.handle_map_request(data)
        elif command == b"reduce":
            self.handle_reduce_request(data)
        elif command == b"disconnect":
            self.connection_lost(None)
        else:
            print(f"Unknown command received: {command}")

    def mapfn(self, filename: str) -> T.Dict[str, T.List[int]]: #
        print(f"Running map for {filename}") #B
        word_counts: T.Dict[str, T.List[int]] = {} #B
        with open(filename, "r", encoding=ENCODING) as f: #B
            for line in f: #B
                words = re.split("\W+", line) #B
                for word in words: #B
                    word = word.lower() #B
                    if word != "": #B
                        if word not in word_counts: #B
                            word_counts[word] = [] #B
                        word_counts[word].append(1) #B
        return word_counts #B

    def combinefn(self, results: T.Dict[str, T.List[int]]) -> Occ
        combined_results: Occurrences = {} #C
        for key in results.keys(): #C
            combined_results[key] = sum(results[key]) #C
        return combined_results #C

    def reducefn(self, map_files: T.Dict[str, str]) -> Occurrence

```

```

reduced_result: Occurrences = {} #D
for filename in map_files.values(): #D
    with open(filename, "r") as f: #D
        print(f"Running reduce for {filename}") #D
        d = json.load(f) #D
        for k, v in d.items(): #D
            reduced_result[k] = v + reduced_result.get(k,
return reduced_result #D

def handle_map_request(self, map_file: FileWithId) -> None:
    print(f"Mapping {map_file}")
    temp_results = self.mapfn(map_file[1]) #E
    results = self.combinefn(temp_results) #F
    temp_file = self.save_map_results(results) #G
    self.send_command( #H
        command=b"mapdone", data=(map_file[0], temp_file)) #H

def save_map_results(self, results: Occurrences) -> str: #I
    temp_dir = self.get_temp_dir() #I
    temp_file = os.path.join(temp_dir, f"{uuid4()}.json") #I
    print(f"Saving to {temp_file}") #I
    with open(temp_file, "w") as f: #I
        d = json.dumps(results) #I
        f.write(d) #I
    print(f"Saved to {temp_file}") #I
    return temp_file #I

def handle_reduce_request(self, data: T.Dict[str, str]) -> No
    results = self.reducefn(data) #J
    with open(RESULT_FILENAME, "w") as f: #K
        d = json.dumps(results) #K
        f.write(d) #K
    self.send_command(command=b"reducedone", #L
                      data=("0", RESULT_FILENAME)) #L

def main():
    event_loop = asyncio.get_event_loop()
    coro = event_loop.create_connection(Worker, HOST, PORT)
    event_loop.run_until_complete(coro)
    event_loop.run_forever()
    event_loop.close()

if __name__ == "__main__":
    main()

```

Workers during the Map phase invoke the `mapfn` function to parse the data, then invokes `combinefn` function to merge the results and write intermediate

(key, value) results. Worker during the reduce phase gets the intermediate data and then calls the function `reducefn` once for each unique key and gives it a list of all values that were generated for that key. It then writes its final output to a single file that the user's program can access once the program has completed.

Our scheduler implementation:

```
# Chapter 13/wordcount/scheduler.py
import asyncio
from enum import Enum
import typing as T

from protocol import FileWithId

class State(Enum):
    START = 0
    MAPPING = 1
    REDUCING = 2
    FINISHED = 3

class Scheduler:
    def __init__(self, file_locations: T.List[str]) -> None:
        self.state = State.START
        self.data_len = len(file_locations)
        self.file_locations: T.Iterator = iter(enumerate(file_locations))
        self.working_maps: T.Dict[str, str] = {}
        self.map_results: T.Dict[str, str] = {}

    def get_next_task(self) -> T.Tuple[bytes, T.Any]:
        if self.state == State.START:
            print("STARTED")
            self.state = State.MAPPING

        if self.state == State.MAPPING:
            try: #A
                map_item = next(self.file_locations) #A
                self.working_maps[map_item[0]] = map_item[1] #A
                return b"map", map_item #A
            except StopIteration: #A
                if len(self.working_maps) > 0: #A
                    return b"disconnect", None #A
                self.state = State.REDUCING #A

        if self.state == State.REDUCING:
            return b"reduce", self.map_results
```

```

if self.state == State.FINISHED:
    print("FINISHED.")
    asyncio.get_running_loop().stop()
    return b"disconnect", None

def map_done(self, data: FileWithId) -> None: #B
    if not data[0] in self.working_maps: #B
        return #B
    self.map_results[data[0]] = data[1] #B
    del self.working_maps[data[0]] #B
    print(f"MAPPING {len(self.map_results)}/{self.data_len}")

def reduce_done(self) -> None: #C
    print("REDUCING 1/1") #C
    self.state = State.FINISHED #C

```

This is the central scheduler; in our implementation it is divided into several states. Start state – where it initializes the necessary data structures. The Mapping state – where it distributes all Map tasks. Each task is a separate file, so when the server requests the next task, the scheduler simply returns the next unprocessed file. Reducing state – where it stops all but one workflow for a single Reduce task. Finished state – simply to stop the server and therefore the program.

#### **NOTE**

For testing I have used books from Gutenberg (<https://www.gutenberg.org/help/mirroring.html>) and the overall system was able to work quite fast for a couple of gigabytes of data.

## **13.5 Recap**

- In the first 12 chapters we laid out a puzzle called concurrency. This chapter connected all the pieces of knowledge we learned so far.
- We described an approach to designing concurrent systems and we've illustrated its application to a couple of problems.
  - Before starting to write a concurrent program, the developer first examines the problem to be solved and makes sure that the effort to create a concurrent program is justified by the task at hand.

- The next step is to make sure that the problem can be divided into tasks. and
- Make communication and coordination of the tasks possible.
- In the third and fourth steps, the abstract algorithm becomes tangible and efficient by considering the class of parallel computers on which it is to run. Is it a centralized multiprocessor or a multicomputer? What communication paths are supported? How should we combine tasks to efficiently distribute them across processors?

## 13.6 Epilogue

Throughout this book, we've utilized a variety of abstract concepts to illustrate the intricacies of designing concurrent systems. From symphony orchestras to hospital waiting rooms, fast food processes to home maintenance, we've drawn comparisons to help readers understand complex topics. While we acknowledge that this book only scratches the surface of this vast field, even this nominal level of detail emphasizes several strategies for developing concurrent applications.



These 13 chapters should give you a solid foundation to go deeper into the field of concurrency. There is still a lot to discover there! Now, hit it! (♪♪♪ background drums and rock music ♪♪♪).

[1] Ian Foster, “Designing and building parallel programs”,  
<https://www.mcs.anl.gov/~itf/dbpp/>