Tom Johnson

THE COMPUTER SCIENCE BOOK

A complete introduction to computer science in one book.

The Computer Science Book

A complete introduction to computer science in one book

Tom Johnson

This book is for sale at http://leanpub.com/computer-science

This version was published on 2021-03-04

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

To my wife Georgie, for her unceasing support.

Contents

Introduction

Welcome to *The Computer Science Book*! This book contains ten chapters covering the main areas of a computer science degree. Together they will give you a comprehensive introduction to computer science.

I learned to program through self-study and then attending a bootcamp. My bootcamp did a wonderful job of preparing me for work but there simply wasn't time to dig under the surface of web programming. When I started my first developer job I was painfully aware of how little I knew. There were things I'd heard of but had never had time to investigate properly. That was fine – at least I knew about them. Pausing to think more deeply, I realised I was programming this thing that was mostly a black box. Who knew what was in it? How did my code actually get to the processor and how did the processor know what to do with it? How did the server know when requests came and how did they get there? It felt like I was building a career on sand.

Clearly, the solution was to study the computer science fundamentals I was missing. In some ways computer science is very easy to study independently. There is a huge wealth of resources freely available on the Internet. But that abundance can be paralysing if you don't know where to start. Which textbooks are the most useful? Do I really need to know everything they teach or are some bits not so relevant now? In what order should I study topics?

The Computer Science Book is intended to guide you through the field of computer science. The ten chapters chart a sensible route through the subject, each one building on the concepts introduced in the preceding. Each chapter is focused on delivering the essential knowledge that will help you improve as a developer. I've generally erred on the side of the practical rather than the theoretical. Nevertheless in some places the abstract theory is unavoidable – but very interesting! Each chapter also includes a further reading section giving suggestions for independent study and introducing deeper topics that didn't fit in the chapter.

Modern Operating Systems is 1136 pages long. *Database Systems: The Complete Book* is 1140. *TCP Illustrated* is 1060 (just the first volume!). This single volume can never match such textbooks in depth or comprehensiveness. To cover a full computer science course it must be very selective. I've chosen to focus on topics and concepts that I've encountered in my programming career – things that I know are important. My intention is not to cover absolutely everything you might need to know but to chart the unknown and give you the necessary knowledge and confidence to explore further as your curiosity desires.

I hope you enjoy reading *The Computer Science Book* and that you find it useful. If you have any suggestions, find anything unclear or come across any errors please do let me know.

Theory of computation

Introduction

We begin our exciting journey through computer science by looking at just what it is computers actually do: computation. The branch of computer science that explores the capabilities and limitations of computation is known as **theory of computation** or computational theory. It builds and analyses mathematical models to probe computation in the abstract.

I promised lots of practical computer science information in this book. So why are we starting off with all this abstraction? A good craftsman should have a deep understanding of how their tools work. As programmers, we structure computation in order to achieve results. The better we understand what our computers can and can't do, the better programmers we will be. You don't want to waste a week trying to solve a problem that's been proven to be unsolvable! Secondly, the terms and concepts from theory of computation pop up surprisingly often in day-to-day programming and it's useful to have at least an acquaintance with them.

Finally, theory of computation is where computer science anchors itself to mathematics, logic and philosophy. It provides the foundations for everything else and will help you to develop a much more sophisticated understanding of what computing is and *why* computers work. It's super interesting!

Because of the mathematics, theory of computation has a reputation for being very abstract and dense. Certainly it's very maths heavy and the textbooks contain plenty of proofs. If you're not mathematically confident, don't let that hold you back! Lots of important results are straightforward to grasp without any mathematical background.

Theory of computation is made up of three main areas: **automata theory**, **computability theory** and **complexity theory**.

Automata theory is all about using mathematics to create computational models and explore what they can do. The reason for doing this is that physical computers are hugely complex devices that vary wildly in their design and performance. By using mathematical models, we can strip away all of these superfluous details and better understand the capabilities of the underlying model. We'll see that an surprisingly simple model of computation is capable of representing any computation.

Once we have the necessary mathematical models we can use them to explore the capabilities and limitations of computation itself. This is computability theory. As we'll see, there are some surprising limits to what can be computed. Besides being a fascinating intersection of computing and philosophy, you'll regularly hit against these limitations in everyday programming so it's vital that you're aware of them.

After those two sections, we'll be close to having a solid theoretical base. We'll still need to sort out a few simple concepts: time and space. Complexity theory is the branch of computational theory

that investigates the complexity of algorithms by analysing the resources, primarily time and space, that they require. It gives us ways to understand and measure algorithmic performance and how to classify algorithms according to their complexity.

Automata theory

Let's begin by asking ourselves: what do we actually mean by the words "computation" and "computer"?

In mathematics we don't really care how results are worked out. I can assign the square root of a number to a variable and blithely use that variable in equations without worrying about how to actually calculate the square root. My calculator doesn't have it so easy. It has to know some sequence of steps that will calculate the square root so that it can show me the result. My calculator is performing computation.

A computer, then, is any machine that performs a computation by executing a defined sequence of operations, known as an **algorithm**. It may optionally generate some kind of result at the end. The particular state of the machine at any given moment indicates the state of the computation. This is a really important concept. A computer is something that takes a list of operations and converts them into some kind of observable form. In my trippier moments I like to imagine that a computer takes a *verb* (the instruction) and converts it into an *object* (the state) that represents the outcome of that instruction. The next instruction is then executed against the object, generating a new object that forms the input for the next instruction and so on. Far out!

Note that nothing in this definition requires the computer to be an actual, physical device. We could represent a computer by drawing on paper some states and arrows specifying rules for moving between them. We'll see diagrams like this below. When defined with a more robust mathematical notation these models are known as **automata**, from the Greek *automatos* meaning "self-acting".

Automata theory determines the capabilities and limitations of various automata designs. In our exploration of the subject we'll start with the most basic model: the finite automaton. We'll see that it can do a surprising amount but is ultimately limited by its simple design. We'll then see how to amend its design to create the more powerful push-down automaton. Finally, we'll look at the most powerful automaton of all: the Turing machine.

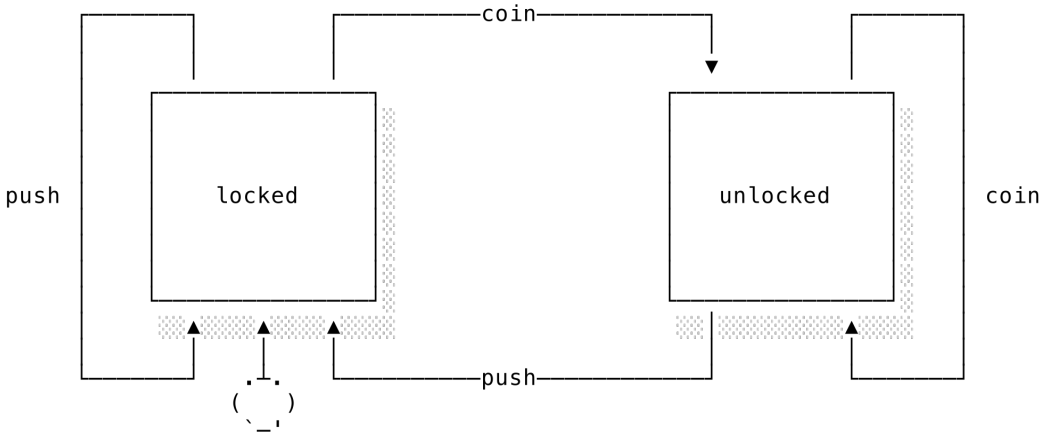
The point of all this is not to come up with a blueprint for a real computer. The point is to find a mathematical model that is both simple to understand yet powerful enough to model any arbitrary computation. Spoiler: such a thing exists and is the Turing machine. Once we have a model for performing arbitrary computation, we can analyse that model to better understand computation itself. Our aim here is to put together the necessary tools so that we can study computation in the abstract.

Finite automata

We have already defined a computer as a machine that can transition through a series of states, each reflecting a step in some algorithm. If there is only a finite number of states and a finite number

of transitions, then such a machine is known as a **finite automaton** or **finite state machine** (FSM). The finite automaton is the simplest kind of computational model. It's very useful for modelling primitive devices.

For example, this state diagram represents a turnstile. You may not have thought of a turnstile as a computer but look carefully and you'll see that it meets our definition above:



A state diagram for a turnstile

We designate one state to be the **start state**. This is marked on the diagram by a circle with an arrow pointing to `locked`. This is the state our turnstile sits in, waiting for something to happen. At each state there are two possible inputs: `push` and `coin`. Pushing a locked turnstile doesn't do anything – it remains locked. We represent this with an arrow looping back to the same state. Inserting a coin causes the turnstile to **transition** to the `unlocked` state. Inserting more coins has no effect. As the turnstile is now unlocked, a person can push through, thus resetting the turnstile back to its original `locked` state.

The possible ways of moving between states are defined by the **transition function**:

```
1  transition :: (oldState, input) -> (newState, output?)
```

I'm using Haskell-like type signature syntax here. The double colons mean "is of type", so `transition` is a function that takes a state and an input and outputs a new state and an output. The question mark on `output` is my way of indicating that it's optional.

Laying out all of the valid transitions creates a representation of the FSM known as a **state transition table**:

Start state	Input	New state	Output
locked	coin	unlocked	lock mechanism opens
locked	push	locked	
unlocked	coin	unlocked	
unlocked	push	locked	lock mechanism closes

Even this very simple automaton has a limited form of memory. If the automaton is in the `unlocked` state we know that the previous input was a coin. But once we transition back to `locked` we lose this information. We have no way of telling whether a locked turnstile has never been unlocked or whether it's been unlocked thousands of times.

A machine is **deterministic** if it consistently gives the same output for a given input. This is generally a desirable quality. An automaton is a **deterministic finite automaton** (DFA) if there is only ever a single possible transition from each state for a given input. You can see from the state diagram above that the turnstile is deterministic because each input only appears on a single transition at each state.

We can define a **non-deterministic finite automaton** (NFA) simply by allowing multiple transitions from a state for a given input. This requires a minor change to the transition function:

```
1 outcome :: (newState, output?)
2 transition :: (oldState, input) -> [outcome1, outcome2...]
```

For each pair of state and input there may be zero or more transitions. Each transition outputs a new state and optional output. On a state machine diagram this is represented simply by having multiple labels on transition arrows. This raises an obvious question: if a NFA has two available transitions for a given input, how does it choose which transition to take?

The surprising answer is that the NFA follows every possible transition. Remember that we're talking about mathematical models here. Each time it is faced with multiple transitions, the NFA duplicates itself and each duplicate follows a different transition. Duplicates can in turn duplicate themselves whenever they encounter more transitions. The NFA works a bit like the broomstick from the Sorcerer's Apprentice, duplicating and reduplicating itself as necessary to cover every possible path.

The duplicates all run in parallel, each one following a different computational path. If one of the duplicates reaches a state where it has no more open transitions, then it's stuck in a dead end and can be removed. We'll end up with a whole herd of automata, representing the complete set of possible paths through the transitions. If one of them reaches an accept state, the path it represents is a valid path. We say that the automaton has **accepted** the input if there is at least one valid path.

This definition applies to deterministic automata too. A DFA is a NFA that just happens to only ever have one possible transition for a given input. This means that a DFA is also a valid NFA. What's more, any NFA can be represented as a valid DFA. In fact, deterministic and non-deterministic automata are equivalent. Anything that can be computed with a DFA can be computed with an NFA and vice versa. Given this, it seems that NFAs add quite a lot of complexity for a questionable advantage. Why bother? The benefit is that some computations can be more concisely expressed (using fewer states) as a non-deterministic automaton than as a deterministic automaton.

Finite automata, whether deterministic or non-deterministic, are very simple beasts. In fact, they're so basic that you might at first not even recognise what they're doing as computation. It appears that finite automata are nothing more than a starting point on our journey. Yet they have another

trick up their sleeve. Finite automata pop up surprisingly often *embedded* in other programs. I'll refer to code versions of finite automata as state machines. They will appear in codebases you work on, either explicitly or implicitly defined. They're super useful for modelling processes that flow through a series of states. When you recognise that you've got an implicit state machine lurking in your code, pulling it out into an explicit structure can be a great way to manage complexity. This is called the **state machine pattern**.

Take a complex, multiple step user flow. A good example would be Airbnb's identify verification flow. Each step in the flow can be represented by a state in a state machine. The actions in the flow are the transitions between states. You might have a state in which you ask the user for basic contact information. When they input their details, the state machine transitions to another state in which the user is asked to upload a photo of their identity card. The upload triggers another transition to a state in which the photo is sent to an analysis service that compares the identity card's details to the information provided by the user in the first state. If everything matches, the state machine transitions to a success state. If there's a problem, the state machine transitions to an error-handling state. The user is informed and given the option to repeat the upload process.

It's possible to do all of the state management and validation manually but it's brittle and error-prone. Let's see how the above flow might be implemented without a state machine and then see whether a state machine improves things:

```
class User
  enum status: [:new, :info_provided, :pending, :verified, :rejected]
end

class UserSignupFlow
  def new(user)
    @user = user
  end

  def perform
    if !@user.status == :info_provided
      @user.contact_info = get_contact_info
      @user.status = :info_provided
    end

    if !(@user.status == :pending || @user.status == :verified)
      @user.photo = get_id_photo
      @user.status = :pending
    end

    if IDValidator.valid?(photo, contact_info)
      @user.status = :verified
    else
```

```
      @user.status = :rejected
    end
  end
end
```

That looks like a mess and I haven't even handled retrying the photo upload. Admittedly, the code could be tidied up a little by extracting things out into methods and so on. Either way, you can't get around the fact that `perform` has to check the `status` attribute of the `user` to determine if an action is permitted. In a small flow this is probably manageable, especially with good unit testing. However, it will be very easy to cause subtle errors by performing an incorrect check. This will lead to the user incorrectly transitioning to an invalid state that might not manifest itself until much later in the program. The transitions are not explicitly defined and instead have to be inferred from the conditional clauses, which can easily be incorrectly modified. This is a design that will not scale well at all.

The use of the `status` attribute is a sure sign that we have an implicit state machine on our hands. Let's make it explicit:

```
class UserSignupFlow
  include AASM

  aasm do
    state :new, initial: true
    state :info_provided
    state :pending
    state :verified
    state :rejected

    event :get_info
      transitions from: :new,
                  to: :info_provided,
                  before: :get_contact_info
    end
    event :get_photo
      transitions from: [:info_provided, :rejected],
                  to: :pending,
                  before: :get_id_photo
    end
    event :validate
      transitions from: :pending, to: :verified do
        guard do
          IDValidator.valid?(photo, contact_info)
        end
```

```
26        end
27        transitions from: :pending,
28                    to: :rejected,
29                    after: :notify_user
30      end
31    end
32  end
```

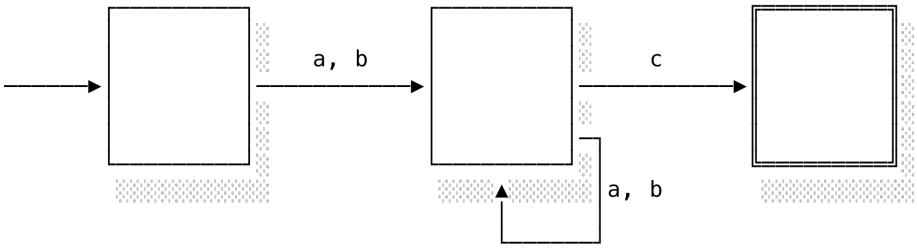
Here `UserSignupFlow` is re-implemented using the AASM[1] gem in Ruby. Other implementations are available and will vary syntactically somewhat but the fundamentals should remain the same. AASM is neat because it provides callbacks (`before`, `guard`, `after` etc.) that allow you to schedule behaviour around transitions. AASM will also raise an error if we attempt to perform an event on invalid state. Such features give us reassurance that, for example, the user will never end up verified without first having their photo and contact information validated. In a real world application it would be trivial to refactor this by adding more states and transitions, creating more complex events and so on. Things are much easier to reason about because we explicitly model the possible states and valid transitions between them.

Regular expressions

There is one other hugely important use of finite state machines that I haven't touched on yet: regular expressions (regexes).

Recall that a finite automaton can define accept states. If an input causes the automaton to finish on an accept state, then the input is accepted. Another way of phrasing this is to say that the automaton **recognises** the input. This is exactly what regular expressions do! Whenever you write a regex you are actually using very terse syntax to define a tiny finite state machine that will recognise, or *match*, particular strings.

Take the regex `(a|b)+c`. We can express this as a NFA that accepts at least one `a` or `b` and then a single `c`. The double border identifies the accept state:



A finite automaton for `(a|b)+c`

We refer to the set of valid inputs (e.g. `ac`, `bc`, `abac`) as the **language** of the automaton. Languages that can be recognised by finite state machines are called **regular languages**, from which we get the term "regular expression".

[1] https://github.com/aasm

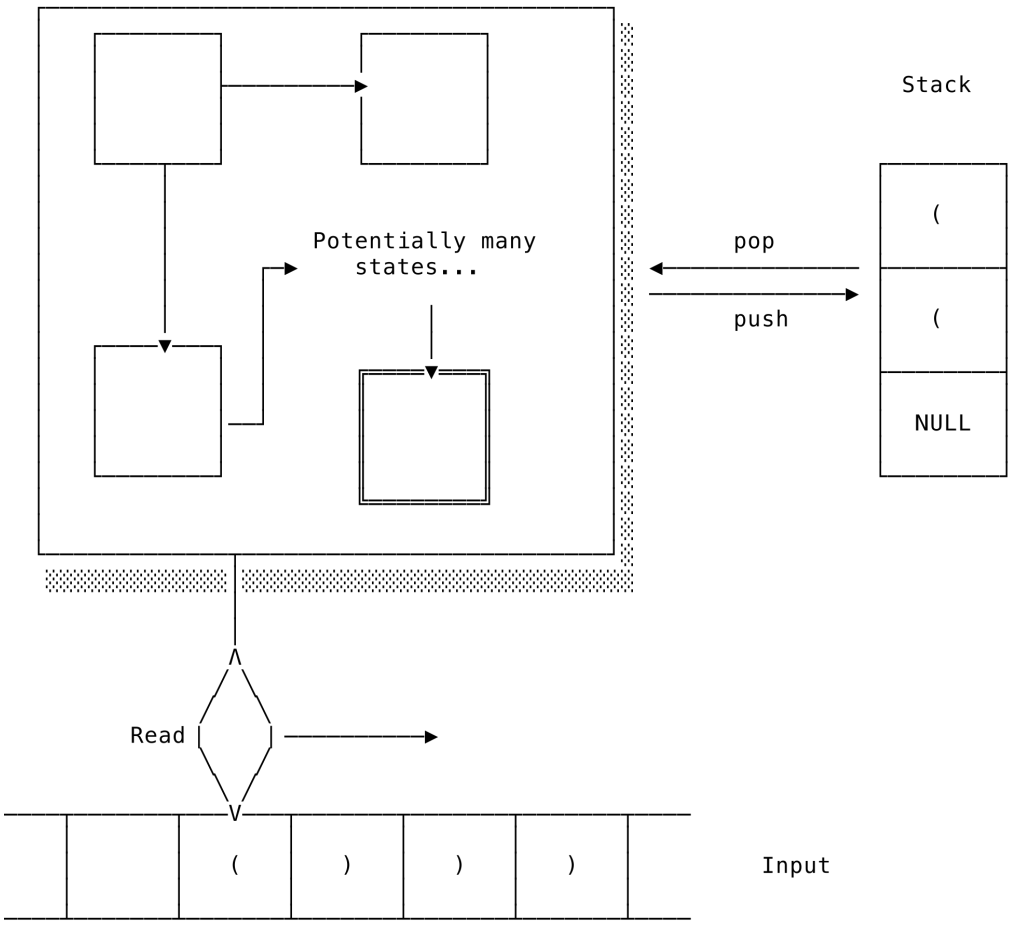
Many theory of computation textbooks define automata in terms of what languages they can recognise. Languages and their automatons are more or less equivalent – one can be defined in terms of the other. The more complex the automaton, the more sophisticated the class of languages it can recognise. For now, it suffices to know that finite automata can only recognise regular languages. They are at the bottom of the hierarchy.

I think it's pretty cool that we've only looked at the simplest automaton and we can already understand how regexes are constructed. Next time you write a regex, try and work out how it could be expressed as a finite automaton. Sadly, this neat pedagogical picture is complicated by reality. Many modern regex engines include features (e.g. back referencing) that allow them to accept non-regular languages. Why would they need to do this? Though finite automata are surprisingly capable little creatures, their simple structure does impose sometimes significant limitations.

For instance, we cannot create a regular expression (i.e. define a finite state machine) that tells us whether a given string has a balanced number of parentheses. What we want is an automaton that accepts an input like `(((()(())))` but rejects `((()))`. It's easy to sketch out a state machine that can recognise our example input (try it yourself) but we cannot create a finite automaton that accepts every valid input. The problem is that the number of parentheses is potentially infinite, meaning that we'd need to have an infinite number of states. This is obviously incompatible with it being a *finite* automaton. We need some way for the automaton to "remember" what it has seen before without having to encode it as a state.

Push-down automata

As we saw previously, a finite automaton has no memory beyond its current state. To recognise matching parentheses we need some kind of scratch space where the automaton can keep track of how many parentheses it has seen so far. This requires a new, more complex model of computation known as a **push-down automaton**:



A push-down automaton

A push-down automaton is a finite automaton combined with a **stack**: a last-in, first-out data structure. Think of a stack of trays in a cafeteria. Clean trays get put on top of the older ones and the next customer to come along will take a tray from the top. The last tray to be added is the first to be taken. We can interact with the stack via two operations. We can **push** a value on to the top of the stack. We can also **pop**, or remove, the newest value off the stack. Doing so exposes the second-newest value as the new top. The number of items that can be pushed on to the automaton's stack is infinite. The set of values that can be written to the stack is called the **stack alphabet**. We need to start the stack off with an **initial value** from this alphabet.

We must update the transition function to enable access to the stack. We can read the value on the top of the stack and optionally perform an operation on the stack:

```
1  stackOperation :: Pop | Push
2  transition :: (state, input, topOfStack) -> (state, stackOperation)
```

Equipped with this new functionality we can make short work of checking for balanced parentheses. Every time the input is a `(` we push a `(` on to the stack. Whenever we come to a `)`, we check that

the top of the stack is a `(` and pop it off. If we encounter a `)` or an empty stack, then we know that the parentheses are unbalanced. If we get to the end of the input and have an empty stack, then we move to an accept state and recognise the input as valid.

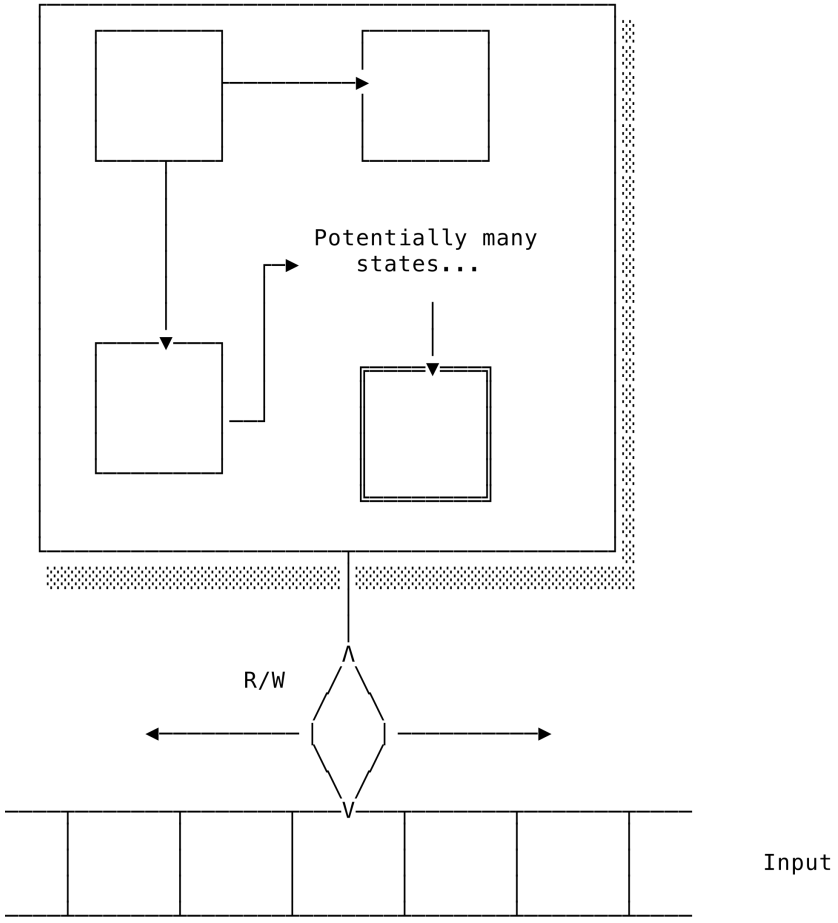
In the diagram above, the input is recorded as symbols on the tape along the bottom of the diagram. The tape is read from left to right. The stack that the finite automaton can manipulate is on the right. Note that the values of the stack alphabet don't have to match the input alphabet. They happen to do so in this case but it's not required. The diagram shows a computation in progress and so there are a couple of parentheses already on the stack. Will this particular example end up in an accept state? Well, so far we can see two `(` have already been pushed and there is another about to be read from the input tape. After that we can see three `)`. Assuming that the input ends after these three values, the automaton will pop three `(` off the stack, leaving an empty stack at the end of input. The input is valid.

The stack is a very simple data structure but it gives the automaton an unlimited amount of memory to work with. We can use this to keep track of what the automaton has already seen without requiring explicit states for each possible input. This is from where the push-down automaton derives its additional computational power.

Turing machines

Though useful, a stack is still limited in its own way. We can only ever read the top-most value. When we want to get at a value further down the stack we need to pop off all of the values on top. This entails discarding the values since we have nowhere else to store them. What we would really like is a way to read or write to *any* location in memory. We need a more complex automaton.

In 1936, before any physical computer had ever been built, the mathematician Alan Turing published a paper describing an automaton that meets our requirements. It consists of a finite automaton "control unit" that reads instruction symbols from an infinitely long tape, just like our push-down automaton above. The two crucial differences are that the tape head can move left as well as right and that it can write values as well as read them. Turing called this automaton an "a-machine" (for "automatic machine"). Nowadays it is more commonly known as the **Turing machine** in his honour.



A Turing machine

The machine starts with the tape head at the beginning of the instruction sequence. The rest of the tape is blank. At each step of the computation the machine reads the current value under the tape head, updates the current state, writes a value to the current tape position and then moves a single step to the left or right. The transition function therefore looks like this:

```
1 shiftDirection :: Left | Right
2 transition :: (state, tapeValue) -> (state, newTapeValue, shiftDirection)
```

Note that `newTapeValue` might be identical to `tapeValue`, resulting in no change to the tape. We say that the machine **halts** if it eventually moves into an accept or reject state. If it doesn't halt, it will loop indefinitely.

What can we do with these Turing machines? On the surface it seems simpler than the push-down automaton because it does away with the separate stack. In fact, having read/write access to any arbitrary location on the tape makes the Turing machine much more powerful. One of Turing's great achievements was to prove that the Turing machine is *the* most powerful automaton possible. If a solution to a problem can't be computed on a Turing machine, there is simply nothing else that

might be able to compute it. Therefore, we can actually use Turing machines to define computation. A problem is computable only if it can be solved on a Turing machine. This is known as the **Church-Turing thesis** after Alonzo Church and Turing, who both independently found the same results (see the further reading for more on Alonzo Church). The implication of the thesis is that every computable problem has a corresponding Turing machine. Discoveries like this are why automata theory is worth studying. It gives us direct insight into the nature of computation itself.

Yet how was Turing so sure that there is nothing more powerful than his machine? Recall that the transition function takes the current state and tape value and outputs a new state, new tape value and shift direction. As with a finite automaton, the transition functions for each state and tape value pair form a state table that defines the machine's operation. If we wanted to change how the machine operated, we would need to change the transition functions. We can't because these are hard coded into the structure of the finite automaton control unit. Changing this would effectively create an entirely new machine.

There's a better way. Turing demonstrated that it is possible to encode the structure of the control unit as a string. That means that it can be written on the input tape. We can therefore encode a "virtual" Turing machine on to the tape along with the instruction symbols. This tape can be read by a second, "real" Turing machine with a control unit designed to decode and emulate the virtual Turing machine. The real Turing machine shuttles back and forth between the instruction symbols and the virtual Turing machine definition, executing the instructions just as the virtual machine would have done. What we end up with is a Turing machine that emulates the behaviour of the virtual one:

```
Host(Virtual, input) accepts if Virtual(input) accepts
Host(Virtual, input) rejects if Virtual(input) rejects
Host(Virtual, input) loops if Virtual(input) loops
```

Such a Turing machine is known as a **universal Turing machine**. Thanks to the Church-Turing thesis, we know that every computable problem has a Turing machine. A machine that can emulate any Turing machine can therefore emulate a Turing machine solving any computable problem, which is the same as actually solving it. Therefore any computable problem can be solved by a universal Turing machine.

Bear in mind here that we're talking about the capabilities of mathematical models. It's certainly possible to create a physical computer that can perform faster or more efficiently than some other computer. In this sense computers have become vastly more powerful. But there is nothing that a modern computer can compute that a Turing machine can't also compute, given enough time. This might seem surprising but it also makes intuitive sense. When you upgrade your computer's processor you don't suddenly gain access to a whole new class of computation that was previous inaccessible. You can simply do the same computation in less time. This performance improvement might be enough to enable new *functionality* – perhaps moving from rendering 2D graphics to 3D – but this is solely because the time taken by the slower processor was deemed unacceptable by the user.

Is a real computer a Turing machine? After all, they look a little similar. If you squint a bit, the tape head capable of reading and writing at any point on the tape is a bit like a processor reading and writing at arbitrary memory locations. A Turing machine, being a mathematical concept, is able to use an infinitely long tape. Real computers, however, have to make do with a limited (though nowadays very large) amount of memory. Something that is computable on a Turing machine may therefore not be computable on a particular physical computer without running out of memory.

Remember that a Turing machine is not meant to be a simple blueprint for a computer. It's a simple model of *computation* that is powerful enough to model any arbitrary computation. We can now explore the capabilities of the Turing machine in order to determine the properties of computation itself.

Computability

Computability theory is the second of the three branches of theory of computation. It asks things like: what is a computable problem? Are there problems that can't be computed? Thanks to our exploration of automata theory, we can approach these questions with the assistance of the Turing machine.

What is a computable problem?

As we saw above, there is no computational model more powerful than the Turing machine. Therefore, a problem is **computable** if it can be solved on a Turing machine. The execution steps performed by the Turing machine form an algorithm and so we can also define "algorithm" in terms of a Turing machine: an algorithm is anything that can be executed on a Turing machine. It is not possible to have an algorithm that can't run on a Turing machine.

We saw above that Turing created the universal Turing machine: a Turing machine that can simulate another Turing machine. If a computational method is capable of simulating a Turing machine, and therefore performing any computation, we say that it is **Turing complete**. One Turing complete computational method is as powerful as another. Usually Turing completeness is what you want. If you're writing a general-purpose programming language, then it's essential that users can write arbitrary computations. The majority of programming languages are therefore Turing complete. Anything you can do with one Turing complete language you can achieve with another. After all, it would be absurd to look at a problem and think "well, this is solvable in Ruby but not in Go".

Let's prove that intuition. Suppose that a problem did exist that was solvable in Ruby but not in Go. It's possible to write a Go program, known as an interpreter (see the compiler and interpreters chapter if this concept is new to you), that takes Ruby source code as input and simulates executing the Ruby code. That means we can just express the problem in Ruby and run the Ruby code through our interpreter written in Go. Therefore the problem is solvable in Go.

In practice, of course, different languages might be better suited to different tasks This doesn't imply any difference in language capability. It's merely a question of what's more ergonomic for the programmer.

Amusingly, things sometimes become Turing complete by accident. Pokémon Yellow is a great example. In a game, every player input triggers some computation that updates the program state. Normally, a game will only allow you to perform limited computation. Moving your character around computes a new position and writes it to memory, acquiring an item updates the inventory and so on. It's been demonstrated that a bug in Pokémon Yellow allows you to update arbitrary memory locations by performing specific actions. You can basically reprogram the game to do what you want. In the further reading section there's an example of someone reprogramming Pokémon Yellow to become a music player. Once you have demonstrated Turing completeness, all bets are off!

It's hard to overstate the importance of Turing's 1936 paper. In one fell swoop he designed an automaton capable of modelling any computation and demonstrated that other computational models could be equivalently powerful but no more. Unfortunately for the fledgling discipline of computer science, Turing also demonstrated that his machine had some striking limitations.

Are there uncomputable problems?

We know that any computable problem can be solved using a Turing machine. Are there problems that are uncomputable altogether? Some problems with subjective answers self-evidently don't make sense as a computational problem e.g. "which of these paintings is the more beautiful?". There are also problems that seem like they should be straightforward but are actually uncomputable:

> Do `functionA` and `functionB` behave in the same way for all inputs?
> Does a given function print "hello, world"?

I use the term "function" here in order to express them as problems that might come up in programming. A function defines an algorithm and an algorithm defines a Turing machine so the three terms are broadly interchangeable.

You might think that the first problem could be easily solved by passing in many different inputs to both functions and checking that they behave in the same way. Such an approach can give us reasonable confidence but we can never be absolutely sure. Since the number of possible inputs is infinite, we can never be sure that the two functions might not behave differently for some as yet untested input.

The second problem is more surprising. Remember that a Turing machine will either halt or continue indefinitely. If it halts then we can just check whether it outputted "hello, world" before halting. The problem occurs when it does not terminate. We can tell that it hasn't printed "hello, world" *yet*, but we can never be sure, in every possible case, that it won't eventually print the string. We need to let it run for a potentially infinite length of time. We can therefore rephrase this problem in more general terms:

> Does a given function halt on a given input?

This is known as the **halting problem** because we can't know if a Turing machine implementing the computation will ever halt. Let's look at an example.

You work at some hot new startup that provides lambda functions over blockchain. Your users upload small functions that you execute on a server farm before writing the results out on to a blockchain. Your users are happy because they don't have to worry about servers and your investors are happy because your startup uses a blockchain!

You, however, are not happy. Some of your users keep writing broken functions that end up in infinite loops, hogging your server resources and requiring you to go in and stop them manually. Your boss, who has an MBA background, asks if you can write a program that will scan users' uploaded code and check for infinite loops before letting them run on the server farm. How would you go about solving this? You write a static analysis tool that examines the source code without running it and finds obvious loops like this:

```
function f() { g(); }
function g() { f(); }
```

Your analysis tool runs on every uploaded function. It catches many infinite loops but still plenty get past your checks. You're missing something. Looking at what your users upload, you see that often mutually recursive functions don't directly call each other, as `f` and `g` do. There might be any number of intermediate function calls obscuring the relationship. Sometimes the uploaded code stores functions in data structures and moves them around or even generates them at runtime. No matter how thorough your tool is, static analysis just isn't going to cut it. You're going to have to actually run the code to see how it behaves. This raises a new problem. How can you tell the difference between a program caught in an infinite loop and one in a very slow but finite loop? Try as you might, what you will eventually discover is that there is no way to write a program that will tell you *in all cases* whether a given program will terminate or loop infinitely on a given input. This is the halting problem.

It's surprisingly easy to write a proof by contradiction of the halting problem in code. If you haven't come across a proof by contradiction before, the idea is simple. Assume that the thing in question is true. Work through the consequences until you hit a contradiction. Since there's a contradiction, the thing in question must not be true. Let's assume that it is possible to write a function that determines whether a given function will terminate (i.e. halt) on a given input:

```
function terminates(program, input) // returns bool
```

This function signature shows that it takes a function called `program` and an input and returns a boolean value showing whether or not `program` returns when given that input. Assuming the existence of this function, we can write an `inverse` function that does the opposite:

```
function inverse(program, input) {
  if terminates(program, input) {
    while(true) { continue; } // loop indefinitely
  }
  return // halt
}
```

If `program(input)` halts, then `inverse(program, input)` continues indefinitely and vice versa. What happens if we pass `inverse` to itself?

```
function inverse(inverse, input) {
  if terminates(inverse, input) {
    while(true) { continue; } // loop indefinitely
  }
  return // halt
}
```

Don't worry if you find the recursion a bit mind-bending. If `terminates(inverse, input)` evaluates to `true` then we loop indefinitely. But that means that `terminates(inverse, input)` should return `false`! But if it returns `false` then `inverse` will return and so `terminates(inverse, input)` should return true! It seems that `terminates(inverse, input)` has to return both `true` and `false` at the same time. We have found our contradiction and conclude that `terminates` does not exist after all.

Expressed in more abstract terms, we have proved that it is impossible to create a Turing machine that takes another Turing machine and determines whether it will loop or halt on a given input. In some respects this is actually quite a reassuring result. For one thing, if it were possible to determine whether a computation would halt, it would have huge implications for mathematics. We'd be able to prove lots of conjectures by showing that a program would eventually halt after finding a counterexample to the conjecture. For example, the Goldbach conjecture states that every even number greater than two is the sum of two prime numbers. Currently it's unproven. We could write a program that would iterate through every even number greater than two and check whether it's the sum of two primes. It would run until it finds the first counterexample, at which point it halts by returning the counterexample. If we could solve the halting problem, we could determine whether this program would ever terminate and so prove or disprove the Goldbach conjecture. Such a program would in effect "short circuit" the computation by getting the result without having to do the work to find the counterexample. Intuitively it doesn't make sense.

Moving back from mathematics, the key take away for a working programmer is to be wary of any task that requires you to know how long a computation will take. You may be able to make a reasonable guess most of the time but you can never have a general solution that works for all cases. There's a reason why real cloud services just put a basic timeout on their users' lambda functions: there's really no better way to detect infinite loops!

The Entscheidungsproblem and the incompleteness theorem

The **Entscheidungsproblem**, or decision problem, was posed in 1928 by mathematician David Hilbert. His question, expressed in rough terms, was: "is there an algorithm that can prove whether a given logical statement is true?". Hilbert's hope was that it would be possible to find a computational means of discovering mathematical theorems (i.e. true mathematical statements). Once written, this program could be left to beaver away and uncover wonderful new theorems.

Kurt Gödel dealt a blow to Hilbert with his first **incompleteness theorem**, published in 1931. This states that any "sufficiently expressive" formal system for expressing truth will be incomplete, meaning that it will include statements that are true but unprovable by the system's rules. Turing's 1936 paper used Turing machines to demonstrate this. His reasoning was that the "algorithm" to solve the decision problem could be expressed by asking whether the algorithm's corresponding Turing machine would halt. Since there was no general solution to the halting problem, there could be no general solution to the decision problem.

If every true statement can be proved, then we could write a program that took a statement and generates candidate proofs until it halted on one that either proved or disproved the statement. Since the number of possible proofs is infinite, we are back to asking whether a Turing machine will halt or loop indefinitely. It appears that there are some problems, such as the Entscheidungsproblem and the halting problem, for which it is simply impossible to write an algorithm that will always give the correct answer. Such problems are **undecidable**.

Is it just me or are things getting pretty heavy? Let's look at a concrete example.

After the great Bitcoin crash, your investors lost interest in your lambda function blockchain startup. You find a new job building a digital catalogue system for a library. At least here you should be safe from impossible tasks! Your first job is to create a catalogue of every resource in each section of the library: cars, computing, fiction etc. This is straightforward. Flushed with success, you next create an overall catalogue of every resource in the entire library. Just as you're finishing, you realise that the section catalogues need to be included in the overall catalogue too. In fact, the overall catalogue is itself a library resource and so should be included in itself. Clever!

Your boss is pleased by your industriousness but is getting confused by the profusion of catalogues on the system. She asks you to create a catalogue of all the catalogues. Easy! On showing it to your boss you remark that you'd remembered that the catalogue of all the catalogues should include itself since it's a catalogue too. This is all a bit too much for your boss and she tells you that she'd much rather have a catalogue of just the catalogues that don't include themselves. When you put this together you run into a problem: should this catalogue include itself? If it doesn't include itself then it will be an example of a catalogue that doesn't include itself and you would be remiss in leaving it out. Yet if you do add it in, the catalogue will include itself and so won't meet the requirements for inclusion. Whatever you do will be wrong!

Does this remind you of the proof by contradiction of the halting problem? An even simpler example is the liar's paradox: "this sentence is false". Its simplicity makes the root of the problem obvious: the statement is self-referential and leads to paradox. When you follow the logic of either the catalogue

example or the liar's paradox, you find yourself caught in an infinite loop: it is false, therefore it is true, therefore it is false, therefore it is true and so on. The infinite loop means that we cannot prove the statement to be either true or false. Because the algorithm loops infinitely, a corresponding Turing machine will also never halt.

Gödel's work is complex and requires a sophisticated understanding of logic to really comprehend deeply. My explanation is by necessity very high level, but I hope that these analogies give you some glimpse of the incompleteness theorem's implications. It is perhaps easier to understand in terms of its connections to other fields. Finding a solution to the halting problem, for example, would imply that a solution to all the above paradoxes existed. The behaviour of the Turing machine gives an intuitive, mechanical expression to Gödel's profound insights.

Algorithmic complexity

We have a better understanding of what computation is and isn't. Next, we consider how computation *behaves*. How does one algorithm compare to another? A very important metric is the algorithm's **complexity**. We measure the complexity of an algorithm in terms of how many resources it needs. The two most important resources in a computer are the amount of processing **time** something needs and the amount of **space** in memory it consumes. Usually the resource usage depends on the size of the input so we'd like to know how these requirements change as the input changes.

The time an algorithm takes to complete is known as the **running time**. It's commonly measured by how many steps the algorithm takes or by how long a computer spends executing the algorithm. Obviously the exact running time will vary from machine to machine but what we're most interested in is the relationship between the input size and running time. By "size" we technically mean how many symbols a Turing machine needs to encode the input, but it's easier to think of it as literally referring to the length of a string input. Let's say we have an algorithm that takes a certain amount of time to execute for an input of length n. If the input length increases to $n + 1$ will the algorithm use the same time, a little bit longer, twice as long or maybe even longer? Failing to consider this could lead you to write code that works wonderfully on small, test inputs but falls over as soon as it has to handle large inputs. That is undesirable.

Turning to space requirements, an algorithm might require a large working space to hold intermediate values or generate data structures. On a Turing machine this is no concern because the tape is infinitely long. Real computers don't have this luxury and must make do with a finite amount of memory. Does the computer have sufficient memory to contain the working space required by the algorithm? Will it run out of memory if we give the algorithm a larger input?

Nowadays computers have absurd amounts of memory and so space is generally less of a concern than time. It's fairly common, as I will do, to focus on the running time of an algorithm as the key measure of performance. Bear in mind that the exact same principles apply to measuring space requirements. This can still be a major problem on memory-constrained mobile devices.

It's often possible to improve an algorithm's efficiency to use less space or take less time. But as the computing gods give with one hand, they take with another. There is usually a trade-off between optimising for time and for space. For example, in some cases it's possible to reduce running time by using a technique called **memoisation** to cache intermediate values and avoid recomputing them at a later step in the algorithm. The time saved comes at the cost of using more space to store those intermediate results.

We want to describe how the time and space requirements of an algorithm change as the input size changes. We need a way of expressing this that's both simple and makes it easy to compare algorithms. Computer scientists have developed a pleasingly rough form of estimation called **big-O notation**. The basic observation behind big-O notation is that we can model the running time of an algorithm as a ratio of the input size to the running time. The ratios of simple algorithms can be combined to describe the behaviour of more complex ones. Let's take two hypothetical functions:

function	running time
`double`	2n
`square`	n^2

As the names suggest, the running time of `double` is always double the input size and `square`'s is the square of the input size. Now let's look at another function that first performs `double` and adds it to the result of `square`. Its running time would therefore be the sum of both running times:

function	running time
`squareDouble`	$n^2 + 2n$

Let's see how the running times of `double` and `square` change as the input size increases:

n	double(n)	square(n)
1	2	1
10	20	100
100	200	1,000
1,000	2,000	1,000,000

As you can see, the running time of `square` increases at a much faster rate than that of `double`. This means that as the input size grows, the contribution `double` makes to the total running time of `squareDouble` becomes smaller and smaller. For really large inputs it's insignificant. The key idea behind big-O notation is that we can get a rough idea of an algorithm's complexity just by looking at the ratio of its most significant component. That's the bit that makes the biggest contribution to the algorithm's running time. In the case of `squareDouble`, it's `square`. Even if we have a big, complicated algorithm with lots of bits contributing to the total running time, we can just ignore everything except the element that makes the biggest difference to the running time.

We can simplify even further. What if we had a function called `triple` that had a time requirement of three times the input size? Well, it has a ratio of $3n$, which is clearly bigger than `double`'s $2n$, but as the input size increases, they both quickly pale into insignificance compared to `square`'s n^2. In a wonderful flourish of contempt, big-O notation declares that the coefficient, the number in front of

the n, is too insignificant to bother worrying about. We focus solely on the ratio. Here's how we can express the running time of our four functions using big-O notation:

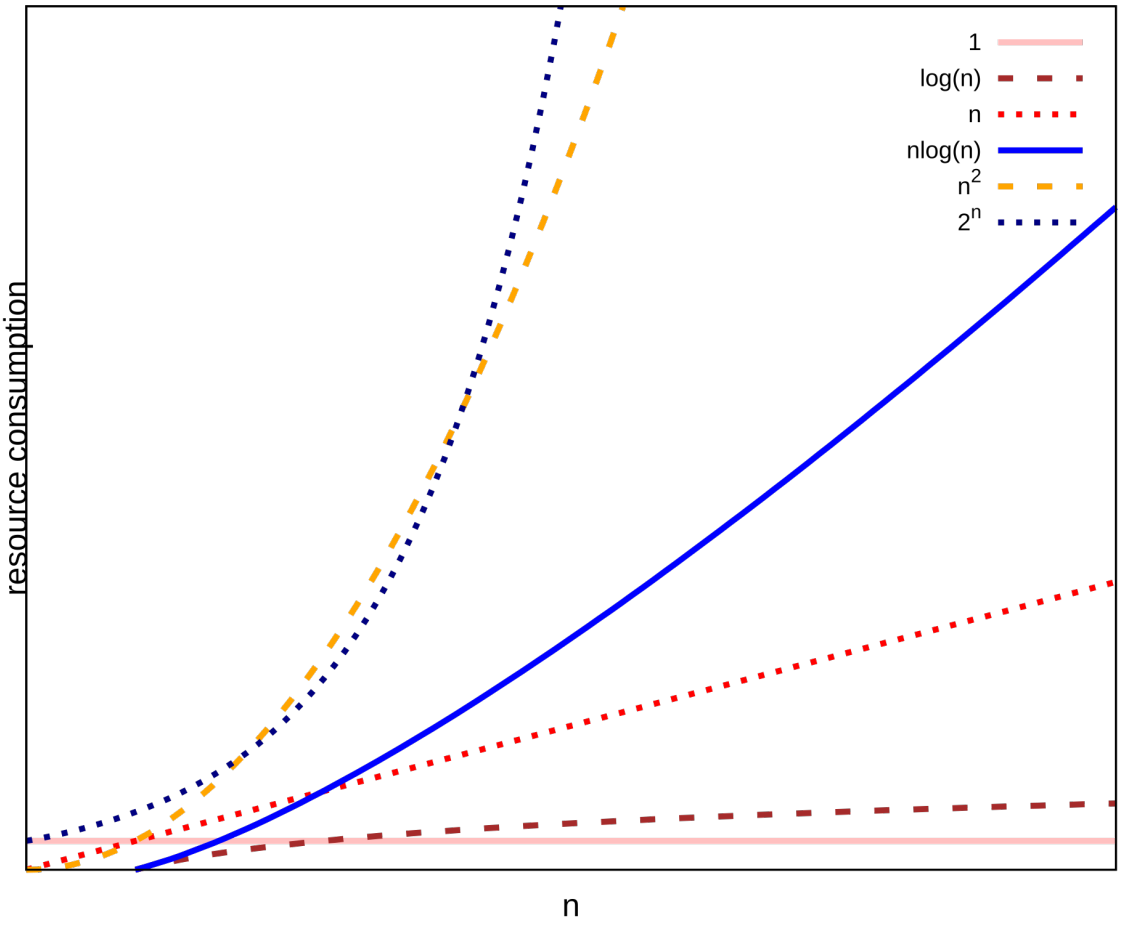
function	running time	big-O notation
`double`	2n	O(n)
`triple`	3n	O(n)
`square`	n^2	$O(n^2)$
`squareDouble`	$n^2 + 2n$	$O(n^2)$

We don't care that `triple` is actually slightly slower than `double` because, for very large inputs, they perform roughly the same as each other in comparison to `square`. Similarly, `squareDouble` is slightly slower than `square`, because of the additional `2n` in its running time, but the difference becomes negligible for large inputs.

When using big-O notation, we are only talking about the **worst-case performance**. The time it takes to iterate through an array to find a value will depend on where in the array the value is located. If the value is at the very beginning of the array, it'll take less time than if it's at the very end. Being pessimists, we always assume the worst case. Big-O notation shows us the upper bound on the algorithm's requirements – at least it can't get any worse!

There are two main ways to determine the complexity of an algorithm. The first is to simply look at the code and see how it behaves. This works for simple cases written in relatively low level languages, such as C, where the structure of the code closely matches the operation of the algorithm (hence the popularity of teaching algorithms in C). In higher-level languages, such as JavaScript or Ruby, a lot of computational steps can be hidden behind function or method calls, not to mention the behaviour of the underlying interpreter. Often the easiest thing to do is just benchmark the running time for various inputs and see what ratio emerges.

We can categorise algorithms by their $inputsize : runningtime$ ratio. The most common categories are graphed below:



Algorithm growth

The flatter the curve, the better. The best is $O(1)$ because it's a completely flat line and so doesn't change at all. Some lines, such as $O(\log n)$, increase very slowly. This is great because it means that an increase in input size doesn't have *too* much impact on the running time. Others, such as $O(2^n)$, increase almost vertically! This is very bad because even a tiny change in input size has a huge impact on the running time. Look carefully and you'll see that some curves start off quite slowly and only increase steeply later. There's a jumbled point in the bottom left where curves cross each other, meaning that one ratio is better up to a certain point and then another is better after that. An algorithm with a "bad" ratio might actually perform well in practice for small inputs. In theory, though, we're only interested in what happens with really big inputs.

Let's look at an example algorithm for each ratio. I'm using a C-like pseudo-code so that every step in the algorithm is clear.

Constant $O(1)$:

Constant algorithms, as we've just seen, have the same running time no matter the input size. Capitalising the first character of a string is an example of a constant time algorithm. It never has

to go past the first character and so the input size has no effect on the running time.

```
string capitaliseInitial(string input) {
  input[0] = toUpperCase(input[0]);
  return input;
}
```

Constant algorithms are fairly uncommon, unfortunately.

Logarithmic $O(\log n)$:

If each increase in input size adds a decreasing amount to the running time, then we have logarithmic complexity. In the chart you can see that the line gradually becomes flatter. In fact, the increase in running time trends towards zero. The cool thing about this is that very large inputs barely require any more resources than smaller ones.

One of the most famous examples of a logarithmic algorithm is binary search (see the algorithms chapter for more details). Below is an implementation that looks for a key in an array of inputs. If it finds the key, it returns the key's index.

```
int binarySearch(int[] inputs, int low, int high, int key) {
  int middle = (low + high) / 2;

  if (key == inputs[middle]) {
    return middle;
  }
  if (key < inputs[middle]) {
    return binarySearch(inputs, low, middle - 1, key);
  }
  if (key > inputs[middle]) {
    return binarySearch(inputs, middle + 1, high, key);
  }
  return -1 // not found
}
```

Observe that each recursive call to `binarySearch` discards one half of the input. At first `low` and `high` cover the whole array, then half, then a quarter, then an eighth and so on. This is a classic indication of a logarithmic algorithm. Sadly, algorithms with logarithmic performance are also fairly uncommon.

Linear $O(n)$:

The resource requirements of linear algorithms increase in proportion to the input size. Each increase in input size adds the same amount to the running time, as you can see by the straight line in the chart above.

Strings in C are terminated with the null character `\0`. To find the length of a string you iterate through the string until you find the null character:

```
int stringLength(string input) {
  int i = 0;
  while (input[i] != '\0') {
    i++;
  }
  return i;
}
```

Each additional character in the `input` string adds one extra iteration to the `while` loop and so the algorithm is linear. Such algorithms are very common, particularly for basic operations on collections of values. For more complex problems a solution with $O(n \log n)$ running time – a combination of linear and logarithmic – is generally seen as pretty good.

Polynomial $O(n^x)$:

An algorithm's performance is polynomial if an increase in the input size causes the running time to increase by multiples of the entire input size. Often this happens because the algorithm contains nested loops. The running time increases at a much faster than linear rate. The exact rate depends on the value of x. n^3 increases more rapidly than n^2.

```
printPairs(string letters, int length) {
  for (int i = 0; i < length; i++) {
    for (int j = 0; j < length; j++) {
      print(letters[i], letters[j]);
    }
  }
}
```

This algorithm prints every possible pair of characters from a string. The nested iteration is a giveaway that we're dealing with polynomial complexity. Rather than making a single pass through the input (as we would if the algorithm were linear), we make a pass through the entire input once for *every* value in the input. In total we make $n \times n$ (i.e. n^2) iterations, so the performance is **quadratic**. If there were another nested loop we'd have n^3 or **cubic** performance.

Polynomial performance is bearable but generally undesirable. Nested loops indicate places where a more efficient solution may be possible but they are not always as obvious as the example above. For example, the following code has a performance flaw that is a common gotcha in C code:

```
string capitalise(string input) {
  for (int i = 0; i < stringLength(input); i++) {
    string[i] = toUpperCase(string[i]);
  }
}
```

Do you see the problem? What performance does this function have? On the surface it looks to be linear because there's only one loop and we know that `toUpperCase` has constant performance. However, the condition check within the `for` loop makes a call to `stringLength` once for each element in the input. Since `stringLength` performs its own iteration through the input, we end up looping through the whole input once for every input element: quadratic performance!

Exponential $O(x^n)$:

The resource consumption of an exponential algorithm grows more quickly than any polynomial. An algorithm with 2^n complexity will double its running time if the input increases by just one!

```
int fibonacci(int num) {
    if (num <= 1) {
      return num;
    }
    return fibonacci(num - 2) + fibonacci(num - 1);
}
```

This inefficient implementation of the Fibonacci sequence has exponential running time because it doubles the number of steps every time n increases by one. Try writing out the steps it performs for $n = 3$ and $n = 4$ if you don't see this. Exponential algorithms often use a **brute-force search** to solve the problem by trying every possible solution. This is usually impractical for anything but small inputs.

Factorial $O(n!)$:

If you're not familiar with factorials, $4!$ is equivalent to $4*3*2*1$ and $5!$ is equivalent to $5*4*3*2*1$. Each increase in the input size multiplies the running time by n. Factorial algorithms very quickly lead to huge resource requirements and are rarely useful beyond a very small input size. An example of a factorial algorithm is printing every possible permutation of a list.

Looking at the big-O notation again, we can see that constant time and linear time are actually both examples of polynomial time ($O(n^x)$). Constant time is a special case where x is 0 and linear time is a special case where x is 1. You may have noticed that we crossed a boundary when we went from polynomial to exponential. A problem is **tractable** if it has a solution in polynomial time. This means that the algorithm is efficient enough to be useful, albeit maybe somewhat slow.

A problem is considered **intractable** if it doesn't have a polynomial time solution. There might be some exponential or factorial solution but it will be so inefficient as to be unusable in practice.

For example, cubic algorithms ($O(n^3)$) are pretty slow and you'd want to avoid using them where you can. But look how the running time compares to an exponential algorithm ($O(2^n)$):

n	n^3	2^n
1	1	2
10	1,000	1,024
100	1,000,000	1,267,650,600,228,229,401,496,703,205,376

You wouldn't want a cubic algorithm sitting in a server's request handler but it's still unimaginably more efficient than an exponential algorithm.

Complexity classes

A complexity class is a group of problems that all require similar computational resources. They therefore have roughly the same level of complexity. To better understand this we need to introduce a couple of new concepts.

Problem A can be **reduced** to problem B if you can easily write a solution to problem A by using a solution to problem B. For example, finding the smallest value in an array can be solved by first sorting the array in ascending order and then taking its first element. Once you've got the array sorted, taking the first element is trivially easy. The problem of finding the smallest value is reduced to the problem of sorting the array. Because finding the first element is a constant time operation, as we saw above, we say that the problem *reduces in constant time*.

```
function smallest(values) {
  return sort(values)[0];
}
```

You might be thinking that this makes no sense! Sorting the array will take longer than just iterating through it once, keeping track of the lowest value seen so far. How can you reduce problem A to problem B if the solution to B is less efficient? At the risk of more theoretical hand-waviness, we treat the solution to problem B as an imaginary **oracle** that provides the correct answer straight away (i.e. in constant time). Bear with me on this. Just imagine that `sort` in the example above returns immediately.

Moving on, some problems are known as **decision problems**. They only provide true/false answers. The problem "is this a valid path between these two points?" is a decision problem. It only asks whether the given solution is valid. The corresponding **search problem** asks: "which is the fastest path between these two points?". You can see that the decision problem is easier to solve than the search problem. Answering a search question requires generating a valid solution or demonstrating that none exists.

Armed with these new terms we can begin to explore complexity classes. Problems are categorised in different classes based on the complexity of generating and verifying solutions.

All problems in the **polynomial** (P) class have a polynomial time solution. These are the tractable problems for which we have efficient solutions.

A problem is in the **non-deterministic polynomial** (NP) class if a possible solution can be verified in polynomial time. This says nothing about whether an actual solution can be generated in polynomial time. If someone comes to you claiming to have solved a Sudoku puzzle, you can easily verify their solution in polynomial time by checking that no row, column or square has duplicated digits. Actually coming up with a correct solution is a substantially more difficult task.

It appears that this definition bears no relation to the name. Why "non-deterministic"? An alternative definition for NP involves non-deterministic Turing machines that generate guesses for solutions and then verify them. This definition has fallen somewhat out of use, probably because it's harder to understand, but the name has stuck. Let's just accept it for what it is and roll with it.

It is self-evident that all P problems must also be in NP. If I give you a claimed solution to a P problem you can verify it in polynomial time simply by rerunning my computation and checking you get the same answer. By the definition of P, this verification can be done in polynomial time. Therefore the problem is in both P and NP.

Is it true that every problem with efficiently verifiable solutions has an efficient algorithm to generate solutions? Every problem in P is in NP but is every problem in NP in P? This is known as the **P = NP? problem**. It's hugely important but as yet unanswered. In fact, it's the biggest unsolved problem in computer science. Theorists generally suspect, but do not know for sure, that $P \neq NP$. A great deal of effort has gone into finding efficient solutions to certain NP problems with no success. On the other hand, efficient solutions *have* been found for some NP problems and no-one has managed to actually prove that $P \neq NP$. The question remains open.

The whole issue might seen rather arcane but the implications of proving $P = NP$ would be huge. It would prove that efficient solutions definitely exist for any problem that can be efficiently verified. Public-key cryptography, which is used to encrypt much of the communication on the Internet, relies on $P \neq NP$. It's based on the assumption that brute-force guessing the correct key to decrypt a message is too computationally expensive to be feasible. If it were true that $P = NP$, this would imply that an efficient key search algorithm exists. Although, if you worked at a three letter agency and found a proof that $P = NP$ that enabled you to decrypt much of the world's encrypted communications, would you tell the world?

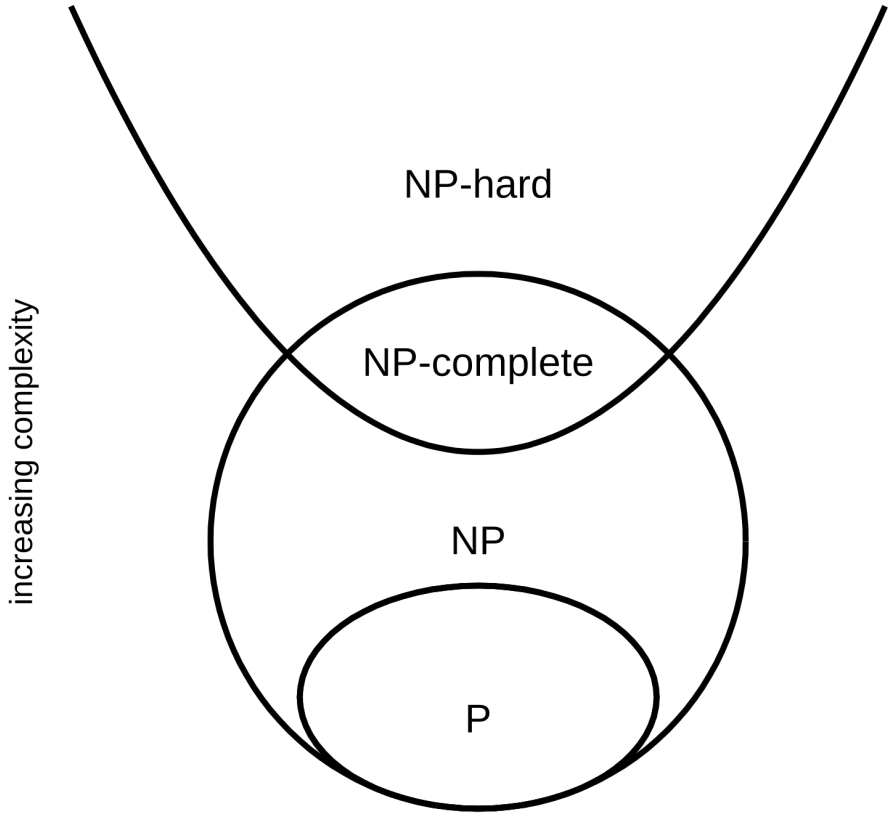
Taking off my tinfoil hat, let's move on to **NP-hard** problems. They are, well, hard. They are at least as hard as the hardest problems in NP. A problem is NP-hard if **every** problem in NP can be reduced to it in polynomial time. A consequence of this is that a solution to one NP-hard problem would allow *all* NP problems to be solved in polynomial time. As things stand, there is no known efficient solution to any NP-hard problem. To better understand how a single NP-hard problem can be used to solve all NP problems, let's turn to our old friend: the halting problem. Any NP problem can be expressed as a Turing machine that halts when it finds a valid solution to the problem. It can be

reduced to the halting problem simply by asking whether that Turing machine will ever halt. The halting problem is therefore an NP-hard decision problem.

An interesting class of problems are NP-hard but are efficiently verifiable so they are in NP too. Such problems are known as **NP-complete**. Often the decision version of an NP-hard problem is NP-complete. The most famous example is probably the travelling salesman problem: what is the shortest possible route that goes through a set of cities and returns to the start? Coming up with a solution is an NP-hard problem. But checking whether a proposed route is indeed the shortest is an NP-complete problem. Solving a Sudoku puzzle is another example of an NP-hard problem with a polynomial verification.

Is the halting problem also in NP? Well, it is indeed a decision problem but we can't verify a solution to the halting problem in polynomial time because the halting problem can't give us a solution at all. It is **undecidable**.

Here are the various complexity classes represented as a Venn diagram (assuming that $P \neq NP$). You can see how NP-complete represents the overlap between NP and NP-hard:



Venn diagram of complexity classes

I promised less theoretical hand-waving and here I am talking about polynomial complexity and magic oracles. Stick with me – this stuff really is useful. You need to know about it because many problems are NP-complete. This means that no-one has been able to find an efficient solution to them. Without meaning to be a downer, it's unlikely that you'll suddenly come up with one yourself, so you need to be able to recognise when a particular task is an instance of an NP-complete problem. Unfortunately, many of them have quite obtuse names like the "subset sum problem" or the "subgraph isomorphism problem". In the further reading is a link to Karp's *21 NP-Complete problems*, which provides an overview of common problems.

To give some concrete examples, maybe you were fired from the library after the whole catalogue incident but have been taken on at a new Uber-for-bikes startup. Given a list of people who want to be picked up at specific locations and times, how best do you allocate all of the available bikes? This is an NP-complete problem. How do you arrange the seating plan at a wedding so that people sit with people they like and not next to people they dislike? This is NP-complete. If you have a list of backup files of varying sizes and a set of disks with varying amounts of free space, how do you fit all of the backups on the disks? Again, NP-complete.

Before spending a lot of time trying to find an efficient solution to a tricky problem, it's worth reviewing some common NP-complete problems and considering whether your problem might be an NP-complete problem in disguise.

Conclusion

In this chapter we examined the theoretical basis of computer science. The reason that computer scientists study theory of computation is to better understand computation itself. They develop mathematical models, known as automata, to examine the nature of computation. We looked at finite automata, push-down automata and Turing machines. Each class of automaton is more powerful than the one before it. Even simple finite automata can be very useful embedded in other programs. Turing demonstrated that no automaton can be more powerful than a Turing machine. Although anything computable can be computed on a Turing machine, there are significant limits to what is computable. In particular, it is impossible to compute anything that relies on knowing whether a given computation will terminate. This is known as the halting problem. Many deep insights can be more easily understood by modelling them as Turing machines. We looked at theoretical approaches to measuring algorithmic performance in terms of time and space requirements. We saw that problems could be grouped into different complexity classes depending on whether solutions could be efficiently generated and verified. Of particular interest is the NP-hard and NP-complete classes of problems that have no known efficient solutions.

Further reading

Even if you have no desire to read any further into theory of computation, I highly recommend that you check out the **lambda calculus**. It is another model for describing computation that was

developed by Alonzo Church in 1936. Instead of starting a huge dispute over whose model was better, Church and Turing sat down like sensible adults and realised that their two computational models were equivalent. And thus we call it the Church-Turing thesis. Everything that we demonstrated with Turing machines could have been demonstrated with the lambda calculus. It's just that the mechanical nature of the Turing machine makes it a bit easier to think about. This blog post[2] gives a good overview (and this one[3] explains it in terms of alligators!), but speaking very generally the lambda calculus defines computation in terms of function application. The programming language Lisp is notable for starting off as not much more than an implementation of the lambda calculus. The focus on function application also makes the lambda calculus popular with functional programming fans. You don't need to have a deep understanding of the lambda calculus but it is interesting as an alternative model for computation.

Top of many programmers' list of "favourite books I've never read" is *Gödel, Escher, Bach* by Douglas Hofstadter. This book is *very* difficult to describe succinctly but I'll do my best. It attempts to explain how complex behaviour can emerge from simpler elements, just as an ant nest displays behaviour no individual ant is capable of. Along the way Hofstadter builds up an intuition for Gödel's incompleteness theorems. It's an incredibly rich, varied book about computation in the very broadest sense and I highly recommend it.

A good introductory overview of theoretical computer science can be found in the *Great Ideas in Theoretical Computer Science* course from Carnegie Mellon University. The lectures are available on Youtube[4]. The course starts with theory of computation but then ranges more widely. I recommend you watch at least the first half dozen or so. You might want to come back to the later lectures on graphs.

The canonical textbook for theory of computation is *Introduction to the Theory of Computation* by Michael Sipser. Be aware that it is aimed at senior undergraduate or graduate students and includes plenty of proofs. You don't need to understand the proofs to get something out of the textbook but it is fairly light on intuitive explanations. It covers formal definitions of automata in terms of grammars, time/space complexity, decidability and several advanced topics. If you're comfortable with logic or mathematical proofs and want to go deeper, you can't go wrong with this textbook. If you want to improve your mathematical proof skills before tackling Sipser (which I would recommend), *How to Solve It* by George Polya is a good place to start.

If you want something more formal but aren't quite ready for full-on Sipser, a shorter (though still challenging) resource is *Models of Computation* by Jeff Erickson (available for free here[5]). I particularly liked the example of a Turing machine performing binary addition.

Karp's *21 NP-Complete problems* is more properly known as Reducibility among combinatorial problems[6]. It's not necessarily to memorise the details of all 21 problems, but if you comfortable with mathematical notation it is a useful overview.

[2] https://palmstroem.blogspot.com/2012/05/lambda-calculus-for-absolute-dummies.html
[3] http://worrydream.com/AlligatorEggs/
[4] https://www.youtube.com/playlist?list=PLm3J0oaFux3aafQm568blS9blxtA_EWQv
[5] http://jeffe.cs.illinois.edu/teaching/algorithms/
[6] https://people.eecs.berkeley.edu/~luca/cs172/karp.pdf

For a deeper understanding of Turing machines, my recommendation is *The Annotated Turing* by Charles Petzold. It's a companion to Turing's 1936 paper *On computable numbers, with an application to the Entscheidungsproblem*. The paper is freely available online but quite hard going by itself. Petzold does an admirable job of explaining the subtleties of Turing's arguments and includes plenty of background exposition to help you understand the implications of Turing's discoveries. There is some useful background on basic number theory and logic, a detailed walk through a Turing machine's operation and a broader discussion of computability.

And finally here[7] is blog post describing how to make a music player out of Pokémon Yellow.

[7] http://aurellem.org/vba-clojure/html/total-control.html

Algorithms and data structures

Introduction

We have reached a state of piercing theoretical insight. In this chapter, we will apply our newfound analytical skills to commonly used algorithm and data structures. We'll also briefly cover some abstract data types that will pop up in other chapters.

My approach here is going to be a bit different to what you'll see in most textbooks on algorithms and data structures. They put a lot of emphasis on showing how to implement a wide variety of data structures and algorithms, usually in C. I don't think that's a particularly useful way for a self-taught person to start off. Learning how to implement a new data structure or algorithm in isolation is a sure way to immediately forget it. You need the reinforcement of regularly applying what you've learned to solve problems that you encounter daily. Computer science students have the luxury of being set problems and projects that test this new knowledge.

As a web developer, you'll probably spend the majority of your time using the built-in data structures and algorithms provided by your language's standard library or the browser environment. We'll therefore focus on the data structures and core algorithms that are included in the languages and browsers you're likely using every day. You'll learn how they're implemented and what their performance characteristics are.

I won't spend any time on the fancy stuff behind the algorithmic brainteasers that some companies like to use for interviews. There are already many excellent interview preparation resources and there is no need for me to reinvent the wheel. As ever, the further reading section will have plenty of suggestions if you want to continue study.

Data structures

Look at any computer program and you'll see two things: data and operations on that data. From the computer's perspective, everything is just a huge sequence of ones and zeroes (i.e. bits, see the computer architecture chapter for more information). It is the *programmer* who creates meaning by telling the computer how to interpret these ones and zeroes.

Data types are a way of categorising the different forms of data a computer can utilise. A data type determines the (possibly infinite) set of possible values, the operations that can be performed on a value and maybe how a value can be physically implemented as a sequence of bits. Usually a programming language contains a few built-in, **primitive types** and allows you to make new types that build on the primitives. Common primitive types include integers, booleans, characters, floating point (i.e. decimal) numbers and strings.

Some data types are bound to a particular implementation. These are known as **concrete data types**. For example, the range of numbers that can be held by an `int` type depends on how many bits the underlying system uses to represent the number. A 64-bit `int` holds bigger numbers than a 32-bit `int`. Primitive types are nearly always concrete.

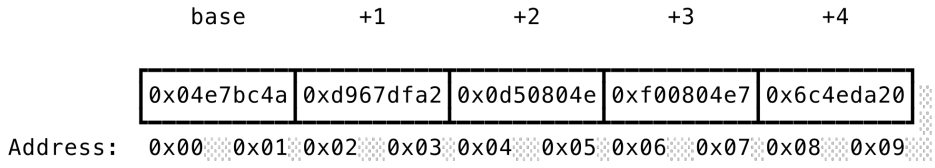
When a data type doesn't specify an implementation, it is called an **abstract data type** (ADT). ADTs are merely descriptions of the set of possible values and the permitted operations on those values. Without an implementation you can't create a value of the type. All you've done is specify how you'd like the value to behave and the computer doesn't know how to create a value that behaves that way. I can want the moon on a stick but without a way of getting the moon on to a stick I'm not going to have it.

To actually use an ADT you need to create a **data structure** that implements the interface. A data structure defines a way of organising data that allows some operations to be performed efficiently. Data structures implement data types. Whether a particular data structure is a good implementation for a data type depends on whether the data structure efficiently implements the operations specified by the data type. The operations a data structure provides can be thought of as algorithms bound to the data held in the structure. We can therefore use the techniques we've seen above to measure the performance of a data structure's operations and determine which tasks it is best suited to.

There exists a veritable zoo of data structures out there. In this section we'll stick to the most fundamental ones: arrays, linked lists and hash maps. They're incredibly useful and built-in to most programming languages.

Arrays

An array is the simplest possible data structure. It is a contiguous block of memory. That's it. The array holds a sequence of identically sized elements. They have to be the same size so that the computer can know where the bits making up one element end and the bits making up another element begin. An array is therefore associated with the particular data type that it contains. The length, or size, of an array is determined by the size of the memory block divided by the element size. It indicates how many elements can fit into the array.



Structure of an array

You can **iterate** through an array, by examining each element in sequence, or you can **index**, by jumping directly to any position in the array. We're able to do this because the structure of the array allows us to work out the location of any element, provided we know the address of where the array starts and the element's position within the array. Imagine that we have encoded the characters of the Latin alphabet into binary numbers that are two bytes long (a byte is eight bits). A array of size ten therefore requires a memory block twenty bytes long. We can immediately index any element

in the array by calculating its memory address. We take the array's **base** address (the address of the first element) and calculate an **offset** by multiplying the element size by the element's index. If we assume our array of ten characters starts at memory address `1000`, the addresses of the first four characters are as follows:

```
array[0] = 1000 + (0 * 2) = 1000
array[1] = 1000 + (1 * 2) = 1002
array[2] = 1000 + (2 * 2) = 1004
array[3] = 1000 + (3 * 2) = 1006
```

The strength of the array is its speed. Indexing an element in an array is a constant time operation. No matter how far away the indexed element is, we can always jump directly to it via a single calculation.

The simplicity of the array is also its main limitation. There is nothing in the structure of an array that marks the end of the array and so there is no way for the computer to know when it's reached the end. In C a variable holding an array actually just holds the array's base address. This means that the size of the array has to be stored in a separate variable and passed around with the array. The programmer is responsible for correctly updating the size variable when necessary. If you mistakenly tell the computer an incorrect size value, it will blithely iterate past the final element, possibly overwriting any values stored in those locations. This is known as a **buffer overflow** and is a major source of often very serious bugs. A second limitation of C-style arrays is that increasing the size of the array requires increasing the size of the allocated memory block. It's often not possible to simply increase the size of the existing one because the adjacent memory addresses might already be in use. We frequently have to find a new, unused block of sufficient size and copy over all of the existing elements. Handling this manually is a bit of a pain.

More modern programming languages provide a slightly different data structure known as the **dynamic array**. They are so called because they have no fixed size, in contrast to **static**, C-style arrays. A dynamic array holds a raw array in a data structure that also keeps track of the number of elements in the array and its total capacity, thus preventing invalid indexing operations and removing the need to store the size in a separate variable. Dynamic arrays can resize automatically when they get filled up. If you don't have to worry about the size of an array – as in Ruby, Python or JavaScript – you are actually using a dynamic array.

In many languages it's hard to find out how a dynamic array actually works under the hood. In JavaScript, for example, there's no simple way to get the current address of an array or any of its elements. This is by design: the whole point of dynamic arrays is to take this responsibility away from the programmer. Instead, let's turn to Go. It offers an interesting intermediate point between the static arrays in C and the dynamic arrays in Ruby, Python and JavaScript. Go has C-style static arrays but builds on top of them another data structure, known as the *slice*, that works like a dynamic array. Go doesn't hide any details of the underlying array. A slice is defined in the Go source code (runtime/slice.go[8]) as a wrapper around an array:

[8] https://golang.org/src/runtime/slice.go

```
type slice struct {
        array unsafe.Pointer
        len   int
        cap   int
}
```

A **pointer** is a value that is the address of another value. It *points* to the value's real location. A pointer can be **dereferenced** by accessing the value stored at the address held by the pointer. The slice doesn't actually contain the array; it just holds the address where the array is. As well as the address of its underlying array, a slice tracks the `len`, the number of elements in the array, and the `cap`, the total capacity of the array. Let's see them in action:

```
package main

import "fmt"

func main() {
        slice := []byte{1, 2, 3}
        array := [3]byte{47}

        fmt.Printf("Slice %#v has length: %d and capacity: %d\n",
                  slice, len(slice), cap(slice))
        fmt.Printf("Slice element addresses: [%#p, %#p, %#p]\n\n",
                  &slice[0], &slice[1], &slice[2])

        fmt.Printf("Array %#v has length: %d\n", array, len(array))
        fmt.Printf("Array address: %#p\n\n", &array)

        fmt.Println("Appending another element...\n")
        slice = append(slice, 4)

        fmt.Printf("Slice %#v has length: %d and capacity: %d\n",
                  slice, len(slice), cap(slice))
        fmt.Printf("Slice element addresses: [%#p, %#p, %#p, %#p]\n",
                  &slice[0], &slice[1], &slice[2], &slice[3])
}
```

This program defines a slice and initialises it with three values. Immediately afterwards, we define a static array of size `3` but with only one initial value. Here is the output:

```
Slice []byte{0x1, 0x2, 0x3} has length: 3 and capacity: 3
Slice element addresses: [c000086000, c000086001, c000086002]

Array [3]byte{0x2f, 0x0, 0x0} has length: 3
Array address: c000086003
```

The addresses are written in hexadecimal. Even though `array` only contains one actual value, space for two more has already been allocated. Notice that the elements held by `slice` are all adjacent to each other in memory and the address of `array` comes directly after. That means there is no empty space between the two variables. What do we expect to happen when we append another element to the slice? We know that the slice is a wrapper around a static array with a capacity of three. There won't be room for a fourth element. We can't just expand the underlying array because the memory address it would use is already occupied by `array`. We therefore expect that Go will have to find a new, bigger memory block for `slice` and move all the existing elements. Sure enough:

```
Appending another element...

Slice []byte{0x1, 0x2, 0x3, 0x4} has length: 4 and capacity: 8
Slice element addresses: [c000086030, c000086031, c000086032, c000086033]
```

The elements all have new addresses because they have been reallocated to a different block of memory. Go allocates a bit of extra space, raising the capacity to 8, to allow more elements to be appended without having to immediately resize again. It's common for dynamic arrays to increase their size by a large amount whenever they reallocate because reallocations are expensive and it makes sense to try and minimise how many are needed.

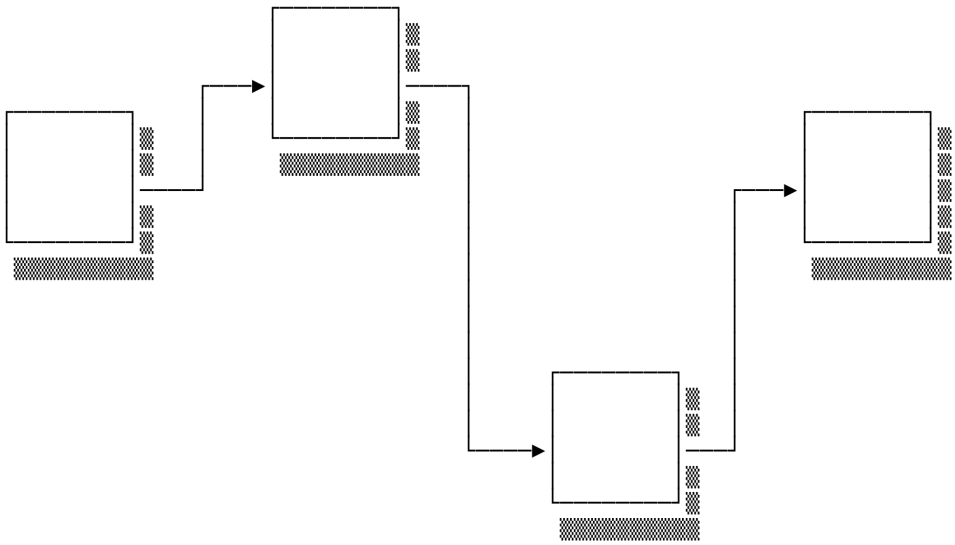
A consequence of having all the elements adjacent to each other is that inserting an element at the beginning or a point in the middle of the array requires shuffling every subsequent element along to make space. Deleting an element entails shuffling in the other direction to fill the gap. This gives us the following performance summary:

operation	**performance**
index	O(1)
insert/delete at beginning	O(n)
insert/delete at middle	O(n)
append	O(1) amortised

Appending an element only causes a reallocation if the array is full. This makes it a bit tricky to work out the performance of appends because it can vary a lot depending on whether a reallocation is needed. Rather than saying that most appends are constant but those that trigger a reallocation are linear, we average out or **amortise** the cost of reallocation across all append operations. Amortised in this way, the cost of the occasional reallocation becomes negligible and we say that appending is a constant operation.

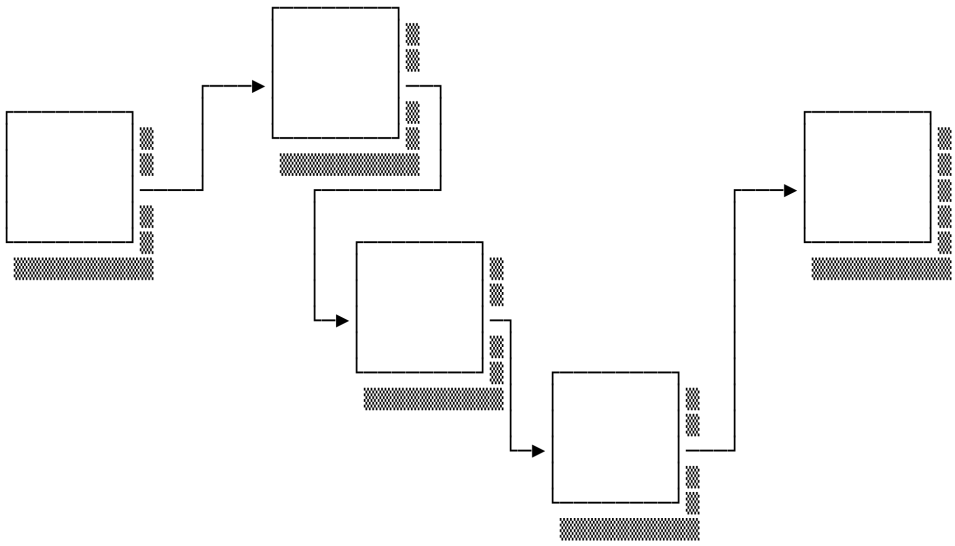
Linked lists

We've seen that arrays are contiguous blocks of memory of a fixed length. Increasing the size of the array requires reallocating the elements to a new, bigger memory block. It is possible to overcome this limitation by building a dynamic array on top of a static one. A completely different approach is the **linked list**. It's an extremely simple, recursive data structure in which each element contains a value and a link to the next element. The first element is known as the **head** and all of the following elements are the **tail**. The structure is recursive because the tail is itself a linked list with a head and a tail.



Structure of a list

The "link" in each element is a pointer to the next element. If you have the head, you can iterate through the whole list by following the head's pointer to the next element, then the next pointer to the following element and so on. Because we explicitly record the elements' addresses they don't need to be next to each other in memory. In fact, they can be stored anywhere. This makes it easy to insert or delete elements. There's no need to maintain a contiguous memory block and reallocate when it's full. Insertions and deletions can be performed in constant time by rewriting the pointers in the surrounding elements. In the diagram below you can see how most of the elements are unaffected by the insertion of a new element:



List insertion

However, to insert or delete an element in the middle or end of the list we first need to find the preceding element. The strength of the linked list, its explicit addressing, is also its downside. Since we have no idea where a particular element might be in memory, we need to iteratively follow each pointer from the head until we get to the element. This means that accessing an element in a linked list is a linear time operation. The access time needs to be factored in when considering insertions and deletions that aren't at the head of the list.

Linked lists are particularly popular in functional programming languages such as Haskell, mostly for reasons of convention and style. Functional languages, which we'll see more of in the chapter on programming languages, prefer to use recursion over explicit iteration with loop counters and so on. The recursive structure of the linked list fits this nicely. In Haskell you move through a list by splitting a list into its head and tail and recursively repeating the process on the tail. Here is an example of searching for a character in a list:

```
1  find :: [Char] -> Char -> Bool
2  find haystack needle =
3    case haystack of
4      [] -> False
5      head : tail -> if head == needle then True else find tail needle
```

`haystack` is a list of characters and `needle` is the character we are looking for. In the `case` structure we check what `haystack` contains. If it's an empty list (`[]`), it clearly doesn't contain what we're looking for and we can return `False`. Otherwise, we use the `head : tail` syntax to pull the head off the list. If it's a match, we return `True`. If not, we recursively call `find` with the tail. Each time we call `find` we peel off the first element from the list and `haystack` gets smaller and smaller. Eventually, we either find the value or end up with an empty list. And to prove that it works:

```
> find ['a', 'b', 'c'] 'b'
True
> find ['a', 'b', 'c'] 'd'
False
```

Indexing an element in an array is a constant time operation but insertion or deletion is linear due to the shuffling required. For linked lists it's the other way around. Indexing is a linear operation but making the actual insertion or deletion is constant once the element has been accessed. A simple optimisation for appending is to maintain a pointer to the final element in a variable. This gives direct access to the end of the list and avoids needing to iterate through the whole list to reach it.

operation	performance
index	O(n)
insert/delete at beginning	O(1)
insert/delete at middle	access time + O(1)
append	O(n), O(1) if end known

The lists we have seen here are known as **singly-linked lists** because each element holds only a single link: the address of its successor. If you want to find an element's predecessor you have to search from the beginning. The links in the chain only go one way. In a **doubly-linked list** each element also contains a link to its predecessor, which makes it possible to move back up the list at the cost of having to store two addresses per element instead of just one. Whether the extra space overhead is worth the speed improvement depends on the size of the elements and how the program uses the linked list.

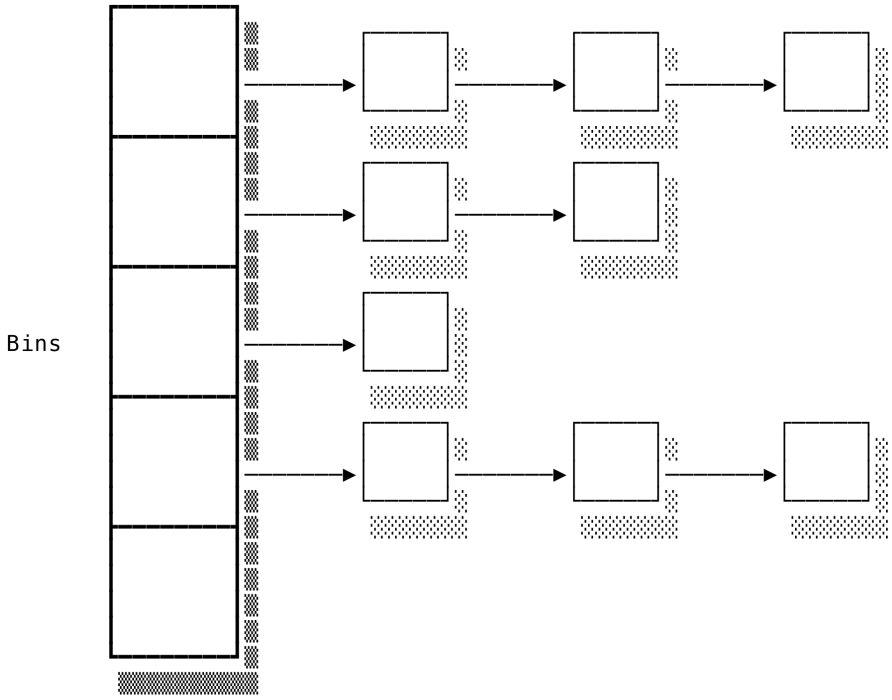
Hash maps

Sequential data structures have many uses but the only way to refer to an element is by its index. Very often we want to refer to a value by its name or some other identifier that we shall call its **key**. To get this functionality we need the **hash map**. This data structure stores elements as key-value pairs and allows the user to retrieve a value using its key. Hash maps go by a variety of names. In Ruby they're called hashes, in Python they're dictionaries and in JavaScript they're objects. The slight differences between each implementation aren't important for us.

Hash maps are so-called because they rely on **hashing**. This is one of those computer science things that pop up a lot and sound very complicated but actually aren't. A **hash function** maps an input to an output within a given range. For example, `string.length() % 5` will convert all strings to a number between zero and four. It's important that the function is **stable** and always gives the same output for the same input because that creates a reliable mapping between the input and the output. Ideally, a hash function will distribute the inputs evenly across the range of output values. Coming up with good hashing functions is a tricky mathematical problem. Happily we can leave that to one side and simply rely on existing implementations.

A hash map uses a hash function to convert the key into an integer that is then used as an index into an array. Each array element is referred to as a **bin** that holds a value. The hash function maps the

key to a location in the array, making the key a kind of indirect index into the array. As we insert more values into the hash map, we will most likely encounter **collisions** where two different values are hashed into the same bin. A common solution to this problem is to put both values into a linked list and store that list in the bin. When we want to retrieve an element, we hash the key to find the bin as usual and then iterate through the list until we find the desired element. So the hash map is itself built out of the two data structures we've already seen!



Structure of a hash map

When everything's going well, hash maps offer constant time performance. The only significant cost is performing the hashing operation. However, a hash map's performance can become degraded if the hash function puts too many elements into a single bin. If too many values end up in just a few bins, the hash map spends most of its time iterating through the lists and becomes a glorified linked list. As we know from the previous section, iterating through a linked list is a linear operation. A good hash map implementation will detect when the lists are getting too long and amend its hash function to redistribute the values over a larger number of bins. An occasional resizing keeps the lists short and ensures good performance. The cost of the resizing is amortised.

For an in depth analysis of how Ruby implemented a hash map similar to the above, I highly recommend *Ruby under the microscope* (see further reading). Ruby's implementation has changed somewhat since the book's publication but it's still well worth a read. In the further reading is a Heroku blog post giving a good overview of the changes that came with Ruby 2.4.

Abstract data types

An abstract data type is a data type without an implementation. They are useful because code using them doesn't have to know or care about the details of the implementation. If you find a more suitable implementation of the interface, you can swap it with the old one without having to modify any consuming code. As an example, see below how Java handles the `List` interface.

We'll briefly review some important abstract data types and how they can be implemented with the data types we've seen. We've only seen three data structures, but that's already enough to implement a few interesting ADTs. I'll capitalise the names of ADTs to distinguish them from data structures, which are uncapitalised.

The **Array** ADT is an ordered collection of elements with two operations:

```
get(array, index) // returns value at index
set(array, index, value) // writes value to index
```

The **List** ADT is an ordered collection of elements with three operations:

```
cons(value, list) // prepend value as new head
head(list) // returns head of list
tail(list) // returns tail of list
```

Don't worry if you find the distinction between Arrays and Lists confusing. They are very similar but have a difference in emphasis. The operations specified by the Array prioritise indexing anywhere in the Array, which favours an array data structure, while the operations specified by the List prioritise manipulating heads and tails, which favours linked lists. Partly it's a question of history. Lists originated with the programming language Lisp, which used linked lists (and is from where the `cons` term comes). The situation is more confused nowadays because Lists and Arrays seem to be almost synonymous. The Array ADT in JavaScript[9] and the List interface in Java[10] specify very similar sets of methods that combine operations from both.

One nice thing about Java is that it makes the distinction between a data type and a data structure clear when you initialise a variable. In Java ADTs are defined as *interfaces*, which are literally a set of method signatures that an implementing data structure needs to provide. It is not possible to directly create an interface. You need to create a data structure that implements the interface. In Java this is done by instantiating a class using `new`. Here's how you create a List of strings implemented by an array:

```
List<String> myStringList = new ArrayList<String>();
```

`ArrayList` is Java's name for a dynamic array. If you notice that your program spends a lot of time adding new elements to `myStringList` but rarely indexes into it, you can change the implementation to use a linked list simply by instantiating a different class:

[9] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array
[10] https://docs.oracle.com/javase/8/docs/api/java/util/List.html

```
List<String> myStringList = new LinkedList<String>();
```

The variable's type remains `List<String>` and so you can swap the implementing data structure without having to change the data type.

We already encountered the **Stack** in the previous chapter. To recap, a Stack is a collection of elements with two operations:

```
push(stack, value) // add value to collection
pop(stack) // remove most recently added value
```

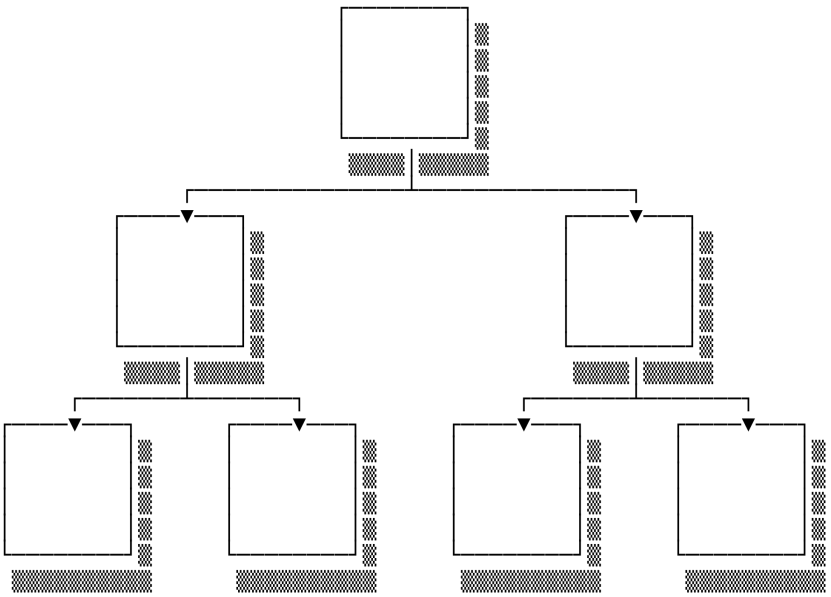
A `peek` operation, exposing the top value without removing it, is often present too. The Stack operates on the principle of last-in, first-out (LIFO). Stacks appear throughout computer science and we'll encounter them again numerous times in this book. They're particularly well suited to managing nested operations that exhibit LIFO behaviour. A Stack can be implemented with arrays or linked lists.

A **Queue** is another collection of elements with two operations:

```
enqueue(queue, value) // add value to back of queue
dequeue(queue) // remove value from front of queue
```

The Queue is so named because it provides first-in, first-out (FIFO) behaviour. It's just like a queue of people. As you might expect, Queues are used when it's important to process elements in the order in which they're added. Queues can be efficiently implemented by linked lists (single or double) as long as there is a reference to the final element as well as the head. They can be implemented by arrays with one index representing the head and another representing the final element. When an element is enqueued or dequeued the indexes are adjusted, wrapping from the end of the array back to the beginning if necessary. This avoids having to shuffle the values in the array.

A super cool and useful ADT is the **Tree**. Conceptually a Tree looks like this:



Structure of a tree

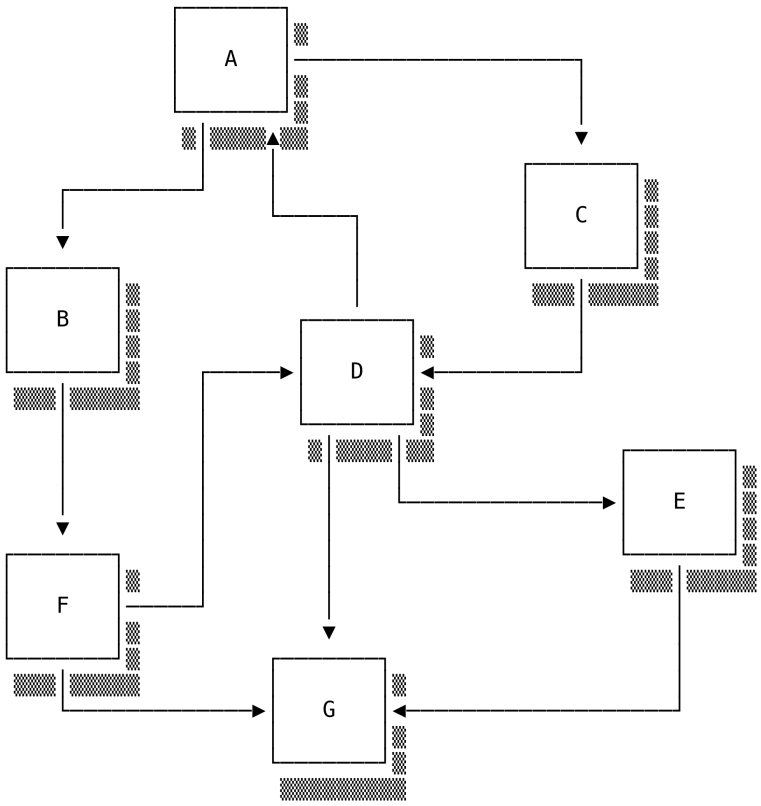
A Tree consists of a root node (the one at the top) with zero or more child nodes. Two nodes cannot have the same child. The structure is recursive because each child forms the root of its own subtree. The minimum necessary operations are therefore really simple:

```
value(node) // get value held in a node
children(node) // get child nodes from a node
```

One way of implementing a Tree is as a linked list where each element in the list can have zero or more pointers to other elements. In this view, a linked list is just a special Tree where every node has a single child.

Trees appear in many areas of computer science. They're an obvious natural fit for data with hierarchical relationships. A very useful variation is the **binary search tree** (BST). In a BST each node has at most two children. All of the values in the left subtree will be less than the current node's value and all of the values in the right subtree will be greater than the current node's value. A BST is structured so that traversing it is equivalent to a binary search, which we will look at in the next section.

Finally, a **Graph** is a Tree in which multiple nodes can point to the same child. It ends up more like a web of relationships:



An unweighted, directed, cyclic Graph

In the world of Graphs, the nodes are called **vertices** and the links between them **edges**. Graphs are very useful any time you need to model the relationships between elements. In an **undirected** Graph an edge between A and B implies a relationship both from A to B and from B to A. In a **directed** Graph the edges are more properly called **arrows** and only go in one direction. A social network might use a Graph to model the web of friendships between its users. An undirected Graph would only indicate friendship but a directed Graph could additionally indicate who "friended" whom on the website.

When there is a value associated with each edge or arrow, the Graph is **weighted** and **unweighted** if there is not. If you modelled towns as vertices, the edges between them could be weighted with the distance between each pair of towns. Finding the shortest route between two towns could then be solved by finding the path between the two vertices with the lowest total weighting. The Graph above is an unweighted, directed Graph.

Traversing a Graph is trickier than traversing a Tree because of the possibility of **cycles**: vertices that connect together in a loop. In the example above there are two cycles: A-B-F-D-A and A-C-D-A. When traversing a Graph, you must keep track of which vertices you have already visited to avoid infinitely looping around a cycle. A directed, acyclic Graph (DAG) has directed edges with no cycles. DAGs are commonly used to model different kinds of information with dependencies e.g. family trees and package manager dependencies.

A straightforward implementation of a Graph is via an **adjacency list**. Vertexes are stored in a list

and every vertex stores a list of the vertices it is adjacent to. There are many different ways of implementing Graphs, and indeed many ways of implementing adjacency lists, each with different performance trade-offs.

It's hard to overestimate how important Trees and particularly Graphs are in computer science. Graphs even have their own branch of mathematics called – you guessed it – graph theory. A vast number of concepts can be efficiently modelled with Trees and Graphs, sometimes in surprisingly elegant ways. In this short section I don't have enough space to do anything but whet your appetite. I strongly encourage you to check out the resources in the further reading section and explore these fascinating data types further.

Algorithms

An algorithm, as we saw in the previous chapter, is a sequence of instructions to carry out computation. A computer program is an algorithm written in code that the computer can understand. We've already seen how to perform algorithmic analysis and work out the performance characteristics of algorithms. In this section we'll look at two very important classes of algorithms: sorting and searching. I've chosen to focus on just these two because they are so fundamental to solving everyday programming problems but are also complex enough to demonstrate a range of design approaches.

Sorting

A sorting algorithm takes a collection of items and arranges them according to some ordering rule e.g. alphabetically or by size. Sorting is a fundamental operation because we often want to be able to impose an order on values. In addition, we can solve some problems with more efficient algorithms if we can provide a sorted input. There is no single best way to sort an input. Different algorithms perform differently, depending on factors such as the size of the input and how the values are distributed (is it mostly sorted already? Is it in reverse order?). Each algorithm will have different time and space requirements for different inputs. An algorithm that is optimal for sorting a small collection of values could be completely unsuitable for sorting very large collections.

The standard textbook approach is to prove the complexity of an algorithm and then implement it in C. I want to take a slightly different approach here. I'm assuming that you program in an environment that already includes fast sorting methods or functions. It might be intellectually interesting to implement a sorting algorithm yourself but you're vanishingly unlikely to ever need to do so in real code and so you'll quickly forget. I know that this has certainly happened to me (a few times!). Instead, we'll look at things from the other direction and see how some important applications choose to implement sorting. We'll dive down in the implementation of JavaScript sorting in Firefox and Chrome. We'll see which algorithms they chose, the performance implications and the trade-offs involved. My hope is that this approach will give you a more intuitive understanding of how sorting works.

Browsers have a lot of leeway in how they implement JavaScript. The specification only requires that it sorts *in place* by modifying the input array and that the default sorting method is lexical

(i.e. alphabetical). Since lexical sorting is pretty useless, `Array.sort` also takes an optional compare function. If `compare(a, b)` returns less than zero, then `a` is sorted before `b` and vice versa:

```
[1, 15, 2, 3, 25].sort()
> [ 1, 15, 2, 25, 3 ]
[1, 15, 2, 3, 25].sort((a, b) => a - b);
> [ 1, 2, 3, 15, 25 ]
```

As long as a browser implements this functionality, it can use whatever sorting algorithm it chooses. So, which ones do they choose and why?

If you dig into the Firefox source code[11] you'll see that it uses **merge sort**. Merge sort is a recursive algorithm that uses a design paradigm known as **divide and conquer**. We'll see other examples below. The idea behind divide and conquer is that you can more simply solve a problem for a large input by breaking it down into smaller sub-problems, solving those sub-problem and then combining each sub-result into the overall solution. Divide and conquer algorithms are often implemented using recursion.

Merge sort is based on the observation that it's easy to merge two sorted lists. You just compare the first element of each list, extract the lowest, add it to a new list and repeat until both inputs are empty. The algorithm works by recursively dividing the input into sublists, known as *runs*, until the runs only contain one element each. It then merges the two single-element runs into a sorted two-element run. Next, it merge those two runs to create a four-element run. It continues until the runs are all merged together into the final, sorted output. The algorithm's name comes from the fact that the sorting is actually performed when merging the runs. A Python implementation might look like this:

```
def merge_sort(input):
    if len(input) <= 1:
        return input # a run with one element is sorted

    middle = len(input) // 2
    left = input[:middle]
    right = input[middle:]

    left = merge_sort(left)
    right = merge_sort(right)
    return list(merge(left, right))
```

The recursive calls to `merge_sort` break the input down into smaller and smaller runs until they only hold a single element. Then each call uses `merge` to construct a sorted list out of `left` and `right`. Since we can rely on `left` and `right` always being sorted (a list of one element is sorted!), `merge` can easily

[11] https://dxr.mozilla.org/mozilla-central/source/js/src/builtin/Array.js#190

construct a new sorted list by repeatedly pulling the lower of the two elements from the heads of `left` and `right`.

Merge sort offers reliable $O(n \log n)$ time performance, which is very good for sorting algorithms. Its weakness is that when performed on arrays it requires a linear amount of memory to store all the runs it generates, though various optimisations exist. This is not such a problem with linked lists. In fact, merge sort is well suited to linked lists because it only ever needs to access the head of each list. Finally, merge sort scales well to huge inputs that don't fit in memory. To sort such an input, divide it into chunks that do fit in memory. Sort each chunk in memory and write the sorted output to a file. Then merge each sorted file together by the same process above to create the final output.

Another advantage of merge sort is that it is **stable**. Assume I have a list of JavaScript objects representing users with `age` and `surname` properties. Many users with different surnames will share the same age. I sort them first by surname and then by age. If the algorithm is stable, I will have a list of users sorted by surname and then, within each surname grouping, sorted by age. If the algorithm is unstable, then the users will be sorted only by age and no longer by surname. The benefit of a stable sort is that we can "stack" sorts by applying them one after the other. If the sort isn't stable we can't do this. The JavaScript specification does not currently require a stable sort but this will change in an upcoming revision. It's pretty clear why Firefox (and Safari, incidentally) use merge sort: it's reliably efficient and stable.

Next up is Chrome, which uses the V8 JavaScript engine. Before v7.0, the V8 engine had a particularly interesting implementation[12] of `Array.sort()`:

```
function ArraySort(comparefn) {
  var array = TO_OBJECT(this);
  var length = TO_LENGTH(array.length);
  return InnerArraySort(array, length, comparefn);
}

function InnerArraySort(array, length, comparefn) {
  // In-place QuickSort algorithm.
  // For short (length <= 10) arrays, insertion sort is used for efficiency.
  function QuickSort(a, from, to) {
    // ...
    while (true) {
      // Insertion sort is faster for short arrays.
      if (to - from <= 10) {
        InsertionSort(a, from, to);
        return;
      }
      // do Quicksort instead...
    }
```

[12] https://github.com/v8/v8/blob/6.0-lkgr/src/js/array.js#L758

```
    // ...
  }
  QuickSort(array, 0, num_non_undefined);
}
```

`ArraySort` calls into `InnerArraySort`, which calls into `QuickSort`, which then actually decides to call `InsertionSort` for short inputs. To understand why V8 does this, we need to look at the performance of the two algorithms.

Insertion sort is a very simple algorithm. The process is similar to how many people sort a hand of cards. Since I'm left handed, I designate the leftmost card to be the first sorted card. I then take each card from the unsorted section and insert it into the sorted section in the correct position. Eventually, all cards are sorted. An implementation in code iterates through the input and moves each element to the correct location in the output. This could be an entirely new array or, more commonly, the algorithm shuffles the elements of the input around to keep the sorted ones at the beginning. This means that the algorithm doesn't require any extra memory. Here is a neat solution in Go:

```
func insertionSort(a []int) {
    for i := 1; i < len(a); i++ {
        value := a[i]
        j := i - 1
        for j >= 0 && a[j] > value {
            a[j+1] = a[j]
            j = j - 1
        }
        a[j+1] = value
    }
}
```

Note that this implementation designates `a[0]` as the first element of the sorted output. At each iteration, the outer loop increases the size of the sorted section and assigns the newly included element to `value`. The inner loop moves down through the sorted section and ensures that `value` ends up in the correct position. This maintains the sorting. The nested `for` loops should tell you that insertion sort's average time performance isn't that great: $O(n^2)$. At least insertion sort is stable.

Insertion sort's quadratic performance is much slower than **Quicksort**, a venerable algorithm that has been in the C standard library for decades. Quicksort is another example of a recursive, divide and conquer algorithm. Quicksort works by dividing the input into two sections around a *pivot*. Values that are bigger than the pivot are moved above it and values that are smaller are moved below. All of the values in the top section are now greater than all of the values in the bottom section but each section needs to be internally sorted. The algorithm recursively calls itself on the two sections until everything is completely sorted. At each step, the size of the problem is halved (in the best case where the value of the pivot is exactly in the middle of the input range). This is characteristic of divide and conquer algorithms. Its implementation in Haskell is remarkably clear:

```
qsort [] = []
qsort (x:xs) = qsort [y | y <- xs, y < x] ++ [x] ++ qsort [y | y <- xs, y >= x]
```

Remember that `x:xs` means `x` is the head of the list and `xs` is the tail (i.e. every other value). As described above, we set `x` to be the pivot. You can see it in the middle. Below the pivot, we use `[y | y <- xs, y < x]` to make a list of every value from `xs` less than `x`. Above the pivot, we make another list of values greater than or equal to `x`. We recursively call `qsort` on the two sections and concatenate the results to the pivot using `++`.

Quicksort usually offers $O(n \log n)$ performance but can degrade to $O(n^2)$ performance when the pivot value happens to be the smallest or largest value in the list. That creates one section holding only one element and another nearly as long as the original input, rather than creating two sections equal to half of the original. Recursively dividing the problem into smaller sub-problems is logarithmic in nature and so Quicksort will make $\log n$ function calls. As we'll see in the operating systems chapter, each function call consumes some memory, so Quicksort has $O(logn)$ space requirements.

If Quicksort's worst-case performance is the same as insertion sort's best-case performance, why did V8 use insertion sort at all? Insertion sort does have something to bring to the party. It performs very well on small inputs, when even a quadratic algorithm performs quickly. When the input size is small, the amount of time spent actually sorting is overwhelmed by the overhead of the recursive function calls in divide and conquer algorithms like Quicksort or merge sort. Insertion sort doesn't need to use recursion and so avoids the overhead. I think this is a great example of why we should take care not to put too much emphasis on big-O notation. It's very approximate and actual performance can be different, especially with small input sizes. The V8 team did their measurements and found that the best way to handle the disadvantages of both algorithms was to use them in concert.

Unfortunately, this pedagogical case study is no more. The problem with Quicksort is that it is not stable. Calling `Array.sort()` in Chrome would give different results to other browsers that used stable sorting algorithms. The V8 team were keen to use a stable sorting algorithm for consistency but wanted to keep the performance they had achieved with their insertion/Quicksort combo. Their solution was to switch to *TimSort*, an engagingly-named but vastly more complex algorithm. The team wrote a very interesting blog post about the transition[13].

TimSort is also the sorting algorithm used by Python. A quick look at Ruby's source code[14] reveals that `Array#sort!` uses Quicksort:

[13] https://v8.dev/blog/array-sort
[14] https://github.com/ruby/ruby/blob/a47d6c256ff684392f00516a917735a14aec64b0/array.c#L2777

```
VALUE
rb_ary_sort_bang(VALUE ary)
{
    // loads of stuff ...
    VALUE tmp = ary_make_substitution(ary);
    struct ary_sort_data data;
    long len = RARRAY_LEN(ary);

    RARRAY_PTR_USE(tmp, ptr, {
        ruby_qsort(ptr, len, sizeof(VALUE),
        rb_block_given_p()?sort_1:sort_2, &data);
    });

    // loads more stuff...
}
```

The C code is pretty hard to read but you can see the call to `ruby_qsort` lurking in there, which in turn calls out to the operating system's Quicksort implementation or falls back on its own.

And finally, here is a summary of the algorithmic complexity of the three sorting algorithms we've seen. From our survey of browser implementations, we know that big-O notation is not the last word on performance. Nevertheless, it's useful to have a rough idea of how the algorithms perform:

Algorithm	Best case	Average case	Worst case	Space requirements	Stable?
Quicksort	O(n log n)	O(n log n)	$O(n^2)$	O(log n)	no
Merge sort	O(n log n)	O(n log n)	O(n log n)	O(n)	yes
Insertion sort	O(n)	$O(n^2)$	$O(n^2)$	O(1)	yes

Searching

A search algorithm looks through an input, known as the **problem domain** or **problem space**, to find a solution. The problem domain might be an actual list of values provided to the algorithm but it might also be more abstract, such as the set of possible configurations of a chess board. In this section, we'll stick to the straightforward meaning of looking through an input collection to find a specified value. There are two main approaches: linear and binary search.

A **linear search** is the most straightforward way of finding something. You start at the beginning and iterate through to the end, checking each value as you go until you either find what you're looking for or reach the end of the input. As you might have guessed, a linear search offers linear performance. A second attribute of the algorithm is that it will always find the first matching value. This is super useful with pre-sorted inputs. For example, to find the first occurrence of an error message in a list of log messages you could sort by timestamp and then perform a linear search for the error message.

Ruby's `Array#any?` method is an example of linear search:

```
static VALUE
rb_ary_any_p(int argc, VALUE *argv, VALUE ary)
{
    // ...
    else {
        for (i = 0; i < RARRAY_LEN(ary); ++i) {
            if (RTEST(rb_yield(RARRAY_AREF(ary, i)))) return Qtrue;
        }
    }
    return Qfalse;
}
```

The function is implemented in C, so it's a little hard to read, but you can see that it iterates through the input array and tests each value, returning `QTrue` as soon as it finds a match.

Can we improve on linear search? If the input is unsorted, there's not really much we can do. We cannot make any assumption about where in the input our sought after value might be. If we had a sorted input we could make educated guesses about where the value might be. Take the example of a dictionary. If you're looking for the word "sort", you don't start at the beginning and look through every page. You know that words beginning with "s" are going to be towards the end and so you open the dictionary at roughly the right place. If you overshoot on either side you compensate by moving a little bit in the other direction and repeat the process until you land on the right page. That is how binary search works.

Expressed as an algorithm, it compares the value to the midpoint of a sorted input. If they match, the value has been found. If the value is greater than the midpoint, then the algorithm recursively calls itself on the top half of the input. If the value is smaller, then it calls itself on the bottom half. At each step, the algorithm halves the search space by discarding values that it knows can't contain the value. This should be ringing some bells in your mind. It's a divide and conquer algorithm, just like Quicksort! The binary search tree we saw previously a data structure designed so that traversing through it is equivalent to performing a binary search. It's a neat example of how a data structure can be designed to optimise for specific operations.

Since it's a divide and conquer algorithm, binary search has logarithmic time performance. Although much faster than a linear search, this comes at the cost of requiring the input to be sorted. Whether it is better to perform a linear search or first sort and then perform binary search will depend on the size of the input and how many searches are required.

Ruby offers its own implementation of binary search in the form of `Array#bsearch`:

```
while (low < high) {
    mid = low + ((high - low) / 2);
    val = rb_ary_entry(ary, mid);
    v = rb_yield(val);
    // ...
    if (rb_obj_is_kind_of(v, rb_cNumeric)) {
        const VALUE zero = INT2FIX(0);
        switch (rb_cmpint(rb_funcallv(v, id_cmp, 1, &zero), v, zero)) {
          case 0: return INT2FIX(mid);
          case 1: smaller = 1; break;
          case -1: smaller = 0;
        }
    }
    // ...
    if (smaller) {
        high = mid;
    }
    else {
        low = mid + 1;
    }
}
if (!satisfied) return Qnil;
return INT2FIX(low);
```

I've tried to cut out all but the essential code showing how the algorithm works. You can see that it compares the midpoint to the search value (in the `switch` statement) and redefines the `high` and `low` limits of the search space accordingly.

Binary search is an example of a more general optimisation technique known as **hill climbing**. Imagine that the answer we're looking for is on the summit of a hill. We are on the side of the hill but thick fog hides the summit. Even if we don't know where the summit is, a sensible assumption would be that going up the hill will take us closer to the summit. Binary search does something similar when it compares the sought after value to the midpoint. At each iteration the algorithm can halve the size of the problem space by selecting the half that must contain the value.

Hill climbing is a useful technique when you need to find a good solution to an NP-complete problem. The excuse "sorry, that's intractable" isn't going to wash with your non-technical manager so you need to come up with something that gets close to the optimum. To apply the hill climbing technique, first generate any correct solution. For example, if you're dealing with the travelling salesman problem, just find any valid route that goes through all of the towns. Then make a single change, such as swapping around two towns. Does that lead to a better or worse solution? If worse, undo it. If better, keep it and now make another change. Gradually, over repeated iterations, the algorithm produces an improved solution that approximates the optimum. It's unlikely to be the optimal one but it can get very close in a short time.

Conclusion

In this chapter we looked at the distinction between data types and data structures. We saw that data structures implement data types. We examined some fundamental data structures – the array, linked list and hash map – and analysed their performance characteristics. We saw how dynamic arrays are constructed from static arrays, using Go as an example. We then reviewed some very important abstract data types and got a sense of how they can be implemented and the problems they solve. We then examined and compared the implementation of JavaScript sorting algorithms in the major browsers. Finally, we finished off by looking at the two main ways of performing searches, linear and binary, and saw how the Ruby standard library implements both. We saw that divide and conquer algorithms work by splitting a large problem into many smaller ones, often leading to more efficient solutions. Binary search is an example of a more general optimisation technique known as hill climbing.

And with that we reach the end of the theory! I hope you found the theory interesting but if you're itching to start more practical topics, read on. Our next step is to apply theory to reality and get into the nuts and bolts and bits and bytes of real computers.

Further reading

Computer science courses often approach algorithms and data structures from a mathematical perspective. The aim is to analyse them as mathematical objects to prove their performance characteristics and correctness. If you want to take this approach, *Algorithms* by Robert Sedgewick provides a wide-ranging overview of important algorithms and *Introduction to Algorithms* by Thomas Cormen goes heavy on the mathematical analysis. Sedgewick also offers a companion series of MOOCs on Coursera[15].

As I've already said, unless you're particularly interested in the mathematical side of things, I think a more hands-on approach is a much better way to gain confidence working with algorithms and data structures. Learn by doing.

First of all, Algorithms[16] by Jeff Erickson is a freely available, comparatively streamlined introduction to algorithms. Open Data Structures[17] by Pat Morin is another free resource that covers every data structure you're likely to come across. (Clearly, innovative names for algorithm and data structures textbooks are in short supply.) Skim through these texts to get an idea of what algorithms and data structures exist and how they can be categorised. Don't worry about remembering every detail or memorising implementations. Pat Shaughnessy has made the hash chapter of *Ruby under the Microscope* available online[18]. Even if you don't know Ruby, it's very instructive to see how a real programming language handles hashes.

[15] https://www.coursera.org/learn/algorithms-part1
[16] http://jeffe.cs.illinois.edu/teaching/algorithms/
[17] https://opendatastructures.org/
[18] http://patshaughnessy.net/Ruby-Under-a-Microscope-Rough-Draft-May.pdf

Now practice implementing important algorithms and data structures. The aim is not to rote learn implementations, but instead to practice working with them until you develop an intuition for how they work. The implementations of many algorithms and data structure operations are actually pretty straightforward when you have a good mental model of how they work. So, where to get practice? A great starting point is Harvard's CS50[19] MOOC. They've created lots of instruction material and you will implement some common data structures and sorting algorithms in C. Many people find algorithms easier to grasp visually. Sorting.at[20] generates pretty visualisations for more than a dozen sorting algorithms. If you want to prepare for algorithm-focused job interviews, the go-to resource is *Cracking the Coding Interview* by Gayle Laakmann McDowell. You will probably also need to spend some time grinding through problems on HackerRank[21] or Leetcode[22]. I must admit, though, that they're rather tedious and unimaginative.

My main suggestion is therefore a bit left field. Once you've got a bit of practice under your belt, try to get access to Google's Foobar program. If you've got a search history full of programming terms, searching for "python list comprehension" should show you an invitation. Treat each challenge as a learning opportunity: if you can't solve it at first, find a working solution online, read up on the related terms you find (e.g. dynamic programming, the Bellman-Ford algorithm) and re-implement the solution yourself. You'll gain a much better intuition for the concepts this way. Personally, I found the cute challenge descriptions helped me remember the problem and the relevant algorithms.

Finally, I'm a fan of Mazes for Programmers[23] by Jamis Buck. It turns out that mazes can be represented as trees or graphs. Writing algorithms to draw beautiful mazes and then solve them is a wonderfully fun way to develop your intuition for graphs and their associated algorithms.

[19] https://cs50.harvard.edu/
[20] http://sorting.at/
[21] https://www.hackerrank.com/
[22] https://leetcode.com
[23] http://www.mazesforprogrammers.com/

Computer architecture

Introduction

In this chapter we will look at the architecture, or fundamental structure, of modern computers. In the previous chapters we examined the theoretical underpinnings of *why* computers work. By the end of this chapter, you will have a much better understanding of *how* computers work.

We will begin by studying the micro-architecture of computing components. That covers the very low level, fundamental questions of how information can be physically encoded and computed by a digital circuit. We'll look at binary and hexadecimal number systems and study how computational instructions are encoded. We will study logic gates, the basic building blocks of digital circuits, to understand how computational and storage units are constructed. Having covered that, we'll expand our focus to the overall computer architecture. To understand exactly how computation happens, we'll dive into the internal architecture and operation of the processor. We will also look at the other essential component, memory, and how it is arranged into a hierarchy of caches.

Modern computing components, in particular processors, are incredibly sophisticated. We'll mostly focus on a very simplified architecture so that you can understand the overall design without getting too bogged down in various implementation details. Nevertheless, we'll round off the chapter with a discussion of some sophisticated optimisations found in real world systems.

At various points in the chapter we'll use the example of a simple addition instruction to illustrate what's going on.

Representing information

The definition of a Turing machine is a bit vague about how information should be provided to the machine. It just states that "symbols" are written on the input tape. Which symbols should we use? Over the years, computer designers experimented with a number of approaches but settled fairly quickly, for reasons we'll see, on **binary encoding**. Let's look at what that means.

Number bases

A single binary value is known as a **bit**. It can only be one of two values: `0` or `1`. The equivalent in the decimal system is a digit, which can be one of ten values: 0-9. Having only two possible values doesn't seem all that great. How might we express a value greater than `2` in binary? Well, when we want to express a value bigger than `9` in decimal, we just add more digits. Each new digit allows ten times more numbers:

100s	10s	1s	number of possible values
		9	10
	9	9	100
9	9	9	1000

You might remember learning in school that the rightmost column in a number is the units, the next one to the left is the tens, the next the hundreds and so on. In general, n columns can express 10^n values. The principle is exactly the same in binary, except we use 2^n:

8s	4s	2s	1s	number of possible values
			1	2
		1	1	4
	1	1	1	8
1	1	1	1	16

The value of n is known as the **base** of the counting system. Decimal uses 10 as its base. Binary uses 2.

Try writing out every combination of three bits (`000`, `001`, `010` and so on). You will find that there are eight. Adding an extra bit doubles the number of possible values. So adding more bits allows for a greater range of values. The computing industry has managed to agree that eight bits, known as a **byte**, are the minimum value size. Bytes form the basic building block of every value and data type.

Why eight and not seven or nine? It's mostly a convention, though there are some advantages to eight bits. As we'll see shortly, components capable of handling large values are frequently constructed by combining a pair of smaller components. Each pairing doubles the value size, meaning that values that are a power of two, such as 8 (2^3), are a more natural fit.

Bytes can be written bit by individual bit. Here is 255: `0b11111111`. Binary numbers are conventionally written with a leading `0b`, indicating that `0b11` represents binary three and not decimal eleven. However, for humans it's slow and difficult to read long sequences of 1s and 0s. Is `0b1111111111111111` the same as `0b111111111111111`? It's difficult to tell without counting each bit.

Hexadecimal is an alternative number system that uses sixteen as its base. It uses the standard 0-9 for the first ten values and then A-F to express the remaining six (10-15 in decimal). Hexadecimal values are conventionally prefixed with `0x` to make clear that we're working with hexadecimal.

4,096s	256s	16s	1s	number of possible values
			F	16
		F	F	256
	F	F	F	4,096
F	F	F	F	65,536

Looking at the tables above, we see that four bits (`0b1111`) can hold the same number of values as a single hexadecimal value (`0xF`). This means that a byte can be expressed as two hexadecimal values, which is much easier to read. Here is how we convert from binary to hexadecimal:

1. Take a byte value e.g. `0b10110011` and split into halves: `0b1011 0b0011`
2. Convert each half to decimal: `11 3`
3. Convert each half to hexadecimal: `0xB 0x3`
4. Squish halves together: `0xB3`

It's important to see that expressing a binary number in hexadecimal doesn't change its value. It's just a different way of expressing the same thing. `0x54` is not the same as decimal 54. Can you work out what it is in decimal? We have four ones and then five sixteens, so it is $5 \times 16 + 4 \times 1$ or 84 in decimal.

As a web developer, your main exposure to hexadecimal is almost certainly going to be CSS colour values, which are RGB (red, green, blue) colours encoded in hexadecimal. The RGB format has one byte encoding the amount of red, one for the amount of green and one for the amount of blue. White is therefore full red, full green and full blue: `0xFF FF FF`. Black is no red, no green and no blue: `0x00 00 00`. Armed with this understanding of hexadecimal, you can impress your friends and colleagues by editing colour values directly! You might have a grey like `0xADADAD` but think that it looks a little dark. You can brighten it by increasing the value for each component by the same amount: `0xDEDEDE`. Hours of fun!

According to how I've written the binary numbers, as you move from right to left, each column represents a value double the size of the previous one. This matches how we write decimal numbers, where the most significant digit comes first: in 123 the 1 represents hundreds and the 2 represents tens. The 1 is more significant because the number it represents (100) is bigger than the number the 2 represents (20). This is just a convention. Everyone could decide tomorrow to do things in reverse and write 123 as 321 and everything would be fine as everyone followed the same convention. Sadly, the computing industry hasn't quite reached the same level of cohesion on how to order the bytes in binary numbers. In a binary number made up of multiple bytes, one will be the most significant byte (MSB) and another will be the least significant byte (LSB). In the number `0x5566`, `0x55` is most significant because it represents `0x5500`, which is bigger than `0x0066`. In a system using **big endian ordering**, that number would be represented with the following two bytes: `[0x55, 0x66]`. In a **little endian** system it will be represented the other way around: `[0x66, 0x55]` Nowadays the majority of processors are little endian but network protocols are big endian, just to keep people on their toes.

Negative numbers

We've seen that a byte can express a value between 0 and 255. We can encode bigger numbers by sticking bytes together. Two bytes is sixteen bits and so can encode 2^{16} (65,536) values. We can always express a bigger positive integer by adding more bytes. How can we go in the other direction and express a negative number? All we need is an agreed way to encode them. The most common encoding is known as **two's complement**.

To encode a negative number in two's complement, just take the positive number, flip all the bits and add one. So to encode -2:

1. Encode $+2$ in binary: 0b00000010

2. Flip each bit: 0b11111101
3. Add one: 0b11111110

The reason for using this particular encoding is that negative and positive numbers can be added, subtracted and multiplied together as normal without any special handling. Look at what happens when we add 1 to -2 three times. Things just work out properly:

```
0b11111110 + 0b00000001 = 0b11111111 // -1
0b11111111 + 0b00000001 = 0b00000000 // 0
0b00000000 + 0b00000001 = 0b00000001 // 1
```

There are two interesting things to note. Firstly, this is where the distinction between **signed** and **unsigned** integers comes from. An unsigned integer holds only positive numbers. A signed integer uses two's complement to encode both positive and negative numbers. This means it can't hold as many positive numbers because half of the possible bit patterns are for negative numbers. For example, an unsigned byte can hold 0 to 255 and a signed byte can hold -128 to 127.

Secondly, observe that there is nothing inherent in `0b11111110` that makes it a negative number. It all depends on how we instruct the computer to interpret it. If we say it's an unsigned byte, the computer will interpret it as 254. If we say it's a signed byte, it will interpret it as -2. Everything is just a series of bits to the computer. It is the programmer who imposes meaning by instructing the computer how to interpret them.

Decimal numbers

All of the above applies only to whole numbers. Decimal numbers, such as 1.222, are encoded in a representation known as **floating point**. The value is stored as a significand scaled by an exponent. 1.222 would be represented as 1222×10^{-3}. The name comes from the fact that the decimal point moves around depending on the exponent's value.

The exact details of how values are encoded will vary with the processor's floating point implementation, but they all suffer from the same problem of imprecision. Between any two decimal values there is an infinite number of other decimals. It's not at all obvious how the finite number of bit patterns should map to these infinite values.

In order to encode a value into a fixed number of bits (normally 32 or 64), the significand must be approximated to a fixed number of bits. For example, imagine that a decimal floating point encoding only allowed four digits for the significand. 1.2223 and 1.2219 would be rounded down and up respectively to 1222×10^{-3}. Floating point values are therefore imprecise. The deviation compounds when you perform arithmetic and this can lead to confusing results. In my Firefox browser, 3.0×2.02 does not equal 6.06, as you would expect, but 6.0600000000000005. Whenever you perform calculations that need to be exact, such as when dealing with money, it is better to convert the values to whole numbers (e.g. convert dollars to cents), perform the calculation and then reverse the scaling.

Instruction sets

Recall the Turing machine. It reads a series of symbols from the input tape. Each symbol causes the finite state control to respond in a certain way. We can therefore think of the symbols as **instructions** to the Turing machine. To implement instructions for a real computer, we must first answer two questions: what instructions do we have and how should we encode them in binary? Let's use the example of an instruction to add two numbers together. We now know that the two values can be stored in binary-encoded bytes. Since the computer only understands binary, the instruction must also be in a binary form. Binary-encoded instructions are known as **machine code**.

Each processor accepts an **instruction set** of valid, binary-encoded instructions. The architecture of the processor – its internal components and how they are wired together – determines its capabilities and so defines the set of valid instruction and how they are encoded in binary. Naturally, it wouldn't make sense for the instruction set to include an instruction which the processor lacked the circuitry to execute. The problem then is that every processor design would have a different architecture and so a completely different instruction set. Writing portable code would be exceptionally difficult.

The **instruction set architecture** (ISA) provides the missing layer of abstraction. It's an abstract model of the processor's architecture that separates the external, programmer-facing interface from the internal implementation details. Processors can have different internal architectures but still implement the same ISA, meaning that they are **binary compatible**: machine code for one processor will run on the other without modification.

Certain ISAs have become dominant within the computer industry. Since code written for one ISA is not compatible with other ISAs, if you're an upstart processor manufacturer it makes sense to design a processor that implements the incumbent's ISA. Your processor will be more attractive if it can run existing code without modification and you can still compete on cost or implementation performance. This is exactly what AMD did when they took on Intel. Intel's x86 instruction set was so dominant that AMD had to make sure their processors implemented the x86 interface, even if they worked very differently internally.

Let's look at two popular ISAs: x86 and ARM. They are both extremely popular and represent two opposing design philosophies. They are mutually incompatible. x86 code will not run on an ARM processor and vice versa.

Processor manufacturers initially competed by adding more and more complicated instructions to their ISA. The thinking was that a programmer would prefer a processor that offered sophisticated instructions for things such as finding a square root because that would mean less work for the programmer. Rather than having to write and test a square root function, they could just use the built-in instruction. The problem is that more complex instructions require more complex circuitry, making the processor bigger, slower and more expensive. This approach is known as **complex instruction set computing** (CISC). Intel's x86 is the classic example of a CISC ISA. It began with Intel's wildly popular 8086 processor in 1978. That processor's instruction set is actually relatively straightforward by modern standards, but over the years more and more backwards-compatible extensions were added and what we have now is a bit of a beast. The majority of desktop and laptop computers contain x86 processors.

Another approach is to provide a relatively limited set of simple instructions. If the programmer wants to do anything fancy, they'll have to implement it themselves out of these building blocks. The benefit of this is that the processor has a less complex design, making it easier to optimise for size, speed and energy consumption. This approach is called **reduced instruction set computing** (RISC) because the set of available instructions is deliberately reduced compared to what a CISC processor offers. A program written for a RISC processor will have more instructions than one written for a CISC processor. However, with time it's proven easier to optimise RISC processors and so the RISC processor will be able to churn through those instructions at a faster rate. This is the approach that ARM processors take. Their high efficiency means they are particularly attractive for mobile devices. The majority of phones and tablets use processors that implement the ARM instruction set, making it the most popular ISA overall.

RISC versus CISC was one of those interminable wars that the computer industry loves to wage against itself. As time has passed, the sharp distinction between the two philosophies has blurred somewhat. ARM has picked up various extensions that have added complexity. Intel found the x86 ISA so difficult to optimise that they actually added an extra step (invisible to the programmer) that converts x86 instructions into a RISC-like internal instruction set. It's these *micro-ops* that are actually executed by Intel processors. This further demonstrates how ISAs act as an interface. Just as with a programming interface, it's possible for a processor designer to completely change the interface's implementation, as long as the externally-facing parts remain constant.

The MIPS ISA

In this section we will study the MIPS ISA to better understand how an ISA is designed. I'm using the MIPS ISA because it's specifically designed for teaching purposes. It's comparatively simple but has enough complexity to demonstrate the tricky design challenges.

Very roughly, processor instructions fit into three categories: computation, control flow and memory operations. In MIPS these correspond to R instructions (for "register"), J instructions (for "jump") and I instructions (for "immediate"). Every MIPS instruction is a sequence of 32 bits. Each instruction format specifies how the bit sequence is divided into fields specifying various locations, operations and so on.

It's difficult for humans to look at a sequence of 32 bits and determine where one field starts and the other ends. Instructions are therefore usually expressed in **assembly language**. In assembly, each instruction is given a short mnemonic with arguments corresponding to the instruction's bit fields. Assembly is not a programming language per se in the same vein as C or JavaScript. Each ISA will have its own, unique assembly language. A program written in one processor's assembly will not work on another. Translating assembly code into machine code is a relatively straightforward task performed by an **assembler**. For each of the instructions examined below, I've also included their MIPS assembly name.

To achieve our goal of adding two numbers, we need to use an `ADD` instruction, which is an R-format instruction.

MIPS R instructions:

A **register** is a special memory location which holds a single, working value. They are located right in the heart of the processor, meaning that they can be accessed extremely quickly but there is only enough space for a few dozen. The processor cannot operate directly on values stored in memory. Whenever it wants to operate on a value, it has to first load that value from system memory into a register. The output of the instruction's execution is also written into a register, from where another instruction can write it back to memory.

The size of the registers determines the **word size**, which is the processor's unit size for data. It is nearly always a multiple of bytes. MIPS has a 32 bit word size so its registers are 32 bits wide.

In MIPS, the R instruction specifies two registers in which the input values are expected to be stored, a destination register and an **opcode** which specifies the operation to perform. Common operations include addition, multiplication, negation and logical bitwise operations (e.g. AND, OR). MIPS doesn't provide a subtraction operation because that doesn't need a dedicated instruction. You can subtract by negating one input and then adding – an example of a reduced instruction set!

opcode	rs	rt	rd	shift (shamt)	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

We can see that an R instruction, like all MIPS instructions, is 32 bits long. The first 6 bits specify the opcode. 6 bits means we can have 2^6, or 64, different operations. Next, we have the three registers. `rs` and `rt` are inputs and `rd` is the destination. 5 bits per register mean we can specify up to 32 different registers. We'll skip over the `shamt` as it's not relevant. Finally, we have 6 bits for the `funct`. Being able to specify 64 opcodes seems like a lot, but it's actually fairly limiting. To get around this, some instructions share an opcode but are uniquely identified by the `funct` at the end. All R instructions share the same `0b000000` opcode. We'll look at why there's this split between the `opcode` and `funct` fields later.

Here's how our `ADD` instruction looks in MIPS assembly:

```
// means a comment
ADD $d, $s, $t // $d = $s + $t
```

And the binary encoding:

```
0000 00ss ssst tttt dddd d000 0010 0000
```

The processor knows it's an R instruction because it has `0b000000` at the left. It then knows that the bits in the middle encode the various registers. It can identify that it's an `ADD` instruction because the `funct` field on the right contains `0b100000`, which the MIPS ISA defines to be the `funct` code for addition.

We can create instructions for other R operations just by changing the `funct` value. For example, `0b100101` is the `funct` code for bitwise OR, which compares two values bit by bit and writes a `1` to the corresponding destination bit if either of the inputs are `1` and a `0` if not.

MIPS J instructions:

Normally the processor executes instructions one after the other. Sometimes we need the ability to move, or **jump**, to a different instruction. Perhaps we want to execute a function located elsewhere in memory. J instructions tell the processor to jump to the instruction at a specified address.

The encoding of J instructions is much simpler. Apart from the opcode, the whole instruction is given over to specifying the memory address to jump to. The instruction directly encodes the address:

opcode	pseudo-address
6 bits	26 bits

```
J 0x0003
0000 1000 0000 0000 0000 0000 0000 0011
```

The processor takes the 26-bit pseudo-address, does a little bit of processing to generate the actual address and sets the result as the next instruction to be executed. The processor knows that it has a J instruction because the six bits on the left are `0b000010`.

MIPS I instructions:

We are missing two crucial abilities: making decisions and moving values in and out of memory. In MIPS, such instructions are in the I, or immediate, format. They are a hybrid of the previous two. They can specify a number of registers for argument values, just like the R format, but they also contain a offset value encoded directly, or *immediately*, in the instruction:

opcode	rt	rs	offset
6 bits	5 bits	5 bits	16 bits

As mentioned above, a value needs to be loaded into a register before the processor can operate on it. This is handled by the `LB` (load byte) instruction:

```
LB $t, offset($s) // $t = MEMORY[$s + offset]
1000 00ss ssst tttt iiii iiii iiii iiii
```

Processors often offer a variety of sometimes quite complex ways to compute a memory address. They are all designed to suit a particular use case. Here we expect `$s` to already contain a memory address. The offset is added to the address and then the value at that location loaded into `$t`. This

may initially appear a convoluted way of doing things. Why not simply specify the address in the instruction? This instruction is useful because we often know that a value will be at a fixed offset to some base element but we don't know where the base will be located in memory. Can you think of when this situation might occur? Referring back to the chapter on data structures, we iterate through an array by loading the base value into `$s` and adding offsets encoded directly in the instruction.

When we execute a conditional statement, we perform a **branch**. `BEQ` (branch on equal) is a MIPS conditional that compares the values in two registers and if they are equal branches, or jumps, to a specified offset from the current instruction. Otherwise, the subsequent instruction is executed as normal. MIPS also includes `BNE` (branch on not equal) and instructions to branch if one value is less than another.

```
BEQ $s, $t, offset // if $s == $t then jump by the specified offset else continue
0001 00ss ssst tttt iiii iiii iiii iiii
```

Standard if/else control flow can be encoded in branch instructions by putting the "else" branch directly after the conditional and the "if" branch after the "else" branch. Since we know the size of the "else" branch, we can encode the offset to the "if" code in the instruction. If the condition holds, the processor will jump to that location and execute the "if" branch instructions. If not, it will continue as normal to the "else" branch instructions.

Some ISAs, like x86, have variable-length instructions. In MIPS, each instruction is the same length because that makes the circuitry easier to design. A fixed length means that you have to be clever to make the best use of the space available. This explains why the R instructions have both an opcode and a function specifier. The obvious approach would be to just make the opcode long enough for every instruction to have a unique opcode. The problem is that a longer opcode in every instruction would reduce the number of bits available in the J and C instructions for the memory addresses and offset values. Since function selection is only required for R instructions, it makes more sense to shorten the opcode as much as possible and only include the additional `funct` field in the R instructions, where it's actually needed.

Circuits and computation

We now know how data and instructions can be encoded efficiently in a binary format. Why go to all the bother of using binary in the first place?

In theory, there's nothing preventing us from making a computer that uses ten different voltage levels to encode decimal values. In fact, some early computers did so. The decision to use binary encoding is really just a practical engineering decision. Binary and decimal encodings have the same capabilities – there's nothing that can be done in binary that can't be done in decimal, and vice versa – so the decision to use binary is based on what's physically easier and cheaper to manufacture.

You might remember making simple electrical circuits in school. You hook up a little light bulb to a battery, press a switch and the bulb shines! Such circuits are **analogue** because the voltage across

the circuit has an infinite number of possible values. We could encode bits in an analogue circuit by declaring that one voltage level represents binary 1 and another represents binary 0. Whenever the voltage is at either level, the circuit is outputting the corresponding bit. The problem with an analogue circuit is that there will often be situations where the voltage doesn't match our specified values. What we really want is a **digital** circuit in which the voltage can only be one of two possible values. To do this we need components, known as **transistors**, which can only output two voltage levels. One of the great engineering achievements of the 20th century was the production of very cheap and very small transistors.

Logic gates

Transistors can be combined together to make basic components called **gates**. A gate receives a number of input voltages and will output the voltage corresponding to either 0 or 1 depending on the inputs it receives. This is how we represent elementary logic in a physical circuit. The internal arrangement of transistors within the gate determines how it responds to input. Circuits using these components implement what is known as **combinational logic**. There is no concept of state or time within these circuits. Changing the input values immediately leads to a different output.

Some transistor arrangements are particularly useful and have been given names. A NOT gate takes one input and outputs the opposite. An AND gate takes two inputs and outputs 1 if *both* inputs are 1. An OR gate will output 1 if *either* of its inputs is 1. A NAND gate does the opposite of an AND gate: it outputs 1 unless both inputs are 1. With these simple components, we can construct circuits implementing arbitrary logical statements simply by arranging gates in the correct order. One of the interesting properties of these gates is that they can all be constructed from just NAND gates. It's fun to see how this works.

Below is the **truth table** of a NAND gate. It shows a NAND gate's output for all possible inputs:

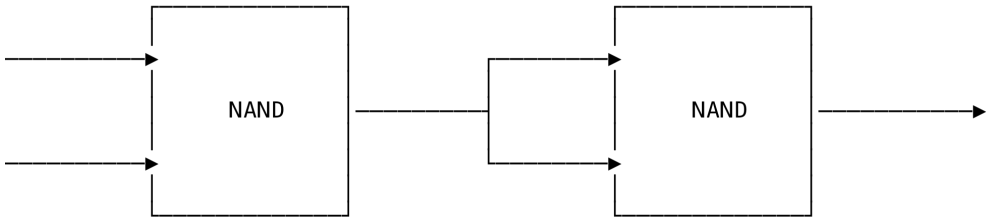
input A	input B	output
0	0	1
0	1	1
1	0	1
1	1	0

Below is the truth table for an AND gate:

input A	input B	output
0	0	0
0	1	0
1	0	0
1	1	1

How could you arrange NAND gates so that the circuit implements the AND truth table?

Well, clearly the output of AND is the inverse of NAND. One solution would be to pass our inputs to a NAND gate and then invert the output. How do we perform the inversion? We need to make a NOT gate and then feed the NAND output into the NOT gate. Looking more closely at the NAND truth table, we see that the output of a NAND gate is always the opposite of the input when the input values are the same. To create a NOT gate, all we need to do is send the same voltage to *both* inputs of a NAND gate!



Making an AND gate

Using the NOT and AND gates we constructed above, try to work out how you would construct an OR gate implementing the following truth table:

input A	input B	output
0	0	0
0	1	1
1	0	1
1	1	1

Simple components

In programming, we reduce complexity by composing larger components out of smaller ones. Ideally, a component will expose a minimal interface to the rest of the application. Anything utilising the component only needs to know the interface and can be blissfully ignorant of the component's nitty-gritty implementation details. We saw this idea in the distinction between data types and data structures. Digital circuits are no different.

We can package some NAND gates into a particular arrangement and call it an AND gate. Once we've verified that this arrangement of NAND gates demonstrates the behaviour specified by the AND truth table, we can treat it simply as an AND gate and ignore its implementation details. Multiple gates can be combined into simple components and then these components can be combined into more complex components. Each stage abstracts away the details of the previous stage, thus keeping the complexity manageable.

One of the simplest components is the **half-adder**, which adds two binary digits. It has two outputs: the sum and the carry. Just like you need to carry a digit if a sum is greater than 9, you need to carry a bit if the output is greater than 1. The half-adder truth table looks like this:

input A	input B	sum	carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

If you add a `1` and `0` you get `1`, as expected. If you add `1` and `1`, you get a `0` and a `1` carried over. This is `0b10` or binary two. Can you figure out how to construct a half-adder from NAND gates?

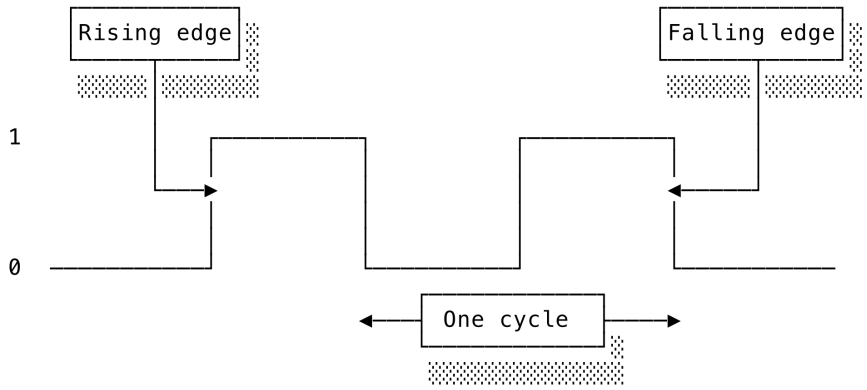
If we chain a series of half-adders together, the output of one feeding into an input of the next and so on, we can create adders capable of adding binary numbers of arbitrary length. This is a huge step up in our capabilities. Two other fundamental components are **multiplexers**, which take multiple inputs and use a control signal to determine which one to output, and **demultiplexers**, which take one input and use a control signal to determine which of multiple outputs to send it to. These components can in turn be composed together to make still more complex components.

It's very tempting to anthropomorphise computers and say things like "it's thinking" or "it wants me to do something". What computers can do is so complex that it seems like there *must* be some kind of intelligence in there. Yet when you dig down into the circuits, you see that computers are nothing more than electrical signals shunting along through lots of very simple components arranged in particular patterns. There is no real decision-making ability in the gates and components. Their design compels them to respond to their inputs in a particular way. There is intelligence in the design of circuits, but it is in the minds of the humans who arranged the components in that particular pattern to achieve those particular outcomes.

Circuits with memory

Combinational logic had no concept of state. The output of the circuit is entirely determined by the current inputs. This poses a problem: how can we store values to use later? It seems that the circuitry we've seen so far has no way of building components that can remember things. No combinational logical component will do the job. We need additional concepts.

To be able to use a value "later" implies some kind of time that is different to now. Combinational logic is all about the now. It has no idea what happened before the inputs took their current form. We need a way of dividing time into before and after: a clock. In digital circuits a **clock** is a component that outputs a signal that alternates rapidly between two values at a regular frequency. Either the rising edge or falling edge can be defined as the start of a new **clock cycle**.



A clock cycle

Armed with our ability to measure time, we can now examine some new components. The first is known as a **flip-flop**. A flip-flop has an input and a control signal. Whenever it receives a control signal, it "latches" on to the input value at that moment and continues outputting that value until another signal tells it to latch on to a new value. It thus "remembers" the input value at the time of the control signal. Each flip-flop can hold one bit. This is our first example of **memory** within a circuit.

Computer memory is not much more complex than huge banks of flip-flops arranged into bytes, kilobytes, megabytes and so on. Flip-flops require electricity to keep holding on to the value. This is why your computer memory is wiped clean when you power off. Think of how you might temporarily remember a phone number by repeating it to yourself. As soon as you stop repeating it, you forget it.

The clock speed determines the overall performance of the processor. Flip-flops latch on to their input at the beginning of a cycle. This means that we need to have the value we want to store ready by the end of the preceding cycle. It doesn't matter whether we were ready right at the start of the cycle or only just managed it before the cycle ended. The value won't be used until the start of the next cycle. A particularly fast bit of circuitry that could perform five calculations within a single clock cycle would be useless because the rest of the system wouldn't be ready to consume any values until the next clock cycle. The clock speed sets the speed of every component in the processor.

This is why clock speed is such an important number and used as a yardstick for a computer's performance. It's measured in hertz (Hz), or cycles per second. Modern processors operate at a few GHz, or billion cycles per second. Every component needs to receive the clock signal at exactly the same time to maintain synchronisation. This means that the wiring connecting each component to the clock has to be the same length, creating a sort of fractal design that you can see when you look at a close-up photograph of a processor.

Putting a clock and flip-flops together, we have what is known as **sequential logic**. This can do everything combinational logic can do. In addition, it has the ability to store values and utilise them in later computation.

If you find the micro-architecture of digital circuits even slightly interesting, I highly recommend that you work through the first five chapters of *nand2tetris* (see further reading). If you really want

to demystify the workings of your computer, there is nothing better than constructing one yourself!

The processor

We are now ready to learn how a computer is put together. Computers typically have a few key components. The processor is the component that decodes and executes binary-encoded instructions. Instructions and data needed for a program's execution are held in memory. Since memory is volatile and cannot maintain its contents without power, a separate component, known as the hard drive, is needed for persistent storage. A computer will also include various input/output devices (e.g. keyboard, monitor, network card) so that the processor can communicate with the outside world.

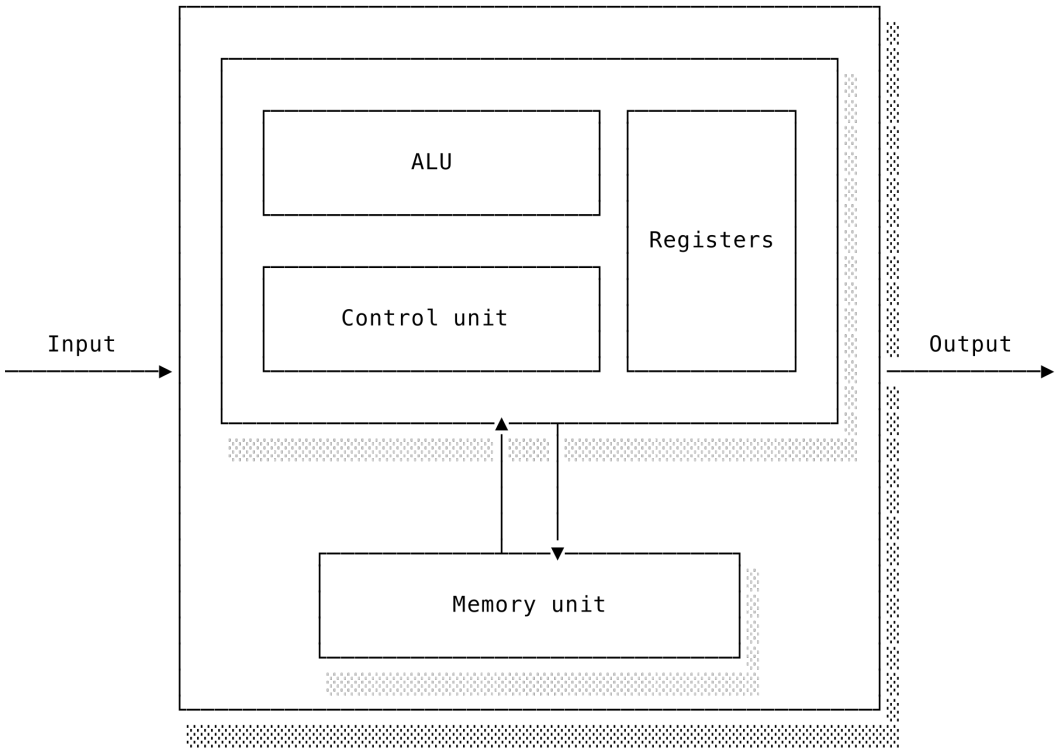
The processor, or central processing unit (CPU), is the brains of the computer. It operates as a pipeline. Instructions go in one end, the processor executes them and the results of their computation come out at the other end. The processor operates an **instruction cycle** of fetching an instruction from memory, decoding it, executing it and finally writing back the output. The instruction's execution will generate some output that changes the computer's state e.g. writing a new value to memory. In doing so, it modifies the execution context of the subsequent instructions. Complex functionality is implemented by a succession of simple steps, each one building on the work of the last.

Our little `ADD` begins its exciting journey by being **fetched** from memory. When the user initialises a program, the operating system loads the machine code into memory and tells the processor where to begin. It keeps track of where the next instruction is by holding the instruction's address in a special register called the **program counter** (PC). Normally, the processor moves sequentially through the instructions by incrementing the PC each time it executes an instruction. As we've seen, some instructions can jump to a new location by writing a new location to the PC. Once loaded from the hard disk, our instruction will sit patiently in memory waiting for the processor to work its way to it. Eventually, the value in the PC matches the location of our instruction and it is fetched from memory and transferred to the processor.

The processor now has our instruction in its maw. Before it can do anything, it must **decode** the instruction to understand what computation it specifies. Based on the design of the instruction set architecture, the processor will have circuitry that detects the instruction format and identifies the various fields within the instruction. The processor determines which registers it needs to use, the function specifier and so on. Within the processor is a component called the **control unit** that identifies and coordinates the bits of the processor that are needed to execute the instruction. When it decodes our `ADD` instruction, the processor will recognise that it needs to add two values together and it will identify where the values are held.

Finally, the processor is ready to **execute** the instruction. The control unit orchestrates moving the values from the inputs to the circuitry that performs the calculations to generate the output. To execute our `ADD` instruction, the control unit takes the values from the two registers and passes them to a component called the **arithmetic logic unit** (ALU). The control unit instructs the ALU to perform the requested addition. The ALU computes the result by passing the values to some adders and outputs the result. The control unit takes the value from the ALU's output and **writes it back** into the destination register specified in the instruction.

And with that our little instruction has fulfilled its destiny! The processor fetches the next instruction and the whole circle of instruction life continues.



The ALU

Let's examine the components that make up the processor.

Registers

Registers are very small, very fast bits of memory located right in the processor itself. They hold the values the processor needs for the current instruction's execution. Before the processor can operate on a value it must be loaded from memory into a register. After an instruction executes, the result is written back to a register. Processors typically have at most a few dozen registers and so making efficient use of them is very important.

Imagine that you're studying in a library. The only books you can immediately refer to are the ones on your desk. You can return some books and withdraw other ones as your needs change, but you can only ever work with the books you have in front of you on your desk.

The control unit

The control unit is the component that coordinates work by sending appropriate signals to the rest of the processor. The control unit ensures that every other component receives the correct value at the correct time. Each component sits waiting for control signals that tell it what to do. The control unit

is physically connected to the components via control lines, along which the control signals pass. I like to think of the control unit as an octopus with many tentacles worming their way in around all the components, triggering them all at just the right time to correctly perform the computation. The values themselves are passed around on **buses**, which are parallel rows of connections to allow multiple bits to transfer simultaneously.

The arithmetic logic unit (ALU)

The processor components that actually perform the execution are called **execution units**. The ALU is the execution unit that contains specialised circuitry for integer arithmetic (e.g. addition, multiplication) and logical (e.g. bitwise AND) operations. This is the real heart of the processor: it's here that the actual computation happens. The ALU receives its input values, or **operands**, from the control unit along with an opcode extracted from the instruction. The ALU uses the opcode to activate the correct circuitry sections to perform the requested operation and outputs the result.

The ALU does not know or care where the operands come from or where the output should go. It is responsible purely for performing the calculations specified by its inputs. From the point of view of the control unit, the ALU is a black box. The control unit routes the inputs to the ALU, tells it what to do and then takes the ALU's output for further processing. What we have here is a example of encapsulation within the processor.

In addition to outputting a value, the ALU can change its state by setting **flags**. These are normally stored in a special register, known as the status or flag register, in which each bit represents the presence or absence of a certain condition. It's like a little set of switches squished into a byte. The ALU raises a flag by setting the flag's corresponding bit to 1. This indicates that the relevant condition is present. To know what bit indicates what, you must refer to the processor's programming manual. The exact set of flags varies from processor to processor but common conditions include whether the ALU output is negative, whether it's zero and whether an operation resulted in a carry.

Subsequent instructions can command the processor to inspect the flags and execute the instruction only if certain flags are set. For example, a conditional branch might instruct the processor to only branch to a new location if the previous instruction raised the zero flag. Setting flags is a neat way to decouple the processor's internal components. The ALU doesn't need to know or care about which other components are interested in whether its result is a zero. It just sets the zero flag and moves on to its next task. Other components that do care can check the flag without having to directly monitor previous instructions.

Flags are also used for error handling. Attempting to divide by zero is a mathematical impossibility that a processor cannot compute. When asked to divide by zero, it must alert the rest of the system by raising a divide-by-zero flag. Error handling code will detect that the divide-by-zero flag has been set, interrupt normal execution and report it to the programmer. We'll look at this "interrupt" process in much more detail in the operating systems chapter.

Floating point calculations are handled by a specialised **floating point unit**.

Memory

If an instruction's output is immediately consumed by a subsequent instruction, it might stay in its register and never leave the processor. If we want to keep the output for later use, however, we need to write it out to the system's main memory. A computer's memory is the physical analogue of the Turing machine's infinite tape. It is built from huge numbers of flip-flops, organised into bytes, that can maintain a value for as long as power is supplied. Memory is the computer's working space, holding instructions and data that the processor needs to have accessible. All but the most simple processors interact with memory through a **memory management unit** (MMU). This is a component designed to work efficiently with the operating system to perform the calculations and look-ups that working with memory entails. We'll explore how it works in the OS chapter.

Memory addressing

The size of the processor's basic operating unit is known as the **word size**. This determines how many bits the processor can operate on at the same time and so specifies the size of the registers and buses. For example, a processor with a 32-bit word size will load 32-bit values from memory, store them in 32-bit registers and perform computation on 32-bit values. The exact size of a word will vary from architecture to architecture but will be a multiple of a byte. A few years ago there was a shift from 32-bit to 64-bit computers that completely bemused everyone non-technical. What that meant was that computers shifted from having a word size of four bytes (32 bits) to a word size of eight bytes (64 bits). The reason for doing so was to overcome limitations to how much memory a system could use.

The word size has important implications for memory. Each memory location must have a unique address, otherwise the computer would have no way of referring to the location. The number of unique addresses the computer can use is determined by how many bits the **address size** is. Typically, the address size corresponds to the word size. So on a 32-bit machine with 32-bit addresses there are 2^{32} or 4,294,967,296 addressable memory locations. Over four billion seems large but it's not in the context of computers. On x86 architectures each address refers to an individual byte value (known as **byte-addressing**). This means that a 32-bit computer can only address around four gigabytes of memory. There is simply no space in the address to refer to a location above that. It's like trying to express the number 10,234 using only three digits – there's no easy solution.

Various workarounds do exist, such as dividing a larger address space into segments and indexing into them using complicated addressing instructions, but this comes at a performance and complexity cost. Being able to address larger memory spaces was the main driver behind the switch from 32-bit to 64-bit machines. A 64-bit machine can address 2^{64} locations, which corresponds to around 17 billion gigabytes. That should hopefully be enough for a while.

The memory hierarchy

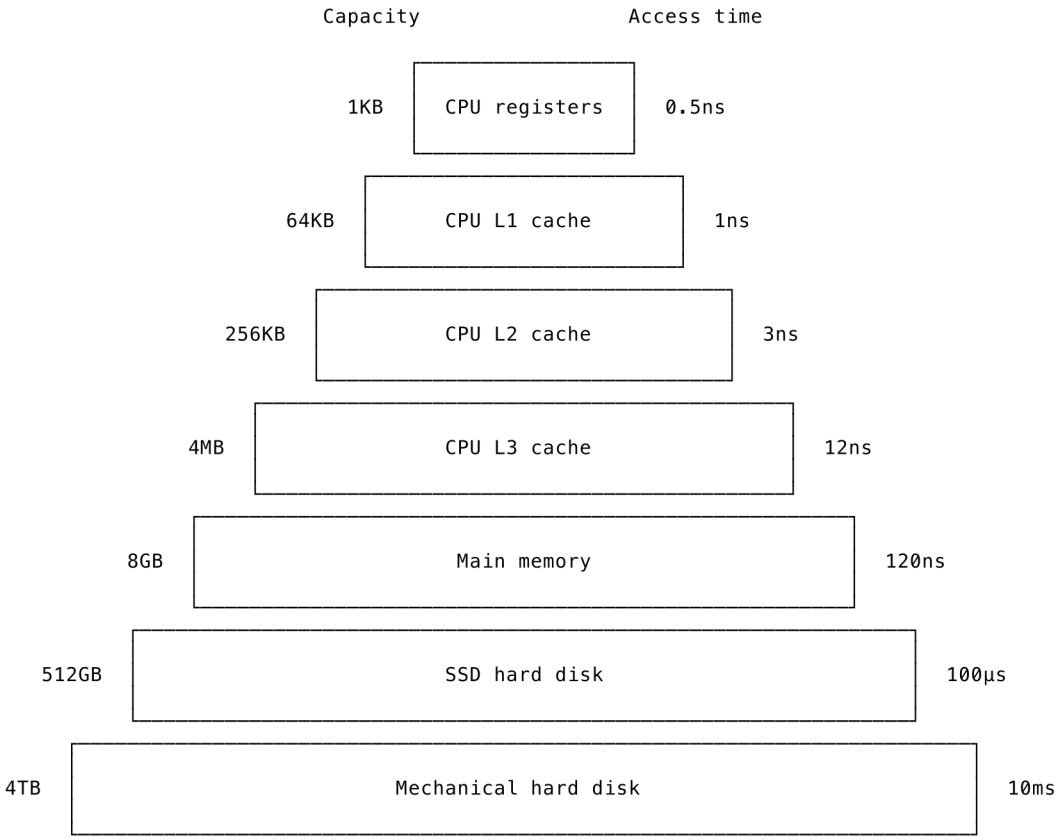
An important characteristic of modern computers is that the memory operates at a much slower rate than the CPU. This is a fairly recent phenomenon. In the olden days, memory and processor

operated at the same speed. Over the decades, however, increases in processor speeds vastly outpaced increases in memory speeds. Memory accesses are now a major performance bottleneck because the processor is forced to spend a lot of time waiting for the operation to complete. On a 3GHz processor, each execution step takes around $0.3ns$ (nanoseconds). Accessing a value in main memory might take around $120ns$ – 400 times slower!

The solution is **caching**. The principle is very simple. A program is more likely to reuse a recently accessed value than one it accessed a long time ago. This is called **temporal locality**. It makes intuitive sense if you think of how often computers iterate over the same set of values and operations. A program is also more likely to use a value if it's located close to a previously used value, such as the adjacent elements in an array. This is called **spatial locality**. Processors can exploit temporal locality by holding recently used values in a small amount of very fast memory known as the **cache**. They can exploit spatial locality by caching values from adjacent addresses whenever they retrieve something from memory. If the program is accessing an array element, it is very likely to soon need the subsequent elements so it makes sense to pull in several each time.

When caching works well, the majority of values accessed are already held in the cache, greatly reducing the amount of time taken to retrieve them. The problem is that space in the cache is extremely limited. To be fast the cache must be small, which means that it can only hold a few values. Making the cache bigger makes it slower. To return to our library analogy, rather than returning books to the stack you could keep them easily accessible on a nearby set of shelves. They need to be small to be effective. The more books the shelves hold, the longer it will take you to find a particular book. The processor has to be very selective about what it keeps in the cache.

Modern processors use multiple caches to create a **memory hierarchy**. As we descend the hierarchy, each level offers bigger but slower storage. At the top of the hierarchy are the registers, which are the fastest but also smallest memory units in the system. Below them is the larger but slower level one (L1) cache. A modern L1 cache holds a few kilobytes. Below that is the L2 cache, holding a few hundred kilobytes, and below that the L3 cache, holding a few megabytes. Below the caches comes main memory itself. It's markedly slower but can hold gigabytes of data. At the bottom, offering terabytes of storage but much slower speeds, is the hard disk.



The memory hierarchy with approximate capacities and access times

There are several algorithms for deciding what to keep in a particular cache and what to evict down to the next level. One simple approach is to let the cache fill up and then push out the least recently used value. For example, when a value in a register is no longer needed, it is moved from the register to the L1 cache. If it's not used again, it'll eventually get pushed out to the L2 cache and so on down the hierarchy. If a value falls to a lower level cache and is then reused, it moves back up to the top of the hierarchy. In this way, the most frequently used values stay at the top of the cache hierarchy and rarely used values move down to the lower levels.

When studying the performance characteristics of the memory hierarchy, I find it helpful to use a human-scale analogy. It can be hard to understand how nanoseconds and microseconds relate to each other. Instead, imagine that each processor step lasts for a single heartbeat. An L1 cache access then takes a few seconds. Enough time for a sip of coffee. An L2 access allows enough time, roughly ten seconds, for a yawn. An L3 access takes about forty seconds. That's some twenty times slower than a L1 cache access but much faster than main memory. After initiating a main memory access, the processor would have to wait around five minutes for the response. You can see how avoiding even a few main memory accesses makes a huge difference. Even worse, accessing a SSD (solid-state drive) hard disk would take a couple of days and accessing a rotational hard disk would take months. Just for completeness, a $100ms$ network request would take several *years* to complete! Our

poor processor could get a university degree in the time it would spend waiting.

Caching is completely transparent to the programmer. Processors generally don't offer any instructions to query cache contents or to explicitly insert something into the cache. All you can do is measure how long it takes to retrieve a value and infer from that where it was stored in the memory hierarchy. When caching works well, it delivers average access speeds that are much faster than main memory accesses. The performance gains come at the cost of greater implementation complexity. The processor needs to have substantial circuitry to keep track of modifications to cached values and prevent stale values from being accessed.

Caching is a complex topic that crops up whenever there are time-expensive operations. We'll come across it again when looking at networking. Many textbooks detail how to write "cache aware" code, meaning code that optimises its memory operations to keep the most important values in the cache. While processor caching is interesting, it's important to keep things in perspective. As a web developer, the time cost of network operations is going to dwarf any performance gains you could possibly get from more cache aware code.

Performance optimisations

The picture I have presented in this chapter is deliberately simple. The processor described above executes one instruction at a time. This is wasteful. When the instruction is being fetched there is nothing for the decoder or execution units to do. The rest of the processor is stuck waiting. Common optimisations therefore introduce **parallelism** into the processor, allowing the processor to work on multiple things at once and reduce the amount of time components spend idle. These optimisations all add greatly to the complexity of the processor design, to the extent that it takes some effort to understand fully how a modern processor works.

Instruction-level parallelism tries to increase instruction throughput. Rather than executing one instruction to completion before starting the next, it is better to process the incoming instructions in a **pipeline**. For example, as the processor's execution circuitry executes one instruction, the decoding circuitry can start decoding the next instruction. This increases the throughput without actually increasing how many instructions are executed at once. Modern processors feature deep pipelines of a dozen steps or more.

A pipelined processor still only handles one instruction at each step of the pipeline. An obvious optimisation is to widen the pipeline and allow multiple instructions at each step. **Superscalar** refers to pipelined processors that include multiple execution units (ALUs etc.) and so can process multiple instructions in parallel. The challenges now are keeping the pipeline as full as possible and working out which instructions can safely execute in parallel without changing the meaning of the program.

Conditional instructions create branch points where the processor has multiple possible execution paths. Processors use **branch prediction** circuitry to judge which path is more likely to be taken. The instructions from this path can be fed into the pipeline to keep it full without having to wait until the conditional instruction is executed. Branch prediction can sometimes be surprisingly straightforward. If your code contains an iteration such as `for (i = 0; i < 1000000; i++) { ...`

}, the processor will quickly assume that the loop will continue to iterate. When the loop eventually finishes, the processor will have incorrectly predicted another iteration and will incur the cost of a branch mis-prediction. It will have to flush the pipeline of all the instructions and refill it with the instructions following the loop.

A processor may even perform **out-of-order execution** and reorder instructions if it determines that it can make more efficient use of the pipeline's resources without changing the program's meaning. This requires circuitry to detect any dependencies between the instructions and ensure that the reordering does not change the output.

Branch prediction is an example of **speculative execution**, in which a processor makes use of spare pipeline capacity to perform a computation that may or may not be needed in the future. If the computation turns out to be needed, the processor has saved time. If the computation isn't needed, because the speculatively executed branch isn't taken after all, the processor needs to be very careful to completely discard the work and revert any changes to the processor's state. The recent Meltdown and Spectre[24] security vulnerabilities were caused by processors not correctly removing cached values generated by speculatively executed instructions. That made it possible to infer the results of the instructions by determining which values were in the cache.

Instruction-level parallelism is good at executing a single task more quickly but even in superscalar processors the instructions need to share parts of the processor's state. This means that it's not possible to run two completely separate tasks at once. The logical progression is to achieve **task-level parallelism**, either by combining hardware support for multiple threads in a single processor (known as **hardware threads**), or by combining multiple, independent processors (now referred to as **cores**) in the same chip. So called multi-core processors are capable of performing truly independent tasks in parallel. Nowadays, consumer grade processors usually contain at least two or four cores and server chips might have four times as many. The shift to multi-core processors has huge implications for programming, which are the subject of the concurrency chapter.

The modern processor is a technical achievement of almost unbelievable complexity and ingenuity. It's not a requirement that you understand all of this in depth. but I do encourage you to explore the further reading material to get a better sense of just *how much stuff* is going on in our phones and laptops. It's really magical.

Conclusion

In this chapter we examined how computers operate as machines. We saw that all information is encoded in binary as a sequence of bits. We looked at the MIPS instruction set to better understand how this encoding might work. We saw how digital circuits perform logical computations by passing voltages through arrangements of transistors known as logic gates. These gates can themselves be arranged to form the simple components out of which more sophisticated elements are constructed. The most sophisticated component in a computer is the processor, which performs computation by decoding and executing machine code instructions. We followed an addition instruction through

[24] https://meltdownattack.com/

the processor and saw how the control unit, arithmetic logic unit and registers work together to compute a value. A component known as the flip-flop enables sequential logic and the construction of memory components that can remember previous values. Unfortunately, memory speeds have not kept pace with increasing processor speeds. One solution is to create a memory hierarchy containing multiple caches. Modern processors also include many complex optimisations designed to keep the processor's execution units as busy as possible.

Further reading

Some of my favourite computing books are written about computer architecture!

Start with *Code* by Petzold (who also wrote *The Annotated Turing*, recommended in chapter one). It builds a foundational model of computing by starting from the absolute basics and gradually piling on the complexity. I still remember the revelation I had reading *Code* during a long coach journey. I suddenly realised that a computer is nothing more than lots and lots and *lots* of really dumb components working together. If you've ever thought to yourself "but how did the computer actually *know* that?", then you need to read this book. *Code* explains from first principles how digital computers work in a gentle, didactic manner.

Next up is nand2tetris[25]. It's available as a free online course or as a textbook called *The Elements of Computer Systems*. In this book we take everything we learned in *Code* and write our own implementation. We start off designing logic gates and gradually work our way up to writing a complete computer system including the processor, assembler, compiler and operating system. Though challenging, it's hard to overstate the satisfaction you get from really understanding how code is executed right down to the level of individual logic gates. The book itself is rather slim, which is why I recommend beginning with *Code* to make sure you have a good grasp of the concepts. The real value is in the projects so be sure to take the time to do them. It's a popular course so you'll have no trouble finding help online if you get stuck.

A limitation of nand2tetris is that we only implement simplified, toy versions of the components. This is intentional: implementing a modern processor would be an impossibly complex task. The danger is that we are left with an overly simplistic mental model that doesn't reflect how things actually work. So, I advise you to continue your study at least a little further.

Computer Systems: A Programmer's Perspective, by Bryant and O'Hallaron, is a wonderful textbook. As the name suggests, it approaches computer systems from the perspective of a programmer seeking to understand their system better. Exactly what we want! It's my favourite textbook on computer architecture. The canonical textbook in this area is *Computer Organisation and Design*, by Patterson and Hennessy. It's a bit more advanced than *Computer Systems* and focuses much more on the hardware implementation details. I particularly appreciated its in depth description of how a modern, superscalar, out-of-order processor works.

What every programmer should know about memory[26] by Ulrich Depper is a wide-ranging treatment of memory in computing systems. I'd argue that it goes into significantly more depth

[25] https://www.nand2tetris.org
[26] https://people.freebsd.org/~lstewart/articles/cpumemory.pdf

than every programmer should actually know but at least chapters three and four, on CPU caching and virtual memory respectively, are important reading.

A fun way to learn about a particular architecture is to write some assembly code. You'll learn about the architecture's ISA and how to express functionality in low level code. You could write assembly for your PC, possibly with Randall Hyde's *The Art of Assembly Language*, but this means starting off with the complex x86 instruction set. A simpler starting point could be an Arduino board, which uses the AVR instruction set. I found *AVR Programming* by Elliot Williams to be a very well-rounded introduction with some amusing project ideas. If you don't want to muck around with buying hardware, the game TIS-100[27] is a fun way to get a taste of assembly programming.

Another great way to really learn an architecture is to write an emulator for it. An emulator is a program that imitates the physical behaviour of the emulated hardware. CHIP-8 is a simple architecture designed for easy emulation. From there, if you're already a confident programmer, you could progress to a Nintendo NES or GameBoy.

[27] http://www.zachtronics.com/tis-100/

Operating systems

Introduction

In the previous chapter, we learned how a real computer is put together. You may be forgiven for thinking that it still seems very far from how you actually use a computer. That's because there's a whole layer sitting between the hardware and you: the **operating system** (OS).

The OS provides solutions to some tricky problems. How does the processor know where to begin executing code or where to find the next program? How do you change between running programs? How do I stop an evil program locking me out of the entire system? To handle these issues we need hardware and software to work very closely together. The software side of the solution is the OS. It's the first program that runs on the computer and it acts as an interface between the user and the hardware. It abstracts away many of the details and peculiarities of different hardware components and provides a consistent platform on which other programs can run. It makes the computer easier to use, more secure and more performant.

Not giving user programs direct access to the hardware is a very good thing. Imagine writing a program where you had to handle the cursor yourself. You'd need to check for and identify any attached mice or trackpads, listen for and correctly interpret the hardware signals generated, recognise different types of clicks, gestures, taps and so on. It would be a nightmare! It's much better for everyone concerned if the OS can take care of all this and simply feed the user program a stream of notifications about where the cursor is and what it is doing.

Modern OSes include Microsoft Windows, Apple's macOS and GNU/Linux. They are all immensely complex conglomerations of software components that can handle a huge range of tasks. In this chapter, we'll try to pare things down to the core commonalities. We'll see how a computer starts up. Then we'll briefly review some necessary additions to our hardware model. We'll examine the implications they have on the OS architecture. Then we'll look at the core abstractions an OS provides to manage the key system components: the processor, memory and storage. In each case, the OS provides a virtual representation of the underlying resource.

Common operating systems

There have been many operating systems over the years. Let's look at a few of the most popular and how they differ from each other.

Windows is a family of closed source operating systems developed by Microsoft. The very first Windows was not much more than a visual interface on top of the older MS-DOS (Disk Operating System). DOS is interesting as a salutary tale because it was a very simple OS that lacked a lot

of what we'd now consider to be standard features. It gave user programs more or less complete control over the hardware and relied on them to behave themselves. This worked as well as you might expect. Later versions of Windows moved away from DOS entirely in a bid to improve its terrible reputation for reliability and security. Windows has traditionally focused on providing an easy to use interface for non-technical users, backwards compatibility and supporting a wide range of hardware.

Unix was an OS developed in the 1970s at Bell Labs that spawned a whole family of derivatives and reimplementations. "Unix" nowadays is more like a description of how to design an OS, formalised in a specification known as POSIX. Strictly speaking *Linux* refers to an open source reimplementation of the original Unix kernel, dating from the early 1990s. Many of the standard Linux system utilities were provided by a separate project, GNU (a recursive acronym of GNU's Not Unix), and so you'll sometimes see the complete OS referred to as GNU/Linux. There are many **distributions** (or distros) consisting of the Linux kernel packaged with a variety of extensions and user programs to form a complete operating system. Ubuntu, CentOS and Arch are examples of popular Linux distros, each targeting a different use case. Of course, the world has moved on a lot since the 1990s and Linux has continued to develop and add new features that never existed in Unix. Linux is popular with programmers and technically minded users because it offers complete control over the system to those who are prepared to learn how to use it. Being open source means that the user can read and, if they feel up to it, modify the operating system's code. It's basically the opposite of Windows. Linux is extremely popular on servers due to its reliability, performance and tweakability.

macOS (previously *OS X*) is the closed source, proprietary OS that runs on Apple computers. It is built around an open source, Unix-like core called Darwin, which is itself based on a branch of the Unix family known as the Berkeley Software Distribution (BSD). The main distinction between BSD and Linux is that the BSD licence allows it to be used in closed source projects, unlike Linux. On top of this open source foundation, Apple has built a large amount of proprietary, closed source code that makes macOS distinct from Linux. Because they share the same Unix-like fundamentals, Linux and macOS have many similarities. Despite that, programs built to run on Linux will not run on macOS (and vice versa) because the code on top of the common base differs too much. The combination of Unix-like foundation and good user interface makes Apple products popular with developers. Some will tell you that *real* developers use Linux. Ignore them.

The boot process

To understand why we want an operating system, we'll start at the very beginning. Let's imagine that we've built a computer like the one described in the architecture chapter. It's basic but sufficient to run programs and do useful work. How do we actually get it to start running a program? Let's think about what we've learned already. We know that we need to load the program instructions into memory and set the processor's program counter to the address of the first instruction. The processor will then start fetching and executing our program.

How do we actually get the instructions into memory and how do we tell the processor where to begin? Both of these puzzles can be solved easily if we're willing to make changes to the hardware.

We can write our program to a piece of **read-only memory** (ROM) and wire it up so that it lives at a fixed memory address. ROM is memory that can only be written once. The benefit is that it doesn't need electricity to maintain the data. Our program will persist in the ROM even when the machine is powered off. We can then hardcode the ROM start address into the processor's program counter. Now every time we power on the processor, it will start running our application!

Many simple microcontrollers work like this. Though workable, we have a few problems. For one thing, how do we change the running program? We could load different programs on to different ROMs and just physically switch them out. This is what old games consoles used to do but clearly this won't work for general computing. A better idea would be to put a special program on the ROM that would let the user select what they wanted to run. It would then load the chosen program into memory and tell the processor where to start executing. What we've just invented is a **bootloader**.

When a computer **boots**, or starts up, it is hardwired to first execute code stored in a small piece of ROM. Previously, this would have been the **BIOS** (basic input/output system) but **EFI** (extensible firmware interface) is more common nowadays. The distinction isn't terribly important because both perform the same role: work out what hardware the system has and initialise it, find the OS's code, load it into memory and start executing it.

Every operating system includes a bootloader which is stored on the computer's disk at the location specified by the BIOS/EFI standard. The bootloader's sole job is to get the OS up and running. Due to various terribly tedious technical constraints imposed on bootloaders over the years, the modern process actually involves the first bootloader calling a second, more powerful bootloader that then loads and starts the OS. Happily, there's no need to know all the arcane details as long as you get the general concept.

The boot process begins by the computer powering up. The BIOS/EFI runs and then hands control over to the bootloader, which loads the OS. Finally, the boot process is complete when the OS has finished initialising and is ready to receive user input.

Interrupts: hardware support for software

Under the processing model we developed in the previous chapter, there's no obvious way to force a program to relinquish control of the processor. Obviously, switching programs is something that it would be convenient to be able to do. There are important security implications too. If I can't stop a running program, I have no way to stop a program that's trapped in an infinite loop from freezing the whole system. Similarly, I can't intercept attempts by an evil program to reformat the hard disk or steal my passwords.

The problem is that our processor gives the running program complete control. The program can use that control to prevent any attempts to transfer control. Frankly, the program can do whatever it likes, up to and including destroying the system. One solution could be to only ever allow the OS to run directly on the processor. The OS would load other programs, analyse their instructions, pass them through to the processor if they looked safe and flag them if they seemed dangerous. This

approach could work but would entail a high performance cost. Every user program instruction would need many OS instructions to analyse and verify its safety.

A simple hardware modification, the **interrupt**, means that user code can run directly on the processor without the OS completely relinquishing control. An interrupt is a hardware signal that causes the processor to stop what it's currently doing and switch to executing a chunk of code known as an **interrupt handler**. The processor circuitry is amended to trigger an interrupt on a fixed timer or whenever a program attempts to perform a potentially risky operation. The OS sets up handlers for each interrupt. The handlers will examine the cause of the interrupt and take corrective action if need be.

There are two main types of interrupts. **Hardware interrupts** are triggered by – guess what! – hardware devices such as network interfaces and keyboards. They essentially have a direct line into the processor through which they can shout "hey! The user has pushed down a key. You need to handle this right now!". They are also known as **asynchronous interrupts** because they occur outside the normal program flow and can happen at any point in the execution cycle. This means that the processor has to be able to stop an instruction mid-execution and deal with the interrupt.

Software interrupts are used when the processor itself detects that something is preventing it from carrying on. For example, it might realise that it's been asked to divide by zero, which is a mathematical impossibility. It can't proceed so all it can do is report the error and hope something else can handle it. Such interrupts are **synchronous** because they occur within the normal flow of program execution. They are also called **exceptions** because they indicate that something exceptional has happened to the program.

The implication of adding interrupts is that we now have two categories of code. Code that runs whenever an interrupt is triggered is **privileged** because it has oversight and control over **unprivileged** code. We add a new hardware flag to monitor how privileged the currently executing code is. When the current code is privileged, we set the flag and the processor is said to be running in **supervisor mode**. The idea is that any code running in this mode has a supervisory role over code that does not run in this mode. If the flag is not set, the processor is in **user mode** and running unprivileged code. Before executing sensitive instructions, such as modifying files, the processor checks that the flag is set and triggers an interrupt if it is not set.

On receiving an interrupt, the processor stops its normal execution. It stores the state of the registers, flags and so on for safekeeping. The interrupt signal includes a number to identify which kind of interrupt has occurred. The processor uses this number to index into a list of **interrupt descriptors** maintained by the OS. Each descriptor tells the processor where the corresponding interrupt handler is located. The processor fetches the handler and then executes it. Once the handler has finished dealing with the situation, the processor restores the original program's saved state and carries on from where it left off.

Thanks to this hardware addition we can deal with unresponsive programs. By triggering an interrupt on a timer, we can regularly suspend a running program and give the OS a chance to run. The unresponsive program won't be on the processor forever. The OS will get at least some processor time which it can use to check the status of all running programs and terminate any

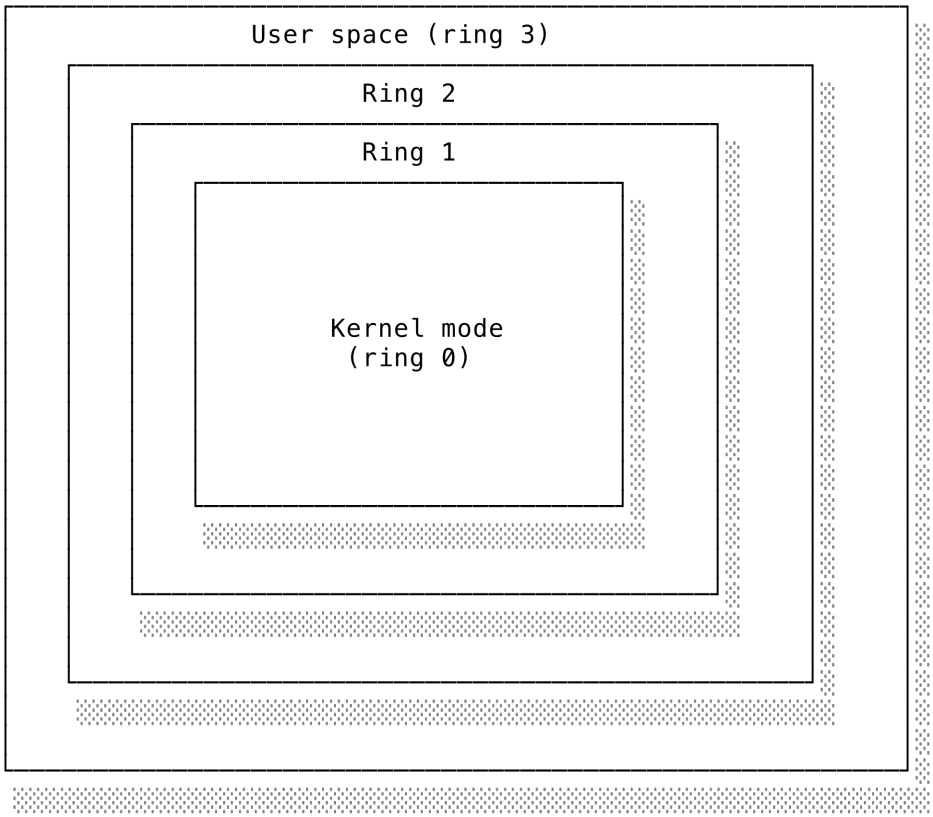
unresponsive ones. Because the timer is triggered by hardware, it can't be blocked by a malicious or incompetent user program.

The kernel

The OS has power that other programs don't have. Thanks to interrupts and its privileged mode, it will always regain control of the processor and can stop any other program. Your browser can't arbitrarily terminate your text editor but your OS can. With this power comes great responsibility. In particular, it's the responsibility of the OS to not have critical design flaws or bugs that can be exploited to allow unprivileged code to run at the OS's privilege level. This means that the huge range of services and features offered by the modern OS becomes a bit of a security risk. If the whole OS runs with super special privileges, then a bug in some obscure component, such as a font manager, could offer an attacker a way to gain privileged access to the whole system.

The solution to this problem (apart from reviewing OS code very, very carefully) is to divide the OS into two parts: a small **kernel** of trusted code that runs with full privileges and a bigger **userspace** that runs in unprivileged user mode. Supervisor mode is often known as **kernel mode** for this reason. The only code that goes into the kernel is code that absolutely needs to have higher privileges in order to do its work. The kernel is the core functionality around which everything else is built. Less important programs, such as our poor font manager, can be pushed out into userspace. A bug in a rogue userspace program is obviously still undesirable but doesn't entail the same security risk.

It's common to see these modes referred to as **rings**. In the ring model, kernel mode is ring 0, the innermost. Around that are rings of decreasing privilege levels. Userspace corresponds to ring 3. Rings 1 and 2 are less common and mainly used by specialised programs such as device drivers. Since only ring 0 and ring 3 are regularly used, I personally find it easier to just think of them as kernel mode and user mode.



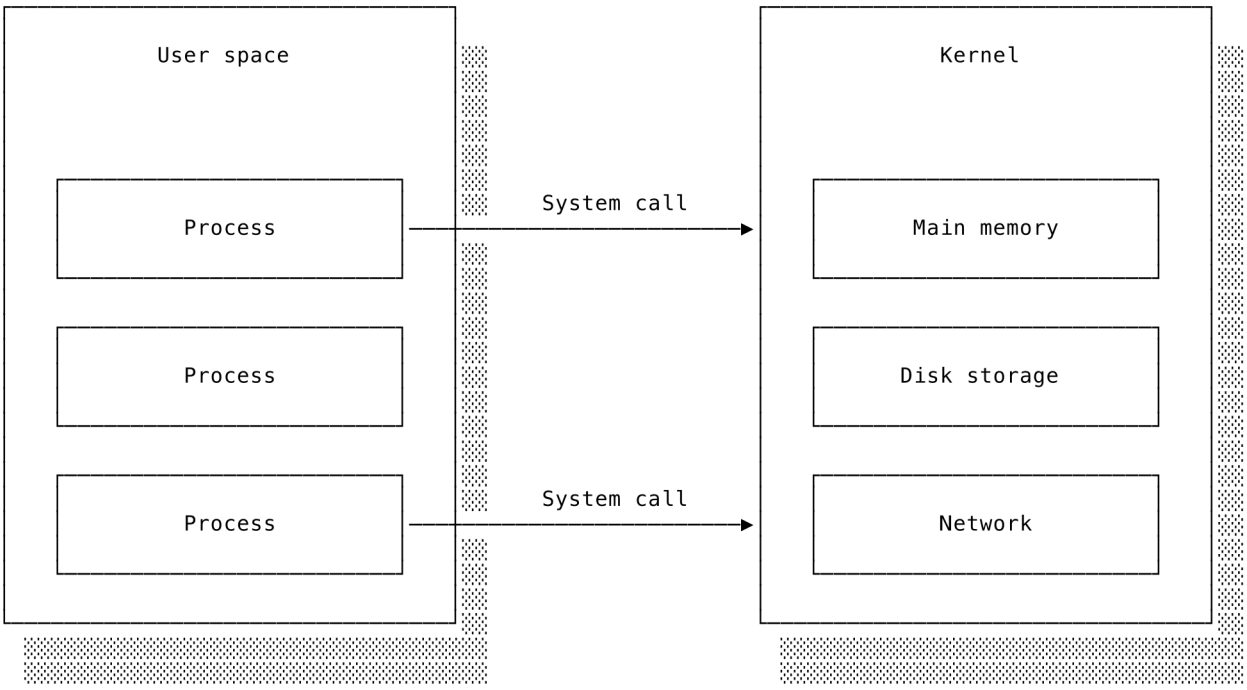
The ring system

The basic rule of thumb is that a program should run in userspace unless it absolutely has to run in the kernel.

Gates and system calls

What happens when a userspace program legitimately needs to do something that can only be done in kernel mode? There must be a secure way to move between rings. If you imagine the rings as giant walls arranged concentrically around a vulnerable core, we need ways through the walls. To stop unauthorised access attempts, we also need some kind of stern bouncer deciding who can go through. On x86 processors, the interrupt descriptors are known as **gates**. The gate's interrupt handler is the stern bouncer deciding whether or not to permit the access request.

As well as the interrupt handler's address, each interrupt descriptor specifies in which mode the handler should be executed. Specifying kernel mode provides a secure way to switch between user mode and kernel mode. It's secure because we can't just jump into kernel mode as we please – we can only go through the gate. The user program has no control over what the handler does or whether it chooses to permit the requested action. The handler in each gate decides whether or not to permit the operation requested by the user program. It's just like trying to get into a nightclub. The bouncer decides whether you get in and doesn't have to explain their decision to you. Trying to argue doesn't end well.



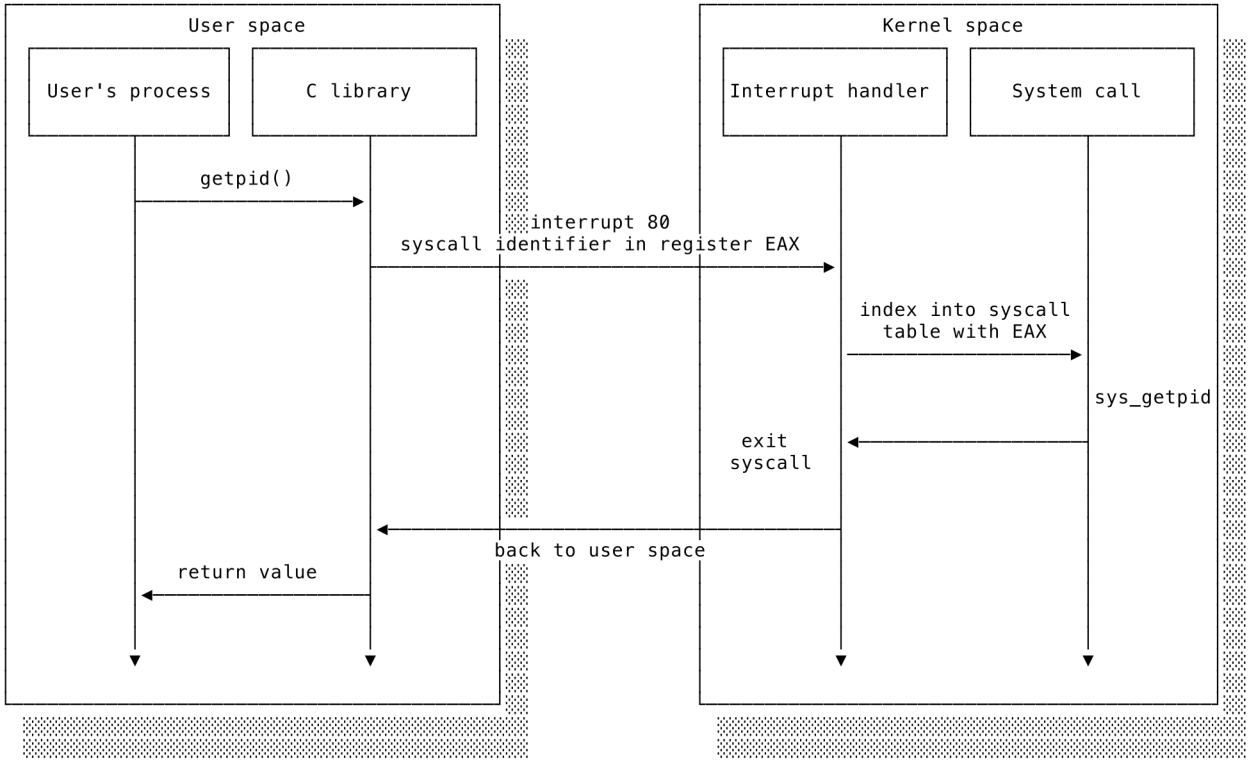
The kernel and user space

The OS utilises this hardware support to provide special **system calls** to allow user programs to request services from the kernel. For example, a user program cannot write data directly to the hard drive wherever and whenever it pleases. If this were permitted, a poorly written program could overwrite critical system files. Instead, the OS provides a set of system calls for interacting with the file system. The user program must use the appropriate system call to request that data written to the desired file. The kernel checks that the request is valid, performs the operation and then reports back on its success or failure to the user program. Only the kernel actually interacts with and modifies the file system and disk. The user program only knows whether its request succeeded and what errors, if any, occurred.

Due to the fact that system calls need to be able to switch into kernel mode, their implementation is very low level (often in assembly) and specific to the processor architecture. The OS therefore provides a C library of wrapper functions as an interface between the system calls and userspace code. The requesting program first calls the C wrapper function. It also runs in userspace. The wrapper function calls the actual system call, which uses a gate to transfer to kernel mode and perform the requested operation.

Each OS handles the details differently. Linux defines one interrupt, `0x80`, to be for system calls. Each system call is additionally assigned a unique syscall identifier. Invoking a system call entails raising an `0x80` interrupt and passing the syscall identifier and any arguments to the processor. The processor uses the interrupt identifier `0x80` to find the interrupt handler. The interrupt handler, running in kernel mode, examines the syscall identifier to identify the correct syscall handler. We have two layers, one hardware and one software: the interrupt identifier is used by the processor's hardware to identify the interrupt as a system call and then the syscall identifier is used by kernel

code to identify the particular system call. The processor doesn't care about the syscall identifier – that's a software matter for the kernel.



The steps of a syscall

On Linux, you can find lots of information about the available system calls in the manual pages (`man syscalls`). They will give you an overview of the available system calls, their identifiers and the arguments they accept. I recommend skimming through this to get a sense of the functionality offered by system calls.

The process I've described here is technically referred to as the legacy Linux approach for 32-bit architectures. Switching between CPU modes is comparatively expensive because the processor has to store the state of the current program and then restore it. The modern Linux kernel on 64-bit architectures uses a faster method that avoids the mode switch at the cost of greater complexity. I have avoided describing this method because, though conceptually similar, the additional complexity obscures the clean distinction between user mode and kernel mode.

What we have here is a neat separation of concerns. The system calls act as an interface defining the services available to userspace programs. The user program has no knowledge of or control over how each system call is implemented. This greatly reduces the complexity of interacting with the system. Any program written for Linux knows that it can request to read a file by using the correct system call. This will be the same across all machines running Linux. The program doesn't have to worry about interacting directly with the hard drive or whether it needs to use this or that gate. All of the complexity is hidden away behind the system call and dealt with by the OS. Security is improved because the user program cannot tamper with the system call implementation. Of course,

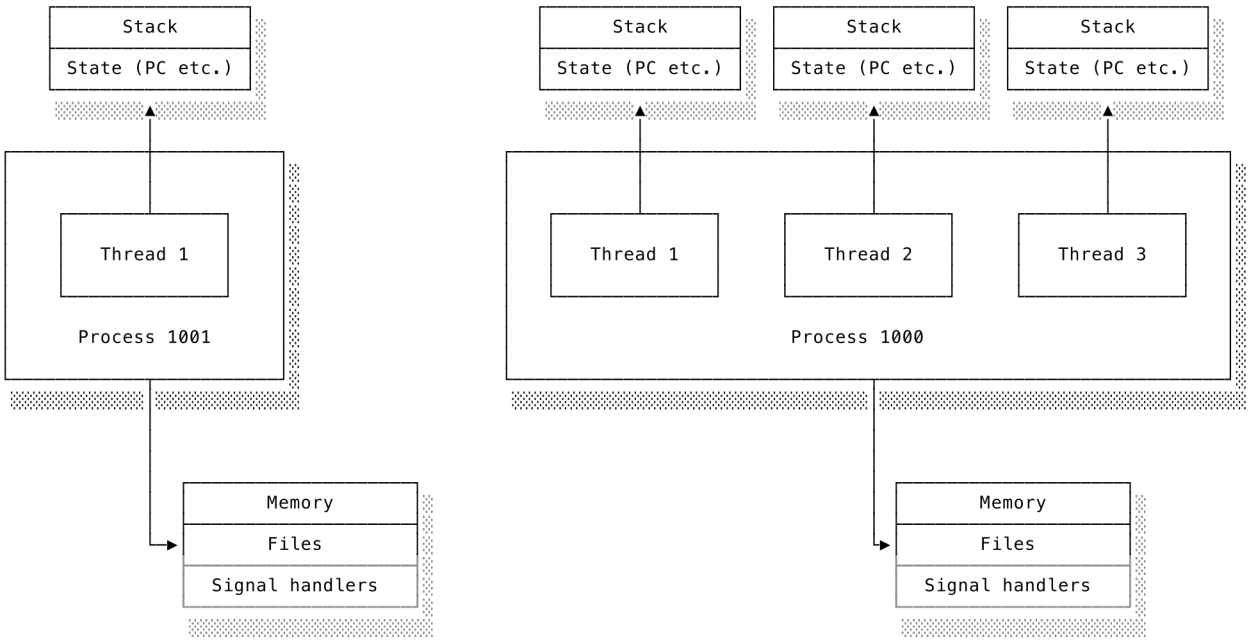
we're still reliant on the kernel code being written correctly in the first place but at least we only have to verify the correctness of the system calls and their wrappers. We have successfully limited how much damage dangerous user code can do.

Managing the processor

The operating system is a special program that starts up the system and orchestrates every other program, jumping in whenever the processor alerts it to naughty behaviour. How does it do this?

Normally we think of running a program as an activity. It's something that *happens*. A **process** is a clever abstraction that turns this activity into an object that can be moved around and manipulated. A process is an instance of an executing program. It includes the program's instructions, any memory used by the program, the processor's flags and registers (including the program counter), any files the program is using and a few other bits and bobs. A process is a complete description of a particular point in an instance of the program's execution. When we talk about "running a program" we should more precisely say "running a process". The OS runs programs by loading them into processes, so it is possible to have multiple instances of the same program running at the same time in different processes.

When a program runs, it follows a particular route through the code. A conditional sends it here, a jump sends it over there. Perhaps it'll take a different path the next time it runs, depending on how its execution context and user input change its behaviour. If we had two separate program counters, we could have two simultaneous but separate **threads of execution** through the instructions. Every process contains at least one thread and modern processors include hardware support for multiple threads within the same process. By using multiple threads, it's possible to handle different paths through the code at the same time. This is known as concurrency and we will explore it in much more detail in a later chapter.



Processes and threads

In Linux, processes are represented by **process descriptors**. These are just big, fat data structures that hold all of the above information. The kernel maintains a list of descriptors, known as the **task list**, that tracks all of the system's processes. Processes are uniquely identified by their **process ID** (PID). To learn about currently running processes, we can use the `ps` utility (`-c` tidies the output and `-f` provides some extra information):

```
$ ps -cf
  UID   PID  PPID   C STIME   TTY           TIME CMD
  501 51112 51110   0  6:14PM ttys000    0:00.03 iTerm2
  501 51114 51113   0  6:14PM ttys000    0:00.21 -bash
```

You can see immediately that I have a Bash shell running in a terminal (iTerm 2). `UID` refers to the ID of the owning user (i.e. me) and we know `PID`. But what is `PPID`? Every process is created by a parent process. PPID refers to the parent's PID. Processes are like bacteria: they reproduce by splitting themselves in two. When a process wants to create another process, it has to **fork** itself (via a system call, of course). The OS creates a duplicate of the process that differs only by its PID and PPID. If process 123 forks itself, the new child process will have a new PID – let's say 124 – and a PPID of 123. The child process is emptied out and a new program loaded into it. We end up with a child process running an entirely different program. This is how the OS handles starting new programs.

If I want to find the process that started my terminal, I need to search for the process with a PID matching the shell's PPID:

```
$ ps -cf -p51110
UID   PID  PPID   C STIME   TTY           TIME CMD
501 51110     1   0  6:14PM ??         0:40.86 iTerm2
```

It turns out that iTerm 2 created process `51110` too. Apparently, iTerm 2 creates a new process for each tab it opens. What initialised the original iTerm2 process? Let's have a look at this process `1`:

```
$ ps -cf -p1
UID   PID  PPID   C STIME   TTY           TIME CMD
  0     1     0   0 Tue05PM ??         1:53.20 launchd
```

What is this new devilry? We have a process with no parent PID! How is this possible? If every child has a parent then what is the first parent? `PID 1` belongs to a special program known as `init` which is directly initialised by the kernel during the boot process. It's the first process to run and is responsible for starting every other system process. On my macOS system, the program is called `launchd` and it has started iTerm 2 because I configured macOS to run iTerm 2 on startup. On Linux, the exact command depends on the distribution though `systemd` is the most popular (and the most unpopular).

In the Unix model, processes are modelled as pipes. User input goes in one end, known as standard input, and output comes out the other, known as standard output (there is also standard error for error output). To communicate with a process without sending it input, the OS needs a separate communication channel. The OS can send **signals** to a process. A signal is a number indicating what the OS wants the process to do. Most signals have defined meanings that the process should respect.

For example, when you terminate a process by typing `Ctrl-c` on the terminal, you are actually sending the process a signal known as `SIGINT` (interrupt). The idea is that the program will receive this signal and know that it's time to gracefully shut itself down, perhaps writing any state to disk and closing any resources its opened. There's nothing that compels a program to respond in this way, however. A pathologically deviant programmer could respond to a `SIGINT` by doing whatever they want. Only convention and good grace prevent anarchy. Misbehaving programs can be terminated by using the `kill` command. Standard usage will send a `SIGTERM` (terminate) signal, which is more forceful than `SIGINT` but still fairly polite way. If the process is still recalcitrant, you can bring out the big guns in the form of `kill -9`, which sends a `SIGKILL` (kill) signal which immediately terminates the program without allowing it to do any clean up. The flag is `-9` because `SIGKILL` is assigned that number. You can read about available signals by calling `man signal`.

Processes communicate with the OS via return codes. They work just like the return values of ordinary functions. Normally, of course, the user is more interested in the **side effects** of the program's execution – output to the screen, files changed and so on – so the return code is mostly used to check whether a program ran without errors. A return code of `0` conventionally indicates success and a non-zero value indicates some kind of error. In a Bash terminal you can query the return code of the last command using `$?`:

```
$ touch file.txt
$ echo $?
0 # touch exited successfully
$ rm incorrect_name.txt
rm: incorrect_name.txt: No such file or directory
$ echo $?
1 # an error occurred
```

The exact meaning of each return code depends on the program. The man pages for `rm` are rather unhelpful:

> If an error occurs, rm exits with a value >0.

Scheduling processes

Processes give the OS the ability to stop, pause and restart executing programs. Swapping between multiple processes is known as **multitasking**. Time available on the processor is divided into slices. Each process runs for a few time slices and is then swapped out so that another process can run. From the perspective of the processor, it's sequentially executing a succession of different processes. From the point of view of a human user with its slow, meaty brain, the switching between processes is so rapid that all of the processes appear to run *at the same time*. It's just an illusion, of course, but it's very convincing when your computer appears to be playing music, rendering a moving cursor, loading a website and compiling some code all at once. When a computer appears to perform multiple tasks at the same time, we say that it is performing them *concurrently*. In the concurrency chapter we'll look at what this means for programming. In this section, we'll confine ourselves to how the OS implements concurrency.

The act of stopping one process, storing its state and loading another is called a **context switch**. Performing a context switch is a comparatively slow operation because the OS needs to save absolutely everything belonging to the process and then recreate it identically. When the OS initialises a new process, execution begins afresh from the first instruction. After running for a while, the process' state will have changed to reflect the effects of the computation. The program counter will have a new value, for example, and maybe the process opened a file or two and updated some values in memory. When the OS performs a context switch, it must record the information it needs to pick up execution from exactly where it left off. Since the process isn't computing anything when not on the processor, it won't even be aware that it's been suspended, just as we aren't aware of anything when we're asleep. From its perspective, it has exclusive use of the processor and the only slightly unusual thing is that the clock time suddenly jumps forwards every now and then, indicating the time during which it was suspended.

Programmers like to moan about "context switches" when they are asked to switch the task they're working on. Building up a mental model of a particular problem often takes some time and switching frequently between tasks means you have to continually rebuild your mental model for each task. Either that or it's just an elaborate excuse to not have to deal with tech support issues. Who knows!

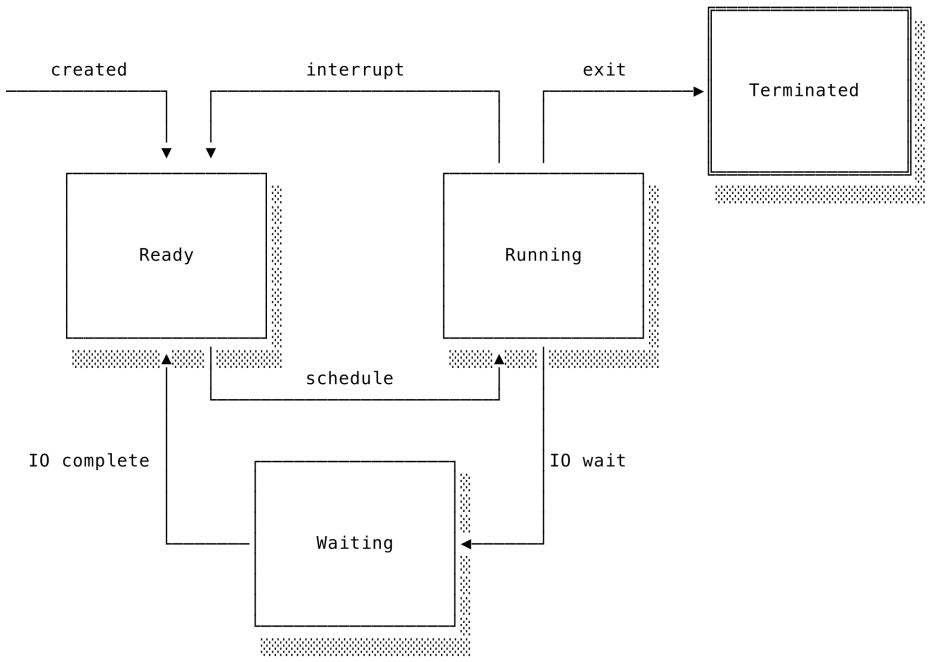
To recap, a multitasking OS will divide the processor's execution time among multiple processes, each taking it in turns to run for a short period. This raises a couple of thorny questions: which processes should run, in what order and for how long? The part of the OS responsible for answering these questions is known as the **scheduler**.

From the OS designer's perspective, the easiest thing is to just leave it up to the programs to sort it out amongst themselves. The OS can sit back and watch as programs voluntarily cede control to each other in a polite and considerate manner. This approach, known as **cooperative multitasking**, was used on early versions of Windows and Mac OSes. Sadly, this approach is akin to expecting small children to voluntarily cede control of toys. One badly- or maliciously-written program could behave like a troublesome two year old and block the whole system by refusing to cede control. Just as adults sometimes need to step in and put someone on the naughty chair, cooperative multitasking has been replaced by a more interventionist approach from the OS known as **preemptive multitasking**. In a preemptive system, a periodic interrupt suspends the current process and invokes the scheduler, which can decide whether to keep the current process going or switch to another.

To implement multitasking, the scheduler needs the concept of **process state** to describe whether a process is currently running and whether it is runnable. The precise states vary between scheduler implementations but they generally follow this simple model:

- A *running* process is currently executing on the processor.
- A *waiting* process is ready to run but hasn't yet been loaded on to the processor.
- A *blocked* process is unable to run because it needs some resource to become available before it can proceed.
- A *terminated* process has finished executing or has been sent a signal by the OS telling it to stop.

The state machine looks like this:



The states of a process

This model enables the OS to make much more efficient use of system resources by minimising dead time waiting for resources to become available. At any given moment, a process can have one of two constraints preventing it from successfully completing. It might have instructions that still need to be executed by the processor. Such processes are **CPU-bound** because the limiting factor is time on the CPU. A process might otherwise be waiting for some kind of input/output (I/O) action to complete such as reading from memory or disk, getting user input or sending something over a network. Such processes are **I/O-bound** because they can't continue until the I/O action completes.

If the scheduler puts an I/O-bound process into a blocked state until the I/O request completes, it can use the time to run CPU-bound processes. Once the I/O action is complete, the process is unblocked and ready to continue. The process is now CPU-bound because the only thing stopping it from continuing is lack of CPU time. The scheduler will eventually schedule it to run on the processor. If, while executing, the process becomes I/O-bound again, the scheduler just repeats the process. This approach is preemptive because the scheduler takes control of when and for how long each process runs.

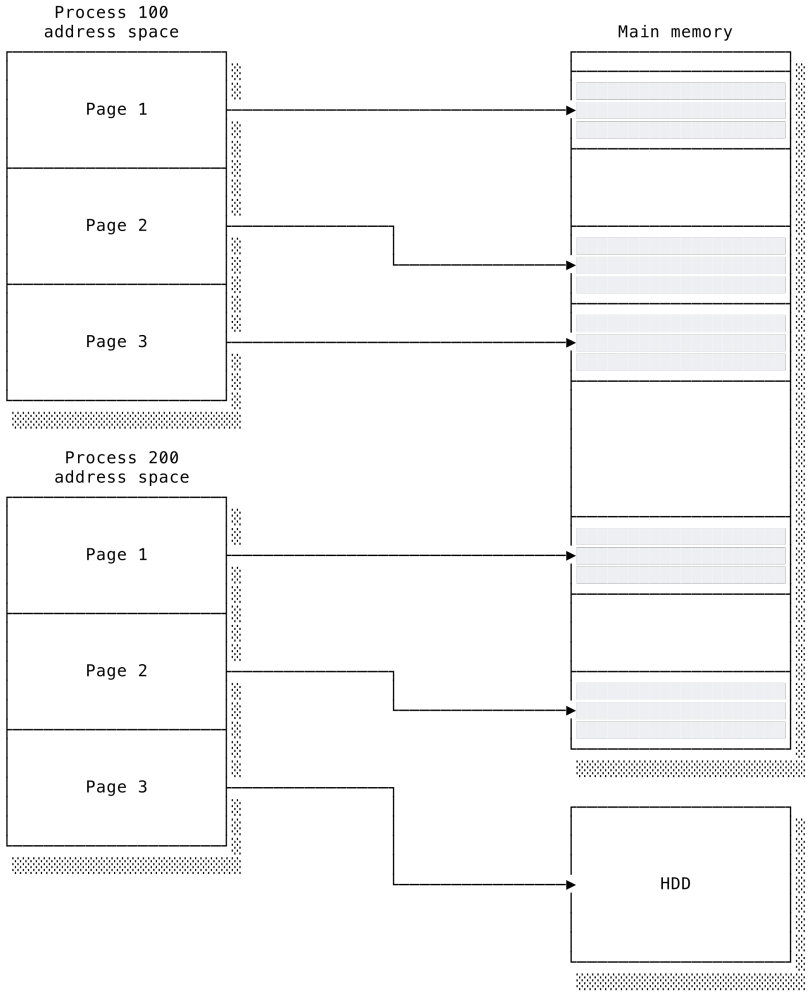
This is works so well because, as we saw in the previous chapter, the processor is much, much faster than everything else. I/O requests are several orders of magnitude slower than CPU operations. It's clearly undesirable for the processor to sit idle while it waits for an I/O operation to complete. Rather than waste huge numbers of CPU cycles, the scheduler will unload the blocked process and load a CPU-bound one instead. The aim is keep the processor's utilisation rate as high as possible. The computer can keep doing meaningful work while it waits for the I/O to complete.

A preemptive scheduler might decide to swap out a running CPU-bound process, even though the process would have been able to continue executing, in order to give other processes a slice of processor time. This avoids one process hogging all of the CPU time and **starving** other processes

by not letting them have any CPU time at all. Even among CPU-bound tasks some might be higher priority than others. Decoding a streaming video is time sensitive and so higher priority than a background task. Scheduling lots of processes in a way that's considered fair is a challenging and active area of research.

Managing memory

We've seen how the OS forms a protective layer above the physical hardware. Nowhere is this more evident than in how memory is handled. The OS and processor work closely together to fool user processes into thinking that they have the computer's entire address space to themselves. Every memory address that a user process sees and uses is actually fake. This deceit goes by the name of **virtual memory**.



Virtual memory

The OS allocates each process a contiguous block of virtual memory addresses known as the **virtual address space**. This seemingly contiguous block is actually cobbled together by the OS from unused

chunks of real memory. The OS and processor map the virtual addresses to the real addresses in a way that is completely invisible to the process. To do this efficiently is difficult and requires an entirely new subsystem in the processor: the **memory management unit** (MMU). Why go to all this bother? Why not just give the process real addresses?

Virtual memory is a bit like a well-meaning lie. It makes things much easier for the process if it believes that it has access to a nice, big empty block of memory. If the process could look behind the veil and see the true state of affairs, it would be appalled. It would see that the OS had actually allocated it many chunks of memory distributed across the address space. The process would have to keep track of the size and location of each chunk, which ones are filled, which addresses it controls, which it doesn't and lots of other irritating considerations. It's far easier to take the blue pill and continue living in the fake world constructed by the operating system. I'm sure we all know the feeling.

Virtual memory is also desirable because it leads to more efficient use of available memory. If a program requests, say, 100MB of memory, the OS doesn't need to find a single, 100MB chunk. It can combine smaller sections of free space that maybe would otherwise be too small to allocate. This reduces the amount of **memory fragmentation**, where the blocks of used memory are interspersed with lots of unusably small bits of free space.

When the OS converts a virtual address into a physical address, it also has the opportunity to check that the requesting process has permission to read the requested location. After all, we don't want a process reading memory that belongs to the kernel or to a different process. This is known as **process isolation** and is another example of how the OS acts as a gatekeeper, protecting the system's security.

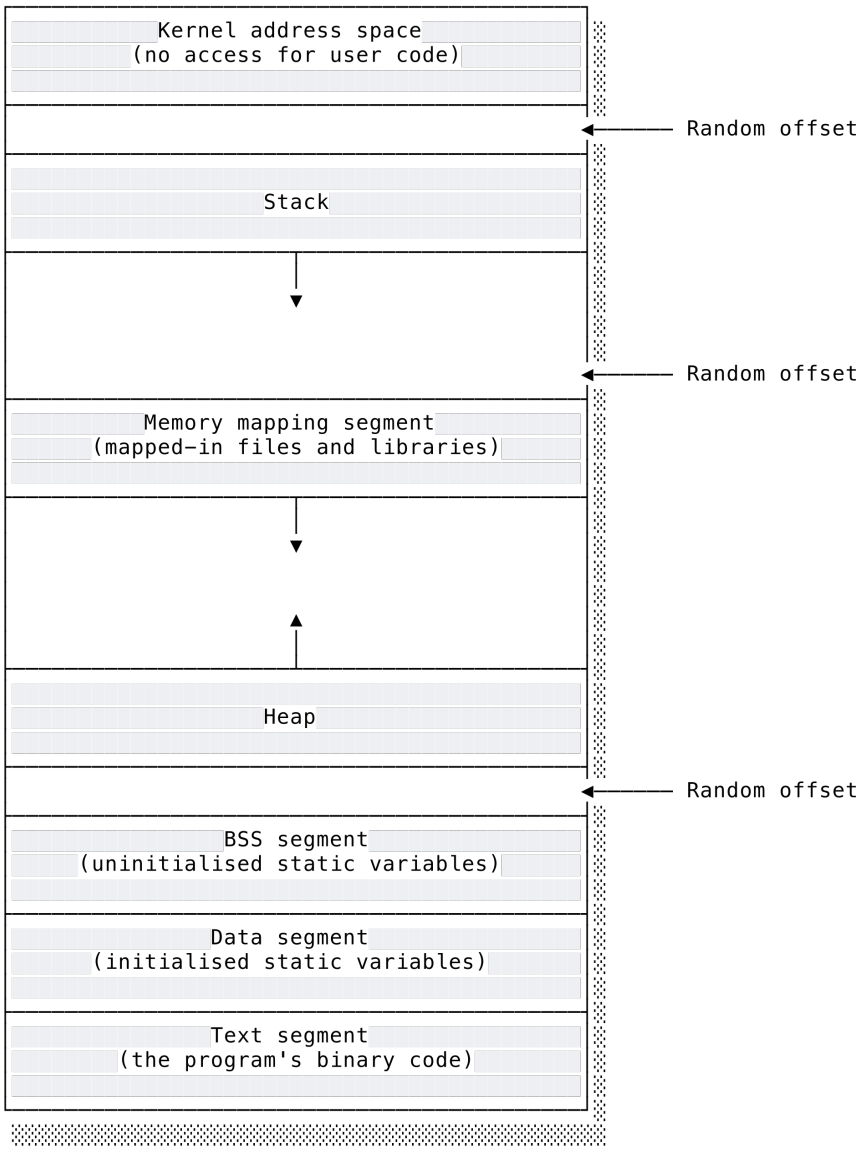
Finally, virtual memory allows the process to make simplifying assumptions about its address space. Each process will think that it has access to the same address range every time it runs. Programs can therefore make assumptions about where their resources will be located in memory because they will always have the same virtual addresses. Without this it would be much more difficult for a program to locate all of its resources.

The OS deals in chunks of memory known as **pages**. Usually a page contains a few kilobytes of memory. The process's address space will be made up of multiple pages. The total amount of virtual memory allocated in pages can even exceed the amount of physical memory in the machine. Not every physical page backing a virtual page needs to be actually held in memory. The OS can **swap** unused pages out of memory and on to the hard disk. Imagine that a process has been allocated 50 virtual pages but there is only space in physical memory for 30. The remainder will have to be stored on the disk. Whenever the process attempts to read a virtual address and the processor's MMU can't find the backing page in memory, it will trigger a **page fault**. This is a type of exception that tells the OS to swap the required page back into memory, swapping out some other page to make room if need be. The OS repeats the read attempt, which will now succeed since the page is in memory. The process is completely unaware that this is going on. All of the work is handled by the OS. From the process's point of view, it has access to all 50 pages of memory. Some memory operations will simply appear to take a little longer than others. It is blissfully unaware that the OS is busily swapping pages in and out of memory. In effect, the OS treats main memory as a giant cache for the hard drive.

Having studied the memory hierarchy, we are painfully aware that read operations to the hard disk will be much, much slower than reads to an in-memory page. When a computer has very limited physical memory available it can get itself into a terrible situation known as **thrashing**. The OS is forced to constantly swap pages back and forth between memory and the hard disk. This takes so much time that the computer doesn't have time to do anything else. This is why you might have noticed your computer slowing to a crawl before showing an out of memory error. Modern paging systems are very sophisticated and make heavy use of optimisations, including hardware caches in the MMU, to try and avoid this happening.

The process address space

When a process forks itself, the child process initially has a copy of the parent's (virtual) address space. In order to execute a different program, we need to set up a fresh address space. The OS loads the new program's code into memory and sets up the address space. The precise structure of the process address space will vary slightly from OS to OS, but the general approach doesn't change much. A program's executable is a combination of machine instructions and data. The kernel loads the different elements into various **segments** that make up the address space:



Address space of a process

The program's instructions (in machine code form) are loaded into the **text segment**. The instructions are the program "text" that the processor will read and execute. Once everything else has been loaded, the OS will set the program counter to the address of the first instruction and off we go.

All but the most trivial programs will contain variables. Initialised variables are those that have a value hardcoded in the program code. They live in the **data segment**. Uninitialised variables are just variables without an initial value. They live in the snappily-named **block started by symbol (BSS) segment**. The reason for keeping them separate from the initialised variables is so that the kernel knows that they have to be initialised to zero when the program starts. If they were stored in the data segment, they'd need to include their zero values in the program code, which would be a waste of space. To easily remember the purpose of the BSS segment, just say to yourself "better save

space!".

The **stack segment** is the process's working space. You might remember from the data structures chapter that a stack is a last-in-first-out (LIFO) data structure. Whenever a function is called, a structure called a **stack frame** is added to the top of the stack. It contains everything that function call needs: the function's arguments, any local variables and the return address.

An executing function has access to its own stack frame and the frames of enclosing functions. All of the variables in these frames are in the function's **scope**. The frames below the current frame in the stack would have been created by the function's ancestors: the parent function that called the current one, the function that called the parent function and so on back to the very first function. You've probably seen a **stack trace**, which is logged when an unhandled exception occurs. The stack trace shows the complete set of function calls that resulted in the current, exception triggering function call.

When a function returns, its stack frame is removed from the top of the stack. This is why you can't access a function's local variables from outside of the function. When we return from a function and go back to the parent scope, the child's stack frame is removed and its variables become unavailable. Here is an example in JavaScript. Note that JavaScript code executes in a runtime environment that provides the JavaScript program a separate stack but the principle is the same:

```
function run() {
  let A = 'a';
  let B, C;

  console.log(A, B, C); // 1

  function inner() {
    let B = 'b';
    C = 'c';
    console.log(A, B, C); // 2
  }

  inner();
  console.log(A, B, C); // 3
}

run();

// output:
a undefined undefined
a b c
a undefined c
```

When `run` is called, it declares three variables: `A`, `B` and `C`. It only assigns a value to `A`. At the first

comment `B` and `C` have undefined values. Next, we call `inner`, which assigns a value to `C` and declares its own, local version of `B` and gives it a value. At the second comment all three variables have values. However, the `B` with a value is local to `inner`. It only exists in `inner`'s stack frame. It **shadows**, or masks, the undefined `B` in the outer scope. The assignment to `C`, on the other hand, is an assignment to the original `C` in `run`'s stack frame – it doesn't create a local variable. When we return from `inner`, its stack frame is destroyed and we lose all knowledge that the local `B` ever existed. At the third comment we can see the change to `C` has persisted but the `B` in `run`'s stack frame remains undefined.

A problem with the stack is that the amount of space allocated to local variables in each stack frame is hardcoded into the program's source code. This means that they have to be known when the program is written. What do we do if the size of the variable will only be known when the program runs? For example, imagine that we write a C function that asks the user to enter their name. The user's input will be written into an array declared in the function. How can we allocate enough space for the array if we don't know how long the name will be?

One (pretty poor) solution would be to decree that no name may be longer than 100 characters and to allocate space for an array of 100 characters on the stack. To avoid buffer overflows, we'll need to check that the input length is less than 100 and discard anything extra. Forgetting to do this would be a major security hole because C will let you write past the end of the array's allocated space on the stack, overwriting anything following the array. This is a problem because it could mean overwriting important things such as the function's return address. An attacker could craft an input that runs past the end of the array and replaces the return address with the address of some malicious code, causing the computer to start executing it when the function returns.

Security concerns aside, what about users who happen to have very long names? Clearly our approach is far from ideal. The better solution would be for the program to keep track of how full the array is and increase the size when necessary. It can't do this in the stack frame because the array is already surrounded by other values. The process address space has an area known as the **heap** for just this kind of thing. The process can request (via an OS system call) chunks of memory that are allocated from the heap. This is known as **dynamic allocation** because the amount allocated is determined by the running process rather than being hardcoded into its instructions, as in the case of the stack.

In a language like C, which provides no **memory management**, it's up to you to keep track of what's on the stack and what you're allocating on the heap. Getting memory management right is notoriously tricky and a huge source of bugs. It's therefore very common for languages to offer some kind of automatic memory management. When you run a Ruby program, the Ruby runtime is in charge of deciding where to allocate variables and performing all the allocations and deallocations. Writing some simple programs in C is a good way to improve your understanding of how the stack and heap both work.

In terms of address space layout, it's important to notice that the *stack grows down* and the *heap grows up*. The first element of the stack lives at a high memory address. As the stack grows, the addresses of the frames decrease. Both the stack and heap need space to grow dynamically as the program runs. By having one grow up and other grow down, they can both grow into the empty space in the middle. Of course, it's still possible for the stack to extend so far downwards that it uses

up all of the available space and butts up against the memory mapping segment. This is known as a **stack overflow** and usually happens because an incorrect use of recursion creates too many stack frames.

The random offsets you see in the diagram implement **address space layout randomisation** (ASLR). If the stack and heap always begin from exactly the same location, it would be easy for a malicious program to work out where particular variables (e.g. passwords) will be. Adding randomly sized offsets makes it harder for an attacking user to guess their locations.

Managing persistence

Any data that the user wants to survive a system power-off needs to be written to **persistent storage**. Mechanical hard disks commonly provide large capacities and relatively slow performance. In recent years **solid-state disks** (SSD) have proven popular as a much faster, albeit less capacious, form of persistent storage. Usually persistent storage, either mechanical or SSD, forms the bottom tier of the memory hierarchy. The storage device will typically present itself to the system as a flat expanse of space. It is up to the OS to impose structure.

The OS subsystem responsible for organising persistent storage is known as the **file system**. A file system groups related data into a single unit known as a **file**. The file system will provide some way of structuring files and expressing their relations to one another. On Unix-like systems, the OS creates a **virtual file system**, organised as a hierarchical tree with `/` at the root of the tree. Files path, which therefore always begin with `/` on Unix-like systems, indicate the route to a unique resource in the hierarchy. The file system is "virtual" because sub-trees may contain different concrete file systems from different storage devices.

Operating systems support many file systems offering competing balances of performance, security and reliability. Due to space constraints I'm not going to cover file systems in any detail. As ever, resources are available in further reading should you wish to study them further.

There are two points I do want to cover. Firstly, a process accesses each file through a **file descriptor**: a reference to a per-process data structure that records the state of the process' interactions with the file. On a Unix-like system, every process starts with three file descriptors representing standard input, standard output and standard error. File descriptors are a limited system resource. It's a common mistake to forget to close a file descriptor after opening. Eventually the OS will run out of file descriptors, probably causing your program to crash unless you handle this possibility. Some languages, such as Python, include constructs which automatically close a descriptor when it's no longer in use.

Secondly, the huge speed differential between the CPU and persistent storage devices means that caching is used heavily. To improve performance, the OS will not write a change directly to persistent storage. Instead, it will update the in-memory page and only write the page to disk when the page needs to be removed from the cache. The benefit is that multiple writes to the page will be batched together and only one write to persistent storage is needed. Storage devices usually have their own internal caches, completely invisible to the OS, as a further performance improvement.

This caching can create problems for some applications, particular databases, that need to be sure that a write has actually been written to persistent storage. By default the OS will consider the write to have succeeded once the in-memory page has been updated, even though the persistent storage hasn't yet been updated. Should a system failure occur at that point, the supposedly successful write would be lost and the database would have an incorrect understanding of what is in persistent storage. Similar problems arise if the storage device says that the write has succeeded when it has only reached the internal cache. To handle such cases where performance is secondary to reliability, OSes and storage device drivers usually provide the ability to **flush** writes directly to persistent storage.

Conclusion

In this chapter we saw how the operating system forms an intermediary layer between the hardware and the user's programs. The job of the OS is to protect the hardware, present a consistent execution environment and improve performance. To achieve these objectives, OSes rely on hardware safety mechanisms provided by the processor. The kernel is a trusted, privileged program that has complete control over the system's resources. It orchestrates and supervises the execution of untrusted, unprivileged user programs. The OS mediates all attempts to access system resources through the system call interface, supported by the interrupt mechanism in hardware.

To manage resources effectively, the OS creates virtualised representations of these resources. In each case the purpose is to abstract away the hardware specific details and provide a consistent, secure and easy-to-use interface to user programs. A process is a representation of an executing program. All computation happens within a process, allowing the OS to schedule programs to permit concurrency and optimal resource usage. Virtual memory is a virtualised process address space. It allows the OS to efficiently allocate memory while also presenting a simple, consistent address space to the process. The process address space is split into different segments. Each function call generates a stack frame. The stack therefore represents the current state of the program's execution. The heap is used for dynamically allocated values. A file system is a representation of how data is arranged and stored in persistent storage.

We have reached an important milestone on our journey. Over the course of the first four chapters we developed our mental model of how a computing system functions. We understand every element right down to the logical bedrock. In the rest of the book we will explore what we can do with this platform.

Further reading

If you read only one book on operating systems, I recommend Operating Systems: Three Easy Pieces[28] (OSTEP), a wonderful, freely available resource. It approaches operating systems by focusing on three concepts: virtualisation, concurrency and persistence. You'll cover Linux processes, virtual

[28] http://pages.cs.wisc.edu/~remzi/OSTEP/

memory and file systems. For more detail you could try *Operating System Concepts* by Silberschatz, Galvin and Gagne. It includes advanced chapters on security, virtual machines (the basis of cloud computing) and distributed systems. If you find yourself fascinated by operating systems and want a very thorough treatment, *Modern Operating Systems* by Tannenbaum is the authoritative textbook in the field.

To keep up with developments in the Linux world and learn more about the kernel, I highly recommend the Linux Weekly News (LWN)[29] newsletter. There is a small monthly fee to get the latest issues but the archives are accessible for free. Its articles range from new contributions to the kernel, summaries of recent conferences and deep dives into the design and history of various features.

Finally, nand2tetris includes chapters covering virtual machines and operating systems. I found the operating system project to be fairly lacklustre. It does require you to write a memory allocator, which is very important, but the rest is implementing a few library functions. More useful is the virtual machine project, which will improve your understanding of process' execution environment.

[29] https://lwn.net

Networking

Introduction

In the space of just a few decades, computer networking has gone from being a curiosity involving screeching modems to being the primary reason for having a computer. Many modern laptops and smartphones aren't much more than an interface to the Internet. Using a computer without an Internet connection feels so limited that it's hard to believe that the first version of Windows 95 didn't even ship with Internet Explorer!

A large amount of modern day software development involves at least some web work. For this reason, it's absolutely critical that you have a firm understanding of computer networking, the Internet and the associated web technologies. In this chapter we'll develop a conceptual model of networking using stacked layers of protocols. We'll then use that model to study the Internet, its major protocols and the web.

What is a network?

Let's begin by getting our terminology straight. Computers can be connected together to form a **network**. A **host** is any device connected to a network. Hosts on a network are able to exchange information. The network might involve physical connections using cables (so retro!) or, more likely these days, wireless connections using WiFi or Bluetooth. Either way, there will be a component in each host known as the **network interface card** (NIC) that is responsible for encoding and decoding messages to and from whatever physical signals the network uses.

If you want to call someone, you need to know their phone number. It uniquely identifies the phone you're calling out of all of the phones connected to the phone network. It's the same for computer networks. Each host on the network has a unique identifying address determined by the NIC. One host can talk to another by using its NIC to broadcast across the network a message addressed to the NIC of the receiving host. The receiving host's NIC will listen to all the messages on the network. Whenever it detects a message addressed to it, the NIC will excitedly read the message into the computer's memory and notify the OS using the interrupt system we saw in the OS chapter.

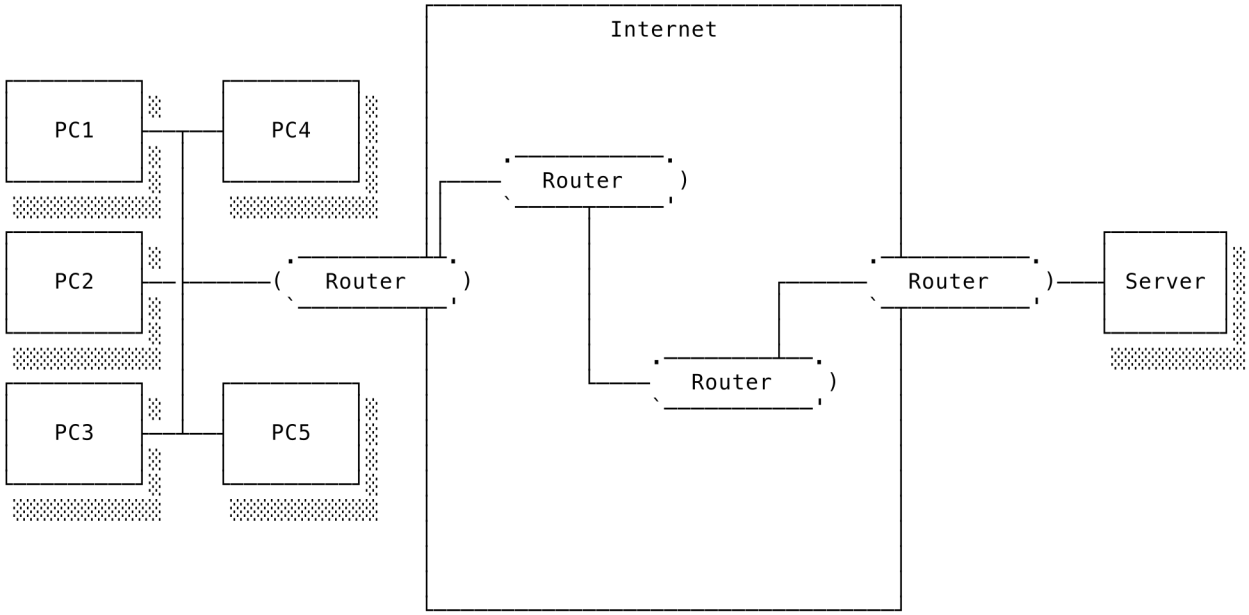
If a network only allows machines to communicate with other machines on the same network, then it is referred to as an **intranet** because all of the communication stays within (*intra*) the bounds of the network. Lots of businesses run their own private intranets, accessible only to computers within the office, to share access to storage, printers and so on.

Some networks include machines that are connected to more than one network. Such machines can be configured to function as **routers** by accepting messages from a host on one network and passing

it on to a host on another network. This enables **inter-network communication**. A message from a sender in one network can reach a receiver in another network even though there is no direct connection between the sender and receiver.

The **Internet**, with a capital I, is one single, planet-wide inter-network of networks. Any two points on the Internet can communicate with each other by sending a message across their own little sub-network to a router, which sends it across another sub-network to another router and so on across successive sub-networks until the message reaches the destination's sub-network and can find the destination machine.

Once the bits have arrived at their destination, whether via an intranet or the Internet, the receiver has to be able to correctly interpret them. We use **protocols** to specify the format of information that's sent from machine to machine. The Internet is based on a bundle of protocols known as **TCP/IP**, clever routing algorithms and systems for managing and retrieving device addresses. We will look at all of these in this chapter.



The Internet

In this diagram we have an intranet on the left. One host on the intranet, the router, also has a connection to the Internet, most likely provided by an **internet service provider** (ISP). There is a path through the Internet routers to the server, which exists on a separate intranet with just its own router for company. The PCs and servers might be on opposite sides of the world.

Messages are not sent in a single burst but are split into chunks known as **packets**. Each packet is sent independently of the others and may take a different route depending on network congestion and availability. The Internet is designed, in theory at least, to be decentralised. No single person or body controls the Internet. It is also designed to be resilient. If a node on the network – one of these machines straddling multiple sub-networks – fails for whatever reason, the network traffic will automatically find another route to its destination. It also improves the network's utilisation by

encouraging each packet to find the fastest route. A physical analogy would be a convoy of cars all heading from one place to another. If a car ahead gets stuck in traffic the cars behind can try taking a different route in the hope of avoiding the congestion.

The Internet forms a reliable communication layer on top of which various "applications" have been built. I don't mean applications in the sense of desktop applications. Internet applications are closer in meaning to use cases for the Internet. Lots of things that you might think of as "the Internet" are actually applications in this sense. **Email** is a collection of protocols which enable electronic messages to be sent across the network. Another, known in the 1990s as the World Wide Web and now just the **web**, allows users to request and receive multimedia content. I'm pushing things here, but the Internet is a little like an operating system for networks. It provides a way to reliably and consistently get a message from A to B across diverse networking machinery, just as an OS reliably and consistently runs programs on many disparate computer architectures. You don't need to worry about how your message is delivered because the Internet handles all of that for you.

In non-technical use, the Internet and the web are basically synonymous. This is because the web has become by far the most dominant application on the Internet. Despite this, it's important to remember that the Internet and the web are *not* the same thing. The Internet is a communication layer that enables devices to talk to each other across network boundaries. The web is just one of many uses we have found for the Internet. The Internet predates the web by several decades.

By this point you may be wondering what exactly a protocol is. Can it be found somewhere in your computer? How does it actually *do* anything? How does the computer know that this huge stream of incoming bits should be interpreted with this or that protocol?

Cast your mind back to how the processor decodes instructions. How did the processor "know" that a particular bit pattern corresponds to a particular instruction? It didn't, really. The humans who designed the processor imposed meaning by designing the processor's decoding circuitry in a particular way. The processor is blindly operating according to its design and it relies on the incoming instructions being encoded as expected. The processor's instruction set architecture forms a protocol between the programmer and the processor: give me an instruction in this format and I'll execute this behaviour.

A protocol is a set of rules and expectations specifying how data should be transmitted. The target of a protocol is really other people. They're the ones who will sit down and program their computers to interpret the bitstream you're sending according to the agreed protocol. As long as every machine follows the protocol correctly, everything will work as desired. Everyone will agree which meanings should be assigned to which bits. The computer doesn't "know" that the incoming communication is using a particular protocol. It merely attempts to interpret it according to whatever protocol the programmer has instructed it to use. It will happily try and interpret a message with an incorrect protocol if you tell it to do so.

The Internet is on a vastly different scale to the gates and components we've looked at in previous chapters. Performance costs that were almost infinitesimally small on the scale of an individual computer suddenly matter very much when we're working on the scale of a planet.

From the processor's perspective, it takes an extremely long time for messages to traverse the Internet.

Network requests are slower than local, system requests by several orders of magnitude. **Latency** measures how long it takes a message to reach its destination. A certain, unavoidable amount of latency is due to the time it takes the message to travel across the Earth. Additional latency can be added by the performance of the machines along the message's path through the network. Latency is usually measured in milliseconds and lower is better. As a rough estimate, it takes 150ms for a message to go from London to San Francisco. That is thousands of times slower than a local memory access.

The other important performance consideration is how much data can fit into the network's piping at once. This is known as the **bandwidth**. Clearly, if we can transmit more bits of data at once, then we can transmit a message more quickly. Bandwidth is measured in bits per second and higher is better. In my lifetime we've gone from a humble 56Kbps (kilobits per second) to megabits and now gigabits per second.

Both latency and bandwidth are important for performance but in different ways. If you want to transfer a lot of content (e.g. streaming video), then you're interested in bandwidth. It doesn't really matter whether it takes 50ms or 500ms for the first bit of content to reach you so long as the bandwidth is sufficient to keep the data buffers full and your video playing. If the network activity involves lots of **round-trips**, when the sender transmits something and waits for a response from the receiver, even fairly small changes in latency can have a large impact on the total transmission time. On the other hand, low latency won't do us much good if a lack of bandwidth means we can only send a minuscule amount of data per second.

The networking stack model

The complexity of the Internet means that each message has to convey a lot of information. Let's take the example of posting a comment on someone's blog. So that your message can get to the right host and so that the receiving computer can know what the hell to do with it, your transmission's packets will need to specify: the receiver's network address; the application protocol; tracking information so that the complete transmission can be reconstructed; and the actual content of your comment itself.

As ever, when faced with great complexity, we can make it more manageable by breaking down the problem. The solution in networking is to layer multiple protocols on top of each other to create a **network stack**. Each layer has a different role. This modularisation creates a highly effective separation of concerns. The layer that's responsible for delivering a message across a network is completely separate to the layer that routes the message between networks and *that* layer is different from the layer that instructs the receiver how to interpret the message.

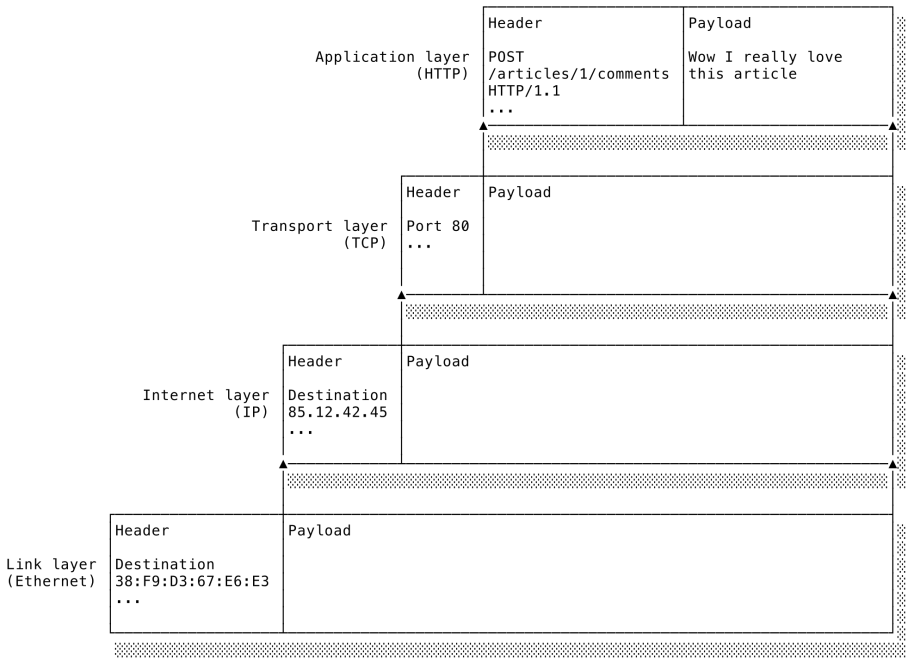
There are two networking stack models that you will come across. The more general one is known as the **OSI model** (after Open Systems Interconnection, though you'll never need to know that). It has seven layers. The simpler model is that of the **Internet protocol suite**, which defines how the Internet works. It has four layers that broadly correspond to specific OSI layers. We'll focus on the Internet protocol suite's stack model in the interests of simplicity.

Internet layer	OSI number	Common protocols
Application	7	HTTP, SMTP, FTP
Transport	4	TCP, UDP
Internet	3	IP
Link	2	Ethernet

The functionality of each layer is implemented by the various protocols that operate at that layer. At the heart of the communication is the **payload**: the actual message that you want to transmit. In this example it's the blog comment you want to post. Each layer wraps around the content of the previous layer, including that layer's metadata. It's like the nested Russian dolls known as *matryoshki*.

The NIC is at the bottom of the stack and user programs are at the top of the stack. Starting from the top, the outgoing payload generated by the user program is wrapped in each layer's metadata. This means that the link layer, at the bottom, forms the outermost metadata wrapper. On the receiving computer, the process works in reverse. First it unwraps the link layer metadata, then the internet layer and so on until the application layer passes the payload to whatever user program is configured to handle the message. The code that performs all of this work is referred to as the operating system's **networking stack** and generally resides in the kernel.

The final, wrapped message that is broadcast or received on the network has layers of metadata that look roughly like this:



A network packet

Let's look at each layer in more detail, starting from the bottom. As mentioned above, the foundations of Internet networking is TCP/IP. This is actually a combination of two protocols working on different layers.

Link layer (OSI layer 2)

The link layer represents the computer's physical connection to the network via the NIC. This layer is responsible for sending the packets, technically referred to as **frames** at this layer, out over the physical network. The protocols at this level can only communicate with hosts that the machine is physically connected to. A common example of a link layer protocol is Ethernet. If you're not using WiFi, you probably plug your computer into the router using an Ethernet cable.

One of the main tasks of the link layer is to handle situations when multiple devices attempt to broadcast over the network simultaneously. This is resolved by having both parties wait for a random time before reattempting the communication. One will go first and successfully broadcast its message. It's a bit like that awkward thing when you're speaking to someone on the phone and there's a bit of a delay (i.e. latency) so you both try to speak at the same time and then wait a bit for the other person to speak. This is technically referred to as *media access control*, or MAC, from which we get the name "MAC addresses".

Every physical connection to the network is assigned a unique MAC address in the form of six bytes usually expressed in hexadecimal e.g. `60:f8:1d:b1:6f:7a`. MAC addresses are usually assigned permanently to the NIC hardware and do not change. The host retains its MAC address as it moves between Ethernet networks.

Usually the OS will allocate a block of memory that is shared between the OS and the NIC. When the NIC receives a packet, it can write it directly to the shared memory. This is known as **direct memory access** and is more efficient that the NIC having to ask the processor to copy the packet into memory. The processor carries on with its own work while in the background the NIC takes care of moving the packet into memory. Once the packet is ready, the NIC notifies the kernel via an interrupt. When the kernel wants to transmit data, it writes to the shared memory and signals to the NIC to start transmission.

Internet layer (OSI layer 3)

At this layer we break out of the confines of a single network. Here is where we meet the IP part of TCP/IP – the Internet Protocol (IP). The IP network is an overlay over interconnected link layer networks.

The Internet Protocol provides **routing** and **packet forwarding** across networks. Each host on the IP network is assigned its own **IP address**. The Internet Protocol is responsible for routing the packet across network boundaries to the correct host. It is thanks to the Internet layer that we can communicate with machines that are not directly connected to our network. One important limitation of IP is that it only offers **best-effort delivery**. It provides no guarantee that the packet will ever reach its destination, nor does it tell you if it arrived. Sending an IP packet is the networking equivalent of screaming into the void.

Since each layer builds on the previous one, a device connected to the Internet will have two addresses: a MAC address from the link layer and an IP address from the Internet layer. Each address

only has meaning at its own layer. It doesn't make sense to address an IP packet to a MAC address. The MAC address represents the physical machine while the IP address, in a more abstract sense, represents a point on the network. Usually your **Internet service provider** (ISP) assigns you an IP address when you connect to the Internet.

We will look at IP in more detail below.

Transport layer (OSI layer 4)

The transport layer provides *services* that the basic Internet layer lacks. For example, we saw that IP connections are unreliable. If we want to be sure that a message has been delivered, we need some additional functionality. Protocols at this layer provide services such as delivery guarantees or the concept of a sender-receiver connection that survives between transmissions.

The Transmission Control Protocol (TCP) is the main protocol the Internet uses at this layer. It builds on IP by providing important functionality that IP does not offer. We'll see some of them when we look at TCP in more detail below. For now, just bear in mind that these features come with a performance cost because the protocol has to send various confirmation and acknowledgement messages back and forth. In TCP, packets are technically referred to as **segments** but we'll stick with "packets" for simplicity. Just remember that the term means something slightly different at each layer.

Another common transport protocol is the User Datagram Protocol (UDP). This protocol forgoes a lot of TCP's bells and whistles in order to deliver a higher throughput of data. It's used for things such as streaming media and online games, where lots of information needs to be sent quickly and it doesn't matter so much if a few packets get lost along the way.

Application layer (OSI layer 7)

At the top of the stack we have the application layer. The lower layers are the networking nuts and bolts which ensure that a message is successfully delivered without really caring about the message itself. The protocols on the application layer play no role in the transmission. They are concerned with the message itself. Each application on this layer specifies a protocol for the data that it communicates over the network. "Application" here doesn't strictly mean a particular program running on your computer. It's more akin to the use case you have for the network in the first place: sharing a file, sending a textual message and so on.

There are a whole range of protocols covering everything you can do on the Internet. Very important protocols include the File Transfer Protocol (FTP) and the Simple Mail Transfer Protocol (SMTP). Hopefully their purposes are obvious. These protocols have been around for decades; more recent examples include BitTorrent and Bitcoin.

By far the most common application layer protocol is the Hyper Text Transfer Protocol (HTTP). We'll look at it in detail later in the chapter. As we'll see, HTTP has expanded far beyond its original use case and morphed into a kind of protocol behemoth that's used as a general communication protocol on the Internet.

The Internet Protocol

Let's now look at the major Internet suite protocols in more detail. We've already seen that the Internet Protocol (IP) is responsible for getting packets of data across network boundaries to the correct destination. Each device on the Internet is allocated an IP address. If a machine needs to be easily accessible, such as a server, it will often have a **static IP address** that doesn't change. That makes it easier to locate on the network. When you connect your home router to your ISP's service, however, your router is likely assigned a **dynamic IP address** from a pool of available addresses. A dynamic IP address might change at any moment but that's not a problem because the only Internet traffic going to your address will be in response to requests initiated by you as you go about your online business.

How do the servers you talk to know where to send their responses? The answer lies within the packets themselves. IP packets are made up of a **header** containing metadata and data containing the message payload. The Internet Protocol defines the structure of the header so that both the sender and receiver have a shared understanding of which bits mean what:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Version|  IHL  |Type of Service|          Total Length         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|         Identification        |Flags|     Fragment Offset     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Time to Live |    Protocol   |        Header Checksum        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         Source Address                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                      Destination Address                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Options                    |    Padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Don't worry too much about what everything means. The important thing is to understand how the information is laid out. Each little rectangle represents a single bit. The numbers along the top count the bits. We have six rows of 32 bits so 192 bits in total. Different bit ranges are allocated to different metadata fields. The most interesting sections are the `Source Address` and `Destination Address` fields. When your computer sends out an IP packet, it writes its own IP address as the source. When the server generates the response, it sets that address as the destination.

IP addresses are 32 bits long. As we know from the computer architecture chapter, a 32-bit value can be split into four bytes, each holding 256 values. This is how an IPv4 address is usually written:

```
154.24.126.19
```

This means that there are in total 2^{32} or 4,294,967,296 unique IP addresses. More than four billion addresses seems like a huge number but in recent years this limit has become an issue. We'll see some workarounds and solutions later.

IP routing

It's pretty straightforward to send a message to a computer on the same network. Every device listens to all of the traffic on the network, hoping that there will be something addressed to it. All you need to do is broadcast a message with the correct address and the receiver will notice. Things aren't so clear when the sender and receiver are on different networks. It isn't immediately obvious how to get from my computer to, say, Netflix's servers. This is where IP's secret sauce comes in.

When the packets leave my home network, the first place they reach is the router belonging to my ISP. Routers aren't anything special. In theory, any computer can be a router so long as it's connected to more than one network. What makes a computer a router is that it's running a program that routes packets. It's the *activity* that makes it a router, not some special physical characteristics. You can make your own home router with a Raspberry Pi, if you like. Of course, ISPs' routers have to handle huge amounts of traffic so they have optimised hardware. They don't do anything a Pi router couldn't do – they just do it on a much, much bigger scale.

A router maintains a list of known IP addresses. When the router receives a packet, it will inspect the packet's destination address. If the router knows the address, it will forward the packet directly to the destination. If the router doesn't know how to deliver the packet, it will instead try and route it to another router that's a bit closer to the destination. Each step from router to router is known as a **hop**. Each hop takes the packet a bit farther along its path until it hops on to a router that *does* know the destination and can successfully deliver the packet. The specifics of how the next-hop router is chosen depends on the particular algorithm used by the router.

In the packet header you can see a field called the **time to live** (TTL). This records how many hops the packet is allowed to make. At each hop the receiving router decrements the packet's TTL. Once the TTL hits zero the router **drops** the packet and will not forward it any further. This prevents packets from getting caught in a loop, endlessly cycling through the same routers and clogging up the network. Packets can also be dropped if a router can't process incoming packets quickly enough. Every packet will eventually leave the network by either reaching its destination or getting dropped.

You can see the route taken by packets across the Internet by using a **tracerouter** such as mtr[30]:

[30] https://www.bitwizard.nl/mtr/

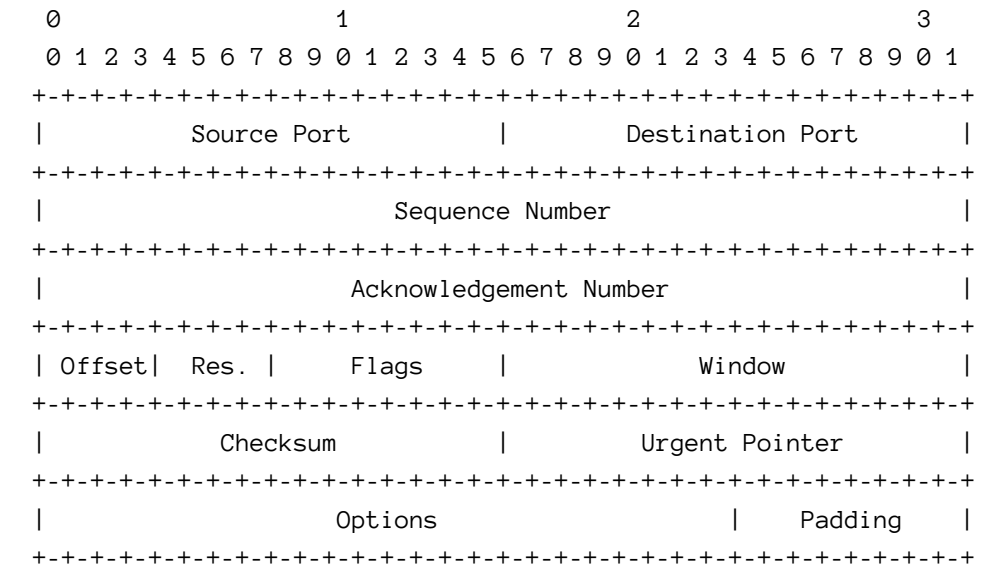
```
$ sudo mtr www.google.com
                                        Packets               Pings
 Host                                 Loss%   Snt   Last   Avg  Best  Wrst StDev
 1. 119.76.44.185.baremetal.zare.com  0.0%    17   33.2  35.3  24.7  51.3   6.5
 2. 1.76.44.185.baremetal.zare.com    0.0%    17  186.4 155.7  76.1 287.0  53.2
 3. ae2.31-rt1-cr.ldn.as25369.net     0.0%    17   29.6  33.6  26.5  42.3   4.2
 4. ae1.rt0-thn.ldn.as25369.net       0.0%    17   43.8  38.0  29.0  86.8  13.3
 5. telia.thn.bandwidth.co.uk         0.0%    17   24.1  35.4  24.1  52.0   7.6
 6. ldn-bb4-link.telia.net            0.0%    17   77.2  45.3  26.2  77.2  16.6
 7. slou-b1-link.telia.net            0.0%    17   34.5  37.2  26.2  54.6   6.1
 8. 72.14.203.102                     0.0%    17   31.1  35.9  24.0  81.9  12.9
 9. ???
10. 172.253.65.208                    0.0%    16   32.7  39.7  27.2  90.2  15.1
11. 216.239.56.193                    0.0%    16   37.4  40.2  27.5  57.3   8.2
12. lhr25s15-in-f4.1e100.net          0.0%    16   35.1  40.6  28.9 100.1  17.1
```

This shows the complete route from my computer to Google's server `1e100.net` (10^{100} being a googol). Host number one is the one closest to me and number twelve is Google's server. Each host is a router on the way. The tracerouter sends out a bunch of packets with increasing TTLs to the destination IP address. The first packet, with a TTL of one, will only get as far as the first hop before it gets dropped, the second, with a TTL of two, will get as far as the second hop. The packets sent by `mtr` use a protocol called Internet Control Message Protocol (ICMP) to request identifying information from each router. Not every router will agree to provide information, which is why you see `???` in the output above. The pings indicate, in milliseconds, how long each hop took.

Note that it's possible for an IP communication to fail if the packet's TTL runs out before it reaches its destination. The packet header also doesn't include any information about whether the packet is part of a bigger sequence. Because of these limitations, the client cannot know whether the packet was successfully delivered and the receiver cannot know whether any packets were dropped by the network.

Transmission Control Protocol

To solve these problems we must turn to the Transmission Control Protocol (TCP). Remember that TCP sits in the layer above IP in the network stack. When the OS receives an IP packet, it strips off the header to leave just the data payload. The payload is a TCP segment, which once again consists of a TCP header and a data payload. Here's what a TCP header looks like:



The TCP header contains useful information such as its position in the overall transmission, known as the **sequence number**. The receiver can use this to put the segments in the correct order and detect gaps where packets were dropped by the network. The receiver can then notify the sender and request their retransmission. TCP can almost guarantee that the packets are delivered in the correct order. I say "almost" because if the underlying IP layer can't deliver the packets, there isn't much TCP can do to remedy that. What it *does* offer is the certainty that it will successfully deliver the packet or generate an error response. This is a big improvement on IP, which doesn't give any response.

TCP implements a hugely helpful abstraction in the form of a two-way **connection** between the sender and receiver. The hosts at either end can read and write streams of data from the connection, much as they would with a local file. The protocol hides the fact that it splits the message up into lots of little packets and recombines them on the other end. On the receiving host, the program that consumes the TCP stream just sees the message pop up in the same form as it was originally sent.

Remember that TCP sends its messages using IP. It builds the abstraction of a reliable connection on top of something that fundamentally lacks both reliability and the concept of a connection. TCP utilises a sophisticated state machine of messages and acknowledgements to help the sender and receiver maintain the same expectations about who is sending what at any given moment. However, because the intermediate network is unreliable, it is impossible to completely *guarantee* consistency between the sender and receiver. The sender could send a message to check that the last packet was successfully received but, because the IP layer is unreliable, that message and any acknowledgement from the receiver could both be lost.

In computer science this is known as the **Two Generals problem** (also Byzantine Generals) after a thought experiment in which two generals try to coordinate an attack by sending messages to each other through territory controlled by an enemy. It has been proven that it is impossible for the generals to guarantee that they will both attack at the same time (i.e. achieve consistency). They can

never be sure that any message they send will be successfully delivered and not intercepted by the enemy.

Nevertheless, a TCP connection is much more reliable and easier to use than sending raw IP packets. Unfortunately, this does not come for free. Negotiating the initial connection requires a few round trips, inevitably adding latency at each step. Time is also spent waiting to see if missing packets turn up and requesting the retransmission of dropped packets. In addition, neither the sender nor the receiver know what data transfer rate the intervening network can handle. Although TCP has the ability to automatically tune the transfer rate to the network's capabilities, each connection has to begin at a conservative rate that gradually increases until packets start dropping off. This ramp up is required every time a new connection is made, even between the same hosts.

TCP ports

Thanks to the combination of TCP/IP, we now have the ability to reliably deliver a complete message to an IP address of our choosing. How does the receiver know what to do with the messages it receives? The TCP header has a defined format but the data payload could be anything. Referring back to the network layers diagram, we observe that above the transport layer is the application layer. How does the receiver know which application protocol to use?

Note that the TCP header specifies a **source port** and a **destination port**. By convention, each application protocol is assigned a port number. The OS takes the delivered message and passes it to the correct port. Programs running on the receiver can **listen** on the port by registering themselves with the OS. The convention is that a program listening on a given port will correctly parse messages using the port's assigned protocol.

The combination of IP address and port number uniquely identifies an application on a particular host:

```
123.84.25.100:80 // HTTP, used by web servers
123.84.25.100:22 // Same host but this time connecting to SSH
44.95.132.12:80 // HTTP but on different host
```

So we have one number that indicates the host and another that tells the host what to do with the message. If you squint and look sideways, this might remind you of how Linux uses one interrupt number (`0x80`) for all system calls and disambiguates them with a second system call identifier. Both are examples of multiplexing. Remember from the computer architecture chapter that multiplexing involves sending signals intended for different destinations along the same route. The receiver demultiplexes the signal by examining the destination of each signal and dispatching it correctly. Networking makes heavy use of multiplexing. The central parts of the network carry lots of packets all on their way to different destinations, just like a motorway carries vehicles that are on their way to different destinations. Within a particular connection between two hosts, there is further multiplexing at the port level. This is like two people travelling in the same car to the same building but then going to separate floors within the building.

On Linux, ports are implemented using **sockets**, a special type of file. At one end of the socket, the OS kernel writes the network messages once they've been reconstructed by the TCP/IP stack. The listening application simply reads the messages from the socket. This creates a neat separation between the network-oriented part of the kernel and the listening applications, bringing the usual benefits of separating concerns. It means that user programs can read from and write to the connection as easily as a local file (via system calls, of course). The OS hides all the hard work and provides user programs with a simple interface.

Ports numbered below 1024 are designated "well-known" ports and have an assigned protocol. There's nothing stopping you from making your own protocol and using one of the well-known ports, but you won't be able to use it to communicate with anyone else because they will expect that port to use the standard protocol. Remember that in networking convention is key!

Service	Port number
SMTP	25
HTTP	80
HTTPS	443
FTP	20, 21
SSH	22

On Unix-like systems, a program normally needs superuser privileges (i.e. using `sudo`) to connect to a well-known port. The original idea was to stop some random user logging into a shared computer and replacing a well-known service such as FTP with their own nefarious program. This restriction has actually become something of a security issue itself because it means that lots of Internet-facing programs need to run with escalated privileges. Servers such as nginx try to mitigate this by using the superuser privileges to listen on the port and then immediately de-escalate themselves to lower privilege levels as much as possible.

Internet addressing and DNS

We've seen that every host on the Internet is assigned a unique IP address. Let's now explore some problems that have arisen as the Internet has grown. The first thing to be aware of is that there are actually two versions of IP in common usage: IPv4 and IPv6 (we don't talk about v5). IPv4 is the currently dominant form and what we've been looking at so far. As we know, IPv4 addresses are 32-bit integers, giving over four billion possible addresses.

Four billion is not so many in a world of billions of Internet-connected phones and devices. The situation is made worse because not all of the address space is available. Various sections of it have been allocated for special purposes. For example, addresses `10.0.0.0 - 10.255.255.255`, `172.16.0.0 - 172.31.255.255` and `192.168.0.0 - 192.168.255.255` are designated for use on private networks (see below) and should not be used for addresses on the open Internet. In the early days of the Internet, many companies and institutions were allocated vast, virgin tracts of the IP address space in a way that seems rather overgenerous nowadays. The U.S. Department of Defense, Apple and

Amazon all own millions of IP addresses, making them unavailable for general use. Simply put, the Internet has run out of IPv4 addresses.

An initial solution involved simply hiding lots of connected devices from the public Internet. You don't need a public IP address if you're not on the public Internet! When I ask Google "what is my IP address?", it tells me `185.44.76.119`. Yet when I ask my computer by running `ifconfig | grep inet`, I get a different address: `10.6.6.128`. What is going on? There is a clue in the IP address my computer tells me. The address is within the range `10.0.0.0 - 10.255.255.255`, which is reserved for private networks. This means my computer isn't directly connected to the Internet! In fact, the only device connected to the Internet is my router. My computer sits behind the router in a **private network** inaccessible from the public Internet. Every time a device connects to my WiFi network, the router assigns it an IP address on my local private network. This is why my computer has a private IP address.

How then does my computer communicate with hosts on the Internet? Every IP packet my computer sends will have the private address in the source address header field. The first destination of every outgoing packet will be the router. Before routing the packet on to the public Internet, my router will replace the private IP address with the router's own public IP address. When the router receives the response, it will reverse the process and redirect the packet to my computer's private IP address.

No matter how many computers are connected to my network, I only need one public IP address for my router. In this way a single public IP address can multiplex a large number of computers. Technically the router is acting as a **gateway** between my local private network and the wider Internet. As well as drastically reducing the number of IP addresses needed, this also offers benefits to network administrators who can put a firewall in front of the gateway or monitor all the traffic going through it.

Useful as private networking is, we have not addressed the fundamental problem of insufficient IP addresses. To increase the number of addresses we need to allocate more bits to the address fields in the packet headers, which in turn requires amending the protocol. IPv6 is an updated version of the protocol using 128-bit addressing. 2^{128} is an absurdly large number that will provide sufficient IP addresses even for a dystopian, Internet-of-things future in which every kitchen utensil has its own Internet connection. An IPv6 address is written as eight blocks of hexadecimal characters:

```
3ffe:190a:4545:3:200:f7ff:fe21:37cf
```

This means that IPv4 and IPv6 are not compatible. An IPv4 address can be expressed as an IPv6 address but a router that only understands IPv4 won't know what to do with an IPv6 address. The uptake of IPv6 has been pretty slow, partly because of this, but you will see more and more of it.

DNS

Using binary addresses, whether 32-bit or 128-bit, is all well and good for computers but not very useful for humans. How would you like to have to remember `216.58.223.110` (Google's IP address) every time you wanted to search for something? Not a great user experience. From a developer's

perspective, IP addresses tie you very tightly to a particular host. If you redeploy a web app on to a different server, the web app's IP address will change. For most purposes an IP address is simply too low level. A big improvement is human-readable **domain addressing**. These are the `www.google.com` addresses that we all know and love. They are implemented by the **Domain Name System** (DNS).

DNS allows hosts on the Internet to be identified by a **domain name** that maps to an IP address. This mapping can be easily updated by the owner of the domain, relegating IP addresses to an implementation detail of no concern or interest to users. The domain owner can update a DNS record to point to a new IP address without the user even realising.

The structure of DNS addressing is hierarchical. This is reflected in the domain name by separating each level of the hierarchy with full stops e.g. `www.example.co.uk`. Reading a domain from right to left will take you from the top level to the most specific. At the very top you have the **root zone**. The identifier for this zone is an empty string so it's not visible in the domain name. Next are the **top-level domains** (TLDs) such as `.com`, `.edu`, `.uk` and so on. At each point of the hierarchy there is an administrator who is responsible for domains within their zone. The administrator of each TLD can allocate domains within the zone. A domain can point directly to a host or contain more subdomains.

We will take as our example `cs.ox.ac.uk`, the domain of the computer science department at the University of Oxford. After the implicit root we have the `uk` TLD, managed by the non-profit company Nominet UK. Within that TLD is the `ac` subdomain, which is reserved for academic institutions. The University of Oxford has been assigned the `ox` domain. The university's administrators are responsible for the domains within this zone. They have decided to give the computer science department its own `cs` domain. The departmental administrators responsible for this domain have decided to create a `www` subdomain pointing to a web server so that whenever you navigate to `www.cs.ox.ac.uk` it renders a web page.

Note that `www` doesn't have any special meaning. It is only by convention that a `www` subdomain points to a web server. As the web has exploded in popularity, it's more common than not that the *only* service a domain offers is a web server and so often the domain administrator will set the domain itself to point to the same place as the `www` subdomain. We'll see how this is done soon.

Since humans prefer to use domains and computers prefer to use IP addresses, we need a way for a computer to find the IP address of a given domain. Let's look at what happens when you type `www.cs.ox.ac.uk` into your browser, both in theory and in practice.

In theory, your computer will make a request for the domain's IP address to the root zone server. The root zone server will tell you to go and ask the **authoritative name server** of the top-level domain. The TLD's name server will tell you to ask the name server of the next level down in the hierarchy. This continues until you query a name server that has the IP address mapping. Here's what it looks like:

```
$ dnstracer -4 -r1 -s. www.cs.ox.ac.uk
Tracing to www.cs.ox.ac.uk[a] via A.ROOT-SERVERS.NET, maximum of 1 retries
A.ROOT-SERVERS.NET [.] (198.41.0.4)
 |\___ dns2.nic.uk [uk] (103.49.80.1)
 |     |\___ ns4.ja.net [ac.uk] (193.62.157.66)
 |     |     |\___ dns1.ox.ac.uk [ox.ac.uk] (129.67.1.191) Got authoritative answer
```

First we query the root server, then Nominet UK (`nic.uk`), then the `ac` domain administrators (`ja.net`) and then finally the name server of the `ox` domain.

Starting at the root means we can always find our way to a particular domain but it's inefficient and puts a lot of pressure on the root servers. *In practice*, the system works in reverse, thanks to caching. When you navigate to `www.cs.ox.ac.uk`, your computer will first check if it has a local cache of the IP address mapping. If not, it requests the address mapping from your ISP's name server. This server maintains a cache of previously-seen domains and very likely can answer your request straight away. If not, it will certainly have cached the authoritative name server for the `ac.uk` subdomain and can request the mapping from that name server. Extensive caching saves the root servers from having to handle every name request on the Internet. They do still play a central role, since they are ultimately authoritative, and so you sometimes see them referred to as the "backbone of the Internet".

Each cached mapping in the name servers has a time-to-live (TTL) value set by the domain's authoritative name server. Once this expires, the mapping is removed from the cache and the name server will need to re-fetch the mapping the next time the domain is requested. If you've ever configured a domain name, you may have encountered a warning that changes might take several hours to propagate across the Internet. This is because you need to allow time for the name servers to flush the old IP address mapping from their caches.

DNS stores its mappings in **DNS records**. There are multiple types but they all map a domain (or subdomain) to another value. `NS` records provide the location of the authoritative *name server*. `MX` records are used for handling email addresses. `A` records contain the actual IP *address* mapping. `CNAME` are *canonical* records which map one subdomain name to another. There can be only one `A` record but there can be multiple `CNAME` records. A typical DNS record for a domain might look like this:

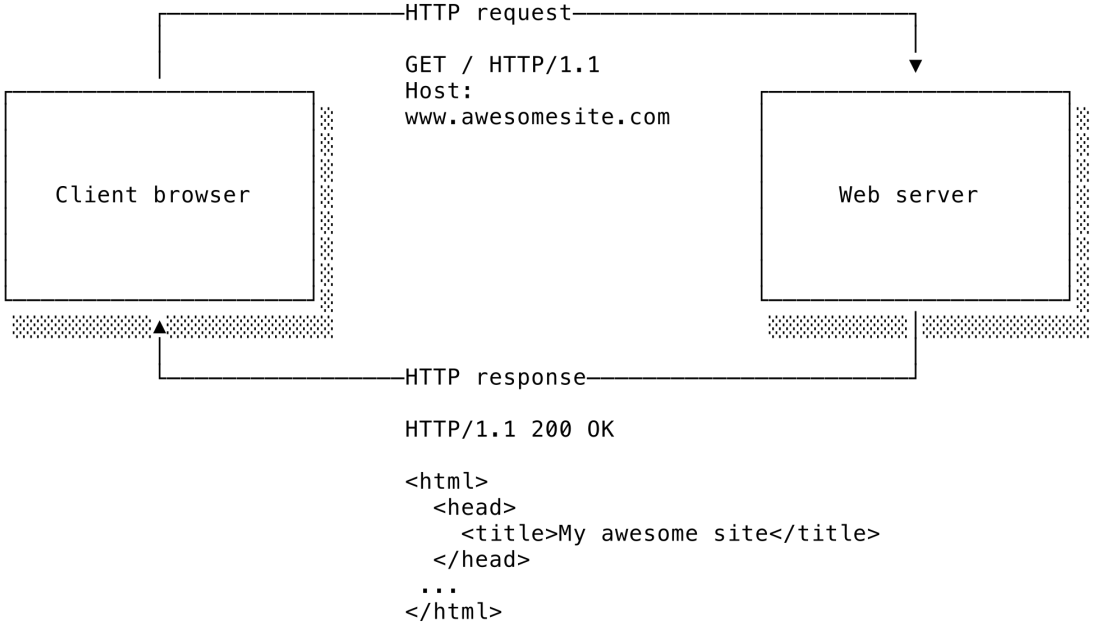
```
example.com     A     100.21.45.53
www.example.com CNAME example.com
```

When you do an `A` record lookup to `www.example.com`, the server will respond by telling you to do the lookup to `example.com`. Repeating the request to the new domain will result in the IP address we're looking for. Sadly, I always get `A` and `CNAME` the wrong way round because to me `A` sounds like it should stand for "alias" and "canonical" should refer to the IP address. I wish you better luck remembering this.

The web, hypertext and HTTP

We end this chapter by looking at the most popular Internet application protocol: the Hypertext Transfer Protocol (HTTP).

HTTP is what powers the web. It's a protocol for requesting and delivering **hypertext**. Despite the futuristic name, hypertext is nothing more than blocks of structured text with connections, or **hyperlinks**, to other blocks of hypertext. The text is structured through the use of **HTML** (Hypertext Markup Language), which semantically tags each bit of hypertext. A **client** sends an **HTTP request**, specifying the requested **resource**, to a server which provides an **HTTP response**. And with that we have the foundations of the web!



HTTP client-server model

The server is so called because it has a set of resources, or content, that it is willing to serve out to anyone who requests it. A server is any machine running a program capable of responding to requests. The term "server" is rather overloaded: you'll see it used to refer to both the physical machine and the program. When the client navigates to a resource, the browser renders the content, requesting any further resources (e.g. images, fonts) that are specified in the content. Clicking on links or otherwise interacting with the resource will generate further requests.

That's pretty much it. By itself HTTP doesn't have the concept of an ongoing relationship between the client and server. It's just a sequence of independent requests. Many requests from one client are dealt with in the same way as one request from many clients.

When the user clicks on a link, the browser generates an HTTP request, described below, to the domain for the specified resource. The browser first asks the OS to perform a DNS lookup to get the server's IP address. It then opens, via OS system calls, a TCP connection to the IP address on port

80. The browser writes the request to the connection. The OS's networking stack is responsible for the various protocol steps: negotiating the connection, transmitting the packets, resending dropped packets etc. When the server's OS receives the packets, it does the work in reverse to reconstruct the original message. The server OS then passes the message off to whatever application is listening on port 80. This will hopefully be an HTTP server. The server parses the request, generates an appropriate response (e.g. retrieving an HTML file from disk), and passes it off to the OS to be sent back to the client over the TCP connection.

HTTP uses TCP port 80. The original idea of different application layer programs using different ports fell foul of network administrators who, in the interests of security, preferred to block traffic on all but the most important ports. HTTP is essential so it's guaranteed to work. This has led to the current situation, in which HTTP is used as a generalised communication protocol simply because it's safe to assume that the communication won't be blocked. Another advantage is that HTTP is simple and easy to read. An HTTP request is just plaintext over multiple lines:

```
1 GET / HTTP/1.1
2 Host: www.my-great-book.com
```

The first line is known as the **request line**. The first word (`GET`) specifies the **HTTP request method**. This tells the server which action the client is requesting. The second element specifies the requested resource. At the end of the line we specify the particular form of the HTTP protocol we are using. `HTTP/1.1` has been dominant since the early 90s, though its successor, `HTTP/2`, is gradually becoming more widespread.

Here we are requesting the root resource (`/`) of the `my-great-book.com` domain, which defaults to `index.html`. If we want to request a different resource, we simply need to specify it: `GET /static/styles.css HTTP/1.1`.

After the request line comes a series of **headers**, which are encoded as key-value pairs separated by a colon. The example request has only a single header specifying the host name. Other headers include `User-Agent`, which provides information about the client, and various `Accept` headers, which specify the response types the client is capable of understanding. After the headers comes an empty line and an optional message body. The body is also known as the **payload**. Data submitted by the client, such as the content of an HTML form, is stored here. The example above has no payload.

There are nine request methods but only a few are commonly used. A useful way of understanding the different methods is to map them to four operations: create, read, update and delete. A server that responds to requests in this way is said to provide a **RESTful interface**.

- `GET` tells the server that we want to *read* the resource at the specified path e.g. `GET /users/1` requests information on the user with ID `1`.
- `POST` says that we want to *create* a resource using the data submitted in the request body e.g. `POST /users/` creates a new user with the submitted parameters. The response should specify how to access the newly created resource e.g. by providing the new user's ID.

- `PUT` says that we want to *update* the specified resource with the request payload e.g. `PUT /users/1` updates the user's parameters to match the submitted payload.
- `DELETE` tells the server to *delete* the resource at the specified path e.g. `DELETE /users/1` deletes the user with ID 1.

It is very important to use request methods correctly. `GET` is known as a *safe* method that only retrieves information and does not change the server's state. A `GET` operation should be **idempotent**, meaning that it produces the same result if performed multiple times. `POST`, `PUT` and `DELETE` may all change the server's state. Webcrawlers, used by search engines to index the web, will follow `GET` links but will not follow links with unsafe methods to avoid changing the server's state. Make sure you get this right. If you implemented a deletion functionality using links like `GET /photos/1/delete`, everything would appear to work correctly until one day a webcrawler came across your site, followed every `GET` link and deleted every resource on your website!

An HTTP response is also plaintext over multiple lines:

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
Content-Length: 55743
Connection: keep-alive
Cache-Control: s-maxage=300, public, max-age=0
Content-Language: en-US
Date: Thu, 06 Dec 2018 17:37:18 GMT
ETag: "2e77ad1dc6ab0b53a2996dfd4653c1c3"
Server: meinheld/0.6.1

<!DOCTYPE html>
<html lang="en">
<head>
  <title>Buy my book</title>
</head>
<body>
  <h1>Best book ever</h1>
</body>
</html>
```

The first line is the **status line**, which specifies the protocol version and the response's **status code**. You may already be familiar with `200 OK`, meaning success. As with the request, the next section is a list of headers containing the response metadata. There are too many to go through in detail, but from a cursory examination we can see the content type and length, server information, metadata for caching and others. No header is absolutely required but many are very useful. After the headers there is an empty line and then an optional message body. In response to a `GET` request it contains the requested resource. In this case, it's the HTML resource requested by the client.

The response code is a number that indicates the response type. Sometimes the response code alone conveys enough information and the response contains no further data. In other cases, the response includes more data in the message body (e.g. the `200 OK` above). Response codes fit into five general categories as indicated by the first digit:

- `1xx`: information responses, rarely used. The server might send `100 Continue` if the client is sending a lot of information and the server wants to reassure the client that everything's going fine.
- `2xx`: successful responses. `200 OK` is most common and means that the request was successful.
- `3xx`: redirection messages. The resource is available but not at the requested path. The response body will indicate to where the client should make the request.
- `4xx`: client error messages. The client has made an invalid request. The classic `404 Not Found` means the client requested a resource that doesn't exist. `400 Bad Request` means that the request was invalid and the server couldn't understand it.
- `5xx`: server error messages. The request is valid but the server has encountered some kind of error. As a web developer, you'll become tragically familiar with `500 Internal Server Error`.

How browsers handle responses depends on the response code. It's very important that you use the correct response code. A common mistake is to indicate an error by returning a JSON encoded error in a `200 OK` response. The browser sees the `200 OK` , thinks everything's fine and so won't pass the JSON object to any JavaScript error handlers, most likely leaving your website in a broken state. Similarly, for the user's browser to automatically follow a redirection response, you must use the correct `3xx` response code.

HTTP specifies simple, human-readable request and response formats. It's easy to observe network traffic just by looking at the HTTP messages going back and forth. You can open your browser's developer tools and read network requests and responses, which is frequently helpful when debugging client-side code.

The main limitation of HTTP is that it is unencrypted. This means that HTTP traffic is visible to anyone watching network traffic. Plain HTTP is obviously unsuitable for anything requiring secrecy. The **HTTPS** protocol, running on port 443, acts as an encrypted wrapper around HTTP. An observer can see that the HTTPS connection has been made but cannot see what passes through it. HTTPS availability has increased massively in recent years, in large part due to concerns about online snoopers. If you are responsible for any web servers, I strongly encourage you to ensure you serve content over HTTPS. You may not think that your content is important or sensitive enough to justify the effort, but don't forget that you are also exposing your users' online activity. You cannot decide on your users' behalf whether their personal activity should be public and so you should give them the option to use a secure connection.

Conclusion

In this chapter we saw how computers can communicate through networking. The Internet is a global network of networks. We saw that a useful conceptual model for networking is a stack of

layers, each responsible for delivering a different part of networking functionality.

We looked in detail at the Internet protocol stack. At the bottom are the physical transmission media like Ethernet and WiFi. Above them is the Internet protocol, which is responsible for routing packets of data across the Internet. In the next layer, TCP adds the abstraction of a persistent, reliable connection between two hosts. The TCP connection is used by applications in the topmost layer. The most popular of these applications is HTTP, which underpins the modern web. We examined the HTTP request-response cycle to better understand how browsers request and render online resources. We also saw how addressing on the Internet is implemented via IP and domain name addressing.

Further reading

There are two ways in which you can continue your study of computer networking. You can focus on the upper half of the networking stack and learn how to deploy applications so that they are secure, performant and scalable. Much of this comes under the modern "devops" banner. Alternatively, you can take the perspective of a network administrator and focus on the lower parts of the stack. I recommend that you start from the application level and continue studying down the stack as far as your interest takes you.

A useful practical exercise is to deploy a web application on a virtual machine in the cloud. If you've only ever deployed things on a managed service such as Heroku, I'd go so far as to say that this is an essential exercise. Take a plain virtual machine on AWS or Google Cloud and set everything up manually yourself: private local network, Internet gateway, firewall, DNS records, load balancing and so on. You'll have a much better understanding of how your application connects to the Internet. Once you've got everything working, it's perfectly acceptable to decide that you'd rather just pay Heroku (or some other service) to save you the hassle in future. By knowing what these services are doing, you can make that decision in an informed way.

My favourite resource for developers is High Performance Browser Networking[31]. In a short space it covers the most important Internet protocols (IP, TCP, HTTP) and includes lots of practical advice on how to make your websites speedy. Basically all of it is required knowledge for web developers. It also includes sections on HTTP/2.0, HTTPS and TLS, which I did not have space to discuss in this chapter. For a deeper understanding, I recommend *Computer Networking: A top down approach* by Kurose and Ross. "Top down" here means that it begins from the perspective of the application developer and works its way down the network stack. Read until you lose interest. Another popular textbook is *TCP/IP Illustrated: Volume 1* by Fall and Stevens. It covers much more than just TCP and IP.

If you make it all the way through those textbooks and still want more, you might be interested in networking from the perspective of a network administrator. Should you wish to go this far, the de facto standard is Cisco's CCNA certification. There is a wealth of free study materials online[32] that you can use without actually signing up to get the certificate.

[31] https://hpbn.co/
[32] https://study-ccna.com

Concurrent programming

Introduction

Programming is undergoing a major transition right before our eyes. In the olden days, programs were executed sequentially by a relatively simple processor. That straightforward model has been upended by the computing industry's pursuit of higher performance. Innovations in chip design require new programming paradigms.

For decades, the processor industry was in thrall to **Moore's law**: an observation by Henry Moore, CEO of Intel, that the number of transistors on a processor doubled every two years. Another observation, **Dennard scaling**, stated that a chip's power consumption would remain constant as the transistors became smaller and denser. Taken together, the two observations correctly predicted that performance per watt would increase rapidly. Between 1998 and 2002, processor speeds increased from 300MHz in a Pentium II to 3GHz in a Pentium 4. That's ten times faster in four years. For programmers this was a good thing. Processor speeds increased so rapidly that upgrading hardware was the easiest way to improve performance.

Such rapid performance increases could not go on forever. Components had to become smaller and smaller so that more could be packed into the same space. Increasingly tiny components made it more and more difficult to dissipate heat and keep the electrons flowing correctly. Clock speeds increased in smaller and smaller increments until finally they increased no more. Manufacturers desperately sought ways to deliver the performance improvements that their customers had come to expect. With no easy way to further increase clock speeds, they came up with the wheeze of bundling multiple processors, or cores, on to the same chip. And thus the multi-core revolution was born.

In theory, a processor with two cores can do twice as much work in the same time as a processor with a single core. In reality, things are not so simple. Doubling the number of cores does not directly improve performance in the same way that doubling the clock speed does. A single-threaded program on a multi-core processor will utilise a single core and leave the others idle. The CPU cannot automatically spread the computation over multiple cores. We must shift to a new programming paradigm in which tasks execute **concurrently** on multiple cores.

There is no single, correct concurrency model. Some programming languages are based on a particular model, while other languages support multiple models. In this chapter, we'll begin by defining some important terms. We'll move on to some basic concurrency primitives provided by the operating system and their limitations. We'll then study the interesting and contrasting concurrency models provided by JavaScript and Go.

Concurrency, parallelism and asynchrony

In this chapter we'll use a few, closely related but distinct concepts. Let's go through each of them in turn.

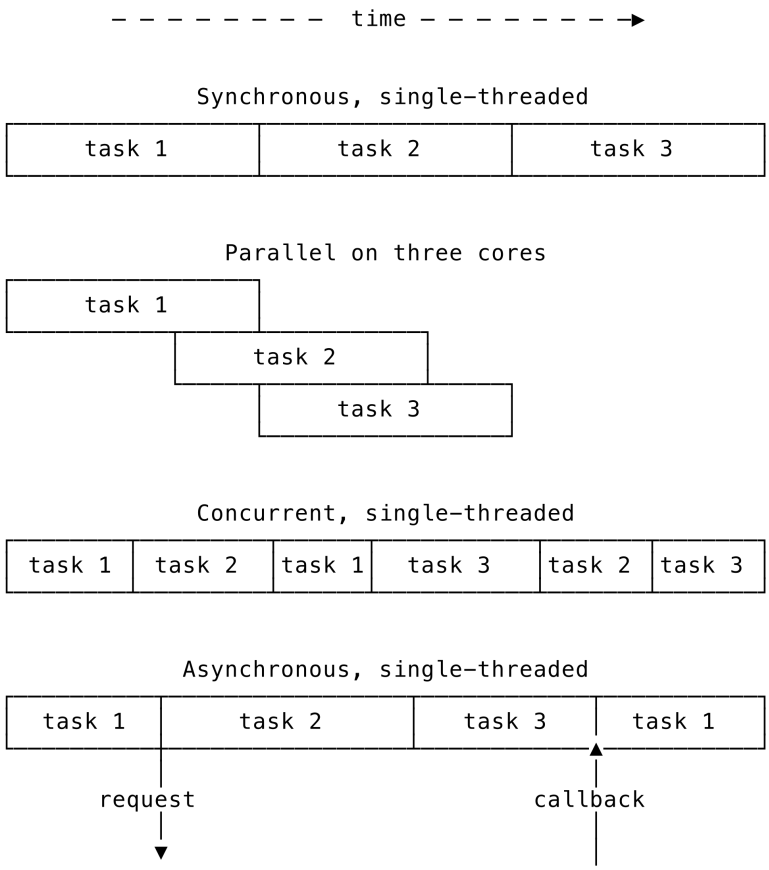
Concurrency means dealing with multiple things at once. A program is concurrent if it is working on multiple tasks at the same time. That doesn't mean that it has to be literally performing multiple tasks at the same time. Cast your mind back to the multi-tasking scheduler we saw in the OS chapter. On a single-threaded processor, there is only ever one process actually executing at any given time, but the scheduler manages to deal with multiple tasks by giving each slices of processor time. The concurrent behaviour observed by the user is created by the scheduler in software.

Parallelism means actually doing multiple things at once. Parallelism requires hardware support. A single-threaded processor physically cannot execute multiple instructions simultaneously. Either hardware threads or multiple cores are necessary. A problem is **parallelisable** if it can be solved by breaking it into sub-problems and solving them in parallel. Counting votes in an election is a parallelisable task: count the votes in each constituency in parallel and then sum the totals to find the national result. Parallelisable tasks are desirable in computing because they can easily make use of all available cores.

To take an example, you are reading this book. We can think of each chapter as a separate task. You might choose to read the book from cover to cover (good for you!), one chapter after another. That is analogous to a processor executing tasks sequentially. Alternatively, you might start one chapter and, midway through, follow a reference to a section in another chapter. You are now reading two chapters concurrently. You are literally reading only one chapter at any given moment, of course, but you are in the process of reading two chapters. To read two chapters in parallel, however, you would need to have two heads and two books!

Make sure that you grasp the distinction between concurrency and parallelism. Concurrency is the slightly vague idea of having multiple, unfinished tasks in progress at the same time. It comes from the behaviour of software. Parallelism means doing multiple things literally at the same time and requires hardware support.

The third concept is **asynchrony**. In synchronous execution, each section of code is executed to completion before control moves to the next. A program that initiates a system operation (e.g. reading a file) will wait for the operation to complete before proceeding. We say that the operation is **blocking** because the program is blocked from progressing until the operation completes. In asynchronous execution, the program will initiate the operation and move on to other tasks while it waits for the operation to complete. The operation is therefore **non-blocking**. In JavaScript, an asynchronous language, to perform an operation you must request it from the runtime environment and provide a **callback** function for the runtime to execute when the request completes. Asynchronous execution is therefore a form of concurrency.



Concurrency, parallelism and asynchronous execution

We saw in the scheduling section of the OS chapter that threads can be either CPU-bound, meaning they need to do work on the processor, or I/O-bound, meaning they are waiting for an I/O operation e.g. reading a value from memory or sending a file over a network. When a thread makes an I/O request via a system call, the scheduler marks the thread as blocked pending request completion and schedules something else. The problem is that one blocked thread is enough to block an entire process, even though other threads within the process might still have available work to do.

The aim of concurrent programming is to multiplex multiple tasks on to a single thread. From the scheduler's perspective, the thread will seem completely CPU-bound and so won't be suspended. There are a few things that need to be in place for this to work. The operating system will need to provide non-blocking versions of system calls. The program will need to somehow manage scheduling its own tasks. This is difficult to get right and so it is normal to rely on a library or a language-specific runtime that takes care of all the scheduling. When I refer to "scheduler" from now on, I mean this scheduler internal to the program, rather than the OS scheduler, unless I specify otherwise.

Determinacy and state

Concurrency seems pretty cool. We can write code that handles multiple things at once and we don't have to worry about anything blocking the whole program. What's not to like?

The problem is **non-determinacy**. In a synchronous program, we know the order in which tasks execute and so we can predict how they will affect the system. For example, function A writes a value to memory and then function B reads that value. The code generates the same output every time it is called. In a concurrent program, the executing tasks are arbitrarily interleaved by the scheduler. It is entirely possible that function B will run first and read the value before function A writes it, thus generating a different output. Worse, that will only happen sometimes: the scheduler can choose a different ordering every time. The upshot is that the program's output may vary depending on how its tasks have been scheduled. This is non-determinism. It is very difficult to reason about code and have assurances about its correctness in the presence of non-determinism. How can you reliably test code that breaks only rarely? Perhaps when you write and test the code, the scheduler consistently chooses an order that functions correctly. Then, weeks from now, for whatever reason, the scheduler chooses an ordering that breaks your code and everything blows up in production.

The root cause of the problem with functions A and B above is not so much the random interleaving but the fact that the two functions have **shared**, **mutable state**. I say "shared" because they both have read access to the same memory addresses and "mutable" because they both can modify the data in those addresses. If there were no shared, mutable state between the two functions, it wouldn't really matter in which order they were executed.

A classic example of the problems in shared, mutable state is a broken counter implemented using two threads (in Java-like pseudo-code):

```
function main() {
    Thread[] threads = new Thread[2];
    int count = 0;

    for (thread : threads) {
        thread.run() {
            for (int i = 0; i < 10000; i++) {
                count++;
            }
        }
    }

    System.out.println("Got count " + count + ", expected " + (2 * 10000));
}
```

In this code, we start two threads, each incrementing `count` 10,000 times. You might think that this is a faster way to count to 20,000 because the work is split between two threads. In fact, on a system

with multiple hardware threads or cores, the program doesn't even work properly and `count` will be lower than expected. Its exact value will vary each time the program runs, due to non-determinacy.

Each thread has to read `count`, increment its value and write the modification back to the variable. The problem is that both threads are trying to do that at the same time. Here's an example of how things will go wrong. Thread one starts and reads the initial value. Depending on the scheduler's mood, there is a chance that it will suspend thread one and schedule thread two. The second thread will run for a while, incrementing `count` by some amount. The scheduler then swaps the threads and thread one picks up from where it left off. Crucially, thread one will still hold the same value it had for `count` when it was first suspended. It will be completely unaware of the changes made by thread two and so its modifications will overwrite thread two's work. This happens repeatedly, leaving `count`'s final value far off the expected 20,000.

When multiple threads compete to update shared, mutable state, we have what is know as a **data race**. More generally, competition for access to any resource is known as a **race condition**. The observed outcome depends on which thread wins the "race", as determined by the order the scheduler chooses to run them in. This is the source of the observed non-determinism and a relatively funny joke (by computing joke standards):

> What do we want?
>
> - Now!
>
> When do we want it?
>
> - No race conditions!

Race conditions occur when two or more threads simultaneously execute a **critical section** of code. Only one thread may be in the critical section at any time. The critical section in the example above is `count++`, which is just a single line of code but actually involves reading the current value, incrementing it and then writing it back out. The way to achieve safe, concurrent code without race conditions and non-determinism is to control, or **synchronise**, access to critical sections to prevent multiple threads entering them at the same time. All of the concurrency models in this chapter solve this problem in different ways.

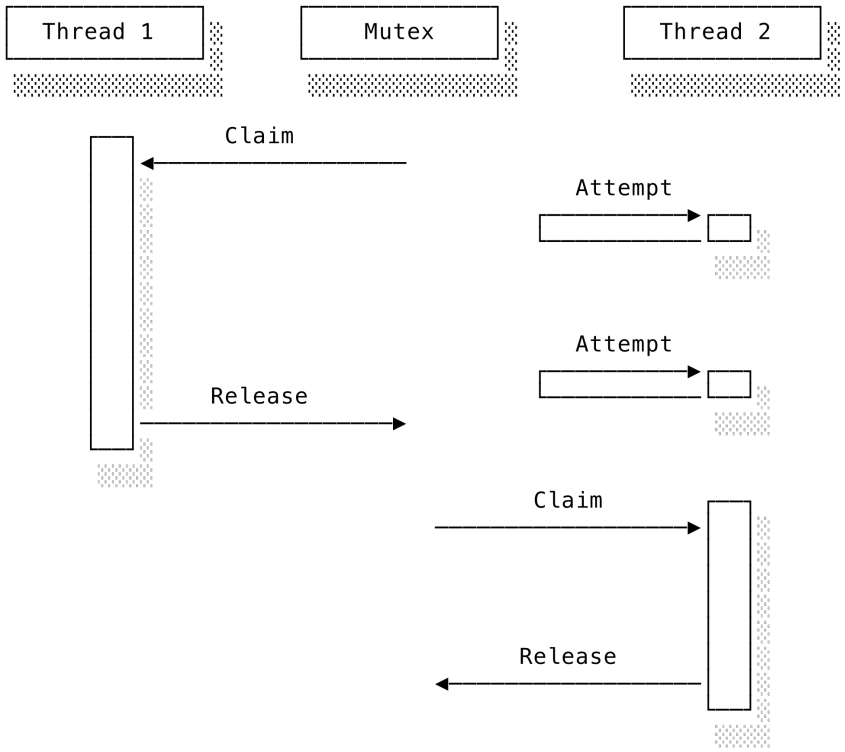
Threads and locks

The simplest concurrency approach a language can take is to only support the basic concurrency primitives provided by the underlying operating system. If the programmer wants to do anything fancy, they'll have to build it themselves (or find a library implementation). In Ruby, for example, a network request will block by default:

```ruby
response = Net::HTTP.get('example.com', '/index.html')
# execution waits until response is ready
```

If you want concurrency within a Ruby program, you need to create threads. Remember that threads operate within the same process's address space. If a program creates three threads, it will have a total of four (including the original process thread) independently executing threads, all jumping around the address space and changing things as they please. Since they share the same address space, a change made by one thread will be immediately visible to the others: shared, mutable state.

Operating systems provide a very simple synchronisation mechanism known as the **lock**. If you want to make something inaccessible, you lock it away. A computing lock is no different. A lock is a record that a particular thread has exclusive access to a particular resource. A thread *claims* a resource by recording that it is using the resource. It does its work and, when finished, *releases* the lock by erasing the record. A second thread coming along and trying to access the resource must wait for the lock to be released before it can claim the lock for itself. By requiring that a thread claim a particular lock before entering a critical section and releasing the lock when it leaves, we ensure that only one thread is ever in the critical section. Such locks are also known as **mutexes** (from "mutual exclusion").



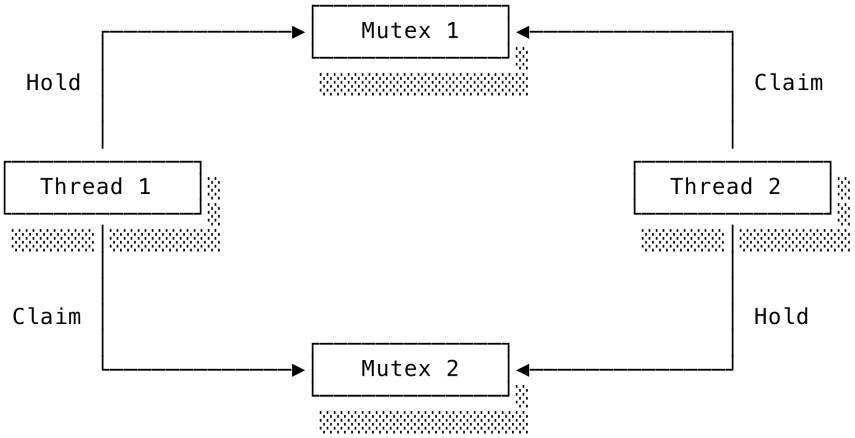
Two threads competing for a mutex

Note that thread two is blocked when it cannot claim the lock. Mutual exclusion means that accesses to the resource are now performed synchronously. The synchronisation solves our race condition problem but removes the concurrency, which was the whole purpose in the first place. Achieving the

optimal balance of correctness and concurrency via locks requires choosing the correct **granularity**. Ideally, we want to keep critical sections as small as possible while still preventing non-determinism.

More sophisticated locks allow for more precise access control. Threads needing only to read a value can each claim read-only, **shared locks**. Should another thread need to modify the value, it can claim an exclusive lock but only when there are no shared locks. Databases frequently use such mechanisms to coordinate concurrent reads and writes. A problem here is when there are so many reads that there is always at least one shared lock in place and so a modifying thread is **starved** and can never claim its lock.

Still worse, using locks on multiple resources can lead to a situation known as **deadlock**. This occurs when two threads have each claimed one lock and are waiting for the other thread to release the other lock. Both threads are stuck in an infinite loop.



Two threads in deadlock

To avoid deadlocks, locks must always be claimed and released in the same order. That sounds straightforward but is no simple task in a complex application. Many very competent developers have horrendous stories of days spent tracking down very subtle bugs caused by incorrect lock usage. I think part of the problem is that you have to read code in a very different way. Look at the example below (again in Java-like pseudo-code):

```
function transfer(Account sender, Account receiver, uint amount) {
  claimLock(sender) {
    claimLock(receiver) {
      sender.debit(amount);
      receiver.credit(amount);
    }
  }
}

Thread1.run() { transfer(a, b, 50); }
Thread2.run() { transfer(b, a, 10); }
```

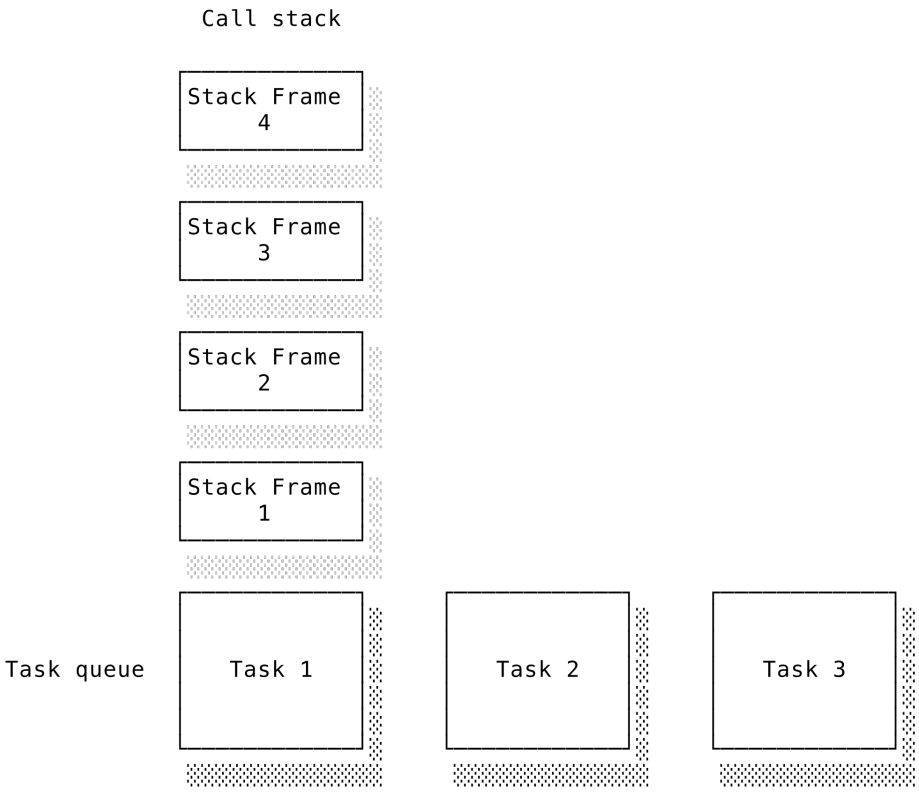
At first glance, this seems fine because we consistently claim first the sender lock and then the receiver lock before performing the transfer. When reading the code, however, try to imagine the various ways the two threads might be interleaved. One possible ordering is where `Thread1` claims a lock on `a` and then `Thread2` is scheduled and claims a lock on `b`, resulting in deadlock because neither thread can claim the second lock. In this example, it's fairly easy to spot how the lock orderings are reversed Now imagine that the two calls to `transfer` are in completely different areas of the codebase and only occasionally run at the same time. The potential for deadlock would not be at all obvious.

When reading code, it's natural to make a sequential, synchronous mental thread of execution. When dealing with multiple threads, you have to remember that between any two operations the thread may have been paused and another scheduled thread might have changed any of the shared, mutable state. A value that was `true` in the conditional of an `if` statement might have become `false` by the time the block of the statement is executed. This makes multi-threaded programming using locks deceptively difficult to do correctly. Operating systems and databases use locks extensively to implement very sophisticated concurrent systems, but they are notorious for having very difficult to understand sections of code. For mortal developers who don't write operating systems and databases on a regular basis, it is better to think of locks as the building blocks from which easier to use concurrency models are constructed. In the next two sections, we'll examine the concurrency models provided by JavaScript and Go.

JavaScript and the event loop

JavaScript was originally designed to run in web browsers. The browser is a perfect example of a highly concurrent application. At any one moment, it might have to parse HTML, dispatch events, respond to a network request, react to user input and more. Many of those tasks will require some JavaScript execution. For example, when the browser detects that the user has clicked on a DOM element, it must dispatch an event to any registered JavaScript event listeners. JavaScript execution is organised around a simple design pattern known as the **event loop**, which is frequently found in event-driven programs. In an event loop, a main thread continually checks for and handles incoming events or tasks. Other threads generate events to be processed by the main loop.

Every browser has a **JavaScript engine**, or runtime, which is in charge of running JavaScript code. In the case of Chrome, the engine is called V8 (also used by node.js). Whenever some JavaScript needs to be executed, it is bundled into a task. The engine maintains a queue of pending tasks known as the **task queue**. The engine runs a loop, continually pulling tasks from the head of the task queue and passing them to the JavaScript thread for execution. A separate structure, known as the **call stack**, tracks the execution of the current task. There is only one thread consuming the queue, so there is only one call stack and the engine can only execute one JavaScript task at once. The call stack operates just like the process's stack managed by the OS. The engine pushes and pops stack frames to and from the call stack as the JavaScript thread calls and returns from functions in the task. When the call stack is empty, the task is complete and the engine moves on to the next task.



The task queue and call stack

The engine uses only a single JavaScript thread to execute the tasks sequentially. This single-threaded form of concurrency avoids all of the nastiness of shared, mutable state. Programming in JavaScript is therefore significantly easier than programming in a multi-threaded environment, although getting used to callbacks and asynchronous code does take some effort.

You might be wondering at this point: "how can JavaScript be concurrent and non-blocking if it only has a single thread?". While there is only one thread executing JavaScript, the engine will have other, background threads performing network requests, parsing HTML parsing, responding to user input and so on. The JavaScript execution model relies heavily on the multi-threaded engine to handle all of the tricky, asynchronous stuff by processing operations concurrently and maintaining the task queue.

Asynchronous, non-blocking I/O is simple to implement in this model. When JavaScript triggers an I/O operation, it provides the engine with a callback function to be executed when the operation completes. The engine uses a background thread to request the operation from the OS. While that thread waits for the OS to provide the response, the JavaScript thread continues on its merry way working through the task queue. When the OS completes the I/O request, it wakes the requesting thread. That thread creates and pushes to the queue a new task in which the callback receives the I/O response as an argument. Eventually, that task reaches the head of the queue and is executed.

The downside is that the programmer has very limited control over which tasks are executed in which order. There is no way to suspend the correct task or reorder pending tasks. Since there's only

a single JavaScript thread, a computationally intensive task that takes a long time to complete will block subsequent tasks. You might see JavaScript mistakenly described as non-blocking in general. That is incorrect. It only offers non-blocking I/O. A node.js server, for instance, can easily become unresponsive by performing a computationally intensive task in a request handler. Since browsers usually do rendering updates after each task completes, slow tasks can even freeze web pages by blocking UI updates. Just try running the following in your browser console:

```
function blocker() {
  while(true) {
    console.log("hello")
  }
}
blocker()
```

This loop runs synchronously and blocks the event loop. The current task only finishes when the call stack is empty, but this script will never have an empty call stack because it contains an infinite loop. Eventually, Firefox has had enough and asks me to terminate the script. Thus, the first rule of writing performant JavaScript is: *don't block the event loop.*

Though simpler than multi-threaded code, JavaScript's asynchronous style still takes some getting used to. It can be hard to work out in what order things will happen if you aren't familiar with the event loop, but there are actually only a few simple rules. For example:

```
function example() {
  console.log("1");

  window.setTimeout(() => console.log("2)", 0);

  console.log("3");
}

example();
```

What do you think the output of this example will be? The JavaScript engine starts off executing the top level in the first task. It defines the `example` function and executes it at line nine, creating a stack frame in the process. JavaScript requests that the engine perform the `console.log("1")` operation, which it does synchronously (without enqueuing a task) by outputting `1` to the console. We are still within the first task. Next, JavaScript requests that the engine starts a timer for `0` milliseconds and provides it a callback for when the timer is ready. The timer ends immediately and the runtime enqueues a new task with the callback. However, the first task has not finished yet! While the runtime is dealing with the timer, the JavaScript thread makes the second `console.log` request. It then returns from `example`, removing its stack frame. The call stack is now empty and the first task is complete. Assuming there are no other tasks, the JavaScript thread executes the callback task and requests the final `console.log`. The output is therefore:

```
1
3
2
```

Using `window.setTimeout(callback, delay)` with no delay value (or zero) doesn't execute the callback immediately, as you might expect. It tells the runtime to *immediately enqueue* the callback task. Any tasks already queued will be executed before the timeout callback. Since JavaScript can only ever see the head of the queue, we have no idea how many tasks are already queued or how long it'll take the callback task to reach the head of the queue. This makes JavaScript a real pain to use for anything requiring very precise timing.

The picture is further complicated by **micro-tasks** (also known as *jobs*). These slippery little creatures are like high priority tasks. They go in a separate queue that is processed after callbacks and between tasks. Promises and their handlers, which we'll look at in more detail below, are good examples of micro-tasks. In this example, what do you think the output will be?

```
function example() {
  console.log("1");

  window.setTimeout(() => console.log("2"));
  Promise.resolve().then(() => console.log("promise"));

  console.log("3");
}

example();
```

We know that specifying no timeout value for `window.setTimeout` enqueues the callback. The promise callback is also enqueued, but because promise handlers are micro-tasks, it wriggles in ahead of the timeout callback:

```
1
3
promise
2
```

Treating micro-tasks in this way improves the responsiveness of the system. The whole point of promise callbacks is that they are executed as soon as possible after the promise resolves, so it makes sense to give them priority. Sending them to the back of the task queue could cause an unacceptably long delay.

To sum up, JavaScript's concurrency model provides single-threaded, asynchronous execution with non-blocking I/O. It relies heavily on the JavaScript engine to process operations concurrently and

enqueue tasks. The JavaScript thread only sees a sequence of discrete tasks with no shared, mutable state. The programmer doesn't have to worry about data races or deadlocks, but in exchange they give up a lot of control. JavaScript is not particularly well-suited to computationally intensive work because slow running tasks block the event loop until they are complete. It is better suited to I/O-bound work, such as servers. A node.js server can easily handle thousands of concurrent requests in a single thread, all thanks to the event loop.

Managing asynchronous execution

Now that we understand how the underlying runtime works, let's look at how a JavaScript programmer can structure their asynchronous code. The most simple approach, as already seen above, is to provide a function that the runtime should use to *call back* into JavaScript when the operation completes. Callbacks appear throughout the browser APIs (e.g. `setTimeout`, `addEventListener`) but became ubiquitous in node.js. The thing to remember about callbacks is that the callback function executes later, when the response is ready. Performing a sequence of asynchronous operations therefore requires nested callbacks, affectionately known as **callback hell**:

```
function handleRequest(req, callback) {
  validateRequest(req, (error, params) => {
    if (error) return callback(error);
    fetchFromDB(params, (error, results) => {
      if (error) return callback(error);
      generateResponse(results, callback);
    });
  });
}
handleRequest(req, (error, response) => {
  if (error) {
    console.error(error);
  }
  console.log(response);
});
```

There are various techniques to make it look a little better, but you can't get away from the fundamental weirdness that the program's control flows through the callbacks. The nature of asynchronous code is that a variable's value depends on *when* the current scope is executing. An operation's result only exists in the scope of its callback. That makes sense because, as we've just seen, the callback task is only created when the operation completes. We must be careful to structure our code so that tasks needing the result are only scheduled when the result is available. That's why, in the example above, we have to call `fetchFromDB` from `validateRequest`'s callback: it's only in that scope that the necessary `params` exists.

Enter the **promise**. A promise (also known as a *future* in other languages) is a value representing some asynchronous operation. While the operation is pending, the promise has an ambiguous value.

At some future point, when the operation completes, the promise's value *resolves* into the result of the computation (or an error in case of failure). Since promises are values just like any other JavaScript value, they can be stored in data structures and passed to and from functions. We can register handlers to execute when the promise resolves by calling `.then()`, which returns the same promise to allow for chaining:

```
function handleRequest(req) {
  return validateRequest(req)
    .then(fetchFromDb)
    .then(generateResponse);
}

handleRequest(req)
    .then(response => console.log(response)
    .catch(err => console.error(err));
```

Promises introduce the concept of an asynchronous value. Notice that `handleRequest` above returns a promise. That's more meaningful than returning `undefined` and having to pass in a callback to access the result. Yet promises are no panacea. Asynchronous operations have to be sequenced as a chain of `.then()` calls, which are subjectively ugly and objectively very different to standard, synchronous code. Worse, if an error occurs in a promise it is *rejected*, requiring the `.catch()` handler at the bottom of the chain. This is different to the normal `try / catch` error handling in JavaScript and can create subtle bugs if you don't handle everything properly. We still have to jump through all these syntactic hoops to get things working as we want. Wouldn't it be better if we could write asynchronous code in the same way we write synchronous code?

So, just after everyone rewrote their callback-based code to use promises (which arrived officially with ES6 JavaScript), the JavaScript community decided that promises were unacceptable after all. A new challenger arrived with ES2017: **async/await**. The idea is that we tag asynchronous functions with the `async` keyword. We call them and assign their return values as normal on the understanding that the return value may not actually exist until later. Under the hood, async functions actually return promises. The `await` keyword creates a **synchronisation point** by which the promise must be resolved. If the promise has not yet resolved, the task will suspend until the value is available. We can write asynchronous code that looks almost like synchronous code. The main difference is that we must remember to `await` the return value when we want the resolved value:

```
async function handleRequest(req) {
  const params = await validateRequest(req);
  const dbResult = await fetchFromDb(params);
  const response = await generateResponse(dbResult);
  console.log(response);
  return response;
}
```

It's much easier to see here the sequence of actions than in either the callback or promise examples. Since async/await are syntactic sugar around promises, we can use them directly on promises too. That might help clarify what they are doing. The following async function awaits a promise that resolves after a timeout in order to emulate a slow, asynchronous operation (e.g. a database query):

```
async function slowFetchId() {
  await new Promise(resolve => setTimeout(resolve, 5000));
  return 5
}
```

Here we call `slowFetchId` and assign its return value to `id`. We log what it contains before and after we await the result:

```
async function example() {
  var id = slowFetchId()
  console.log("first id:", id)
  await id
  console.log("second id:", id)
}

// Output
> first id: Promise {<pending>}
// Five seconds later...
> second id: Promise {<resolved>: 5}
```

Since `slowFetchId` is async, it immediately returns a pending promise. Note that the returned promise represents the whole function `example` and so it actually contains the separate promise defined within `example`. When we want the actual value from the promise, we use `await` to suspend the current task execution until `id` has *settled* by resolving into a value or rejecting with an error. `slowFetchId` is itself waiting on the timeout promise, so it can't do anything until the runtime triggers the timeout callback, which enqueues a micro-task to resolve the promise and allows `slowFetchId` to return `5`. The promise held in `id` can now resolve and execution moves on to the final logging statement.

The main limitation of async/await is that any function calling an async function also needs to be marked as async. Making just one function async might cause a whole cascade of tedious refactoring.

You might see this referred to as having to "colour" a function. The terminology comes from an influential blog post on the topic, which I have included in the further reading. While not perfect, async/await is a big improvement. There was a dark period when callback hell coexisted with various, mutually incompatible promise libraries. Best practices seemed to change from day to day. Thanks to the combination of built-in promises and async/await, things are now settling down. Using promises, we can express the concept of an asynchronous value and with async/await, we can express asynchronous operations in a clean and consistent manner.

Communicating sequential processes in Go

You might expect Ryan Dahl, the creator of node.js, to be bullish about it as a server platform. In fact, he says[33]:

> I think Node is not the best system to build a massive server web. I would use Go for that. And honestly, that's the reason why I left Node. It was the realisation that: oh, actually, this is not the best server-side system ever.

What does Go offer that node.js doesn't? Recall that JavaScript (and hence node.js) uses a single-threaded, asynchronous concurrency model. That doesn't scale to multiple cores. What if we assigned each task to its own thread and relied on the operating system to schedule them on the available cores? **Multi-threading**, as it is known, is certainly a valid approach. The problem is that context switching between threads requires switching into kernel mode and flushing caches. The operating system doesn't know how much state can be safely shared between threads so it has to be conservative and clear everything. The performance overhead will be too much for many use cases. If the program implemented its own version of threads in userspace, it could avoid the expensive context switches into kernel mode and reduce the cost of switching.

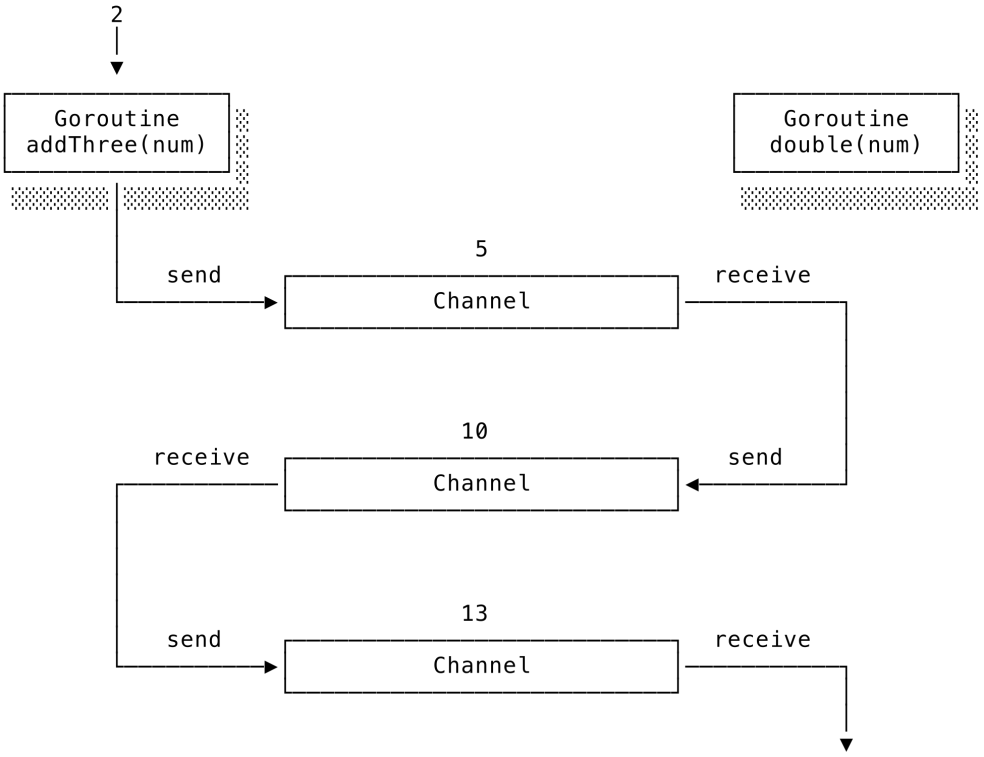
Concurrency in Go is implemented via **goroutines**: lightweight, userspace threads created and managed by the Go runtime. A single Go process might contain thousands or even millions of goroutines. Switching between goroutines is much faster than switching between OS threads and is not much slower than a simple function call. To achieve concurrency, a Go programmer simply has to create a goroutine for each concurrent task. If there is suitable hardware support, the Go runtime will even execute them in parallel with no additional effort required from the programmer.

We need some way for the goroutines to share data safely. Modern JavaScript uses promises to represent a value that might change in the future. This mutable state is not shared because JavaScript is single-threaded. Go takes an entirely different approach. We can create communication pipes between goroutines known as **channels**. Rather than communicating by sharing mutable state, goroutines share data by communicating through channels. They are first class objects in Go, meaning that we can store them in variables and pass them to functions just like any other value.

By default, channels are unbuffered, meaning that they can only hold a single value. Reading from an empty channel and writing to a full channel will both block the current goroutine. That gives

[33] https://mappingthejourney.com/single-post/2017/08/31/episode-8-interview-with-ryan-dahl-creator-of-nodejs/

another goroutine an opportunity to fill or empty the channel as appropriate. Channel operations therefore naturally create synchronisation points in the code. The **select block** is a bit of syntax that allows a goroutine to wait on multiple channel operations. In Go, we use channels and select blocks to express the logical dependencies between concurrent goroutines and leave the scheduler to work out the details.



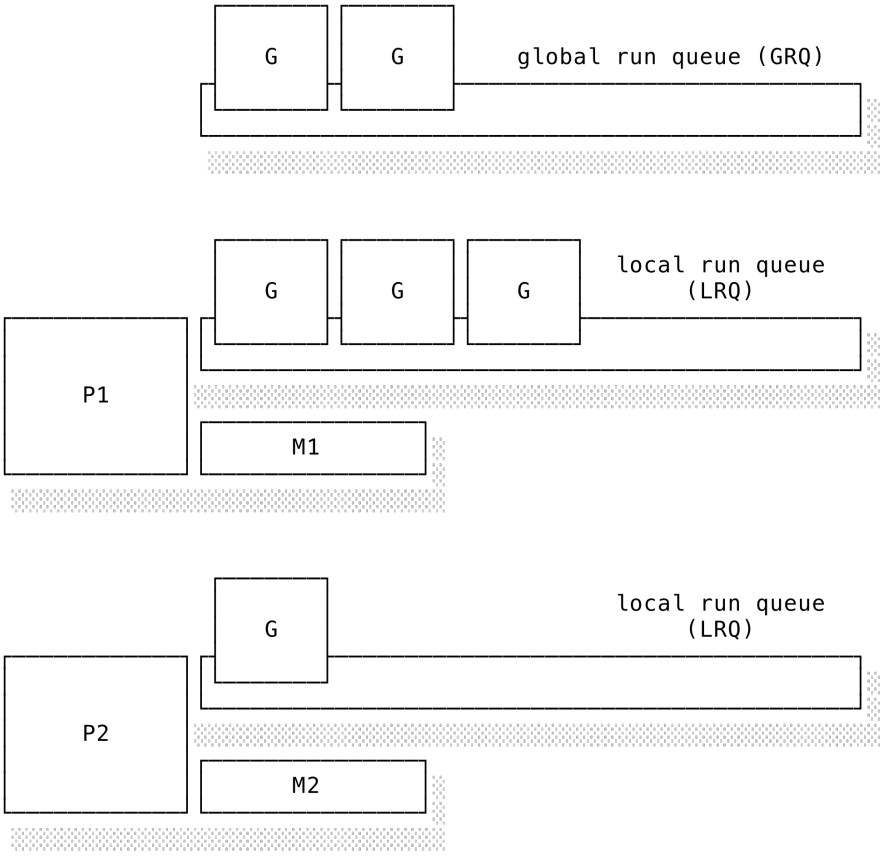
An example of two goroutines synchronising and sharing data

None of these concepts are particularly unique or innovative. The strength of Go's concurrency model comes from being a simple, well-engineered combination of all three. The theoretical underpinning is known as **communicating sequential processes** (CSP) and was first developed by C.A.R. Hoare back in 1978 (see further reading).

Scheduling goroutines

Implementing its own form of userspace threads means that the Go runtime has to implement its own userspace scheduler. The OS scheduler plays no part in scheduling goroutines. It only sees the threads of the Go runtime itself. It's up to the Go runtime to efficiently schedule concurrent goroutines. The Go runtime scheduler tries to avoid the OS scheduler suspending its threads by schedule goroutines so that those threads are continually busy. If the runtime's threads run out of work or become blocked, the OS scheduler might suspend the entire Go runtime process, halting all goroutines. The OS scheduler can't see that some other goroutines might have been able to continue doing useful work.

Let's look at the Go scheduler in more detail. The scheduler creates a *logical processor* (P) for each virtual core (that is, counting each hardware thread in each physical core). Each P is assigned a physical thread called a *machine* (M). The OS schedules the M threads according to its own rules. Goroutines (G) are created when the program starts and by using the special `go` keyword. Each goroutine is scheduled on to an M. Each P has a *local run queue* (LRQ) of goroutines assigned to it. The logical processor P runs the goroutines from the LRQ on the machine M. There is a separate **global run queue** (GRQ) for goroutines that aren't yet allocated a logical processor. It all looks like this:

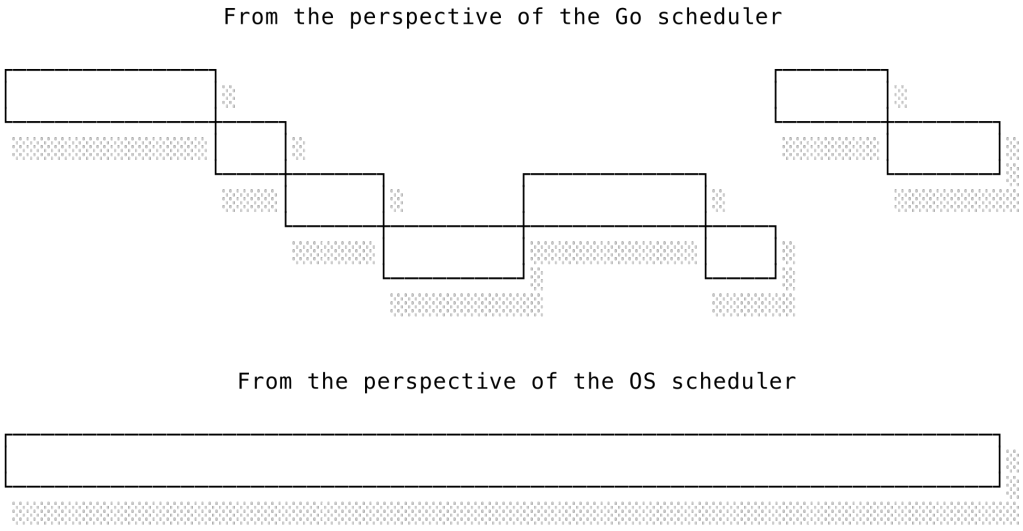


Two logical processors with LRQs and the GRQ

Goroutines can be in one of three states with fairly self-explanatory names: *waiting* when it needs some resource in order to continue, *runnable* when it's ready to go and *executing* when it's actually running on a machine. The scheduler is **cooperative** because goroutines voluntarily relinquish the machine at defined points, such as system calls and when the `go` keyword is used. The scheduler doesn't pre-emptively suspend goroutines as the OS scheduler would do with processes. The Go scheduler implements a **work-stealing** algorithm: when a logical processor empties its LRQ, it will look for pending work in the global queue or steal pending goroutines from another logical processor. It's like an annoyingly high achieving colleague who finishes all their work and starts doing some of your pending tickets to fill their time.

The justification for userspace threads is that switching between them is much faster than switching

between OS threads. The Go scheduler takes a set of CPU-bound and I/O-bound tasks and carefully schedules them so that they appear to be one, unbroken, CPU-bound task to the OS scheduler. If the Go runtime is limited to only use a single physical core, things work in a similar way to JavaScript. But unlike JavaScript, Go can easily take advantage of as many cores as the hardware has available. True parallelism is thus a possibility. Running multiple userspace threads over a single OS thread is known as **M:N threading**, where N goroutines are spread over N system threads.



Multiplexing I/O bound tasks on to one system thread

A disadvantage of goroutines, and userspace threading in general, is that each goroutine requires at least a few kilobytes of memory to store its stack. This is normally inconsequential but something to bear in mind if your application is going to create huge numbers of goroutines. The Go runtime starts each goroutine off with 2Kb of stack space. If the goroutine calls lots of functions and is in danger of running out of stack space, the runtime handles this by allocating a bigger block of memory and copying the entire stack over to the new location. It's a bit like how dynamic arrays automatically reallocate when they reach their capacity. Moving the stack entails rewriting the address of every variable on the stack. Go is only able to do this because it tracks every use of a variable's address and so can rewrite the addresses as necessary.

Working with goroutines

We can rely on the blocking nature of channel operations to automatically create synchronisation points in our code. The runtime will detect when a goroutine is blocked and switch to another runnable goroutine. Eventually, assuming no deadlocks, a goroutine will run that unblocks the first and progress continues. We can write our code in a straightforward, declarative style without worrying about sequencing operations in the correct order. The scheduler takes care of that. There's therefore very little distinction between synchronous and asynchronous Go code. A direct function call occurs synchronously in the caller's goroutine. Calling a function with the `go` keyword creates a new, asynchronous goroutine.

Let's look at how to use channels to synchronise goroutines. In this example, we offload a slow computation to a separate goroutine:

```
package main

import (
        "fmt"
        "time"
)

func slowCompute(a, b int) int {
        time.Sleep(2 * time.Second)
        result := a * b
        fmt.Printf("Calculated result: %d\n", result)
        return result
}

func main() {
        go slowCompute(5, 10)
        fmt.Println("Doing other work...")
}
```

This doesn't work as intended:

```
Doing other work...
```

The program doesn't wait for `slowCompute` before exiting! The problem is that the `main` goroutine has not been instructed to wait for `slowCompute`. What are we missing? Synchronisation! We fix this by having `slowCompute` write to a channel and `main` read from it. As we know, a read operation on an empty channel will block until a value is sent:

```
package main

import (
        "fmt"
        "time"
)

func slowCompute(a, b int, result chan<- int) {
        time.Sleep(2 * time.Second)
        result <- a * b
}
```

```
12
13 func main() {
14         result := make(chan int)
15         go slowCompute(5, 10, result)
16         fmt.Println("Doing other work...")
17         fmt.Printf("Calculated result: %d\n", <-result)
18 }
```

The `<-result` syntax means "read from `result`" and `result <-` means "write to `result`". This works as expected:

```
1 Doing other work...
2 Calculated result: 50 // two seconds later
```

Channels just seem to be a roundabout way of implementing JavaScript's async/await, right? Not so! Channels permit more complex patterns. It is trivial to implement task queues, worker pools and other concurrent patterns using channels. Here's an example of distributing work over a pool of worker goroutines:

```
1  import (
2          "fmt"
3          "sync"
4          "time"
5  )
6
7  func worker(id int, jobs <-chan int, results chan<- int, wg *sync.WaitGroup) {
8          fmt.Printf("Starting worker %d\n", id)
9          for job := range jobs {
10                 fmt.Printf("Worker %d performing job %d\n", id, job)
11                 time.Sleep(2 * time.Millisecond)
12                 results <- job * 2
13         }
14         wg.Done()
15 }
16 func main() {
17         var wg sync.WaitGroup
18         jobs := make(chan int, 100)
19         results := make(chan int, 100)
20
21         for w := 1; w <= 4; w++ {
22                 wg.Add(1)
23                 go worker(w, jobs, results, &wg)
```

```
24          }
25
26          for jobID := 1; jobID <= 20; jobID++ {
27                  jobs <- jobID
28          }
29          close(jobs)
30
31          wg.Wait()
32          close(results)
33  }
```

There's lots of interesting things going on here! We create two channels: `jobs` and `results`. Both have a buffer size of `100`. We start up four workers in separate goroutines. We then write twenty jobs to the `jobs` channel. The workers pull jobs from the channel using the `range` keyword. The workers sleep briefly during processing to simulate performing a time consuming task. Once we've put our jobs into the channel, we close it to indicate that no more values will be written. That stops the workers ranging infinitely over the channel, forever waiting for values that never arrive. A `waitGroup` is a counter that's incremented every time we call `Add(1)` and decremented every time we call `Done()`. `wg.Wait()` blocks the `main` goroutine until the counter is zero. In this case, that means that every worker has finished and all of the work is complete. The output will look something like this, though the exact ordering is non-deterministic and depends on how the scheduler chooses to run things:

```
1  Starting worker 4
2  Worker 4 performing job 1
3  Starting worker 1
4  Worker 1 performing job 2
5  Starting worker 2
6  Worker 2 performing job 3
7  Starting worker 3
8  Worker 3 performing job 4
9  Worker 1 performing job 5
10 Worker 3 performing job 6
11 // ...
```

Go makes it easier to sequence concurrent operations and to reason about how control flows through a concurrent application. Nevertheless, writing correct concurrent code still requires some care. Here is a trivial example that deadlocks:

```
package main

import "fmt"

func work(channel chan int) {
        channel <- 1
}

func main() {
        channel := make(chan int)
        go work(channel)
        select {
        case value := <-channel:
                fmt.Printf("%d\n", value)
        }
        channel <- 2
        close(channel)
}
```

We start off a worker in a separate goroutine and then block until a value is available using `select`. That will work. Trying to send the second value to `channel` is what causes the deadlock. Since `channel` is unbuffered, the write will block the main goroutine until something reads it. That gives the scheduler a chance to run other goroutines and hopefully one of them will read from the channel. However, in this instance there is no other goroutine trying to read from the channel and so the program blocks forever – deadlock! Admittedly, the cause of the deadlock in this contrived example is pretty obvious but it's surprisingly easy to cause subtle deadlocks in more complex code. On the plus side, the Go compiler includes a deadlock detector that can often find deadlocks and even show the faulty line of code.

Conclusion

We live in interesting, concurrent times. In this chapter, we saw that changes in computer architecture are driving changes in how we write software. Programs need to be able to take advantage of multiple processor cores. Concurrent programming involves writing code that can deal with multiple things at once. Parallelism means literally performing multiple operations at once and requires hardware support. Asynchronous programming means writing code that does not block while waiting for operations to complete. All of them require thinking about programming in a different way.

Multiple threads running in the same address space create the problem of shared, mutable state leading to non-determinacy. One simple solution, the lock, is deceptively difficult to use correctly and safely. Concurrent programming models provide mechanisms to manage shared, mutable state

that are easier to reason about. JavaScript's single-threaded, asynchronous model relies on the language runtime to provide concurrency, greatly simplifying things for the programmer. The event loop approach is simple to understand but easy to block and has no scope for parallelism. JavaScript has struggled to provide effective means to coordinate asynchronous operations, though async/await is a big step forward. Go implements userspace threading in the form of goroutines. It avoids data races by communicating state between goroutines via channels. Go programs scale easily to multiple cores with no changes in syntax. The downside is the implementation cost of a userspace scheduler, the cumulative memory cost of many goroutines and the risk of deadlocks.

Further reading

Many of the textbooks cited in the operating systems chapter include thorough treatments of the concurrency primitives provided by operating systems. If you haven't already read *Operating Systems: Three Easy Pieces*, now would be a good time to start. Herb Sutter's article The Free Lunch is Over: A Fundamental Turn Toward Concurrency in Software[34] is a little dated now but provides some useful background context on the hardware changes driving developments in concurrent programming.

When those textbooks warn gravely of how threads and locks lead to dreadful concurrency bugs, it's sometimes difficult to grasp just how subtle the bugs can be. Deadlock Empire[35] is a fun, little game that challenges you to find and exploit concurrency bugs. You must slay an evil dragon that, fortunately for us, uses locking mechanisms to protect itself. The simple fact that finding concurrency bugs in lock-based code can be made into a mind-bending puzzle game illustrates just how hard it is to write correct code with locks.

If you would like to know more about the basis for Go's concurrency model, I recommend the original paper: *Communicating Sequential Processes* by C.A.R. Hoare. It's easy to find online and you can even find the problems from the paper solved in Go in this package[36].

Seven Concurrency Models in Seven Weeks by Paul Butcher provides a pacey introduction to a range of concurrency models implemented in a few languages (including Java, Erlang and Clojure). I recommend it both as a good overview of the problem space and as a window into some environments you might not yet have come across (particularly Erlang).

At the risk of stating the obvious, the best thing you can do to improve your understanding of concurrent programming is to write code using different concurrency models. If you know JavaScript, you're already writing concurrent code, but have you tried the full range of tools at your disposal? Try implementing the same functionality using callbacks, promises and async/await. Which is easier and most ergonomic? What problems do you encounter? (What colour is your function?[37] highlights some of the issues with async/await.) Learning Go, Erlang, Elixir or any language with a distinctive concurrency model is a powerful way to make yourself rethink how programs are executed and to develop a more sophisticated mental model.

[34] http://www.gotw.ca/publications/concurrency-ddj.htm
[35] https://deadlockempire.github.io/
[36] https://godoc.org/github.com/thomas11/csp
[37] https://journal.stuffwithstuff.com/2015/02/01/what-color-is-your-function/

Distributed systems

Introduction

A distributed system is a collection of computers that present themselves to the user as a single, coherent system. The components of the system are spread over multiple computers that communicate over a network. The benefit of distributed systems is that they allow us to achieve levels of reliability and performance that are unobtainable on a single machine. The problem is that it is very difficult to design and run distributed systems correctly. They introduce whole new classes of sometimes very subtle bugs that can make it hard to reason about how the system operates. Leslie Lamport, a computer scientist renowned for his work in distributed computing, wryly describes them thus[38]: "A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable."

By their very nature, distributed systems force us to think about the physical limitations of time and space. **Latency** is the time it takes for a result to become apparent to an observer. When you click a hyperlink, it takes some time for your request to reach the server. During that period of latency, the request has been initiated but it is not yet visible to the server. Communication, even at the speed of light, can never be completely instantaneous and so some latency is inevitable. We know from the architecture chapter that latency within a single machine needs to be handled carefully but for the most part we can leave it to the computer architects to worry about. In a distributed system, spread over multiple machines possibly at great physical distances from each other, the effects of latency are too obvious to ignore. Much of the challenge of distributed systems comes from understanding and managing the effects of latency. As we'll see, the more a distributed system tries to behave like a single-node system, the greater the latency cost.

Distributed systems are increasingly common and some familiarity with them is essential for web developers. Simply put, all web work involves distributed systems to some degree. A web application running in the client's browser and communicating with a back end service forms a distributed system. Front end developers need to understand the problems latency introduces and how to mitigate them. The back end service itself is probably made up of at least an application server and a separate database server. Even that simple design is a distributed system. A production-ready architecture will probably have multiple services, load balancers, caches and who knows what else.

In this chapter we'll first review why distributed systems are so useful. We'll see what kinds of problems they can solve and the new challenges they bring. We'll then build a simple conceptual model of a distributed system and use that to determine the theoretical limitations of distributed computing. We'll then explore a few distributed system architectures via a discussion of consistent replication. Maintaining a consistent view of state across the whole system is essential for many use

[38] https://www.microsoft.com/en-us/research/publication/distribution/

cases and so understanding the challenges of consistent replication is a good entry point to thinking about distributed systems.

Why we need distributed systems

Why do we now find ourselves surrounded by distributed systems? In large part it's because we users have become so demanding. We used to be content with a PC sat in the corner by itself. Then we got the Internet and wanted to be able to talk to other computers. Fast forward a few more years and now we have multiple, mobile computing devices and we expect all of them to remain in sync all of the time. What we want to do online has become more demanding too. Back in the 1990s, web services could get away with having regularly scheduled downtime to perform updates. An unexpected outage would be irritating but not critical for many businesses. In our modern, globalised world, we expect online services to be always available at any time. We want to be able to search the entire Internet in seconds and download any content rapidly whenever we feel like it. The sheer scale of modern demands makes distributed systems essential. Below are the three reasons distributed systems help.

1. More machines can handle more

Tasks are easier when they can be handled by one machine. Sorting an input is a relatively straightforward job when the input fits in memory. Running a web server is easier if a single machine can handle all of the requests. When the problem becomes too big for a single machine, we need to change our approach.

Scalability is the ability of a system to cope with increased demand. We'll take "demand" to have a fairly loose meaning of "the work to be done". That might refer to a single, large job, such as indexing the web, but it could also mean a large number of small jobs such as requests to a web server. The first solution is to **scale up**: get bigger and better hardware so that one machine can handle more. This works very well but only up to a certain point. If you're trying to index the entire web then no amount of scaling up will do. No single machine, no matter how beefy, can handle the task. The second solution is to **scale out**: add more machines to the system. Now you have multiple machines coordinating to solve a problem. Congratulations – you have a distributed system!

Distributed systems can handle huge workloads by dividing them into smaller tasks and spreading those tasks across multiple machines known as **nodes**. They can cope with increased demand by scaling out. This makes them more flexible and probably more cost effective. For example, online retailers predictably experience huge spikes in demand at certain points in the year. The spikes require a high server capacity to handle the load. It would be wasteful to provision the same capacity throughout the rest of the year when demand is predictably much lower. This is the problem with scaling up. A server powerful enough to handle the huge, albeit brief, spikes would spend most of its time working at a small fraction of its full capacity. That's a waste of resources. If the retailer chose to scale out instead, they could provision only the capacity they needed and easily add extra machines when required.

2. More machines improve performance

We know from the concurrency chapter that some problems are parallelisable: they can be solved more quickly by having multiple machines solve parts of the problem in parallel. So for tasks that are easily parallelisable, some form of distributed system will deliver a performance boost.

A different performance benefit comes from globally distributed systems. Remember that communication is never instantaneous and transmission delay has a lower bound determined by the distance between the two nodes. By hosting the same data in multiple data centres dispersed around the globe, a system can serve the user's requests from the physically nearest data centre. Shorter distances result in lower latency and faster response times. Big content providers such as Netflix do this to make their services appear faster and more responsive. Having the same data in multiple locations introduces the challenge of maintaining consistency across all of those data sources.

3. More machines increase resilience and fault tolerance

Global web services don't have the luxury of regular maintenance downtime. They are in constant use. For many online businesses, a brief outage could cost millions in lost revenue. Even non-tech companies would be hobbled if their internal systems became unavailable.

We want our systems to be available for use and functioning correctly as much as possible. A **fault** is when one component of the system (e.g. a node's hard drive) stops working correctly. Faults are sadly inevitable. Software contains bugs and hardware eventually breaks. What we can control is how we respond to faults. **Resilience** means that the system can recover from faults. The user may observe some sign of failure but the rest of the system continues to work. Resilience itself doesn't require a distributed system. A simple example is a web application running on a single server. It might return an error response to one request but continue accepting and correctly processing other requests.

Distributed systems enter the scene when we aim for **high availability**. It's an imprecisely defined term but simply means that the service tries to have very little downtime (i.e. periods of unavailability). Normally this involves having redundant components arranged in a distributed system so that there is no single point of failure. **Fault tolerance** is an even more stringent requirement. It means that the system can continue correct operation even if some components fail. The user doesn't see *any* sign of failure, apart from maybe some increased latency. This obviously requires some form of distributed system, since a single node cannot provide fault tolerance.

The problem for designers of distributed system is that more hardware means more complexity and more places for things to go wrong. Most hardware failures are relatively unlikely for any given machine, but when you are running thousands of machines they become almost inevitable. Stats from Google[39] indicated that a new Google cluster might experience thousands of individual machine failures in a single year. Worse, entirely new classes of errors arise due to the interactions of so many components. Faults can cascade across the system in unexpected ways. We cannot achieve

[39] http://www.cs.cornell.edu/projects/ladis2009/talks/dean-keynote-ladis2009.pdf

fault tolerance simply by throwing more hardware at the problem. True fault tolerance is difficult to achieve and requires a lot of careful planning.

A theoretical model

In this section we will define a basic, theoretical model of a distributed system. We'll use it to explore the distinguishing characteristics and limitations of distributed systems in general. This is analogous to how we used automata to analyse computation way back in the theory of computation chapter. Our model takes the form of a set of assumptions about the system's environment and components. The assumptions are deliberately very simple to make analysis easier.

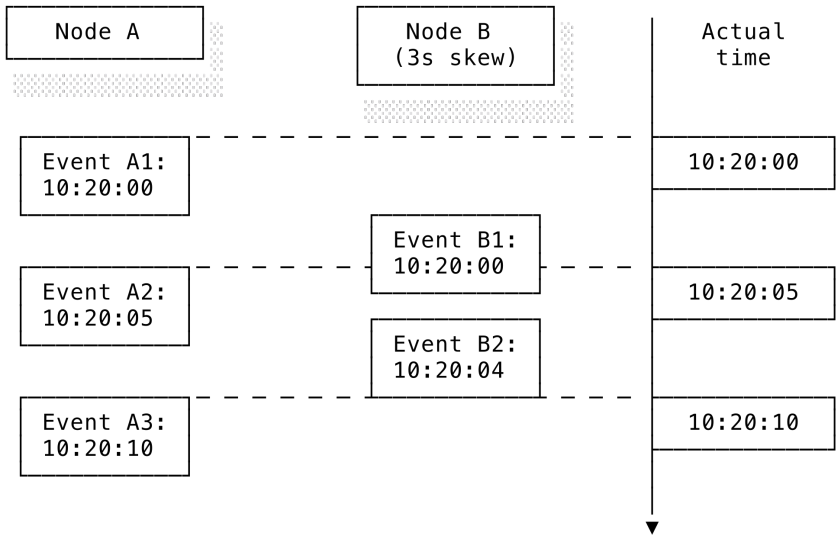
We assume that a single program is executing concurrently on multiple, independent nodes. Each node has its own local, volatile memory and persistent storage. It also has its own internal clock. The program running on each node allows the user to read and modify a single, system-wide variable. This is known as a "register" after a processor register. "Independent" here means that each node can fail and reboot without affecting the others.

Nodes communicate by passing messages over a network. Messages may be delayed or lost. There is no upper bound on transmission delay. The network therefore introduces non-determinacy into the system because we cannot guarantee the order in which messages arrive. We can model the network by giving each node a first-in-first-out (FIFO) message queue. "Sending" a message to a node is equivalent to adding the message to the node's message queue with no guarantee when or whether the message will be successfully added. The structure of the network is not important as long as every node can communicate with every other node.

We need to consider what kind of faults can occur in this system. We have already assumed that the network may arbitrarily delay a message. We will also assume that a node fault means that the process crashes and the node does not respond to requests. This, surprisingly, is the easy assumption. Far more insidious are **partial faults** where a node fails in some way but still responds to requests. It might return erroneous data, delay for an inordinately long time or appear to have correctly accepted a request but actually failed silently. A fault model that allows for components to fail with imprecise information about their failure is known as a **Byzantine fault model**, after the Byzantine Generals thought experiment from the networking chapter. Recall that neither general knows whether their messages were successfully delivered. That's an example of a partial fault. The Byzantine fault model most closely matches reality. Our assumption of a crashing fault is a simpler fault model. Keep in mind that real systems have to handle much more complex faults.

To recap, we have independent nodes running the same program communicating over a fallible network. A consequence of each node having its own local clock is that the system as a whole does not have a consistent view of the current time. Each node has its own, local perception of time. This has really significant implications. Clocks are important because by timestamping operations we can put them in order. In a system where each node has its own clock, timestamps are only comparable when they come from the same node. That means any ordering of operations must be local to a node. We can't guarantee that any two nodes' clocks are in sync and therefore we can't compare

timestamps between nodes. If one node's clock is even a little bit out of sync with the others, that node will perceive a different ordering of events. We as observers can see the "correct" ordering but that is only because we implicitly assume a global clock that the system doesn't have.



Clock skew between nodes

In the diagram above, node B has a **clock skew** (i.e. error) of three seconds. By comparing the relative positions of the events against the global clock on the right, we can see that the true order of events is: A1, B1, A2, B2, A3. But because of the skew, node B is writing incorrect timestamps. The order reported by the timestamps is: A1 and B1 concurrent, B2, A2, A3. Such discrepancies can easily lead to errors if an observing node applies the operations in an incorrect order.

So we can't rely on each node's system clock being accurate. Why can't we use some kind of centralised clock server that provides the system with a global time? While time servers do exist, the speed of light means that each node will have a slightly different perception of that clock. Imagine that the global clock broadcasts to every node the message "it's now 12:00pm exactly". Since each node is at a slightly different distance to the time server, plus random network fluctuations, each message will experience slightly different amounts of latency and so will be delivered to each node at slightly different times. Any other non-clock ordering system we might come up with would still face the same latency problem. We must therefore assume that messaging is asynchronous and make no assumptions about timing.

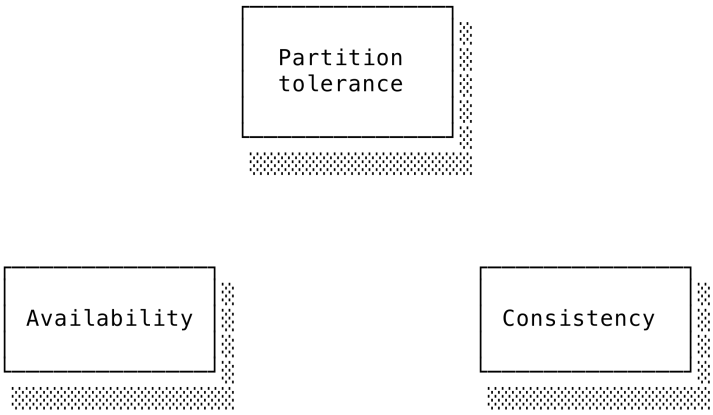
A further detail of our model is that nodes have fast access only to the state in their local memory. While a node can request state from other nodes, it can never be sure that the state hasn't been updated while the response is in transit. Any "global" state shared between nodes is potentially out of date. Knowledge is therefore always local.

Let's look at our network. It is fallible and so messages might be delayed or fail to arrive. A **network partition** happens when a network error cuts off healthy nodes from the rest of the system. The network is partitioned into separate sections. Network partitions make it difficult to reason about error states. How does a node determine whether a message it's sent has been lost by the network

or merely delayed? How does the node distinguish between a node failure and a network failure? A common design technique is for nodes to send a "heartbeat" message to each other to check that they're still available and reachable. If node A sends a heartbeat message to node B, for how long should it wait for a response before deciding that node B is unreachable? There is no obviously correct answer.

Handling network partitions: the CAP theorem

A distinguishing characteristic of distributed systems is how they respond to network partitions. Eric Brewer in 2000 presented[40] a simple idea: a distributed system cannot be simultaneously Consistent, Available and Partition tolerant:



The CAP theorem triumvirate

This idea was formalised into the CAP theorem in 2002 by Gilbert and Lynch, taking its name from the first letter of each property. The theorem assumes a system model very similar to our own, except it has the further simplifying assumption that nodes don't fail. We can use the theorem to determine the limitations of our own model. The theorem proves that when partitions occur, the system cannot preserve both consistency and availability. It's a useful model for thinking about how distributed systems behave when network problems occur but it does have some limitations, which I'll deal with after we take the three sides of the triangle in turn.

Consistency, in an informal sense, means that the system agrees on the current state. In an ideally consistent system, an update to a value would be applied to all nodes instantaneously. The system would behave identically to a single-node system. We already know, however, that "instantaneous" is not possible in reality. It takes at least *some* time for the update to propagate to the other nodes. Therefore what we must settle for is consistency after some time period that doesn't unduly affect users. A variety of consistency models can be created depending on which orderings of events you consider permissible. Very roughly speaking, the "strength" of a consistency model measures how close it gets to the ideal. In the further reading section, I've included a reference to a map of

[40] https://people.eecs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf

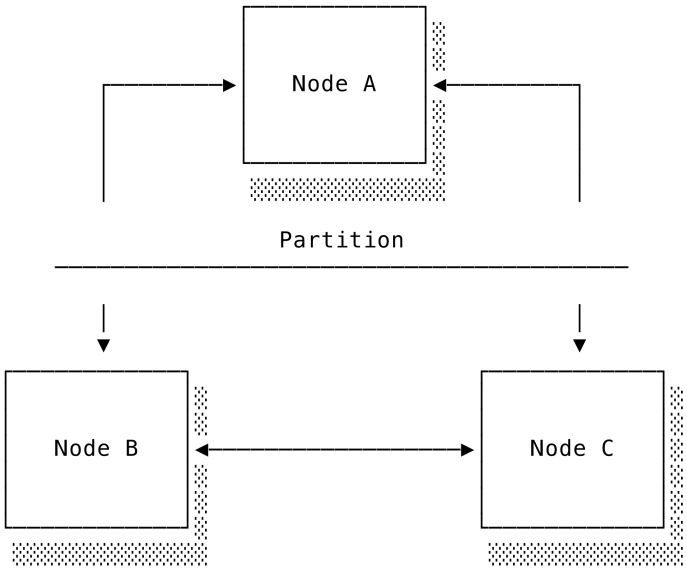
consistency models that contains *sixteen* different models (don't worry, you don't need to know them all!).

In the definition of the CAP theorem, consistency has a precise meaning: every read of a value must return the most recently written value (or an error). It's not instantaneous but, effectively, when queried, all nodes will agree on the same state for a given value. This particularly strong form of consistency is known as **linearizability** and we'll cover it in more detail below.

Availability is a little simpler. It means that a request to a live node eventually generates a non-error response. Informally, the system "works", although possibly with delays. Note that there is no requirement that the response returns the latest value. It's possible that the system will return stale (i.e. out of date) reads or even lose writes. We can see that availability does not require consistency.

Finally, **partition tolerance**. This one is poorly named. As we know, a partition occurs when a network failure breaks the network into separate sections with no communication across the partition. Messages from nodes in one partition never reach nodes in the other. Partition tolerance means that the system will continue to process requests even if a partition occurs. According to how we have modelled our system, partition tolerance means that the system will continue to function despite an arbitrary number of messages failing to appear in their destination node's message queue.

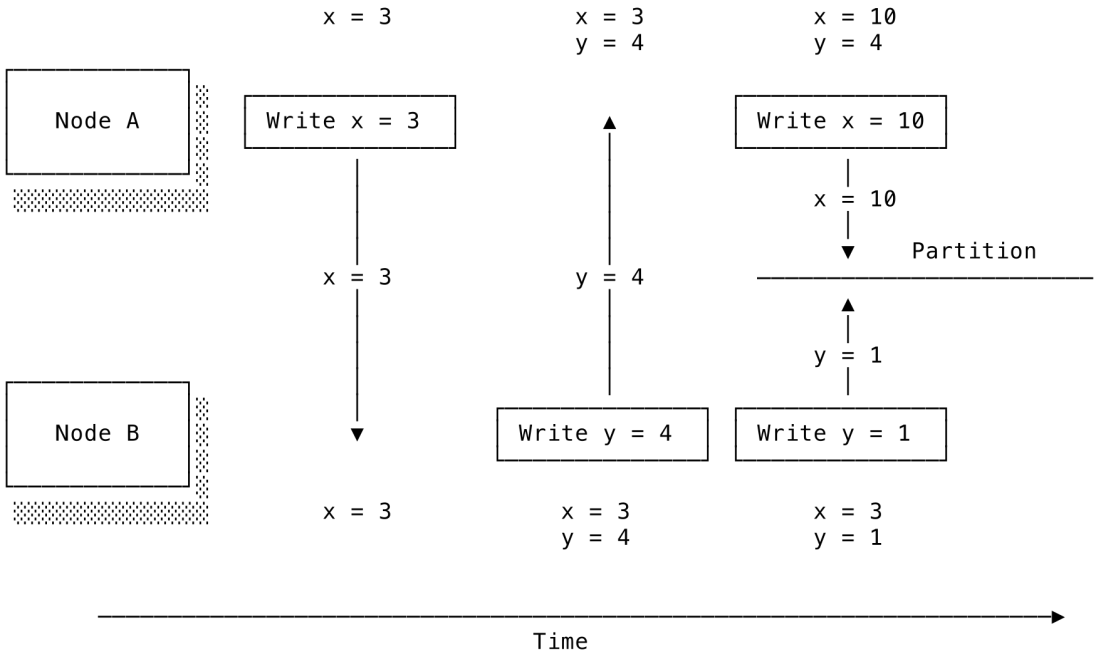
Why do I say "poorly named"? Well, flakey networks are an unfortunate fact of life. A designer of a distributed system doesn't have the luxury of deciding that partitions won't occur. They have to decide how the system will respond when one *does* occurs. This leads us to a common misunderstanding of the CAP theorem. You will often see the theorem summarised thus: "you can have any two of consistency, availability and partition tolerance. Take your pick!" This is incorrect and it's obvious to see why if we take a simple example:



Node A cut off by a network partition

A partition has occurred and separated node A from nodes B and C. Nodes B and C can communicate with each other but not with A. Should A continue to serve requests? If it refuses to serve a request,

then we have a live node returning errors and so we have given up availability. Should it respond to a read request, it has no way of knowing that B and C haven't accepted a newer write. If the user sends a write to C and then reads from A, the system will return a stale value, thus breaking consistency. Even worse, if A also accepts writes, we could end up with two divergent "histories". B and C think the current state is one value and C thinks it's a different value. This is known as **split brain** and is very difficult to fix. It's a bit like the worst merge conflict you can imagine but far, far worse. Horrific, news-worthy outages often involve split brain.



Partitioned nodes both accept writes, leading to divergent histories

The lesson from the CAP theorem is that when partitions inevitably occur, we must choose between availability and (linearizable) consistency. We can have consistency with partition tolerance or availability with partition tolerance but not consistency with availability and certainly not all three. The question for the system architect (i.e. you, dear reader) becomes: which of consistency and availability is more important?

Before answering that, we must discuss the limitations of the CAP theorem. A theorem proves that when certain preconditions hold, the theorem's conclusions must be true. If we deviate from those preconditions, the theorem does not necessarily still apply. The conclusions only apply when using the exact same definitions and assumptions as the proof. The CAP theorem's apparently binary choice between consistency and availability only holds when we are talking about linearizability because that is how the theorem defines consistency.

Similarly, the CAP theorem's definition of availability only considers what happens when requests reach nodes *within* the system. A practical response to a network partition would be to re-route traffic away from the nodes in the minority partition to nodes in the healthy partition. From a user's perspective, the system would continue to work correctly. Nevertheless, the CAP theorem

would not consider this system to be available because its definition of availability only considers what happens when a request actually reaches a node. In practice, weaker consistency models permit more availability because there are fewer constraints on the responses they can provide.

What the CAP theorem is telling us is that stronger consistency implies less availability and weaker consistency permits more availability. The question is not whether we want consistency or availability, but which do we want more of.

If the distributed system holds business-critical data, we probably value consistency over availability. It would be very troubling if an invoice appeared to be paid one moment and unpaid the next. In that case, the system should preserve consistency by refusing to respond to requests if there's a danger of returning stale values or losing data. We'll see various implementations of that idea shortly. Sometimes, by contrast, availability is more important. A Twitter user probably doesn't mind if their timeline doesn't show absolutely up to date results so long as it shows *something*. When shopping online, it's preferable to have a deleted item occasionally reappear in your shopping cart than have the website completely crash. Such systems will allow stale reads and maybe even lost writes as the price of greater availability.

To summarise, what the CAP theorem demonstrates is that no distributed system, even one as simple as our abstract model, can guarantee the same behaviour as a single-node system. Some deviation is an inevitable consequence of non-instantaneous communication. Availability and consistency are competing priorities and the correct balance depends on the intended use case. It's important not to attach too much significance to the CAP theorem. While an important result, its applicability is limited to the model in its definitions and it's not necessarily a good tool for thinking about real world systems.

Consistency models

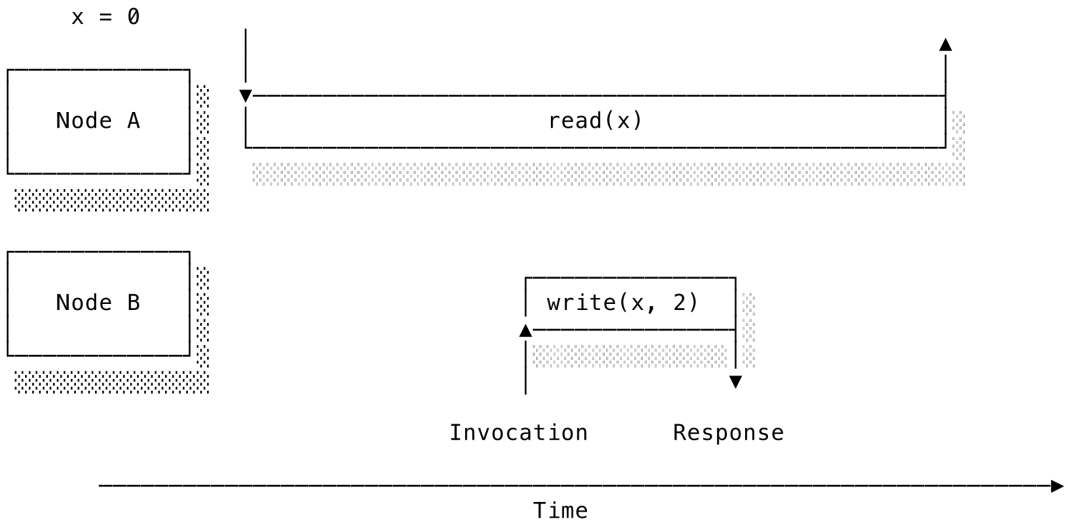
In this section we'll analyse a few common consistency models and their implications for availability. Consistent data replication is a central problem in distributed computing. It's obviously critically important for distributed databases, one of the most common examples of distributed systems, and other systems will most likely have some kind of internal data store that needs to replicated consistently. For example, Kubernetes, a tool for deploying applications on computer clusters, uses a distributed store called "etcd" to hold the cluster state.

I must admit that the terminology in this discussion can be a bit confusing and the differences between consistency models are quite subtle. We'll start with the strongest form of consistency and check out a few interesting models on our way down the strength scale. Generally speaking, the stronger the consistency model, the easier it is to understand but the harder it is to implement. Stronger models also have greater latency costs and are less available.

As an aside, "consistency" as used in the CAP theorem is totally different to the "consistency" in a database system's ACID guarantees. ACID consistency only ensures that each transaction brings the database from one valid state to another. The "isolation" guarantee is actually closer to what the CAP theorem means by consistency. The strongest isolation level is "serializability", which

ensures that concurrent transactions always leave the database in the same state as if the transactions had been performed in some sequential order. This is a weaker guarantee that linearizability. The terminological confusion arises because distributed systems and databases were originally developed and studied separately. The database folks were concerned with transactions, groups of operations modifying multiple objects, while the distributed system people were talking about single operations on single objects distributed across multiple locations.

Let's take our abstract model from the previous section and run a distributed key-value store on it. The two valid operations are `read(key)` and `write(key, value)`, or `r(k)` and `w(k,v)` for short. The initial value for each key is `0`. Operations belonging to the same node must occur in order according to the node's clock. We assume that, due to communication delays, there is some period of time between an operation's invocation and response. Operations that occur during the same time period are said to be concurrent and have no ordering between them. That opens the door to non-determinacy because concurrent operations may be interleaved in different orders, leading to different responses and final states. The system's consistency model determines which interleavings are permitted. A stronger system allows fewer interleavings while a weaker model is more permissive.

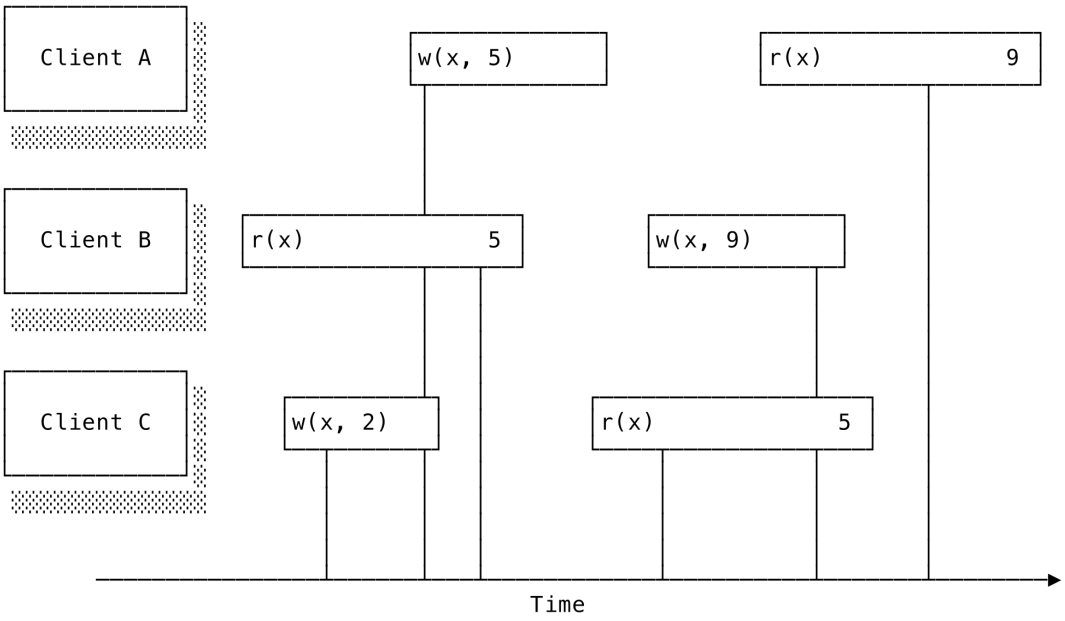


Should the read return the old or the new value?

Linearizability

In simple terms, a linearizable data store appears to behave like a single-node data store. Nodes always agree on the current value. That's very attractive because programmers are already familiar with how single-node systems work. We assume that each operation executes atomically at some point between invocation and response. Once the operation executes, it "takes effect" and every operation executing after that point must be able to see the effects. So once `w(a, 10)` executes, `a` holds the value `10` and all subsequent `r(a)` operations must return `10` (or a later value). Because later operations always see the latest value, every node agrees on the current state. Linearizability

still allows for some non-determinacy. When two operations are concurrent, there is no ordering between them. A read that is concurrent with a write can return either the old or the new value, provided that the new value hasn't yet been read.



In the diagram above, each operation is invoked at its leftmost point and responds at the rightmost. Time goes to the right. The vertical lines mark the moment in which each operation is atomically executed. You can draw a line from execution point to execution point. The system is linearizable if the line always goes from left to right (i.e. never backwards in time) and never reads stale values. The execution points could occur anywhere within the operation's duration – this is just one possible arrangement. Clients A and C both have reads concurrent with B's write. Since they're concurrent, both reads could have returned either the old or the new value. But once client C returns the new value, client A must also return the new value to prevent a stale read.

An assumption in the formal definition of linearizability is that there is a global clock ordering every operation. When I breezily say above that "time goes to the right", I'm implying that each operation in the diagram is positioned relative to this global clock. Now, we already know that a global clock is impossible and so in reality nodes can't be sure exactly when operations execute on other nodes. What is important is that the system behaves *as if* the operations had been ordered by a global clock. There are a number of ways to implement this in the real world.

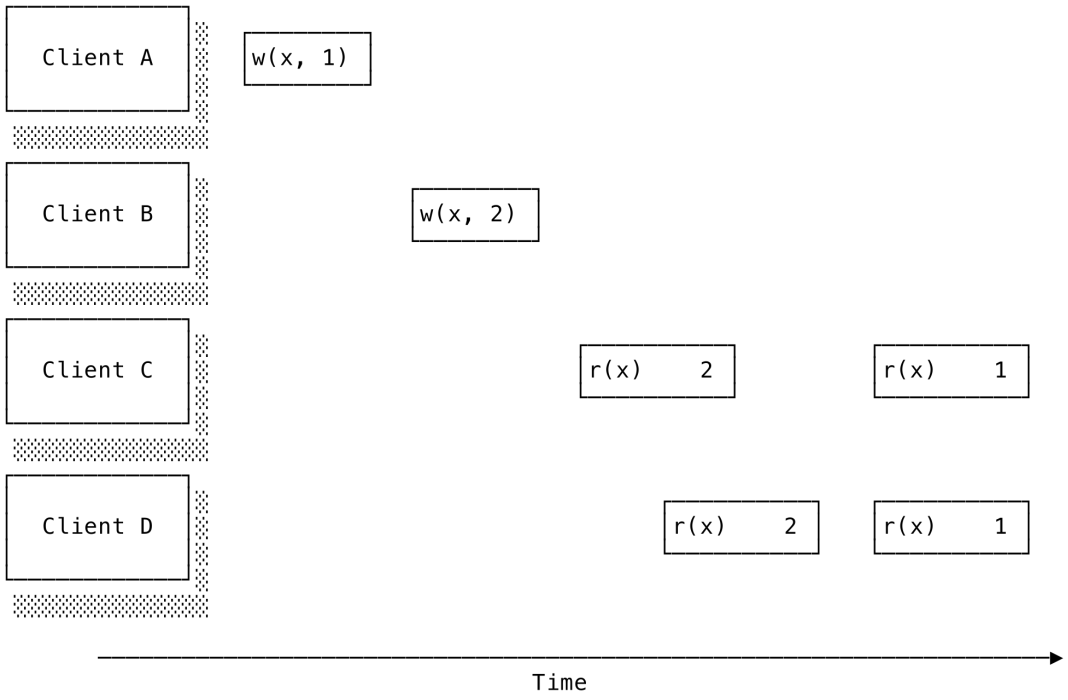
Google have come up with an interesting approach in their distributed database, Spanner. Rather than using timestamps to record a possibly inaccurate moment in time, Spanner uses an API called TrueTime to produce time *ranges* during which the operation definitely happened. To ensure that operation B happens after operation A, the system simply waits until the end of operation A's time range has passed. Spanner received a lot of publicity because Google announced that TrueTime uses machines with fancy atomic clocks and GPS antennae to keep the interval of uncertainty as short as possible. Such specialised hardware is not required in principle. CockroachDB is an alternative

implementation of the same idea that runs on commodity hardware. Google claims that Spanner offers "external consistency", which is linearizability with additional semantics for transactions on multiple objects.

The only reason we only care about global clocks in the first place is because they give us an easy way to put operations in order. A second approach is to find other ways to create a system-wide ordering of events. We'll see below how consensus protocols such as Raft achieve this.

Sequential consistency

If we remove the requirement that operations are ordered according to a global clock, we have sequential consistency. The order of operations must respect each node's internal ordering but there is no required ordering between nodes. Once a node has observed the effects of an operation, it must not return stale state from before the operation. Informally, sequential consistency allows for an operation's effects to take place after the operation completes (or even before its invocation, in theory).



The example above is not linearizable but it is sequentially consistent. Think of all the different ways you might interleave the operations while ensuring that each node sees its own operations occur in order. One consistent ordering would be:

```
B: w(x, 2)
C: r(x) 2
D: r(x) 2
A: w(x, 1)
```

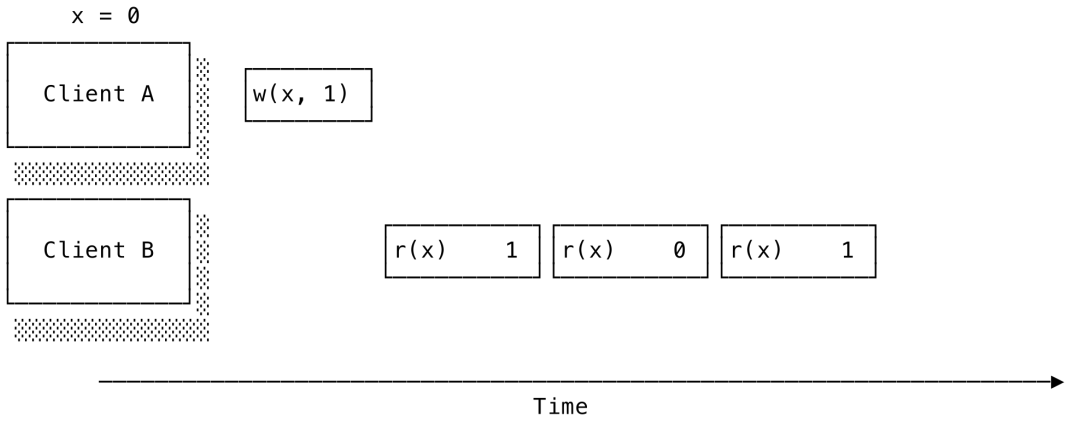
```
C: r(x) 1
D: r(x) 1
```

Although `A: w(x, 1)` was the first operation according to the global clock, in this ordering its effects didn't actually take place until after the first two reads had occurred. I find it helpful to visualise sequential consistency as an asynchronous queue that preserves logical order. Imagine a photo-sharing application popular with influencers trying to hide the emptiness of their existence. When an influencer uploads some photos of their lunch, the uploaded photos go into an asynchronous work queue for resizing and compression. The node that handles the uploading will see its own operations in order and so it will insert the uploaded photos into the processing queue in the same order as they were uploaded. It will take some unknown time for the effects of each successful upload operation to become visible on the influencer's timeline, but the photos will appear in the upload order.

Linearizability and sometimes sequential consistency are normally what is meant by "strong" consistency. Sequential consistency is linearizability without the real time ordering. A strong consistency model will only show consistent state. A value will not flip flop between old and new versions. The appeal of strong consistency is that the system exhibits behaviour that is familiar and intuitive to programmers. The downside is that this consistency requires more coordination. Nodes need to spend a lot of time communicating to keep each other informed. That network chatter entails increased latency and a vulnerability to partitions.

Eventual consistency

A much weaker consistency model is eventual consistency. This only guarantees that, in the absence of further writes, every read will *eventually* return the value of the latest write. In the meantime, reads may return stale values. There is no illusion of a single data store. The system exposes its progression towards the eventual state:



Examples of eventually consistent systems include Amazon DynamoDB, DNS and Kubernetes. Clients should not expect to receive the latest data and it's possible that reads will return fluctuating values. Compared to a traditional, single-node data store, this is very unintuitive. On the plus side, nodes in the system need to spend less time coordinating. A node receiving a write operation can

perform it, immediately respond to the client and then asynchronously propagate the update to other replicas. That results in lower perceived latency for the user. However, they may see old state if they query a node that hasn't yet received the latest updates. The system will also be more available in the event of partitions. Nodes can simply continue serving stale reads and catch up once the partition is fixed. Eventual consistency is appropriate when speed and availability are more important than exactly correct values.

Eventually consistent distributed databases are often said to offer **BASE semantics** instead of the traditional ACID guarantees (don't you love chemistry puns?). What this means is:

- Basically Available: read and write operations are available as much as possible but without any kind of consistency guarantees. Reads might be stale and writes might not persist after resolving conflicts.
- Soft state: the state of the system might change over time, even without further input, as it converges on the eventual state.
- Eventual consistency: over time, and without further input, the system will become consistent.

You can see that they had to stretch a bit to get the definitions to fit the acronym. BASE systems favour availability over strong consistency and often talk about making "best efforts" rather than guarantees. The NoSQL trend has in recent years seen a spate of new database systems offering BASE semantics. Whether eventual consistency is suitable depends on your use case. If in doubt, favour stronger consistency.

Consistency protocols

In the previous section, I defined consistency models in terms of permitted orderings of operations. That tells us how a consistency model behaves but not how it might be implemented. This is where **consistency protocols** come in. Consistency protocols implement consistency models. They are a contract between the node processes and the data store. If the processes implement the protocol correctly, the protocol guarantees that the data store will behave with the expected consistency. The weaker the consistency model, the easier the protocol is to implement in terms of complexity and resource requirements (primarily time). The protocols below offer strong consistency and can be tweaked to provide greater availability and speed at the cost of weaker consistency.

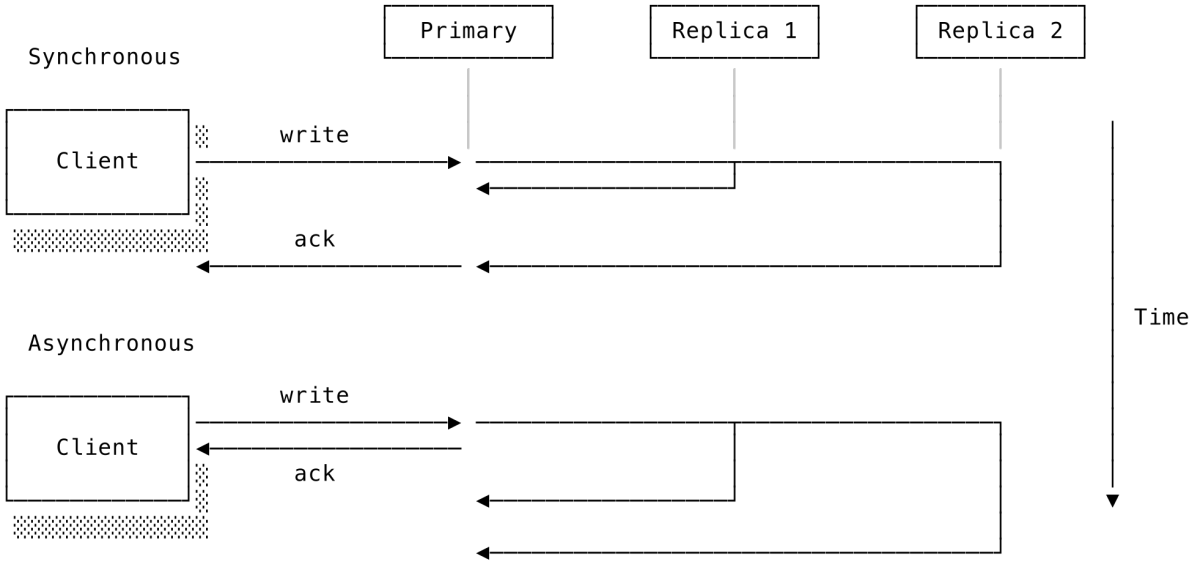
Primary-secondary replication

Also known as leader-based replication or master-slave replication. One node is designated the "primary" and the remainder are "secondary" nodes. The primary accepts read and write operations and the secondaries only accept reads. All changes are performed on the primary and then replicated to the secondaries. This makes it easy to keep the system consistent since all changes go through a single node. Should the primary fail, the system can automatically **failover** by switching a live secondary to be the new primary.

This architecture is a popular way to provide high availability for a database, provided that a brief period of downtime during failover is acceptable. A common work pattern for a database is to have many read requests and few write requests (imagine how many people read Wikipedia and how many edit it). As long as the number of write requests remains low enough to be handled by a single node, the system can easily scale to handle increasing read load by adding new secondaries.

The precise form of consistency on offer depends on how updates are replicated to the secondaries. If the replication is synchronous, the primary doesn't respond until every secondary confirms receipt. This provides strong consistency because every secondary always maintains a full copy of the latest data. A failover would not cause any data loss because the new primary would have the same data as the old one. As you can see in the diagram below, waiting for confirmation from every node increases overall latency and a single slow secondary is enough to hold up progress for the whole system. Availability is also limited since the primary cannot accept a change unless every secondary acknowledges it.

Asynchronous replication provides eventual consistency. The primary responds to the client without waiting for every secondary to confirm receipt. As the diagram shows, this is faster because we don't need to wait for every secondary to acknowledge. The primary can continue accepting requests even if secondaries are partitioned. On the flip side, it's possible that an out-of-date secondary will return stale reads if a read request reaches it before the latest changes are replicated. Data loss could even occur during failover if a secondary gets promoted to primary before it sees an update accepted by the old primary. Since the state of the primary determines the state of the system, any changes not replicated to the new primary would be lost.



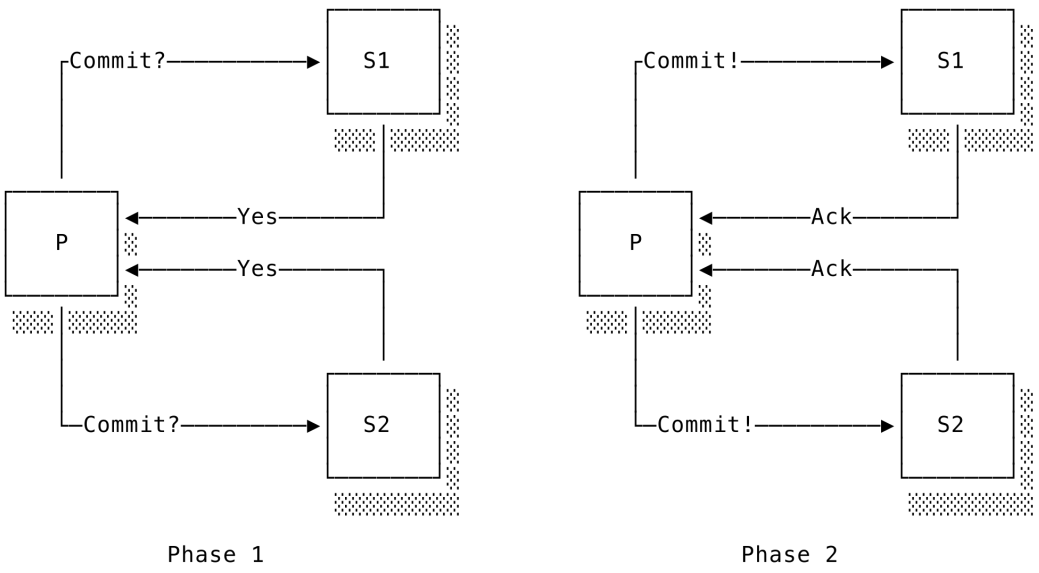
Synchronous and asynchronous replication

Two-phase commit

Two-phase commit (2PC) is a simple protocol that offers strong consistency. It works as a distributed transaction. A primary node instructs the secondary nodes to perform an operation and to report whether they think the transaction should commit or abort. Each secondary performs the work, saving the results to a temporary log, and reports the outcome. If every node reports success then the primary instructs the secondaries to proceed with the commit. If any node reports an error, the primary instructs the secondaries to abort the transaction. A real world example is a marriage ceremony:

> Officiant: Adam, do you take Steve to be your husband?
> Adam: Yes!
> Officiant: Steve, do you take Adam to be your husband?
> Steve: Yes!
> Officiant: You are now married!

The officiant is the primary node and Adam and Steve are both secondaries. The act of marriage is the transaction. Crucially, it is not sufficient for Adam and Steve to both say "Yes!". The marriage is only confirmed (i.e. the transaction committed) when they both accept *and* the officiant confirms it.



A successful two-phase commit

Two-phase commit is superficially similar to synchronous primary-secondary replication. The difference is that in primary-secondary replication, the secondaries have no choice but to accept whatever the primary pushes out to them. In two-phase commit, the primary is seeking **consensus**: it is trying to get a system of potentially faulty nodes to agree to commit a value. Each secondary has the choice to reject the proposed value by responding with an "abort" message. Two-phase commit

can be used to agree on any value. Synchronous primary-secondary replication is a special case of two-phase commit in which the values the primary proposes are always new changes and the secondaries always accept them.

Due to these similarities, two-phase commit suffers from the same limitations as synchronous replication. It is very susceptible to network partitions and delays. Each change requires at least four network transmissions per node. The system can only proceed at the speed of the slowest node and all nodes must be available. Because of this, two-phase commit is relatively rare in production systems. There is a three-phase variant that avoids being blocked by a slow node but it is still partition intolerant. Can we do better? Yes!

Fault-tolerant consensus protocols

The consensus offered by two-phase commit sounds useful. It's a shame that the protocol is so easily blocked by slow nodes and partitions. What if we loosen the requirement that every secondary has to agree? Fault-tolerant consensus protocols achieve consensus while allowing some nodes to fail. At a high level, fault-tolerant consensus protocols work like two-phase commit but only require a response from a majority of secondaries. They achieve this at the cost of much greater implementation complexity. The protocols are sophisticated and hard to implement correctly in real world systems with Byzantine fault models.

The first fault-tolerant consensus protocol was developed in 1989 by Leslie Lamport (mentioned in the introduction) and is known as **Paxos**. Lamport thought that his protocol would be more memorable if he gave it an exciting back story and so he presented it as if he had unearthed the voting rules of the ancient parliament of the Greek island Paxos. He even went so far as to dress up as Indiana Jones when presenting it at conferences. Unfortunately, computer scientists are not known for their sense of humour and the performance did not have the desired effect. It took a few more years for the significance of the protocol to be fully appreciated. Partly this is due to its complexity. Even with the faux archaeology stripped away, Paxos is notoriously difficult to understand and implement correctly. A later protocol called **Raft** offers the same guarantees and is designed to be easier to understand and implement. Both Paxos and Raft can tolerate the failure of up to N nodes if $2N + 1$ remain available. It will seem to clients that they are interacting with a single, reliable system even if a minority of the nodes in the system fail.

Let's get a sense of how fault-tolerance consensus works by looking at Raft. There are far too many details to give a comprehensive explanation here, so I'll focus on the core elements. Nodes in the system reach consensus through successive rounds of voting divided into periods called "terms". Each term begins with an election to be the leader for that term. Every node starts off as a follower. After a random timeout, it will promote itself to candidate and request that the other nodes elect it as leader. A follower will vote for a candidate if it hasn't already voted for another. The candidate that receives a majority of votes is elected leader. Interestingly, leader election is thus itself an example of achieving consensus. The leader acts as the primary node, coordinating work and handling communication with clients.

When the leader receives a request from a client, it writes the change to its local log and broadcasts

the change to the followers, who each write the changes to their own local log and respond with a confirmatory vote. The leader waits for a **quorum** of votes before committing and instructing the followers to commit the changes. A quorum represents the minimum number of votes needed for a decision to be considered valid. In the U.S. Senate, for example, a majority of senators must be present to have a quorum. Let's say the total number of votes in the system is N, the quorum for read operations is R and the quorum for write operations is W. By imposing a few simple restrictions on the quorum values, the system can ensure consensus:

- $W > \frac{N}{2}$. This means a majority of nodes must vote for every write operation. Since a node can only vote for one write operation per round, this restriction ensures that only one write operation succeeds per round.
- $R + W > N$. This means that every read and every write operation will share at least one node. The read quorum will always have at least one member with the latest value.

Setting R and W to smaller values would allow reads and writes to be performed more quickly, since fewer followers have to vote on the operation, at the cost of permitting inconsistent behaviour.

The beauty of Raft is that a set of comparatively simple rules is enough to handle many failure cases. Imagine that a partition cuts off the leader from a majority of the followers. The leader will not receive a quorum of votes and so will not be able to commit any changes. This prevents inconsistency but the system cannot make progress without its leader. When the leader cannot communicate with the other nodes, we don't know whether it's because of a network issue or the leader has become incapacitated in some way. Followers that don't hear from the leader will wait a random amount of time before starting their own leadership candidacy. Due to the leadership voting requirements, any new leader must be able to communicate with at least a majority of nodes. If the partition is later resolved and the previous leader reappears, the leader with the higher term takes precedence. Thus the protocol provides fault-tolerant functionality.

I've skipped over many of the details but I hope the general mechanism is clear. Nodes elect a leader amongst themselves and perform the same operations in the same order specified by the leader. Since the system as a whole agrees on the order of operations, we have linearizability even though there is no global clock. As you might imagine, all of the voting and elections add latency to every operation and so such protocols will never be super performant. What fault-tolerant consensus protocols offer is linearizable consistency and partition tolerance so long as a majority of nodes survive. Paxos and Raft can't quite claim to be available according to the CAP theorem's definition because requests to nodes in a minority partition must fail, but in practice this is not an issue because the nodes in the majority partition continue to operate.

I highly recommend that you look at the Raft visualisation resources in the further reading. The protocols are difficult to understand by reading detailed instructions. You can get a much more intuitive sense of their operation by watching them work. They are truly remarkable. You will find that many distributed systems use some kind of Paxos- or Raft-based implementation for at least some of their internal state. Google Spanner uses Paxos to manage replication across nodes. Kubernetes' etcd store is based on Raft.

Conclusion

Distributed systems offer many alluring benefits but come with a hefty complexity cost. Tasks that were once straightforward become much more complicated when multiple nodes are involved. Things can and frequently do go wrong in confusing, non-obvious and very difficult to debug ways. Yet distributed systems are simply unavoidable if you want to tackle big problems because they free us from the limitations of a single machine.

Distributed systems are characterised by independent nodes communicating over an unreliable network to achieve a common goal. An unreliable network is challenging because it makes it impossible to know if message delivery has succeeded. A particularly interesting (and frustrating!) feature of distributed systems is that the system has no global clock. Each node has its own, local perception of time with no ordering between nodes. This complicates achieving consistency because nodes may have different perceptions of the order in which operations occur, leading to different outcomes.

The CAP theorem declares that when a network failure occurs, the system must either sacrifice availability, by not accepting some requests, or consistency, by allowing conflicting writes (known as split brain) or reading old values (stale reads). The important point is no distributed system can offer the same consistency and performance guarantees as a single-node system. Rather than seeing consistency and availability as binary states, we should see them as ranges of behaviour. If we loosen our consistency requirement to something weaker than the CAP theorem's linearizability, we can have both consistency and availability to some degree. Consistency requires communication to keep data stores in sync and so the stronger the consistency offered by a system, the less it can be available and vice versa. Weaker forms of consistency include sequential consistency, which is like an asynchronous message queue, and eventual consistency, which only guarantees that the system eventually converges on the correct value. All of them are appropriate in certain situations. There is no one size that fits all.

A simple way to ensure strong consistency is to use synchronous primary-secondary replication. One node is designated leader, handles all changes and replicates them to secondary nodes. This works well in the specific case of a database replicating changes across the system. A more general protocol is known as two-phase commit, in which all nodes must agree to commit changes as part of a distributed transaction. The problem with both mechanisms is that they are very sensitive to network errors or slow nodes. Asynchronous replication is one solution but it introduces the risk of inconsistency. Fault-tolerant consensus protocols, such as Paxos and Raft, resemble a two-phase commit protocol in which the nodes elect their own leader and node failures is tolerated as long as a majority of nodes remain.

I hope that this chapter encourages you to think about how networked computers behave as part of a bigger system. We won't all get the opportunity to work with some fancy Kubernetes cluster or Spanner database, but every web developer is working on a distributed system in some way. You are now better equipped to reason about how the system operates, how it handles its state and how it can go wrong.

Further reading

Good written resources for distributed systems are a little thin on the ground. It's an area that really benefits from hard won, practical experience. You'll notice that distributed system experts all have the kind of thousand yard stare that comes from debugging intermittently failing systems. There are a few good blog posts that attempt to distil years of experience and set out best practice. I like Notes on distributed systems for young bloods[41] and a good next step is Distributed systems for fun and profit[42]. It's relatively short but goes into more depth than I've had space for here. Special mention must go to Jepsen[43]. He regularly provides detailed reports on the real world performance of distributed databases and also provides plenty of explanatory and background material. It's very illuminating to see just how hard it is to get things right in this space.

I *highly* recommend that you check out Raft's website[44] for in-browser visualisations of how the protocol works. It'll make so much more sense, trust me.

Until fairly recently, I struggled to find a really good textbook on distributed systems. Tanenbaum and van Steen's *Distributed Systems* is available for free but I honestly found it very dry. Happily, Martin Kleppman has come along with *Designing Data-Intensive Applications*. This book is as close as I've seen to a publishing phenomenon in the world of computer science textbooks. Recommendations for it keep popping up everywhere and with good reason: it's eminently readable, focused on practicalities and very wide-ranging. I've already recommended it in the databases chapter but really the heart of the book is on distributed systems. After getting confused by various technical definitions of linearizability and other consistency models, I was especially grateful for its focus on informal, intuitive explanations. If you're not quite ready for a full textbook treatment, another good starting point is Kleppman's blog post critiquing the CAP theorem[45]. It has links to other useful resources.

An obvious recommendation is to go build a distributed system. That, of course, is not always straightforward. Something I've been mindful of in this chapter is that many people won't have the opportunity to work with exciting, highly available, globally distributed systems. I have two suggestions. Firstly, the Grokking the system design interview course[46], which comes highly rated, teaches you to think through how such systems might have been designed. You'll develop a much better understanding of the considerations and trade-offs involved in architecting reliable systems. Secondly, I recommend familiarising yourself with Kubernetes. A managed cluster can be cheaply and easily deployed on any of the major cloud services. Kubernetes is interesting because it's eventually consistent and so exposes a lot of intermediate state to the user. It's based on the simple concept of control loops (think how a thermostat works) and it's neat to see how the composition of simple loops builds complex behaviour. Deploying applications is a relatively slow process so you can literally watch as the cluster state gradually converges on the desired state.

[41] https://www.somethingsimilar.com/2013/01/14/notes-on-distributed-systems-for-young-bloods/
[42] http://book.mixu.net/distsys/index.html
[43] https://jepson.io
[44] https://raft.github.io
[45] https://martin.kleppmann.com/2015/05/11/please-stop-calling-databases-cp-or-ap.html
[46] https://www.educative.io/courses/grokking-the-system-design-interview

Programming languages

Introduction

A programming language is a language designed for specifying programs. We instruct computers to understand these languages and perform the computations they specify. The branch of computer science that studies programming languages is known as **programming language theory** (PLT). It sits at a really interesting intersection of computing, linguistics, logic and mathematics.

The standard academic analysis of programming languages generally involves working up from logical first principles. We'll approach the subject from the other direction by making a survey of a few popular programming languages. We'll begin by looking at how programming languages can be understood in terms of their syntax and semantics. Then we'll examine how they can be categorised into different paradigms according to a few important, distinguishing features. Type systems are important and interesting enough to merit their own section at the end of the chapter.

Let's begin our survey by looking at what Wikipedia has to say about JavaScript, Go and Haskell, three popular and distinctive languages:

> JavaScript[47] is a high level, just-in-time compiled, object-oriented programming language that conforms to the ECMAScript specification. JavaScript has curly-bracket syntax, dynamic typing, prototype-based object-orientation, and first-class functions.
>
> Go[48] is a statically typed, compiled programming language designed at Google … Go is syntactically similar to C, but with memory safety, garbage collection, structural typing, and CSP-style [communicating sequential processes] concurrency.
>
> Haskell[49] is a general-purpose, statically typed, purely functional programming language with type inference and lazy evaluation.

What's remarkable about all three of these descriptions is that they are at once highly informative to an experienced reader and utterly baffling to everyone else. By the end of this chapter, you'll understand every bit of that techno-babble!

Defining a programming language

Programming languages are examples of **formal languages**: languages defined by a formal specification. They are neat and logical, in comparison with natural languages (e.g. English, Mandarin).

[47] https://en.wikipedia.org/wiki/JavaScript
[48] https://en.wikipedia.org/wiki/Go_‘programming_language‘\protect\char”0024\relax
[49] https://en.wikipedia.org/wiki/Haskell_‘programming_language‘\protect\char”0024\relax

Grammar defines the elements of the language and how they can be combined into correct statements. The rules of the grammar are known as the language's **syntax**. Knowing these rules, we can write syntactically valid source code and the computer can analyse it to derive the structure of the specified program. We'll cover this process, compilation, in much more depth in chapter ten.

Syntax is the surface level of how the language looks and feels. Programmers generally like their languages to feel familiar and so certain syntax patterns are common across languages. C uses curly brackets to delimit blocks of code:

```
int function square(int n) {
  return n * n;
}
```

Both JavaScript and Go have a similar syntax rule. This is what Wikipedia means when it says that JavaScript has "curly-bracket syntax" and Go has "C-like syntax". This is the same function in Go:

```
func square(n int) int {
  return n * n;
}
```

There are a couple of minor differences but structurally they look very similar. The same function in Haskell, which is influenced by a different family of programming languages, has completely different syntax:

```
square :: Int -> Int
square n = n * n
```

Syntax only tells us whether a bit of code is a valid fragment of the language. It doesn't tell us anything about what should happen when that fragment is executed. The behaviour of the language is known as its **semantics**. The semantics of C, Go and Haskell all specify that we have defined a function called `square` that takes an integer called `n` and returns another integer that is the square of `n`. Even though they are syntactically different, all three functions have identical semantics.

The smallest semantic unit is the **expression**. An expression is anything that can be **evaluated**, or computed, to a value. `3 * 3` and `square(3)` are both expressions that evaluate to `9`. Expressions can be composed to create more complex expressions: `square(3 * 3)`.

Many languages have the concept of a **statement**: an executable chunk of code that might contain expressions but does not itself evaluate to a value. Statements are useful because they produce behaviour known as **side effects**. For example, an `if` statement usually has the following structure:

```
if (EXPRESSION) { A } else { B }
```

The statement controls whether `A` or `B` is executed next, depending on whether `EXPRESSION` evaluates to `true` or `false`. Statements such as `if`, `while`, `for` and `return` control the flow of execution through the program. Other important categories of side effects include mutating program state and input/output. These operations are all useful because of the change in system state they effect, not because they evaluate to a useful value.

If you're unsure whether something is an expression or a statement, ask yourself whether it is meaningful to assign it to a variable. Generally, assignment only allows expressions to be assigned. This JavaScript example doesn't work because the `if` statement doesn't evaluate to anything `status` could hold:

```
1  var status = if (user.isValid) { user.status } else { "no user" }
```

Origins and communities

Designing a language is fundamentally a creative act. Which syntax rules do you choose and what are the semantics? What are the core concepts underpinning the language and how do they all fit together? There is no single, correct answer to any of these questions and so there is no single, "best" programming language. What is an expression of sublime beauty to one person is a confusing and ugly mess to another. What works wonderfully in one problem space might be hopelessly awkward in another.

The combination of syntax and semantics forms the **specification** of a language. It defines what is considered syntactically valid code and how that code should behave when executed. The exact format of the specification varies. Commonly, languages start out relying on a single **reference implementation**. This will be a compiler or interpreter, most likely written by the language's initial creator(s), that is seen as definitive. The language is defined by how the reference implementation behaves. Later, if the language gains popularity and multiple implementations arise, it might prove useful to write a standardised specification document so that everyone has an agreed understanding of how the implementations should behave.

This is the path Ruby has taken. It began with a reference implementation called Matz's Ruby Interpreter (MRI) written by Ruby's creator, Yukihiro "Matz" Matsumoto. It has since evolved into multiple implementations standardised by a formal specification. An interesting thing about Ruby is that the specification itself is written in Ruby as a suite of unit tests. If a program successfully passes all of the tests, then it's a valid Ruby implementation!

Every language has its origin story. Sometimes the language is written by a lone developer, as Ruby was by Matz and Python was by Guido van Rossum. Sometimes a company might put together a team specifically to create a new language, as Google did with Go. No matter who the creators are, programming languages are made to fix a perceived deficiency. Other languages are too slow, too complex, too simplistic, too lacking in support for this or that idea.

There are many, many programming languages out there and very few of them escape obscurity, let alone become popular. What determines whether a language "makes it"? Sometimes it's a case

of being the right tool for the task. A language's semantics might make it especially well suited to a particular problem or environment. C originally became popular because it made it easier to write performant code for multiple architectures. Technical merits are not always sufficient. C also enjoyed an association with the killer app of the day: the Unix operating system. Go gained some initial name recognition thanks to its association with Docker and Kubernetes at the start of the container hype cycle.

Technical merits aside, Go undoubtedly owes much of its current popularity to the fact that it is backed by Google and has a paid team of developers working on its core libraries and tooling. Languages without corporate sugar daddies have to rely on their creator successfully building an enthusiastic community of volunteer contributors who will create the necessary ecosystem themselves. The social, community element of programming languages is just as important as it is with natural languages.

The purpose is always to communicate. Writing code is far more than just instructing the computer. That's the easy part, really. It's also a communication to all the other developers, current and future, who will read and interact with your code. Using the right language helps us to better express complex, abstract concepts in a way that is readily understandable by both people and machines.

Programming language concepts

In this section, we'll consider a few questions that will help us understand what distinguishes one programming language from another.

Levels of abstraction

Our first question is: at what level of abstraction does the language operate?

We know from previous chapters that the processor exposes an instruction set architecture (ISA) to software. How closely do the semantics of a programming language match the semantics of the underlying hardware's ISA? That is, what *abstractions* does it offer on top of the hardware? If a language offers few abstractions, we say that it is **low level**. Predictably, a language with a great deal of abstraction is **high level**. Generally, lower level languages are conceptually simpler and offer more control over the hardware. Higher level languages make it easier to express large-scale applications at the cost of performance and hardware control.

Abstraction refers to concepts that are present in the language but not directly supported by the hardware. For example, a processor will almost certainly have instructions supporting functions (e.g. `CALL`, `RET`) but no instructions implementing JavaScript promises. That's because promises are a high level abstraction that JavaScript creates out of more simple primitives.

To understand a programming language's abstraction level, a useful exercise is to describe some code in natural language. The nouns you use indicate the abstractions. Imagine we want to describe what an `if` statement does. An assembly implementation might be expressed in English as follows:

> Write this value into that register. Read the value stored in this memory address into that other register. Compare the two values. If the zero flag is set, add this offset to the program counter register. Otherwise, increment the program counter register.

Note that the program deals directly with the system's hardware elements. It's actually a little hard to work out the intent. Since each assembly is specific to its ISA, assembly programs are not **portable**: a program written in one platform's assembly won't run on another platform without modification. Assembly doesn't "abstract away" the details of the underlying ISA.

At one abstraction level higher, C was designed to be a "portable assembly". Its semantics assume a simple machine model, a little like what we saw in the architecture chapter, which a compiler can easily map to the specifics of the actual hardware. A C implementation could be like so:

> If this value equals the value of this pointer, take this branch else take that branch.

Note that it's much shorter because the precise hardware details aren't specified. We don't know in which registers the values are held. We are operating at a higher level of abstraction, though the reference to a "pointer" indicates that one value is held in some memory location. It's interesting to note that nowadays we think of C as a low level language because its semantics permit using pointers to noodle around the process' address space. Back in the 1970s, when C was first developed, it was actually seen as a high level language because the code wasn't tied to any specific ISA.

In a modern, high level language such as JavaScript, Ruby or Python, the same code might be described as:

> If this user object's ID property matches that user object's ID property, do this else do that.

We've lost all the information indicating where the values are physically stored. The language hides such implementation details from us. We are now operating at a level where the hardware is abstracted away and the source code deals directly with the application's concepts and terminology.

All of the examples ultimately result in the same behaviour but the lower level languages include much more hardware-specific information. Sometimes this is useful (particularly when performance is critical), but often we don't really care about minor details like which register holds a particular value. Greater abstraction allows high level languages to be more expressive. The same behaviour can be expressed in less code at the cost of less control over the hardware. These considerations are usually in conflict. Very roughly speaking, high level languages optimise for speed of development and low level languages optimise for speed of execution. A small, low level language will offer optimal performance and control over the hardware but the lack of abstraction makes it more challenging and time consuming to write large applications. High level languages make it easier for the developer to express their intent concisely but cannot offer the same level of performance and control. Some languages, such as C++ and Rust, make a brave attempt to have their cake and eat it by providing sophisticated abstractions while still allowing nearly full control over the hardware. These solutions come at the cost of high complexity.

First class values

If programming languages offer varying levels of abstraction, the obvious next question is: what abstractions are available? To answer this, we must look at what **first class values** the language has. By "first class", I mean that an instance of the value can be created, stored and passed around just like any other value. Basic types such as numbers and strings are almost always first class, though sometimes with quirks. For instance, JavaScript represents all numbers in 64-bit floating point. Until the introduction of `BigInt` in 2019, it simply wasn't possible to create an integer in JavaScript. Writing `x = 1` actually assigned the floating point `1.0` to `x`. We also saw in the concurrency chapter how promises made asynchronous values first class in JavaScript.

If you have ever programmed in JavaScript, you'll be accustomed to functions as first class values. In JavaScript, functions can be dynamically defined, passed around as arguments, stored in data structures, returned from other functions and so on. Let's imagine that your application has levels of debugging verbosity and you want to prefix each log output with its debug level. One approach would be to create functions for each debug level:

```
function logger(level) {
  return function(message) {
    console.log("[%s]: %s", level, message);
  }
}
warn = logger("WARN");
error = logger("ERROR");
warn("memory running low"); // [WARN]: memory running low
error("value out of bounds"); //[ERROR]: value out of bounds
```

We are doing several magical things in this little bit of code. `logger` is a function that dynamically creates a new function, using the provided `level` argument, every time it's called. The returned function expects a `message` and logs it with the warning level. JavaScript uses function scope, which means that every variable in `logger` is visible to the returned function. The combination of a function and its enclosing environment is known as a **closure** and is a very powerful concept. It enables a function to carry around its own execution environment. In the example above, two logging functions are generated and assigned to `warn` and `error`, each with a different value for `level` in their execution context.

If, like me, you began programming by learning JavaScript, all of this might seen very unremarkable. Yet it's surprisingly hard to do in many languages. In C, you can't easily dynamically generate functions or pass them as arguments to functions. All you can do is pass around a pointer to the function. It is not possible to write a function that creates and returns a new function as we do in `logger` above. Even Ruby, focused as it is on increasing developer joy, makes this tricky. In Ruby, referring to a function is the same as calling it, so passing around a function without calling it requires wrapping it in an executable object known as a `Proc`:

```ruby
def logger(level)
  Proc.new{ |msg|  print "[#{level}]: #{msg}" }
end

warning = logger("WARN")
warning.call("I've got a bad feeling") # [WARN]: I've got a bad feeling
```

Note that `logger`, which is a function, is called differently to `warning`, which is a `Proc`. That distinguishes the `Proc` from a true function.

State management

Most languages have the concept of a **variable**: a named memory location that holds a value. The program's **state** is the collection of all its variables' values and the current execution point. This brings us to our next question: what mechanisms does the language provide for managing state?

Continuing with JavaScript examples, here we declare a variable called `user` and assign to it an object:

```javascript
let user = { firstName: "Tom", surname: "Johnson", isAdmin: false };
```

We have associated the identifier `user` with the object. The memory location assigned to `user` holds the bytes of the object. We can query the object to determine its state. For now, `user.isAdmin` returns `false`. So far, so simple. Yet as the word "variable" implies, `user` can vary. When we modify the variable, things get complicated:

```javascript
let user = { firstName: "Tom", surname: "Johnson", isAdmin: false };
user.isAdmin = true;
```

Now the same identifier is associated with two different objects at two different points in time: one in which `isAdmin` is false and one in which it is true. The question "is the user an admin?" becomes difficult to answer because the answer depends on *when* we ask it. When state can vary, we say that it is **mutable**. To reason correctly about mutable state, we need to include time in our model. The value of a variable may change between two time points. Mutable state aligns with how the underlying hardware works (you can write to a memory cell as much as you like) and so most programming languages allow mutable state. The disadvantage of mutable state is that it can lead to confusing bugs, particularly if the state is shared between multiple threads (see the concurrency chapter).

Immutability

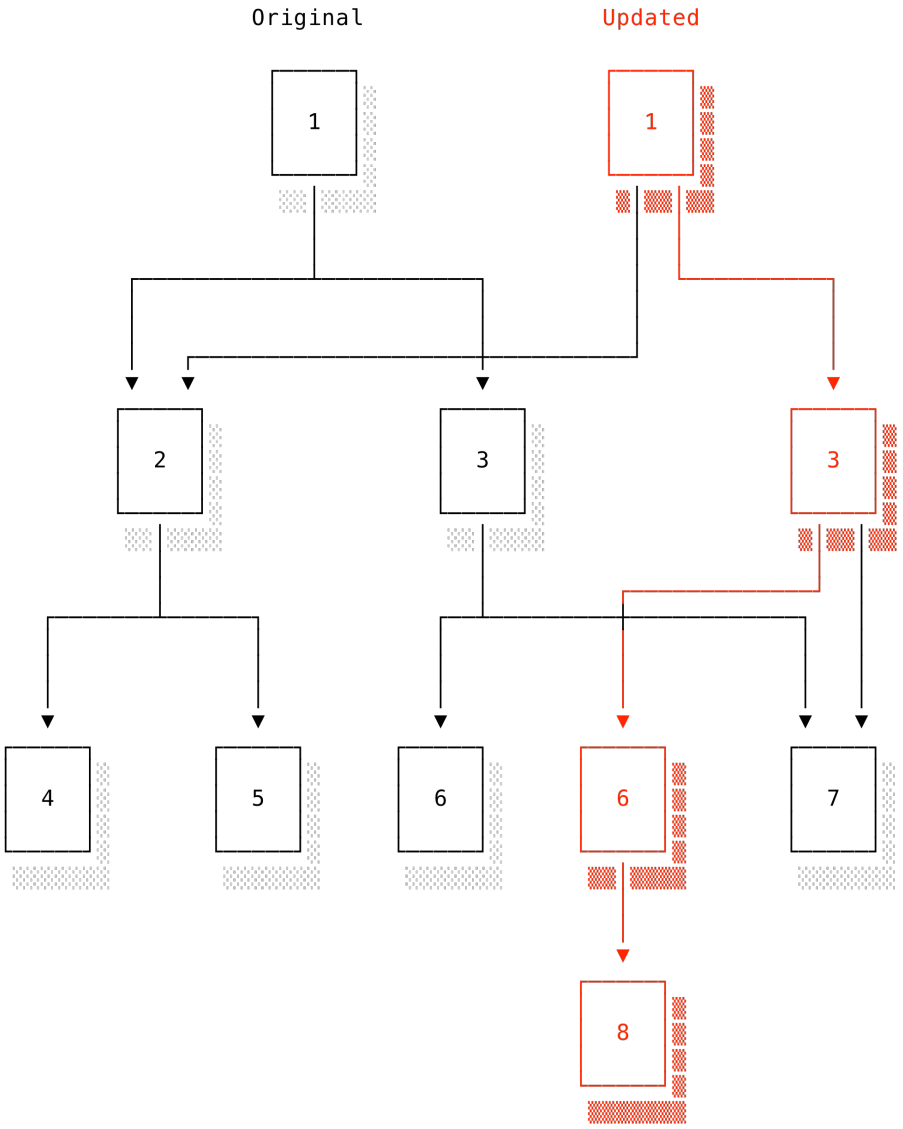
One way to avoid shared, mutable state is to prefer **immutability**. In languages with immutable state, it is not possible to change the value associated with a particular identifier. To perform a modification, you need to create a new value with the desired changes. JavaScript is by default mutable and offers opt-in immutability with the `const` keyword:

```
const name = "Tom";
name = "Timmy"; // TypeError: invalid assignment to const `name'
```

The problem with opt-in immutability is that it's easy to forget. Then you lose the guarantee that a variable will maintain the expected value. Immutability by default is a more extreme approach that's pretty popular in functional programming languages (discussed in more detail below). To them, mutable state is an abomination that should be expunged or, at the very least, used as little as possible.

What we have here are different answers to the old philosophical problem. How does a river remain the same "thing" when its contents and shape are constantly changing? In a language with mutable state, the value associated with an identifier can change. There is only one river but it varies over time. In languages with immutable state, the identifier's value can't be modified. Each change results in a new object.

Clojure is particularly interesting in this regard because it is inspired by Alan Whitehead's process philosophy. Clojure is a functional, immutable-by-default language. Immutability entails a performance cost because it is more expensive to create an entirely new variable than to modify an existing one. Imagine you had an array of one million elements and wanted to append a new one. It would be madness to fully duplicate the entire array just to add one element. To avoid this, Clojure uses "persistent" data structures. Modifications are recorded as incremental changes against the previous version, similar to how version control systems record a series of diffs against an original. Persistent data structures record a single entity with different values at different points in time. In this world view, our river is actually a sequence of different rivers that we can access by sliding time back and forth.

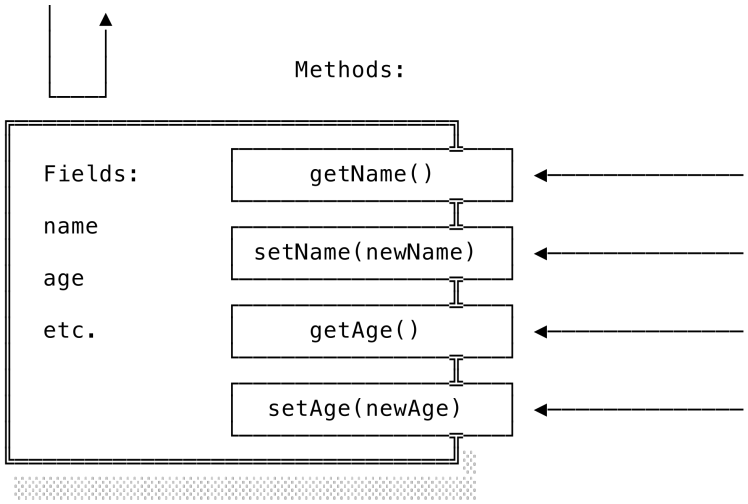


A persistent binary tree

Here we have a tree modified to hold an extra element. The original tree on the left has seven nodes. Each node holds a value and pointer(s) to children. When we modify the tree by adding a new value, the parent node needs to be updated to hold the new pointer. We must create a new version of the parent with a pointer to the new node. Since the parent has been updated, we must update its parent and so on recursively up the tree until we hit the root. The red nodes in the diagram are the new versions. Where the nodes have not been modified, the new tree can reuse the old nodes. Note that the updated tree still makes use of the whole sub-tree with root `2`. By making only the minimum necessary changes, the persistent data structure offers efficient immutability.

Objects

A second solution to shared, mutable state is to keep the mutability but avoid sharing as much as possible. We greatly reduce the occurrence of confusing bugs if we can hide mutable state away and reduce the amount of code that can directly interact with it. The combination of state and associated behaviour is known as an **object**. Objects hold their state (mutable or immutable) in private variables known as **fields**. Only procedures belonging to the object, known as **methods**, are allowed to directly interact with the fields. The object can choose which fields and methods to make accessible to the outside world. The rest of the codebase can only interact with the object through this public **interface**:



An object

The object presents an impermeable barrier to the outside world. The methods on the right straddle the barrier. The only way to access the object's state is through the interface the object chooses to offer via its methods. At the top left, we see a rebuffed attempt to bypass the methods' interface. Instead of having state open to arbitrary mutation, the object controls how its internal state can be accessed and modified. I'll have much more to say about objects when we look at object-oriented programming below.

Memory management

We finish up with a related question: how does the language manage programs' memory usage? In all but the most trivial programs, it will be necessary to allocate memory at runtime. A web server, for example, will need to allocate memory for each request it receives. The programmer cannot possibly know how much space to allocate in advance, so it must be done while the program is running. Once the memory is no longer needed (perhaps because the server has finished handling the request), it must be returned to the pool of available memory. Otherwise, the program will gradually allocate more and more memory until the system runs out of memory.

Memory management is a distinguishing characteristic of programming languages. At one extreme

we have (again) C, where the programmer is responsible for manually managing the dynamic allocation of heap memory (refer to chapter three if you've forgotten that term). The C standard library provides a couple of helper functions to request memory blocks and release them when no longer needed (`malloc` and `free`, respectively). Other than that, you're on your own. "Call `malloc` when you need memory and `free` when you're done" sounds straightforward but it's very easy to get wrong, especially in a large program. For this reason, many languages implement some kind of automatic memory management. When you declare a variable in Go, for example, the compiler decides whether to store it on the stack or the heap, allocating and freeing memory as required.

Automatic memory management often uses **garbage collection**. The language runtime keeps track of which resources are still in use and marks unused ones as "garbage" to be collected and returned to the heap. A simple technique is **reference counting**, in which the runtime tracks how many references there are to each resource. Once there are no more references to a resource, it is marked as garbage. Another commonly used algorithm is **mark and sweep**. Starting from one or more root resources, the garbage collector follows every reference to other resources, marking each one as *reachable* as it goes. Once it has marked every reachable resource, any remaining resources are unreachable and can be swept away.

Commonly, the language runtime will periodically pause normal execution to perform garbage collection. This obviously implies a performance cost. Worse still, the occurrence of these "stop the world" pauses is non-deterministic and not something the program can predict or manage. For these reasons, languages with automatic memory management are rarely used when predictable performance is essential (e.g. operating systems). Rust is a relatively new language that uses its type system to provide a form of compile-time memory management with zero runtime cost.

Programming paradigms

The questions we've considered help us to determine the **programming paradigms** a language supports. In broad terms, a paradigm represents a world view, a particular set of concepts with which the programmer can express their intent. It's common for a language to focus primarily on one paradigm, though some attempt to be genuinely multi-paradigm.

Imperative programming

In the **imperative programming** model, the program is an ordered sequence of statements. The programmer writes instructions to explicitly control execution flow. Each statement may evaluate expressions and perform side effects, such as modifying a variable in memory. Imperative code is so called because it instructs the computer *how* it should generate the desired result. Imagine the programmer as an all-powerful Roman emperor sending out detailed commands. At its core, imperative programming is conceptually quite simple because it maps closely to how the hardware executes the underlying machine code. Assembly languages are good examples of simple imperative languages. Most mainstream programming languages are imperative at heart.

Procedural programming

The first innovation within imperative programming was the invention of the **procedure**, also known as the function or subroutine. Procedures allow related instructions to be encapsulated into reusable chunks. They provide modularity because everything that the procedure does occurs within the procedure's **scope**, which is not visible from outside of the procedure. Scoping and procedures thus help to compartmentalise code into logical units with minimal data sharing. Code that calls a procedure doesn't need to know anything about how the procedure functions internally. It only needs to know the procedure's interface: the arguments it accepts and its return value. Overall, procedures mean that less state is globally accessible and help the programmer to structure their program as blocks of reusable functionality.

Procedural languages arose around the end of the 1950s / early 1960s with languages such as FORTRAN and COBOL. Although they are both still used, by far the most popular procedural language is C (1972). As I mentioned above, C was designed to be close to the hardware but easily portable across systems. The combination of portability, speed and hardware control makes it still hugely popular for "systems" programming tasks, which include operating systems, compilers, servers and so on. C is also closely associated with the Unix operating system and its descendants. The Linux kernel is written in C.

A simple but complete C program looks like this:

```c
#include <stdio.h>

int find_max(int arr[], int count) {
  int i, max;

  max = arr[0];

  for (i = 1; i < count; i++) {
    if (arr[i] > max) {
      max = arr[i];
    }
  }

  return max;
}

int main() {
  int arr[] = {2, 84, 32, 11, -70, 199};
  int n = sizeof(arr) / sizeof(arr[0]);

  printf("Largest element is %d\n", find_max(arr, n));

```

```
23    return 0;
24  }
25  // Prints: Largest element is 199
```

Without going through the code line-by-line, I'd like you to note that the "find the max element in this array" functionality is abstracted away into into the `find_max` function. The function's definition tells us that it takes an `int` array and an `int` as arguments and returns another `int`. The variables `i` and `max` are local to the function and so are not visible outside of the function. They exist in the function's stack frame, which will be destroyed when the function returns. The value of `max` will be copied and returned. It would be easy to change the implementation of `find_max` without having to change any other code, so long as it still fulfilled the interface.

You might be wondering why we calculate `n` (the number of elements) and pass it to `find_max`. That's because arrays are not really first class values in C. When we pass `arr` to `find_max`, all that actually happens is that `find_max` receives a pointer to the first element in the array. The array doesn't record its own length and so with just a pointer to the first element, we don't know how long the array is. We need to work out the length as the byte size of the array divided by the byte size of a single element and then pass it around in a separate variable. Modifying the array without updating the size variable, and vice versa, is a common source of bugs in C. That's why later languages generally offer the "smart" arrays we saw in the data structures chapter as first class values. Note too that the state, the array, and the operation on the state, `find_max`, are conceptually and physically separate.

Object-oriented programming

Object-oriented programming (OOP) develops the ideas of modularity and encapsulation a step further by moving state into objects. As described previously, an object is a bundle of private state coupled with methods that operate on that state. Objects were first popularised by Alan Kay in the Smalltalk language (1970s) but really had their heyday in the 1990s (along with Nirvana and anti-globalisation protests). That era saw the rise of C++, Java and then later Ruby and Python. All are object-oriented. Nowadays, the majority of languages in widespread use include at least some support for OOP.

Most object-oriented languages use **classes** to define objects. A class is a template for generating objects of a particular type. Think of how a cookie cutter stamps out cookies of an identical shape. A constructor function creates objects that are **instances** of the class. Classes are used to model the entities in a program. Their fields hold their private state and their methods define how they interact with each other. Methods are special functions with an extra argument, normally called `self` or `this`, that provides access to the object's private properties. Usually, the extra argument is implicit although Python explicitly requires it: `def my_method(self, my_arg)`. By only allowing access to its private state via this extra argument, the object ensures that only its own methods have unrestricted access to its state. This is known as **encapsulation**. Here's an example using a simplified Java:

```java
public class User {
  private LocalDate dateOfBirth;

  // Constructor
  public User(LocalDate dateOfBirth) {
    this.dateOfBirth = dateOfBirth;
  }

  public boolean canDrink() {
    LocalDate today = LocalDate.now();
    Period age = Period.between(this.dateOfBirth, today);
    return age.getYears() >= 18;
  }
}

User user = new User(LocalDate.of(2010, Month.MAY, 20));
user.canDrink() // false
user.dateOfBirth = LocalDate.of(1990, Month.MAY, 20); // ERROR
```

This Java class declares its `dateOfBirth` property to be `private` so that it cannot be accessed from outside of the class. The `canDrink` method uses the private state to compute whether the user is of legal drinking age. There is no way for code elsewhere in the program to circumvent this restriction by modifying the date of birth. The class has thus enforced its desired behaviour.

Object-oriented programming is not just about managing mutable state. Its proponents argue that it also helps to make software more modular and more easily extensible. At first glance, it doesn't really seem to make much difference whether you stringify a variable by calling a method `variable.toString()` or by calling a function `toString(variable)`. Think what would happen, though, if we added a new data type and wanted to be able to stringify it. If we used a single `toString()` function, we'd need to modify `toString()`'s implementation to support that new data type. Adding a new data type therefore requires changes to other parts of the codebase to support the new data type. In object-oriented programming, we would define a new class for the data type and implement all of its functionality there. The rest of the codebase does not need to change to support the new data type. It can just call the `.toString()` method and rely on the object to provide a suitable implementation. This is known as **polymorphism** and is a central concept in object-oriented design.

In its purest form, object-oriented programming is more than just "programming with objects". An analogy with biological cells is useful. Imagine an object as a cell that has engulfed some tasty data molecules. The cell membrane acts as an interface between those data molecules and the outside world, controlling what may enter and leave the cell interior. It is not possible to access the cell's internal molecules without going through the membrane interface. Similarly, an object's fields are only accessible from within the object's scope. We can't pry into the object's internal structure or state. We can only observe how it chooses to respond to messages. Cells communicate by releasing chemical messages. Objects pass messages by calling methods. When you see a method call like

`name.toString()`, think of it as passing the message (or command) "give me a string representation of yourself" to `name`. It's up to `name` to decide whether and how it wants to respond.

Each object is a self-contained mini-computer focused on a single responsibility. Complex programs can be broken down into lots of simpler mini-computers that only communicate through their method interfaces. Computation becomes more organic, occurring in lots of objects simultaneously. The biological analogy breaks down in reality, however. Real cells emit chemical messages without any knowledge of what other cells are in their vicinity to receive the messages. In order to send a message to an object, you need to have a reference to it so that you can call its method. In object-oriented programs, managing references to objects can become a complex design issue. Often programs end up using some kind of central holding object, very unlike the decentralised biological analogy.

Critics argue that object-oriented programming thus fails to deliver on its promises. Proponents counter that, just like communism, the idea is fine but it has never been implemented properly in reality. Popular object-oriented languages don't do it right and that's why things don't always work out. While perhaps true, this does make you wonder why object-oriented programming is so hard to implement properly in the first place. The best example of actually existing, object-oriented design is, surprisingly, the Internet. Servers are objects with private, internal state that only communicate via public interfaces. Servers broadcast messages with only limited knowledge of which neighbouring servers will receive and process their messages.

When you write object-oriented code, it's easy to fall into the trap of thinking that objects are somehow special. Looking at the Java example above, we have that special `class` keyword and special handling of methods so that `this` is implicitly available. Pretty much all Java code has to live within a class. But don't let the syntax mislead you. Here's the Java class above recreated in JavaScript using functions and closures:

```
function User(dateOfBirth) {
  let _dateOfBirth = dateOfBirth;

  function canDrink() {
    let today = moment.now();
    return _dateOfBirth.diff(today, "years") >= 18;
  }

  return {
    canDrink: canDrink
  }
}

let user = User(moment("2010-5-20"));
user.canDrink(); // false
user._dateOfBirth = moment("1990-5-20"); // ERROR
```

`_dateOfBirth` is declared within the scope of `User` and so it is not accessible from outside of `User`. It is hidden, just like a private class field would be. `User` defines `canDrink`, which has access to `_dateOfBirth`, and makes it publicly accessible by returning it in an object. The `user` variable holds the returned object, so consuming code can call `user.canDrink()` like a method. It's interesting to see that objects and closures, though on the surface appearing to be very different, are actually very similar on a deeper, semantic level.

Declarative programming

Recall that an imperative program is an ordered sequence of statements. In the declarative programming paradigm, the programmer declares the results they want and lets the computer figure out how to compute it. Imperative programming says *how* to generate the result, declarative programming says *what* the result should be.

Declarative languages tend to avoid explicit control flow. After all, we're not telling the computer what to do. As we're only describing the desired result of the computation, declarative languages also tend to avoid side effects such as mutating state. A consequence of this is that they often prefer to use **recursion**, a function calling itself, instead of loops and mutable counters.

Don't worry if declarative programming sounds a little esoteric. Most people learn programming in an imperative style and other approaches initially feel odd. In fact, you've already seen one declarative language: SQL. When we write a SQL query, we define the shape of the desired result and the database engine's query planner goes away and works out how to compute it. That's exactly how a declarative language works. Regular expressions are another example of declarative languages. An interesting branch of declarative programming is known as **logic programming**, in which programs are expressed as a series of logical statements and the computer tries to find a solution to the statements. Due to space constraints, I can't go into more depth here but I encourage you to check out Prolog.

A more common branch of declarative programming is **functional programming**. In this paradigm, a program consists of function application and composition. In other words, lots of functions calling lots of other functions. Haskell is a popular example of a functional programming language. A simple Haskell example serves to show the key concepts of declarative programming:

```
1  length [] = 0                -- empty list
2  length (x:xs) = 1 + length xs -- non-empty list
```

`length` calculates the length of a list. We provide two definitions. The first is for an empty list, which obviously will have a length of zero. In the second definition, we use `(x:xs)` to split the list into the head `x` and the tail `xs`. The length will be equal to the length of the tail plus one. The function calls itself recursively, each time producing a one and a shortened list, until it hits the base case of an empty list, at which point the recursion stops with a zero. The result is the sum of all of the ones. Note that there is no control flow. There is no code that checks whether the list is empty. We just declare what should happen in both cases and leave the computer to decide what to do.

The key feature of functional programming is that functions are first class values. This makes it possible to write a program as a composition of functions. Even non-functional languages can offer support for a functional "style" if they provide first class functions. As an example, JavaScript supports a functional style of programming by allowing functions to be passed to other functions:

```
1  [1, 2, 3].map(x => x * x); // [1, 4, 9]
```

Note that in this style we don't have to maintain a counter and iterate through the array. The `map` implementation handles all of that, only requiring us to plug in the function we want to execute at each step.

The more hardcore form of functional programming requires that functions are **pure**, meaning that they have no side effects. A pure function will always give the same output for a given input. `x => x * x` will always return `9` when passed `3`. This makes the functional expression easier to reason about because it can always be substituted for the value it evaluates to. Haskell is famous for requiring that functions are pure. Of course, reading user input, writing to the screen and other side effects are very useful. Haskell has to provide notoriously confusing mechanisms to contain impure behaviour within pure functions.

Popular functional languages include Haskell, Clojure and Ocaml. Not all of them demand pure functions in the way Haskell does but they do tend to minimise mutable state as far as possible. Proponents of functional programming see it as cleaner and more elegant, closer to a mathematical ideal, than imperative programming, which is hopelessly mired in the reality of mutable state. In recent years, functional programming has become increasingly popular as people find that minimising mutable state makes their programs easier to write, debug and reason about. A lack of shared, mutable state also makes concurrent programming much easier to do in functional languages.

Type systems

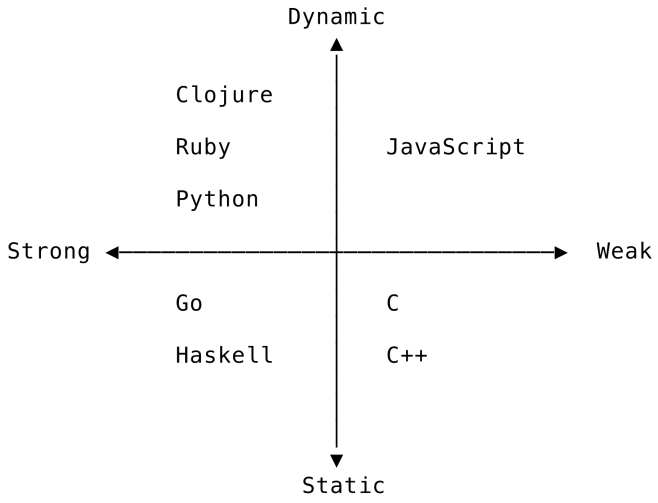
A **data type** defines a "shape" data can have. It defines the set of possible values and the operations that can be performed on those values. For example, the boolean type can be either `true` or `false` and it supports logical operations such as negation, conjunction and disjunction. The string type is the (infinite) set of character sequences and it supports operations like concatenation and substitution. A **type system** defines and enforces a set of **type rules** specifying how types are assigned to expressions and how types interact. Some types are built in to the language and normally there is some mechanism for user-defined types.

Type theory, the academic study of type systems, is one of those areas where computer science meets mathematics. In fact, there is a proven equivalence between computer programs and logical proofs. We'll look at the implications of this shortly, but for now look at the following type rule: $A \rightarrow B$. As a logical proof, this means "A implies B". As an expression in a typed programming language, it's a function that takes type A and returns type B. A concrete example would be `length :: string`

`-> int`. It takes a string and returns an integer. Types provide a more logically robust way to think about computer programs and their correctness.

Types are a powerful tool to express intent both to the computer and to other people. Type systems improve correctness by making explicit how parts of the system should interact with each other. That gives the computer more information with which it can detect errors. **Type checking** is the process of verifying that all of the constraints imposed by the type rules are met by the code. If the computer knows that `length` only accepts strings, it can catch bugs where we mistakenly pass a different type to `length`. The more type errors that a type system can detect, the more **type safe** it is. Some argue that by creating ever more sophisticated type systems, we can encode more and more information in the type system and thus gain more and more type safety. Others argue that complex type systems add unnecessary complication and get in the way of expressing application logic.

Every language has a type system, even if it's not very obvious or helpful. Type systems are categorised along two main axes: static versus dynamic type checking and strong versus weak typing. In this diagram, languages are categorised into quadrants by their type system:



Type systems along two axes

Static and dynamic type checking

Type checking is the process of validating that every expression has a valid type according to the rules of the language's type system. If the program passes, it is type safe. Static type checking happens before the program runs and dynamic type checking happens while the program is running.

A statically typed language verifies that a program is type safe by analysing the source code before the program even runs, normally as part of the compilation process. An obvious implication is that the source code needs to contain enough information for the type checker to work out the types without having to run the code. This might be done by annotating a variable with its type in the source code. In Go, the type comes after the variable name:

```
var message string = "I'm a string!"
```

This is redundant though, right? It's obvious that the value is a string literal, so we shouldn't really need to specify it. The process of automatically determining the type of an expression is known as **type inference**. A type system with inference doesn't require so many manual annotations because the type checker can determine the type of many expressions from context. Go supports type inference using what is known as "short declaration syntax":

```
message := "I'm a message!"
```

Static type checking provides assurance that the types of expressions match our expectations. A function's type signature documents what it accepts and what it returns. In the following example, `welcomeShout` defines its argument to be a string. The last line contains a type error and so the program will not even compile:

```
import (
        "fmt"
        "strings"
)

func welcomeShout(name string) {
        fmt.Printf("Welcome, %s!\n", Strings.ToUpper(name))
}

func main() {
        welcomeShout("Tom") // Welcome, TOM!
        welcomeShout(42) // Compiler error
}
```

In a dynamically typed language, type safety is only verified at runtime. A common implementation is that each variable has a type tag containing type information. When an expression is evaluated, the runtime first checks that the operands' types are compatible with the type system's rules. The problem with dynamic typing is that we can't check in advance whether the operands' types are correct. We can only run the code and see if it works:

```
function welcomeShout(name) {
    console.log(`Welcome, ${name.toUpperCase()}!`);
}
```

In this equivalent JavaScript code, the function implicitly expects `name` to be a string (or another type that responds to `toUpperCase`), but it has no guarantee that the argument is the correct type. If `name` doesn't respond to `toUpperCase`, the code will throw a runtime exception and probably break

the program. Applications in dynamically typed languages require extensive test suites to try and cover every code path. Even with a huge test suite, it is impossible to be completely confident that every broken code path has been found. In a statically typed language, by contrast, the compiler will detect the broken code paths caused by type errors and will refuse to run the program until they are fixed. That can be frustrating when you just want to run a small section of the code or try out an idea and you know that the type errors aren't immediately problematic. The advantage is that once it finally does run, you can be more confident that the code will work.

The main benefit of dynamic typing is that it is easier to get started and try out ideas without the up front work of codifying lots of rules in the type system. It lends itself to a very flexible and permissive style of programming. For example, a log processor might iterate through a stream of logs. In Ruby:

```
class Parser
  def parse(input)
    input.each do |line|
      # ...processing here
    end
  end
end
```

We can pass absolutely anything to `#parse`, as long as it implements the `#each` method. In production, this might be some kind of sophisticated event stream. In testing, we could just use a simple array because Ruby arrays implement `#each`. We don't need to define in advance what types `#parse` is allowed to receive. Python programmers like to call this "duck typing" with the slightly grating explanation that "if it walks like a duck and quacks like a duck, it's a duck". We don't really care about the actual type of `input`. We only care that it implements `each`.

Interfaces are a way to bring the flexibility of duck typing into a statically typed language. An interface defines a specification, normally as a set of method or function signatures, that a type must implement to be considered an instance of the interface. By using interfaces, we only specify the behaviour we expect from a type. **Structural typing** is when a type automatically implements any matching interface. Go and TypeScript (a typed superset of JavaScript) both take this approach. **Nominal typing** requires a type to explicitly declare the interfaces it implements. It isn't enough to just fulfil the interface's specification. Neither approach is better or worse. Structural typing allows for a statically checked version of duck typing while nominal typing better conveys intent.

In Go, the standard library's `Stringer` interface is for types that offer a string representation of themselves. We could rewrite `welcomeShout` to accept a much wider set of types by changing it to expect an interface:

```
import (
        "fmt"
        "strings"
)

type Stringer interface {
    String() string
}

type User struct {
    name string
    id uint
}

func (u User) String() string {
    return fmt.Sprintf("%d: %s", u.id, u.name)
}

func welcomePrint(name Stringer) {
        fmt.Printf("Welcome, %s!\n", strings.ToUpper(name.String()))
}

func main() {
        welcomePrint(User{id: 42, name: "Tom"})
}
// Welcome, 42: TOM!
```

Here I've redefined `Stringer` to show you what an interface's definition looks like. We define a new type, `User`, that has a `String()` method and so implements the `Stringer` interface. `welcomePrint` is amended to accept an argument of type `Stringer`. We have the (statically checked) assurance that any type passed to `welcomePrint` will provide the necessary `String()` method and the freedom to pass in any suitable type of our choosing.

In languages with interfaces, it's good practice to type function arguments as interfaces, rather than concrete types. That gives the function user the freedom to choose the most suitable concrete type for their use case. Interfaces allow us to define new functionality at the type level. For example, `Stringer` creates the concept of "something that has a string representation". Using interfaces allows functions to define the functionality they require without unduly constraining their callers.

To recap, in a static type system, type checking is performed before the program runs, using some combination of manual type annotation and automatic type inference. It catches many bugs before the program even runs and helps to document intent. Dynamic typing defers type checking until runtime. That makes it easier to quickly write flexible, reusable code but creates many opportunities for runtime bugs.

Strong and weak type safety

The second axis of type systems reflects how much type safety the system provides. A stronger type system will have stricter type rules and catch more errors, presenting them as compilation errors or runtime exceptions depending on when the type checking happens. A weaker type system will permit more ill-typed expressions and contain more "loopholes" that allow the programmer to subvert the type system. Obviously that means it will catch fewer mistakes. It may even implicitly convert an invalid type into a valid one in an attempt to avoid type errors. "Strong" and "weak" are not very precisely defined terms. Type safety is more of a spectrum than a binary definition.

C is a weakly typed language with static checking. Variables are typed, but types in C serve mostly to indicate how much space the compiler should allocate for the value. C has weak type safety because we can tell C to reinterpret a variable's bit pattern as a different type by taking a pointer to the variable and *typecasting* it to a pointer to the new type. Imagine that you are studying floating point numbers and want to examine the bit arrangement of a floating point number. You can do this by typecasting a pointer to the variable to a pointer to a byte:

```
1  #include "stdio.h"
2
3  int main() {
4    double value = 0.63;
5    unsigned char *p = (unsigned char *)&value;
6
7    for (int i = 0; i < sizeof(double); i++) {
8        printf("%x ", p[i]);
9    }
10 }
11 // Output: 29 5c 8f c2 f5 28 e4 3f
```

`double` is C's name for a floating point number (normally eight bytes wide) and `unsigned char` is a byte value. The syntax is a little funky but on line five, reading from the right, we use `&` to make a pointer to `value` and then cast it using `(unsigned char *)` from a `double` pointer to an `unsigned char` pointer. If we take the hexadecimal output and interpret it as a floating point number, we get `0.63`: back to where we started. At no point does the underlying sequence of bits in `value` change. We only instruct the computer to interpret those bits as two different types, thus generating two different outputs.

Typecasting is usually avoided because it weakens the assurances type checking provides but it can be helpful when working very close to the hardware. Much of C's design is based on the principle that the programmer knows what they are doing and shouldn't be impeded by the language. That's why it allows the use of pointers to arbitrarily manipulate memory and typecasting as an escape hatch in its type system. C's type system will statically check the types you specify but allows you to deliberately step outside the type system's protection when you deem it necessary.

JavaScript's type system is widely considered to be an unholy mess because it combines weak typing with dynamic checking. Not only are types not checked until runtime, but, in an attempt to avoid type errors, JavaScript will silently change, or *coerce*, an ill-typed value into the expected type, almost certainly resulting in unexpected output. It's not so much a type system with an escape hatch as a giant sieve. Due to the silent coercion, type errors propagate unseen, eventually popping up in the form of bizarre bugs far from the original error site.

Yet even in this madness there is reason. Remember that JavaScript's original purpose was to add a bit of sparkle to static websites. JavaScript content was expected to be very much secondary to the rest of the page's content. JavaScript therefore tries to be resilient so that a JavaScript bug doesn't stop the whole page from rendering. In that context, when faced with a mis-typed operation such as `1 + "2"`, it is better to coerce one operand to the expected type and carry on. In this case, JavaScript cannot add a string to a number so it implicitly converts the number to its string representation. Implicit coercion allows JavaScript to produce something vaguely useful instead of just throwing a runtime exception and crashing the page. Admittedly, `1 + "2" = "12"` might not be what you intended. I like to think of JavaScript as a really, really keen puppy that just wants to play even when it doesn't really know how to:

```
> 4 + 4 // Add two numbers? Sure!!!
8
> "hello " + "there" // Add strings? I'll concatenate them!! Yes!
"hello there"
> "hello " + 5 // Add a string and a number? Hmm... wait I know!!!!
"hello 5"
> "" + {} // You must want the object as a string!! Look!!!!
"[object Object]"
> {} + "" // No problem!!!!!
0
```

Okay, it gets carried away sometimes. Clearly, JavaScript's weak typing rules can lead to some weird outputs. If those weird outputs happen to be valid inputs for another function, you won't notice the error until another part of the codebase starts producing even weirder output.

Ruby's type system is stronger than JavaScript's because its rules don't allow sneaky workarounds like implicit coercion. Its type rules catch more errors and have fewer escape hatches. If you try something ill-typed, the interpreter will point out that you're trying to do some damn fool thing and refuse to proceed:

```
> 1 + "2"
# TypeError (String can't be coerced into Integer)
```

Haskell is the poster child for strong typing because it has a sophisticated type system that allows more of an application's rules and behaviour to be encoded as types. As a simple example, many languages have a `null` type to represent the absence of a value. In some weaker type systems, a typed

value can also be `null`, leading to type errors if the programmer forgets to check for `null`. Java is notorious for this because it allows any reference to an object to be `null`. Haskell provides a type, known as `Maybe`, that encodes a possibly `null` value. Here is an example using numbers:

```
data Maybe Number = Just Number | Nothing
```

`Just` is a kind of "wrapper" type around the number. `Nothing` is Haskell's way of representing `null`. I have slightly simplified this code to avoid getting caught up in unnecessary complexities in how Haskell handles numbers.

Let's see how we might use this type to catch bugs. In floating point calculations, the value `NaN` (not-a-number) represents undefined or unrepresentable values. Haskell returns `NaN` when you try to take the square root of a negative number, since that's an invalid operation. Any arithmetic with `NaN` generates more `NaN` values: `NaN + 2 = NaN`. Awkwardly, `NaN` is still a numeric *type*, even if it's not a valid value, so a function expecting a number will accept `NaN`. It's super easy to forget to check for `NaN` and produce invalid calculations. We can write a safe version of `sqrt` that return a `Maybe Number`:

```
safeSqrt :: Number -> Maybe Number
safeSqrt n | n >= 0 = Just $ sqrt n
           | otherwise = Nothing
```

We declaratively define two implementations for `safeSqrt` and let the computer select the correct one based on whether `n >= 0`. If the input is valid, we compute the square root and wrap it in a `Just`. Otherwise, we return an empty `Nothing`. The joy of `Maybe` is that the Haskell compiler will error if we don't handle both possibilities like so:

```
printOutput n = case safeSqrt n of
  Just x -> putStrLn $ "sqrt of " ++ show n ++ " is " ++ show x
  Nothing -> putStrLn $ "invalid input " ++ show n
```

In Haskell circles there is a popular refrain to "make illegal states unrepresentable". The aim is to encode the business logic of the application into the type system so that code expressing incorrect business logic would generate a type error and fail to compile.

That approach builds on an important theoretical result in computer science known as the **Curry-Howard isomorphism**. It states that type systems are equivalent to mathematical proofs. A type corresponds to a logical proposition and if you can provide an instance of the type then you have proved that the type is *inhabited* and the proposition is true. What the Curry-Howard isomorphism tells us is that in writing correctly typed programs we are proving theorems about the behaviour of our programs. In theory, we could prove any behaviour that could be encoded as a proof. In practice, we are limited by the expressiveness and power of our languages' type systems. An active area of programming language research is to create much more logically sophisticated type systems so that more complex behaviour can be defined and enforced by the type system.

Conclusion

Programming languages are defined by a specification consisting of the language's syntax and semantics. The format of a specification might range from an informally defined reference implementation right up to an international standards committee. Syntax defines what is considered valid code. Semantics define how valid code should behave.

Languages are distinguished by the abstractions they offer the programmer over the basic hardware primitives. Low level languages provide more control over the hardware but offer fewer tools for building complex applications. High level languages focus on developer productivity, portability and expressiveness at the cost of hardware control and speed. How languages control state and memory management influence the abstractions they offer.

Imperative programming languages see programs as a sequence of computational instructions. Declarative programming languages see programs as a specification of the desired computational output. Both paradigms are well represented by modern programming languages, especially by object-oriented programming and function programming respectively.

Type systems are a mechanism for defining and enforcing semantics. Type checking is the process of ensuring that a computer program behaves in accordance with the type system. This can either be performed statically, before the program is executed, or dynamically at runtime. Type systems vary in how strongly their rules restrict the behaviour of programs. Strong, static type systems are favoured when improved runtime correctness is important. Dynamic type checking and weak type systems can be useful when flexibility and ease of development are more important.

Further reading

To improve your understanding of programming languages and their paradigms, not to mention become a better programmer, by far the most useful (and fun!) thing you can do is to learn more programming languages. Pick up something outside of your comfort zone. It doesn't really matter what, so long as it's very different to what you already know. Elm, Elixir, Prolog, Clojure or Haskell will each blow your mind. Even a little exploration will be enough to broaden your perspective. By comparing and contrasting multiple languages, you'll gain a deeper understanding of how more familiar languages work and pick up idioms and techniques that you can apply everywhere.

I highly recommend *Structure and Interpretation of Computer Programs* by Abelson and Sussman. It's one of the most respected textbooks in computer science and for good reason. It takes you through the design and implementation of a programming language, gradually adding more and more capabilities. You will have a much deeper understanding of things like evaluation, assignment and mutable state after implementing them yourself. It is one of those textbooks where the real learning comes from doing the exercises, although don't expect to be able to do them all on your first reading. You might also be interested in *The Little Schemer* by Friedman, the first in a series of books covering recursion, functional programming, logic programming and fancy type systems.

If you're looking for an overview of many languages, The A-Z of programming languages[50] is a series of interviews with the creators of dozens of languages. The interview quality is admittedly variable but there are some fascinating insights into how the designers invented their languages and developed their ideas over time. The importance of community is a recurring theme.

If type theory sounds interesting, a good starting point is Type Systems by Luca Cardelli[51]. It's fairly short and includes an extensive bibliography. If you're very keen, *Types and Programming Languages* by Benjamin Pierce provides a comprehensive overview of the field.

[50] http://www.math.bas.bg/bantchev/misc/az.pdf
[51] http://lucacardelli.name/Papers/TypeSystems.pdf

Databases

Introduction

There comes a point in every budding developer's life when they need to use a database for the first time. The function of a database – reliably storing and retrieving data – is so fundamental that it's hard to do much programming without encountering one. Yet databases sometimes seem a little unfamiliar, a little *other*. The database is separate to your code and interacting with it often requires using a different language, SQL. It's tempting to learn the bare minimum and stick to what you're comfortable with. Databases also have a reputation for being a little boring. Perhaps it's the old-fashioned syntax of SQL or all those tutorials about storing employee records.

In this chapter, I hope to convince you that databases are in fact deeply interesting and well worth engaging with properly. They're like a microcosm of computer science. Parsing a SQL query and generating an execution plan touches on programming language design and compilers. Databases rely on clever data structures to enable fast access to huge amounts of data. Sophisticated algorithms ensure correct concurrent behaviour and that data is never lost. The database is one of those applications that acts as a linchpin, drawing together everything that we've studied so far.

Databases come in a huge variety of shapes and sizes. I'm going to focus solely on **relational databases**, which are by far the most popular. We'll start with the theoretical underpinnings of relational databases. From there, we'll look at the four important guarantees that databases provide, which go by the funky acronym of ACID. The rest of the chapter will explore a typical database architecture. Due to space constraints, I'm going to assume that you have worked with a database before and can do basic SQL queries. Maybe you've even designed a schema for a project. But you don't yet know *why* or *how* it all works. If this is all new to you, worry not. Introductory resources are referenced in the further reading. We won't cover any advanced SQL tricks as I believe that they are easier to learn by solving real life problems as you encounter them.

Throughout this chapter, I'll make reference to two databases, Postgres and SQLite, depending on which seems like the better example for a given point. Both are relational databases and so are conceptually very similar but they have very different implementations. Postgres is a high performance database offering oodles of cutting edge features. It runs as a server that clients must connect to in order to perform queries and operations. SQLite, as the name suggests, is much more lightweight and stores the entire database in a single file. Its small footprint makes it a popular choice for embedding a database in other applications. Google Chrome uses SQLite to store internal data such as browsing history.

Finally, a point on terminology. The term "database" is overloaded. Technically, "database" refers to the collection of entities and their relations that make up the application's data. Programs like Postgres and SQLite are examples of **relational database management systems** (RDBMS). In this

chapter I'll use "database system", "database engine" or just "system" to refer to the program that manages the database.

What does a database offer?

Working with a database entails understanding a whole other programming paradigm, learning a new language and figuring out how to integrate the system into an existing application. Why go to all this bother when we can just store application data in a plain file?

Databases offer numerous benefits over a plain file. They provide assurances that your data is protected from loss and corruption. If you use plain files, you're responsible for writing all that code yourself: opening and parsing the file, making bug-free modifications, gracefully handling failures, catching all of the edge cases where data can get corrupted and so on and so on. Doing all this correctly is a huge task. Just as you probably don't want to be writing your own OS, you probably don't want to be writing your own data management layer. On top of protecting you from disaster, databases also offer a bunch of neat features. They allow concurrent access by multiple users, improving your application's performance. They offer a sophisticated query interface in the form of SQL, meaning you avoid having to hand-code every query. It'll also run those queries much faster than anything you could do yourself.

In short, database systems abstract away the gory details of data storage and management. They provide a query interface that, while sometimes a little clunky, is immensely powerful and creates a clear boundary between your data and your business logic. They provide important assurances so that you don't have to worry about data loss or corruption. Database systems have four properties that guarantee that the data they contain remains safe even in case of failures: atomicity, consistency, isolation and durability. Together they go by the acronym **ACID**. Let's look at each in turn:

Atomicity

Atoms were so called because they were believed to be indivisible. Of course, we now know that they're not indivisible after all – awkward! In computing terms, an operation or sequence of operations is *atomic* if it is either performed to completion or not performed at all. No halfway state is allowed, even if there's some kind of catastrophic power failure mid-way through. This is very important from the point of view of data safety because we never want to get into a situation where an update to the data is only partially performed.

The canonical example in the textbooks is transferring money between two accounts. Peter wants to pay Paul £100. This involves two steps. First, credit Paul's account with £100:

```
1 UPDATE accounts
2 SET balance = balance + 100
3 WHERE account_name = "Paul";
```

Then, debit Peter's account by £100:

```
UPDATE accounts
SET balance = balance - 100
WHERE account_name = "Peter";
```

Note that it doesn't matter which statement is executed first as long as they both execute. However, if an error were to occur after crediting Paul but before debiting Peter (perhaps the database system crashes or Peter doesn't have enough money in his account) then the system erroneously "creates" money by giving Paul money without taking it from Peter. A fortuitous outcome in this case, but if the statements were executed in a different order money would *disappear* from the database!

Of course, the system can't perform its operations instantaneously so there will inevitably be times when it is in the middle of performing some kind of modification to the database. How do we prevent a failure at this point from messing up the entire database? We introduce a concept known as a **transaction**:

```
BEGIN

UPDATE accounts
SET balance = balance + 100
WHERE account_name = "Paul";

UPDATE accounts
SET balance = balance - 100
WHERE account_name = "Peter";

COMMIT
```

The transaction is created by wrapping the steps in `BEGIN` and `COMMIT` statements. A transaction consists of a sequence of operations that must all succeed or all fail. We'll see later how a database system ensures that this happens, but at a high level the system simply doesn't consider a transaction to be completed, or *committed*, until all of its operations have been successfully committed. If one fails, the system performs a **rollback**, undoing all of the changes to return the database to its state before the transaction began.

Consistency

The consistency property ensures that the database always makes sense from the application's point of view. A table **schema** defines the table's attributes and their types. For example, a user may have a "salary" attribute, which must be a number. **Constraints** further restrict the possible values that an attribute can have. The "salary" attribute might be constrained to never contain a negative value because a salary can never be less than zero. If we gather all of the table schemas from a database, we have a set of expectations about the data. We have data safety when the data is consistent with

these expectations. Some expectations may span multiple entities. **Referential integrity** requires that a foreign key in one table refers to a valid primary key in another.

Very common constraints are to state that a column can't contain a `NULL` value or that every value must be unique:

```
CREATE TABLE products (
    product_no integer,
    name text NOT NULL,
    price numeric,
    UNIQUE (product_no)
);
```

In this example table, every product must have a name and its `product_no` must be unique. More sophisticated, custom constraints are also possible:

```
CREATE TABLE products (
    product_no integer,
    name text NOT NULL,
    price numeric CHECK (price > 0),
    UNIQUE (product_no)
);
```

Now the database will **abort**, or completely undo, any attempt to create or update a product with a negative price. This fits with our basic understanding of how prices work. It might sound obviously ridiculous to try to give a product a negative price, but it could happen quite easily if you're applying a series of discounts to a product and make a mistake in the calculations. Constraints don't mean that you don't need to do any validation or testing in your application code. You should still try to detect invalid data as soon as it's generated and prevent it ever getting near the database. Constraints are more like a last line of defence.

How much complex business logic you should encode in the database is a matter of debate. If there are multiple applications connecting to the same database – a web app, a mobile app etc – then it makes sense to write the logic once in the database rather than re-implementing it (and possibly introducing bugs) in each new client app. Sceptics retort that you'll end up with business logic spread through constraints and each client app, creating an unholy mess.

Isolation

Sometimes the database needs to move through intermediate, inconsistent states before it arrives at a new, consistent state. We rely on transactions to make this happen. Consistency guarantees that a transaction can always see the results of all previous, committed transactions. Isolation ensures that a transaction's intermediate states cannot be observed from outside of the transaction. As long as

the transaction ends in a consistent state, the rest of the database will only see a change from one consistent state to another.

Without isolation we could have situations like the following. Transaction A begins. Halfway through, when the database is in an inconsistent state, transaction B starts using the inconsistent state as its starting point. Even though transaction A eventually reaches a consistent state, the inconsistency leaks out to transaction B. Isolation ensures that the effects of one transaction do not become visible to another until the first has completed.

With proper concurrency controls, which we'll look at soon, multiple transactions can run concurrently and leave the database in the same state as if they'd run sequentially.

Durability

Finally, durability ensures that a change committed to the database is never lost, even in case of power failure. Of course, the database is limited by the hardware it's running on. It can't do much if the hard disk gets smashed to pieces!

The database system has to be sure that changes are written to persistent storage. It must play an interesting game with the operating system. As mentioned in the OS chapter, the OS will try to improve I/O performance by buffering writes to persistent storage. The sneaky OS might tell the database engine that it's written the data to disk when really all it's done is write the data to an in-memory buffer. If the OS then crashes, the data will be lost. It's a bit like if your boss had asked you to complete a task and you'd claimed that it was done when really you'd only asked a colleague to do it. *Hopefully* your colleague will do it correctly and everything works out fine, but maybe they'll get hit by a bus and suddenly your boss is wondering why that thing you said was done hadn't even been started.

The database has to be very careful to ensure that it **logs** important events to persistent storage so that they can be recovered in the event of system failure.

Relational algebra and SQL

Before we delve into the gnarly details of how databases work, it is important to understand a little bit of the theory that underpins relational databases.

Relational databases are so called because they rely on **relational algebra**. In common parlance, algebra is a set of operations – addition, subtraction and so on – that we can perform on numbers. A very important insight is that there are many algebras. The one we learn in school is just one instance of a more general concept. To make an algebra, all you need is a set of elements and a set of operations that can be performed on those elements. Relational algebra defines operations that can be performed on **relations**. It was developed as a model for databases by Edgar Codd in 1970 (see further reading). The idea proved wildly successful and relational databases are by far the most dominant database type.

A relation is a set of **tuples** matching a **schema**. The schema is a set of **attributes**, which are name and type pairs. A tuple is an ordered collection of elements corresponding to each attribute. The schema defines what entity the relation describes and the tuples are instances of the entity. It's an unwritten rule that every book on databases has to include an example modelling employees. This book is no different. Here is our example employee schema:

```
CREATE TABLE Employee(
  employee_id INTEGER PRIMARY KEY,
  name string,
  salary integer,
  department string NOT NULL);
```

SQLite, which I'm using for these examples, will autoincrement the `employee_id` column automatically because it is the primary key. For simplicity I'm treating the department as just a string. Here are the tuples in our example relation:

```
employee_id  name         salary      department
-----------  -----------  ----------  ----------
1            Amy Johnson  40000       Tech
2            Bob Wilson   30000       HR
3            Hafsa Hasad  35000       Sales
4            Anna Jones   25000       Marketing
```

`(2, "Bob Wilson", 30000, "HR")` is a tuple consisting of four values which together represent an individual employee. Note that the ordering of the tuple attributes is significant. The reason we know "Bob Wilson" refers to his name and not his department is due to the values' positions in the tuple.

In database systems, relations are implemented as **tables**. Each tuple in the table is known as a **row**. The schema attributes are **columns** in the table. Between rows there is no ordering. The database has no particular reason to put one employee above another. It's sorted by `employee_id` here but any other attribute is just as valid.

Tuples are uniquely identified by their values. This presents a problem for database systems. What if there are two John Smiths working in Accounts and both earning £45,000? We need to a way to disambiguate them. The solution is to create a **primary key**: a column, or combination of columns, that uniquely identifies a tuple. Sometimes the tuples contain an attribute that naturally works as a unique identifier. The `employee_id` plays that role here. If necessary, an additional, autoincrementing attribute can be added to the schema. With a primary key, we can disambiguate tuples that have otherwise identical attributes.

Even better, we can express the idea of a *relation* by allowing tuples in one relation to hold the primary key of tuples from a different relation. When a relation has an attribute holding the primary key of another relation, we refer to that attribute as a **foreign key**. A more realistic `Employee` relation

would have a separate `Department` relation and store the employee's department as a foreign key. This is how we can express relationships between relations. It's simple but very powerful.

SQL (Structured Query Language) is a programming language designed to express relational algebraic queries. It's a practical implementation of the mathematical ideas behind relational algebra. As we explore the operators of relational algebra, I'll show how they correspond to SQL expressions.

The two most important relational algebraic operations are **selection** and **projection**. Selection filters the relation's tuples based on whether they meet some criterion. In SQL, it's the `WHERE` clause that we all know and love:

```
WHERE Employee.salary > 32000
```

Once we have the desired tuples, we apply a projection to choose elements from the tuples. I like to think of it as shining a light from behind the tuple to project some of its values on to a wall or screen:

```
SELECT name
```

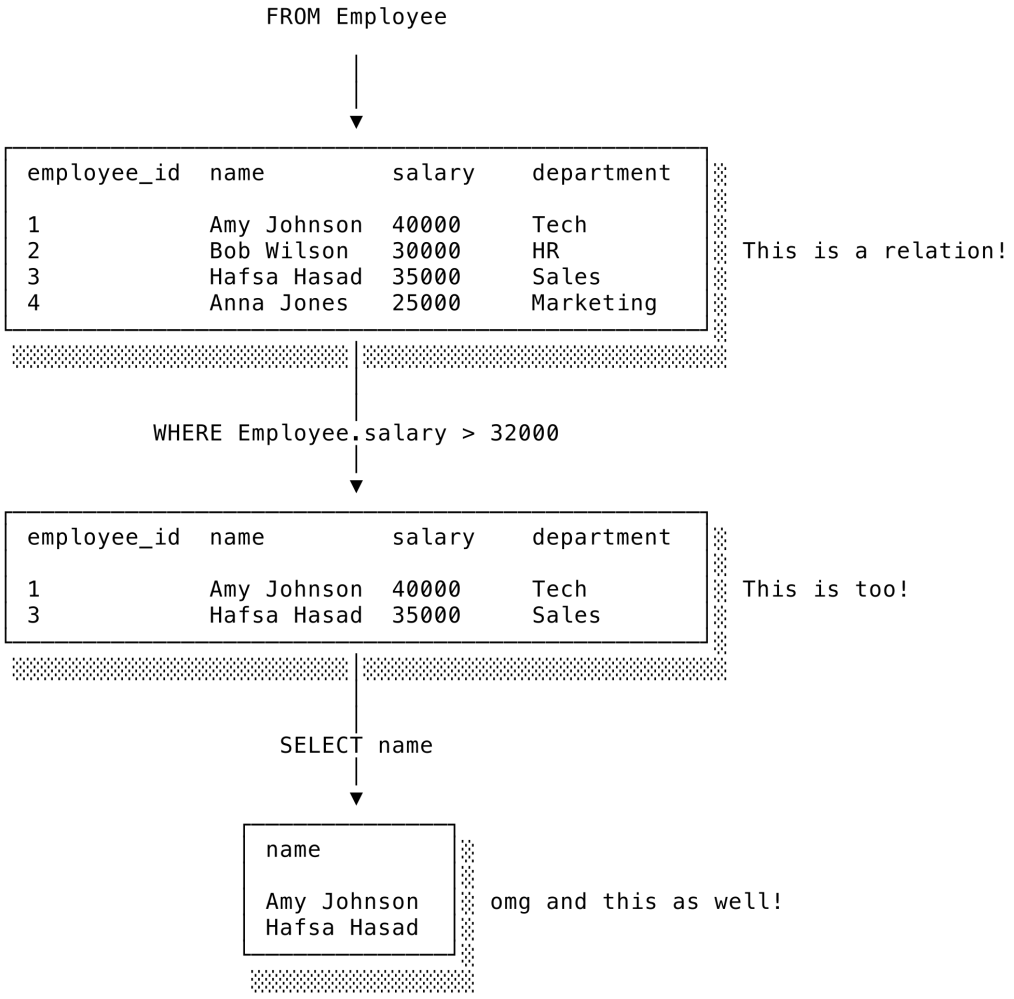
Remember: selection chooses the tuples from the relation and projection chooses the values from the tuples. Altogether our query is:

```
SELECT name
FROM Employee
WHERE Employee.salary > 32000
```

The output shows who the highly-paid employees are:

```
name
-----------
Amy Johnson
Hafsa Hasad
```

When working with relational databases, an important "woah, it's all connected, man" insight is that each relational algebraic operation takes relations as input and generates new relations as output. Those output relations then form the input for other operations, generating yet more relations. Look at how the above query might generate intermediate relations:



Operators take relations and output relations

The intermediate relations only exist in the abstract. The database system might not actually physically generate them if it can find a more efficient way to generate the correct result. In the example above, the system might determine that it can do everything in a single pass by extracting just the name whenever it finds a highly paid employee. This would be much more efficient than first creating a temporary table of highly paid employees and then iterating through that to get the names.

Such considerations are mere implementation details. When writing a query, it's best to think of operations on relations. It's much easier to express complex things in SQL when you think of terms of relational operations. It's very common, indeed good practice, to see queries forming a **sub-query** in a larger query. For example, the inner query below generates a relation containing the average employee salary and then feeds that to an outer query that finds every employee paid less than the average:

```
SELECT name, salary
FROM Employee
WHERE salary < (
  SELECT AVG(salary)
  FROM Employee
);
```

Another very useful operation is to take one relation and join it to another. A **natural join** takes two relations and tries to join the tuples together on matching attributes. Joining on a foreign key works in this way, since the foreign key of one tuple should match the primary key of a tuple in the other relation. A variation on this idea is to only join those tuples that match some predicate. Joining creates a new tuple with the attributes from both relations. It's the relational algebra equivalent of a logical AND operation (think back to the circuits of the architecture chapter). The natural join in SQL is known as an **inner join**:

Hafsa Hasad

Combining rows with joins

Here we see that the only `Department` tuple matching Amy Johnson is the one with the corresponding `department_id`, which is what we would expect. Sometimes, though, you want to show *all* tuples, whether they have a match or not. An **outer join** shows every tuple from one relation with values from a matching tuple if one exists in the other relation or empty values if not.

```
Customer LEFT OUTER JOIN Purchase
ON Customer.id = Purchase.customer_id
```

id	name
1	Susan Thrifty
2	Will Spendalot
3	Clare Saves

id	customer_id	price
1	2	67
2	2	34
3	2	110
4	1	3

id	name	purchase_id	customer_id	price
1	Susan Thrifty	4	1	3
2	Will Spendalot	1	2	67
2	Will Spendalot	2	2	34
2	Will Spendalot	3	2	110
3	Clare Saves			

An outer join

If you want to get a list of all customers and any purchases they've made, then you need a left

outer join so that customers without any purchases are included. This join takes every tuple from the relation on the left of the join (`Customer`, in this case) and creates a new tuple for that customer and each purchase they made *including if they made no purchases*. In the example above, you can see that Clare Saves has been true to her name and made no purchases. She still appears in the results. If we'd used an inner join here, Clare would not appear as there are no matching tuples in the `Purchases` relation.

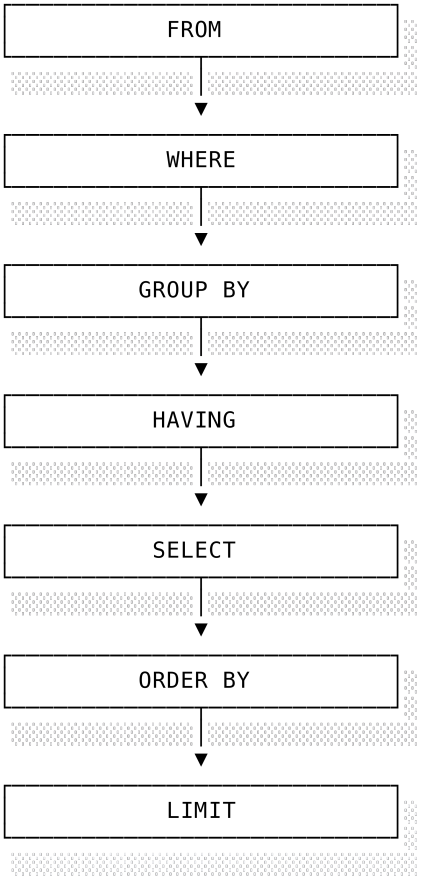
This entails the existence of some kind of empty value that expresses the absence of any value at all. In SQL this is called `NULL`. SQL uses triple-value logic: true, false and null. If you come from the world of JavaScript, you might think that `NULL` implies some kind of falseness. It does not. False is a value. `NULL` is not a value.

The power of relational algebra comes from the fact that it offers a set of operations that are powerful enough to allow the application developer to work their magic but limited enough to allow database systems to create very efficient implementations. A downside is that you need to be able to express your application's data in terms of relations. When starting a new project or adding new functionality to an existing one, it is very important to sit down and design the required schema: what tables do I need, how do they relate to each other, what constraints should I apply?

One of the most interesting things about SQL is that it is a *declarative* programming language. When writing a SQL query, we tell the database engine what results we want in terms of relational algebraic operations (using `FROM`, `WHERE`, `SELECT` and so on). We give absolutely no instruction to the database system about how it should actually go about generating the results. This is very different to standard programming, where we normally tell the computer exactly what to do. As we'll see later, the database system is responsible for taking the query and finding an efficient way to generate the results.

SQL's declarative nature is one of the things that cause people problems. It's a very different way of thinking about programming. Another snag is the structure of SQL queries themselves. True to its declarative nature, SQL was designed to read like an English instruction: select these attributes from these tuples where the tuple matches these criteria. Unfortunately, the *logical* order in which the database engine analyses the query is different to the *lexical* order in which it is written. This leads to confusing errors where the system complains that it doesn't understand a reference you're using even though the reference is right there, goddammit.

We will shortly follow a sample SQL query through the innards of a database system. Before beginning, let's understand the logical ordering of the query and how it might be expressed in relational algebra.



Executing a SQL query

This flowchart describes the logical order of SQL. It's very important to remember that the logical ordering is different to both the lexical ordering and the order in which the database system actually executes the query. The system is free to reorder the query into a more efficient form as long as it returns the correct results.

Our example schema models a ticket booking system. Users buy tickets for events. A user can have many tickets but a ticket has only one purchaser. This is a **one-to-many** relationship. The event is represented by the `event_name` attribute in the ticket. A more realistic schema would have a separate `Events` relation but I want to avoid cluttering the example with too many joins.

```
CREATE TABLE Users(
  id integer primary key,
  first_name string,
  surname string
);

CREATE TABLE Tickets(
  id integer primary key,
  price integer,
  purchaser_id integer references Users,
  event_name string
);
```

Our sample query will return every customer and the total they've paid for each event they have tickets for:

```
SELECT
  users.first_name,
  users.surname,
  tickets.event_name,
  SUM(tickets.price) AS total_paid
FROM users
LEFT JOIN tickets ON tickets.purchaser_id = users.id
GROUP BY users.first_name, users.surname, tickets.event_name;
```

The written SQL query begins with `SELECT`. From what we know of relational algebra, this is a bit odd. A `SELECT` is a projection, meaning that it extracts some subset of attributes from the tuples. We need to have tuples before we can do a projection! Logically, the system actually starts by retrieving the tuples from the relation specified in the `FROM` clause. These are the raw input that will be fed into the query pipeline. Joins are also processed as part of this step. This means that the joined tuples will contain all attributes from the joined relations.

After this, the `WHERE` clause applies a selection operation to filter out unwanted rows. In the `WHERE` clause, we can only reference things that are specified in the `FROM` clause. For example, I might try and amend our query to only show customers who have spent more than £50 on an event:

```
SELECT
  users.first_name,
  users.surname,
  tickets.event_name,
  SUM(tickets.price) AS total_paid
FROM users
LEFT JOIN tickets ON tickets.purchaser_id = users.id
WHERE total_paid > 50
GROUP BY users.first_name, users.surname, tickets.event_name;
```

In SQLite, this fails with the cryptic error:

```
Error: misuse of aggregate: sum()
```

The problem is that `total_paid` is defined in the `SELECT` clause. We haven't got to that yet so `total_paid` is not yet defined. You might think we can get around this by using the **aggregate function** `SUM` directly in the `WHERE` clause. This won't work either because at this point in the query execution we are still selecting which tuples we want. An aggregate function generates a value from multiple input tuples. It stands to reason that we can only perform an aggregation operation once we have a set of tuples to operate on. What might seem like a strange error makes perfect sense when you understand the logical ordering of the query.

Next comes the optional `GROUP BY`. It specifies how the selected tuples should be grouped together. It's only required because we are using `SUM`. Here we want to calculate the sum per user per event. If you're not sure which columns to group by, bear in mind that only columns mentioned in the `GROUP BY` clause or an aggregate function can be used in `SELECT`. After `GROUP BY` comes the optional `HAVING` clause. It's a selection that takes the grouped tuples as input. At this point, aggregate functions are available so we can fix the broken query above:

```
SELECT
  users.first_name,
  users.surname,
  tickets.event_name,
  SUM(tickets.price) AS total_paid
FROM users
LEFT JOIN tickets ON tickets.purchaser_id = users.id
GROUP BY users.first_name, users.surname, tickets.event_name;
HAVING SUM(tickets.price) > 50;
```

Yes, we need to duplicate the `SUM` function. This is because, as we saw above, `total_paid` is not available yet and so we can't reference it. In practice, there's actually a good chance that `total_paid` would work because many database systems allow the use of aliases (defined with `AS`) here as a convenience. They're smart enough to know that you're referring to something in `SELECT` and go

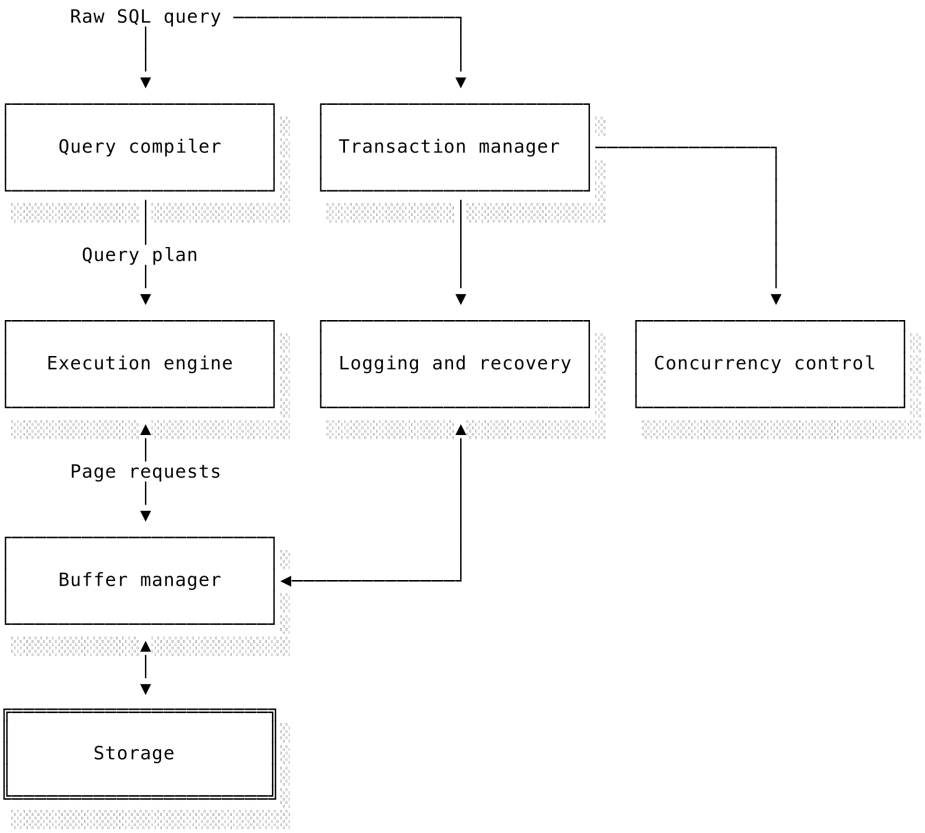
get it from there. Don't rely on this behaviour. It's non-standard and may not work on other database systems.

Next comes the `SELECT` clause. It specifies which attributes we want to project into the temporary results relation. Here we're projecting the user's first name, surname, event name and computing the total paid by that user for the event. We alias the sum using `AS` to give it a name. Once the new attributes have been projected with `SELECT`, the tuples can be ordered with `ORDER BY` and the number of results limited with `LIMIT`.

I often find it easier to write a SQL query in the logical order. I start with a placeholder `SELECT *` (i.e. project everything), then work out which relations I want, then how they're joined, then how to filter them using `WHERE` and finally, once I'm confident that I'm getting the correct results, I go back and specify which columns I want in `SELECT`. Returning all of the columns often shows erroneous results that would have gone unnoticed if I had already cut out most of the columns.

Database architecture

Now that we understand how a query is logically executed, let's look at how the database system actually works. From previous chapters it should be obvious that computer scientists and programmers simply love layering things on top of each other. Out of bare transistors we create the abstraction of logical gates. From these we make components. These in turn are arranged to create processors and memory. A processor exposes an instruction set architecture that forms an abstraction between the user and the hardware. On top of all this we build the operating system, which adds yet more abstractions between the hardware and user programs. A database system is a user program. Amusingly enough, the database system is itself structured as another layer of abstractions.



A typical database architecture

At a high level, a database system consists of a **front end** and a **back end**. The front end takes a computer program specified as a SQL query, parses it, analyses it and generates a plan to execute it correctly and efficiently. The **back end** is responsible for executing the plan and ensuring that data is efficiently and durably stored. It contains the sub-systems that interact with the underlying operating system. The terms "front end" and "back end" are analogous to their usage in web development, where front end converts user actions into HTTP requests and the back end executes the requests. You'll also see in the chapter on compilers that database systems have some interesting parallels with how compilers convert source code into machine instructions. It's useful to think of database systems as programs that run other programs in the form of SQL queries against data. They do this while always maintaining the ACID properties. This is a very cool engineering achievement.

In the beginning, a client makes a connection to the database system. The mechanism varies. Postgres runs a server process listening on TCP port 5432 by default, so we are back in networking territory. SQLite is file-based and so opening a connection simply means opening the database file.

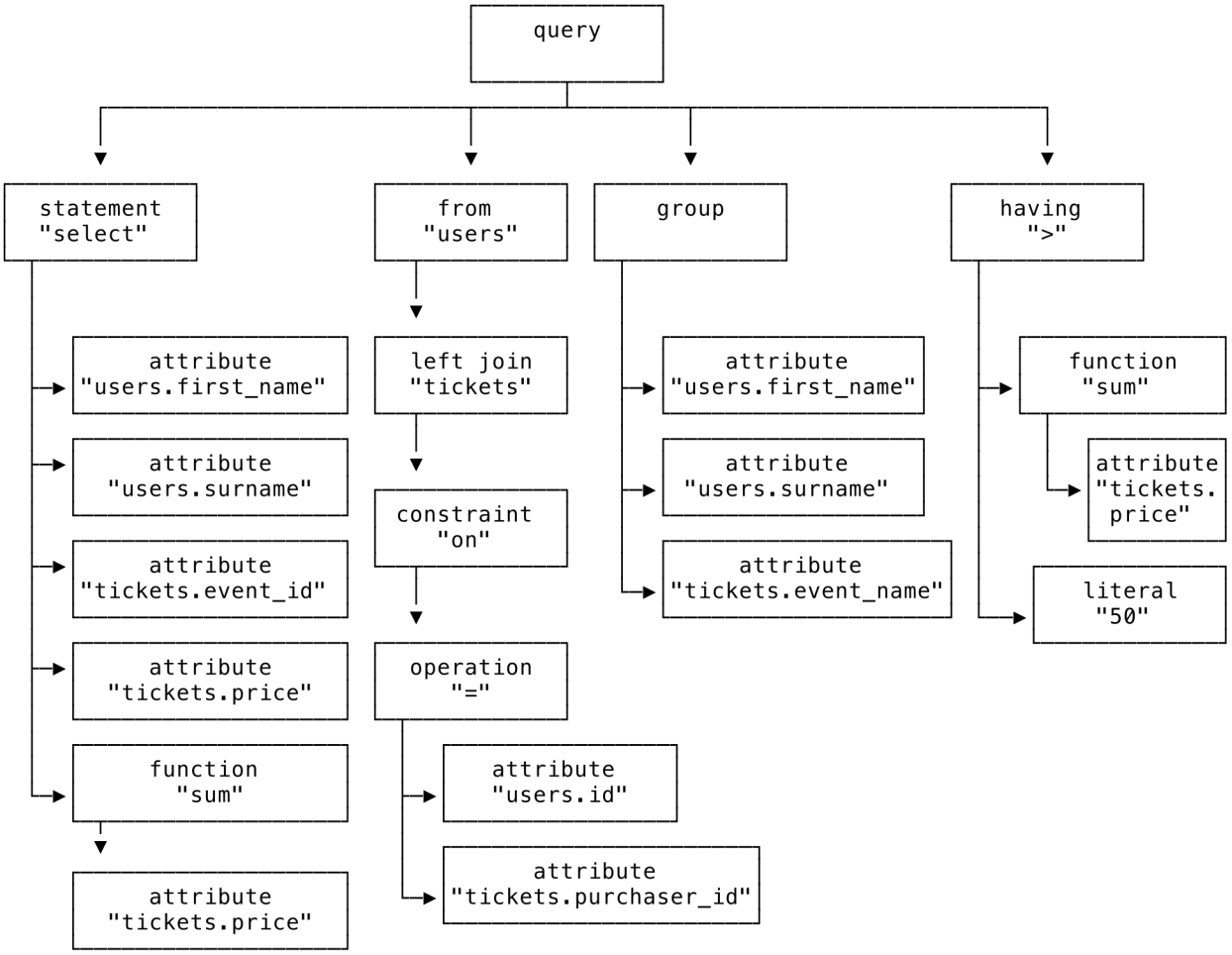
Different database systems handle connections differently. Some create a new worker process for each connection. The worker remains assigned to the connection for as long as it is open and handles every query from that connection. This is how Postgres behaves. An alternative approach is to maintain a pool of workers. An incoming query is assigned to the next available worker. Once the worker finishes the query, it is sent back to the pool where it waits to service the next query. In this model, one client might have each query executed by a different worker. Either way, the maximum

number of available workers sets a limit on how many users can access the database concurrently.

We begin in the **query compiler**. This is part of the front end and is itself made up of a parser, planner and optimiser. It takes a string of SQL and outputs an **execution plan**. The first step is to parse the input string to work out what the hell it means. The worker analyses the string and generates a data structure known as a **query tree**. We'll look at parsing in much more detail when we look at compilers, but for now it's enough to understand that the parser turns this string:

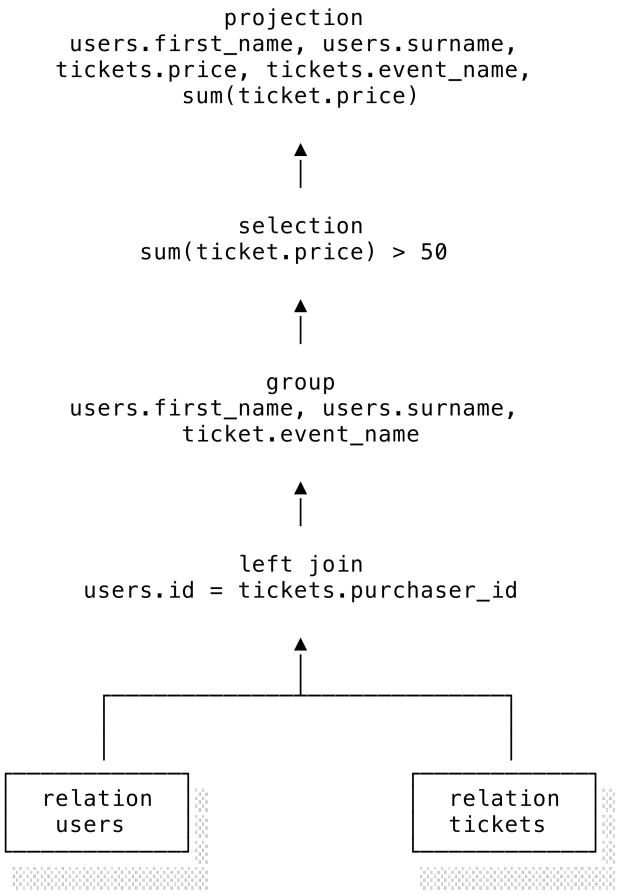
```
SELECT
  users.first_name,
  users.surname,
  tickets.event_name,
  SUM(tickets.price)
FROM users
LEFT JOIN tickets ON tickets.purchaser_id = users.id
GROUP BY users.first_name, users.surname, tickets.event_name;
HAVING SUM(tickets.price) > 50;
```

Into a parse tree like this:



Example parse tree

The query planner takes the parse tree and works out which relations and operations on those relations are required. It generates a logical **query plan**:



Example query plan

The query plan for our query matches the logical order of SQL execution. It begins from the bottom at the leaf nodes, which represent the relations, and works upwards. Each node is a relational algebraic operator. Output flows up to parent nodes and eventually the final result is at the root.

Since SQL is declarative, our query doesn't give the system any instructions about *how* to execute the query. The next step is to convert the logical query plan into a physical query plan. It superficially resembles the logical one but contains lower-level information that the system can use to actually retrieve data and perform operations. You can view the plan generated for a query by prefixing the query with `EXPLAIN`.

Even for very simple queries there will be many ways to physically compute the results. Each different way of putting together the data is called an **access path**. For example, if the system needs to scan through an entire table to find a particular tuple, how many tuples will it have to scan? Is there a valid index (described in more detail below) that might make things faster? If there are multiple indexes available, which is best? If multiple tables need to be joined together, in what order should they be joined and using which algorithm?

The **query optimiser** estimates the cost of each access path and generates an efficient plan as quickly as it can. For non-trivial queries just generating and assessing the possible access paths

is a challenging task in itself. As an example, the number of join orderings increases factorially with the number of join relations. Database systems make the problem more tractable by only attempting to find a reasonably efficient plan reasonably quickly.

To execute the plan we move to the back end. The **execution manager** is responsible for carrying out the required operations. SQLite has an interesting implementation where the query plan contains instructions that are executed by a virtual machine. The idea is to abstract away from the details of the underlying hardware and file system.

The leaf nodes of the physical query plan do not contain data directly. Instead they tell the execution manager where it can retrieve the data. A separate storage layer is responsible for actually persisting the data in an efficient and safe way. The execution manager requests pages of data from the **buffer manager**. The buffer manager interacts with the OS to read and write data and usually maintains a cache of recently used pages to aid performance. The buffer manager is also responsible for preserving durability by verifying that writes to permanent storage are actually carried out.

The **transaction manager** is a component that wraps the query execution steps in transactions to preserve atomicity. Important points in the transaction are logged to persistent storage so that consistency and durability are preserved, including recovery in case of system failure. The transaction manager also implements concurrency control by scheduling transactions to preserve isolation.

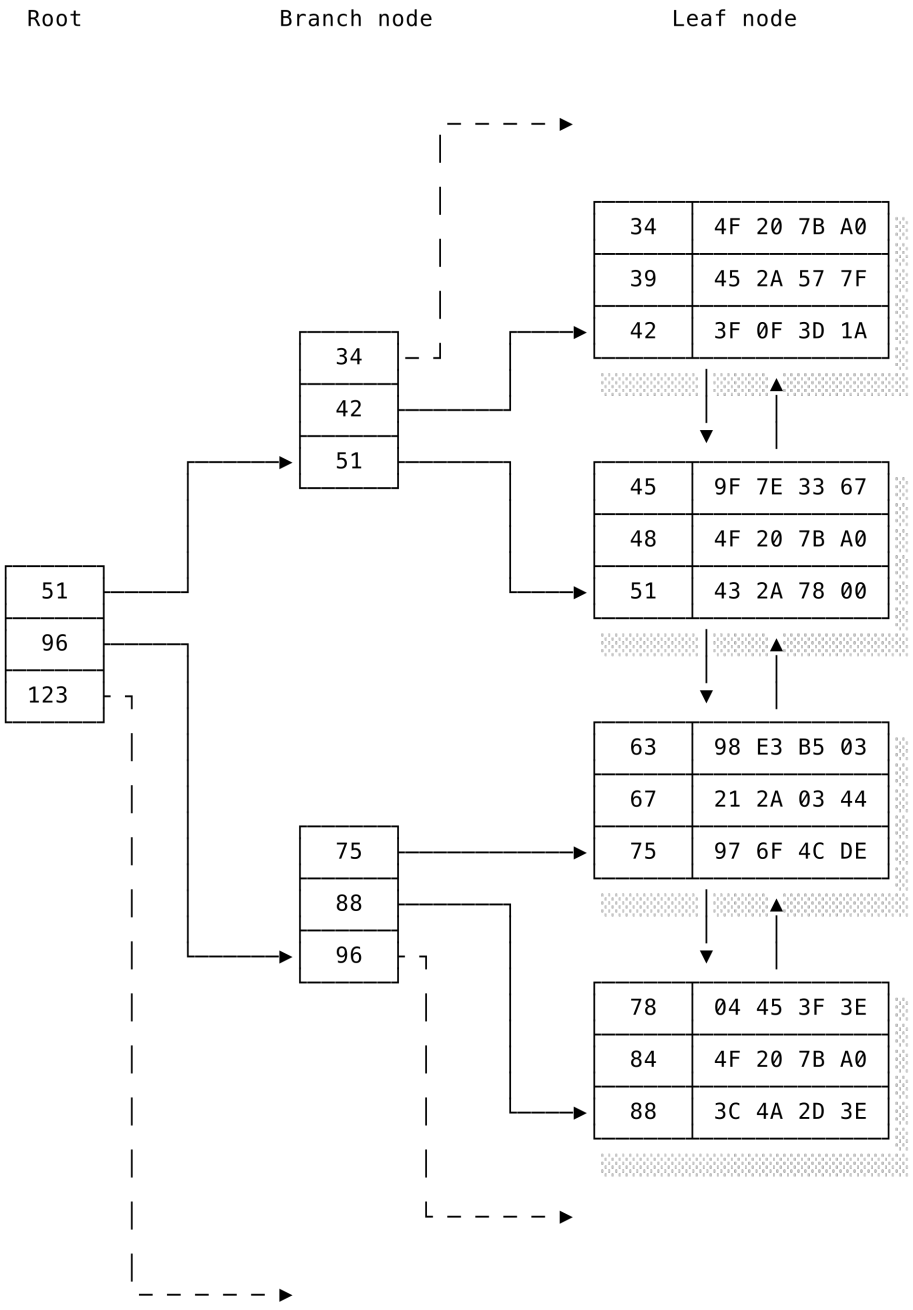
B-trees

Database systems have to quickly search through very large amounts of data. The choice of data structure is therefore very important. A popular choice for database system implementations is the **B-tree**. It's a specialised form of the tree abstract data type that we first saw in the data structures chapter. The "B" in B-tree does not stand for "binary". It stands for "balanced" (or possibly the creator's name – it's a bit unclear).

Recall that in a binary tree each node consists of a value and two children. When searching for a value, you compare it against the current node's value. If the searched-for value is greater, you move to the right child; if less you move to the left. Knowing which subtree to pursue makes lookup much faster because the ordering tells us which branch the value must be in. Database systems routinely store millions of rows in tables. At this scale the cost of moving through all of those levels becomes substantial. Unless the tree has been explicitly written to be cache efficient, it is likely that each node will reside in a different page on disk. Each step down a layer of the tree therefore requires reading a new page from disk and we know how slow that can be. Furthermore, the binary tree is pretty space inefficient. Each node needs space for its value and a pointer to each of its children. If we assume that the pointers and the value are the same size then our space usage grows at $(3n)$.

A B-tree is a generalisation of a binary tree. The data is stored in the leaf nodes, but rather than storing just a single value at each node we store many, potentially hundreds. The values are stored in the leaf in sorted order, allowing for a fast binary search within the leaf. Internal nodes consist of pointers to nodes sorted by their key – the highest value in the child node. To find a value, you

traverse the root node's pointers until you find one with a key greater than the searched-for value. Follow the pointer and repeat until you hit the leaf node. Scan or search through the node to find the value. A diagram might make things clearer:



A B-tree

The leaf nodes hold pointers to the previous and next leaf node so that they also form a doubly-linked list. Traversing the B-tree will always put you in the leaf node with the first instance of the searched-for value. You can then search for other matches by iterating through the leaf nodes in

order.

The strength of the B-tree is that it very rarely needs to have more than a few layers. Finding a value requires correspondingly fewer page reads. The key factor is the number of entries in each node. With fifty entries per node, increasing the depth of the tree by just one means that fifty times more entries can be stored. The cost of keeping the B-tree balanced after modifications is insignificant compared to the increased lookup performance.

Indexes

Database systems store rows without any inherent ordering. To find a particular row, it's therefore necessary to scan sequentially through the whole table. When I first started using databases, I found this kind of weird. It seemed like a really inefficient way of doing things. Even though database systems are very fast, it *is* still a slow way of finding a value. If all you have is an unordered set of rows, there's not much else you can do, just as you can't use a binary search on an unordered input. It doesn't make sense to store the data in sorted order because then every `INSERT` or `DELETE` modification to the database would require shuffling huge numbers of rows around to preserve the ordering.

If we can't maintain an ordering in the data itself, can we maintain an ordering somewhere else? Let's look at an analogue example. The words in a textbook are alphabetically unordered. If you look through a textbook for every instance of a given word, you have no reason to assume that it will appear on one page and not on another. You need to search all of them. Unless, that is, the publishers have seen fit to add an index: a sorted list of words and the pages they occur on. Being sorted means that it's easy to find a particular index entry and the page numbers point us directly to where we want to go.

A **database index** works in just the same way. It's a sorted mapping of a value to the locations of every row in which the value occurs. Database systems use different implementations but a common one is a B-tree in which the leaf nodes hold the indexed values and the locations of each matching row. The index B-tree is entirely separate from the data B-tree. It will be much smaller than the data B-tree but still adds a space cost on the database system. The time benefit is that the index B-tree will probably be small enough to keep in memory. The only disk accesses required are to retrieve the data pages specified by the index. Keeping the smaller index B-tree sorted is much easier than keeping the entire table sorted.

Using indexes is probably the simplest and most effective database optimisation you can perform. An index is created for a particular set of columns, possibly including a condition. If you know that you're likely to query a particular column a lot, you can simply add an index when creating the table. Otherwise, you can use `EXPLAIN` to uncover queries that might benefit from an index. By itself, `EXPLAIN` shows only an estimated cost for a query. In Postgres, `EXPLAIN ANALYZE` actually runs the query and shows the true cost.

Let's modify our earlier query to limit our results to tickets bought by someone with a particular name:

```
SELECT
  users.first_name,
  users.surname,
  tickets.event_name,
  SUM(tickets.price)
FROM users
LEFT JOIN tickets ON tickets.purchaser_id = users.id
WHERE users.first_name = 'Tim'
GROUP BY users.first_name, users.surname, tickets.event_name;
HAVING SUM(tickets.price) > 50;
```

Here's what Postgres estimates for our query (using `EXPLAIN`) when performed on tables containing millions of users and tickets:

```
GroupAggregate  (cost=14543.59..14543.62 rows=1 width=47)
  Group Key: users.first_name, users.surname, tickets.event_name
  Filter: (sum(tickets.price) > 50)
  ->  Sort  (cost=14543.59..14543.60 rows=1 width=43)
     Sort Key: users.surname, tickets.event_name
     ->  Nested Loop Left Join  (cost=1000.00..14543.58 rows=1 width=43)
       Join Filter: (tickets.purchaser_id = users.id)
       ->  Gather  (cost=1000.00..14541.43 rows=1 width=35)
         Workers Planned: 2
         ->  Parallel Seq Scan on users  (cost=0.00..13541.33 rows=1 width=35)
           Filter: (first_name = 'Tim'::text)
       ->  Seq Scan on tickets  (cost=0.00..1.51 rows=51 width=16)
```

The output is a little easier to read when you think back to the query plan diagram. Starting from the bottom, Postgres performs a sequential scan on `users` and `tickets`, joins them, sorts by user and ticket name and then groups and filters. The important thing is that without an index on the `users` table, Postgres has no alternative but to sequentially scan every tuple to find a match. The output of `EXPLAIN ANALYZE` is too verbose to reproduce here, but it shows that the execution time is about 63ms. Since the sequential scan has such a high cost, let's see if things are improved by adding an index:

```
CREATE INDEX first_name on USERS (first_name)
```

When we analyse the query again, the query planner detects that a valid index is available. It estimates the cost of a query plan using the index and, finding that it's much faster, selects it:

```
1  GroupAggregate  (cost=10.11..10.14 rows=1 width=47)
2    Group Key: users.first_name, users.surname, tickets.event_name
3    Filter: (sum(tickets.price) > 50)
4    ->  Sort  (cost=10.11..10.11 rows=1 width=43)
5      Sort Key: users.surname, tickets.event_name
6      ->  Hash Right Join  (cost=8.46..10.10 rows=1 width=43)
7        Hash Cond: (tickets.purchaser_id = users.id)
8        ->  Seq Scan on tickets  (cost=0.00..1.51 rows=51 width=16)
9        ->  Hash  (cost=8.44..8.44 rows=1 width=35)
10         ->  Index Scan using first_name on users  (cost=0.42..8.44 rows=1 width=35)
11           Index Cond: (first_name = 'Tim'::text)
```

The query optimiser notices that there is a suitable index and chooses it (line 10), greatly reducing the time taken to scan `users`. The overall execution time drops from 63ms to 0.1ms! This huge performance boost is why indexes are so important. Always think about what indexes you should have. If you're not sure, analyse your queries to inform your thinking.

Concurrency control in SQLite and Postgres

The simplest way to write a database system is to only allow one user at a time. In many situations, this is perfectly adequate but it clearly won't scale very well. Imagine that you have a website running as a standard Rails app backed by a database. Each visitor is handled by a different process. If each process has its own connection to the database, only one user at a time will be able to access your website. Clearly inadequate! What we want to do is handle multiple, concurrent users.

Concurrency is an amazingly interesting topic that we cover thoroughly separately. Concurrency is challenging for database systems because they hold state. We can't just give each user their own copy of the database because then we'd have conflicting versions, to say nothing of the cost of duplicating the data. Database systems must implement very sophisticated concurrency systems to allow multiple users to operate on the database without causing conflicts or losing data.

The isolation property requires that partial changes are invisible to other concurrent users. Database systems achieve this via transactions. Every read/write operation on the database is wrapped in a transaction. It begins before the first change is made and is **committed**, or finished, when the system verifies that all modifications have been written to permanent storage. From within a transaction, a user can only see the state of the database at the point when the transaction began. Once it commits, a transaction's changes become visible to subsequent transactions. This preserves isolation between users.

The gold standard of concurrent usage is known as **serializability**. A set of concurrent events are serializable if there exists some sequential ordering that achieves the same result. Let's now compare how SQLite and Postgres handle concurrency and durability.

SQLite uses **locking** to implement concurrency control. A lock is a straightforward way of claiming access to a resource. SQLite's default, persistent, transaction log is called the rollback journal. It's

a form of **undo logging**. Before modifying anything, the original value is written to the rollback journal. If the system needs to recover from a failure, it can roll back any uncommitted transactions by writing the original values back into the database.

Before reading from a database file, the user's process tries to acquire a **shared read lock**. As the names suggest, multiple users can have a shared read lock on the same resource. When a user wants to modify a resource, it must first acquire a **modify lock**. This declares the intention to modify the resource. Any shared read locks are allowed to continue but no new ones are permitted, guaranteeing that the modifying process will eventually have exclusive access. The modifying process now creates a separate rollback journal file containing the original version of the pages that will be modified. SQLite requests that the OS flush the file to disk to avoid any caching layers and write straight to persistent storage. Once the journal file is in place, the modifying process can make its changes to the database pages. At first, the changes will only exist in the process's address space and so other reading processes will still see the original data in the database file. When the modifying process is ready to commit, it claims an **exclusive write lock**, preventing anything else from accessing the resource. All changes are written to the database file and flushed to disk. The transaction is committed by deleting the journal file. Finally, any locks are released.

Rollbacks and crash recovery are very straightforward. When SQLite starts, it checks for any journal files. The presence of a journal file indicates that a transaction started but did not commit. The pages in the database file contain potentially invalid data. The system performs a rollback simply by writing the pages from the journal file back into the database file, undoing any modifications performed by the transaction.

Locking is a **pessimistic** form of concurrency control. It assumes that conflicting operations happen frequently and so takes preventative action to stop them from ever happening. If conflicts happen less frequently than anticipated, we're wasting lots of time obtaining all these locks that aren't often needed. **Optimistic** concurrency control allows operations to proceed as normal on the assumption that there will rarely be conflicts. Only when a conflict actually occurs does it take corrective action.

Postgres has an optimistic form of concurrency control known as **multi-version concurrency control** (MVCC). It works by maintaining different versions of the same resource. It uses complex rules to determine which version to show a particular transaction to maintain transaction isolation. To give you just a rough idea of how things work, each transaction is assigned a transaction ID in ascending order. Each version of a resource is tagged with the ID of the transactions that created and last modified it. Each transaction works with a "snapshot" of the database's state at the time it began. A transaction cannot see versions of a resource made by transactions with a higher ID. From the transaction's perspective, these are changes that happen in the future because the transaction with a higher ID must have started later.

Postgres uses write-ahead logging (WAL), which is a form of **redo logging**. Rather than store the old data, as with undo logging, the *new* versions are written to the log. Transactions are committed when a special commit record is written to a separate commit log. Committed changes are periodically written into the database and removed from the WAL at **checkpoints**. If the system needs to recover from a failure, it works its way through the WAL, redoing all of the changes made by committed transactions not yet written into the database. All committed transactions are restored

and uncommitted ones ignored. The main benefit of WAL is that writes happen outside of the main database so reads and writes can proceed concurrently. SQLite also offers write-ahead logging as an option.

Conclusion

Databases are incredibly sophisticated pieces of software. They store data while preserving atomicity, consistency, isolation and durability (ACID). The relational data model, based on relational algebra, is very popular. It represents data as relations and queries as operations on relations. A database system works like a mini computer, taking a declarative SQL statement and generating an efficient execution plan. A typical database system is made up of many sub-components. The transaction, a sequence of operations that must be performed atomically, is the key abstraction that helps the database system to preserve the ACID properties. Transactions take the database from one consistent state to another. Concurrent transactions do not see each other's partial changes. Durability is achieved by writing logs to persistent storage so that the correct database state can be recovered in the event of system failure.

Further reading

It's difficult to find really good resources for databases. All of the textbooks I've read are *really* long and I don't think they're a good starting point. *Designing Data-Intensive Applications* by Kleppmann focuses more on distributed data systems but it does include well-explained sections on data querying (including SQL), storage and retrieval. I'd recommend it as the best overview of modern, web-scale data systems.

Stanford's Jennifer Widom offered a course on databases that was one of the first MOOCs. It's now been expanded into a complete program of introductory database courses[52]. I recommend doing at least the "data models", "querying relational databases" and "database design" modules. If you're still interested you can do the advanced topics.

Architecture of a Database System[53] is a short and readable overview of – you guessed it! – database system architectures. My firm belief is that having a good understanding of a database system's internals helps you to utilise it more effectively.

SQL Performance Explained by Markus Winand is chock full of advice on writing performant SQL. While it's true that you don't need to worry about performance right away, basic things like indexes are table stakes optimisations that even junior developers should be comfortable using. Much of the basic advice in *SQL Performance Explained* is available for free on the sister website Use the Index, Luke[54]. You can see the main thrust of the advice!

[52] https://lagunita.stanford.edu/courses/DB/2014/SelfPaced/about
[53] https://dsf.berkeley.edu/papers/fntdb07-architecture.pdf
[54] https://use-the-index-luke.com

Out of traditional database textbooks, *Database Systems: the Complete Book* by Garcia-Molina, Ullman and Widom was my favourite but I would only recommend it for those who are very interested in databases.

Compilers

Introduction

We come to the final chapter of *The Computer Science Book*. Compilers are well worth studying for two reasons: understanding how your code is transformed and executed is an essential skill for high performing developers. Secondly, truly understanding how a compiler works shows that you have a good understanding of computer science as a whole. Parsing source code draws on the automata and formal language theory we saw in theory of computation as well as the semantics of programming languages and type systems. Modelling and analysing program structures uses many of the algorithms and data structures we studied earlier. Optimising a program and generating suitable machine code requires a solid understanding of the target's system architecture and operating system conventions. Compilers are frequently taught at the end of computer science courses because they are the perfect capstone topic.

We didn't always have compilers. The first computer programs were written directly in machine code. Grace Hopper in the 1950s developed the A-0 system, which could convert a form of mathematical code into machine code. She thus pioneered the idea that programs could be expressed in a more natural form and mechanically translated by the computer into its own machine language. That is the idea at the core of compilation. Further work in the 1950s and 1960s led to the first high level languages such as FORTRAN, COBOL and LISP (uppercase letters were in abundance in those days). These languages abstracted away the low level hardware details and allowed the programmer to focus more on the high level architecture and logic of their programs.

Today, the vast majority of programmers rely on compilers or their close cousin, interpreters. We generally prefer to write in more expressive, high level languages and let the computer itself convert our beautiful code into the bits and bytes that the machine actually understands. It used to be the case that hand-written assembly or machine code would be better than a compiler's output. Nowadays, optimising compilers apply a large suite of sophisticated optimisations. Their output can approach or even exceed the performance of hand-written code.

A modern, optimising compiler is a complex beast. It represents the culmination of decades of research and development. Don't let their complexity intimidate you. In this chapter I hope to demystify the compilation process while instilling a respect and admiration for their capabilities. Popular compilers include Visual Studio on Windows, GCC (the GNU Compiler Collection) and the LLVM platform. Even if you don't regularly use these compilers, many of their ideas and techniques have been incorporated into the advanced JavaScript engines found in modern browsers.

Compilation and interpretation

Compilation and interpretation are two distinct but closely related processes. In general terms, both processes take a high level representation of a program (i.e. its source code) and prepare it for execution. A compiler takes the entire input program and converts it to a low level representation that can be executed later. An interpreter takes the program and executes it directly, incrementally converting the program as it goes. When you compile a program, you are not running the program but instead preparing it to be run later. When you interpret a program, on the other hand, you actually run the program immediately. Interpreters are therefore unable to catch bugs before the program starts. Interpreted programs have a faster start up time but generally aren't as fast as compiled programs.

When the low level representation is raw machine code, we call the compiler output **object code**. Blobs of object code are then combined according to the operating system's specifications to create an executable file, known as a **binary**, that is ready to load into memory and execute. This is how C and C++ approach compilation. The compiler must "target" a specific architecture and operating system so that it can use the correct conventions, instruction sets and so on. The compiled output therefore performs optimally but is tightly bound to the target architecture. Binaries are not portable: a binary compiled for a Linux system won't work on a macOS system and vice versa. This is why you see popular programs offering different builds for various system architectures.

Another approach is for the compiler to output assembly-like **bytecode** that is executed by another program known as a **virtual machine**. This is the approach Java takes. The Java compiler converts Java source code to a simpler but semantically identical format known as Java bytecode. To run the program, we must pass the bytecode to the Java virtual machine (JVM), which can run Java bytecode. This two-step approach might seem unnecessarily complex. The purpose is to make compiled code more portable. Any Java bytecode can be executed on any system with a JVM implementation. There is no need to compile multiple versions for each supported architecture. Virtual machines are so called because they create a software-based, virtual execution environment. It is the job of the VM implementation to map the virtual machine's semantics to the actual hardware. In doing so, it abstracts away the details of the underlying hardware and creates a consistent execution environment. Not all interpreters are virtual machines but many interpreters use virtual machines for these reasons.

To complicate things yet further, compilation can happen at different points. The examples so far are all **ahead-of-time compilation**. The developer writes their code, compiles it and then distributes it in executable format (either binary or bytecode). Compilation happens before the program runs. The benefit of this approach is that the compilation work is only done once. The developer can distribute their program to many users who can directly execute the program without having to first compile it themselves. The downside of ahead-of-time compilation is that compilation can take a long time, especially if the compiler is trying to generate optimised output. If the source code changes frequently then having to regularly recompile can really harm developer productivity.

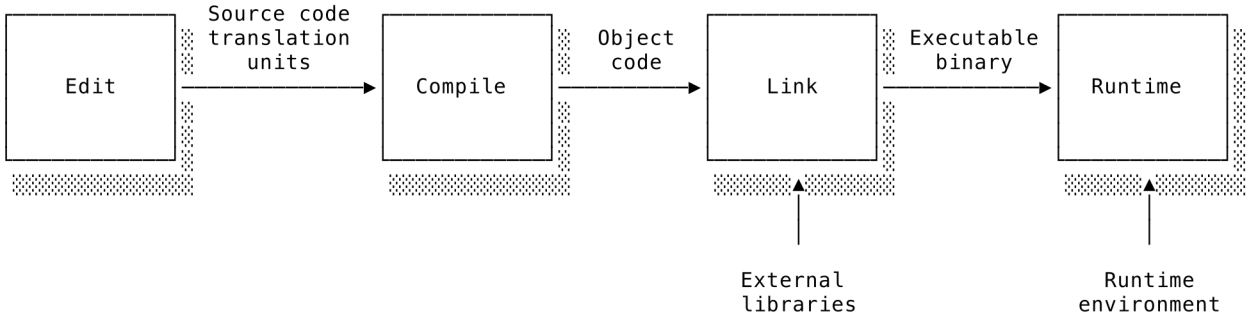
It seems that we can have either fast start up time or fast execution but not both. This is a particularly acute problem for web browsers. JavaScript is served as source code and needs to start up quickly,

which would favour interpretation, but many websites expect JavaScript to handle computationally intensive tasks, which would favour optimised, compiled code. An increasingly popular solution is known as **just-in-time (JIT) compilation**. This attempts to square the circle by starting the program in an interpreter and then compiling the program to faster machine code *as it is running*. Usually the compiler doesn't attempt to compile the entire program but instead monitors which code blocks are executed most frequently and compiles them incrementally. By observing the behaviour of the running program, the compiler might even be able to generate more optimised code than it could ahead of time. JIT compilation is common in JVM implementations and modern browser JavaScript engines.

Techniques such as JIT compilation blur the boundaries between compilation and interpretation. The water is further muddied by the fact that for performance reasons many "interpreted" languages, such as Python and Ruby, are actually compiled to bytecode and then interpreted in a virtual machine! If you find yourself getting muddled, just hold on to a simple guideline: compilation prepares a program for future execution, interpretation executes the program directly. To avoid confusion, in this chapter we'll only consider a straightforward, ahead-of-time compiler targeting a specific architecture.

The program life-cycle

Computer programs go through a series of stages known as the **program life-cycle**. The earlier in the life-cycle we can detect and fix bugs, the better. When caught earlier, bugs are more likely to appear close to their source and will be easier to find and correct.



The life-cycle of a compiled program

Edit time is when the developer is writing or modifying the program's source code. Tools like linters and code formatters are able to statically analyse the source code and detect some types of errors. Here "static" means that the tool doesn't actually execute the program. It merely analyses the source code.

The program enters **compile time** when it is being compiled, naturally enough. The compiler performs its own static analysis to ensure that the input is syntactically valid and well-typed. The compiler operates on chunks of code known as **translation units**. Depending on the language's semantics, a translation unit might be an individual file, a class or a module. As we'll cover in detail

below, the compiler builds up an in-memory representation of the input's structure and semantics and generates suitable output code (e.g. object code, bytecode). Depending on the semantics of the language (particularly its type system), the compiler may be able to detect many bugs at this stage including invalid syntax, incorrect types or even more subtle bugs such as non-exhaustive handling of case statements.

By itself object code can't be directly executed. Each blob of object code is just a sequence of machine code instructions implementing the required functionality. If the code references other functions, global variables and so on, the executable code will need to know how to find them. For example, if I import a function from another translation unit, the object code will contain references to the function but will not have the function's code or the address of its implementation. At **link time** a program known as the **linker** combines the individual blobs of object code into one, coherent executable bundle. It will lay out the object code blobs and write in missing addresses. Often the linker is a different program but is invoked automatically by the compiler. Linking is a fairly mechanical process, so we won't look at it in great detail.

There are different ways of handling linking. **Static linking** is where the linker copies library functions into the binary. The Go compiler statically links the entire Go runtime into every binary. This leads to bigger binary sizes but makes it easier to deploy the binary because it's self-contained. Alternatively, **dynamic linking** defers linking until runtime. The linker relies on the OS to load the necessary libraries into memory along with the program's binary code. A common example of this approach is the C standard library. Since the C standard library is so frequently used, it makes more sense for the OS to make one copy accessible to all programs than to duplicate it in every C binary. Dynamic linking saves on binary sizes and means that improvements in a linked library are immediately available to every program linking to that library. However, it can lead to confusing portability issues when a program is compiled against one version of the library and another one is available at runtime.

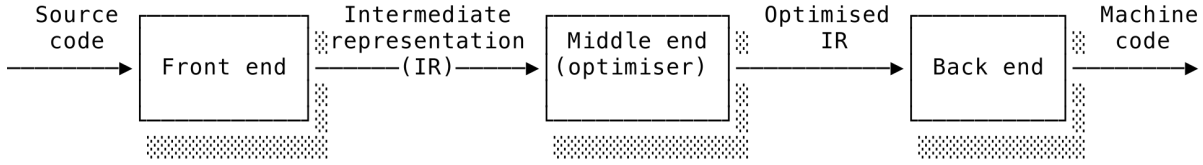
Finally, **runtime** is when the program is actually loaded into memory and executing. Errors occurring now need to be handled gracefully in some way or they could cause the whole program to crash. Even worse, an error might not cause a crash but instead cause incorrect behaviour, possibly propagating far from the original site of the bug. This is not ideal for anyone. The **runtime environment** is the available functionality not directly provided by the running program. The richness of the runtime environment depends on the language and how the program is executed. At one extreme, the runtime environment might be an entire virtual machine and its interface to the physical system. Go's runtime provides a lot of complex functionality such as automatic memory management and cooperative goroutine scheduling. C, at the other extreme, is pretty lazy and leaves all of that work to the programmer. The C runtime is therefore just a standard library of useful functions and a little bit of platform-specific code to handle passing control to and from the kernel.

Building a compiler

It's tempting to think of compilers as somehow magical programs that reach deep into the depths of computation itself in order to give life to our squalid source code. In reality they work just the

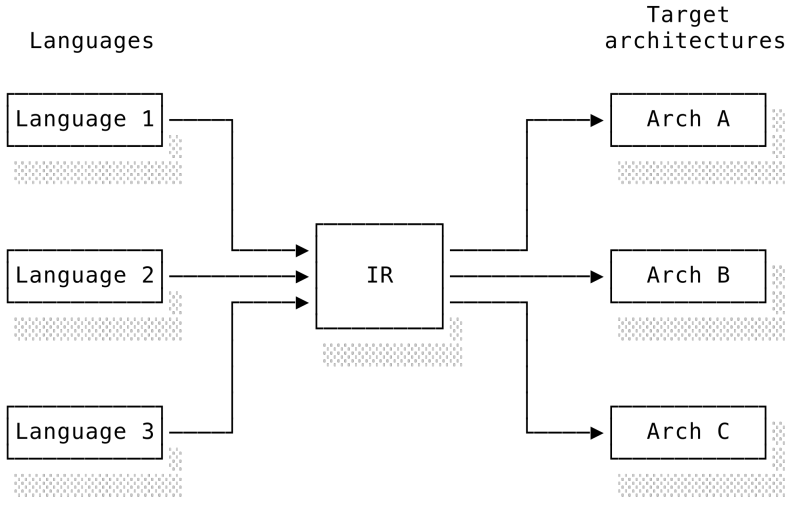
same as any other program: they take input, process it and generate output. What is special about compilers is that their input and output are other programs. High level source code goes in one end and machine code comes out the other.

As ever in computing (and life!), when we face a complex task it is helpful to break it down into simpler sub-tasks. Compilers are generally structured as a pipeline of processing steps. Each step is responsible for a different part of the compilation process. The **front end** of the pipeline works directly with the incoming source code. It uses its understanding of the language's syntax and semantics to generate a language-agnostic, compiler-specific **intermediate representation** (IR) of the program. The front end encodes into the IR all of the knowledge it can glean from the source code. The **middle end** (or **optimiser**) takes the IR from the front end and applies a sequence of optimisations. This is where a lot of the super cool stuff goes on. Each optimisation involves analysing the IR to find areas ripe for optimisation and then manipulating the IR into a more efficient form. The **back end** is responsible for converting the optimised IR into target-specific machine code.



The compiler pipeline

Each pipeline step should ideally be independent of the others, communicating only by passing IR to later steps. The motivation for this architecture is not just a clean separation of concerns. Code reuse is also extremely important. Imagine that you want to compile m languages on n architectures. If each language-architecture pair needed its own compiler, you'd need to write $m \times n$ compilers. The pipeline approach avoids lots of this work. We only need to write a front end for each language and a back end for each architecture, so $m + n$ in total. The optimising middle end can be reused by every language-architecture pair.

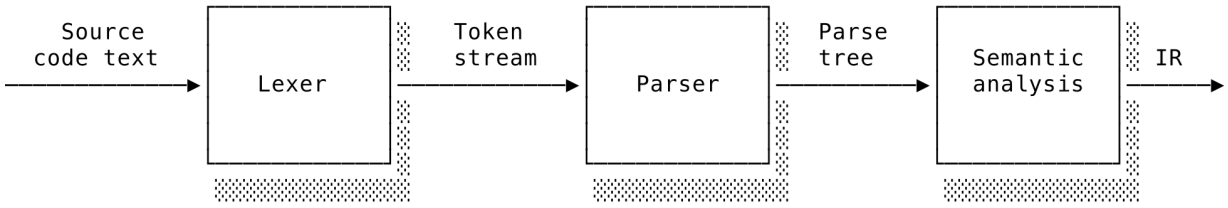


Avoiding $m \times n$ compilers by using IR

The LLVM compiler project is a great example of this approach. It is structured as a low level, compiling virtual machine with back ends for many targets. Using LLVM is an easy way for language designers to have their language supported on a wide range of platforms. They only need to write a single front end that compiles their language into LLVM IR to gain access to all of the optimisations and targets supported by LLVM. Innovations and improvements to LLVM benefit all languages using it.

The front end

The front end converts source code into the compiler's IR.



The front end pipeline

The input will be a flat sequence of characters in a file. The front end first splits the character sequence into its constituent parts in a process known as **lexical analysis**. It then **parses** the input to figure out its structure. Finally it performs **semantic analysis** to check that the program structure makes sense according to the language's semantics. All of the information captured by the front end is encoded into IR.

Throughout this section, I'll demonstrate how these processes work on a fragment of JSON:

```
1 { "names": ["Tom", "Tim", "Tam"] }
```

Decoding a JSON string into an object is similar to parsing source code into IR.

Lexical analysis

The front end's work begins with lexical analysis (also known as lexing or scanning), which is performed by the **lexer**. The source code input will be a long sequence of characters with no clear sub-divisions or structure. Lexical analysis is the process of breaking the input into individually meaningful units known as **tokens**. A token is a pair of a lexical category and an optional attribute, called a **lexeme**, extracted from the character sequence by pattern matching. The lexer performs **tokenisation** by going through the character sequence and matching the input against a list of token patterns. The output is a sequence of tokens. Here's how our JSON example might look tokenised:

```
LBRACE, { STRING, names }, COLON, LBRACKET, { STRING, Tom }, COMMA, { STRING, Tim },\
 COMMA, { STRING, Tam }, RBRACKET, RBRACE
```

`{ STRING, Tam }` indicates a `STRING` token with the attribute `Tam`. Note that the double quotes in the input are not present in the tokenised output. Their only purpose is to delimit strings and so once we tokenise the input, we don't need them any more. Most of the tokens don't need an attribute because the token type conveys all of the necessary information e.g. `LBRACE`. Another valid approach would be to have a single `PUNCTUATION` token specialised via the attribute e.g. `{ PUNCTUATION, ] }`.

The reference to "pattern matching" above should tip you off that tokens use regular expressions to specify the lexemes they match. Sometimes they're pretty obvious. The pattern for the `LBRACKET` token is just `/[/`. Whenever the lexer matches a left bracket, it outputs a `LBRACKET` token. The patterns for identifiers and numbers can be more complicated, particularly when numbers can have negatives, decimals and exponentiation. A programming language's lexical grammar will define the rules for matching every type of token. Here's a simple example of how C identifiers are defined:

```
letter     = [a-zA-Z_]
digit      = [0-9]
identifier = letter(letter|digit)*
```

A valid C identifier is any combination of letters (including underscore) and digits, provided it begins with a letter. As we know from the theory of computation, regexes can be encoded as finite automata. Lexers are often implemented by defining each token's match pattern as a regex and then using tools called "lexer generators" to automatically combine each regex's finite automaton into a single, deterministic finite automaton that matches every token. This is relatively easy to do programmatically because regular expressions are computationally simple. The regular expression pattern for the C identifier rules above would be: `[a-zA-Z_][a-zA-Z0-9_]*`. You can see clearly how it is constructed from the constituent lexical rules.

The simplicity of regular languages brings limitations. Recall from our theory of computation discussion that a finite automaton must have a limited number of states. It is not possible to create regular expressions that can "remember" how many times they've seen a character. When checking for syntactically valid code, it is very common to need to do this. For example, a common rule is that every opening brace must be followed by a corresponding closing brace. The lexer can catch simple errors such as an invalid identifier but it cannot catch these more complex errors. Had I left off the closing brace in the JSON example above, the lexer would have just generated the same token sequence minus the final `RBRACE`. More sophisticated analysis requires a more powerful tool.

Syntax analysis

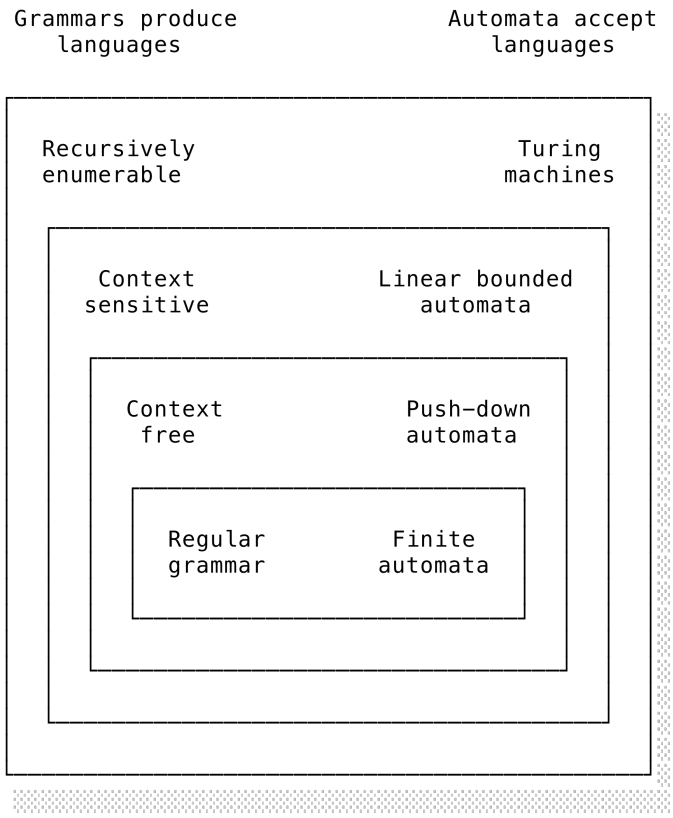
The lexer has generated a sequence of meaningful tokens. The **parser** takes the token sequence and uses the language's syntax rules to recreate the structure of the program. The parser either reports the syntax errors it finds or generates IR for consumption by subsequent stages. This is the last point at which the compiler works directly with the source code.

The syntax rules are normally defined in a language's specification as a grammar: a collection of **production rules**. Each rule describes how tokens may be combined to produce a syntactically valid statement. The syntax rules for programming languages can become quite complex. Designing good syntax is a balancing act between making code readable for humans and making it easily parseable for the compiler. Rules are often defined in **Backus-Naur Form (BNF)**. A simplified grammar for JSON is below. I've used an extended form of BNF that includes regex-style syntax for expressing repetition and optionality.

```
1  json   ::= value
2  value  ::= string | number | object | array | 'true' | 'false' | 'null'
3
4  string ::= [a-zA-Z_]*
5  number ::= 0 | [1-9] [0-9]*
6
7  pair   ::= string ':' value
8  object ::= '{' pair (',' pair)* '}' | '{' '}'
9  array  ::= '[' value (',' value)* ']' | '[' ']'
```

The value to the left of each `::=` is known as a **term**. The definition of the term is on the right. If a term's definition includes another term, we say the term is *non-terminating* (try saying that ten times quickly). Thus, `pair` is non-terminating because its definition includes `string` and `value`. The major difference from the lexical grammar above is that BNF allows us to "remember" matching elements. The definition of `array` allows an arbitrarily nested and arbitrarily long list of values followed by a closing bracket. That would not be possible to express with a regex.

Before moving on to how parsing actually works, I want to focus for a moment on the relationship between language and grammar. Recall from the theory of computation that the set of valid inputs to an automaton is known as the automaton's "language". We also know that a grammar defines how valid statements in the language can be constructed. The three concepts of automaton, language and grammar are all equivalent. Each type of automata accepts (or recognises) a set of languages defined by a particular type of grammar. This is what we want our parser to do. We want some way to verify that the input token sequence conforms to the syntax rules of the language. The relationship between grammars, automata and languages is captured by the **Chomsky hierarchy** of formal languages and their capabilities:



The Chomsky hierarchy

Each grammar is more powerful than the ones below. We know that regular expressions construct finite automata that accept regular languages. Push-down automata (PDA) are more powerful than finite automata, which means that they can implement more complex grammar rules and accept a bigger set of languages. We know from our previous study of push-down automata that they can check for things like balanced parentheses, which regexes can't do. Grammars that can be implemented with push-down automata are known as **context-free** grammars. Obviously, a Turing machine can accept everything that can be accepted by an automaton.

We won't talk too much about context-sensitive grammars and linear bounded automata, which are a restricted form of Turing machines. The problem with a context-sensitive or recursively enumerable grammar is that parsing that language would require a Turing machine. The simple act of parsing a program would entail arbitrary computation and, due to the halting problem, would not be guaranteed to terminate. This does sometimes actually happen, usually by accident. In C++, it is possible to write a program that is only syntactically correct if a particular number is prime (see further reading).

Is our JSON grammar context free? A context-free grammar requires that each production rule consist of a single nonterminal on the left of the `::=` and a sequence of terminals and/or non-terminals on the right. You can see that every rule in our JSON grammar meets this definition. Therefore we have a context-free grammar. The appeal of context-free grammars is that they afford enough power and flexibility to write sophisticated syntax rules but are simple enough to

be efficiently parseable. Parsers make use of a stack data structure to hold the parsing state, which is of course a concrete implementation of a push-down automaton.

The purpose of parsing is to take the flat sequence of tokens and turn them into something that reflects the structure of the program. A tree is a suitable data structure because the statements are hierarchical and nested. Parsing our JSON example produces a **parse tree** that would look like this:

```
            json
              |
            value
              |
           object
              |
         '{' pair '}'
              |
 string      ':'      value
   |                    |
"names"               array
                       |
        '[' value ',' value ',' value ']'
              |         |         |
            string   string     string
              |         |         |
            "Tom"     "Tim"     "Tam"
```

Note that I've used punctuation symbols, rather than `LBRACE` and so on, purely to make the tree easier to read. Parsers construct the parse tree by consuming the input token by token and trying to match the observed sequence to production rules. There are two main approaches within this model. One is to work **top-down** from the most highest level of the parse tree to the lowest. Top-down parsers commonly parse the input from Left to right using Leftmost derivation (we'll cover this shortly) and so are also known as **LL parsers**. The other approach is to work **bottom-up** by starting at the lowest level of the parse tree and constructing upwards. Bottom-up parsers commonly work from Left-to-right using Rightmost derivation and so are also known as **LR parsers**. I'll use LL and LR from now on, just for concision.

So what does top-down, leftmost derivation actually mean? The LL parser iterates through the input token sequence. At each step, the parser can read the current token and the top of the stack. Based on those two values, it selects a production rule and, working from left to right, expands the rule's non-terminal symbols until it reaches the terminal values. When the current token matches the predicted token, it is consumed and the parser moves to the next token. Let's step through a section of the parsing process. Remember we are parsing the token sequence:

```
LBRACE, { STRING, names }, COLON, LBRACKET, { STRING, Tom }, COMMA, { STRING, Tim },\
 COMMA, { STRING, Tam }, RBRACKET, RBRACE
```

1. The parser begins with `LBRACE` and an empty stack.
2. Our starting rule is `json ::= value`.
3. `value` is a non-terminal so we must expand it. We have several possibilities but the only one that can start with `LBRACE` is `object`. We expand the rule to `json ::= object` by pushing `object` on to the stack.
4. `json ::= object` expands to `json ::= '{' pair '}'` by popping `object` off the stack and pushing `'}'`, `pair` and `'{'`.
5. We match the left brace at the top of the stack to the input `LBRACE`. We pop it off the stack and move to the next input token.
6. Now we must expand `pair` to `json ::= '{' string ':' value '}'` by popping off `pair` and pushing `value`, `':'` and `string` on to the stack.
7. We match first the string and then the colon to the input tokens, popping each one off the stack and moving through the input sequence.
8. Now we have a `value` at the top of the stack and `LBRACKET` as the current token. The only matching rule is `array`.

Expanding the array is left as an exercise to the reader. Note that the right brace from `object` would remain on the stack until after the array had been fully parsed. That is how the parser keeps track of symbols it expects to see later. In this example, I was always able to choose the correct rule by looking solely at the current value. Depending on how the rules are formulated, just one token might not be sufficient to unambiguously select the next rule. A LL parser that can peek ahead by k tokens is known as a LL(k) parser.

You might have wondered exactly how I chose each rule correctly. A common way to implement LL parsers is to automatically generate from the production rules a **parsing table** that unambiguously specifies which rule to choose for each token and top of stack pair. The benefit of using a table-based construction is that the parser can be modified to parse a different language simply by changing the table.

Another method of constructing top-down parsers is called **recursive descent**. Instead of using an automatically generated table, recursive descent parsers are usually written by hand as a collection of mutually recursive functions. An explicit stack is not required because the call stack does the same job of keeping track of in-progress rules. Each function matches a single production rule. If the rule contains non-terminals, the function will call other functions representing the non-terminal rules:

```
function expect(token) {
  const current = getToken();
  if (current !== token) {
    throw `Expected ${token}, got ${current}`;
  }
}

function matchPair() {
  matchString();
  expect(':');
  matchValue();
}
```

Recursive descent parsers are surprisingly straightforward and fun to implement. Writing your own one is a great way to get a better understanding of how parsing works.

LR parsing works slightly differently. Consuming the input sequence token by token is known as *shifting*. Each shift, the parser creates a new, single-node parse tree containing the current token. The parser remembers previously parsed items. When it recognises that it's matched a whole production rule, the parser *reduces* the elements. It creates a new node for the matched rule, pops all the matched elements off the stack, attaches them as children to the new node and pushes the new node on to the stack. Multiple elements are reduced to a single value. A LR(k) parser decides which operation to perform based on the current value, previously seen values and k lookaheads. Let's step through parsing our JSON token sequence with a LR parser:

```
LBRACE, { STRING, names }, COLON, LBRACKET, { STRING, Tom }, COMMA, { STRING, Tim },\
 COMMA, { STRING, Tam }, RBRACKET, RBRACE
```

1. We shift to `LBRACE` and create a parse tree node for it. We don't yet know which rule it is part of so we push it on to the stack.
2. Repeat the same operation for each token as far as the `LBRACKET`.
3. Based on the fact that we have just seen a `LBRACKET` and have a `COMMA` just ahead, we are in an `array` rule. Reduce the `string` to a `value`.
4. Repeat for the next two `STRING` values.
5. We have previously seen an `LBRACKET` and a list of comma-separated values. The current `RBRACKET` completes the `array` rule match. Reduce the brackets and comma-separated values to an `array`. We have now constructed the bottom right section of the parse tree.
6. By reducing the `array` to a `value`, we can then reduce the previously seen `STRING` and `COLON` into a `pair`.
7. Shifting again, we find an `RBRACE`. We can reduce the `LBRACE`, `pair` and `RBRACE` into an `object`.
8. Reduce `object` into `json`.

LR parsers take a bottom-up approach by incrementally combining terminals into rules and those rules into larger, non-terminal rules until it reaches the root term at the top of the tree. Values it

doesn't yet know how to deal with are pushed on to a stack for use later. LR parsers are more powerful than LL parsers, in that they can accept a wider range of grammars, but they are also harder to write by hand and so are nearly always auto-generated by programs such as `yacc` and `bison`.

Whereas a LL parser starts constructing the expected rule while in the process of matching, a LR parser waits until it has seen every part of a rule before committing to it. You might find it helpful to imagine how you would traverse the parse tree in the order that each parser creates the nodes. A LL parser corresponds to a **pre-order** binary tree traversal: it starts at the parent node, takes the left branch and then the right. A LR parser corresponds to a **post-order** binary tree traversal: it starts at the left branch, then the right branch and then the parent node.

Parsers, whether LL or LR, work best when they can always deterministically choose the correct rule to apply. If the parser chooses an incorrect one, it might continue to progress but will eventually hit an error state when it has an input token it cannot match to any rule. It must then **backtrack** through the token sequence, select another candidate rule and discard all the work from parsing the incorrect rule. If the parser has to backtrack frequently, this has a hugely detrimental impact on performance. Programming languages are usually designed to be *deterministically context-free*, which means that they can be efficiently parsed by an LR(1) parser (with just a single lookahead). Thanks to these developments, parsing is nowadays seen as a more or less solved problem for compilers.

Semantic analysis

The final step of the front end is to perform a semantic analysis. So far the compiler has only looked at whether the input is syntactically valid. As we've seen, parsing is a fairly mechanical process that can determine structure but not meaning. For example, the English sentence "red wind thinks abruptly" is syntactically correct but semantically meaningless.

Type checking is one very common way of checking that the operations in the parse tree are semantically meaningful. The compiler must analyze the program and assign a type to each name and each expression. It must check these types to ensure that they are used in contexts where they are correct according to the language's type system. Depending on the strength of the type system, the compiler might be able to detect many errors. For example, `a + b` might be a syntactically correct expression but semantically incorrect (i.e. a type error) if `a` is a string and `b` is a number.

In languages that support operator overloading, semantic analysis is required to determine the correct operator. "Operator overloading" means that operators such as `+` may have multiple meanings depending on the type of their operands. In Java, `1 + 2` is addition but `"hello " + "world"` is concatenation. Parsing alone cannot tell the compiler which operation `+` represents. It must consider the semantic meaning of the expression in order to generate the correct IR.

At this stage it is common for compilers to create secondary data structures, separate from the IR, to hold useful bits of data. An example is a **symbol table** recording ever declared identifier and all the information the compiler has managed to find out about it. The exact attributes vary from language to language but will probably include things such as identifier name, type, scope and so on. If the language specification requires that a variable be declared before it is used (which is impossible to

express in BNF), the compiler can easily enforce this by checking that every referenced identifier already has an entry in the symbol table.

Intermediate representations

Intermediate representation (IR) is an internal, compiler-specific data structure or language in which the front end encodes all of the information it has derived from the source code. It doesn't contain any new information. It is merely the input program re-expressed in a format that is easier for the middle and back ends to process. There are two main varieties: the abstract syntax tree (AST) and linear IR.

The parse tree, also known as a *concrete* syntax tree, contains lots of syntactic cruft that's not really important. Once we know the structure, there's not really much point holding on to syntax-specific details such as the brackets delimiting arrays. An **abstract syntax tree** (AST) is a simplification of the parse tree that reflects the structure of the program without unnecessary details. The AST of our JSON example might look like the following:

```
            json
             |
           object
             |
 string             array
   |                  |
"names"   string   string   string
             |        |        |
           "Tom"    "Tim"    "Tam"
```

Alternatively, if the compiler uses a virtual machine internally (e.g. LLVM) it's common for the front end to generate **linear IR** in the form of bytecode for the virtual machine. The format of bytecode is generally a compact, simplified assembly. Though the bytecode looks like assembly, it is still "abstract" in the sense that it does not map directly to the target architecture's ISA. The compiler back end is responsible for later translating bytecode to whatever sequence of machine instructions is required.

Linear IR is often expressed in **static single-assignment form** (SSA), where every variable is defined before it is used and is only assigned once. Reassignments in the source code are re-expressed as assignments to uniquely-indexed variables. This sounds unnecessarily complex but it actually makes things much clearer. Giving every assignment its own name makes it very clear which variables are in use. In the following example, the compiler would have to perform analysis to determine that the first assignment isn't used:

```
name := "Tom"
name := "Tom Johnson"
full_name := name
```

Giving each assignment its own name makes it trivially obvious to the compiler that `name1` plays no part in determining `full_name1`:

```
name1 := "Tom"
name2 := "Tom Johnson"
full_name1 := name2
```

Middle end / optimiser

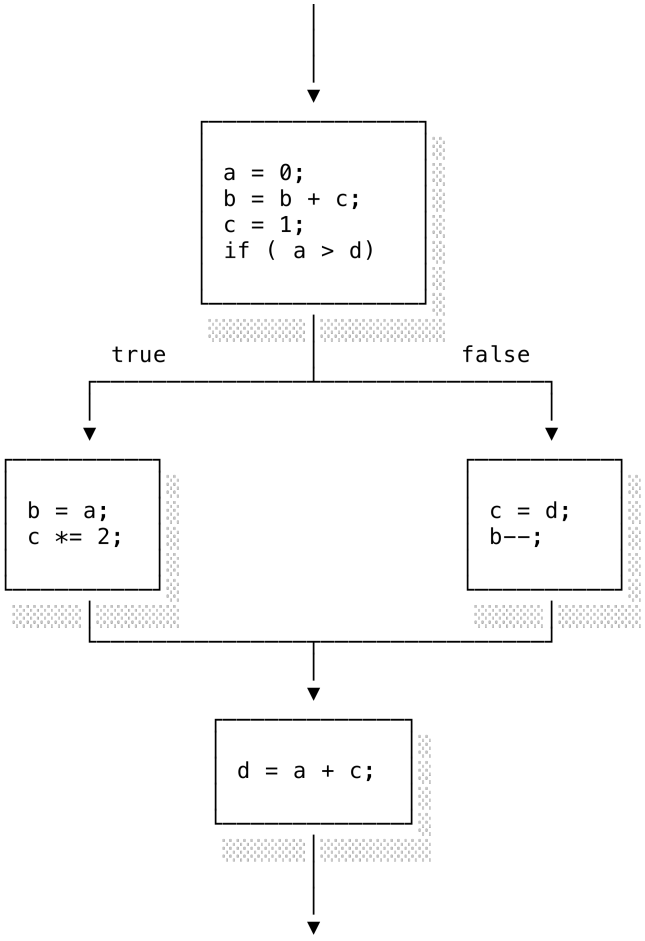
The job of the middle end, also known as the **optimiser**, is to take the IR generated by the front end, analyse its content and transform it to perform the same computation in a more efficient way. Efficiency may be measured in terms of space (the number of instructions) or execution time ($opcount \times cyclesperop$). The objective is a smaller, faster program that does the same thing with fewer resources.

Optimisation is where compilers can really set themselves apart from the competition. Selecting effective optimisations is something of an art and requires a deep understanding of the trade-offs involved. For example, modern processor performance is very sensitive to efficient cache utilisation and so minimising cache misses for both instructions and data is very important. An optimisation that generates faster but more verbose machine code might perform better in isolation but, when applied to a real program, the more verbose machine code might push important values out of the cache and so lead to worse performance overall. In this section I'll illustrate a few common optimisations by showing equivalently optimised source code examples. Remember that optimisations are actually performed on the IR. Don't try and manually optimise your source code.

An optimisation is a combination of analysis to identify inefficiencies and a transformation to remove the inefficiency. At this compilation stage, the optimisations are independent of the target architecture. Target-specific optimisations are performed in the back end. To be an effective optimisation, the compiler needs to get a good return on the time it invests in analysis. An optimisation is impractical if the analysis takes too long. Many optimisation problems are NP-complete or even undecidable if you want to identify absolutely every optimisation opportunity. Compilers therefore use heuristics to get approximate answers. The analysis algorithm only needs to be "good enough" to find a reasonable number of inefficiencies in a reasonably short time. An optimisation must be applicable to any program and ideally affect "hot" parts of the code that are executed frequently, thereby producing a more noticeable improvement for the analysis time invested.

Compilers construct supplementary data structures to help them analyse the program. A **basic block** is a sequence of operations that always execute together e.g. a function body with no control flow statements. Control enters the block at the first operation and always leaves at the last. A **control**

flow graph (CFG) is a directed graph in which basic blocks are the nodes and possible control flow transfers are the edges. CFGs are prerequisites for many useful optimisations.



A control-flow graph. Basic blocks are connected via control flow transfers

There are many, many possible optimisations. Optimising compilers come with whole suites of optimisations and generally allow the programmer to choose the optimisation aggressiveness via command line flags. Optimisations are performed in a sequence. We'll start by looking at optimisations that act within a basic block and gradually increase our scope to the whole program. Please don't think you need to memorise all of these optimisations or start writing your code according to the examples. I merely want to demonstrate how something that sounds as magical as "optimising a program" is nothing more than the accumulation of relatively simple techniques.

Local optimisations

Analysis and transformations for local optimisations all occur within a single basic block. No control flow needs to be considered, aiding efficient analysis.

Local common subexpression elimination detects when the same thing is computed multiple times within the same basic block:

```
shifted := math.Pi * freq + 100
scaled := math.Pi * freq * 0.5
```

It will most likely be faster to store the value in a register rather than recompute it twice:

```
tmp := math.Pi * freq
shifted := tmp + 100
scaled := tmp * 0.5
```

A related optimisation is **local constant folding**. If a value doesn't change at runtime, the compiler can compute it at compile time and insert the constant value into the compiled code:

```
const MEGA_BYTE = 1024 * 1024
```

Why do it at runtime, repeating the work every single time the program runs, when the compiler has all the information it needs to do it now?

```
const MEGA_BYTE = 1048576
```

Strength reduction means replacing slow, complex operations with simpler, faster ones. This can be performed at multiple levels of analysis. Within a basic block, the compiler might replace multiplication and division by two with bit-shifting operations. Division in particular is slow for many processors. Dividing by 2^n is equivalent to shifting n bits to the right. For example, $28/4$ is the same as shifting two bits right:

```
0b00011100 // 28
0b00000111 // 7
```

Dead code elimination deletes code that contributes nothing to the computation. It's semantically meaningless and serves no useful purpose:

```
function deadCodeCalc(num) {
  const a = 200 * 4 * num;
  const warningState = a > (1024 / a * 2);
  return a;
}
```

`warningState` is local to the function and so can't affect anything outside of the function. It doesn't do anything inside the function either so the compiler can remove it without changing the program's behaviour. A dead variable plays no role in the computation.

Loading and storing values comes at a cost, particularly if any of the values are held in memory (remember memory accesses are comparatively slow). Detecting **redundant loads and stores** can eliminate this cost, or at least make it more likely that the essential values can all fit in the processor's registers.

```
initial := x + 5
copy := initial
copy2 := copy
res := copy2 * 5
```

In this admittedly contrived example, the intermediate variables can be optimised away:

```
res := (x + 5) * 5
```

Of course, the compiler needs to be sure that `copy` and `copy2` aren't used anywhere else.

Global optimisations

These optimisations occur within a CFG. That means the analysis must consider behaviour across basic blocks. These optimisations might be more analytically expensive but can offer greater efficiency gains than local optimisations.

Many local optimisations can be performed at a global level. For example, the compiler might determine that the expression in an `if` statement always evaluates to `true`. Since that branch will always be taken, the compiler can eliminate the check and the entire `else` branch. That might eliminate whole chunks of the CFG if the compiler can further determine that the only way to access those code paths was through the initial `else` branch.

```
def dead_code_calc(number)
  a = 200 * 4 * number
  return a
  1000.times { |n| n * n }
end
```

In this function, the code after `return` can never possibly be reached. Cutting out code creates opportunities for further optimisation. For example, if `number` is always the same, a later optimisation might replace the whole call to `dead_code_calc(number)` with a constant.

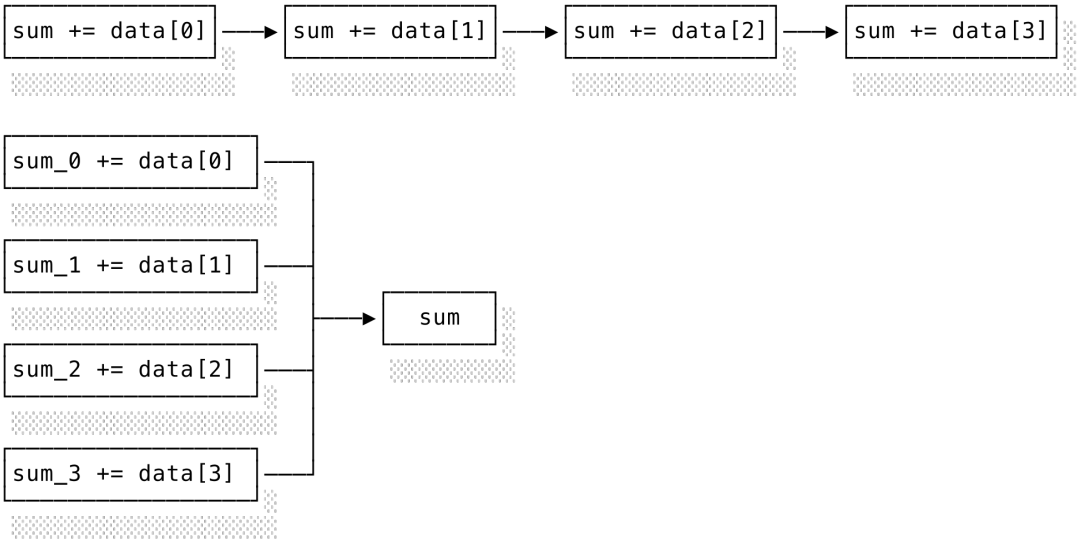
Of particular interest are loop optimisations because loops are usually hot paths. Even a small optimisation in a loop can have a huge impact over many iterations. **Loop unrolling** involves rewriting a loop to perform more steps in each iteration. In this basic C example, we are summing an array of numbers:

```
for (int i = 0; i < n; i++) {
  sum += data[i];
}
```

To perform n additions we need n iterations. An optimisation might transform the code into this:

```
for (int i = 0; i < n; i += 4) {
  sum_0 += data[i + 0];
  sum_1 += data[i + 1];
  sum_2 += data[i + 2];
  sum_3 += data[i + 3];
}
sum = sum_0 + sum_1 + sum_2 + sum_3;
```

In this version, we only need $n/4$ iterations to perform n additions. We have reduced the iteration overhead. Depending on the target processor's support for parallelisation, it might even be possible to perform all four additions in parallel. In the unoptimised example, each addition depends on the value of the previous one and so cannot be parallelised at all.



Loop unrolling reduces data dependencies, enabling parallel computation

Loop-invariant code motion moves operations outside of a loop without changing the loop's semantics. That means less work per loop iteration.

```
res := 10.0
for t := 0.0; t < cycles * 2 * math.Pi; t += res {
    // ...
}
```

Assuming that `cycles` doesn't change within the loop body, there is no point recomputing `cycles * 2 * math.Pi` every iteration. The compiler will move it outside of the loop:

```
res := 10.0
limit := cycles * 2 * math.Pi
for t := 0.0; t < limit; t += res {
    // ...
}
```

If it can determine that `cycles` is constant, this would also be an opportunity for constant folding. It's important that the compiler knows for sure that the value won't change (i.e. that it is *invariant*). You might think that a clever C compiler would move `strlen(input)` out of this loop:

```
for (int i = 0; i < strlen(input); i++) {
    // ...
}
```

In fact, due to the semantics of C, it is very difficult for the compiler to guarantee that `input` won't change between iterations and so it would most likely call `strlen(input)` on every iteration.

Inter-procedural optimisations

Some analysis can only be performed on the program as a whole. This is even more expensive than global (i.e. function) analysis but offers even greater optimisation benefits.

In any architecture, calling a function will entail some overhead. There will be some kind of `CALL` operation, arguments need to be moved into registers and a stack frame needs to be created. At the end there'll be a `RET` operation, the return value needs to be written into a register and the stack needs to be updated. **Function inlining** seeks to avoid all that overhead by replacing a function call with the actual code from the function body.

Here's an example of a function that computes the area of a triangle. Assume that `height` can be negative to indicate a downwards-pointing triangle. To compute the area we need the absolute value of the height:

```
function abs(number) {
  return number >= 0 ? number : -number;
}

function triangleArea(width, height) {
  return width * abs(height) / 2;
}
```

Making a separate call to `abs` is a lot of overhead for a simple conditional. The compiler can replace `abs(height)` with the function body:

```
function triangleArea(width, height) {
  return width * (height >= 0 ? height : -height) / 2;
}
```

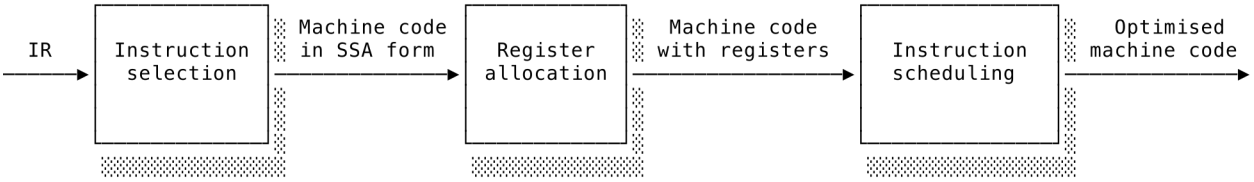
Moving code directly into the function creates further local optimisation opportunities by allowing the compiler to see more of what is going on without having to cross function boundaries.

Remember how I talked about caching earlier? The compiler needs to be careful when performing function inlining. The whole point of a function, of course, is to reuse code and reduce duplication. Each inlining creates another copy of the function's instructions and so increases the size of the program. Function inlining is therefore most suitable for very small functions that are called frequently.

The optimiser essentially refactors the IR to make it more efficient. Lots of small optimisations can build on each other to produce dramatically faster or smaller output. To reiterate my earlier advice, always let the compiler do the work of analysing and rewriting your code as it sees best. It will almost certainly do a much better job. When you first learn about optimisations and code efficiency, it's easy to start stressing about tiny details. Focus on writing clean, correct code and leave the optimisation to the compiler.

Back end

We come to the end of the pipeline. The back end takes the optimised IR and generates output suitable for the compilation target. This involves generating the actual machine code, allocating registers and some target-specific optimisations.



The compiler back end pipeline

Instruction selection

The first step is to generate machine code for the target architecture. The compiler identifies the various constructs in the IR and outputs corresponding machine code. This sounds complicated but it's conceptually quite simple. Imagine that you are compiling a Python function into JavaScript (technically compiling from one language to another is known as **transpiling**). The function looks like this:

```
def circle_area(radius):
    return math.pi * radius ** 2
```

After parsing, you'll have some IR showing that a function named `circle_area` takes an argument called `radius` and returns its square multiplied by π. All you need to do is convert each element of the IR to JavaScript. You know how to define a function in JavaScript:

```
function FUNC_NAME(ARG_LIST) {
    FUNC_BODY
}
```

It's pretty trivial to take the details from the IR and plug them into the template:

```
function circle_area(radius) {
    return Math.PI * radius ** 2
}
```

The principle is the same for machine code. When two values are added, output an `ADD $1, $2` instruction. When there is a conditional, output the comparison followed by a suitable conditional jump instruction and arrange the code branches correctly. The back end knows how each IR element can be expressed in the target architecture's machine code. As a simplification at this point, the back end usually assumes for now that there are an infinite number of registers available. Register allocation is handled later.

Once the code is generated, we have further opportunities for optimisation. The compiler can analyse the auto-generated machine code and identify inefficiencies just as it did in the middle end. The difference is that the back end knows the compilation target architecture and can apply architecture-specific optimisations. For example, to set a register to zero on x86 it is faster and shorter machine code to perform `xor $reg, $reg` (exclusive or with itself) than `mov $reg, 0` (write zero into the register).

Register allocation

Program variables need to be held somewhere. Ideally, they will stay in the processor's registers so that they can be accessed very quickly. So far we've pretended that the target has an infinite number of registers. This simplified the instruction selection task but, of course, in reality the architecture will have a limited amount. The 64-bit x86 ISA only has 16 general use registers! 64-bit ARM has a little more at 31. Either way, we're constrained. The task of register allocation is to assign each active variable to a register, being mindful of data dependencies between instructions. When there are too many values to fit into the registers, the rest must "spill out" to main memory. Since main memory accesses are so much slower, this has a big impact on performance and is generally undesirable.

Interestingly, register allocation can be modelled as a variation of the graph colouring problem. Envisage a graph in which each node is an in-use variable. Edges go between nodes that interfere

with one another, meaning that they must both be live and accessible at the same point in the program. Each register in the system is assigned a colour. Allocating registers then reduces to the problem of colouring each node (i.e. assigning a register) such that no two connected nodes have the same colour. Unfortunately for the poor compiler, this is an NP-complete problem and so an optimal solution is not feasible. The best it can hope to achieve is a reasonably good attempt in a reasonable time.

Instruction scheduling

Pipeline performance is the primary concern of this final step. Recall from our discussion of computer architectures that modern processors use instruction pipelining to improve performance. Rather than work on just one instruction at a time, the processor works on multiple instructions at different stages of the execution cycle. While executing one instruction, it can also decode the next and read in the one after that. The performance of pipelined processors is very sensitive to blockages known as **pipeline stalls**. If an instruction in the decode stage of the pipeline needs to read a register written by an instruction in the execution stage, the pipeline must stall until the executing instruction writes its value into the register.

When faced with a branching instruction, the processor doesn't know which branch to take until the instruction has fully executed. It can either wait and let the pipeline empty or, more likely, try to guess which branch will be taken and begin executing that branch before the branching instruction has even finished executing. A wrong guess means the whole pipeline has to be flushed and work restarted. The back end must detect such dependencies and arrange the instructions to avoid stalls in a way that doesn't change the semantics of the code. This obviously requires the compiler to have a deep understanding of the target architecture.

Once the compiler has scheduled the machine code instructions, its work is done. It has taken a flat sequence of characters, reconstructed the program structure, analysed and optimised it and finally written it back out in the compilation target's language. All that remains is for the linker to gather the object code from each translation unit into a single executable.

Who to trust?

I want to round off this chapter, and indeed the whole book, with a more philosophical point. One recurring idea in this book is that modern computing is a huge tower of abstraction. Look at all the mangling and wrangling your source code goes through to become an executable. And look at how many levels of abstraction there are below your program in the runtime environment, operating system and hardware. How do we actually *know* that the program is doing what we told it to do and nothing else?

In general, we can't be sure about what a program is doing without looking at its binary code. We can use a program known as a **disassembler** to convert raw machine code back into assembly and examine that. Disassembled code is notoriously hard to read because all of the useful comments and names will have been removed or mangled by the original compiler. Assembly code concerns

itself with the low level details and it is hard to reconstruct the overall intent. With time, dedication and skill, it's definitely possible to **reverse engineer** a program and work out what it does, but it's infeasible to do that routinely for anything but the most trivially small programs.

Reading source code is easier but we need to be sure that the compiler's output matches the original source code's semantics and behaviour. We could examine the compiler's source code to check that it faithfully preserves behaviour. Then again, since all binary programs have been compiled it stands to reason that a compiler's binary executable must itself have been compiled. How do we know that the compiler was compiled correctly?

The simple answer is that we need to trust. In *Reflections on Trusting Trust* (see further reading) Ken Thompson, one of the original Unix developers, described how he added a "hack" to the `login` command's source code. The hack gave him backdoor access to any system using the bugged binary. Obvious, Thompson did this for demonstration purposes only! A security-conscious system administrator might try and prevent this class of attack by compiling `login` from source that they had carefully reviewed themselves. Thompson's backdoor would be easily detected by a code review:

```
if (user_name == "KEN_THOMPSON") {
  permitAccessAllAreas();
}
```

So he moved the code for the hack out of `login`'s source code and into the C compiler's source code. After compiling a bugged C compiler to include the new backdoor, Thompson had a bugged compiler that would detect when it was compiling `login` and would inject the original hack into the output. The hack wouldn't appear anywhere in the `login` source code.

Of course, our security-conscious administrator might think to review the compiler's source code before compiling it. The hack would still very obvious:

```
if (input_program == "login") {
  append_code("if (user_name == \"KEN_THOMPSON\") {\
                permitAccessAllAreas();\
              }");
}
```

So Thompson made his hack even more sophisticated. He changed the C compiler hack to detect when it was compiling a *compiler* and inject the compiler hack into the output compiler.

```
if (input_program == "c_compiler") {
  append_code("if (input_program == \"login\") {\
                 append_code(\"if (user_name == \\"KEN_THOMPSON\\") {\
                   permitAccessAllAreas();\
                 }\")\
               }");
}
```

The clean compiler would output a bugged compiler that would output a bugged `login`. Thompson then removed the hack from the compiler's source. If our crafty system administrator had discovered that the bugged compiler was mis-compiling `login`, scouring the compiler's source code for hacks and recompiling a clean version wouldn't work. The compiler would always reinsert its own hack. No amount of source code scrutiny would reveal that the compiler was sneaking in nefarious code.

Thompson's conclusion was that you cannot trust code that you did not completely create yourself, all the way down to machine code. You have to trust the code's creator. And even then, you can't be sure that the processor is executing it as you intended. You have to trust the processor's designers to design the processor to behave according to its published ISA. Someone else had to write an assembler in raw machine code. Then another person used that assembler to write a simple C compiler. And then yet another person used that compiler to compile a more complex compiler written in C. Another team of people produced the operating system that runs all these programs and (hopefully) faithfully and honestly mediates their access to the hardware. We must put our trust in the many people who put in so much work over decades to build modern computing.

Conclusion

Compilers are sophisticated and fascinating programs that convert programs from a high level form, normally source code, into a lower-level form ready for execution. Compilers may generate executable binaries for direct execution or intermediate bytecode intended to run in some kind of interpreter or virtual machine. They may do their work ahead of time or as the program is running. Compilers are just programs like any other, but they are a little bit special because their input and output is other programs.

Compilers are typically structured as pipelines. The front end handles converting source code into the compiler's intermediate representation (IR). First it lexes the input character sequence into a sequence of meaningful tokens. These tokens are then parsed according to the rules of the language's grammar to derive the structure of the program and the IR is generated. The optimiser refactors the IR into an more efficient but semantically-equivalent form by analysing the IR for inefficiencies and rewriting them. Finally, the back end converts the IR into the compilation target's language (object code or bytecode), taking care to schedule instructions and allocate registers as efficiently as it can. Each translation unit is compiled into object code that is combined by the linker into an executable binary. External libraries or runtime functionality may also be linked into the executable.

Further reading

The best way to understand compilers is to write one yourself. The final chapters of nand2tetris[55] require you to write a virtual machine and a compiler that compiles a simple language into the VM's bytecode. Both are challenging projects but immensely rewarding. Writing the recursive descent parser in particular is a fun and satisfying task. If that feels like a step too far for you, try starting by writing a simple calculator program. As you implement more complex functionality (nested expressions and so on), you will get exposure to how parsing works.

I have previously recommended Ruby under the microscope[56]. It has a well-explained and illustrated overview of how Ruby tokenises and parses code, compiles it to bytecode and runs it in a virtual machine. Unfortunately, it describes an older version of Ruby's implementation but this shouldn't detract from its usefulness unless you particularly need to know about Ruby's current implementation.

Thompson's Turing Award lecture is known as Reflections on trusting trust[57]. It's very short and very influential.

I mentioned in the chapter that C++'s parsing is accidentally Turing complete. This comes about as new syntax rules get added to a language and have unintended consequences. This Stack Overflow answer[58] demonstrates a program that is only syntactically correct if a number is prime.

Recently there have been a number of high quality compiler and interpreter books aimed at developers without computer science backgrounds. *Crafting Interpreters* by Bob Nystrom and Thorsten Ball's *Writing an Interpreter in Go* and *Writing a Compiler in Go* are all highly recommended for a more in-depth but practically oriented treatment of compilers.

When it comes to compiler textbooks, you are spoiled for choice. The tricky thing is finding the most appropriate one for your needs. The canonical compiler "Bible" is *Compilers: Principles, Techniques, and Tools* (Aho, Lam, Sethi, Ullman), also known as the "Dragon book" after its cover. However, it's now considered rather dated. At the time, efficient parsing was a major challenge so it spends a lot of time on that and less time on the more modern challenges of optimisation. *Engineering a Compiler* is more up to date. It covers plenty of optimisations and is focused on the practicalities (should you ever wish to write your own optimising compiler). It's a very substantial textbook and although it does start from the basics you might want to have worked through one of the introductory texts first.

[55] https://www.nand2tetris.org
[56] https://nostarch.com/rum
[57] http://users.ece.cmu.edu/~ganger/712.fall02/papers/p761-thompson.pdf
[58] https://stackoverflow.com/a/14589567