1ST EDITION

Asynchronous Programming with C++

Build blazing-fast software with multithreading and asynchronous programming for ultimate efficiency

JAVIER REGUERA-SALGADO
JUAN ANTONIO RUFES

Asynchronous Programming with C++

Build blazing-fast software with multithreading and asynchronous programming for ultimate efficiency

Javier Reguera-Salgado

Juan Antonio Rufes

packt

Asynchronous Programming with C++

The authors acknowledge the use of cutting-edge AI, such as ChatGPT and Copilot, with the sole aim of enhancing the language and clarity within the book, thereby ensuring a smooth reading experience for readers. It's important to note that the content itself has been crafted by the author and edited by a professional publishing team.

Portfolio Manager: Kunal Sawant

Publishing Product Manager: Debadrita Chatterjee

Book Project Manager: Prajakta Naik

Technical Review Editor: Farheen Fathima

Senior Editor: Aditi Chatterjee

Technical Editor: Kavyashree K S

Copy Editor: Safis Editing

Proofreader: Aditi Chatterjee

Indexer: Manju Arasan

Production Designer: Shankar Kalbhor

First published: Novemeber 2024

Production reference: 1151124

Published by Packt Publishing Ltd.
Grosvenor House
11 St Paul's Square
Birmingham
B3 1RB, UK.

ISBN 978-1-83588-424-9

www.packtpub.com

To my wife, Raquel, for being my loving partner throughout our joint life journey, a constant source of love, strength, and joy. To my daughter, Julia, whose presence alone fills my heart with hope and wonder.

To my parents, Marina and Estanislao, for their sacrifices, love, support, and inspiration.

– Javier Reguera-Salgado

To my sister, Eva María.

In loving memory of my parents.

– Juan Antonio Rufes

Contributors

About the authors

Javier Reguera-Salgado is a seasoned software engineer with 19+ years of experience, specializing in high-performance computing, real-time data processing, and communication protocols.

Skilled in C++, Python, and a variety of other programming languages and technologies, his work spans low-latency distributed systems, mobile apps, web solutions, and enterprise products. He has contributed to research centers, start-ups, blue-chip companies, and quantitative investment firms in Spain and the UK.

Javier holds a PhD cum laude in high-performance computing from the University of Vigo, Spain.

I would like firstly to thank my wife, Raquel, and my daughter, Julia, for their love and encouragement, inspiring me every day. To my parents, Marina and Estanislao, for instilling in me the values of hard work and perseverance. I am also grateful to my wider family and in-laws for their love, and to my friends for their support. Also, thanks to Juan for co-authoring this book and the Packt Publishing team and reviewers for their dedication and expertise.

Juan Antonio Rufes is a software engineer with 30 years of experience, specializing in low-level and systems programming, primarily in C, C++, 0x86 assembly, and Python.

His expertise includes Windows and Linux optimization, Windows kernel drivers for antivirus and encryption, TCP/IP protocol analysis, and low-latency financial systems such as smart order routing and FPGA-based trading systems. He has worked with software companies, investment banks, and hedge funds.

Juan holds an MSc in electrical engineering from the Polytechnic University of Valencia, Spain.

I kindly thank my parents for all their teachings and support. Also, thanks to Javier for co-authoring this book. Many thanks to the Packt Publishing team and the technical reviewer for their advice and attention to detail.

About the reviewer

Eduard Drusa started programming at the age of 8. He later went on to receive a master's degree in computer science. During his career, his passion for knowledge has led his way to diverse industries ranging from automotive to commercial desktop software development. He used this opportunity to take on diverse challenges and learn different approaches to common problems. Later, he applied this knowledge outside of his original industry to bring novel solutions to old problems. Recently, his interest has been aimed at embedded systems security where he has started to work on a project to create an innovative real-time operating system. To leverage his knowledge to an even greater extent, he leads programming courses.

Table of Contents

Part 2: Advanced Thread Management and Synchronization Techniques

3

How to Create and Manage Threads in C++ 39

4

Thread Synchronization with Locks 69

Part 4: Advanced Asynchronous Programming with Boost Libraries

9

Asynchronous Programming Using Boost.Asio 237

10

Coroutines with Boost.Cobalt 283

Part 5: Debugging, Testing, and Performance Optimization in Asynchronous Programming

11

12

13

Preface

Asynchronous programming is an essential practice for building efficient, responsive, and high-performance software, particularly in today's world of multi-core processors and real-time data processing. This book dives deep into the principles and practical techniques for mastering asynchronous programming in C++, equipping you with the knowledge needed to handle everything from thread management to performance optimization.

There are several key pillars to developing asynchronous software:

- Thread management and synchronization
- Asynchronous programming concepts, models, and libraries
- Debugging, testing, and optimizing multithreaded and asynchronous software

While many resources focus on the basics of parallel programming or generic software development, this book is designed to offer a comprehensive exploration of these pillars. It covers the essential techniques for managing concurrency, debugging complex systems, and optimizing software performance, all while grounding these concepts in real-world scenarios.

We will guide you through various aspects of asynchronous programming with practical examples based on the following:

- Our extensive experience developing high-performance software
- Best practices learned from working across diverse industries, from finance to research centers

As multi-core processors and parallel computing architectures become increasingly integral to modern applications, the demand for asynchronous programming expertise is growing rapidly. Mastering the techniques covered in this book will help you not only tackle today's complex software development challenges but also prepare for future advancements in performance-critical software.

Whether you're working with low-latency financial systems, developing high-throughput applications, or simply looking to improve your programming skills, this book will provide you with the tools and knowledge to succeed.

Who this book is for

This book is designed for software engineers, developers, and technical leads seeking to deepen their understanding of asynchronous programming using the latest C++ versions and optimize their software performance. The primary target audience includes:

- **Software engineers**: Those looking to enhance their skills in C++ and gain practical insights into multithreading and asynchronous programming, debugging, and performance optimization.
- **Technical leads**: Leaders aiming to implement efficient asynchronous systems will find strategies and best practices for managing complex software development and improving team productivity.
- **Students and enthusiasts**: Individuals eager to learn about high-performance computing and asynchronous programming will benefit from the comprehensive explanations and examples, helping them to advance their careers in technology.

This book will empower readers to tackle real-world challenges and excel in technical interviews, equipping them with the knowledge to thrive in today's fast-paced software landscape.

What this book covers

Chapter 1, *Parallel Programming Paradigms*, explores different architectures and models for building parallel systems, along with various parallel programming paradigms and their performance metrics.

Chapter 2, *Processes, Threads, and Services*, delves into processes in operating systems, examining their life cycle, inter-process communication, and the role of threads, including daemons and multithreading.

Chapter 3, *How to Create and Manage Threads in C++*, instructs on how to create and manage threads, pass arguments, retrieve results, and handle exceptions to ensure efficient execution in multithreaded environments.

Chapter 4, *Thread Synchronization with Locks*, explains the use of C++ Standard Library synchronization primitives, including mutexes and condition variables, while addressing race conditions, deadlocks, and livelocks.

Chapter 5, *Atomic Operations*, explores insights into C++ atomic types, memory models, and how to implement a basic SPSC lock-free queue, preparing for future performance enhancements.

Chapter 6, *Promises and Futures*, introduces asynchronous programming concepts, including promises, futures, and packaged tasks, and shows how to tackle real-life problems using these tools.

Chapter 7, *The Async Function*, explores the functionality of `std::async` for executing asynchronous tasks, defining launch policies, handling exceptions, and optimizing performance.

Chapter 8, *Asynchronous Programming Using Coroutines*, describes C++ coroutines, their basic requirements, and how to implement generators and parsers, while handling exceptions within coroutines.

Chapter 9, *Asynchronous Programming Using Boost.Asio*, explains how to use Boost.Asio for managing asynchronous tasks related to external resources, focusing on I/O objects, execution contexts, and event processing.

Chapter 10, *Coroutines with Boost.Cobalt*, explores the Boost.Cobalt library to implement coroutines easily, avoiding low-level complexities and focusing on functional programming needs.

Chapter 11, *Logging and Debugging Asynchronous Software*, explains how to effectively use logging and debugging tools to identify and resolve issues in asynchronous applications, including deadlocks and race conditions.

Chapter 12, *Sanitizing and Testing Asynchronous Software*, explores how to use sanitizers for multithreaded code and explores testing techniques tailored for asynchronous software with the GoogleTest library.

Chapter 13, *Improving Asynchronous Software Performance*, examines performance measurement tools and techniques, including high-resolution timers, cache optimization, and strategies to avoid false and true sharing.

To get the most out of this book

You will need to have previous experience with programming using C++ and how to use debuggers to find bugs. As we are using C++20 features, and in some examples C++23, you will need to install GCC 14 and Clang 18. All source code examples have been tested in Ubuntu and macOS, but as they are platform-independent, they should compile and run on any platform.

Software/hardware covered in the book	**Operating system requirements**
C++20 and C++23	Linux (tested in Ubuntu 24.04)
GCC 14.2	macOS (tested in macOS Sonoma 14.x)
Clang 18	Windows 11
Boost 1.86	
GDB 15.1	

Each chapter includes a *Technical requirements* section highlighting relevant information on how to install the tools and libraries needed to compile the chapter's examples.

If you are using the digital version of this book, we advise you to type the code yourself or access the code from the book's GitHub repository (a link is available in the next section). Doing so will help you avoid any potential errors related to the copying and pasting of code.

Download the example code files

You can download the example code files for this book from GitHub at `https://github.com/PacktPublishing/Asynchronous-Programming-with-CPP`. If there's an update to the code, it will be updated in the GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Conventions used

There are a number of text conventions used throughout this book.

`Code in text`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "The code above generates a vector of normally distributed random numbers and then sorts the vector with both `std::sort()` and `std::stable_sort()`."

A block of code is set as follows:

```
#include <iostream>
#include <thread>

int main() {
    std::thread t1([]() {
        for (int i = 0; i < 100; ++i) {
            std::cout << "1 " << "2 " << "3 " << "4 "
                      << std::endl;
        }
    });
```

Any command-line input or output is written as follows:

```
$ clang++ -O0 -g -fsanitize=address -fno-omit-frame-pointer test.cpp
-o test

$ ASAN_OPTIONS=suppressions=myasan.supp ./test
```

> **Tips or important notes**
> Appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, email us at `customercare@packtpub.com` and mention the book title in the subject of your message.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit `www.packtpub.com/support/errata` and fill in the form.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packt.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `authors.packtpub.com`.

Share Your Thoughts

Once you've read *Asynchronous Programming with C++*, we'd love to hear your thoughts! Please click here to go straight to the Amazon review page for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below

`https://packt.link/free-ebook/9781835884249`

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly

Part 1: Foundations of Parallel Programming and Process Management

In this part, we delve into the fundamental concepts and paradigms that form the foundation of parallel programming and process management. You will gain a deep understanding of the architectures used to build parallel systems and explore the various programming paradigms available for developing efficient parallel, multithreading, and asynchronous software. Additionally, we will cover critical concepts related to processes, threads, and services, highlighting their importance in operating systems, especially in the context of process life cycle, performance, and resource management.

This part has the following chapters:

- *Chapter 1*, *Parallel Programming Paradigms*
- *Chapter 2*, *Processes, Threads, and Services*

1
Parallel Programming Paradigms

Before we dive into **parallel programming** using C++, throughout the first two chapters, we will focus on acquiring some foundational knowledge about the different approaches to building parallel software and how the software interacts with the machine hardware.

In this chapter, we will introduce parallel programming and the different paradigms and models that we can use when developing efficient, responsive, and scalable concurrent and asynchronous software.

There are many ways to group concepts and methods when classifying the different approaches we can take to develop parallel software. As we are focusing on software built with C++ in this book, we can divide the different parallel programming paradigms as follows: concurrency, asynchronous programming, parallel programming, reactive programming, dataflows, multithreading programming, and event-driven programming.

Depending on the problem at hand, a specific paradigm could be more suitable than others to solve a given scenario. Understanding the different paradigms will help us to analyze the problem and narrow down the best solution possible.

In this chapter, we're going to cover the following main topics:

- What is parallel programming and why does it matter?
- What are the different parallel programming paradigms and why do we need to understand them?
- What will you learn in this book?

Technical requirements

No technical requirements apply for this chapter.

Throughout the book, we will develop different solutions using C++20 and, in some examples, C++23. Therefore, we will need to install GCC 14 and Clang 8.

All the code blocks shown in this book can be found in the following GitHub repository: `https://github.com/PacktPublishing/Asynchronous-Programming-with-CPP`.

Getting to know classifications, techniques, and models

Parallel computing occurs when tasks or computations are done simultaneously, with a task being a unit of execution or unit of work in a software application. As there are many ways to achieve parallelism, understanding the different approaches will be helpful to write efficient parallel algorithms. These approaches are described via paradigms and models.

But first, let us start by classifying the different parallel computing systems.

Systems classification and techniques

One of the earliest classifications of parallel computing systems was made by Michael J. Flynn in 1966. Flynn's taxonomy defines the following classification based on the **data streams** and number of instructions a parallel computing architecture can handle:

- **Single-instruction-single-data (SISD) systems**: Define a sequential program
- **Single-instruction-multiple-data (SIMD) systems**: Where operations are done over a large dataset, for example in signal processing of GPU computing
- **Multiple-instructions-single-data (MISD) systems**: Rarely used
- **Multiple-instructions-multiple-data (MIMD) systems**: The most common parallel architectures based in multicore and multi-processor computers

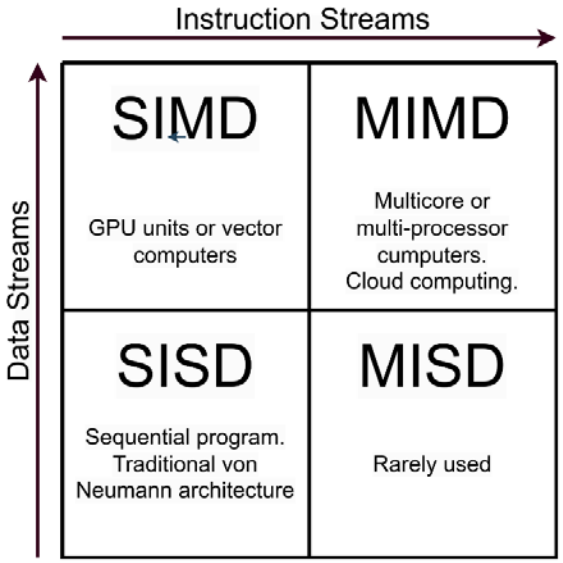


Figure 1.1: Flynn's taxonomy

This book is not only about building software with C++ but also about keeping an eye on how it interacts with the underlying hardware. A more interesting division or taxonomy can probably be done at the software level, where we can define the techniques. We will learn about these in the subsequent sections.

Data parallelism

Many different data units are processed in parallel by the same program or sequence of instructions running in different processing units such as CPU or GPU cores.

Data parallelism is achieved by how many disjoint datasets can be processed at the same time by the same operations. Large datasets can be divided into smaller and independent data chunks exploiting parallelism.

This technique is also highly scalable, as adding more processing units allows for the processing of a higher volume of data.

In this subset, we can include SIMD instruction sets such as SSE, AVX, VMX, or NEON, which are accessible via intrinsic functions in C++. Also, libraries such as OpenMP and CUDA for NVIDIA GPUs. Some examples of its usage can be found in machine learning training and image processing. This technique is related to the SIMD taxonomy defined by Flynn.

As usual, there are also some drawbacks. Data must be easily divisible into independent chunks. This data division and posterior merging also introduces some overhead that can reduce the benefits of parallelization.

Task parallelism

In computers where each CPU core runs different tasks using processes or threads, **task parallelism** can be achieved when these tasks simultaneously receive data, process it, and send back the results that they generate via message passing.

The advantage of task parallelism resides in the ability to design heterogeneous and granular tasks that can make better usage of processing resources, being more flexible when designing a solution with potentially higher speed-up.

Due to the possible dependencies between tasks that can be created by the data, as well as the different nature of each task, scheduling and coordination are more complex than with data parallelism. Also, task creation adds some processing overhead.

Here we can include Flynn's MISD and MIMD taxonomies. Some examples can be found in a web server request processing system or a user interface events handler.

Stream parallelism

A continuous sequence of data elements, also known as a **data stream**, can be processed concurrently by dividing the computation into various stages processing a subset of the data.

Stages can run concurrently. Some generate the input of other stages, building a **pipeline** from stage dependencies. A processing stage can send results to the next stage without waiting to receive the entire stream data.

Stream parallel techniques are effective when handling continuous data. They are also highly scalable, as they can be scaled by adding more processing units to handle the extra input data. Since the stream data is processed as it arrives, this means not needing to wait for the entire data stream to be sent, which means that memory usage is also reduced.

However, as usual, there are some drawbacks. These systems are more complex to implement due to their processing logic, error handling, and recovery. As we might also need to process the data stream in real time, the hardware could be a limitation as well.

Some examples of these systems include monitoring systems, sensor data processing, and audio and video streaming.

Implicit parallelism

In this case, the compiler, the runtime, or the hardware takes care of parallelizing the execution of the instructions transparently for the programmer.

This makes it easier to write parallel programs but limits the programmer's control over the strategies used, or even makes it more difficult to analyze performance or debugging.

Now that we have a better understanding of the different parallel systems and techniques, it's time to learn about the different models we can use when designing a parallel program.

Parallel programming models

A **parallel programming model** is a parallel computer's architecture used to express algorithms and build programs. The more generic the model the more valuable it becomes, as it can be used in a broader range of scenarios. In that sense, C++ implements a parallel model through a library within the **Standard Template Library** (**STL**), which can be used to achieve parallel execution of programs from sequential applications.

These models describe how the different tasks interact during the program's lifetime to achieve a result from input data. Their main differences are related to how the tasks interact with each other and how they process the incoming data.

Phase parallel

In **phase parallel**, also known as the agenda or loosely synchronous paradigm, multiple jobs or tasks perform independent computations in parallel. At some point, the program needs to perform a synchronous interaction operation using a barrier to synchronize the different processes. A barrier is a synchronization mechanism that ensures that a group of tasks reach a particular point in their

execution before any of them can proceed further. The next steps execute other asynchronous operations, and so on.

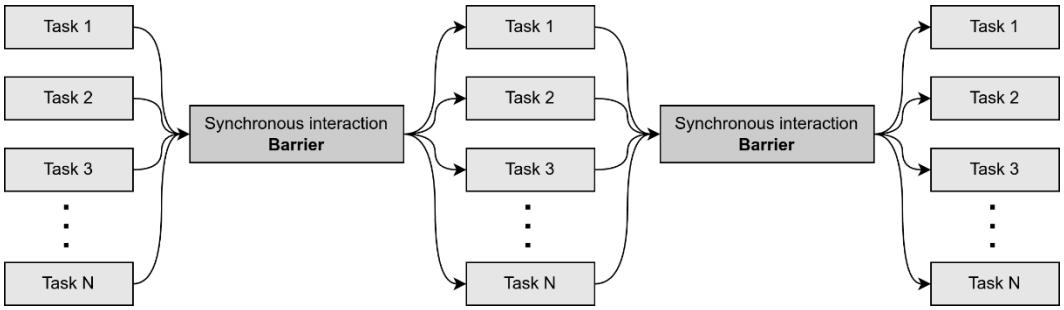


Figure 1.2: Phase parallel model

The advantage of this model is that the interaction between tasks does not overlap with computation. On the other hand, it is difficult to reach a balanced workload and throughput among all processing units.

Divide and conquer

The application using this model uses a main task or job that divides the workload among its children, assigning them to smaller tasks.

Child tasks compute the results in parallel and return them to the parent task, where the partial results are merged into the final one. Child tasks can also subdivide the assigned task into even smaller ones and create their own child tasks.

This model has the same disadvantage as the phase parallel model; it is difficult to achieve a good load balance.

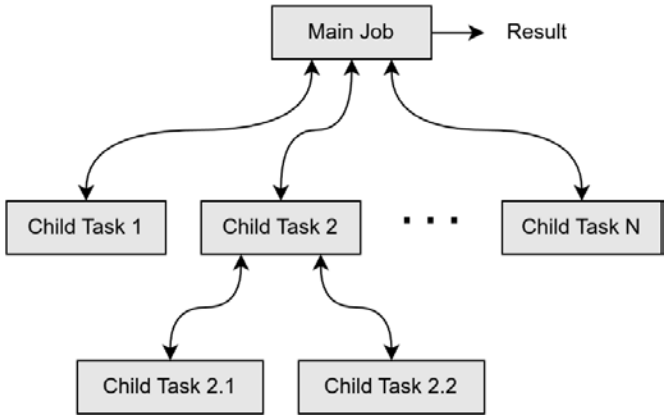


Figure 1.3: Divide and conquer model

In *Figure 1.3*, we can see how the main job divides the work among several child tasks, and how **Child Task 2**, in turn, subdivides its assigned work into two additional tasks.

Pipeline

Several tasks are interconnected, building a virtual pipeline. In this pipeline, the various stages can run simultaneously, overlapping their execution when fed with data.

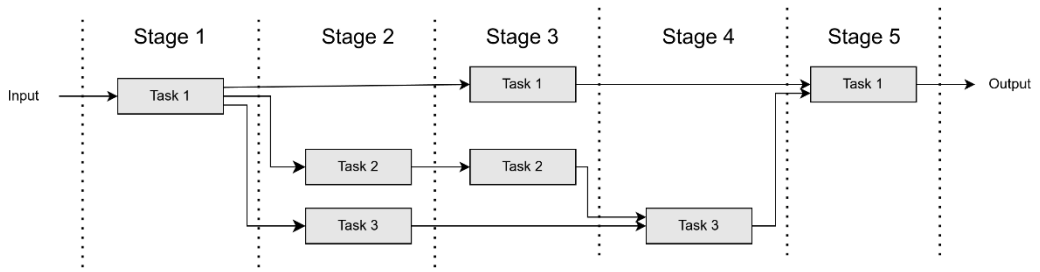


Figure 1.4: Pipeline model

In the preceding figure, three tasks interact in a pipeline composed of five stages. In each stage, some tasks are running, generating output results that are used by other tasks in the next stages.

Master-slave

Using the **master-slave model**, also known as **process farm**, a master job executes the sequential part of the algorithm and spawns and coordinates slave tasks that execute parallel operations in the workload. When a slave task finishes its computation, it informs the master job of the result, which might then send more data to the slave task to be processed.

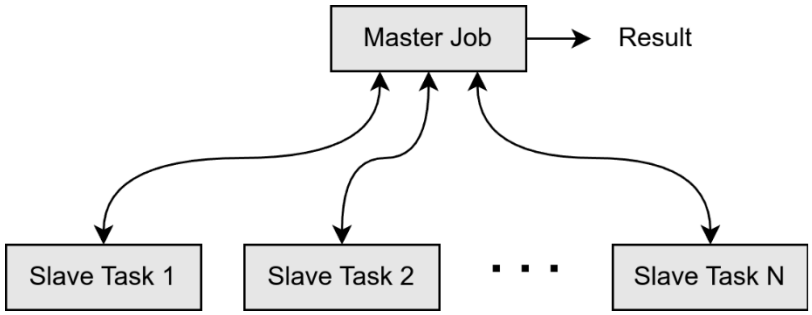


Figure 1.5: The master-slave model

The main disadvantage is that the master can become a bottleneck if it needs to deal with too many slaves or when tasks are too small. There is a tradeoff when selecting the amount of work to be performed by

each task, also known as **granularity**. When tasks are small, they are named fine-grained, and when they are large, they are coarse-grained.

Work pool

In the work pool model, a global structure holds a pool of work items to do. Then, the main program creates jobs that fetch pieces of work from the pool to execute them.

These jobs can generate more work units that are inserted into the work pool. The parallel program finishes its execution when all work units are completed and the pool is thus empty.

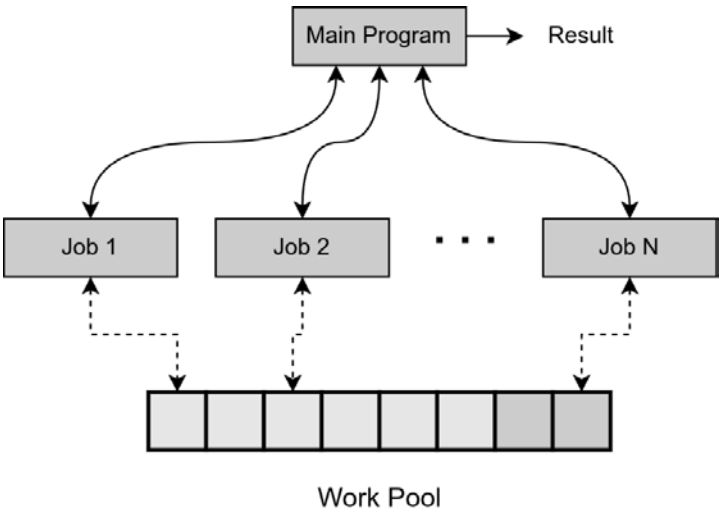


Figure 1.6: The work pool model

This mechanism facilitates load balancing among free processing units.

In C++, this pool is usually implemented by using an unordered set, a queue, or a priority queue. We will implement some examples in this book.

Now that we have learned about a variety of models that we can use to build a parallel system, let's explore the different parallel programming paradigms available to develop software that efficiently runs tasks in parallel.

Understanding various parallel programming paradigms

Now that we have explored some of the different models used for building parallel programs, it is time to move to a more abstract classification and learn about the fundamental styles or principles of how to code parallel programs by exploring the different parallel programming language paradigms.

Synchronous programming

A **synchronous programming** language is used to build programs where code is executed in a strict sequential order. While one instruction is being executed, the program remains blocked until the instruction finishes. In other words, there is no multitasking. This makes the code easier to understand and debug.

However, this behavior makes the program unresponsive to external events while it is blocked while running an instruction and difficult to scale.

This is the traditional paradigm used by most programming languages such as C, Python, or Java.

This paradigm is especially useful for reactive or embedded systems that need to respond in real time and in an ordered way to input events. The processing speed must match the one imposed by the environment with strict time bounds.

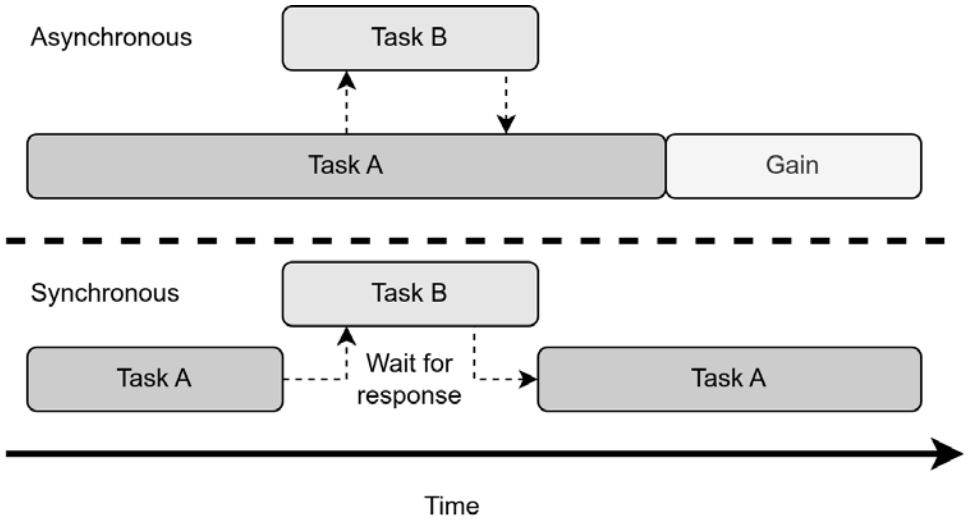


Figure 1.7: Asynchronous versus synchronous execution time

Figure 1.7 shows two tasks running in a system. In the synchronous system, task A is interrupted by task B and only resumes its execution once task B has finished its work. In the asynchronous system, tasks A and B can run simultaneously, thus completing both of their work in less time.

Concurrency programming

With **concurrency programming**, more than one task can run at the same time.

Tasks can run independently without waiting for other tasks' instructions to finish. They can also share resources and communicate with each other. Their instructions can run asynchronously, meaning

that they can be executed in any order without affecting the outcome, adding the potential for parallel processing. On the other hand, that makes this kind of program more difficult to understand and debug.

Concurrency increases the program throughput, as the number of tasks completed in a time interval increases with concurrency (see the formula for Gustafson's law in the section *Exploring the metrics to assess parallelism* at the end of this chapter). Also, it achieves better input and output responsiveness, as the program can perform other tasks during waiting periods.

The main problem in concurrent software is achieving correct concurrency control. Exceptional care must be taken when coordinating access to shared resources and ensuring that the correct sequence of interactions is taking place between the different computational executions. Incorrect decisions can lead to race conditions, deadlocks, or resource starvation, which are explained in depth in *Chapter 3* and *Chapter 4*. Most of these issues are solved by following a consistency or memory model, which defines rules on how and in which order operations should be performed when accessing the shared memory.

Designing efficient concurrent algorithms is done by finding techniques to coordinate tasks' execution, data exchange, memory allocations, and scheduling to minimize the response time and maximize throughput.

The first academic paper introducing concurrency, *Solution of a Problem in Concurrent Programming Control*, was published by Dijkstra in 1965. Mutual exclusion was also identified and solved there.

Concurrency can happen at the operating system level in a preemptive way, whereby the scheduler switches contexts (switching from one task to another) without interacting with the tasks. It can also happen in a non-preemptive or cooperative way, whereby the task yields control to the scheduler, which chooses another task to continue work.

The scheduler interrupts the running program by saving its state (memory and register contents), then loading the saved state of a resumed program and transferring control to it. This is called **context switching**. Depending on the priority of a task, the scheduler might allow a high-priority task to use more CPU time than a low-priority one.

Also, some special operating software such as memory protection might use special hardware to keep supervisory software undamaged by user-mode program errors.

This mechanism is not only used in single-core computers but also in multicore ones, allowing many more tasks to be executed than the number of available cores.

Preemptive multitasking also allows important tasks to be scheduled earlier to deal with important external events quickly. These tasks wake up and deal with the important work when the operating system sends them a signal that triggers an interruption.

Older versions of Mac and Windows operating systems used **non-preemptive multitasking**. This is still used today on the RISC operating system. Unix systems started to use preemptive multitasking in 1969, being a core feature of all Unix-like systems and modern Windows versions from Windows NT `3.1` and Windows 95 onward.

Early-days CPUs could only run one path of instructions at a given time. Parallelism was achieved by switching between instruction streams, giving the illusion of parallelism at the software level by seemingly overlapping in execution.

However, in 2005, Intel® introduced multicore processors, which allowed several instruction streams to execute at once at the hardware level. This imposed some challenges at the time of writing software, as hardware-level concurrency now needed to be addressed and exploited.

C++ has supported concurrent programming since C++11 with the `std::thread` library. Earlier versions did not include any specific functionality, so programmers relied on platform-specific libraries based on the POSIX threading model in Unix systems or on proprietary Microsoft libraries in Windows systems.

Now that we better understand what concurrency is, we need to distinguish between concurrency and parallelism. Concurrency happens when many execution paths can run in overlapping time periods with interleaved execution, while parallelism happens when these tasks are executed at the same time by different CPU units, exploiting available multicore resources.

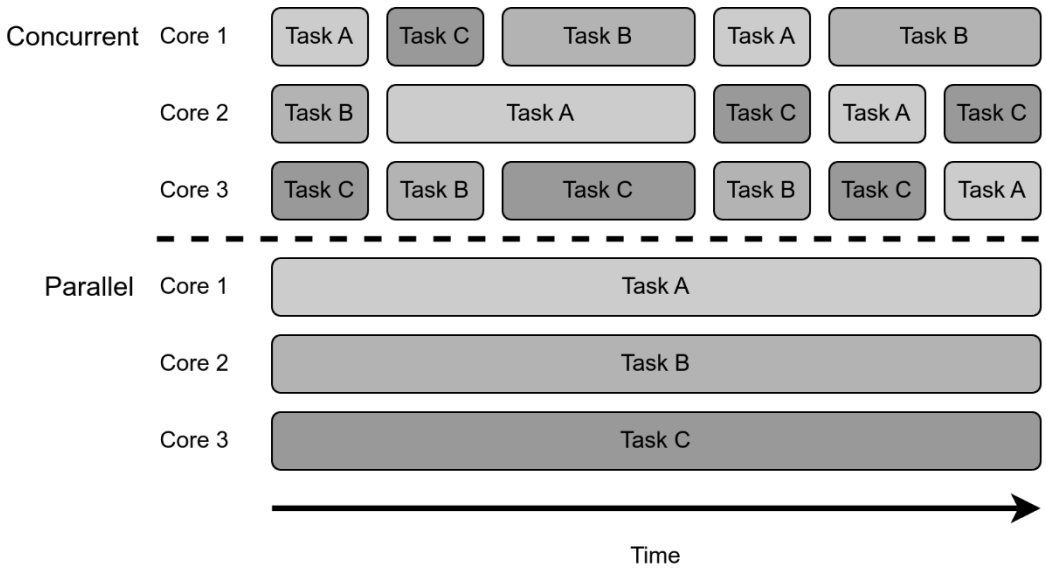


Figure 1.8: Concurrency versus parallelism

Concurrent programming is considered more general than parallel programming as the latter has a predefined communication pattern while the former can involve arbitrary and dynamic patterns of communication and interaction between tasks.

Parallelism can exist without concurrency (without interleaved time periods) and concurrency without parallelism (by multitasking by time-sharing on a single-core CPU).

Asynchronous programming

Asynchronous programming allows some tasks to be scheduled and run in the background while continuing to work on the current job without waiting for the scheduled tasks to finish. When these tasks are finished, they will report their results back to the main job or scheduler.

One of the key issues of synchronous applications is that a long operation can leave the program unresponsive to further input or processing. Asynchronous programs solve this issue by accepting new input while some operations are being executed with non-blocking tasks and the system can do more than one task at a time. This also allows for better resource utilization.

As the tasks are executed asynchronously and they report results back when they finish, this paradigm is especially suitable for event-driven programs. Also, it is a paradigm usually used for user interfaces, web servers, network communications, or long-running background processing.

As hardware has evolved toward multiple processing cores on a single processor chip, it has become mandatory to use asynchronous programming to take advantage of all the available compute power by running tasks in parallel across the different cores.

However, asynchronous programming has its challenges, as we will explore in this book. For example, it adds complexity, as the code is not interpreted in sequence. This can lead to race conditions. Also, error handling and testing are essential to ensure program stability and prevent issues.

As we will learn in this book, modern C++ also provides asynchronous mechanisms such as coroutines, which are programs that can be suspended and resumed later, or futures and promises as a proxy for unknown results in asynchronous programs for synchronizing the program execution.

Parallel programming

With parallel programming, multiple computation tasks can be done simultaneously on multiple processing units, either with all of them in the same computer (multicore) or on multiple computers (cluster).

There are two main approaches:

- **Shared-memory parallelism**: Tasks can communicate via shared memory, a memory space accessible by all processors
- **Message-passing parallelism**: Each task has its own memory space and uses message passing techniques to communicate with others

As with the previous paradigms, to achieve full potential and avoid bugs or issues, parallel computing needs synchronization mechanisms to avoid tasks interfering with each other. It also calls for load balancing the workload to reach its full potential, as well as reducing overhead when creating and managing tasks. These needs increase design, implementation, and debugging complexity.

Multithreading programming

Multithreading programming is a subset of parallel programming wherein a program is divided into multiple threads executing independent units within the same process. The process, memory space, and resources are shared between threads.

As we already mentioned, sharing memory needs synchronization mechanisms. On the other hand, as there is no need for inter-process communication, resource sharing is simplified.

For example, multithreading programming is usually used to achieve **graphical user interface** (**GUI**) responsiveness with fluid animations, in web servers to handle multiple clients' requests, or in data processing.

Event-driven programming

In event-driven programming, the control flow is driven by external events. The application detects events in real time and responds to these by invoking the appropriate event-handling method or callback.

An event signals an action that needs to be taken. This event is listened to by the event loop that continuously listens for incoming events and dispatches them to the appropriate callback, which will execute the desired action. As the code is only executed when an action occurs, this paradigm improves efficiency with resource usage and scalability.

Event-driven programming is useful to act on actions happening in user interfaces, real-time applications, and network connection listeners.

As with many of the other paradigms, the increased complexity, synchronization, and debugging make this paradigm complex to implement and apply.

As C++ is a low-level language, techniques such as callbacks or functors are used to write the event handlers.

Reactive programming

Reactive programming deals with data streams, which are continuous flows of data or values over time. A program is usually built using declarative or functional programming, defining a pipeline of operators and transformations applied to the stream. These operations happen asynchronously using schedulers and **backpressure** handling mechanisms.

Backpressure happens when the quantity of data overwhelms the consumers and they are not able to process all of it. To avoid a system collapse, a reactive system needs to use backpressure strategies to prevent system failures.

Some of these strategies include the following:

- Controlling input throughput by requesting the publisher to reduce the rate of published events. This can be achieved by following a pull strategy, where the publisher sends events only when the consumer requests them, or by limiting the number of events sent, creating a limited and controlled push strategy.
- Buffering the extra data, which is especially useful when there are data bursts or a high-bandwidth transmission over a short period.
- Dropping some events or delaying their publication until the consumers recover from the backpressure state.

Thus, reactive programs can be **pull-based** or **push-based**. Pull-based programs implement the classic case where the events are actively pulled from the data source. On the other hand, push-based programs push events through a signal network to reach the subscriber. Subscribers react to changes without blocking the program, making these systems ideal for rich user interface environments where responsiveness is crucial.

Reactive programming is like an event-driven model where event streams from various sources can be transformed, filtered, processed, and so on. Both increase code modularity and are suitable for real-time applications. However, there are some differences, as follows:

- Reactive programming reacts to event streams, while event-driven programming deals with discrete events.
- In event-driven programming, an event triggers a callback or event handlers. With reactive programming, a pipeline with different transformation operators can be created whereby the data stream will flow and modify the events.

Examples of systems and software using reactive programming include the X Windows system and libraries such as Qt, WxWidgets, and Gtk+. Reactive programming is also used in real-time sensors data processing and dashboards. Additionally, it is applied to handling network or file I/O traffic and data processing.

To reach full potential, there are some challenges to address when using reactive programming. For example, it's important to debug distributed dataflows and asynchronous processes or to optimize performance by fine-tuning the schedulers. Also, the use of declarative or functional programming makes developing software by using reactive programming techniques a bit more challenging to understand and learn.

Dataflow programming

With **dataflow programming**, a program is designed as a directed graph of nodes representing computation units and edges representing the flow of data. A node only executes when there is some available data. This paradigm was invented by Jack Dennis at MIT in the 1960s.

Dataflow programming makes the code and design more readable and clearer, as it provides a visual representation of the different computation units and how they interact. Also, independent nodes can run in parallel with dataflow programming, increasing parallelism and throughput. So, it is like reactive programming but offers a graph-based approach and visual aid to modeling systems.

To implement a dataflow program, we can use a hash table. The key identifies a set of inputs and the value describes the task to run. When all inputs for a given key are available, the task associated with that key is executed, generating additional input values that may trigger tasks for other keys in the hash table. In these systems, the scheduler can find opportunities for parallelism by using a topological sort on the graph data structure, sorting the different tasks by their interdependencies.

This paradigm is usually used for large-scale data processing pipelines for machine learning, real-time analysis from sensors or financial markets data, and audio, video, and image processing systems. Examples of software libraries using the dataflow paradigm are Apache Spark and TensorFlow. In hardware, we can find examples for digital signal processing, network routing, GPU architecture, telemetry, and artificial intelligence, among others.

A variant of dataflow programming is **incremental computing**, whereby only the outputs that depend on changed input data are recomputed. This is like recomputing affected cells in an Excel spreadsheet when a cell value changes.

Now that we have learned about the different parallel programming systems, models, and paradigms, it's time to introduce some **metrics** that help measure parallel systems' performance.

Exploring the metrics to assess parallelism

Metrics are measurements that can help us understand how a system is performing and to compare different improvement approaches.

Here are some metrics and formulas commonly used to evaluate parallelism in a system.

Degree of parallelism

Degree of parallelism (**DOP**) is a metric that indicates the number of operations being simultaneously executed by a computer. It is useful to describe the performance of parallel programs and multi-processor systems.

When computing the DOP, we can use the maximum number of operations that could be done simultaneously, measuring the ideal case scenario without bottlenecks or dependencies. Alternatively, we can use either the average number of operations or the number of simultaneous operations at a given point in time, reflecting the actual DOP achieved by a system. An approximation can be done by using profilers and performance analysis tools to measure the number of threads during a particular time period.

That means that the DOP is not a constant; it is a dynamic metric that changes during application execution.

For example, consider a script tool that processes multiple files. These files can be processed sequentially or simultaneously, increasing efficiency. If we have a machine with `N` cores and we want to process `N` files, we can assign a file to each core.

The time to process all files sequentially would be as follows:

$$t_{total} = t_{file1} + t_{file2} + t_{file3} + \cdots + t_{fileN} \cong N \cdot avg(t_{file})$$

And, the time to process them in parallel would be:

$$t_{total} = max(t_{file1}, t_{file2}, t_{file3}, \cdots, t_{fileN})$$

Therefore, the DOP is `N`, the number of cores actively processing separate files.

There is a theoretical upper bound on the speed-up that parallelization can achieve, which is given by **Amdahl's law**.

Amdahl's law

In a parallel system, we could believe that doubling the number of CPU cores could make the program run twice as fast, thereby halving the runtime. However, the speed-up from parallelization is not linear. After a certain number of cores, the runtime is not reduced anymore due to different circumstances such as context switching, memory paging, and so on.

The Amdahl's law formula computes the theoretical maximum speed-up a task can perform after parallelization as follows:

$$S_{max}(s) = \frac{s}{s + p(1 - s)} = \frac{1}{1 - p + \frac{p}{s}}$$

Here, `s` is the speed-up factor of the improved part and `p` is the fraction of the parallelizable part compared to the entire process. Therefore, `1-p` represents the ratio of the task not parallelizable (the bottleneck or sequential part), while `p/s` represents the speed-up achieved by the parallelizable part.

That means that the maximum speed-up is limited by the sequential portion of the task. The greater the fraction of the parallelizable task (`p` approaches `1`), the more the maximum speed-up increases up to the speed-up factor (`s`). On the other hand, when the sequential portion becomes larger (`p` approaches `0`), `Smax` tends to `1`, meaning that no improvement is possible.

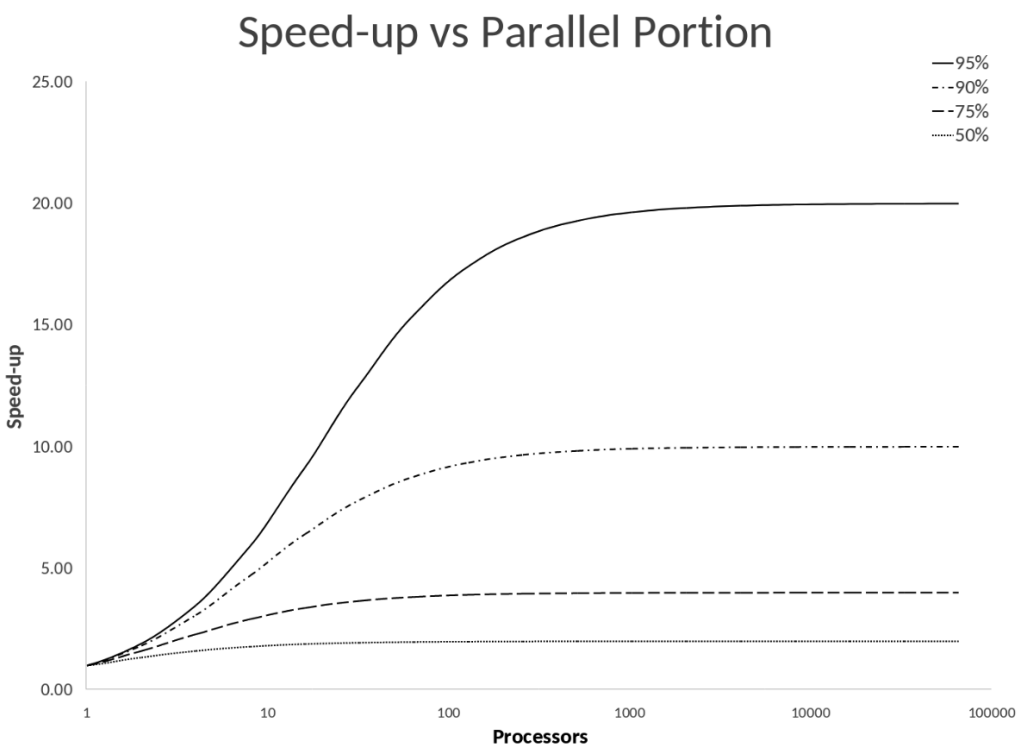


Figure 1.9: The speed-up limit by the number of processors and percentage of parallelizable parts

The **critical path** in parallel systems is defined by the longest chain of dependent calculations. As the critical path is hardly parallelizable, it defines the sequential portion and thus the quicker runtime that a program can achieve.

For example, if the sequential part of a process represents 10% of the runtime, then the fraction of the parallelizable part is `p=0.9`. In this case, the potential speed-up will not exceed 10 times the speed-up, regardless of the number of processors available.

Gustafson's law

The Amdahl's law formula can only be used with fixed-sized problems and increasing resources. When using larger datasets, time spent in the parallelizable part grows much faster than that in the sequential part. In these cases, the Gustafson's law formula is less pessimistic and more accurate, as it accounts for fixed execution time and increasing problem size with additional resources.

The Gustafson's law formula computes the speed-up gained by using `p` processors as follows:

$$S_p = p + (1 - f) \cdot p$$

Here, `p` is the number of processors and `f` is the fraction of the task that remains sequential. Therefore, `(1-f)*p` represents the speed-up achieved with the parallelization of the `(1-f)` task distributed across `p` processors, and `p` represents the extra work done when increasing resources.

Gustafson's law formula shows that the speed-up is affected by parallelization when lowering `f` and by scalability by increasing `p`.

As with Amdahl's law, Gustafson's law formula is an approximation that provides valuable perspective when measuring improvements in parallel systems. Other factors can reduce efficiency such as overhead communication between processors or memory and storage limitations.

Summary

In this chapter, we learned about the different architectures and models we can use to build parallel systems. Then we explored the details of the variety of parallel programming paradigms available to develop parallel software and learned about their behavior and nuances. Finally, we defined some useful metrics to measure the performance of parallel programs.

In the next chapter, we will explore the relationship between hardware and software, as well as how software maps and interacts with the underlying hardware. We will also learn what threads, processes, and services are, how threads are scheduled, and how they communicate with each other. Furthermore, we will cover inter-process communication and much more.

Further reading

- Topological sorting: `https://en.wikipedia.org/wiki/Topological_sorting`
- C++ compiler support: `https://en.cppreference.com/w/cpp/compiler_support`
- C++20 compiler support: `https://en.cppreference.com/w/cpp/compiler_support/20`
- C++23 compiler support: `https://en.cppreference.com/w/cpp/compiler_support/23`

2

Processes, Threads, and Services

Asynchronous programming involves initiating operations without waiting for them to complete before moving on to the next task. This non-blocking behavior allows for developing highly responsive and efficient applications, capable of handling numerous operations simultaneously without unnecessary delays or wasting computational resources waiting for tasks to be finished.

Asynchronous programming is very important, especially in the development of networked applications, user interfaces, and systems programming. It enables developers to create applications that can manage high volumes of requests, perform **Input/Output** (**I/O**) operations, or execute concurrent tasks efficiently, thereby significantly enhancing user experience and application performance.

The Linux operating system (in this book, we will focus on development on the Linux operating system when the code cannot be platform-independent), with its robust process management, native support for threading, and advanced I/O capabilities, is an ideal environment for developing high-performance asynchronous applications. These systems offer a rich set of features such as powerful APIs for process and thread management, non-blocking I/O, and **Inter-Process Communication** (**IPC**) mechanisms.

This chapter is an introduction to the fundamental concepts and components essential for asynchronous programming within the Linux environment.

We will explore the following topics:

- Processes in Linux
- Services and daemons
- Threads and concurrency

By the end of this chapter, you will possess a foundational understanding of the asynchronous programming landscape in Linux, setting the stage for deeper exploration and practical application in subsequent chapters.

Processes in Linux

A process can be defined as an instance of a running program. It includes the program's code, all the threads belonging to this process (which are represented by the program counter), the stack (which is an area of memory containing temporary data such as function parameters, return addresses, and local variables), the heap, for memory allocated dynamically, and its data section containing global variables and initialized variables. Each process operates within its own virtual address space and is isolated from other processes, ensuring that its operations do not interfere directly with those of others.

Process life cycle – creation, execution, and termination

The life cycle of a process can be broken down into three primary stages: creation, execution, and termination:

- **Creation**: A new process is created using the `fork()` system call, which creates a new process by duplicating an existing one. The parent process is the one that calls `fork()`, and the newly created process is the child. This mechanism is essential for the execution of new programs within the system and is a precursor to executing different tasks concurrently.
- **Execution**: After creation, the child process may execute the same code as the parent or use the `exec()` family of system calls to load and run a different program.

  If the parent process has more than one thread of execution, only the thread calling `fork()` is duplicated in the child process. Consequently, the child process contains a single thread: the one that executed the `fork()` system call.

  Since only the thread that called `fork()` is copied to the child, any **Mutual Exclusions** (**mutexes**), condition variables, or other synchronization primitives that were held by other threads at the time of the fork remain in their then-current state in the parent but do not carry over to the child. This can lead to complex synchronization issues, as mutexes that were locked by other threads (which do not exist in the child) might remain in a locked state, potentially causing deadlocks if the child tries to unlock or wait on these primitives.

  At this stage, the process performs its designated operations such as reading from or writing to files and communicating with other processes.
- **Termination**: A process terminates either voluntarily, by calling the `exit()` system call, or involuntarily, due to receiving a signal from another process that causes it to stop. Upon termination, the process returns an exit status to its parent process and releases its resources back to the system.

The process life cycle is integral to asynchronous operations as it enables the concurrent execution of multiple tasks.

Each process is uniquely identified by a **Process ID** (**PID**), an integer that the kernel uses to manage processes. PIDs are used to control and monitor processes. Parent processes also use PIDs to communicate with or control the execution of child processes, such as waiting for them to terminate or sending signals.

Linux provides mechanisms for process control and signaling, allowing processes to be managed and communicated with asynchronously. Signals are one of the primary means of IPC, enabling processes to interrupt or to be notified of events. For example, the `kill` command can send signals to stop a process or to prompt it to reload its configuration files.

Process scheduling is how the Linux kernel allocates CPU time to processes. The scheduler determines which process runs at any given time, based on scheduling algorithms and policies that aim to optimize for factors such as responsiveness and efficiency. Processes can be in various states, such as running, waiting, or stopped, and the scheduler transitions them between these states to manage execution efficiently.

Exploring IPC

In the Linux operating system, processes operate in isolation, meaning that they cannot directly access the memory space of other processes. This isolated nature of processes presents challenges when multiple processes need to communicate and synchronize their actions. To address these challenges, the Linux kernel provides a versatile set of IPC mechanisms. Each IPC mechanism is tailored to suit different scenarios and requirements, enabling developers to build complex, high-performance applications that leverage asynchronous processing effectively.

Understanding these IPC techniques is crucial for developers aiming to create scalable and efficient applications. IPC allows processes to exchange data, share resources, and coordinate their activities, facilitating smooth and reliable communication between different components of a software system. By utilizing the appropriate IPC mechanism, developers can achieve improved throughput, reduced latency, and enhanced concurrency in their applications, leading to better performance and user experiences.

In a multitasking environment, where multiple processes run concurrently, IPC plays a vital role in enabling the efficient and coordinated execution of tasks. For example, consider a web server application that handles multiple concurrent requests from clients. The web server process might use IPC to communicate with the child processes responsible for processing each request. This approach allows the web server to handle multiple requests simultaneously, improving the overall performance and scalability of the application.

Another common scenario where IPC is essential is in distributed systems or microservice architectures. In such environments, multiple independent processes or services need to communicate and collaborate to achieve a common goal. IPC mechanisms such as message queues and sockets or **Remote Procedure Calls** (**RPCs**) enable these processes to exchange messages, invoke methods on remote objects, and synchronize their actions, ensuring seamless and reliable IPC.

By leveraging the IPC mechanisms provided by the Linux kernel, developers can design systems where multiple processes can work together harmoniously. This enables the creation of complex, high-performance applications that utilize system resources efficiently, handle concurrent tasks effectively, and scale to meet increasing demands effortlessly.

IPC mechanisms in Linux

Linux supports several IPC mechanisms, each with its unique characteristics and use cases.

The fundamental IPC mechanisms supported by the Linux operating system include shared memory, which is commonly employed for process communication on a single server, and sockets, which facilitate inter-server communication. There are other mechanisms (which are briefly described here), but shared memory and sockets are the most commonly used:

- **Pipes and named pipes**: Pipes are one of the simplest forms of IPC, allowing for unidirectional communication between processes. A named pipe, or **First-in-First-out** (**FIFO**), extends this concept by providing a pipe that is accessible via a name in the filesystem, allowing unrelated processes to communicate.
- **Signals**: Signals are a form of software interrupt that can be sent to a process to notify it of events. While they are not a method for transferring data, signals are useful for controlling process behavior and triggering actions within processes.
- **Message queues**: Message queues allow processes to exchange messages in a FIFO manner. Unlike pipes, message queues support asynchronous communication, whereby messages are stored in a queue and can be retrieved by the receiving process at its convenience.
- **Semaphores**: Semaphores are used for synchronization, helping processes manage access to shared resources. They prevent race conditions by ensuring that only a specified number of processes can access a resource at any given time.
- **Shared memory**: Shared memory is a fundamental concept in IPC that enables multiple processes to access and manipulate the same segment of physical memory. It offers a blazing-fast method for exchanging data between different processes, reducing the need for time-consuming data copying operations. This technique is particularly advantageous when dealing with large datasets or requiring high-speed communication. The mechanism of shared memory involves creating a shared memory segment, which is a dedicated portion of physical memory accessible by multiple processes. This shared memory segment is treated as a common workspace, allowing processes to read, write, and collaboratively modify data. To ensure data integrity and prevent conflicts, shared memory requires synchronization mechanisms such as semaphores or mutexes. These mechanisms regulate access to the shared memory segment, preventing multiple processes from simultaneously modifying the same data. This coordination is crucial to maintain data consistency and avoid overwriting or corruption.

Shared memory is often the preferred IPC mechanism in single-server environments where performance is paramount. Its primary advantage lies in its speed. Since data is directly shared in physical memory without the need for intermediate copying or context switching, it significantly reduces communication overhead and minimizes latency.

However, shared memory also comes with certain considerations. It requires careful management to prevent race conditions and memory leaks. Processes accessing shared memory must adhere to well-defined protocols to ensure data integrity and avoid deadlocks. Additionally, shared memory is typically implemented as a system-level feature, requiring specific operating system support and potentially introducing platform-specific dependencies.

Despite these considerations, shared memory remains a powerful and widely used IPC technique, particularly in applications where speed and performance are critical factors.

- **Sockets**: Sockets are a fundamental mechanism for IPC in operating systems. They provide a way for processes to communicate with each other, either within the same machine or across networks. Sockets are used to establish and maintain connections between processes, and they support both **connection-oriented** and **connectionless communication**.

  Connection-oriented communication is a type of communication in which a reliable connection is established between two processes before any data is transferred. This type of communication is often used for applications such as file transfer and remote login, where it is important to ensure that all data is delivered reliably and in the correct order. Connectionless communication is a type of communication in which no reliable connection is established between two processes before data is transferred. This type of communication is often used for applications such as streaming media and real-time gaming, where it is more important to have low latency than to guarantee reliable delivery of all data.

  Sockets are the backbone of networked applications. They are used by a wide variety of applications, including web browsers, email clients, and file-sharing applications. Sockets are also used by many operating system services, such as the **Network File System** (**NFS**) and the **Domain Name System** (**DNS**).

  Here are some of the key benefits of using sockets:

  - **Reliability**: Sockets provide a reliable way to communicate between processes, even when those processes are located on different machines.
  - **Scalability**: Sockets can be used to support a large number of concurrent connections, making them ideal for applications that need to handle a lot of traffic.
  - **Flexibility**: Sockets can be used to implement a wide variety of communication protocols, making them suitable for a wide range of applications.
  - **Use in IPC**: Sockets are a powerful tool for IPC. They are used by a wide variety of applications and are essential for building scalable, reliable, and flexible networked applications.

Microservices-based applications are an example of asynchronous programming using different processes communicating between them in an asynchronous way. A simple example would be a log processor. Different processes generate log entries and send them to another process for further processing such as special formatting, deduplication, and statistics. The producers just send the lines of the log without waiting for any reply from the process they are sending to the log.

In this section, we saw processes in Linux, their life cycles, and how IPC is implemented by the operating system. In the next section, we will introduce a special kind of Linux process called **daemons**.

Services and daemons in Linux

In the realm of Linux operating systems, daemons are a fundamental component that runs quietly in the background, silently executing essential tasks without the direct involvement of an interactive user. These processes are traditionally identified by their names ending with the letter *d*, such as **sshd** for the **Secure Shell** (**SSH**) daemon and `httpd` for the **web server daemon**. They play a vital role in handling system-level tasks crucial for both the operating system and the applications running on it.

Daemons serve an array of purposes, ranging from file serving, web serving, and network communications to logging and monitoring services. They are designed to be autonomous and resilient, starting at system boot and running continuously until the system is shut down. Unlike regular processes initiated and controlled by users, daemons possess distinct characteristics:

- **Background operation**:
  - Daemons operate in the background
  - They lack a controlling terminal for direct user interaction
  - They do not require a user interface or manual intervention to perform their tasks
- **User independence**:
  - Daemons operate independently of user sessions
  - They function autonomously without direct user involvement
  - They wait for system events or specific requests to trigger their actions
- **Task-oriented focus**:
  - Each daemon is tailored to execute a specific task or a set of tasks
  - They are designed to handle specific functions or listen for particular events or requests
  - This ensures efficient task execution

Creating a daemon process involves more than merely running a process in the background. To ensure effective operation as a daemon, developers must consider several key steps:

1. **Detaching from the terminal**: The `fork()` system call is employed to detach the daemon from the terminal. The parent process exits after the fork, leaving the child process running in the background.
2. **Session creation**: The `setsid()` system call creates a new session and designates the calling process as the leader of both the session and the process group. This step is crucial for complete detachment from the terminal.
3. **Working directory change**: To prevent blocking the unmounting of the filesystem, daemons typically change their working directory to the root directory.
4. **File descriptor handling**: Inherited file descriptors are closed by daemons, and `stdin`, `stdout`, and `stderr` are often redirected to `/dev/null`.
5. **Signal handling**: Proper handling of signals, such as `SIGHUP` for configuration reloading or `SIGTERM` for graceful shutdown, is essential for effective daemon management.

Daemons communicate with other processes or daemons through various IPC mechanisms.

Daemons are integral to the architecture of many asynchronous systems, providing essential services without direct user interaction. Some prominent use cases of daemons include the following:

- **Web servers**: Daemons such as `httpd` and nginx serve web pages in response to client requests, handling multiple requests concurrently and ensuring seamless web browsing.
- **Database servers**: Daemons such as mysqld and postgresql manage database services, allowing for asynchronous access and manipulation of databases by various applications.
- **File servers**: Daemons such as **smbd** and **nfsd** provide networked file services, enabling asynchronous file sharing and access across different systems.
- **Logging and monitoring**: Daemons such as **syslogd** and **snmpd** collect and log system events, providing asynchronous monitoring of system health and performance.

In summary, daemons are essential components of Linux systems, silently performing critical tasks in the background to ensure smooth system operation and efficient application execution. Their autonomous nature and resilience make them indispensable for maintaining system stability and providing essential services to users and applications.

We have seen processes and demons, a special type of process. A process can have one or more threads of execution. In the next section, we will be introducing threads.

Threads

Processes and threads represent two fundamental ways of executing code concurrently, but they differ significantly in their operation and resource management. A process is an instance of a running program that owns its private set of resources, including memory, file descriptors, and execution context. Processes are isolated from each other, providing robust stability across the system since the failure of one process generally does not affect others.

Threads are a fundamental concept in computer science, representing a lightweight and efficient way to execute multiple tasks within a single process. In contrast to processes, which are independent entities with their own private memory space and resources, threads are closely intertwined with the process they belong to. This intimate relationship allows threads to share the same memory space and resources, including file descriptors, heap memory, and any other global data structures allocated by the process.

One of the key advantages of threads is their ability to communicate and share data efficiently. Since all threads within a process share the same memory space, they can directly access and modify common variables without the need for complex IPC mechanisms. This shared environment enables rapid data exchange and facilitates the implementation of concurrent algorithms and data structures.

However, sharing the same memory space also introduces the challenge of managing access to shared resources. To prevent data corruption and ensure the integrity of shared data, threads must employ synchronization mechanisms such as locks, semaphores, or mutexes. These mechanisms enforce rules and protocols for accessing shared resources, ensuring that only one thread can access a particular resource at any given time.

Effective synchronization is crucial in multithreaded programming to avoid race conditions, deadlocks, and other concurrency-related issues.

To address these challenges, various synchronization primitives and techniques have been developed. These include mutexes, which provide exclusive access to a shared resource, semaphores, which allow for controlled access to a limited number of resources, and condition variables, which enable threads to wait for specific conditions to be met before proceeding.

By carefully managing synchronization and employing appropriate concurrency patterns, developers can harness the power of threads to achieve high performance and scalability in their applications. Threads are particularly well-suited for tasks that can be parallelized, such as image processing, scientific simulations, and web servers, where multiple independent computations can be executed concurrently.

Threads, as described previously, are system threads. This means that they are created and managed by the kernel. However, there are scenarios, which we will explore in depth in *Chapter 8*, where we will require a multitude of threads. In such cases, the system might not have sufficient resources to create numerous system threads. The solution to this problem is the use of **user threads**. One approach to implementing user threads is through **coroutines**, which have been included in the C++ standard since C++20.

Coroutines are a relatively new feature in C++. Coroutines can be defined as functions that can be paused and resumed at specific points, allowing for cooperative multitasking within a single thread. Unlike standard functions that run from start to finish without interruption, coroutines can suspend their execution and yield control back to the caller, which can later resume the coroutine from the point it was paused.

Coroutines are much more lightweight than system threads. This means that they can be created and destroyed much more quickly, and that they require less overhead.

Coroutines are cooperative, which means that they must explicitly yield control to the caller in order to switch execution context. This can be a disadvantage in some cases, but it can also be an advantage, as it gives the user program more control over the execution of coroutines.

Coroutines can be used to create a variety of different concurrency patterns. For example, coroutines can be used to implement tasks, which are lightweight work units that can be scheduled and run concurrently. Coroutines can also be used to implement channels, which are communication channels that can pass data between them.

Coroutines can be classified into stackful and stackless categories. C++20 coroutines are stackless. We will see these concepts in depth in *Chapter 8*.

Overall, coroutines are a powerful tool for creating concurrent programs in C++. They are lightweight, cooperative, and can be used to implement a variety of different concurrency patterns. They cannot be used to implement parallelism entirely because coroutines still need CPU execution context, which can be only provided by a thread.

Thread life cycle

The life cycle of a system thread, often referred to as a lightweight process, encompasses the stages from its creation until its termination. Each stage plays a crucial role in managing and utilizing threads in a concurrent programming environment:

1. **Creation**: This phase begins when a new thread is created in the system. The creation process involves using the function, which takes several parameters. One critical parameter is the thread's attributes, such as its scheduling policy, stack size, and priority. Another essential parameter is the function that the thread will execute, known as the start routine. Upon its successful creation, the thread is allocated its own stack and other resources.
2. **Execution**: After creation, the thread starts executing its assigned start routine. During execution, the thread can perform various tasks independently or interact with other threads if necessary. Threads can also create and manage their own local variables and data structures, making them self-contained and capable of performing specific tasks concurrently.

3. **Synchronization**: To ensure orderly access to shared resources and prevent data corruption, threads employ synchronization mechanisms. Common synchronization primitives include locks, semaphores, and barriers. Proper synchronization allows threads to coordinate their activities, avoiding race conditions, deadlocks, and other issues that can arise in concurrent programming.
4. **Termination**: A thread can terminate in several ways. It can explicitly call the function to terminate itself. It can also terminate by returning from its start routine. In some cases, a thread can be canceled by another thread using the function. Upon termination, the system reclaims the resources allocated to the thread, and any pending operations or locks held by the thread are released.

Understanding the life cycle of a system thread is essential for designing and implementing concurrent programs. By carefully managing thread creation, execution, synchronization, and termination, developers can create efficient and scalable applications that leverage the benefits of concurrency.

Thread scheduling

System threads, managed by the operating system kernel's scheduler, are scheduled preemptively. The scheduler decides when to switch execution between threads based on factors such as thread priority, allocated time, or mutex blocking. This context switch, controlled by the kernel, can incur significant overhead. The high cost of context switches, coupled with the resource usage of each thread (such as its own stack), makes coroutines a more efficient alternative for some applications because we can run more than one coroutine in a single thread.

Coroutines offer several advantages. First, they reduce the overhead associated with context switches. Since context switching on coroutine yield or await is handled by the user space code rather than the kernel, the process is more lightweight and efficient. This results in significant performance gains, especially in scenarios where frequent context switching occurs.

Coroutines also provide greater control over thread scheduling. Developers can define custom scheduling policies based on the specific requirements of their application. This flexibility allows for fine-tuned thread management, resource utilization optimization, and desired performance characteristics achievement.

Another important feature of coroutines is that they are generally more lightweight compared to system threads. Coroutines don't maintain their own stack, which is a great resource consumption advantage, making them suitable for resource-constrained environments.

Overall, coroutines offer a more efficient and flexible approach to thread management, particularly in situations where frequent context switching is required or where fine-grained control over thread scheduling is essential. Threads can access the memory process and this memory is shared among all the threads, so we need to be careful and control memory access. This control is achieved by different mechanisms called synchronization primitives.

Synchronization primitives

Synchronization primitives are essential tools for managing concurrent access to shared resources in multithreaded programming. There are several synchronization primitives, each with its own specific purpose and characteristics:

- **Mutexes**: Mutexes are used to enforce exclusive access to critical sections of code. A mutex can be locked by a thread, preventing other threads from entering the protected section until the mutex is unlocked. Mutexes guarantee that only one thread can execute the critical section at any given time, ensuring data integrity and preventing race conditions.
- **Semaphores**: Semaphores are more versatile than mutexes and can be used for a wider range of synchronization tasks, including signaling between threads. A semaphore maintains an integer counter that can be incremented (signaling) or decremented (waiting) by threads. Semaphores allow for more complex coordination patterns, such as counting semaphores (for resource allocation) and binary semaphores (similar to mutexes).
- **Condition variables**: Condition variables are used for thread synchronization based on specific conditions. Threads can block (wait on) a condition variable until a particular condition becomes true. Other threads can signal the condition variable, causing waiting threads to wake up and continue execution. Condition variables are often used in conjunction with mutexes to achieve more fine-grained synchronization and avoid busy waiting.
- **Additional synchronization primitives**: In addition to the core synchronization primitives discussed previously, there are several other synchronization mechanisms:
    - **Barriers**: Barriers allow a group of threads to synchronize their execution, ensuring that all threads reach a certain point before proceeding further
    - **Read-write locks**: Read-write locks provide a way to control concurrent access to shared data, allowing multiple readers but only a single writer at a time
    - **Spinlocks**: Spinlocks are a type of mutex that involves busy waiting, continuously checking a memory location until it becomes available

In *Chapters 4* and *5*, we will see the synchronization primitives implemented in the C++ **Standard Template Library** (**STL**) in depth and examples of how to use them.

Choosing the right synchronization primitive

The choice of the appropriate synchronization primitive depends on the specific requirements of the application and the nature of the shared resources being accessed. Here are some general guidelines:

- **Mutexes**: Use mutexes when exclusive access to a critical section is required to ensure data integrity and prevent race conditions

- **Semaphores**: Use semaphores when more complex coordination patterns are needed, such as resource allocation or signaling between threads
- **Condition variables**: Use condition variables when threads need to wait for a specific condition to become true before proceeding

Effective use of synchronization primitives is crucial for developing safe and efficient multithreaded programs. By understanding the purpose and characteristics of different synchronization mechanisms, developers can choose the most suitable primitives for their specific needs and achieve reliable and predictable concurrent execution.

Common problems when using multiple threads

Threading introduces several challenges that must be managed to ensure application correctness and performance. These challenges arise from the inherent concurrency and non-deterministic nature of multithreaded programming.

- **Race conditions** occur when multiple threads access and modify shared data concurrently. The outcome of a race condition depends on the non-deterministic sequencing of threads' operations, which can lead to unpredictable and inconsistent results. For example, consider two threads that are updating a shared counter. If the threads increment the counter concurrently, the final value may be incorrect due to a race condition.
- **Deadlocks** occur when two or more threads wait indefinitely for resources held by each other. This creates a cycle of dependencies that cannot be resolved, causing the threads to become permanently blocked. For instance, consider two threads that are waiting for each other to release locks on shared resources. If neither thread releases the lock it holds, a deadlock occurs.
- **Starvation** occurs when a thread is perpetually denied access to resources it needs to make progress. This can happen when other threads continuously acquire and hold resources, leaving the starved thread unable to execute.
- **Livelocks** are like deadlocks, but instead of being permanently blocked, the threads remain active and repeatedly try to acquire resources, only without making any progress.

Several techniques can be used to manage the challenges of threading, including the following:

- **Synchronization mechanisms**: As described previously, synchronization primitives such as locks and mutexes can be used to control access to shared data and ensure that only one thread can access the data at a time.
- **Deadlock prevention and detection**: Deadlock prevention algorithms can be used to avoid deadlocks, while deadlock detection algorithms can be used to identify and resolve deadlocks when they occur.

- **Thread scheduling**: Thread scheduling algorithms can be used to determine which thread should run at any given time, as well as which can help to prevent starvation and improve application performance. We will see the different solutions to multithreading issues in much more detail.

Strategies for effective thread management

There are different ways to handle threads to avoid multithreading issues. The following are some of the most common ways to handle threads:

- **Minimize shared state**: Designing threads to operate on private data as much as possible significantly reduces the need for synchronization. By allocating memory for thread-specific data using thread-local storage, the need for global variables is eliminated, further reducing the potential for data contention. Careful management of shared data access through synchronization primitives is essential to ensure data integrity. This approach enhances the efficiency and correctness of multithreaded applications by minimizing the need for synchronization and ensuring that shared data is accessed in a controlled and consistent manner.
- **Lock hierarchy**: Establishing a well-defined lock hierarchy is crucial for preventing deadlocks in multithreaded programming. A lock hierarchy dictates the order in which locks are acquired and released, ensuring a consistent locking pattern across threads. By acquiring locks in a hierarchical manner, from the coarsest to the finest granularity, the possibility of deadlocks is significantly reduced.

  The coarsest level of granularity refers to locks that control access to a large portion of the shared resource, while the finest granularity locks are used for specific, fine-grained parts of the resource. By acquiring the coarse-grained lock first, threads can gain exclusive access to a larger section of the resource, reducing the likelihood of conflicts with other threads attempting to access the same resource. Once the coarse-grained lock is acquired, finer-grained locks can be obtained to control access to specific parts of the resource, providing more granular control and reducing the waiting time for other threads.

  In some cases, lock-free data structures can be employed to eliminate the need for locks altogether. Lock-free data structures are designed to provide concurrent access to shared resources without explicit locks. Instead, they rely on atomic operations and non-blocking algorithms to ensure data integrity and consistency. By utilizing lock-free data structures, the overhead associated with lock acquisition and release is eliminated, resulting in improved performance and scalability in multithreaded applications:
- **Timeouts**: To prevent threads from waiting indefinitely when trying to acquire a lock, it is important to set timeouts for lock acquisition. This ensures that if a thread cannot acquire the lock within the specified timeout period, it will automatically give up and try again later. This helps prevent deadlocks and ensures that no thread is left waiting indefinitely.

- **Thread pools**: Managing a pool of reusable threads is a key technique for optimizing the performance of multithreaded applications. By creating and destroying threads dynamically, the overhead of thread creation and termination can be reduced significantly. The size of the thread pool should be tuned based on the application's workload and resource constraints. A too-small pool may result in tasks waiting for available threads, while a too-large pool may waste resources. Work queues are used to manage tasks and assign them to available threads in the pool. Tasks are added to the queue and processed by the threads in a FIFO order. This ensures fairness and prevents the starvation of tasks. The use of work queues also allows for load balancing, as tasks can be distributed evenly across the available threads.
- **Synchronization primitives**: Understand the different types of synchronization primitives, such as mutexes, semaphores, and condition variables. Choose the appropriate primitive based on the synchronization requirements of the specific scenario. Use synchronization primitives correctly to avoid race conditions and deadlocks.
- **Testing and debugging**: Test multi-threaded applications thoroughly to identify and fix threading issues. Use tools such as thread sanitizers and profilers to detect data races and performance bottlenecks. Employ debugging techniques such as step-by-step execution and thread dumps to analyze and resolve threading problems. We will see testing and debugging in *Chapters 11* and *12*.
- **Scalability and performance considerations**: Design thread-safe data structures and algorithms to ensure scalability and performance. Balance the number of threads with the available resources to avoid over-subscription. Monitor system metrics such as CPU utilization and thread contention to identify potential performance bottlenecks.
- **Communication and collaboration**: Foster collaboration among developers working on multi-threaded code to ensure consistency and correctness. Establish coding guidelines and best practices for thread management to maintain code quality and readability. Regularly review and update the threading strategy as the application evolves.

Threading is a powerful tool that can be used to improve the performance and scalability of applications. However, it is important to understand the challenges of threading and to use appropriate techniques to manage these challenges. By doing so, developers can create multithreaded applications that are correct, efficient, and reliable.

Summary

In this chapter, we explored the concept of processes in operating systems. Processes are fundamental entities that execute programs and manage resources on the computer. We delved into the process life cycle, examining the various stages a process goes through from creation to termination. Additionally, we discussed IPC, which is crucial for processes to interact and exchange information with each other.

Furthermore, we introduced daemons in the context of Linux operating systems. Daemons are special types of processes that run in the background as services and perform specific tasks such as managing system resources, handling network connections, or providing other essential services to the system. We also explored the concepts of system and user threads, which are lightweight processes that share the same address space as the parent process. We discussed the advantages of multithreaded applications, including improved performance and responsiveness, as well as the challenges associated with managing and synchronizing multiple threads within a single process.

Knowing the different issues created by multithreading is fundamental to understanding how to fix them. In the next chapter, we will see how to create threads, and then in *Chapter 4* and *Chapter 5*, we will study the different synchronization primitives the standard C++ offers and their different applications in depth.

Further reading

- [Butenhof, 1997] David R. Butenhof, *Programming with POSIX Threads*, Addison Wesley, 1997.
- [Kerrisk, 2010] Michael Kerrisk, *The Linux Programming Interface*, No Starch Press, 2010.
- [Stallings, 2018] William Stallings, *Operating Systems Internals and Design Principles*, Ninth Edition, Pearson Education 2018.
- [Williams, 2019] Anthony Williams, *C++ Concurrency in Action*, Second Edition, Manning 2019.

Part 2: Advanced Thread Management and Synchronization Techniques

In this part, we build upon the foundational knowledge of parallel programming and dive deeper into advanced techniques for managing threads and synchronizing concurrent operations. We will explore essential concepts such as thread creation and management, exception handling across threads, and efficient thread coordination, acquiring a solid understanding of key synchronization primitives, including mutexes, semaphores, condition variables, and atomic operations. All this knowledge will equip us with the tools needed to implement both lock-based and lock-free multithreaded solutions, offering a glimpse into high-performance concurrent systems, and providing the skills necessary to avoid common pitfalls such as race conditions, deadlocks, and livelocks when managing multithreaded systems.

This part has the following chapters:

- *Chapter 3*, *How to Create and Manage Threads in C++*
- *Chapter 4*, *Thread Synchronization with Locks*
- *Chapter 5*, *Atomic Operations*

3

How to Create and Manage Threads in C++

As we learned in the previous two chapters, threads are the smallest and most lightweight units of execution within a program. Each thread takes care of a unique task defined by a sequence of instructions running on allocated CPU resources by the OS scheduler. Threads play a critical role when managing concurrency within a program aiming to maximize the overall utilization of CPU resources.

During the program's startup process, after the kernel passes the execution to the process, the C++ runtime creates the main thread and executes the `main()` function. After that, additional threads can be created to split the program into different tasks that can run concurrently and share resources. This way, the program can handle multiple tasks, improving efficiency and responsiveness.

In this chapter, we will learn the basics of how to create and manage threads using modern C++ features. In the subsequent chapters, we will come across explanations of C++ lock synchronization primitives (mutexes, semaphores, barriers, and spinlocks), lock-free synchronization primitives (atomic variables), coordination synchronization primitives (condition variables), and approaches using C++ to solve or avoid potential problems when using concurrency or multithreading (race conditions or data races, deadlocks, livelocks, starvation, oversubscription, load balancing, and thread exhaustion).

In this chapter, we are going to cover the following main topics:

- How to create, manage, and cancel threads in C++
- How to pass arguments to threads and get results back from the thread
- How to sleep a thread or yield execution to other threads
- What jthread objects are and why they are useful

Technical requirements

In this chapter, we will develop different solutions using C++11 and C++20. Therefore, we will need to install the **GNU Compiler Collection** (**GCC**), specifically GCC 13, as well as Clang 8 (see `https://en.cppreference.com/w/cpp/compiler_support` for further details on C++ compiler support).

You can find more information about GCC at `https://gcc.gnu.org`. You can find information on how to install GCC here: `https://gcc.gnu.org/install/index.html`.

For more information about Clang, a compiler frontend supporting several languages including C++, visit `https://clang.llvm.org`. Clang is part of the LLVM compiler infrastructure project (`https://llvm.org`). C++ support in Clang is documented here: `https://clang.llvm.org/cxx_status.html`.

In this book, some code snippets do not show the libraries included. Additionally, some functions, even the main ones, might be simplified, showing only the relevant instructions. You can find all the complete code in the following GitHub repository: `https://github.com/PacktPublishing/Asynchronous-Programming-with-CPP`.

Under the `scripts` folder in the root directory in the preceding GitHub repository, you can find a script called `install_compilers.sh` that might be of help with installing the required compilers in Debian-based Linux systems. The script has been tested in Ubuntu 22.04 and 24.04.

The examples for this chapter are located under the `Chapter_03` folder. All source code files can be compiled using C++20 with CMake as follows:

```
cmake . && cmake –build .
```

Executables will be generated under the `bin` directory.

The thread library – an introduction

The main library to create and manage threads in C++ is the thread library. First, let's go through a recap about threads. Then we will dive into what the thread library offers.

What are threads? Let's do a recap

The purpose of threads is to execute multiple simultaneous tasks in a process.

As we have seen in the previous chapter, a thread has its own stack, local data, and CPU registers such as **Instruction Pointer** (**IP**) and **Stack Pointer** (**SP**), but shares the address space and virtual memory of its parent process.

In the user space, we can differentiate between **native threads** and **lightweight or virtual threads**. Native threads are the ones created by the OS when using some kernel APIs. The C++ thread objects create and manage these types of threads. On the other hand, lightweight threads are like native threads, except that they are emulated by a runtime or library. In C++, **coroutines** belong to this group. As described in the previous chapter, lightweight threads have faster context switching than native threads. Also, multiple lightweight threads can run in the same native thread and can be much smaller than native threads.

In this chapter, we will start learning about native threads. In *Chapter 8*, we will learn about lightweight threads in the form of coroutines.

The C++ thread library

In C++, threads allow multiple functions to run concurrently. The `thread` class defines a type-safe interface to a native thread. This class is defined in the `std::thread` library, in the `<thread>` header file in the **Standard Template Library** (**STL**). It is available from C++11 onward.

Before the inclusion of the thread library in the C++ STL, developers used platform-specific libraries such as the POSIX thread (`pthread`) library in Unix or Linux OSs, the **C Runtime** (**CRT**) and Win32 libraries for Windows NT and CE systems, or third-party libraries such as `Boost.Threads`. In this book, we will only use modern C++ features. As `<thread>` is available and provides a portable abstraction on top of platform-specific mechanisms, none of these libraries will be used or explained. In *Chapter 9*, we will introduce `Boost.Asio`, and in *Chapter 10*, `Boost.Cobalt`. Both libraries provide advanced frameworks to deal with asynchronous I/O operations and coroutines.

Now it's time to learn about the different thread operations.

Thread operations

In this section, we will learn how to create threads, pass arguments during their construction, return values from threads, cancel threads execution, catch exceptions, and much more.

Thread creation

When a thread is created, it executes immediately. It is only delayed by the OS scheduling process. If there are not enough resources to run both parent and child threads in parallel, the order in which they will run is not defined.

The constructor argument defines the function or `function` object to be executed by the thread. This callable object should not return anything, as its return value will be ignored. If for some reason the thread execution ends with an exception, `std::terminate` is called unless an exception is caught, as we will see later in this chapter.

In the following examples, we create six threads using different callable objects.

`t1` using a function pointer:

```
void func() {
    std::cout << "Using function pointer\n";
}
std::thread t1(func);
```

`t2` using a lambda function:

```
auto lambda_func = []() {
    std::cout << "Using lambda function\n";
};
std::thread t2(lambda_func);
```

`t3` using an embedded lambda function:

```
std::thread t3([]() {
    std::cout << "Using embedded lambda function\n";
});
```

`t4` using a `function` object where `operator()` is overloaded:

```
class FuncObjectClass {
   public:
    void operator()() {
        std::cout << "Using function object class\n";
    }
};
std::thread t4{FuncObjectClass()};
```

`t5` using a non-static `member` function by passing the address of the `member` function and the address of an object to call the `member` function:

```
class Obj {
  public:
    void func() {
        std::cout << "Using a non-static member function"
                  << std::endl;
    }
};
Obj obj;
std::thread t5(&Obj::func, &obj);
```

`t6` using a static `member` function where only the address of the `member` function is needed as the method is static:

```
class Obj {
  public:
    static void static_func() {
        std::cout << "Using a static member function\n";
    }
};
std::thread t6(&Obj::static_func);
```

Thread creation incurs some overhead that can be reduced by using thread pools, as we will explore in *Chapter 4*.

Checking hardware concurrency

One of the strategies for effective thread management, which is related to scalability and performance and was commented on in the previous chapter, is to balance the number of threads with the available resources to avoid over-subscription.

To retrieve the number of concurrent threads supported by the OS, we can use the `std::thread::hardware_concurrency()` function:

```
const auto processor_count = std::thread::hardware_concurrency();
```

The value returned by this function must be considered to only provide a hint about the number of threads that will run concurrently. It is also sometimes not well defined, thus returning a value of `0`.

Synchronized stream writing

When we print messages to the console by using `std::cout` from two or more threads, the output result can be messy. This is due to a **race condition** happening in the output stream.

As commented in the previous chapter, race conditions are bugs in software that happen in concurrent and multithreaded programs, whose behavior depends on the sequence of events happening on a shared resource where at least one of the actions is not atomic. We will learn more about how to avoid them in *Chapter 4*. Additionally, we will learn how to debug race conditions using Clang's sanitizers in *Chapter 12*.

The following code snippet shows two threads printing a sequence of numbers. The `t1` thread should print lines containing the `1 2 3 4` sequence. The `t2` thread should print the `5 6 7 8` sequence. Each thread prints its sequence 100 times. Before the main thread exits, it waits for `t1` and `t2` to finish by using `join()`.

More about joining threads later in this chapter.

```
#include <iostream>
#include <thread>

int main() {
    std::thread t1([]() {
        for (int i = 0; i < 100; ++i) {
            std::cout << "1 " << "2 " << "3 " << "4 "
                      << std::endl;
        }
    });
    std::thread t2([]() {
        for (int i = 0; i < 100; ++i) {
            std::cout << "5 " << "6 " << "7 " << "8 "
                      << std::endl;
        }
    });
    t1.join();
    t2.join();
    return 0;
}
```

However, running the previous example shows some lines with the following content:

```
6 1 2 3 4
1 5 2 6 3 4 7 8
1 2 3 5 6 7 8
```

To avoid these issues, we can simply write from a specific thread or use a `std::ostringstream` object that makes atomic calls to the `std::cout` object:

```
std::ostringstream oss;
oss << "1 " << "2 " << "3 " << "4 " << "\n";
std::cout << oss.str();
```

From C++20 onward, we also can use the `std::osyncstream` objects. They behave similarly to `std::cout` but with writing synchronization between threads accessing the same stream. However, as only the transfer step from its internal buffer to the output stream is synchronized, every thread needs its own `std::osyncstream` instance.

The internal buffer is transferred when the stream is destroyed, which is when `emit()` is explicitly called.

The following is a simple solution to allow synchronization on each printed line:

```
#include <iostream>
#include <syncstream>
#include <thread>

#define sync_cout std::osyncstream(std::cout)

int main() {
    std::thread t1([]() {
        for (int i = 0; i < 100; ++i) {
            sync_cout << "1 " << "2 " << "3 " << "4 "
                      << std::endl;
        }
    });
    std::thread t2([]() {
        for (int i = 0; i < 100; ++i) {
            sync_cout << "5 " << "6 " << "7 " << "8 "
                      << std::endl;
        }
    });
    t1.join();
    t2.join();
    return 0;
}
```

Both solutions will output the sequences without interleaving them

```
1 2 3 4
1 2 3 4
5 6 7 8
```

As this approach is now the official C++20 way of avoiding race conditions when outputting content, we will use `std::osyncstream` as the default approach throughout the rest of this book.

Sleeping the current thread

`std::this_thread` is a namespace. It gives access to functions from the current thread to yield the execution to another thread or block the execution of the current task and wait for a period.

The `std::this_thread::sleep_for` and `std::this_thread::sleep_until` functions block the execution of the thread for a given amount of time.

`std::this_thread::sleep_for` sleeps for at least a given duration. The blockage can be longer depending on how the OS scheduler decides to run tasks, or due to some resource contention delays.

Resource contention

Resource contention occurs when demand exceeds supply for a certain shared resource, leading to performance degradation.

`std::this_thread::sleep_until` works like `std::this_thread::sleep_for`. However, instead of sleeping for a duration, it sleeps until a specific time point has been reached. The clock where the time point is computed must meet the `Clock` requirement (you can find more information on this here: `https://en.cppreference.com/w/cpp/named_req/Clock`). It is recommended by the standard to use a steady clock instead of the system clock to set up the duration.

Identifying a thread

When debugging multithreaded solutions, it is useful to know which thread is executing a given function. Each thread can be identified by an identifier, making it possible to log in its value for traceability and debugging.

`std::thread::id` is a lightweight class that defines a unique identifier of thread objects (`std::thread` and `std::jthread`, which we will introduce later in this chapter). This identifier is retrieved by using the `get_id()` function.

Thread identifier objects can be compared, serialized, and printed via an output stream. They can also be used as a key in mapping containers, as they are supported by the `std::hash` function.

The following example prints the identifier of the `t` thread. Later in this chapter, we will learn how to create a thread and sleep for an interval of time:

```
#include <chrono>
#include <iostream>
#include <thread>

using namespace std::chrono_literals;

void func() {
    std::this_thread::sleep_for(1s);
}

int main() {
    std::thread t(func);
    std::cout << "Thread ID: " << t.get_id() << std::endl;
    t.join();
    return 0;
}
```

Remember that when a thread finishes, its identifier can be reused by a future thread.

Passing arguments

Arguments can be passed to the thread by value, by reference, or as pointers.

Here we can see how to pass arguments by value:

```
void funcByValue(const std::string& str, int val) {
    sync_cout << «str: « << str << «, val: « << val
              << std::endl;
}
std::string str{"Passing by value"};
std::thread t(funcByValue, str, 1);
```

Passing by value avoids data races. However, it is much more costly, as data is copied across.

The next example shows how to pass values by reference:

```
void modifyValues(std::string& str, int& val) {
    str += " (Thread)";
    val++;
}
std::string str{"Passing by reference"};
int val = 1;
std::thread t(modifyValues, std::ref(str), std::ref(val));
```

Or as `const-reference`:

```
void printVector(const std::vector<int>& v) {
    sync_cout << "Vector: ";
    for (int num : v) {
        sync_cout << num << " ";
    }
    sync_cout << std::endl;
}
std::vector<int> v{1, 2, 3, 4, 5};
std::thread t(printVector, std::cref(v));
```

Passing by reference is achieved by using `ref()` (non-const references) or `cref()` (const-references). Both are defined in the `<functional>` header file. This lets the variadic template define the thread constructor to treat the argument as a reference.

These helper functions are used to generate `std::reference_wrapper` objects, which wrap a reference in a copyable and assignable object. Missing these functions when passing arguments makes the arguments to be passed by value.

You can also move an object into a thread as follows:

```
std::thread t(printVector, std::move(v));
```

However, note that trying to access the `v` vector in the main thread after it is moved into the `t` thread would result in undefined behavior.

Finally, we can also allow threads to access variables by lambda captures:

```
std::string str{"Hello"};
std::thread t([&]() {
    sync_cout << "str: " << str << std::endl;
});
```

In this example, the `str` variable is accessed by the `t` thread as a reference captured by the embedded lambda function.

Returning values

To return values that have been computed in a thread, we can use a shared variable with a synchronization mechanism such as a mutex, lock, or atomic variable.

In the following code snippet, we can see how to return a value computed by a thread by using an argument passed by a non-const reference (using `ref()`). The `result` variable is computed within the `t` thread in the `func` function. The resulting value can be seen from the main thread. As we will learn in the next section, the `join()` function simply waits for the `t` thread to finish before letting the main thread continue running and checking the `result` variable afterward:

```
#include <chrono>
#include <iostream>
#include <random>
#include <syncstream>
#include <thread>

#define sync_cout std::osyncstream(std::cout)

using namespace std::chrono_literals;

namespace {
int result = 0;
};

void func(int& result) {
    std::this_thread::sleep_for(1s);
    result = 1 + (rand () % 10);
```

```
}

Int main() {
    std::thread t(func, std::ref(result));
    t.join();
    sync_cout << "Result: " << result << std::endl;
}
```

The `reference` argument can be a reference to the input object itself, or to another variable where we want the result to be stored, as done in this example with the `result` variable.

We can also return the value using a lambda capture, as shown in the following example:

```
std::thread t([&]() { func(result); });
t.join();
sync_cout << "Result: " << result << std::endl;
```

We can also do this by writing into a shared variable protected by a mutex, locking the mutex (using `std::lock_guard`, for example) before executing the writing operation. However, we will dive deeper into these mechanisms in *Chapter 4*:

```
#include <chrono>
#include <iostream>
#include <mutex>
#include <random>
#include <syncstream>
#include <thread>

#define sync_cout std::osyncstream(std::cout)

using namespace std::chrono_literals;

namespace {
int result = 0;
std::mutex mtx;
};

void funcWithMutex() {
    std::this_thread::sleep_for(1s);
    int localVar = 1 + (rand() % 10);
    std::lock_guard<std::mutex> lock(mtx);
    result = localVar;
}
Int main() {
    std::thread t(funcWithMutex);
```

```
    t.join();
    sync_cout << "Result: " << result << std::endl;
}
```

There is a more elegant way of returning values from threads. This involves using futures and promises, which we will learn about in *Chapter 6*.

Moving threads

Threads can be moved but not copied. This is to avoid having two different thread objects to represent the same hardware thread.

In the following example, `t1` is moved to `t2` using `std::move`. Therefore, `t2` inherits the same identifier as `t1` had before being moved and `t1` is not joinable, as it no longer contains any valid thread anymore:

```
#include <chrono>
#include <thread>

using namespace std::chrono_literals;

void func() {
    for (auto i=0; i<10; ++i) {
        std::this_thread::sleep_for(500ms);
    }
}

int main() {
    std::thread t1(func);
    std::thread t2 = std::move(t1);
    t2.join();
    return 0;
}
```

When a `std::thread` object is moved to another `std::thread` object, the move-from-thread object will reach a state where it does not represent a real thread anymore. This situation also happens to thread objects resulting from the default constructor after detaching or joining them. We will introduce these operations in the next section.

Waiting for a thread to finish

There are use cases where a thread needs to wait for another thread to finish so that it can use the result computed by the latter thread. Other use cases involve running a thread in the background, detaching it, and continuing to execute the main thread.

Joining a thread

The `join()` function blocks the current thread while waiting for the completion of the joining thread identified by the thread object where the `join()` function is being called. This ensures that the joining thread has terminated after `join()` returns (see the *Thread life cycle* section in *Chapter 2* for more details).

It is easy to forget to use a `join()` function. **Joining Thread** (**jthread**) solves that problem. It is available from C++20 onward. We will introduce it in the next section.

Checking whether a thread is joinable

A thread is considered joinable and therefore active if the `join()` function has not been called in that thread. This is true even if the thread has finished executing code but still has not been joined. On the other hand, a default constructed thread or a thread that has already been joined is *not* joinable.

To check whether a thread is joinable, just use the `std::thread::joinable()` function.

Let us see the usage of `std::thread::join()` and `std::thread::joinable()` in the following example:

```
#include <chrono>
#include <iostream>
#include <thread>

using namespace std::chrono_literals;

void func() {
    std::this_thread::sleep_for(100ms);
}

int main() {
    std::thread t1;
    std::cout << "Is t1 joinable? " << t1.joinable()
              << std::endl;

    std::thread t2(func);
    t1.swap(t2);
    std::cout << "Is t1 joinable? " << t1.joinable()
              << std::endl;
    std::cout << "Is t2 joinable? " << t2.joinable()
              << std::endl;

    t1.join();
    std::cout << "Is t1 joinable? " << t1.joinable()
```

```
                << std::endl;
}
```

After `t1` has been constructed using the default constructor (not specifying a callable object), the thread will not be joinable. As `t2` is constructed specifying a function, `t2` is joinable after construction. However, when `t1` and `t2` are swapped, `t1` becomes joinable again and `t2` is not joinable anymore. Then the main thread waits for `t1` to join, so it is no longer joinable. Trying to join `t2`, a non-joinable thread, results in undefined behavior. Finally, not joining a joinable thread will lead to resource leaks or potential program crashes due to the unexpected use of shared resources.

Daemon thread by detaching

If we want a thread to continue running in the background as a **daemon thread** but finish the execution of the current thread, we can use the `std::thread::detach()` function. A daemon thread is a thread that performs some tasks in the background that do not need to run to completion. If the main program exits, all daemon threads are terminated. As commented earlier, a thread must join or detach before the main thread terminates, otherwise the program will abort its execution.

After calling `detach`, the detached thread cannot be controlled or joined (as it is waiting for its completion) using the `std::thread` object, as this object no longer represents the detached thread.

The following example shows a daemon thread called `t` that is detached just after construction, running the `daemonThread()` function in the background. This function executes for three seconds and then exits, finishing the thread execution. Meanwhile, the main thread sleeps for one more second than the thread execution time before exiting:

```
#include <chrono>
#include <iostream>
#include <syncstream>
#include <thread>

#define sync_cout std::osyncstream(std::cout)

using namespace std::chrono_literals;

namespace {
int timeout = 3;
}

void daemonThread() {
    sync_cout << "Daemon thread starting...\n";
    while (timeout-- > 0) {
        sync_cout << "Daemon thread is running...\n";
```

```
        std::this_thread::sleep_for(1s);
    }
    sync_cout << "Daemon thread exiting...\n";
}

int main() {
    std::thread t(daemonThread);
    t.detach();

    std::this_thread::sleep_for(
              std::chrono::seconds(timeout + 1));

    sync_cout << "Main thread exiting...\n";
    Return 0;
}
```

Joining threads – the jthread class

From C++20 onward, there is a new class: `std::jthread`. This class is like `std:thread` but with the additional functionality that the thread rejoins on destruction, following the **Resource Acquisition is Initialization** (**RAII**) technique. It can be canceled or stopped in some scenarios.

As you can see in the following example, a `jthread` thread has the same interface as `std::thread`. The only difference is that we do not need to call the `join()` function to ensure that the main thread waits for the `t` thread to join:

```
#include <chrono>
#include <iostream>
#include <thread>

using namespace std::chrono_literals;

void func() {
    std::this_thread::sleep_for(1s);
}

int main() {
    std::jthread t(func);
    sync_cout << "Thread ID: " << t.get_id() << std::endl;
    return 0;
}
```

When two `std::jthreads` are destroyed, their destructors are called in reverse order from their constructors. To demonstrate this behavior, let us implement a thread wrapper class that prints some messages when the wrapped thread is created and destroyed:

```
#include <chrono>
#include <functional>
#include <iostream>
#include <syncstream>
#include <thread>

#define sync_cout std::osyncstream(std::cout)

using namespace std::chrono_literals;

class JthreadWrapper {
   public:
    JthreadWrapper(
       const std::function<void(const std::string&)>& func,
       const std::string& str)
        : t(func, str), name(str) {
        sync_cout << "Thread " << name
                  << " being created" << std::endl;
    }
    ~JthreadWrapper() {
        sync_cout << "Thread " << name
                  << " being destroyed" << std::endl;
    }
   private:
    std::jthread t;
    std::string name;
};
```

Using this `JthreadWrapper` wrapper class, we start three threads that execute the `func` function. Each will wait for a second before exiting:

```
void func(const std::string& name) {
    sync_cout << "Thread " << name << " starting...\n";
    std::this_thread::sleep_for(1s);
    sync_cout << "Thread " << name << " finishing...\n";
}

int main() {
```

```
    JthreadWrapper t1(func, «t1»);
    JthreadWrapper t2(func, "t2");
    JthreadWrapper t3(func, "t3");
    std::this_thread::sleep_for(2s);
    sync_cout << "Main thread exiting..." << std::endl;
    return 0;
}
```

This program will show the following output:

```
Thread t1 being created
Thread t1 starting...
Thread t2 being created
Thread t2 starting...
Thread t3 being created
Thread t3 starting...
Thread t1 finishing...
Thread t2 finishing...
Thread t3 finishing...
Main thread exiting...
Thread t3 being destroyed
Thread t2 being destroyed
Thread t1 being destroyed
```

As we can see, `t1` is created first, then `t2`, and finally `t3`. The destructors follow the reverse order, with `t3` being destroyed first, then `t2`, and `t1` last.

As jthreads avoid pitfalls when we forget to use `join` in a thread, we simply prefer to use `std::jthread` over `std::thread`. There might be cases where we need to use explicit calls to `join()` to be sure that threads have been joined and resources properly freed up before moving to another task.

Yielding thread execution

A thread can also decide to pause its execution, let the implementation reschedule the execution of threads, and give the chance to other threads to run.

The `std::this_thread::yield` method provides a hint to the OS to reschedule another thread. The behavior is implementation-dependent, depending on the OS scheduler and the current state of the system.

Some Linux implementations suspend the current thread and move it back to a queue of threads to schedule all threads with the same priority. If this queue is empty, the yield has no effect.

The following example shows two threads, `t1` and `t2`, executing the same work function. They randomly choose to either do some work (locking a mutex, as we will learn about in the next chapter, and waiting for three seconds) or yield the execution to the other thread:

```
#include <iostream>
#include <random>
#include <string>
#include <syncstream>
#include <thread>

#define sync_cout std::osyncstream(std::cout)

using namespace std::chrono;

namespace {
int val = 0;
std::mutex mtx;
}

int main() {
    auto work = [&](const std::string& name) {
        while (true) {
            bool work_to_do = rand() % 2;
            if (work_to_do) {
                sync_cout << name << ": working\n";
                std::lock_guard<std::mutex> lock(mtx);
                for (auto start = steady_clock::now(),
                          now = start;
                          now < start + 3s;
                          now = steady_clock::now()) {
                }
            } else {
                sync_cout << name << ": yielding\n";
                std::this_thread::yield();
            }
        }
    };
    std::jthread t1(work, "t1");
    std::jthread t2(work, "t2");
    return 0;
}
```

When running this example, when the execution reaches the yield command, we can see how the thread that is currently running stops and enables the other thread to restart its execution.

Threads cancellation

If we are no longer interested in the result that a thread is computing, we will want to cancel that thread and avoid more computation costs.

Killing a thread could be a solution. However, that leaves resources that belong to the thread handling, such as other threads started from that thread, locks, connections, and so on. This could mean ending the program with undefined behavior, a critical section locked under a mutex, or any other unexpected issue.

To avoid these problems, we need a data race-free mechanism to let the thread know about the intention to stop its execution (to request a stop) so that the thread can take all the specific steps needed to cancel its work and terminate gracefully.

One of the possible ways of achieving this is by using an atomic variable that is periodically checked by the thread. We will explore atomic variables at length in the next chapter. For now, let's define an atomic variable as a variable that many threads can write or read from without any locking mechanism or data race due to its atomic transaction operations and memory model.

As an example, let us create a `Counter` class that calls a callback every second. This is done infinitely until the `running` atomic variable is set to `false`, when a caller uses the `stop()` function:

```
#include <chrono>
#include <functional>
#include <iostream>
#include <syncstream>
#include <thread>

#define sync_cout std::osyncstream(std::cout)

using namespace std::chrono_literals;

class Counter {
    using Callback = std::function<void(void)>;
  public:
    Counter(const Callback &callback) {
        t = std::jthread([&]() {
            while (running.load() == true) {
                callback ();
                std::this_thread::sleep_for(1s);
            }
        });
    }
    void stop() { running.store(false); }
  private:
```

```
    std::jthread t;
    std::atomic_bool running{true};
};
```

In the caller function, we will instantiate `Counter` as follows. Then, when desired (here, that is after three seconds), we will call the `stop()` function, letting `Counter` exit the loop and terminate the thread execution:

```
int main() {
    Counter counter([&]() {
        sync_cout << "Callback: Running...\n";
    });
    std::this_thread::sleep_for(3s);
    counter.stop();
}
```

Since C++20, there has been a new mechanism called **cooperative interruption** of a thread. This is available via `std::stop_token`.

The thread knows that a stop was requested by checking the result of calling the `std::stop_token::stop_requested()` function.

To produce `stop_token`, we will use a `stop_source` object via the `std::stop_source::get_token()` function.

This thread cancellation mechanism is implemented in `std::jthead` objects via an internal member of the `std::stop_source` type where the shared stop state is stored. The `jthread` constructor accepts `std::stop_token` as its first argument. This is used when a stop is requested during execution.

Therefore, `std::jthread` exposes some additional functions to manage stop tokens compared with `std::thread` objects. These functions are `get_stop_source()`, `get_stop_token()`, and `request_stop()`.

When `request_stop()` is called, it issues a stop request to the internal stop state, which is atomically updated to avoid race conditions (you will learn more about atomic variables in *Chapter 4*).

Let us check how all these functions work in the following example.

First, we will define a template function to show the properties of a stop item object (`stop_token` or `stop_source`):

```
#include <chrono>
#include <iostream>
#include <string_view>
#include <syncstream>
```

```
#include <thread>

#define sync_cout std::osyncstream(std::cout)

using namespace std::chrono_literals;

template <typename T>
void show_stop_props(std::string_view name,
                     const T& stop_item) {
    sync_cout << std::boolalpha
              << name
              << ": stop_possible = "
              << stop_item.stop_possible()
              << ", stop_requested = "
              << stop_item.stop_requested()
              << '\n';
};
```

Now, within the `main()` function, we will start a worker thread, acquire its stop token object, and show its properties:

```
auto worker1 = std::jthread(func_with_stop_token);
std::stop_token stop_token = worker1.get_stop_token();
show_stop_props("stop_token", stop_token);
```

`Worker1` is running the `func_with_stop_token()` function that is defined in the ensuing code block. In this function, the stop token is checked by using the `stop_requested()` function. If this function returns `true`, a stop was requested, so the function simply returns, terminating the thread execution. Otherwise, it runs the next loop iteration, sleeping the current thread for further 300 ms until the next stop request check:

```
void func_with_stop_token(std::stop_token stop_token) {
    for (int i = 0; i < 10; ++i) {
        std::this_thread::sleep_for(300ms);
        if (stop_token.stop_requested()) {
            sync_cout << "stop_worker: "
                      << "Stopping as requested\n";
            return;
        }
        sync_cout << "stop_worker: Going back to sleep\n";
    }
}
```

We can request a stop from the main thread by using the stop token returned by the thread object as follows:

```
worker1.request_stop();
worker1.join();
show_stop_props("stop_token after request", stop_token);
```

Also, we can request a stop from a different thread. For this, we need to pass a `stop_source` object. In the following code snippet, we can see how a thread stopper is created with a `stop_source` object as an argument, acquired from the `worker2` worker thread:

```
auto worker2 = std::jthread(func_with_stop_token);
std::stop_source stop_source = worker2.get_stop_source();
show_stop_props("stop_source", stop_source);

auto stopper = std::thread( [](std::stop_source source) {
        std::this_thread::sleep_for(500ms);
        sync_cout << "Request stop for worker2 "
                  << "via source\n";
        source.request_stop();
    }, stop_source);

stopper.join();
std::this_thread::sleep_for(200ms);
show_stop_props("stop_source after request", stop_source);
```

The `stopper` thread waits for 0.5 seconds and requests a stop from the `stop_source` object. Then `worker2` becomes aware of that request and terminates its execution, as explained earlier.

We can also register a callback function that will invoke a function when a stop is requested via a stop token or stop source. This can be done by using the `std::stop_callback` object, as shown in the ensuing code block:

```
std::stop_callback callback(worker1.get_stop_token(), []{
    sync_cout << "stop_callback for worker1 "
              << "executed by thread "
              << std::this_thread::get_id() << '\n';
});

sync_cout << "main_thread: "
          << std::this_thread::get_id() << '\n';
std::stop_callback callback_after_stop(
    worker2.get_stop_token(),[] {
```

```
        sync_cout << "stop_callback for worker2 "
                  << "executed by thread "
                  << std::this_thread::get_id() << '\n';
});
```

If a `std::stop_callback` object is destroyed, its execution is prevented. For example, this scoped stop callback will not execute, as the callback object is destroyed when going out of scope:

```
{
    std::stop_callback scoped_callback(
        worker2.get_stop_token(), []{
          sync_cout << "Scoped stop callback "
                    << "will not execute\n";
      }
    );
}
```

After a stop has already been requested, a new stop callback object will execute immediately. In the following example, if a stop has been requested for `worker2`, `callback_after_stop` will execute the lambda function just after construction:

```
sync_cout << "main_thread: "
          << std::this_thread::get_id() << '\n';
std::stop_callback callback_after_stop(
    worker2.get_stop_token(), []{
        sync_cout << "stop_callback for worker2 "
                  << "executed by thread "
                  << std::this_thread::get_id() << '\n';
    }
);
```

Catching exceptions

Any unhandled exception thrown within a thread needs to be caught within that thread. Otherwise, the C++ runtime calls `std::terminate`, causing the program to terminate abruptly. This causes unexpected behavior, data loss, or even program crashes.

One solution is to use try-catch blocks within the thread to catch exceptions. However, only exceptions thrown within that thread will be caught. Exceptions do not propagate to other threads.

To propagate an exception to another thread, one thread can capture it and store it into a `std::exception_ptr` object, then use shared memory techniques to pass it to another thread, where the `std::exception_ptr` object will be checked and the exception re-thrown if needed.

The following example shows this approach:

```
#include <atomic>
#include <chrono>
#include <exception>
#include <iostream>
#include <mutex>
#include <thread>

using namespace std::chrono_literals;

std::exception_ptr captured_exception;
std::mutex mtx;

void func() {
    try {
        std::this_thread::sleep_for(1s);
        throw std::runtime_error(
                  "Error in func used within thread");
    } catch (...) {
        std::lock_guard<std::mutex> lock(mtx);
        captured_exception = std::current_exception();
    }
}

int main() {
    std::thread t(func);
    while (!captured_exception) {
        std::this_thread::sleep_for(250ms);
        std::cout << „In main thread\n";
    }
    try {
        std::rethrow_exception(captured_exception);
    } catch (const std::exception& e) {
        std::cerr << "Exception caught in main thread: "
                  << e.what() << std::endl;
    }
    t.join();
}
```

Here, we can see how a `std::runtime_error` exception is thrown when executing the `func` function by the `t` thread. The exception is caught and stored in `captured_exception`, a `std::exception_ptr` shared object protected by a mutex. The type and value of the thrown exception are determined by calling the `std::current_exception()` function.

In the main thread, the `while` loop runs until an exception is captured. The exception is re-thrown in the main thread by calling `std::rethrow_exception(captured_exception)`. It is caught again by the main thread where the `catch` block is executed, printing a message to the console via the `std::cerr` error stream.

We will learn a better solution in *Chapter 6* by using futures and promises.

Thread-local storage

Thread-local Storage (**TLS**) is a memory management technique that allows each thread to have its own instance of a variable. This technique allows threads to store thread-specific data that is not accessible by other threads, avoiding race conditions and improving performance. This is because the overhead of synchronization mechanisms to access these variables is removed.

TLS is implemented by the OS and accessible by using the `thread_local` keyword, which has been available since C++11. `thread_local` provides a uniform way to use the TLS capabilities of many OSs and avoid compiler-specific language extensions for accessing the TLS feature (some examples of such extensions are the TLS Windows API, the `__declspec(thread)` MSVC compiler language extension, or the `__thread` GCC compiler language extension).

To use TLS with compilers that do not support C++11 or newer versions, use `Boost.Library`. This provides the `boost::thread_specific_ptr` container, which implements portable TLS.

Thread-local variables can be declared as follows:

- Globally
- In namespaces
- As class static member variables
- Inside functions; it has the same effect as variables allocated with the `static` keyword, meaning that the variables are allocated for the lifetime of the program and their value are carried through the next function call

The following examples show three threads calling the `multiplyByTwo` function with different arguments. This function sets the value of the `val` thread-local variable to the argument value, multiplies it by 2, and prints to the console:

```
#include <iostream>
#include <syncstream>
#include <thread>

#define sync_cout std::osyncstream(std::cout)

thread_local int val = 0;

void setValue(int newval) { val = newval; }
```

```
void printValue() { sync_cout << val << ' '; }
void multiplyByTwo(int arg) {
    setValue(arg);
    val *= 2;
    printValue();
}

int main() {
    val = 1;  // Value in main thread
    std::thread t1(multiplyByTwo, 1);
    std::thread t2(multiplyByTwo, 2);
    std::thread t3(multiplyByTwo, 3);
    t1.join();
    t2.join();
    t3.join();
    std::cout << val << std::endl;
}
```

Running this code snippet will show the following output:

```
2 4 6 1
```

Here, we can see that each thread operated on its input argument, resulting in `t1` printing `2`, `t2` printing `4`, and `t3` printing `6`. The main thread running the main function can also access its thread local variable, `val`, which has a value that is set to `1` when the program starts but only used when printed out to console at the end of the main function before exiting the program.

As with any technique, there are some drawbacks. TLS increases memory usage, as a variable is created per thread, so it could be problematic in resource-constrained environments. Also, accessing TLS variables might have some overhead compared with regular variables. This can be problematic in performance-critical software.

Using many of the techniques we have learned so far, let's build a timer.

Implementing a timer

Let us implement a timer that accepts intervals and callback functions. The timer will execute the callback function at each interval. Also, the user will be able to stop the timer by calling its `stop()` function.

The following snippet shows an implementation of the timer:

```
#include <chrono>
#include <functional>
#include <iostream>
#include <syncstream>
```

```
#include <thread>

#define sync_cout std::osyncstream(std::cout)

using namespace std::chrono_literals;
using namespace std::chrono;

template<typename Duration>
class Timer {
   public:
    typedef std::function<void(void)> Callback;
    Timer(const Duration interval,
          const Callback& callback) {
        auto value = duration_cast<milliseconds>(interval);
        sync_cout << "Timer: Starting with interval of "
                  << value << std::endl;

        t = std::jthread([&](std::stop_token stop_token) {
            while (!stop_token.stop_requested()) {
                sync_cout << "Timer: Running callback "
                          << val.load() << std::endl;
                val++;
                callback();
                sync_cout << "Timer: Sleeping...\n";
                std::this_thread::sleep_for(interval);
            }
            sync_cout << „Timer: Exit\n";
        });
    }
    void stop() {
        t.request_stop();
    }
   private:
    std::jthread t;
    std::atomic_int32_t val{0};
};
```

The `Timer` constructor accepts a `Callback` function (a `std::function<void(void)>` object) and a `std::chrono::duration` object defining the period or interval when the callback will be executed.

Then a `std::jthread` object is created with a lambda expression, whereby a loop calls the callback in intervals of time. This loop checks whether a stop has been requested via `stop_token`, which is enabled by using the `stop() Timer` API function. When this is the case, the loop exits and the thread terminates.

Here is how to use it:

```
int main(void) {
    sync_cout << "Main: Create timer\n";
    Timer timer(1s, [&]() {
        sync_cout << "Callback: Running...\n";
    });

    std::this_thread::sleep_for(3s);
    sync_cout << "Main thread: Stop timer\n";
    timer.stop();

    std::this_thread::sleep_for(500ms);
    sync_cout << "Main thread: Exit\n";
    return 0;
}
```

In this example, we started the timer that will print the `Callback: Running` message every second. After three seconds, the main thread will call the `timer.stop()` function, terminating the timer thread. The main thread then waits for 500 milliseconds before exiting.

This is the output:

```
Main: Create timer
Timer: Starting with interval of 1000ms
Timer: Running callback 0
Callback: Running...
Timer: Sleeping...
Timer: Running callback 1
Callback: Running...
Timer: Sleeping...
Timer: Running callback 2
Callback: Running...
Timer: Sleeping...
Main thread: Stop timer
Timer: Exit
Main thread: Exit
```

As an exercise, you can slightly modify this example to implement a timeout class that calls a callback function if there is no input event within a given timeout interval. This is a common pattern when dealing with network communications where a packet replay request is sent if no packets have been received for some time.

Summary

In this chapter, we learned how to create and manage threads, how to pass arguments or retrieve results, how TLS works, and how to wait for a thread to finish. We also learned how to make a thread yield control to others or cancel its execution. If something goes wrong and an exception is thrown, we now know how to pass the exception between threads and avoid an unexpected program termination. Finally, we implemented a `timer` class that periodically runs a callback function.

In the next chapter, we will learn about thread safety, mutual exclusion, and atomic operations. That will include mutexes, locking and lock-free algorithms, and memory synchronization ordering, among other topics. That knowledge will help us develop thread-safe data structures and algorithms.

Further reading

- Compiler support: `https://en.cppreference.com/w/cpp/compiler_support`
- GCC releases: `https://gcc.gnu.org/releases.html`
- Clang: `https://clang.llvm.org`
- Clang 8 documentation: `https://releases.llvm.org/8.0.0/tools/clang/docs/index.html`
- LLVM project: `https://llvm.org`
- Boost.Threads: `https://www.boost.org/doc/libs/1_78_0/doc/html/thread.html`
- P0024 – Technical Specification for Parallelism: `https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0024r0.html`
- TLS proposal: `https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2659.htm`
- *Thread Launching for C++0X*: `https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2184.html`
- TLS from IBM: `https://docs.oracle.com/cd/E19683-01/817-3677/chapter8-1/index.html`
- *Data that is private to a thread*: `https://www.ibm.com/docs/en/i/7.5?topic=techniques-data-that-is-private-thread`
- **Resource Acquisition Is Initialization** (**RAII**): `https://en.cppreference.com/w/cpp/language/raii`
- Bjarne Stroustrup, *A Tour of C++*, Third Edition, 18.2 and 18.7.

4

Thread Synchronization with Locks

In *Chapter 2*, we learned that threads can read and write memory shared by the process they belong to. While the operating system implements process memory access protection, there is no such protection for threads accessing shared memory in the same process. Concurrent memory write operations to the same memory address from multiple threads require synchronization mechanisms to avoid data races and ensure data integrity.

In this chapter, we will describe in detail the problems created by concurrent access to shared memory by multiple threads and how to fix them. We are going to study in detail the following topics:

- Race conditions – what they are and how they can happen
- Mutual exclusion as a synchronization mechanism and how it is implemented in C++ by `std::mutex`
- Generic lock management
- What condition variables are and how to use them with mutexes
- Implementing a fully synchronized queue using `std::mutex` and `std::condition_variable`
- The new synchronization primitives introduced with C++20 – semaphores, barriers, and latches

These are all lock-based synchronization mechanisms. Lock-free techniques are the subject of the next chapter.

Technical requirements

The technical requirements for this chapter are the same as for the concepts explained in the previous chapter, and to compile and run the examples, a C++ compiler with C++20 support is required (for semaphores, latches, and barriers examples). Most of the examples require just C++11. Examples have been tested on Linux Ubuntu LTS 24.04.

The code in this chapter can be found on GitHub:

`https://github.com/PacktPublishing/Asynchronous-Programming-with-CPP`

Understanding race conditions

A race condition happens when the outcome of running a program depends on the sequence in which its instructions are executed. We will begin with a very simple example to show how race conditions happen, and later in this chapter, we will learn how to resolve this problem.

In the following code, the `counter` global variable is incremented by two threads running concurrently:

```
#include <iostream>
#include <thread>

int counter = 0;

int main() {
    auto func = [] {
        for (int i = 0; i < 1000000; ++i) {
            counter++;
        }
    };

    std::thread t1(func);
    std::thread t2(func);

    t1.join();
    t2.join();

    std::cout << counter << std::endl;
    return 0;
}
```

After running the preceding code three times, we get the following `counter` values:

```
1056205
1217311
1167474
```

We see two main issues here: first, the value of `counter` is incorrect; second, every execution of the program ends with a different value of `counter`. The results are non-deterministic and most frequently incorrect. If you are very lucky, you may get the right values, but that is very unlikely.

This scenario involves two threads, `t1` and `t2`, that run concurrently and modify the same variable, which is essentially some memory region. It seems like it should work fine because there is only one line of code that increases the `counter` value and thus modifies the memory content (by the way, it doesn't matter if we use the post-increment operator like in `counter++` or the pre-increment operator like in `++counter`; the results will be equally wrong).

Looking closer at the preceding code, let's study the following line carefully:

```
        counter++;
```

It increments `counter` in three steps:

- The contents of the memory address where the `counter` variable is stored are loaded into a CPU register. In this case, an `int` data type is loaded from memory into a CPU register.
- The value in the register is incremented by one.
- The value in the register is stored in the `counter` variable memory address.

Now, let us consider a possible scenario when two threads attempt to increment the counter concurrently. Let us look at *Table 4.1*:

THREAD 1	**THREAD 2**
[1] Load counter value into register	[3] Load counter value into register
[2] Increment register value	[5] Increment register value
[4] Store register in counter	[6] Store register in counter

Table 4.1: Two threads incrementing the counter concurrently

Thread 1 executes [1] and loads the current value of the counter (let's assume it is 1) into a CPU register. Then, it increments the value in the register by one [2] (now, the register value is 2).

Thread 2 is scheduled for execution and [3] loads the current value of the counter (remember – it has not been modified yet, so it is still 1) into a CPU register.

Now, thread 1 is scheduled again for execution and [4] stores the updated value into memory. The value of `counter` is now equal to two.

Finally, thread 2 is scheduled again, and [5] and [6] are executed. The register value is incremented by one and then the value two is stored in memory. The `counter` variable has been incremented just once when it should have been incremented twice and its value should be three.

The previous issue happened because the increment operation on the counter is not atomic. If each thread could execute the three instructions required to increment the `counter` variable without being interrupted, `counter` would be incremented twice as expected. However, depending on the order in which the operations are executed, the result can be different. This is called a **race condition**.

To avoid race conditions, we need to ensure that shared resources are accessed and modified in a controlled manner. One way to achieve this is by using locks. A **lock** is a synchronization primitive that allows only one thread to access a shared resource at a time. When a thread wants to access a shared resource, it must first acquire the lock. Once the thread has acquired the lock, it can access the shared resource without interference from other threads. When the thread has finished accessing the shared resource, it must release the lock so that other threads can access it.

Another way to avoid race conditions is by using **atomic operations**. An atomic operation is an operation that is guaranteed to be executed in a single, indivisible step. This means that no other thread can interfere with an atomic operation while it is being executed. Atomic operations are typically implemented using hardware instructions that are designed to be indivisible. Atomic operations will be explained in *Chapter 5*.

In this section, we have seen the most common and important problem created by multithreaded code: race conditions. We have seen how, depending on the order of the operations performed, the results can be different. With this problem in mind, we are going to study how to solve it in the next section.

Why do we need mutual exclusion?

Mutual exclusion is a fundamental concept in concurrent programming that ensures that multiple threads or processes do not simultaneously access a shared resource such as a shared variable, a critical section of code, or a file or network connection. Mutual exclusion is crucial for preventing race conditions such as the one we have seen in the previous section.

Imagine a small coffee shop with a single espresso machine. The machine can only make one espresso at a time. This means the machine is a critical resource that all baristas must share.

The coffee shop is attended by three baristas: Alice, Bob, and Carol. They use the coffee machine *concurrently*, but they cannot use it simultaneously because that could create problems: Bob puts the right amount of freshly ground coffee in the machine and starts making an espresso. Then, Alice does the same but first removes the coffee from the machine, thinking that Bob just forgot to do it. Bob then takes the espresso from the machine, and after that, Alice finds that there is no espresso! This is a disaster – a real-life version of our counter program.

To fix the problems in the coffee shop, they may appoint Carol as a machine manager. Before using the machine, both Alice and Bob ask her if they can start making a new espresso. That would solve the issue.

Back to our counter program, if we could allow just one thread at a time to access `counter` (what Carol did in the coffee shop), our software problem would be solved too. Mutual exclusion is a mechanism that can be used to control concurrent thread access to memory. The C++ Standard Library provides the `std::mutex` class, a synchronization primitive used to protect shared data from being simultaneously accessed by two or more threads.

This new version of the code we saw in the previous section implements two ways of concurrently incrementing `counter`: free access, as in the previous section, and synchronized access using mutual exclusion:

```
#include <iostream>
#include <mutex>
#include <thread>

std::mutex mtx;
int counter = 0;

int main() {
    auto funcWithoutLocks = [] {
        for (int i = 0; i < 1000000; ++i) {
            ++counter;
        };
    };

    auto funcWithLocks = [] {
        for (int i = 0; i < 1000000; ++i) {
            mtx.lock();
            ++counter;
            mtx.unlock();
        };
    };

    {
        counter = 0;
        std::thread t1(funcWithoutLocks);
        std::thread t2(funcWithoutLocks);

        t1.join();
        t2.join();

        std::cout << "Counter without using locks: " << counter <<
std::endl;
    }
    {
        counter = 0;
        std::thread t1(funcWithLocks);
        std::thread t2(funcWithLocks);

        t1.join();
```

```
        t2.join();

        std::cout << "Counter using locks: " << counter << std::endl;
    }

    return 0;
}
```

When a thread runs `funcWithLocks`, it acquires a lock with `mtx.lock()` before incrementing `counter`. Once `counter` has been incremented, the thread releases the lock (`mtx.unlock()`).

The lock can only be owned by one thread. If, for example, `t1` acquires the lock and then `t2` tries to acquire it too, `t2` will be blocked and will wait until the lock is available. Because only one thread can own the lock at any time, this synchronization primitive is called a **mutex** (from *mutual exclusion*). If you run this program a few times, you will always get the correct result: `2000000`.

In this section, we introduced the concept of mutual exclusion and learned that the C++ Standard Library provides the `std::mutex` class as a primitive for thread synchronization. In the next section, we will study `std::mutex` in detail.

C++ Standard Library mutual exclusion implementation

In the previous section, we introduced the concept of mutual exclusion and mutexes and why they are needed to synchronize concurrent memory access. In this section, we will see the classes provided by the C++ Standard Library to implement mutual exclusion. We will also see some helper classes the C++ Standard Library provides to make the use of mutexes easier.

The following table summarizes the mutex classes provided by the C++ Standard Library and their main features:

Mutex Type	Access	Recursive	Timeout
`std::mutex`	EXCLUSIVE	NO	NO
`std::recursive_mutex`	EXCLUSIVE	YES	NO
`std::shared_mutex`	1 - EXCLUSIVE N - SHARED	NO	NO
`std::timed_mutex`	EXCLUSIVE	NO	YES
`std::recursive_timed_mutex`	EXCLUSIVE	YES	YES
`std::shared_timed_mutex`	1 - EXCLUSIVE N - SHARED	NO	YES

Table 4.2: Mutex classes in C++ Standard Library

Let us explore these classes one by one.

std::mutex

The `std::mutex` class was introduced in C++11 and is one of the most important and most frequently used synchronization primitives provided by the C++ Standard Library.

As we have seen earlier in this chapter, `std::mutex` is a synchronization primitive that can be used to protect shared data from being simultaneously accessed by multiple threads.

The `std::mutex` class offers exclusive, non-recursive ownership semantics.

The main features of `std::mutex` are the following:

- A calling thread owns the mutex from the time it successfully calls `lock()` or `try_lock()` until it calls `unlock()`.
- A calling thread must not own the mutex before calling `lock()` or `try_lock()`. This is the non-recursive ownership semantics property of `std::mutex`.
- When a thread owns a mutex, all other threads will block (when calling `lock()`) or receive a `false` return value (when calling `try_lock()`). This is the exclusive ownership semantics of `std::mutex`.

If a thread owning a mutex tries to acquire it again, the resulting behavior is undefined. Usually, an exception is thrown when this happens, but this is implementation-defined.

If, after a thread releases a mutex, it tries to release it again, this is also undefined behavior (as in the previous case).

A mutex being destroyed while a thread has it locked or a thread terminating without releasing the lock are also causes of undefined behavior.

The `std::mutex` class has three methods:

- `lock()`: Calling `lock()` acquires the mutex. If the mutex is already locked, then the calling thread is blocked until the mutex is unlocked. From the application's point of view, it is as if the calling thread waits for the mutex to be available.
- `try_lock()`: When called, this function returns either `true`, indicating that the mutex has been successfully locked, or `false` in the event of the mutex being already locked. Note that `try_lock` is non-blocking, and the calling thread either acquires the mutex or not, but it is not blocked like when calling `lock()`. The `try_lock()` method is generally used when we don't want the thread to wait until the mutex is available. We will call `try_lock()` when we want the thread to proceed with some processing and try to acquire the mutex later.
- `unlock()`: Calling `unlock()` releases the mutex.

std::recursive_mutex

The `std::mutex` class offers exclusive, non-recursive ownership semantics. While exclusive ownership semantics are always required at least for a thread (it is a mutual exclusion mechanism, after all), in some instances, we may need to recursively acquire the mutex. For example, a recursive function may need to acquire a mutex. We may also need to acquire a mutex in function `g()` called from another function `f()`, which acquired the same mutex.

The `std::recursive_mutex` class offers exclusive, recursive semantics. Its main features are the following:

- A calling thread may acquire the same mutex more than once. It will own the mutex until it releases the mutex the same number of times it acquired it. For example, if a thread recursively acquires a mutex three times, it will own the mutex until it releases it for the third time.
- The maximum number of times a recursive mutex can be recursively acquired is unspecified and hence implementation-defined. Once a mutex has been acquired for the maximum number of times, calls to `lock()` will throw `std::system_error`, and calls to `try_lock()` will return `false`.
- Ownership is the same as for `std::mutex`: if a thread owns a `std::recursive_mutex` class, any other threads will block if they try to acquire it by calling `lock()`, or they will get false as a return when calling `try_lock()`.

The `std::recursive_mutex` interface is exactly the same as for `std::mutex`.

std::shared_mutex

Both `std::mutex` and `std::shared_mutex` have exclusive ownership semantics, and just one thread can be the mutex owner at any given time. There are some cases, though, when we may need to let several threads simultaneously access the protected data and give just one thread exclusive access.

The counter example required exclusive access to a single variable for every thread because they were all updating `counter` values. Now, if we have threads that only require reading the current value in `counter` and just one thread to increment its value, it would be much better to let the reader threads access `counter` concurrently and give the writer exclusive access.

This functionality is implemented using what is called a Readers-Writer lock. The C++ Standard Library implements the `std::shared_mutex` class, with a similar (but not exactly the same) functionality.

The main difference between `std::shared_mutex` and other mutex types is that it has two access levels:

- **Shared**: Several threads can share the ownership of the same mutex. Shared ownership is acquired/released calling `lock_shared()`, `try_lock_shared()`/`unlock shared()`. While at least one thread has acquired shared access to the lock, no other thread can get exclusive access to it, but it can acquire shared access.

- **Exclusive**: Only one thread can own the mutex. Exclusive ownership is acquired/released by calling `lock()`, `try_lock()`/`unlock()`. While a thread has acquired exclusive access to the lock, no other thread can acquire either shared or exclusive access to it.

Let's see a simple example using `std::shared_mutex`:

```
#include <algorithm>
#include <chrono>
#include <iostream>
#include <shared_mutex>
#include <thread>

int counter = 0;

int main() {
    using namespace std::chrono_literals;

    std::shared_mutex mutex;

    auto reader = [&] {
        for (int i = 0; i < 10; ++i) {
            mutex.lock_shared();
            // Read the counter and do something
            mutex.unlock_shared();
        }
    };

    auto writer = [&] {
        for (int i = 0; i < 10; ++i) {
            mutex.lock();
            ++counter;
            std::cout << "Counter: " << counter << std::endl;
            mutex.unlock();

            std::this_thread::sleep_for(10ms);
        }
    };

    std::thread t1(reader);
    std::thread t2(reader);
    std::thread t3(writer);
    std::thread t4(reader);
    std::thread t5(reader);
    std::thread t6(writer);
```

```
    t1.join();
    t2.join();
    t3.join();
    t4.join();
    t5.join();
    t6.join();

    return 0;
}
```

The example uses `std::shared_mutex` to synchronize six threads: two threads are writers, and they increment the value of `counter` and require exclusive access. The remaining four threads just read `counter` and only require shared access. Also, note that in order to use `std::shared_mutex`, we need to include the `<shared_mutex>` header file.

Timed mutex types

The mutex types we have seen until now behave in the same way when we want to acquire the lock for exclusive use:

- `std::lock()`: The calling thread blocks until the lock is available
- `std::try_lock()`: Returns `false` if the lock is not available

In the case of `std::lock()`, the calling thread may be waiting for a long time, and we may need to just wait for a certain period of time and then let the thread proceed with some processing if it has not been able to acquire the lock.

To achieve this goal, we can use the timed mutexes provided by the C++ Standard Library: `std::timed_mutex`, `std::recursive_timed_mutex`, and `std::shared_time_mutex`.

They are similar to their non-timed counterparts and implement the following additional functions to allow waiting for the lock to be available for a specific period of time:

- `try_lock_for()`: Tries to lock the mutex and blocks the thread until the specified time duration has elapsed (timed out). If the mutex is locked before the specified time duration, then it returns `true`; otherwise, it returns `false`.

  If the specified time duration is less than or equal to zero (`timeout_duration.zero()`), then the function behaves exactly like `try_lock()`.

  This function may block for longer than the specified duration due to scheduling or contention delays.

- `try_lock_until()`: Tries to lock the mutex until the specified timeout time or the mutex is locked, whichever comes first. In this case, we specify an instance in the future as a limit for the waiting.

The following example shows how to use `std::try_lock_for()`:

```
#include <algorithm>
#include <chrono>
#include <iostream>
#include <mutex>
#include <thread>
#include <vector>

constexpr int NUM_THREADS = 8;
int counter = 0;
int failed = 0;

int main() {
    using namespace std::chrono_literals;

    std::timed_mutex tm;
    std::mutex m;

    auto worker = [&] {
        for (int i = 0; i < 10; ++i) {
            if (tm.try_lock_for(10ms)) {
                ++counter;
                std::cout << "Counter: " << counter << std::endl;
                std::this_thread::sleep_for(10ms);
                m.unlock();
            }
            else {
                m.lock();
                ++failed;
                std::cout << "Thread " << std::this_thread::get_id()
<< " failed to lock" << std::endl;
                m.unlock();
            }

            std::this_thread::sleep_for(12ms);
        }
    };

    std::vector<std::thread> threads;
```

```
    for (int i = 0; i < NUM_THREADS; ++i) {
        threads.emplace_back(worker);
    }

    for (auto& t : threads) {
        t.join();
    }

    std::cout << "Counter: " << counter << std::endl;
    std::cout << "Failed: " << failed << std::endl;

    return 0;
}
```

The preceding code uses two locks: `tm`, a timed mutex, to synchronize access to `counter` and writing to the screen if acquiring `tm` is successful, and `m`, a non-timed mutex, to synchronize access to `failed` and writing to the screen if acquiring `tm` is not successful.

Problems when using locks

We have seen examples using just a mutex (lock). If we only need one mutex and we acquire and release it properly, in general is not very difficult to write correct multithreaded code. Once we need more than one lock, the code complexity increases. Two common problems when using multiple locks are *deadlock* and *livelock*.

Deadlock

Let's consider the following scenario: to perform a certain task, a thread needs to access two resources, and they cannot be accessed simultaneously by two or more threads (we need mutual exclusion to properly synchronize access to the required resources). Each resource is synchronized with a different `std::mutex` class. In this case, a thread must acquire the first resource mutex then acquire the second resource mutex, and finally process the resources and release both mutexes.

When two threads try performing the aforementioned processing, something like this may happen:

Thread 1 and *thread 2* need to acquire two mutexes to perform the required processing. *Thread 1* acquires the first mutex and *thread 2* acquires the second mutex. Then, *thread 1* will be blocked forever waiting for the second mutex to be available, and *thread 2* will be blocked forever waiting for the first mutex to be available. This is called a **deadlock** because both threads will be blocked forever waiting for each other to release the required mutex.

This is one of the most common issues in multithreaded code. In *Chapter 11*, about debugging, we will learn how to spot this problem by inspecting the running (deadlocked) program with a debugger.

Livelock

A possible solution for deadlock could be the following: when a thread tries to acquire the lock, it will block just for a limited time, and if still unsuccessful, it will release any lock it may have acquired.

For example, *thread 1* acquires the first lock and *thread 2* acquires the second lock. After a certain time, *thread 1* still has not acquired the second lock, so it releases the first one. *Thread 2* may finish waiting too and release the lock it acquired (in this example, the second lock).

This solution may work sometimes, but it is not right. Imagine this scenario: *Thread 1* has acquired the first lock and has acquired the second lock. After some time, both threads release their already acquired locks, and then they acquire the same locks again. Then, the threads release the locks, then acquire them again, and so on.

The threads are unable to do anything but acquire a lock, wait, release the lock, and do the same again. This situation is called **livelock** because the threads are not just waiting forever (as in the deadlock case), but they are kind of alive and acquire and release a lock continuously.

The most common solution for both deadlock and livelock situations is acquiring the locks in a consistent order. For example, if a thread needs to acquire two locks, it will always acquire the first lock first, and then it will acquire the second lock. The locks will be released in the opposite order (first releasing the second lock and then the first). If a second thread tries to acquire the first lock, it will have to wait until the first thread releases both locks, and deadlock will never happen.

In this section, we have seen the mutex classes provided by the C++ Standard Library. We have studied their main features and the issues we may experience when using more than one lock. In the next section, we will see the mechanisms that the C++ Standard Library provides to make acquiring and releasing mutexes. easier.

Generic lock management

In the previous section, we saw the different types of mutexes provided by the C++ Standard Library. In this section, we will see the provided classes to make the use of mutexes easier. This is done by using different wrapper classes. The following table summarizes the lock management classes and their main features:

Mutex Manager Class	Supported Mutex Types	Mutexes Managed
`std::lock_guard`	All	1
`std::scoped_lock`	All	Zero or more
`std::unique_lock`	All	1
`std::shared_lock`	`std::shared_mutex` `std::shared_timed_mutex`	1

Table 4.3: Lock management classes and their features

Let's see each of the mutex management classes and their main features.

std::lock_guard

The `std::lock_guard` class is a **Resource Acquisition Is Initialization** (**RAII**) class that makes it easier to use mutexes and guarantees that a mutex will be released when the `lock_guard` destructor is called. This is very useful, for example, when dealing with exceptions.

The following code shows the use of `std::lock_guard` and how it makes handling exceptions easier when a lock is already acquired:

```
#include <format>
#include <iostream>
#include <mutex>
#include <thread>

std::mutex mtx;
uint32_t counter{};

void function_throws() { throw std::runtime_error("Error"); }

int main() {
    auto worker = [] {
        for (int i = 0; i < 1000000; ++i) {
            mtx.lock();
            counter++;
            mtx.unlock();
        }
    };

    auto worker_exceptions = [] {
        for (int i = 0; i < 1000000; ++i) {
            try {
                std::lock_guard<std::mutex> lock(mtx);
                counter++;
                function_throws();
            } catch (std::system_error& e) {
                std::cout << e.what() << std::endl;
                return;
            } catch (...) {
                return;
            }
        }
```

```
    };

    std::thread t1(worker_exceptions);
    std::thread t2(worker);

    t1.join();
    t2.join();

    std::cout << "Final counter value: " << counter << std::endl;
}
```

The `function_throws()` function is just a utility function that will throw an exception.

In the previous code example, the `worker_exceptions()` function is executed by `t1`. In this case, the exception is handled to print meaningful messages. The lock is not explicitly acquired/released. This is delegated to `lock`, a `std::lock_guard` object. When the lock is constructed, it wraps the mutex and calls `mtx.lock()`, acquiring the lock. When `lock` is destroyed, the mutex is released automatically. In the event of an exception, the mutex will also be released because the scope where `lock` was defined is exited.

There is another constructor implemented for `std::lock_guard`, receiving a parameter of type `std::adopt_lock_t`. Basically, this constructor makes it possible to wrap an already acquired non-shared mutex, which will be released automatically in the `std::lock_guard` destructor.

std::unique_lock

The `std::lock_guard` class is just a simple `std::mutex` wrapper that automatically acquires the mutex in its constructor (the thread will be blocked, waiting until the mutex is released by another thread) and releases the mutex in its destructor. This is very useful, but sometimes we need more control. For example, `std::lock_guard` will either call `lock()` on the mutex or assume the mutex is already acquired. We may prefer or really need to call `try_lock`. We also may want the `std::mutex` wrapper not to acquire the lock in its constructor; that is, we may want to defer the locking until a later moment. All this functionality is implemented by `std::unique_lock`.

The `std::unique_lock` constructor accepts a tag as its second parameter to indicate what we want to do with the underlying mutex. There are three options:

- `std::defer_lock`: Does not acquire ownership of the mutex. The mutex is not locked in the constructor, and it will not be unlocked in the destructor if it is never acquired.
- `std::adopt_lock`: Assumes that the mutex has been acquired by the calling thread. It will be released in the destructor. This option is also available for `std::lock_guard`.
- `std::try_to_lock`: Try to acquire the mutex without blocking.

If we just pass the mutex as the only parameter to the `std::unique_lock` constructor, the behavior is the same as in `std::lock_guard`: it will block until the mutex is available and then acquire it. It will release the mutex in the destructor.

The `std::unique_lock` class, unlike `std::lock_guard`, allows you to call `lock()` and `unlock()` to respectively acquire and release the mutex.

std::scoped_lock

The `std::scoped_lock` class, as with `std::unique_lock`, is a `std::mutex` wrapper implementing an RAII mechanism (remember – the mutexes will be released in the destructor if they are acquired). The main difference is that `std::unique_lock`, as its name implies, just wraps one mutex, but `std::scoped_lock` wraps zero or more mutexes. Also, the mutexes are acquired in the order they are passed to the `std::scoped_lock` constructor, hence avoiding deadlock.

Let's look at the following code:

```
std::mutex mtx1;
std::mutex mtx2;

// Acquire both mutexes avoiding deadlock
std::scoped_lock lock(mtx1, mtx2);

// Same as doing this
// std::lock(mtx1, mtx2);
// std::lock_guard<std::mutex> lock1(mtx1, std::adopt_lock);
// std::lock_guard<std::mutex> lock2(mtx2, std::adopt_lock);
```

The preceding code snippet shows how we can work with two mutex locks very easily.

std::shared_lock

The `std::shared_lock` class is another general-purpose mutex ownership wrapper. As with `std::unique_lock` and `std::scoped_lock`, it allows deferred locking and transferring the lock ownership. The main difference between `std::unique_lock` and `std::shared_lock` is that the latter is used to acquire/release the wrapped mutex in shared mode while the former is used to do the same in exclusive mode.

In this section, we saw mutex wrapper classes and their main features. Next, we will introduce another synchronization mechanism: condition variables.

Condition variables

Condition variables are another synchronization primitive provided by the C++ Standard Library. They allow multiple threads to communicate with each other. They also allow for several threads to wait for a notification from another thread. Condition variables are always associated with a mutex.

In the following example, a thread must wait for a counter to be equal to a certain value:

```
#include <chrono>
#include <condition_variable>
#include <iostream>
#include <mutex>
#include <thread>
#include <vector>

int counter = 0;

int main() {
    using namespace std::chrono_literals;

    std::mutex mtx;
    std::mutex cout_mtx;
    std::condition_variable cv;

    auto increment_counter = [&] {
        for (int i = 0; i < 20; ++i) {
            std::this_thread::sleep_for(100ms);
            mtx.lock();
            ++counter;
            mtx.unlock();
            cv.notify_one();
        }
    };

    auto wait_for_counter_non_zero_mtx = [&] {
        mtx.lock();
        while (counter == 0) {
            mtx.unlock();
            std::this_thread::sleep_for(10ms);
            mtx.lock();
        }
        mtx.unlock();
        std::lock_guard<std::mutex> cout_lck(cout_mtx);
        std::cout << "Counter is non-zero" << std::endl;
```

```
    };

    auto wait_for_counter_10_cv = [&] {
        std::unique_lock<std::mutex> lck(mtx);
        cv.wait(lck, [] { return counter == 10; });

        std::lock_guard<std::mutex> cout_lck(cout_mtx);
        std::cout << "Counter is: " << counter << std::endl;
    };

    std::thread t1(wait_for_counter_non_zero_mtx);
    std::thread t2(wait_for_counter_10_cv);
    std::thread t3(increment_counter);

    t1.join();
    t2.join();
    t3.join();

    return 0;
}
```

There are two ways to wait for a certain condition: one is waiting in a loop and using a mutex as a synchronization mechanism. This is implemented in `wait_for_counter_non_zero_mtx`. The function acquires the lock, reads the value in `counter`, and releases the lock. Then, it sleeps for 10 milliseconds, and the lock is acquired again. This is done in a `while` loop until `counter` is nonzero.

Condition variables help us to simplify the previous code. The `wait_for_counter_10_cv` function waits until `counter` is equal to 10. The thread will wait on the `cv` condition variable until it is notified by `t1`, the thread increasing `counter` in a loop.

`The wait_for_counter_10_cv` function works like this: a condition variable, `cv`, waits on a mutex, `mtx`. After calling `wait()`, the condition variable locks the mutex and waits until the condition is `true` (the condition is implemented in the lambda passed as a second parameter to the `wait` function). If the condition is not `true`, the condition variable remains in a *waiting* state until it is signaled and releases the mutex. Once the condition is met, the condition variable ends its waiting state and locks the mutex again to synchronize its access to `counter`.

One important issue is that the conditional variable may be signaled by an unrelated thread. This is called **spurious wakeup**. To avoid errors due to spurious wakeups, the condition is checked in `wait`. When the condition variable is signaled, the condition is checked again. In the event of a spurious wakeup and the counter being zero (the condition check returns `false`), the waiting would resume.

A different thread increments the counter by running `increment_counter`. Once `counter` has the desired value (in the example, this value is 10), it signals the waiting thread condition variable.

There are two functions provided to signal a condition variable:

- `cv.notify_one()`: Signal only one of the waiting threads
- `cv.notify_all()`: Signal all of the waiting threads

In this section, we have introduced condition variables, and we have seen a simple example of synchronization using condition variables and how in some cases it can simplify the synchronization/waiting code. Now, let us turn our attention to implementing a synchronized queue using a mutex and two condition variables.

Implementing a multithreaded safe queue

In this section, we will see how to implement a simple **multithreaded safe queue**. The queue will be accessed by multiple threads, some of them adding elements to it (**producer threads**) and some of them removing elements from it (**consumer threads**). For starters, we are going to assume just two threads: one producer and one consumer.

Queues or **first-in-first-outs** (**FIFOs**) are a standard way of communication between threads. For example, if we need to receive packets containing data from a network connection as fast as possible, we may not have enough time in just one thread to receive all the packets and process them. In this case, we use a second thread to process the packets read by the first thread. Using just one consumer thread is simpler to synchronize (we will see how this is the case in *Chapter 5*), and we have a guarantee that the packets will be processed in the same order as they arrived and were copied to the queue by the producer thread. It is true that the packets will really be read in the same order they were copied to the queue irrespective of the number of threads we have as consumers, but the consumer threads may be scheduled in and out by the operating system, and the full sequence of processed packets could be in a different order.

In general, the easiest problem is that of a **single-producer-single-consumer** (**SPSC**) queue. Different problems may require multiple consumers if the processing of each item is too costly for just a thread, and we may have different sources of data to be processed and need multiple producer threads. The queue described in this section will work in every case.

The first step in designing the queue is deciding what data structure we will use to store the queued items. We want the queue to contain elements of any type *T*, so we will implement it as a template class. Also, we are going to limit the capacity of the queue so that the maximum number of elements we can store in the queue will be fixed and set in the class constructor. It is possible, for example, to use a linked list and make the queue unbounded, or even use a **Standard Template Library** (**STL**) queue, `std::queue`, and let the queue grow to an arbitrary size. In this chapter, we will implement a fixed-size queue. We will revisit the implementation in *Chapter 5* and implement it in a very different way (we won't be using any mutex or waiting on condition variables). For our current implementation, we will use an STL vector, `std::vector<T>`, to store the queued items. The vector will allocate memory for all the elements in the queue class constructor, so there will be no memory allocations after

that. When the queue is destroyed, the vector will destroy itself and will free the allocated memory. This is convenient and simplifies the implementation.

We will use the vector as a **ring buffer**. This means that, once we store an element at the end of the vector, the next one will be stored at the beginning, so we *wrap around* both locations to write and read elements from the vector.

This is the first version of the queue class, quite simple and not useful yet:

```
template <typename T>
class synchronized_queue {
public:
    explicit synchronized_queue(size_t size) :
        capacity_{ size }, buffer_(capacity_)
        {}

private:
    std::size_t head_{ 0 };
    std::size_t tail_{ 0 };
    std::size_t capacity_;
    std::vector<T> buffer_;
};
```

The `head` and `tail` variables are used to indicate where to read or write the next element respectively. We also need to know when the queue is empty or full. If the queue is empty, the consumer thread won't be able to get any item from the queue. If the queue is full, the producer thread will not be able to put any items in the queue.

There are different ways to indicate when a queue is empty and when it is full. In this example, we follow this convention:

- If `tail_ == head_`, then the queue is empty
- If `(tail_ + 1) % capacity_ == head_`, then the queue is full

Another way to implement it would require just checking if `tail_ == head_` and using an extra flag to indicate if the queue is full or not (or using a counter to know how many items there are in the queue). We avoid any extra flag or counter in this example because the flag will be read and written by both the consumer and the producer threads, and we aim to minimize sharing data among threads as much as we can. Also, reducing sharing data will be the only option when we revisit the implementation of the queue in *Chapter 5*.

There is a small issue here. Because of the way we check if the queue is full, we lose one slot in the buffer, so the real capacity is `capacity_ - 1`. We will consider the queue as full when there is just one empty slot. Because of this, we lose one queue slot (note that the slot will be used, but the queue will still be full when the number of items is `capacity_ - 1`). In general, this is not an issue.

The queue we are going to implement is a bounded queue (fixed size) implemented as a ring buffer.

There is another detail to be considered here: `head_ + 1` must take into account that we wrap around the indices to the buffer (it is a ring buffer). So, we must do `(head_ + 1) % capacity_`. The modulo operator calculates the remainder of the index value divided by the queue capacity.

The following code shows the basic utility functions implemented as helper functions in the synchronized queue:

```
template <typename T>
class synchronized_queue {
public:
    explicit synchronized_queue(size_t size) :
        capacity_{ size }, buffer_(capacity_) {
    }

private:
    std::size_t next(std::size_t index) {
        return (index + 1)% capacity_;
    }
    bool is_full() const {
        return next(tail_) == head_;
    }
    bool is_empty() const {
        return tail_ == head_;
    }

    std::size_t head_{ 0 };
    std::size_t tail_{ 0 };
    std::size_t capacity_;
    std::vector<T> buffer_;
};
```

We have implemented a few useful functions to update both the head and the tail of the ring buffer and to check if the buffer is full or empty. Now, we can start implementing the queue functionality.

The code for the full queue implementation is in the accompanying GitHub repo for the book. *Here, we only show the important bits* for the sake of simplicity and focus just on the synchronization aspects of the queue implementation.

The interface to the queue has the following two functions:

```
void push(const T& item);
void pop(T& item);
```

The `push` function inserts an element in the queue, while `pop` gets an element from the queue.

Let's start with `push`. It inserts an item in the queue. If the queue is full, `push` will wait until the queue has at least an empty slot (a consumer removed an element from the queue). This way, the producer thread will be blocked until the queue has at least one empty slot (the not-full condition is met).

We have seen earlier in this chapter that there is a synchronization mechanism called a condition variable that does just that. The `push` function will check if the condition is met, and when it is met, it will insert an item in the queue. If the condition is not met, the lock associated with the condition variable will be released, and the thread will wait on the condition variable until the condition is satisfied.

It is possible for the condition variable to just wait until the lock is released. We still need to check if the queue is full because a condition variable may end its waiting due to a spurious wakeup. This happens when the condition variable receives a notification not sent explicitly by any other thread.

We add the following three member variables to the queue class:

```
std::mutex mtx_;
std::condition_variable not_full_;
Std::condition_variable not_empty_;
```

We need two condition variables – one to notify the consumers that the queue is not full (`not_full_`) and another to notify the producers that the queue is not empty (`not_empty_`).

This is the code implementing `push`:

```
void push(const T& item) {
    std::unique_lock<std::mutex> lock(mtx_);
    not_full_.wait(lock, [this]{ return !is_full(); });

    buffer_[tail_] = T;
    tail_ = increment(tail_);

    lock.unlock();
    not_empty_.notify_one();
}
```

Let's think about a scenario with a single producer and a single consumer. We will see the `pop` function later, but as an advance, it also synchronizes with the mutex/condition variable. Both threads try to access the queue at the same time – the producer when inserting an element and the consumer when removing it.

Let's assume the consumer acquires the lock first. This happens in `[1]`. The use of `std::unique_lock` is required by condition variables to use a mutex. In `[2]`, we wait on the condition variable until the condition in the predicate of the `wait` function is met. If it is not met, the lock is released for the consumer thread to be able to access the queue.

Once the condition is met, the lock is acquired again, and the queue is updated in `[3]`. After updating the queue, `[4]` releases the lock and then `[5]` notifies one consumer thread that may be waiting on `not_empty` that the queue is effectively not empty now.

The `std::unique_lock` class could release the mutex lock in its destructor, but we needed to release it in `[4]` because we didn't want to release the lock after notifying the condition variable.

The `pop()` function follows a similar logic, as shown in the following code:

```
void pop(T& item)
{
    std::unique_lock<std::mutex> lock(mtx_);
    not_empty_.wait(lock, [this]{return !is_empty()});

    item = buffer_[head_];
    head_ = increment(head_);

    lock.unlock();
    not_full_.notify_one();
}
```

The code is very similar to that in the `push` function. `[1]` creates the `std::unique_lock` class required to use the `not_empty_` condition variable. `[2]` waits on `not_empty_` until it is notified that the queue is not empty. `[3]` reads the item from the queue, assigning it to the `item` variable, and then in `[4]`, the lock is released. Finally, in `[5]`, the `not_full_` condition variable is notified to indicate to the consumer that the queue is not full.

Both `push` and `pop` functions are blocking and waiting until the queue is not full or not empty respectively. We may need the thread to keep on running in the event of not being able to either insert or get a message to/from the queue – for example, to let it do some independent processing – and then try again to access the queue.

The `try_push` function does exactly that. If the mutex is free to be acquired and the queue is not full, then the functionality is the same as the `push` function, but in this case, `try_push` doesn't need to use any condition variable for synchronization (but it must notify the consumer). This is the code for `try_push`:

```
bool try_push(const T& item) {
    std::unique_lock<std::mutex> lock(mtx_, std::try_to_lock);
    if (!lock || is_full()) {
        return false;
    }

    buffer_[tail_] = item;
    tail_ = next(tail_);
```

```
    lock.unlock();

    not_empty_.notify_one();

    return true;
}
```

The code works like this: `[1]` tries to acquire the lock and returns without blocking the calling thread. If the lock was already acquired, then it will evaluate to `false`. In `[2]`, in case the lock has not been acquired or the queue is full, `try_push` returns `false` to indicate to the caller that no item was inserted in the queue and delegates the waiting/blocking to the caller. Note that `[3]` returns `false` and the function terminates. If the lock was acquired, it will be released when the function exits and the `std::unique_lock` destructor is called.

After the lock is acquired and has checked that the queue is not full, then the item is inserted in the queue, and `tail_` is updated. In `[5]`, the lock is released, and in `[6]`, the consumer is notified that the queue is not empty anymore. This notification is required because the consumer may call `pop` instead of `try_pop`.

Finally, the function returns `true` to indicate to the caller that the item was successfully inserted in the queue.

The code for the corresponding `try_pop` function is shown next. As an exercise, try to understand how it works:

```
bool try_pop(T& item) {
     std::unique_lock<std::mutex> lock(mtx_, std::try_to_lock);
     if (!lock || is_empty()) {
         return false;
     }

     item = buffer_[head_];
     head_ = next(head_);

     lock.unlock();

     not_empty_.notify_one();

     return true;
 }
```

This is the full code for the queue we have implemented in this section:

```
#pragma once

#include <condition_variable>
#include <mutex>
#include <vector>

namespace async_prog {

template <typename T>
class queue {
public:
    queue(std::size_t capacity) : capacity_{capacity}, buffer_
(capacity) {}

    void push(const T& item) {
        std::unique_lock<std::mutex> lock(mtx_);
        not_full_.wait(lock, [this] { return !is_full(); });

        buffer_[tail_] = item;
        tail_ = next(tail_);

        lock.unlock();

        not_empty_.notify_one();
    }

    bool try_push(const T& item) {
        std::unique_lock<std::mutex> lock(mtx_, std::try_to_lock);
        if (!lock || is_full()) {
            return false;
        }

        buffer_[tail_] = item;
        tail_ = next(tail_);

        lock.unlock();

        not_empty_.notify_one();

        return true;
```

```
    }

    void pop(T& item) {
        std::unique_lock<std::mutex> lock(mtx_);
        not_empty_.wait(lock, [this] { return !is_empty(); });

        item = buffer_[head_];
        head_ = next(head_);

        lock.unlock();

        not_full_.notify_one();
    }

    bool try_pop(T& item) {
        std::unique_lock<std::mutex> lock(mtx_, std::try_to_lock);
        if (!lock || is_empty()) {
            return false;
        }

        item = buffer_[head_];
        head_ = next(head_);

        lock.unlock();

        not_empty_.notify_one();

        return true;
    }

private:
    [[nodiscard]] std::size_t next(std::size_t idx) const noexcept {
        return ((idx + 1) % capacity_);
    }

    [[nodiscard]] bool is_empty() const noexcept { return (head_ ==
tail_); }

    [[nodiscard]] bool is_full() const noexcept { return (next(tail_)
== head_); }

   private:
    std::mutex mtx_;
    std::condition_variable not_empty_;
```

```
    std::condition_variable not_full_;

    std::size_t head_{0};
    std::size_t tail_{0};
    std::size_t capacity_;
    std::vector<T> buffer_;
};
}
```

In this section, we have introduced condition variables and implemented a basic queue synchronized with a mutex and two condition variables, the two basic synchronization primitives provided by the C++ Standard Library since C++11.

The queue example shows how synchronization is implemented using these synchronization primitives and can be used as a basic building block for more elaborate utilities such as, for example, a thread pool.

Semaphores

C++20 introduces new synchronization primitives to write multithreaded applications. In this section, we will look at semaphores.

A **semaphore** is a counter that manages the number of permits available for accessing a shared resource. Semaphores can be classified into two main types:

- A **binary semaphore** is like a mutex. It has only two states: 0 and 1. Even though a binary semaphore is conceptually like a mutex, there are some differences between a binary semaphore and a mutex that we will see later in this section.
- A **counting semaphore** can have a value greater than 1 and is used to control access to a resource that has a limited number of instances.

C++20 implements both binary and counting semaphores.

Binary semaphores

A binary semaphore is a synchronization primitive that can be used to control access to a shared resource. It has two states: 0 and 1. A semaphore with a value of 0 indicates that the resource is unavailable, while a semaphore with a value of 1 indicates that the resource is available.

Binary semaphores can be used to implement mutual exclusion. This is achieved by using a binary semaphore to control access to the resource. When a thread wants to access the resource, it first checks the semaphore. If the semaphore is 1, the thread can access the resource. If the semaphore is 0, the thread must wait until the semaphore is 1 before it can access the resource.

The most significant difference between mutexes and semaphores is that mutexes have exclusive ownership, whereas binary semaphores do not. Only the thread owning the mutex can release it. Semaphores can be signaled by any thread. A mutex is a locking mechanism for a critical section, and a semaphore is more like a signaling mechanism. In this respect, a semaphore is closer to a condition variable than a mutex. For this reason, semaphores are commonly used for signaling rather than for mutual exclusion.

In C++20, `std::binary_semaphore` is an alias for the specialization of `std::counting_semaphore`, with `LeastMaxValue` being 1.

Binary semaphores must be initialized with either 1 or 0, such as follows:

```
std::binary_semaphore sm1{ 0 };
std::binary_semaphore sm2{ 1 };
```

If the initial value is `0`, acquiring the semaphore will block the thread trying to acquire it, and before it can be acquired, it must be released by another thread. Acquiring a semaphore decreases the counter, and releasing it increases the counter. As previously stated, if the counter is `0` and a thread tries to acquire the lock (semaphore), the thread will be blocked until the semaphore counter is greater than `0`.

Counting semaphores

A counting semaphore allows access to a shared resource by more than one thread. The counter can be initialized to an arbitrary number, and it will be decreased every time a thread acquires the semaphore. As an example of how to use counting semaphores, we will modify the multithread safe queue we implemented in the previous section and use semaphores instead of condition variables to synchronize access to the queue.

The member variables of the new class are the following:

```
template <typename T>
class queue {
 // public methods and private helper methods

private:
    std::counting_semaphore<> sem_empty_;
    std::counting_semaphore<> sem_full_;

    std::size_t head_{ 0 };
    std::size_t tail_{ 0 };
    std::size_t capacity_;
    std::vector<T> buffer_;
};
```

We still need `head_` and `tail_` to know where to read and write an element, `capacity_` for the wraparound of the indices, and `buffer_`, a `std::vector<T>` vector. But for now, we are not using a mutex, and we will use counting semaphores instead of condition variables. We will use two of them: `sem_empty_` to count the empty slots in the buffer (initially set to `capacity_`) and `sem_full_` to count the non-empty slots in the buffer, initially set to 0.

Now, let's see how to implement `push`, the function used to insert items in a queue.

In `[1]`, `sem_empty_` is acquired, decreasing the semaphore counter. If the queue is full, then the thread will block until `sem_empty_` is released (signaled) by another thread. If the queue is not full, then the item is copied to the buffer, and `tail_` is updated in `[2]` and `[3]`. Finally, `sem_full_` is released in `[4]`, signaling another thread that the queue is not empty and there is at least one item in the buffer:

```
void push(const T& item) {
    sem_empty_.acquire();

    buffer_[tail_] = item;
    tail_ = next(tail_);

    sem_full_.release();
}
```

The `pop` function is used to get elements from a queue:

```
void pop(T& item) {
    sem_full_.acquire();
    item = buffer_[head_];
    head_ = next(head_);

    sem_empty_.release();
}
```

Here, in `[1]`, we successfully acquire `sem_full_` if the queue is not empty. Then, the item is read and `head_` updated in `[2]` and `[3]` respectively. Finally, we signal the consumer thread that the queue is not full, releasing `sem_empty`.

There are several issues in our first version of `push`. The first and most important one is that `sem_empty_` allows more than one thread to access the critical section in the queue (`[2]` and `[3]`). We need to synchronize this critical section and use a mutex.

Here is the new version of `push` using a mutex for synchronization.

In [2], the lock is acquired (using std::unique_lock), and in [5], it is released. Using the lock will synchronize the critical section, preventing several threads from simultaneously accessing it and updating the queue concurrently without any synchronization:

```
void push(const T& item)
{
    sem_empty_.acquire();

    std::unique_lock<std::mutex> lock(mtx_);
    buffer_[tail_] = item;
    tail_ = next(tail_);
    lock.unlock();

    sem_full_.release();
}
```

A second issue is that acquiring a semaphore is blocking, and as we have seen previously, sometimes the caller thread can do some processing instead of just waiting. The try_push function (and its corresponding try_pop function) implements this functionality. Let's study the code of try_push. Note that try_push may still block on the mutex:

```
bool try_push(const T& item) {
    if (!sem_empty_.try acquire()) {
        return false;
    }

    std::unique_lock<std::mutex> lock(mtx_);

    buffer_[tail_] = item;
    tail_ = next(tail_);

    lock.unlock();

    sem_full_.release();

    return true;
}
```

The only changes are [1] and [2]. Instead of blocking when acquiring the semaphore, we just try to acquire it, and if we fail, we return false. The try_acquire function may spuriously fail and return false even if the semaphore can be acquired (count is not zero).

Here is the complete code for the queue synchronized with semaphores:

```
#pragma once

#include <mutex>
#include <semaphore>
#include <vector>

namespace async_prog {

template <typename T>
class semaphore_queue {
   public:
    semaphore_queue(std::size_t capacity)
        : sem_empty_(capacity), sem_full_(0), capacity_{capacity},
buffer_(capacity)
    {}

    void push(const T& item) {
        sem_empty_.acquire();

        std::unique_lock<std::mutex> lock(mtx_);

        buffer_[tail_] = item;
        tail_ = next(tail_);

        lock.unlock();

        sem_full_.release();
    }

    bool try_push(const T& item) {
        if (!sem_empty_.try_acquire()) {
            return false;
        }

        std::unique_lock<std::mutex> lock(mtx_);

        buffer_[tail_] = item;
        tail_ = next(tail_);

        lock.unlock();
        sem_full_.release();
```

```
        return true;
    }

    void pop(T& item) {
        sem_full_.acquire();

        std::unique_lock<std::mutex> lock(mtx_);

        item = buffer_[head_];
        head_ = next(head_);

        lock.unlock();
        sem_empty_.release();
    }

    bool try_pop(T& item) {
        if (!sem_full_.try_acquire()) {
            return false;
        }

        std::unique_lock<std::mutex> lock(mtx_);

        item = buffer_[head_];
        head_ = next(head_);

        lock.unlock();
        sem_empty_.release();

        return true;
    }

private:
    [[nodiscard]] std::size_t next(std::size_t idx) const noexcept {
        return ((idx + 1) % capacity_);
    }

private:
    std::mutex mtx_;
    std::counting_semaphore<> sem_empty_;
    std::counting_semaphore<> sem_full_;

    std::size_t head_{0};
    std::size_t tail_{0};
```

```
    std::size_t capacity_;
    std::vector<T> buffer_;
};
```

In this section, we have seen semaphores, a new synchronization primitive included in the C++ Standard Library since C++20. We learned how to use them to implement the same queue we implemented before but using semaphores as synchronization primitives.

In the next section, we will introduce **barriers** and **latches**, two new synchronization mechanisms included in the C++ Standard Library since C++20.

Barriers and latches

In this section, we will introduce barriers and latches, two new synchronization primitives introduced in C++20. These mechanisms allow threads to wait for each other, thereby coordinating the execution of concurrent tasks.

std::latch

The `std::latch` latch is a synchronization primitive that allows one or more threads to block until a specified number of operations are completed. It is a single-use object, and once the count reaches zero, it cannot be reset.

The following example is a simple illustration of the use of latches in a multithreaded application. We want to write a function to multiply by two each element of a vector and then add all the elements of the vector. We will use three threads to multiply the vector elements by two and then one thread to add all the elements of the vector and obtain the result.

We need two latches. The first one will be decremented by each of the three threads multiplying by two vector elements. The adding thread will wait for this latch to be zero. Then, the main thread will wait on the second latch to synchronize printing the result of adding all the vector's elements. We can also wait for the thread performing the additions calling `join` on it, but this can be done with a latch too.

Now, let's analyze the code in functional blocks. We will include the full code for the latches and barriers example later in this section:

```
std::latch map_latch{ 3 };
auto map_thread = [&](std::vector<int>& numbers, int start, int end) {
    for (int i = start; i < end; ++i) {
        numbers[i] *= 2;
    }

    map_latch.count_down();
};
```

Each multiplying thread will run this lambda function, multiplying by two elements of a certain range in the vector (from `start` to `end`). Once the thread is done, it will decrease the `map_latch` counter by one. Once all the threads finish their tasks, the latch counter will be zero, and the thread blocked waiting on `map_latch` will be able to go on and add all the elements of the vector together. Note that the threads access different elements of the vector, so we don't need to synchronize access to the vector itself, but we cannot start adding the numbers until all the multiplications are done.

The code for the adding thread is the following:

```
std::latch reduce_latch{ 1 };
auto reduce_thread = [&](const std::vector<int>& numbers, int& sum) {
    map_latch.wait();

    sum = std::accumulate(numbers.begin(), numbers.end(), 0);

    reduce_latch.count_down();
};
```

This thread waits until the `map_latch` counter goes down to zero, then adds all the elements of the vector, and finally decrements the `reduce_latch` counter (it will go down to zero) for the main thread to be able to print the final result:

```
reduce_latch.wait();
std::cout << "All threads finished. The sum is: " << sum << '\n';
```

Having seen a basic application of latches, next, let's learn about barriers.

std::barrier

The `std::barrier` barrier is another synchronization primitive used to synchronize a group of threads. The `std::barrier` barrier is reusable. Each thread reaches the barrier and waits until all participating threads reach the same barrier point (like what happens when we use latches).

The main difference between `std::barrier` and `std::latch` is the reset capability. The `std::latch` latch is a single-use barrier with a countdown mechanism that cannot be reset. Once it reaches zero, it stays at zero. In contrast, `std::barrier` is reusable. It resets after all threads have reached the barrier, allowing the same set of threads to synchronize at the same barrier multiple times.

When to use latches and when to use barriers? Use `std::latch` when you have a one-time gathering point for threads, such as waiting for multiple initializations to complete before proceeding. Use `std::barrier` when you need to synchronize threads repeatedly through multiple phases of a task or iterative computations.

We will now rewrite the previous example, this time using barriers instead of latches. Each thread will multiply by two its corresponding range of vector elements, and then it will add them. The main thread will use `join()` in this example to wait for the processing to be finished and then add the results obtained by each of the threads.

The code for the worker thread is the following:

```
std::barrier map_barrier{ 3 };
auto worker_thread = [&](std::vector<int>& numbers, int start, int
end, int id) {
    std::cout << std::format("Thread {0} is starting...\n", id);

    for (int i = start; i < end; ++i) {
        numbers[i] *= 2;
    }

    map_barrier.arrive_and_wait();

    for (int i = start; i < end; ++i) {
        sum[id] += numbers[i];
    }

    map_barrier.arrive();
};
```

The code is synchronized with a barrier. When a worker thread finishes doing the multiplications, it decreases the `map_barrier` counter and waits for the barrier counter to be zero. Once it goes down to zero, the threads end their waiting and start doing the additions. The barrier counter is reset, and its value is again equal to three. Once the additions are done, the barrier counter is decremented again, but this time, the threads won't wait because their task is done.

Sure – each thread could have done the additions and then multiplied by two. They don't need to wait for each other because the work done by any thread is independent of the work done by any other thread, but this is a good way of explaining how barriers work with an easy example.

The main thread just waits with `join` for the worker threads to finish and then prints the result:

```
for (auto& t : workers) {
    t.join();
}
std::cout << std::format("The total sum is {0}\n",
                         std::accumulate(sum.begin(), sum. End(), 0));
```

Here is the full code for the latches and barriers example:

```
#include <algorithm>
#include <barrier>
#include <format>
#include <iostream>
#include <latch>
#include <numeric>
#include <thread>
#include <vector>

void multiply_add_latch() {
    const int NUM_THREADS{3};

    std::latch map_latch{NUM_THREADS};
    std::latch reduce_latch{1};

    std::vector<int> numbers(3000);
    int sum{};
    std::iota(numbers.begin(), numbers.end(), 0);

    auto map_thread = [&](std::vector<int>& numbers, int start, int
end) {
        for (int i = start; i < end; ++i) {
            numbers[i] *= 2;
        }

        map_latch.count_down();
    };

    auto reduce_thread = [&](const std::vector<int>& numbers, int&
sum) {
        map_latch.wait();

        sum = std::accumulate(numbers.begin(), numbers.end(), 0);

        reduce_latch.count_down();
    };

    for (int i = 0; i < NUM_THREADS; ++i) {
        std::jthread t(map_thread, std::ref(numbers), 1000 * i, 1000 *
(i + 1));
    }
```

```
    std::jthread t(reduce_thread, numbers, std::ref(sum));

    reduce_latch.wait();

    std::cout << "All threads finished. The total sum is: " << sum <<
'\n';
}

void multiply_add_barrier() {
    const int NUM_THREADS{3};

    std::vector<int> sum(3, 0);
    std::vector<int> numbers(3000);
    std::iota(numbers.begin(), numbers.end(), 0);

    std::barrier map_barrier{NUM_THREADS};

    auto worker_thread = [&](std::vector<int>& numbers, int start, int
end, int id) {
        std::cout << std::format("Thread {0} is starting...\n", id);

        for (int i = start; i < end; ++i) {
            numbers[i] *= 2;
        }

        map_barrier.arrive_and_wait();

        for (int i = start; i < end; ++i) {
            sum[id] += numbers[i];
        }

        map_barrier.arrive();
    };

    std::vector<std::jthread> workers;
    for (int i = 0; i < NUM_THREADS; ++i) {
        workers.emplace_back(worker_thread, std::ref(numbers), 1000 *
i,
                             1000 * (i + 1), i);
    }

    for (auto& t : workers) {
        t.join();
    }
```

```
    std::cout << std::format("All threads finished. The total sum is:
{0}\n",
     std::accumulate(sum.begin(), sum.end(), 0));
}

int main() {
    std::cout << "Multiplying and reducing vector using barriers..."
<< std::endl;
    multiply_add_barrier();

    std::cout << "Multiplying and reducing vector using latches..." <<
std::endl;
    multiply_add_latch();
    return 0;
}
```

In this section, we have seen barriers and latches. Though they are not so commonly used as mutexes, condition variables, and semaphores, it is always useful to know what they are. The simple examples presented here have illustrated a common use of barriers and latches: synchronizing threads performing processing in different stages.

Finally, we will see a mechanism to execute code just once, even if the code is called more than once from different threads.

Performing a task only once

Sometimes, we need to perform a certain task just one time. For example, in a multithreaded application, several threads may run the same function to initialize a variable. Any of the running threads may do it, but we want the initialization to be done exactly once.

The C++ Standard Library provides both `std::once_flag` and `std::call_once` to implement exactly that functionality. We will see how to implement this functionality using atomic operations in the next chapter.

The following example will help us to understand how to use `std::once_flag` and `std::call_once` to achieve our goal of performing a task just one time when more than one thread tries to do it:

```
#include <exception>
#include <iostream>
#include <mutex>
#include <thread>

int main() {
    std::once_flag run_once_flag;
```

```
    std::once_flag run_once_exceptions_flag;

    auto thread_function = [&] {
        std::call_once(run_once_flag, []{
            std::cout << "This must run just once\n";
        });
    };

    std::jthread t1(thread_function);
    std::jthread t2(thread_function);
    std::jthread t3(thread_function);

    auto function_throws = [&](bool throw_exception) {
        if (throw_exception) {
            std::cout << "Throwing exception\n";
            throw std::runtime_error("runtime error");
        }

        std::cout << "No exception was thrown\n";
    };

    auto thread_function_1 = [&](bool throw_exception) {
        try {
            std::call_once(run_once_exceptions_flag,
                           function_throws,
                           throw_exception);
        }
        catch (...) {
        }
    };

    std::jthread t4(thread_function_1, true);
    std::jthread t5(thread_function_1, true);
    std::jthread t6(thread_function_1, false);

    return 0;
}
```

In the first part of the example, three threads, `t1`, `t2`, and `t3`, run the `thread_function` function. This function calls a lambda from `std::call_once`. If you run the example, you will see that the message `This must run just once` is printed only one time, as expected.

In the second part of the example, again, three threads, `t4`, `t5`, and `t6`, run the `thread_function_1` function. This function calls `function_throws`, which depending on a parameter may throw or not throw an exception. This code shows that, if the function called from `std::call_once` does not terminate successfully, then it doesn't count as done and `std::call_once` should be called again. Only a successful function counts as a run function.

This final section showed a simple mechanism we can use to ensure that a function is executed exactly once, even if it is called more than once from the same or a different thread.

Summary

In this chapter, we learned how to use the lock-based synchronization primitives provided by the C++ Standard Library.

We started with an explanation of race conditions and the need for mutual exclusion. Then, we studied `std::mutex` and how to use it to solve race conditions. We also learned about the main problems when synchronizing with locks: deadlock and livelock.

After learning about mutexes, we studied condition variables and implemented a synchronized queue using mutex and condition variables. Finally, we saw the new synchronization primitives introduced in C++20: semaphores, latches, and barriers.

Finally, we studied the mechanisms provided by the C++ Standard Library to run a function just one time.

In this chapter, we learned about the basic building blocks of thread synchronization and the foundation of asynchronous programming with multiple threads. Lock-based thread synchronization is the most used method to synchronize threads.

In the next chapter, we will study lock-free thread synchronization. We will start with a review of atomicity, atomic operations, and atomic types provided by the C++20 Standard Library. We will show an implementation of a lock-free bound single-producer-single-consumer queue. We will also introduce the C++ memory model.

Further reading

- David R. Butenhof, *Programming with POSIX Threads*, Addison Wesley, 1997.
- Anthony Williams, *C++ Concurrency in Action*, Second Edition, Manning, 2019.

5
Atomic Operations

In *Chapter 4*, we learned about lock-based thread synchronization. We learned about mutexes, condition variables, and other thread synchronization primitives, which are all based on acquiring and releasing locks. Those synchronization mechanisms are built on top of *atomic types and operations*, this chapter's topic.

We will study what atomic operations are and how they differ from lock-based synchronization primitives. After reading this chapter, you will have a basic knowledge of atomic operations and some of their applications. Lock-free (not using locks) synchronization based on atomic operations is a very complex subject requiring years to master, but we will give you what we hope will be a good introduction to the subject.

In this chapter, we will cover the following main topics:

- What are atomic operations?
- An introduction to the C++ memory model
- What atomic types and operations are provided by the C++ Standard Library?
- Some examples of atomic operations, from a simple counter to be used to gather statistics and a basic mutex-like lock to a full **single-producer-single-consumer** (**SPSC**) lock-free bounded queue

Technical requirements

You will need a recent C++ compiler with C++20 support. Some short code examples will be provided as links to the very useful godbolt website (`https://godbolt.org`). For full code examples, we will use the book repo, which is available at `https://github.com/PacktPublishing/Asynchronous-Programming-with-CPP`.

The examples can be compiled and run locally. We have tested the code on an Intel CPU computer running Linux (Ubuntu 24.04 LTS). For atomic operations and especially for memory ordering (more on this later in this chapter), Intel CPUs are different from Arm CPUs.

Please note here that code performance and profiling will be the subject of *Chapter 13*. We will just make some remarks on performance in this chapter to avoid making it unnecessarily long.

Introduction to atomic operations

Atomic operations are indivisible (hence the word atomic, from the Greek *ἄτομος*, *atomos*, indivisible).

In this section, we will introduce atomic operations, what they are, and some reasons to use (and not to use!) them.

Atomic operations versus non-atomic operations – an example

If you remember the simple counter example from *Chapter 4*, we needed to use a synchronization mechanism (we used a mutex) for modifying the counter variable from different threads to avoid race conditions. The cause of the race condition was that incrementing the counter required three operations: reading the counter value, incrementing it, and writing the modified counter value back to memory. If only we could do that in one go, there would be no race condition.

This is exactly what could be achieved with an atomic operation: if we had some kind of `atomic_increment` operation, each thread would read, increment, and write the counter in a single instruction, avoiding the race condition because at any time, incrementing the counter would be fully done. By fully done we mean that each thread would either increment the counter or do nothing at all, making interruptions in the middle of a counter increment operation impossible.

The following two examples are for illustration purposes only and are not multithreaded. We focus here on just the operations, whether atomic or non-atomic.

Let's see this in the code. For the C++ code and the generated assembly language shown in the following example, refer to `https://godbolt.org/z/f4dTacsKW`:

```
int counter {0};
int main() {
    counter++;
    return 0;
}
```

The code increments a global counter. Now let's see the assembly code generated by the compiler and what instructions the CPU executes (the full assembly can be found in the previous link):

```
    Mov    eax, DWORD PTR counter[rip]
    Add    eax, 1
    Move    DWORD PTR counter[rip], eax
```

`[1]` copies the value stored in `counter` to the `eax` register, `[2]` adds `1` to the value stored in `eax`, and finally, `[3]` copies back the content of the `eax` register to the `counter` variable. So, a thread could execute `[1]` and then be scheduled out, and another thread execute all three instructions after that. When the first thread finishes incrementing the result, the counter will be incremented just once and thus the result will be incorrect.

The following code does the same: it increments a global counter. This time, though, it uses atomic types and operations. To get the code and the generated assembly in the following example, refer to `https://godbolt.org/z/9hrbo31vx`:

```
#include <atomic>
std::atomic<int> counter {0};
int main() {
    counter++;
    return 0;
}
```

We will explain the `std::atomic<int>` type and the atomic increment operation later.

The generated assembly code is the following:

```
    lock add    DWORD PTR counter[rip], 1
```

Just one instruction has been generated to add `1` to the value stored in the `counter` variable. The `lock` prefix here means that the following instruction (in this case `add`) is going to be executed atomically. Hence, in this second example, a thread cannot be interrupted in the middle of incrementing the counter. As a side note, some Intel x64 instructions execute atomically and don't use the `lock` prefix.

Atomic operations allow threads to read, modify (for example, increase a value), and write indivisibly, and can also be used as synchronization primitives (similar to the mutexes we saw in *Chapter 4*). In fact, all the lock-based synchronization primitives we have seen so far in this book are implemented using atomic operations. Atomic operations must be provided by the CPU (as in the `lock add` instruction).

In this section, we have introduced atomic operations, defined what they are, and studied a very simple example of how they are implemented by looking at the assembly instructions that the compiler generates. In the next section, we will look at some of the advantages and disadvantages of atomic operations.

When to use (and when not to use) atomic operations

Using atomic operations is a complex subject and it can be very difficult (or at least quite tricky) to master. It requires a lot of experience, and we have attended some courses on this very subject where we were advised not to do it! Anyway, you can always learn the basics and experiment as you do so. We hope this book will help you progress in your learning journey.

Atomic operations can be used in the following cases:

- **If multiple threads share a mutable state**: The need to synchronize threads is the most common case. Of course, it is possible to use locks such as mutexes, but atomic operations, in some cases, will provide better performance. Please note, however, that the use of atomic operations *does not* guarantee better performance.
- **If synchronized access to shared state is fine-grained**: If the data we must synchronize is an integer or a pointer or any other variable of a C++ intrinsic type, then using atomic operations may be better than using locks.
- **To improve performance**: If you want to achieve maximum performance, then atomic operations can help reduce thread context switches (see *Chapter 2*) and reduce the overhead introduced by locks, thus lowering latency. Remember to always profile your code to be sure that performance is improved (we will see this in depth in *Chapter 13*).

Locks can be used in the following cases:

- **If the protected data is not fine-grained**: For example, we are synchronizing access to a data structure or an object bigger than 8 bytes (in modern CPUs).
- **If performance is not an issue**: Locks are much simpler to use and reason about (in some cases using locks gives better performance than using atomic operations).
- **To avoid the need to acquire low-level knowledge**: To get the maximum performance out of atomic operations, a lot of low-level knowledge is required. We will introduce some of it in the section, *The C++ memory model*.

We have just learned when to use and when not to use atomic operations. Some applications such as low-latency/high-frequency trading systems require maximum performance and use atomic operations to achieve the lowest latency possible. Most applications will work just fine synchronizing with locks.

In the next section, we will study the differences between blocking and non-blocking data structures and some related concept definitions.

Non-blocking data structures

In *Chapter 4* we studied the implementation of a synchronized queue. We used mutexes and condition variables as synchronization primitives. Data structures synchronized with locks are called **blocking data structures** because threads are *blocked* (by the operating system), waiting until the locks become available.

Data structures that don't use locks are called **non-blocking data structures**. Most (but not all) of them are lock-free.

A data structure or algorithm is considered lock-free if each synchronized action completes in a finite number of steps, not allowing indefinite waiting for a condition to become true or false.

The types of lock-free data structures are the following:

- **Obstruction-free**: A thread will complete its operation in a bounded number of steps if all other threads are suspended
- **Lock-free**: A thread will complete its operation in a bounded number of steps while multiple threads are working on the data structure
- **Wait-free**: All the threads will complete their operations in a bounded number of steps while multiple threads are working on the data structure

Implementing lock-free data structures is very complicated and before doing it, we need to be sure it's necessary. The reasons to use lock-free data structures are the following:

- **Achieving maximum concurrency**: As we saw earlier, atomic operations are a good choice when the data access synchronization involves fine-grained data (such as native-type variables). From the preceding definitions, a lock-free data structure will allow at least one of the threads accessing the data structure to make some progress in a bounded number of steps. A wait-free structure will allow all the threads accessing the data structure to make some progress.

  When we use locks, however, a thread owns the lock while the rest of the threads are just waiting for the lock to be available, so the concurrency achievable with lock-free data structures can be much better.
- **No deadlocks**: Because there are no locks involved, it is impossible to have any deadlocks in our code.
- **Performance**: Some applications must achieve the lowest latency possible and so waiting for a lock can be unacceptable. When a thread tries to acquire the lock, and it is not available, then the operating system blocks the thread. While the thread is blocked, there is a context switch for the scheduler to be able to schedule another thread for execution. These context switches take time, and that time may be too much in a low-latency application such as a high-performance network packet receiver/processor.

We have now looked at what blocking and non-blocking data structures are and what lock-free code is. We will introduce the C++ memory model in the next section.

The C++ memory model

This section explains the C++ memory model and how it deals with concurrency. The C++ memory model comes with C++11, and defines the two main features of memory in C++:

- How objects are laid out in memory (that is, structural aspects). This subject won't be covered in this book, which is about asynchronous programming.
- Memory modification order (that is, concurrency aspects). We will see the different memory modification orders specified in the memory model.

Memory access order

Before we explain the C++ memory model and the different memory orderings it supports, let's clarify what we mean by memory order. Memory order refers to the order in which memory (that is, the variables in a program) is accessed. Memory access can be either read or write (load and store). But what is the actual order in which the variables of a program are accessed? For the following code, there are three points of view: the written code order, the compiler-generated instructions order, and finally, the order in which the instructions are executed by the CPU. These three orderings can all be the same or (more likely) different.

The first and obvious ordering is the one in the code. An example of this is in the following code snippet:

```
void func_a(int& a, int& b) {
    a += 1;
    b += 10;
    a += 2;
}
```

The `func_a` function first adds 1 to variable `a`, then adds 10 to variable `b`, and finally, adds 2 to variable `a`. This is our intention and the order in which we define the statements to be executed.

The compiler will transform the preceding code into assembly instructions. The compiler can change the order of our statements to make the generated code more efficient if the outcome of the code execution is unchanged. For example, with the preceding code, the compiler could either do the two additions with variable `a` first and then the addition with variable `b`, or it could simply add 3 to `a` and then 10 to `b`. As we mentioned previously, the compiler can do whatever it wants to optimize the code if the result is the same.

Let's now consider the following code:

```
void func_a(int& a, int& b) {
    a += 1;
    b += 10 + a;
    a += 2;
}
```

In this case, the operation on `b` depends on the previous operation on `a`, so the compiler cannot reorder the statements, and the generated code will be like the code we write (same order of operations).

The CPU (which used in this book is a modern Intel x64 CPU) will run the generated code. It can execute the compiler-generated instructions in a different order. This is called out-of-order execution. The CPU can do this, again, if the result is correct.

See this link for the generated code shown in the preceding example: `https://godbolt.org/z/Mhrcnsr9e`

First, the generated instructions for `func_1` show an optimization: the compiler combined both additions into one by adding 3 to variable `a` in one instruction. Second, the generated instructions for `func_2` are in the same order as the C++ statements we wrote. In this case, the CPU could execute the instructions out of order, as there is no dependency among the operations.

To conclude, we can say that the code that the CPU will run can be different from the code we wrote (again, given that the result of the execution is the same as we intended in the program we wrote).

All the examples we have shown are fine for code that runs in a single thread. The code instructions may be executed in different order depending on the compiler optimizations and the CPU out-of-order execution and the result will still be correct.

See the following code for an example of out-of-order execution:

```
mov    eax, [var1]  ; load variable var1 into reg eax
inc    eax          ; eax += 1
mov    [var1], eax  ; store reg eax into var1
xor    ecx, ecx     ; ecx = 0
inc    ecx          ; ecx += 1
add    eax, ecx     ; eax = eax + ecx
```

The CPU could execute the instructions in the order shown in the preceding code, that is, `load var1 [1]`. Then, while the variable is being read, it could issue some of the later instructions, such as `[4]` and `[5]`, and then, once `var1` has been read, execute `[2]`, then `[3]`, and, finally, `[6]`. The instructions were executed in a different order, but the result is still the same. This is a typical example of out-of-order execution: the CPU issues a load instruction and instead of waiting for the data to become available, it executes some other instructions, if possible, to avoid being idle and to maximize performance.

All the optimizations we have mentioned (both compiler and CPU) are always done without considering the interactions between threads. Neither the compiler nor the CPU knows about different threads. In these cases, we need to tell the compiler what it can and cannot do. Atomic operations and locks are the way to do this.

When, for example, we use atomic variables, we may not only require the operations to be atomic but also to follow a certain order for the code to work properly when running multiple threads. This cannot be done by just the compiler or the CPU because neither has any information involving multiple threads. To specify what order we want to use, the C++ memory model offers different options:

- **Relaxed ordering**: `std::memory_order_relaxed`
- **Acquire and release ordering**: `std::memory_order_acquire`, `std::memory_order_release`, `std::memory_order_acq_rel`, and `std::memory_order_consume`
- **Sequential consistency ordering**: `std::memory_order_seq_cst`

The C++ memory model defines an abstract machine to achieve independence from any specific CPU. However, the CPU is still there and the features available in the memory model may not be available to a specific CPU. For example, the Intel x64 architecture is quite restrictive and enforces quite a strong memory order.

The Intel x64 architecture uses a processor-ordered memory-ordering model that can be defined as being *write-ordered with store-buffer forwarding*. In a single-processor system, the memory-ordering model respects the following principles:

- Reads are not reordered with any reads
- Writes are not reordered with any writes
- Writes are not reordered with older reads
- Reads may be reordered with older writes (if the read and write to be reordered refer to different memory locations)
- Reads and writes are not reordered with locked (atomic) instructions

There are more details in the Intel manuals (see the references at the end of the chapter), but the preceding principles are the most relevant.

In a multi-processor system, the following principles apply:

- Each of the individual processors uses the same ordering principles as in a single-processor system
- Writes by a single processor are observed in the same order by all processors
- Writes from an individual processor are not ordered with respect to the writes from other processors
- Memory ordering obeys causality
- Any two stores are seen in a consistent order by processors other than those performing the store
- Locked (atomic) instructions have total order

The Intel architecture is strongly ordered; the store operations (write instructions) for each processor are observed by other processors in the same order they were performed, and each processor executes the stores in the same order as they appear in the program. This is called **Total Store Ordering** (**TSO**).

The ARM architecture supports **Weak Ordering** (**WO**). These are the main principles:

- Reads and writes can be performed out of order. In contrast to TSO where, as we have seen, there is no local reordering except of reads after writes to different addresses, the ARM architecture allows local reordering (unless otherwise specified using special instructions).
- A write is not guaranteed to be visible to all threads at the same time as it was in the Intel architecture.

- In general, this relatively non-restrictive memory ordering allows the cores to reorder instructions more freely, potentially increasing multicore performance.

We must say here that the more relaxed the memory order is, the more difficult it is to reason about the executed code, and the more challenging it becomes to correctly synchronize multiple threads with atomic operations. Also, you should bear in mind that the atomicity is always guaranteed irrespective of the memory order.

In this section, we have seen what is meant by order when accessing memory and how the ordering we specify in the code may not be the same order in which the CPU executes the code. In the next section, we will see how to enforce some ordering using atomic types and operations.

Enforcing ordering

We have seen already in *Chapter 4* and earlier in this chapter that non-atomic operations on the same memory addresses executed from different threads may cause data races and undefined behavior. To enforce the ordering of the operations between threads, we will use atomic types and their operations. This section will explore what the use of atomics achieves in multithreaded code.

The following simple example will help us to see what can be done with atomic operations:

```
#include <atomic>
#include <chrono>
#include <iostream>
#include <string>
#include <thread>

std::string message;
std::atomic<bool> ready{false};

void reader() {
    using namespace std::chrono::literals;

    while (!ready.load()) {
        std::this_thread::sleep_for(1ms);
    }

    std::cout << "Message received = " << message << std::endl;
}

void writer() {
    message = "Hello, World!";
    ready.store(true);
}
```

```
int main() {
    std::thread t1(reader);
    std::thread t2(writer);

    t1.join();
    t2.join();

    return 0;
}
```

In this example, `reader()` waits until the `ready` variable is `true` and then prints a message set by `writer()`. The `writer()` function sets the message and then sets the `store` variable to `true`.

Atomic operations provide us with two features for enforcing a certain order of execution in multithreaded code:

- **Happens before**: In the preceding code, `[1]` (setting the `message` variable) happens before `[2]` (setting the atomic `ready` variable to `true`). Also, `[3]`, reading the `ready` variable in a loop until it is `true`, happens before `[4]`, printing the message. In this case, we are using sequential consistency memory order (the default memory order).
- **Synchronizes with**: This only happens between atomic operations. In the preceding example, this means that when `ready` is set by `[1]` the value will be visible for subsequent reads (or writes) in different threads (of course, it is visible in the current thread), and when `ready` is read by `[3]`, the changed value will be visible.

Now that we have seen how atomic operations enforce memory access order from different threads, let's see in detail each of the memory order options provided by the C++ memory model.

Before we start, let's remember here that the Intel x64 architecture (Intel and AMD desktop processors) is quite restrictive in relation to memory order, that there is no need for any additional instructions for acquire/release, and sequential consistency is cheap in terms of performance cost.

Sequential consistency

Sequential consistency guarantees the execution of the program in the way you wrote it. In 1979 Leslie Lamport defined sequential consistency as being "*the result of an execution is the same as if the reads and writes occurred in some order, and the operations of each individual processor appear in this sequence in the order specified by its program.*"

In C++, sequential consistency is specified with the `std::memory_order_seq_cst` option. This is the most stringent memory order and it's also the default one. If no ordering option is specified, then sequential consistency will be used.

The C++ memory model by default ensures sequential consistency in the absence of race conditions within your code. Consider it a pact: if we properly synchronize our program to prevent race conditions, C++ will maintain the appearance that the program executes in the sequence it was written.

In this model, all threads must see the same order of operations. Operations can still be reordered as far as the visible result of the computation has the same result as the result of the unordered code. The instructions and operations can be reordered if the reads and writes are performed in the same order as in the compiled code. The CPU is free to reorder any other instructions between the reads and writes if the dependencies are satisfied. Because of the consistent ordering it defines, sequential consistency is the most intuitive form of ordering. To illustrate sequential consistency, let's consider the following example:

```
#include <atomic>
#include <chrono>
#include <iostream>
#include <thread>

std::atomic<bool> x{ false };
std::atomic<bool> y{ false };
std::atomic<int> z{ 0 };

void write_x() {
    x.store(true, std::memory_order_seq_cst);
}

void write_y() {
    y.store(true, std::memory_order_seq_cst);
}

void read_x_then_y() {
    while (!x.load(std::memory_order_seq_cst)) {}

    if (y.load(std::memory_order_seq_cst)) {
        ++z;
    }
}

void read_y_then_x()
{
    while (!y.load(std::memory_order_seq_cst)) {}

    if (x.load(std::memory_order_seq_cst)) {
        ++z;
    }
```

```
}

int main() {
    std::thread t1(write_x);
    std::thread t2(write_y);
    std::thread t3(read_x_then_y);
    std::thread t4(read_y_then_x);

    t1.join();
    t2.join();
    t3.join();
    t4.join();

    if (z.load() == 0) {
        std::cout << "This will never happen\n";
    }
    {
        std::cout << "This will always happen and z = " << z << "\n";
    }

    return 0;
}
```

Because we are using `std::memory_order_seq_cst` when running the code, we should note the following:

- Operations in each thread are executed in the given order (no reordering of atomic operations).
- `t1` and `t2` update `x` and `y` in order, and `t3` and `t4` see the same order. Without this property, `t3` could see `x` and `y` change in order, but `t4` could see the opposite.
- Any other ordering may print `This will never happen` because `t3` and `t4` could see the changes to `x` and `y` in the opposite order. We will see an example of this in the next section.

The sequential consistency in this example means that the following two things will happen:

- Each store is seen by all the threads; that is, each store operation synchronizes with all the load operations for each variable, and all the threads see these changes in the same order they are made
- The operations happen in the same order for each thread (operations run in the same order as in the code)

Please note that the order between operations in different threads is not guaranteed and instructions from different threads may be executed in any order because the threads may be scheduled.

Acquire-release ordering

Acquire-release ordering is less stringent than sequential consistency ordering. We don't get the total ordering of operations we had with sequential consistency ordering, but some synchronization is still possible. In general, as we add more freedom to the memory ordering we may see a performance gain, but it will get more difficult to reason about the execution order of our code.

In this ordering model, the atomic load operations are the `std::memory_order_acquire` operations, the atomic store operations are the `std::memory_order_release` operations, and the atomic read-modify-write operations may be the `std::memory_order_acquire`, `std::memory_order_release` or `std::memory_order_acq_rel` operations.

Acquire semantics (used with `std::memory_order_acquire`) ensure that all of the read or write operations in one thread that appear *after* the acquire operation in the source code happen after the acquire operation. This prevents the memory from reordering the reads and writes that follow the acquire operation.

Release semantics (used with `std::memory_order_release`) ensure that the read or write operations in one thread that appear *before* the release operation in the source code are completed before the release operation. This prevents the memory reordering of the reads and writes that follow the release operation.

The following example shows the same code as that shown in the previous section about sequential consistency, but in this case, we use the acquire-release memory order for the atomic operations:

```
#include <atomic>
#include <chrono>
#include <iostream>
#include <thread>

std::atomic<bool> x{ false };
std::atomic<bool> y{ false };
std::atomic<int> z{ 0 };

void write_x() {
    x.store(true, std::memory_order_release);
}

void write_y() {
    y.store(true, std::memory_order_release);
}

void read_x_then_y() {
    while (!x.load(std::memory_order_acquire)) {}
```

```
    if (y.load(std::memory_order_acquire)) {
        ++z;
    }
}

void read_y_then_x() {
    while (!y.load(std::memory_order_acquire)) {}

    if (x.load(std::memory_order_acquire)) {
        ++z;
    }
}

int main() {
    std::thread t1(write_x);
    std::thread t2(write_y);
    std::thread t3(read_x_then_y);
    std::thread t4(read_y_then_x);

    t1.join();
    t2.join();
    t3.join();
    t4.join();

    if (z.load() == 0) {
        std::cout << "This will never happen\n";
    }
    {
        std::cout << "This will always happen and z = " << z << "\n";
    }

    return 0;
}
```

In this case, it is possible for the value of `z` to be 0. Because we don't have sequential consistency anymore after `t1` sets `x` to `true` and `t2` sets `y` to `true`, `t3` and `t4` may have different views of how memory access is being performed. Because of the use of the acquire-release memory ordering, `t3` may see `x` as `true` and `y` as `false` (remember, there is no enforce ordering) and `t4` may see `x` as `false` and `y` as `true`. When this happens, the value of `z` will be 0.

Besides `std::memory_order_acquire`, `std::memory_order_release`, and `std::memory_order_acq_rel`, the acquire-release memory ordering also includes the `std::memory_order_consume` option. We won't be describing it because according to the online C++ reference, "*the specification of release-consume ordering is being revised, and the use of std::memory_order_consume is temporarily discouraged.*"

Relaxed memory ordering

To perform the atomic operation with **relaxed memory ordering**, we specify `std::memory_order_relaxed` as the memory order option.

Relaxed memory ordering is the weakest form of synchronization. It offers two guarantees:

- Atomicity of the operations.
- Atomic operations on the same atomic variable in a single thread are not reordered. This is called **modification order consistency**. There is no guarantee, however, that the other threads will see these operations in the same order.

Let's consider the following scenario: one thread (`th1`) stores values into an atomic variable. After a certain random interval of time, the variable will be overwritten with a new random value. We should assume for the sake of this example, that the sequence written is 2, 12, 23, 4, 6. Another thread, `th2`, reads the same variable periodically. The first time the variable is read, `th2` gets the value 23. Remember that the variable is atomic and that both load and store operations are done using the relaxed memory order.

If `th2` reads the variable again, it can get the same value or any value written *after* the previously read value. It cannot read any value written before because the modification order consistency property would be violated. In the current example, the second read may get 23, 4, or 6 but not 2 or 12. If we get 4, th1 will go on to write 8, 19, and 7. Now th2 may get 4, 6, 8, 19, or 7 but not any number before 4 and so on.

Between two or more threads, there is no guarantee of any order, but once a value is read, a previously written value cannot be read.

The relaxed model cannot be used to synchronize threads, because there is no visibility order guarantee, but it is useful in scenarios where operations do not need to be coordinated tightly between threads, which can lead to performance improvements.

It is generally safe to use when the order of execution does not affect the correctness of the program, such as incrementing counters used for statistics or reference counters where the exact order of increment is not important.

In this section, we learned about the C++ memory model and how it allows the order and synchronization of atomic operations with different memory order constraints. In the next section, we will see the atomic types and operations provided by the C++ Standard Library.

C++ Standard Library atomic types and operations

We will now introduce the data types and functions provided by the C++ Standard Library to support atomic types and operations. As we have already seen, an atomic operation is an indivisible operation. To be able to perform atomic operations in C++, we need to use the atomic types provided by the C++ Standard Library.

C++ Standard Library atomic types

The atomic types provided by the C++ Standard Library are defined in the `<atomic>` header file.

You can see the documentation for all the atomic types defined in the `<atomic>` header in the online C++ reference, which you can access at `https://en.cppreference.com/w/cpp/atomic/atomic`. We won't include all the content in this reference here (that's what the reference is for!), but we will introduce the main concepts and use examples to further elaborate our explanations.

The atomic types provided by the C++ Standard Library are the following:

- `std::atomic_flag`: Atomic Boolean type (but different from `std::atomic<bool>`). It is the only atomic type that is guaranteed to be lock-free. It does not provide load or store operations. It is the most basic atomic type of all. We will use it to implement a very simple mutex-like lock.
- `std::atomic<T>`: This is a template for defining atomic types. All the intrinsic types have their own corresponding atomic type defined using this template. The following are some examples of these types:
  - `std::atomic<bool>` (and its alias `atomic_bool`): We will use this atomic type to implement the lazy one-time initialization of a variable from several threads.
  - `std::atomic<int>` (and its alias `atomic_int`): We have seen this atomic type already in the simple counter example. We will use it again in an example to gather statistics (very similar to the counter example).
  - `std::atomic<intptr_t>` (and its alias `atomic_intptr_t`).
  - C++20 introduced atomic smart pointers: `std::atomic<std::shared_ptr<U>>` and `std::atomic<std::weak_ptr<U>>`.
- Since the release of C++20, there is a new atomic type, `std::atomic_ref<T>`.

In this chapter, we will focus on `std::atomic_flag` and some of the `std::atomic` types. For the other atomic types we have mentioned here, you can access the online C++ reference using the previous link.

Before any further explanation of some of these types, there is a very important clarification to be made: just because a type is *atomic*, that doesn't guarantee it is *lock-free*. By atomic here, we mean indivisible operation, and by lock-free, we mean with special CPU atomic instructions support. If there is no hardware support for certain atomic operations, they will be implemented using locks by the C++ Standard Library.

To check whether an atomic type is lock-free we can use the following member function of any of the `std::atomic<T>` types:

- `bool is_lock_free() const noexcept`: This returns `true` if all the atomic operations of this type are lock-free, and `false` otherwise (except for `std::atomic_flag`, which is guaranteed to always be lock-free). The rest of the atomic types can be implemented using locks such as mutexes to guarantee the atomicity of the operations. Also, some atomic types may be lock-free only sometimes. If only aligned memory access can be lock-free in a certain CPU, then the misaligned objects of that same atomic type will be implemented using locks.

There is also a constant used to indicate whether an atomic type is always lock-free:

- `static constexpr bool is_always_lock_free = /* implementation defined */`: The value of this constant will be `true` if the atomic type is always lock-free (even for misaligned objects, for example)

It is important to be aware of this: an atomic type is not guaranteed to be lock-free. The `std::atomic<T>` template is not a magic mechanism that can turn all atomic types into lock-free atomic types.

C++ Standard Library atomic operations

There are two main types of atomic operations:

- **Member functions of atomic types**: For example, `std::atomic<int>` has the `load()` member function to atomically read its value
- **Free functions**: The `const std::atomic_load(const std::atomic<T>* obj)` function does exactly the same as the previous one

You can access the following code (and the generated assembly code, if you are interested) at `https://godbolt.org/z/Yhdr3Y1Y8`. This code shows the use of both member functions and free functions:

```
#include <atomic>
#include <iostream>
std::atomic<int> counter {0};
int main() {
    // Using member functions
    int count = counter.load();
```

```
    std::cout << count << std::endl;
    count++;
    counter.store(count);

    // Using free functions
    count = std::atomic_load(&counter);
    std::cout << count << std::endl;

    count++;
    std::atomic_store(&counter, count);

    return 0;
}
```

Most of the atomic operation functions have a parameter to indicate the memory order. We have already explained what the memory order is, and what memory ordering types are provided by C++ in the section about the C++ memory model.

Example – simple spin-lock implemented using the C++ atomic flag

The `std::atomic_flag` atomic type is the most basic standard atomic type. It only has two states: set and not set (which we can also call true and false). It is always lock-free, in contrast to any other standard atomic type. Because it is so simple, it is mainly used as a building block.

This is the code for the atomic flag example:

```
#include <atomic>
#include <chrono>
#include <iostream>
#include <thread>
#include <vector>

class spin_lock {
public:
    spin_lock() = default;

    spin_lock(const spin_lock &) = delete;

    spin_lock &operator=(const spin_lock &) = delete;

    void lock() {
        while  (flag.test_and_set(std::memory_order_acquire)) {
        }
```

```
    }

    void unlock() {
        flag.clear(std::memory_order_release);
    }

private:
    std::atomic_flag flag = ATOMIC_FLAG_INIT;
};
```

We need to initialize `std::atomic_flag` before using it. The following code shows how to do that:

```
std::atomic_flag flag = ATOMIC_FLAG_INIT;
```

This is the only way to initialize `std::atomic_flag` to a definite value. The value of `ATOMIC_FLAG_INIT` is implementation defined.

Once the flag is initialized, we can perform two atomic operations on it:

- `clear`: This atomically sets the flag to `false`
- `test_and_set`: This atomically sets the flag to `true` and obtains its previous value

The `clear` function can only be called with a relaxed, release, or sequential consistency memory order. The `test_and_set` function can only be called with relaxed, acquire, or sequential consistency. Using any other memory order will result in undefined behavior.

Now let's see how we can implement a simple spinlock using `std::atomic_flag`. First, we know that the operations are atomic, so the thread either clears the flag or it doesn't, and if a thread clears the flag, it is fully cleared. It is not possible for the thread to *half-clear* the flag (remember this would be possible for some non-atomic flags). The `test_and_set` function is atomic too, so the flag is set to `true`, and we get the previous state in just one go.

To implement the basic spinlock, we need an atomic flag to atomically handle the lock status and two functions: `lock()` to acquire the lock (as we have for a mutex) and `unlock()` to release the lock.

Simple spin lock unlock() function

We will begin with `unlock()`, the simplest function. It will only reset the flag (by making it false) and nothing more:

```
void unlock()
{
    flag.clear(std::memory_order_release);
}
```

The code is straightforward. If we leave out the `std::memory_order_seq_cst` parameter, the strictest memory order option, sequential consistency, will be applied.

Simple spin lock lock() function

The lock function has more steps. First, let's explain what it does: `lock()` must see whether the atomic flag is on. If it is off, then turn it on and finish. If the flag is on, then keep on looking until another thread turns it off. We will use `test_and_set()` to make this function work:

```
void lock()
{
    while (flag.test_and_set(std::memory_order_acquire)) {}
}
```

The preceding code works in the following way: inside a `while` loop, `test_and_set` sets the flag to `true` and returns the previous value. If the flag is already set, setting it again doesn't change anything and the function returns `true`, so the loop keeps on setting the flag. When, eventually, `test_and_set` returns `false`, this means that the flag was cleared and we can exit the loop.

Simple spin lock issues

The simple spin lock implementation has been included in this chapter to introduce the use of atomic types (`std::atomic_flag`, the simplest standard atomic type) and operations (`clear` and `test_and_set`), but it has some serious issues:

- The first of these is its bad performance. The code in the repo will let you experiment. Expect the spinlock performance to be much worse than that of the mutex.
- The thread is spinning all the time waiting for the flag to be cleared. This busy wait is something to avoid, especially if there is thread contention.

You can try out the preceding code for this example. We got these results, shown in *Table 5.1*, when we ran it. The code adds 1 to a counter 200 million times in each thread.

	std::mutex	**spinlock**	**atomic counter**
One thread	1.03 s	1.33 s	0.82 s
Two threads	10.15 s	39.14 s	4.52 s
Four threads	24.61 s	128.84 s	9.13 s

Table 5.1: Synchronization primitives profiling results

We can see from the preceding table how poorly the simple spinlock works and how it worsens with the addition of threads. Note that this simple example is only for learning and that both the simple `std::mutex` spinlock and the atomic counter can be improved so that the atomic type performs better.

In this section, we have looked at `std::atomic_flag`, the most basic atomic type provided by the C++ Standard Library. For further information about this type and about the new functionality added in C++20 please refer to the online C++ reference, which is available at `https://en.cppreference.com/w/cpp/atomic/atomic_flag`.

In the following section, we will look at how to create a simple way for a thread to tell the main thread how many items it has processed.

Example – thread progress reporting

Sometimes we want to check the progress of a thread or be notified when it finishes. This can be done in different ways, for example, using a mutex and a condition variable, or a shared variable synchronized by a mutex, as we have seen in *Chapter 4*. We also saw how to use atomic operations to synchronize a counter in this chapter. We will use a similar counter in the following example:

```
#include <atomic>
#include <chrono>
#include <iostream>
#include <thread>

constexpr int NUM_ITEMS{100000};

int main() {
    std::atomic<int> progress{0};

    std::thread worker([&progress] {
        for (int i = 1; i <= NUM_ITEMS; ++i) {
            progress.store(i, std::memory_order_relaxed);
            std::this_thread::sleep_for(std::chrono::milliseconds(1));
        }
    });

    while (true) {
        int processed_items = progress.load(std::memory_order_
relaxed);
        std::cout << "Progress: "
                  << processed_items << " / " << NUM_ITEMS
```

```
                    << std::endl;
        if (processed_items == NUM_ITEMS) {
            break;
        }
        std::this_thread::sleep_for(std::chrono::seconds(10));
    }

    worker.join();

    return 0;
}
```

The preceding code implements a thread (`worker`) that handles a certain number of items (here the handling is simulated just by making the thread sleep). Every time the thread handles an item, it increments the variable progress. The main thread executes a `while` loop and, in each iteration, it accesses the `progress` variable and writes a report of the progress (number of items handled). Once all the items are handled, the loop is finished.

In this example, we use the `std::atomic<int>` atomic type (an atomic integer) and two atomic operations:

- `load()`: This atomically retrieves the value of the `progress` variable
- `store()`: This atomically modifies the value of the `progress` variable

The `worker` thread processing `progress` is read and written atomically, so no race conditions occur when two threads access the `progress` variable.

The `load()` and `store()` atomic operations have an extra parameter to indicate the memory order. In this example, we have used `std::memory_order_relaxed`. This is a typical example of the use of the relaxed memory order: one thread increases a counter, and another reads it. The only ordering we need is reading increasing values and for that, the relaxed memory order is enough.

Having introduced the `load()` and `store()` atomic operations to atomically read and write a variable, let's see another example of a simple statistic-gathering application.

Example – simple statistics

This example builds on the same idea as the previous one: a thread can use atomic operations to communicate progress (for example, the number of items processed) to another thread. In this new example, one thread will produce some data that another thread will read. We need to synchronize memory access because we have two threads sharing the same memory and at least one of them is changing the memory. As in the previous example, we will use atomic operations for this purpose.

The following code declares the atomic variables we are going to use to gather statistics – one for the number of items processed and two more (for the total processing time and average processing time for each item, respectively):

```
std::atomic<int> processed_items{0};
std::atomic<float> total_time{0.0f};
std::atomic<double> average_time{0.0};
```

We use atomic float and double for total time and average time. In the full example code, we make sure both types are lock-free, which means they use atomic instructions from the CPU (all modern CPUs should have that).

Now let's see how the worker thread uses the variables:

```
processed_items.fetch_add(1, std::memory_order_relaxed);
total_time.fetch_add(elapsed_s, std::memory_order_relaxed);
average_time.store(total_time.load() / processed_items.load(),
std::memory_order_relaxed);
```

The first line increments the processed items by 1 in an atomic way. The `fetch_add` function adds `1` to the variable value and gives back the old value (we are not using it in this case).

The second line adds `elapsed_s` (the time it took to process one item in seconds) to the `total_time` variable, which we use to keep track of the time it takes to process all the items.

Then, the third line computes the mean time for each item by atomically reading `total_time` and `processed_items` and atomically writing the result in `average_time`. Alternatively, we could use the values from the `fetch_add()` calls to calculate the mean time, but they don't include the last item that was processed. We could also do the calculation of `average_time` in the main thread, but we do it in the worker thread here, just as an example and to practice using atomic operations. Keep in mind that our aim (at least in this chapter) is not so much speed but learning how to use atomic operations.

The following is the full code for the statistics example:

```
#include <atomic>
#include <chrono>
#include <iostream>
#include <random>
#include <thread>

constexpr int NUM_ITEMS{10000};

void process() {
    std::random_device rd;
    std::mt19937 gen(rd());
```

```
    std::uniform_int_distribution<> dis(1, 20);

    int sleep_duration = dis(gen);
        std::this_thread::sleep_for(std::chrono::milliseconds(sleep_
duration));
}

int main() {
    std::atomic<int> processed_items{0};
    std::atomic<float> total_time{0.0f};
    std::atomic<double> average_time{0.0};

    std::thread worker([&] {
        for (int i = 1; i <= NUM_ITEMS; ++i) {
            auto now = std::chrono::high_resolution_clock::now();
            process();
            auto elapsed =
                std::chrono::high_resolution_clock::now() - now;
            float elapsed_s =
                std::chrono::duration<float>(elapsed).count();

            processed_items.fetch_add(1, std::memory_order_relaxed);
            total_time.fetch_add(elapsed_s, std::memory_order_
relaxed);
            average_time.store(total_time.load() / processed_items.
load(), std::memory_order_relaxed);
        }
    });

    while (true) {
        int items = processed_items.load(std::memory_order_relaxed);
        std::cout << "Progress: " << items << " / " << NUM_ITEMS <<
std::endl;

        float time = total_time.load(std::memory_order_relaxed);
        std::cout << "Total time: " << time << " sec" << std::endl;

        double average = average_time.load(std::memory_order_relaxed);
        std::cout << "Average time: " << average * 1000 << " ms" <<
std::endl;

        if (items == NUM_ITEMS) {
            break;
        }
```

```
        std::this_thread::sleep_for(std::chrono::seconds(5));
    }

    worker.join();

    return 0;
}
```

Let's summarize what we have seen up to this point in the current section:

- C++ standard atomic types: we used `std::atomic_flag` to implement a simple spinlock and we have used some of the `std::atomic<T>` types to implement communication of simple data between threads. All the atomic types that we have seen are lock-free.
- The `load()` atomic operation to atomically read the value of an atomic variable.
- The `store()` atomic operation to atomically write a new value to an atomic variable.
- `clear()` and `test_and_set()`, the special atomic operations provided by `std::atomic_flag`.
- `fetch_add()`, to atomically add some value to an atomic variable and get its previous value. Integral and floating-point types also implement `fetch_sub()`, to subtract a certain value from an atomic variable and return its previous value. Some functions for performing bitwise logic operations have been implemented just for integral types: `fetch_and()`, `fetch_or()`, and `fetch_xor()`.

The following table summarizes atomic types and operations. For an exhaustive description, please refer to the online C++ reference: `https://en.cppreference.com/w/cpp/atomic/atomic`

The table shows three new operations: `exchange`, `compare_exchange_weak`, and `compare_exchange_strong`. We will explain them using an example later. Most of the operations (that is, the functions, not the operators) have another parameter for the memory order.

Operation	**atomic_flag**	**atomic <bool>**	**atomic <integral>**	**atomic <floating-point>**	**atomic <other>**
`test_and_set`	YES				
`Clear`	YES				
`Load`		YES	YES	YES	YES
`Store`		YES	YES	YES	YES
`fetch_add, +=`			YES	YES	
`fetch_sub, -=`			YES	YES	

Operation	atomic_ flag	atomic <bool>	atomic <integral>	atomic <floating-point>	atomic <other>
fetch_and, &=			YES		
fetch_or, |=			YES		
fetch_xor, ^=			YES		
++, --			YES		
Exchange		YES	YES	YES	YES
compare_ exchange_weak, compare_ exchange_ strong		YES	YES	YES	YES

Table 5.2: Atomic types and operations

Let's review the `is_lock_free()` function and the `is_always_lock_free` constant. We saw that if `is_lock_free()` is true, then the atomic type has lock-free operations with special CPU instructions. An atomic type can be lock-free only sometimes, so the `is_always_lock_free` constant tells us if the type is always lock-free. So far, all the types we have seen are lock-free. Let's see what happens when an atomic type is non-lock-free.

The following shows the code for the non-lock-free atomic type:

```
#include <atomic>
#include <iostream>

struct no_lock_free {
    int a[128];

    no_lock_free() {
        for (int i = 0; i < 128; ++i) {
            a[i] = i;
        }
    }
};

int main() {
    std::atomic<no_lock_free> s;

    std::cout << "Size of no_lock_free: " << sizeof(no_lock_free) << "
```

```
bytes\n";
    std::cout << "Size of std::atomic<no_lock_free>: " << sizeof(s) <<
" bytes\n";

    std::cout << "Is std::atomic<no_lock_free> always lock-free: " <<
std::boolalpha
              << std::atomic<no_lock_free>::is_always_lock_free <<
std::endl;
    std::cout << "Is std::atomic<no_lock_free> lock-free: " <<
std::boolalpha << s.is_lock_free() << std::endl;

    no_lock_free s1;
    s.store(s1);

    return 0;
}
```

When you execute the code, you will notice that the `std::atomic<no_lock_free>` type is not lock-free. Its size, 512 bytes, is the cause of this. When we assign a value to the atomic variable, that value is written *atomically*, but this operation does not use CPU atomic instructions, that is, it is not lock-free. The implementation of this operation depends on the compiler but, in general, it uses either a mutex or a special spinlock (such as Microsoft Visual C++).

The lesson here is that all atomic types have atomic operations, but they are not all magically lock-free. If an atomic type is not lock-free, it is always better to implement it using locks.

We learned that some atomic types are not lock-free. Now we will look at another example that shows the atomic operations we have not covered yet: the `exchange` and `compare_exchange` operations.

Example – lazy one-time initialization

Sometimes initializing an object can be costly. For example, a given object may need to connect to a database or a server, and establishing this connection can take a long time. In these cases, we should initialize the object just before its use, and not when we define it in our program. This is called **lazy initialization**. Now let's assume that more than one thread needs to use the object for the first time. If more than one thread initializes the object, then different connections would be created, and that would be wrong because the object opens and closes only one connection. For this reason, multiple initializations must be avoided. To ensure the object is initialized only once, we will utilize a method known as lazy one-time initialization.

The following shows the code for lazy one-time initialization:

```
#include <atomic>
#include <iostream>
#include <random>
```

```
#include <thread>
#include <vector>

constexpr int NUM_THREADS{8};

void process() {
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> dis(1, 1000000);

    int sleep_duration = dis(gen);

    std::this_thread::sleep_for(std::chrono::microseconds(sleep_
duration));
}

int main() {
    std::atomic<int> init_thread{0};

    auto worker = [&init_thread](int i) {
        process();

        int init_value = init_thread.load(std::memory_order::seq_cst);
        if (init_value == 0) {
            int expected = 0;
            if (init_thread.compare_exchange_strong(expected, i,
std::memory_order::seq_cst)) {
                std::cout << "Previous value of init_thread: " <<
expected << "\n";
                std::cout << "Thread " << i << " initialized\n";
            } else {
                // init_thread was already initialized
            }
        } else {
            // init_thread was already initialized
        }
    };


    std::vector<std::thread> threads;
    for (int i = 1; i <= NUM_THREADS; ++i) {
        threads.emplace_back(worker, i);
    }
```

```
    for (auto &t: threads) {
        t.join();
    }

    std::cout << "Thread: " << init_thread.load() << " initialized\n";

    return 0;
}
```

There are some operations in the atomic type operations table that we saw earlier in this chapter that we have not yet discussed. We will now explain `compare_exchange_strong` using an example. In the example, we have a variable that starts with a value of 0. Several threads are running, each with a unique integer ID (1, 2, 3, and so on). We want to set the variable's value to the ID of the thread that sets it first and initialize the variable only once. In *Chapter 4*, we learned about `std::once_flag` and `std::call_once`, which we could use to implement this one-time initialization, but this chapter is about atomic types and operations, so we will use those to achieve our goal.

To be sure that the initialization of the `init_thread` variable is done only once and to avoid race conditions due to write access from more than one thread, we use an atomic `int`. Line `[1]` atomically reads the content of `init_thread`. If the value is not 0, then that means it has been already initialized and the worker thread does nothing else.

The current value of `init_thread` is stored in the `expected` variable, which represents the value we expect `init_thread` will have when we try to initialize it. Now line `[2]` performs the following steps:

1. Compare the `init_thread` current value to the `expected` value (which, again, is equal to 0).
2. If the comparison is not successful, copy the `init_thread` current value into `expected` and return `false`.
3. If the comparison is successful, copy the `init_thread` current value into `expected`, then set the `init_thread` current value to `i` and return `true`.

The current thread will have initialized `init_thread` only if `compare_exchange_strong` returns `true`. Also, note that we need to perform a comparison again (even if line `[1]` returned 0 as the current value of `init_thread`) because it is possible that another thread has already initialized the variable.

It is very important to note that if `compare_exchange_strong` returns `false`, then the comparison has failed, and if it returns `true`, then the comparison was successful. This is always true of `compare_exchange_strong`. On the other hand, `compare_exchange_weak` can fail (i.e., return `false`) even if the comparison is successful. The reason for using it is that in some platforms it gives better performance when it is called inside a loop.

For more information on these two functions, please refer to the online C++ reference: `https://en.cppreference.com/w/cpp/atomic/atomic/compare_exchange`

In this section about the C++ Standard Library atomic types and operations, we have seen the following:

- The most commonly used standard atomic types, such as `std::atomic_flag` and `std::atomic<int>`
- The most-used atomic operations: `load()`, `store()`, and `exchange_compare_strong()/exchange_compare_weak()`
- Basic examples incorporating these atomic types and operations, including lazy one-time initialization and thread progress communication

We have mentioned several times that most of the atomic operations (functions) let us pick the memory order we want to use. In the next section, we will implement a lock-free programming example: an SPSC lock-free queue.

SPSC lock-free queue

We have already looked at the C++ Standard Library's features for atomics, such as atomic types and operations and the memory model and orderings. Now we will see a complete example of using atomics to implement an SPSC lock-free queue.

The main features of this queue are the following:

- **SPSC**: This queue is designed to work with two threads, one pushing elements to the queue and another getting elements from the queue.
- **Bounded**: This queue has a fixed size. We need a method for checking when the queue reaches its capacity and when it has no elements).
- **Lock-free**: This queue uses atomic types that are always lock-free on modern Intel x64 CPUs.

Before you begin to develop the queue, keep in mind that lock-free is not the same as wait-free (also keep in mind that wait-free does not eliminate waiting entirely; it just ensures that there is a limit to the number of steps required for each queue push/pop). Some aspects that mostly affect performance will be discussed in *Chapter 13*. In that chapter, we will also optimize the queue's performance. For now, in this chapter, we will build an SPSC lock-free queue that is correct and performs adequately – we will show how its performance can be improved later.

We used mutex and condition variables to make an SPSC queue in *Chapter 4* that consumer and producer threads could access safely. This chapter will use atomic operations to achieve the same goal.

We will store the items in the queue using the same data structure: `std::vector<T>` with a fixed size, that is, a power of 2. This way, we can improve performance and find the next head and tail indices quickly without using the modulo operator that needs a division instruction. When using lock-free atomic types for better performance, we need to pay attention to everything that affects performance.

Why do we use a power of 2 buffer size?

We will use a vector to hold the queue items. The vector will have a fixed size, say `N`. We will make the vector act similarly to a ring buffer, meaning that the index for accessing an element in the vector will loop back to the start after the end. The first element will follow the last one. As we learned in *Chapter 4*, we can do this with the modulo operator:

```
size_t next_index = (curr_index + 1) % N;
```

If the size is, for example, four elements, the index to the next element will be calculated as in the preceding code. For the last index, we have the following code:

```
next_index = (3 + 1) % 4 = 4 % 4 = 0;
```

Therefore, as we said, the vector will be a ring buffer because, after the last element, we will go back to the first one, then the second one, and so on.

We can use this method to get the next index for any buffer size `N`. But why do we only use sizes that are powers of 2? The answer is easy: performance. The modulo (`%`) operator requires a division instruction, which is expensive. When the size `N` is a power of 2, we can just do the following:

```
size_t next_index = curr_index & (N - 1);
```

This is much faster than using the modulo operator.

Buffer access synchronization

To access the queue buffer, we need two indices:

- `head`: The index of the current element to be read
- `tail`: The index of the next element to be written

The consumer thread will use the head index to read and write. The producer thread will use the tail index to read and write. We need to synchronize access to these variables because of this:

- Only one thread (the consumer) writes `head`, meaning that it can read it with relaxed memory ordering because it always sees its own changes. Reading `tail` is done by the reader thread and it needs to synchronize with the producer's writing of `tail`, so it needs acquire memory ordering. We could use sequential consistency for everything, but we want the best performance. When the consumer thread writes `head`, it needs to synchronize with the producer's read of it, so it needs release memory ordering.
- For `tail`, only the producer thread writes it, so we can use relaxed memory ordering to read it, but we need release memory ordering to write it and synchronize it with the consumer thread's reading. To synchronize with the consumer thread's writing, we need acquire memory ordering to read `head`.

The queue class member variables are the following:

```
const std::size_t capacity_; // power of two buffer size
std::vector<T> buffer_; // buffer to store queue items handled like a
ring buffer
std::atomic<std::size_t> head_{ 0 };
std::atomic<std::size_t> tail_{ 0 };
```

In this section, we have seen how to synchronize access to queue buffer.

Pushing elements into the queue

Once we have decided on the data representation of the queue and how to synchronize access to its elements, let's implement the function for pushing elements into the queue:

```
bool push(const T& item) {
    std::size_t tail =
        tail_.load(std::memory_order_relaxed);

    std::size_t next_tail =
       (tail + 1) & (capacity_ - 1);

    if (next_tail != head_.load(std::memory_order_acquire)) {
        buffer_[tail] = item;
        tail_.store(next_tail, std::memory_order_release);
        return true;
    }

    return false;
}
```

The current tail index, which is the buffer slot where the data item is to be pushed (if possible) into the queue, is atomically read in line `[1]`. As we mentioned earlier, this read can use `std::memory_order_relaxed` because only the producer thread changes this variable, and it is the only thread that calls push.

Line `[2]` calculates the next index modulo capacity (remember that the buffer is a ring). We need to do this to check whether the queue is full.

We perform the check in line `[3]`. We first atomically read the current value of the head using `std::memory_order_acquire` because we want the producer thread to observe the modifications that the consumer thread has made to this variable. Then we compare its value with the next head index.

If the next tail value is equal to the current head value, then (as per our convention) the queue is full, and we return `false`.

If the queue is not full, line `[4]` copies the data item to the queue buffer. It is worth commenting here that the data copy is not atomic.

Line `[5]` atomically writes the new tail index value into `tail_`. Then, `std::memory_order_release` is used to make the changes visible to the consumer thread that atomically reads this variable with `std::memory_order_acquire`.

Popping elements from the queue

Let's now see how the `pop` function is implemented:

```
bool pop(T& item) {
    std::size_t head =
        head_.load(std::memory_order_relaxed);

    if (head == tail_.load(std::memory_order_acquire)) {
        return false;
    }

    item = buffer_[head];

    head_.store((head + 1) & (capacity_ - 1), std::memory_order_
release);

    return true;
}
```

Line `[1]` atomically reads the current value of `head_` (index for the next item to be read). We use `std::memory_order_relaxed` because no order enforcement is required due to the `head_` variable being modified only by the consumer thread, which is the only thread calling `pop`.

Line `[2]` checks whether the queue is empty. If the current value of `head_` is the same as the current value of `tail_`, then the queue is empty, and the function just returns `false`. We atomically read the value of `tail_` with `std::memory_order_acquire` to see the latest change done to `tail_` by the producer thread.

Line `[3]` copies the data from the queue to the item reference passed as an argument to `pop`. Again, this copy is not atomic.

Finally, line `[4]` updates the value of `head_`. Again, we atomically write the value using `std::memory_order_release` for the consumer thread to see the changes made to `head_` by the consumer thread.

The code for the SPSC lock-free queue implementation is the following:

```
#include <atomic>
#include <cassert>
#include <iostream>
#include <vector>
#include <thread>

template<typename T>
class spsc_lock_free_queue {
public:
    // capacity must be power of two to avoid using modulo operator
when calculating the index
    explicit spsc_lock_free_queue(size_t capacity) : capacity_
(capacity), buffer_(capacity) {
        assert((capacity & (capacity - 1)) == 0 && "capacity must be a
power of 2");
    }

    spsc_lock_free_queue(const spsc_lock_free_queue &) = delete;

    spsc_lock_free_queue &operator=(const spsc_lock_free_queue &) =
delete;

    bool push(const T &item) {
        std::size_t tail = tail_.load(std::memory_order_relaxed);
        std::size_t next_tail = (tail + 1) & (capacity_ - 1);
        if (next_tail != head_.load(std::memory_order_acquire)) {
            buffer_[tail] = item;
            tail_.store(next_tail, std::memory_order_release);
            return true;
        }

        return false;
    }

    bool pop(T &item) {
        std::size_t head = head_.load(std::memory_order_relaxed);
        if (head == tail_.load(std::memory_order_acquire)) {
            return false;
        }

        item = buffer_[head];
        head_.store((head + 1) & (capacity_ - 1), std::memory_order_
release);
```

```
        return true;
    }

private:
    const std::size_t capacity_;
    std::vector<T> buffer_;
    std::atomic<std::size_t> head_{0};
    std::atomic<std::size_t> tail_{0};
};
```

The code for the full example can be found in the following book repo: `https://github.com/PacktPublishing/Asynchronous-Programming-in-CPP/blob/main/Chapter_05/5x09-SPSC_lock_free_queue.cpp`

In this section, we have implemented an SPSC lock-free queue as an application of atomic types and operations. In *Chapter 13*, we will revisit this implementation and improve its performance.

Summary

This chapter has introduced atomic types and operations, the C++ memory model, and a basic implementation of an SPSC lock-free queue.

The following is a summary of what we have looked at:

- The C++ Standard Library atomic types and operations, what they are, and how to use them with some examples.
- The C++ memory model, and especially the different memory orderings it defines. Bear in mind that this is a very complex subject and that this section was just a basic introduction to it.
- How to implement a basic SPSC lock-free queue. As we mentioned previously, we will demonstrate how to improve its performance in *Chapter 13*. Examples of performance-improving actions include eliminating false sharing (what happens when two variables are in the same cache line and each variable is just modified by one thread) and reducing true sharing. Don't worry if you don't understand any of this now. We will cover it later and demonstrate how to run performance tests.

This is a basic introduction to atomic operations to synchronize memory access from different threads. In some cases, the use of atomic operations is quite easy, similar to gathering statistics and simple counters. More involved applications, such as the implementation of an SPSC lock-free queue, require a deeper knowledge of atomic operations. The material we have seen in this chapter helps build an understanding of the basics and builds a foundation for further study of this complex subject.

In the next chapter, we will look at promises and futures, two fundamental building blocks of asynchronous programming in C++.

Further reading

- [Butenhof, 1997] David R. Butenhof, Programming with POSIX Threads, Addison Wesley, 1997.
- [Williams, 2019] Anthony Williams, C++ Concurrency in Action, Second Edition, Manning, 2019.
- Memory Model: Get Your Shared Data Under Control, Jana Machutová, `https://www.youtube.com/watch?v=L5RCGDAan2Y`.
- *C++ Atomics: From Basic To Advanced*, Fedor Pikus, `https://www.youtube.com/watch?v=ZQFzMfHIxng`.
- *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1*, Intel Corporation, `https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.pdf`.

Part 3: Asynchronous Programming with Promises, Futures, and Coroutines

In this part, we shift our focus to the core subject of this book, asynchronous programming, a critical aspect of building responsive, high-performance applications. We will learn how to execute tasks concurrently without blocking the main execution flow by utilizing tools such as promises, futures, packaged tasks, the `std::async` function, and coroutines, a revolutionary feature enabling asynchronous programming without the overhead of creating threads. We will also cover advanced techniques for sharing futures and examine real-world scenarios where these concepts are essential. These powerful mechanisms allow us to develop efficient, scalable, and maintainable asynchronous software needed for modern software systems.

This part has the following chapters:

- *Chapter 6*, *Promises and Futures*
- *Chapter 7*, *The Async Function*
- *Chapter 8*, *Asynchronous Programming Using Coroutines*

6
Promises and Futures

In previous chapters, we learned the foundations of managing and synchronizing thread execution using C++. We also mentioned in *Chapter 3* that to return values from a thread, we could use promises and futures. Now it's time to learn how to do that and much more using these features in C++.

Futures and **promises** are essential blocks for achieving asynchronous programming. They define a way to manage the result of a task that will be completed in the future, usually in a separate thread.

In this chapter, we're going to cover the following main topics:

- What are promises and futures?
- What are shared futures and how are they different from regular futures?
- What are packaged tasks and when do we use them?
- How do we check future statuses and errors?
- What are the benefits and drawbacks of using promises and futures?
- Examples of real-life scenarios and solutions

So, let's get started!

Technical requirements

Promises and futures have been available since C++11, but some examples implemented in this chapter use features from C++20, such as `std::jthread`, so the code shown in this chapter can be compiled by compilers supporting C++20.

Please check the *Technical requirements* section in *Chapter 3* for guidance on how to install GCC 13 and Clang 8 compilers.

You can find all the complete code in the following GitHub repository:

`https://github.com/PacktPublishing/Asynchronous-Programming-with-CPP`

The examples for this chapter are located under the `Chapter_06` folder. All source code files can be compiled using CMake as follows:

```
cmake . && cmake build .
```

Executable binaries will be generated under the `bin` directory.

Exploring promises and futures

A **future** is an object that represents some undetermined result that will be completed sometime in the future. A **promise** is the provider of that result.

Promises and futures have been part of the C++ standard since version C++11 and are available by including the `<future>` header file, promises via the class `std::promise` and futures via the class `std::future`.

The `std::promise` and `std::future` pair implements a one-shot producer-consumer channel with the promise as the producer and the future as the consumer. The consumer (`std::future`) can block until the result of the producer (`std::promise`) is available.

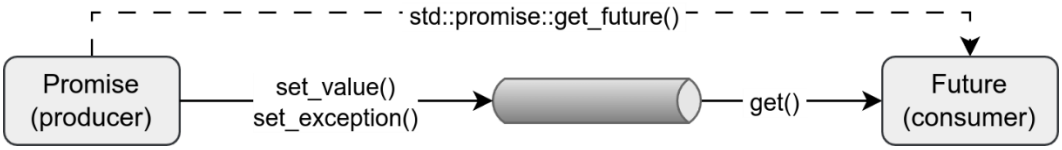


Figure 6.1 – Promise-future communication channel

Many modern programming languages provide similar asynchronous approaches, such as Python (with the `asyncio` library), Scala (in the `scala.concurrent` library), JavaScript (in its core library), Rust (in its **Standard Library** (**std**) or crates such as `promising_future`), Swift (in the Combine framework), and Kotlin, among others.

The basic principle behind achieving asynchronous execution using promises and futures is that a function we want to run to generate a result is executed in the background, using a new thread or the current one, and a future object is used by the initial thread to retrieve the result computed by the function. This result value will be stored when the function finishes, so meanwhile, the future will be used as a placeholder. The asynchronous function will use a promise object to store the result in the future with no need for explicit synchronization mechanisms between the initial thread and the background one. When the value is needed by the initial thread, it will be retrieved from the future object. If the value is still not ready, the initial thread execution will be blocked until the future becomes ready.

With this idea, making a function run asynchronously becomes easy. We only need to be aware that the function can run on a separate thread, so we need to avoid data races, but result communication and synchronization between threads is managed by the promise-future pair.

Using promises and futures improves responsiveness by offloading computations and provides a structured approach to handling asynchronous operations compared to threads and callbacks, as we will explore in this chapter.

Let's now learn about these two objects.

Promises

Promises are defined in the `<future>` header as `std::promise`.

With a promise, we get an agreement that the result will be available at some time in the future. This way, we can let the background task do its work and compute the result. Meanwhile the main thread will also proceed with its task and, when the result is needed, request it. At that time, the result might already be ready.

Also, promises can communicate if an exception was raised instead of returning a valid value and, they will make sure that its lifetime persists until the thread finishes and writes the result to it.

Therefore, a promise is a facility to store a result (a value or an exception) that is later acquired asynchronously via a future. A promise object is only intended to be used once and cannot be modified afterward.

Apart from a result, each promise also holds a shared state. The shared state is a memory area that stores the completion status, synchronization mechanisms, and a pointer to the result. It ensures proper communication and synchronization between a promise and a future by enabling the promise to store either a result or an exception, signal when it's complete, and allowing the future to access the result, blocking if the promise is not yet ready. The promise can update its shared state using the following operations:

- **Make ready**: The promise stores the result in the shared state and makes the state of the promise to become ready unblocking any thread waiting on a future associated with the promise. Remember that the result can be a value (or even void) or an exception.
- **Release**: The promise releases its reference to the shared state, which will be destroyed if this is the last reference. This memory release mechanism is like the one used by shared pointers and their control blocks. This operation does not block unless the shared state was created by `std::async` and is not yet in the ready status (we will learn about this in the next chapter).
- **Abandon**: The promise stores an exception of type `std::future_error` with error code `std::future_errc::broken_promise`, making the shared state ready and then releasing it.

A `std::promise` object can be constructed using its default constructor or using a custom allocator. In both cases, a new promise will be created with an empty shared state. Promises can also be constructed using the move constructor; thus, the new promise will have the shared state owned by the other promise. The initial promise will remain with no shared state.

Moving a promise is useful in scenarios related to resource management, optimization by avoiding extra copies, and keeping correct ownership semantics; for example, it's useful when a promise needs to be completed in another thread, stored in a container, returned to the caller of an API call, or sent to a callback handler.

Promises cannot be copied (their copy-constructor or copy-assignment operator is deleted), avoiding two promise objects sharing the same shared state and in risk of data races when results are stored in the shared state.

As promises can be moved, they can also be swapped. The `std::swap` function from the **Standard Template Library** (**STL**) has a template specialization for promises.

When a promise object is deleted, the associated future will still have access to the shared state. If deletion happens after the promise sets the value, the shared state will be in release mode, thus the future can access the result and use it. However, if the promise was deleted before setting the result value, the shared state will be moved to abandoned, and the future will obtain `std::future_errc::broken_promise` when trying to get the result.

A value can be set by using the `std::promise` function `set_value()` and an exception by using the `set_exception()` function. The result is stored atomically in the promise's shared state, making its state ready. Let's see an example:

```
auto threadFunc = [](std::promise<int> prom) {
    try {
        int result = func();
        prom.set_value(result);
    } catch (...) {
        prom.set_exception(std::current_exception());
    }
};

std::promise<int> prom;
std::jthread t(threadFunc, std::move(prom));
```

In the previous example, the `prom` promise is created and moved into the `threadFunc` lambda function as a parameter. As the promise is non-copyable, we need to use pass-by-value together with moving the promise into the parameter to avoid copies.

Inside the lambda function, the `func()` function is called, and its result is stored in the promise using `set_value()`. If `func()` throws an exception, it's captured and stored into the promise using `set_exception()`. As we will learn later, this result (value or exception) can be extracted in the calling thread by using a future.

In C++14, we can also use generalized lambda capture to pass the promise into the lambda capture:

```
using namespace std::literals;
std::promise<std::string> prom;
auto t = std::jthread([prm = std::move(prom)] mutable {
    std::this_thread::sleep_for(100ms);
    prm.set_value("Value successfully set");
});
```

Therefore, `prm = std::move(prom)` is moving the external promise, `prom`, into the lambda's internal promise, `prm`. By default, parameters are captured as constants, so we need to specify the lambda as mutable to allow `prm` to be modified.

`set_value()` can throw a `std::future_error` exception if the promise has no shared state (error code set to `no_state`) or the shared state has already a stored result (error code set to `promise_already_satisfied`).

`set_value()` can also be used without specifying a value. In that case, it simply makes the state ready. That can be used as a barrier, as we will see later in this chapter after introducing futures.

Figure 6.2 shows a diagram representing the different shared state transitions.

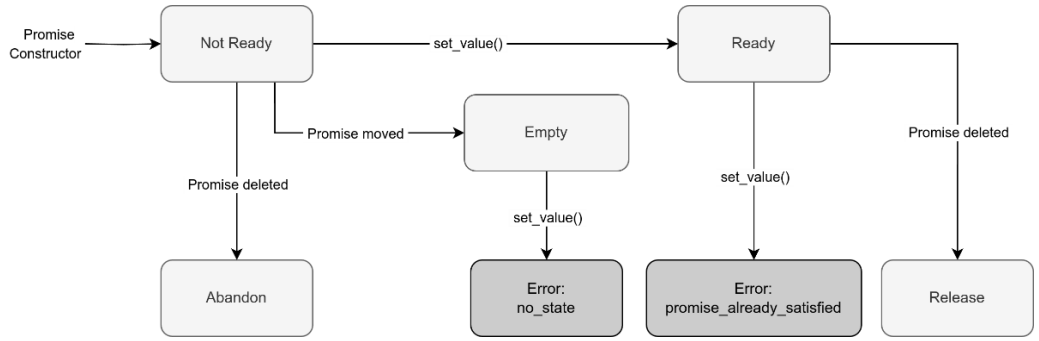


Figure 6.2 – Promise shared state transitions diagram

There are two other functions to set the value of a promise, `set_value_at_thread_exit` and `set_exception_at_thread_exit`. As before, the result is stored immediately, but using these new functions, the state is not made ready yet. The state becomes ready when the thread exits after all thread-local variables have been destroyed. This is useful when we want threads to manage resources that need to be cleaned up before exiting, even if an exception happens, or if we want to provide accurate log activity or monitor when threads exit.

In terms of exceptions thrown or synchronization mechanisms to avoid data races, both functions behave as `set_value()` and `set_exception()`.

Now that we know how to store a result in a promise, let's learn about the other member of the duo, the future.

Futures

Futures are defined in the `<future>` header file as `std::future`.

As we saw earlier, a future is the consumer side of the communication channel. It provides access to the result stored by the promise.

A `std::future` object must be created from a `std::promise` object by calling `get_future()`, or through a `std::packaged_task` object (more details later in this chapter) or a call to the `std::async` function (in *Chapter 7*):

```
std::promise<int> prom;
std::future<int> fut = prom.get_future();
```

Like promises, futures can be moved but not copied for the same reasons. To reference the same shared state from multiple futures, we need to use shared futures (explained in the next section, *Shared futures*).

The `get()` method can be used to retrieve the result. If the shared state is still not ready, this call will block by internally calling `wait()`. When the shared state becomes ready, the result value is returned. If an exception was stored in the shared state, that exception will be rethrown:

```
try {
    int result = fut.get();
    std::cout << "Result from thread: " << result << '\n';
} catch (const std::exception& e) {
    std::cerr << "Exception: " << e.what() << '\n';
}
```

In the preceding example, the result is retrieved from the `fut` future by using the `get()` function. If the result is a value, it will be printed with a line starting with `Result from thread`. On the other hand, if an exception were thrown and stored into the promise, it would be rethrown and captured in the caller thread, and a line starting with `Exception` would be printed out.

After calling the `get()` method, `valid()` will return `false`. If for some reason `get()` is called when `valid()` is `false`, the behavior is undefined, but the C++ standard recommends that a `std::future_error` exception is thrown with the `std::future_errc::no_state` error code. Futures in which the `valid()` function returns `false` can still be moved.

When a future is destroyed, it releases its shared state reference. If that were the last reference, the shared state would be destroyed. These actions will not block unless in a specific case when using `std::async`, which we will learn about in *Chapter 7*.

Future errors and error codes

As we have seen in the preceding examples, some functions that deal with asynchronous execution and shared states can throw `std::future_error` exceptions.

This exception class is inherited from `std::logic_error`, which in turn is inherited from `std::exception`, defined in the `<stdexcept>` and `<exception>` header files, respectively.

As with any other exception defined in the STL, the error code can be checked by using its `code()` function or an explanatory string by using its `what()` function.

Error codes reported by futures are defined by `std::future_errorc`, a scoped enumeration (`enum` class). The C++ standard defines the following error codes, but the implementation may define additional ones:

- `broken_promise`: Reported when a promise is deleted before setting the result, so the shared state is released before being valid.
- `future_already_retrieved`: Occurring when `std::promise::get_future()` is called more than once.
- `promise_already_satisfied`: Reported by `std::promise:: set_value()` if the shared state already has a stored result.
- `no_state`: Reported when some methods are used but there is no shared state as the promise was created by using the default constructor or moved from. As we will see later in this chapter, this happens when calling some packaged tasks (`std::packaged_task`) methods, such as `get_future()`, `make_ready_at_thread_exit()`, or `reset()`, when their shared state has not been created, or when using `std::future::get()` with a not-yet-ready future (`std::future::valid()` returns `false`).

Waiting for results

`std::future` also provides functions to block the thread and wait for a result to be available. These functions are `wait()`, `wait_for()`, and `wait_until()`. The `wait()` function will block indefinitely until the result is ready, `wait_for()` for a period, and `wait_until()` until a specific time has been reached. All will return as soon as the result is available within those waiting periods.

These functions must be called only when `valid()` is `true`. Otherwise, the behavior is undefined but encouraged by the C++ standard to throw a `std::future_error` exception with the `std::future_errc::no_state` error code.

As commented previously, using `std::promise::set_value()` without specifying a value sets the shared state as ready. This together with `std::future::wait()` can be used to implement a barrier and stop a thread from progressing until it is signaled. The following example shows this mechanism in action.

Let's start by adding the required header files:

```
#include <algorithm>
#include <cctype>
#include <chrono>
#include <future>
#include <iostream>
#include <iterator>
#include <sstream>
#include <thread>
#include <vector>
#include <set>
using namespace std::chrono_literals;
```

Inside the `main()` function, the program will start by creating two promises, `numbers_promise` and `letters_promise`, and their corresponding futures, `numbers_ready` and `letters_ready`:

```
std::promise<void> numbers_promise, letters_promise;
auto numbers_ready = numbers_promise.get_future();
auto letter_ready = letters_promise.get_future();
```

Then, the `input_data_thread` emulates two I/O thread operations running in sequence, one copying numbers into a vector and another inserting letters into a set:

```
std::istringstream iss_numbers{"10 5 2 6 4 1 3 9 7 8"};
std::istringstream iss_letters{"A b 53 C,d 83D 4B ca"};
std::vector<int> numbers;
std::set<char> letters;

std::jthread input_data_thread([&] {
    // Step 1: Emulating I/O operations.
    std::copy(std::istream_iterator<int>{iss_numbers},
              std::istream_iterator<int>{},
              std::back_inserter(numbers));

    // Notify completion of Step 1.
    numbers_promise.set_value();

    // Step 2: Emulating further I/O operations.
    std::copy_if(std::istreambuf_iterator<char>
```

```
                                {iss_letters},
                    std::istreambuf_iterator<char>{},
                    std::inserter(letters,
                                letters.end()),
                                ::isalpha);

    // Notify completion of Step 2.
    letters_promise.set_value();
});

// Wait for numbers vector to be filled.
numbers_ready.wait();
```

While this is happening, the main thread stops its execution by using `numbers_ready.wait()`, waiting for the counterpart promise, `numbers_promise`, to be ready. Once all numbers are read, `input_data_thread` will call `numbers_promise.set_value()`, waking up the main thread and continuing its execution.

Numbers will then be sorted and printed if letters have not already been read by using the `wait_for()` function from the `letters_ready` future and checking whether it timed out:

```
std::sort(numbers.begin(), numbers.end());

if (letter_ready.wait_for(1s) == std::future_status::timeout) {
    for (int num : numbers) std::cout << num << ' ';
    numbers.clear();
}
// Wait for letters vector to be filled.
letter_ready.wait();
```

This section of the code shows how the main thread can do some work. Meanwhile, `input_data_thread` continues processing incoming data. Then, the main thread will wait again by calling `letters_ready.wait()`.

Finally, when all letters are added to the set, the main thread will wake up by being signaled again by using `letters_promise.set_value()`, and numbers (if not yet printed) and letters will be printed in order:

```
for (int num : numbers) std::cout << num << ' ';
std::cout << std::endl;
for (char let : letters) std::cout << let << ' ';
std::cout << std::endl;
```

As we have seen in the previous example, a future status object is returned by the wait functions. Next, let's learn what these objects are.

Future status

`wait_for()` and `wait_until()` return a `std::future_status` object.

A future can be in any of the following statuses:

- **Ready**: The shared state is **ready**, indicating that the result can be retrieved.
- **Deferred**: The shared state contains a **deferred** function, meaning that the result will only be computed when explicitly requested. We will learn more about deferred functions in the next chapter when introducing `std::async`.
- **Timeout**: The specified **timeout period** passed before the shared state could become ready.

Next, we will learn how to share a promise result among multiple futures by using shared futures.

Shared futures

As we saw earlier, `std::future` is only moveable, thus only one future object can refer to a particular asynchronous result. On the other hand, `std::shared_future` is copyable, so several shared future objects can refer to the same shared state.

Therefore, `std::shared_future` allows thread-safe access from different threads to the same shared state. Shared futures can be useful for sharing the result of a computationally intensive task among multiple consumers or interested parties, reducing redundant computation. Also, they can be used to notify events or as a synchronization mechanism where multiple threads must wait for the completion of a single task. Later in this chapter, we will learn how to chain asynchronous operations by using shared futures.

The interface of `std:;shared_object` is the same as the one for `std::future`, so everything explained about waiting and getter functions applies here.

A shared object can be created by using the `std::future::share()` function:

```
std::shared_future<int> shared_fut = fut.share();
```

That invalidates the original future (its `valid()` function will return `false`).

The following example shows how to send the same result to many threads at the same time:

```
#define sync_cout std::osyncstream(std::cout)

int main() {
    std::promise<int> prom;
    std::future<int> fut = prom.get_future();
    std::shared_future<int> shared_fut = fut.share();
```

```
    std::vector<std::jthread> threads;
    for (int i = 1; i <= 5; ++i) {
        threads.emplace_back([shared_fut, i]() {
            sync_cout << "Thread " << i << ": Result = "
                      << shared_fut.get() << std::endl;
        });
    }
    prom.set_value(5);
    return 0;
}
```

We start by creating a promise, `prom`, getting the future, `fut`, from it, and finally, getting a shared future, `shared_fut`, by calling `share()`.

Then, five threads are created and added to a vector, each one having a shared future instance and an index. All these threads will be waiting for the promise, `prom`, to be ready by calling `shared_future.get()`. When a value is set in the promise shared state, the value will be accessible by all the threads. The output of running the previous program is the following:

```
Thread 5: Result = 5
Thread 3: Result = 5
Thread 4: Result = 5
Thread 2: Result = 5
Thread 1: Result = 5
```

Therefore, shared futures can also be used to signal multiple threads at the same time.

Packaged tasks

A **packaged task**, or `std::packaged_task`, also defined in the `<future>` header file, is a class template that wraps a callable object to be invoked asynchronously. Its result is stored in a shared state, which is accessible through a `std::future` object. To create a `std::packaged_task` object, we need to define as the template parameter the function signature that represents the task that will be called and pass the desired function as its constructor argument. Here are some examples:

```
// Using a thread.
std::packaged_task<int(int, int)> task1(
                      std::pow<int, int>);
std::jthread t(std::move(task1), 2, 10);

// Using a lambda function.
std::packaged_task<int(int, int)> task2([](int a, int b)
{
    return std::pow(a, b);
```

```
});
task2(2, 10);

// Binding to a function.
std::packaged_task<int()> task3(std::bind(std::pow<int, int>, 2, 10));
task3();
```

In the preceding example, `task1` is created by using a function and executed by using a thread. On the other hand, `task2` is created by using a lambda function and executed by invoking its method `operator()`. Finally, `task3` is created by using a forwarding call wrapper by using `std::bind`.

To retrieve the future associated with a task, just call `get_future()` from its `packaged_task` object:

```
std::future<int> result = task1.get_future();
```

As with promises and futures, packaged tasks can be constructed with no shared state by using the default constructor, the move constructor, or allocators. Packaged tasks are thus move-only and non-copyable. Also, assignment operators and the swap function have similar behaviors as promises and futures.

The destructor of packaged tasks behaves like the promise destructors; if the shared state is released before being valid, a `std::future_error` exception will be thrown with the `std::future_errc::broken_promise` error code. As with futures, packaged tasks define a `valid()` that function returns `true` if a `std::packaged_task` object has a shared state.

As with promises, `get_future()` can only be called once. If this function is called more than once, a `std::future_error` exception with the `future_already_retrieved` code will be thrown. If the packaged tasks were created from the default constructor, thus with no shared state, the error code will be `no_state`.

As seen in the previous example, the stored callable object can be invoked by using `operator()`:

```
task1(2, 10);
```

Sometimes, it's interesting to make the result ready only when the thread that runs the packaged task exits and all its **thread-local** objects are destroyed. This is achieved by using the `make_ready_at_thread_exit()` function. Even if the result is not ready until the thread exits, it is computed right away as usual. Therefore, its computation is not deferred.

As an example, let's define the following function:

```
void task_func(std::future<void>& output) {
    std::packaged_task<void(bool&)> task{[](bool& done){
        done = true;
    }};
    auto result = task.get_future();
```

```
    bool done = false;
    task.make_ready_at_thread_exit(done);

    std::cout << "task_func: done = "
              << std::boolalpha << done << std::endl;

    auto status = result.wait_for(0s);
    if (status == std::future_status::timeout)
        std::cout << "task_func: result not ready\n";

    output = std::move(result);
}
```

This function creates a packaged task called `task` that sets its Boolean argument to `true`. A future called `result` is also created from this task. When the task is executed by calling `make_ready_at_thread_exit()`, its done argument is set to `true` but the future result is still not marked as ready. When the `task_func` function exits, the `result` future is moved to the passed-by reference. At this point, the thread exits, and the `result` future will be set as ready.

Therefore, say we call this task from the main thread by using the following code:

```
std::future<void> result;

std::thread t{task_func, std::ref(result)};
t.join();

auto status = result.wait_for(0s);
if (status == std::future_status::ready)
    std::cout << «main: result ready\n»;
```

The program will show the following output:

```
task_func: done = true
task_func: result not ready
main: result ready
```

`make_ready_at_thread_exit()` will throw `std::future_error` exceptions if there is no shared state (the `no_state` error code) or the task has already been invoked (the `promise_already_satisfied` error code).

A packaged task state can also be reset by calling `reset()`. This function will abandon the current state and construct a new shared state. Obviously, if there is no state when calling `reset()`, an exception with the `no_state` error code will be thrown. After resetting, a new future must be acquired by calling `get_future()`.

The following example prints the first 10 power-of-two numbers. Each number is computed by calling the same `packaged_task` object. In each loop iteration, `packaged_task` is reset, and a new future object is retrieved:

```
std::packaged_task<int(int, int)> task([](int a, int b){
    return std::pow(a, b);
});

for (int i=1; i<=10; ++i) {
    std::future<int> result = task.get_future();
    task(2, i);
    std::cout << "2^" << i << " = "
              << result.get() << std::endl;
    task.reset();
}
```

This is the output when executing the preceding code:

```
2^1 = 2
2^2 = 4
2^3 = 8
2^4 = 16
2^5 = 32
2^6 = 64
2^7 = 128
2^8 = 256
2^9 = 512
2^10 = 1024
```

As we will learn in the next chapter, `std::async` provides a simpler way to achieve the same result. The only advantage of `std::packaged_task` is the ability to specify in exactly which thread the task will run.

Now that we know how to use promises, futures, and packaged tasks, it's time to understand not only the upsides of this approach but also what downsides can arise.

The benefits and drawbacks of promises and futures

There are advantages as well as some disadvantages of using promises and futures. Here are the main points.

Benefits

As high-level abstractions for managing asynchronous operations, writing and reasoning about concurrent code by using promises and futures is simplified and less error-prone.

Futures and promises enable concurrent execution of tasks, allowing the program to use multiple CPU cores efficiently. This can lead to improved performance and reduced execution time for computationally intensive tasks.

Also, they facilitate asynchronous programming by decoupling the initiation of an operation from its completion. As we will see later, this is particularly useful for I/O-bound tasks, such as network requests or file operations, where the program can continue executing other tasks while waiting for the asynchronous operation to complete. As a result, they can return a value but also an exception, allowing exception propagation from the asynchronous tasks to the caller code section that waits for their completion, which allows for a cleaner way for error handling and recovery.

As we have also mentioned, they also provide a mechanism for synchronizing the completion of tasks and retrieving their results. This helps in coordinating parallel tasks and managing dependencies between them.

Drawbacks

Unfortunately, not everything is positive news; there are also some areas that are impacted.

For example, asynchronous programming with futures and promises can introduce complexity when dealing with dependencies between tasks or managing the life cycle of asynchronous operations. Also, potential deadlocks can happen if there are circular dependencies.

Likewise, using futures and promises may introduce some performance overhead due to the synchronization mechanisms happening under the hood, involved in coordinating asynchronous tasks and managing shared states.

As with other concurrent or asynchronous solutions, debugging code that uses futures and promises can be more challenging compared to synchronous code, as the flow of execution may be non-linear and involve multiple threads.

Now it's time to tackle real-life scenarios by implementing some examples.

Examples of real-life scenarios and solutions

Now that we've learned about some new building blocks of creating asynchronous programs, let's build solutions for some real-life scenarios. In this section, we will learn how to do the following:

- Cancel asynchronous operation
- Return combined results

- Chain asynchronous operations and create a pipeline
- Create a thread-safe **single-producer-single-consumer** (**SPSC**) task queue

Canceling asynchronous operations

As we saw earlier, futures offer the ability to check for completion or timeout before waiting for the result. That can be done by checking the `std::future_status` object returned by the `std::future`, `wait_for()`, or `wait_until()` function.

By combining futures with mechanisms such as cancellation flags (by means of `std::atomic_bool`) or timeouts, we can gracefully terminate long-running tasks if necessary. Timeout cancellation can be implemented by simply using the `wait_for()` and `wait_until()` functions.

Canceling a task by using a cancellation flag or token can be implemented by passing a reference to a cancellation flag defined as `std::atomic_bool`, where the caller thread sets its value to `true` to request the task cancellation, and the worker thread periodically checks this flag and whether it's set. If it is set, it exits gracefully and performs any cleanup work to be done.

Let's first define a long-running task function:

```
const int CHECK_PERIOD_MS = 100;

bool long_running_task(int ms,
            const std::atomic_bool& cancellation_token) {
    while (ms > 0 && !cancellation_token) {
        ms -= CHECK_PERIOD_MS;
        std::this_thrsead::sleep_for(100ms);
    }
    return cancellation_token;
}
```

The `long_running_task` function accepts as arguments a period in milliseconds (`ms`) to run the task and a reference to an atomic Boolean value (`cancellation_token`) representing the cancellation token. The function will periodically check whether the cancellation token is set to `true`. When the running period passes or the cancellation token is set to `true`, the thread will exit.

In the main thread, we can use two `packaged task objects` to execute this function, that is, `task1` lasting for 500 ms and running in thread `t1`, and `task2` running for one second in thread `t2`. Both share the same cancellation token:

```
std::atomic_bool cancellation_token{false};
std::cout << "Starting long running tasks...\n";

std::packaged_task<bool(int, const std::atomic_bool&)>
               task1(long_running_task);
```

```
std::future<bool> result1 = task1.get_future();
std::jthread t1(std::move(task1), 500,
                std::ref(cancellation_token));

std::packaged_task<bool(int, const std::atomic_bool&)>
                task2(long_running_task);
std::future<bool> result2 = task2.get_future();
std::jthread t2(std::move(task2), 1000,
                std::ref(cancellation_token));

std::cout << "Cancelling tasks after 600 ms...\n";
this_thread::sleep_for(600ms);
cancellation_token = true;

std::cout << "Task1, waiting for 500 ms. Cancelled = "
          << std::boolalpha << result1.get() << "\n";
std::cout << "Task2, waiting for 1 second. Cancelled = "
          << std::boolalpha << result2.get() << "\n";
```

After both tasks have started, the main thread sleeps for 600 ms. When it wakes up, it sets the cancellation token to `true`. At that time, `task1` was already finished, but `task2` was still running. Thus, `task2` is being cancelled.

This explanation aligns with the obtained output:

```
Starting long running tasks...
Cancelling tasks after 600 ms...
Task1, waiting for 500 ms. Cancelled = false
Task2, waiting for 1 second. Cancelled = true
```

Next, let's see how to combine several asynchronous computation results into a single future.

Returning combined results

Another common approach in asynchronous programming is to decompose complex tasks into smaller independent subtasks using multiple promises and futures. Each subtask can be launched in a separate thread and its result stored in a corresponding promise. The main thread can then use the futures to wait for all subtasks to finish and combine their results to obtain the outcome.

This approach is useful to achieve parallel processing of multiple independent tasks, allowing the efficient utilization of multiple cores for faster computations.

Let's see an example of a task that emulates a value computation and an I/O operation. We want that task to return a tuple with both results, the computed value as an `int` value, and the information

read from the file as a `string` object. So, we define the `combineFunc` function that accepts as an argument a `combineProm` promise holding a tuple with the result types.

This function will create two threads, `computeThread` and `fetchData`, with their respective promises, `computeProm` and `fetchProm`, and futures, `computeFut` and `fetchFut`:

```
void combineFunc(std::promise<std::tuple<int,
                 std::string>> combineProm) {
    try {
        // Thread to simulate computing a value.
        std::cout << "Starting computeThread...\n";
        auto computeVal = [](std::promise<int> prom)
                          mutable {
            std::this_thread::sleep_for(1s);
            prom.set_value(42);
        };
        std::promise<int> computeProm;
        auto computeFut = computeProm.get_future();
        std::jthread computeThread(computeVal,
                           std::move(computeProm));

        // Thread to simulate downloading a file.
        std::cout << "Starting dataThread...\n";
        auto fetchData = [](
                 std::promise<std::string> prom) mutable {
            std::this_thread::sleep_for(2s);
            prom.set_value("data.txt");
        };
        std::promise<std::string> fetchProm;
        auto fetchFut = fetchProm.get_future();
        std::jthread dataThread(fetchData,
                             std::move(fetchProm));

        combineProm.set_value({
                    computeFut.get(),
                    fetchFut.get()
        });
    } catch (...) {
        combineProm.set_exception(
                    std::current_exception());
    }
}
```

As we can see, both threads will execute asynchronously and independently, generating a result and storing it in their respective promises.

The combined promise is set by calling the `get()` function on each future and combining their result into a tuple that is used to set the value of the combined promise by calling its `set_value()` function, as follows:

```
combineProm.set_value({computeFut.get(), fetchFut.get()});
```

The `combineFunc` task can be called as usual by using a thread and setting up the `combineProm` promise and its `combineFut` future. Calling the `get()` function on this future will return a tuple:

```
std::promise<std::tuple<int, std::string>> combineProm;
auto combineFuture = combineProm.get_future();
std::jthread combineThread(combineFunc,
                           std::move(combineProm));

auto [data, file] = combineFuture.get();
std::cout << "Value [ " << data
          << " ]  File [ « << file << « ]\n»;
```

Running this example will show the following result:

```
Creating combined promise...
Starting computeThread...
Starting dataThread...
Value [ 42 ]  File [ data.txt ]
```

Now, let's continue by learning how to create a pipeline using promises and futures.

Chaining asynchronous operations

Promises and futures can be chained together to perform multiple asynchronous operations sequentially. We can create a pipeline where one future's result becomes the input for the next operation's promise. This allows for composing complex asynchronous workflows where the output of one task feeds into the next.

Also, we can allow branching in the pipeline and keep some tasks switched off until needed. This can be done by using futures with deferred execution, which is useful in scenarios where the computation cost is high, but the result may not always be needed. Thus, we can use futures to initiate the computation asynchronously and retrieve the result only when required. As futures with deferred status can only be created by using `std::async`, we will leave that for the next chapter.

In this section, we will focus on creating the following task graph:

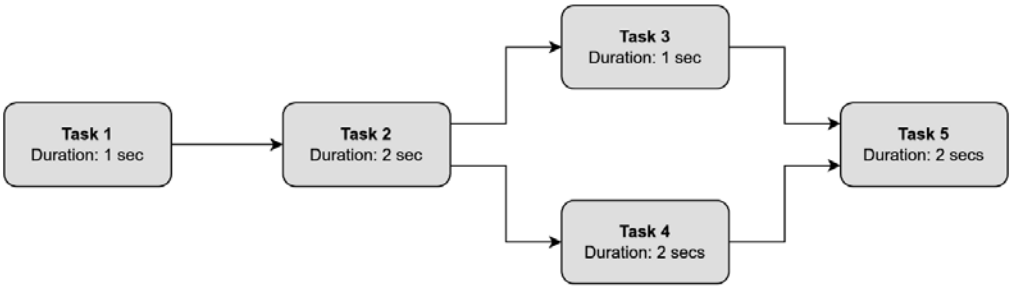


Figure 6.3 – A pipeline example

We start by defining a template class called `Task` that accepts a callable as a template argument, defining the function to execute. This class will also allow us to create tasks that share a future with dependent ones. These will use the shared futures to wait for the predecessor tasks to signal them about their completion by calling `set_value()` in the associated promise before running their own task:

```
#define sync_cout std::osyncstream(std::cout)

template <typename Func>
class Task {
   public:
    Task(int id, Func& func)
        : id_(id), func_(func), has_dependency_(false) {
        sync_cout << "Task " << id
                  << " constructed without dependencies.\n";
        fut_ = prom_.get_future().share();
    }

    template <typename... Futures>
    Task(int id, Func& func, Futures&&... futures)
        : id_(id), func_(func), has_dependency_(true) {
        sync_cout << "Task " << id
                  << « constructed with dependencies.\n";
        fut_ = prom_.get_future().share();
        add_dependencies(
                  std::forward<Futures>(futures)...);
    }

    std::shared_future<void> get_dependency() {
        return fut_;
    }
```

```
    void operator()() {
        sync_cout << "Running task " << id_ << '\n';
        wait_completion();
        func_();
        sync_cout << "Signaling completion of task "
                  << id_ << ‹\n';
        prom_.set_value();
    }

   private:
    template <typename... Futures>
    void add_dependencies(Futures&&... futures) {
        (deps_.push_back(futures), ...);
    }

    void wait_completion() {
        sync_cout << "Waiting completion for task "
                  << id_ << ‹\n';
        if (!deps_.empty()) {
            for (auto& fut : deps_) {
                if (fut.valid()) {
                    sync_cout << "Fut valid so getting "
                              << "value in task "
                              << id_ << ‹\n';
                    fut.get();
                }
            }
        }
    }

   private:
    int id_;
    Func& func_;
    std::promise<void> prom_;
    std::shared_future<void> fut_;
    std::vector<std::shared_future<void>> deps_;
    bool has_dependency_;
};
```

Let's describe step by step how this `Task` class is implemented.

There are two constructors: one to initialize an object of type `Task` that has no dependencies with other tasks and another templated constructor to initialize a task with a variable number of dependent tasks. Both initialize an identifier (`id_`), the function to call to perform the task (`func_`), a Boolean

variable indicating whether the task has dependencies or not (`has_dependency_`), and a shared future, `fut_`, to share with tasks that will depend on this one. This `fut_` future is retrieved from the `prom_` promise used to signal the task completion. The templated constructor also calls the `add_dependencies` function forwarding the futures passed as arguments, which will be stored in the `deps_` vector.

The `get_dependency()` function just returns the shared future used by dependent tasks to wait for the completion of the current task.

Finally, `operator()` waits for the completion of previous tasks by calling `wait_completion()`, which checks whether each shared future stored in the `deps_` vector is valid and waiting until the result is ready by calling `get()`. Once all shared futures are ready, meaning that all previous tasks are complete, the `func_` function is invoked, running the task, and the `prom_` promise is then set to ready by calling `set_value()`, triggering the dependent tasks.

In the main thread, we define the pipeline as follows, creating a graph like the one shown in *Figure 6.3*:

```
auto sleep1s = []() { std::this_thread::sleep_for(1s); };
auto sleep2s = []() { std::this_thread::sleep_for(2s); };

Task task1(1, sleep1s);
Task task2(2, sleep2s, task1.get_dependency());
Task task3(3, sleep1s, task2.get_dependency());
Task task4(4, sleep2s, task2.get_dependency());
Task task5(5, sleep2s, task3.get_dependency(),
                       task4.get_dependency());
```

Then, we need to start the pipeline by triggering all tasks and calling their `operator()`. As `task1` has no dependencies, it will start running straight away. All other tasks will be waiting for their predecessor tasks to complete their work:

```
sync_cout << "Starting the pipeline..." << std::endl;
task1();
task2();
task3();
task4();
task5();
```

Finally, we need to wait for the pipeline to finish the execution of all tasks. We can achieve that by simply waiting for the shared future returned by the last task, `task5`, to be ready:

```
sync_cout << "Waiting for the pipeline to finish...\n";
auto finish_pipeline_fut = task5.get_dependency();
finish_pipeline_fut.get();
sync_cout << "All done!" << std::endl;
```

Here is the output of running this example:

```
Task 1 constructed without dependencies.
Getting future from task 1
Task 2 constructed with dependencies.
Getting future from task 2
Task 3 constructed with dependencies.
Getting future from task 2
Task 4 constructed with dependencies.
Getting future from task 4
Getting future from task 3
Task 5 constructed with dependencies.
Starting the pipeline...
Running task 1
Waiting completion for task 1
Signaling completion of task 1
Running task 2
Waiting completion for task 2
Fut valid so getting value in task 2
Signaling completion of task 2
Running task 3
Waiting completion for task 3
Fut valid so getting value in task 3
Signaling completion of task 3
Running task 4
Waiting completion for task 4
Fut valid so getting value in task 4
Signaling completion of task 4
Running task 5
Waiting completion for task 5
Fut valid so getting value in task 5
Fut valid so getting value in task 5
Signaling completion of task 5
Waiting for the pipeline to finish...
Getting future from task 5
All done!
```

There are some issues that can occur with this approach that we must be aware of. First, the dependency graph must be a **directed acyclic graph** (**DAG**), therefore without cycles or loops between dependent tasks. Otherwise, a deadlock will occur, as a task might be waiting for a task happening in the future and not yet started. Also, we need enough threads to run all the tasks concurrently or launch the tasks orderly; otherwise, it can also lead to a deadlock with threads waiting on the completion of tasks that have still not launched.

One common use case of this approach can be found in **MapReduce** algorithms where large datasets are processed in parallel across multiple nodes, and futures and threads can be used to execute map and reduce tasks concurrently, allowing efficient distributed data processing.

Thread-safe SPSC task queue

In this last example, we will show how to use promises and futures to create an SPSC queue.

The producer thread creates a promise for each item it wants to add to the queue. The consumer thread waits for a future obtained from an empty queue slot. Once the producer finishes adding an item, it sets the value on the corresponding promise, notifying the waiting consumer. This allows for efficient data exchange between threads while maintaining thread safety.

Let's first define the thread-safe queue class:

```
template <typename T>
class ThreadSafeQueue {
   public:
    void push(T value) {
        std::lock_guard<std::mutex> lock(mutex_);
        queue_.push(std::move(value));
        cond_var_.notify_one();
    }

    T pop() {
        std::unique_lock<std::mutex> lock(mutex_);
        cond_var_.wait(lock, [&]{
            return !queue_.empty();
        });
        T value = std::move(queue_.front());
        queue_.pop();
        return value;
    }

   private:
    std::queue<T> queue_;
    std::mutex mutex_;
    std::condition_variable cond_var_;
};
```

In this example, we simply use a mutex to have mutual exclusion over all the queue data structures when pushing or popping elements. We want to keep this example simple and focus on the promise and future interactions. A better approach could be to use a vector or circular array and control the access to individual elements within the queue with mutexes.

The queue also uses a condition variable, `cond_var_`, to wait if the queue is empty when trying to pop an element and to notify one waiting thread when an element is pushed. Elements are moved in and out of the queue by moving them. This is needed as the queue will store futures, and as we learned earlier, futures are movable but non-copiable.

The thread-safe queue will be used to define a task queue that will store futures, as follows:

```
using TaskQueue = ThreadSafeQueue<std::future<int>>;
```

Then, we define a function, `producer`, that accepts a reference to the queue, and a value, `val`, that will be produced. This function just creates a promise, retrieves a future from the promise, and pushes that future into the queue. Then, we simulate a task running and producing the value, `val`, by making the thread wait for a random number of milliseconds. Finally, the value is stored in the promise:

```
void producer(TaskQueue& queue, int val) {
    std::promise<int> prom;
    auto fut = prom.get_future();
    queue.push(std::move(fut));

    std::this_thread::sleep_for(
            std::chrono::milliseconds(rand() % MAX_WAIT));

    prom.set_value(val);
}
```

At the other end of the communication channel, the `consumer` function accepts a reference to the same queue. Again, we simulate a task running on the consumer side by waiting for a random number of milliseconds. Then, a future is popped out of the queue and its result is retrieved, being the value, `val`, or an exception if something went wrong:

```
void consumer(TaskQueue& queue) {
    std::this_thread::sleep_for(
            std::chrono::milliseconds(rand() % MAX_WAIT));

    std::future<int> fut = queue.pop();
    try {
        int result = fut.get();
        std::cout << "Result: " << result << "\n";
    } catch (const std::exception& e) {
        std::cerr << "Exception: " << e.what() << '\n';
    }
}
```

For this example, we will use these constants:

```
const unsigned VALUE_RANGE = 1000;
const unsigned RESULTS_TO_PRODUCE = 10; // Numbers of items to
produce.
const unsigned MAX_WAIT = 500; // Maximum waiting time (ms) when
producing items.
```

In the main thread, two threads are started; the first runs the producer function `producerFunc`, which pushes some futures into the queue, while the second thread runs the consumer function `consumerFunc`, which consumes elements from the queue:

```
TaskQueue queue;

auto producerFunc = [](TaskQueue& queue) {
    auto n = RESULTS_TO_PRODUCE;
    while (n-- > 0) {
        int val = rand() % VALUE_RANGE;
        std::cout << "Producer: Sending value " << val
                  << std::endl;
        producer(queue, val);
    }
};

auto consumerFunc = [](TaskQueue& queue) {
    auto n = RESULTS_TO_PRODUCE;
    while (n-- > 0) {
        std::cout << "Consumer: Receiving value"
                  << std::endl;
        consumer(queue);
    }
};

std::jthread producerThread(producerFunc, std::ref(queue));
std::jthread consumerThread(consumerFunc, std::ref(queue));
```

Here is sample output of executing this code:

```
Producer: Sending value 383
Consumer: Receiving value
Producer: Sending value 915
Result: 383
Consumer: Receiving value
Producer: Sending value 386
Result: 915
```

```
Consumer: Receiving value
Producer: Sending value 421
Result: 386
Consumer: Receiving value
Producer: Sending value 690
Result: 421
Consumer: Receiving value
Producer: Sending value 926
Producer: Sending value 426
Result: 690
Consumer: Receiving value
Producer: Sending value 211
Result: 926
Consumer: Receiving value
Result: 426
Consumer: Receiving value
Producer: Sending value 782
Producer: Sending value 862
Result: 211
Consumer: Receiving value
Result: 782
Consumer: Receiving value
Result: 862
```

With a producer-consumer queue like this one, the consumer and producer are decoupled, and their threads communicate asynchronously, allowing both the producer and consumer to do extra work while the other side generates or processes the values.

Summary

In this chapter, we learned about promises and futures, how to use them to execute asynchronous code in separate threads, and also how to run callables using packaged tasks. These objects and mechanisms constitute and implement the key concepts of asynchronous programming used by many programming languages, including C++.

We also now understand why promises, futures, and packaged tasks cannot be copied, and thus how to share futures by using shared future objects.

Finally, we have shown how to use futures, promises, and packaged tasks to tackle real-life problems.

If you want to explore promises and futures deeper, it is worth mentioning some third-party open source libraries, especially **Boost.Thread** and **Facebook Folly**. These libraries include additional functionality, including callbacks, executors, and combinators.

In the next chapter, we will learn a simpler way to asynchronously invoke callables by using `std::async`.

Further reading

- Boost futures and promises: `https://theboostcpplibraries.com/boost.thread-futures-and-promises`
- Facebook Folly open source library: `https://github.com/facebook/folly`
- Futures for C++11 at Facebook: `https://engineering.fb.com/2015/06/19/developer-tools/futures-for-c-11-at-facebook`
- *'Futures and Promises' – How Instagram leverages it for better resource utilization*: `https://scaleyourapp.com/futures-and-promises-and-how-instagram-leverages-it/`
- SeaStar: Open source C++ framework for high-performance server applications: `https://seastar.io`

7
The Async Function

In the previous chapter, we learned about promises, futures, and packaged tasks. When we introduced packaged tasks, we mentioned that `std::async` provides a simpler way to achieve the same result, with less code and thus being cleaner and more concise.

The **async function** (`std::async`) is a function template that runs a callable object asynchronously where we can also select the method of execution by passing some flags defining the launch policy. It is a powerful tool for handling asynchronous operations, but its automatic management and lack of control over the thread of execution, among other aspects, can also make it unsuitable for certain tasks where fine-grained control or cancellation is required.

In this chapter, we are going to cover the following main topics:

- What is the async function and how do we use it?
- What are the different launch policies?
- What are the differences from previous methods, especially packaged tasks?
- What are the advantages and disadvantages of using `std::async`?
- Practical scenarios and examples

Technical requirements

The async function has been available since C++11, but some examples use features from C++14, such as `chrono_literals`, and C++20, such as `counting_semaphore`, so the code shown in this chapter can be compiled by compilers supporting C++20.

Please check the *Technical requirements* section in *Chapter 3*, for guidance on how to install GCC 13 and Clang 8 compilers.

You can find all the complete code in the following GitHub repository:

`https://github.com/PacktPublishing/Asynchronous-Programming-with-CPP`

The examples for this chapter are located under the `Chapter_07` folder. All source code files can be compiled using CMake as follows:

```
cmake . && cmake –build .
```

Executable binaries will be generated under the `bin` directory.

What is std::async?

`std::async` is a function template in C++ introduced by the C++ standard in the `<future>` header as part of the thread support library from C++11. It is used to run a function asynchronously, allowing the main thread (or other threads) to continue running concurrently.

In summary, `std::async` is a powerful tool for asynchronous programming in C++, making it easier to run tasks in parallel and manage their results efficiently.

Launching an asynchronous task

To execute a function asynchronously using `std::async`, we can use the same approaches we used when starting threads in *Chapter 3*, with the different callable objects.

One approach is using a function pointer:

```
void func() {
    std::cout << "Using function pointer\n";
}
auto fut1 = std::async(func);
```

Another approach is using a lambda function:

```
auto lambda_func = []() {
    std::cout << "Using lambda function\n";
};
auto fut2 = std::async(lambda_func);
```

We can also use an embedded lambda function:

```
auto fut3 = std::async([]() {
    std::cout << "Using embedded lambda function\n";
});
```

We can use a function object where `operator()` is overloaded:

```
class FuncObjectClass {
   public:
    void operator()() {
        std::cout << "Using function object class\n";
```

```
    }
};
auto fut4 = std::async(FuncObjectClass());
```

We can use a non-static member function by passing the address of the member function and the address of an object to call the member function:

```
class Obj {
  public:
    void func() {
        std::cout << "Using a non-static member function"
                  << std::endl;
    }
};
Obj obj;
auto fut5 = std::async(&Obj::func, &obj);
```

We can also use a static member function where only the address of the member function is needed as the method is static:

```
class Obj {
  public:
    static void static_func() {
        std::cout << "Using a static member function"
                  << std::endl;
    }
};
auto fut6 = std::async(&Obj::static_func);
```

When `std::async` is called, it returns a future where the result of the function will be stored, as we already learned in the previous chapter.

Passing values

Again, similarly to when we passed arguments when creating threads, arguments can be passed to the thread by value, by reference, or as pointers.

Here, we can see how to pass arguments by value:

```
void funcByValue(const std::string& str, int val) {
    std::cout << «str: « << str << «, val: « << val
              << std::endl;
}
std::string str{"Passing by value"};
auto fut1 = async(funcByValue, str, 1);
```

Passing by value implies a copy as a temporary object is created and the argument value is copied into it. This avoids data races, but it is much more costly.

The next example shows how to pass values by reference:

```
void modifyValues(std::string& str) {
    str += " (Thread)";
}
std::string str{"Passing by reference"};
auto fut2 = std::async(modifyValues, std::ref(str));
```

We can also pass values as a **const reference**:

```
void printVector(const std::vector<int>& v) {
    std::cout << "Vector: ";
    for (int num : v) {
        std::cout << num << " ";
    }
    std::cout << std::endl;
}
std::vector<int> v{1, 2, 3, 4, 5};
auto fut3 = std::async(printVector, std::cref(v));
```

Passing by reference is achieved by using `std::ref()` (non-constant references) or `std::cref()` (constant references), both defined in the `<functional>` header file, letting the variadic template (a class or function template that supports an arbitrary number of arguments) defining the thread constructor to treat the argument as a reference. Missing these functions when passing arguments means passing the arguments by value, which implies a copy, as mentioned earlier, making the function call more costly.

You can also move an object into the thread created by `std::async`, as follows:

```
auto fut4 = std::async(printVector, std::move(v));
```

Note that the vector `v` is in a valid but empty state after its content being moved.

Finally, we can also pass values by lambda captures:

```
std::string str5{"Hello"};
auto fut5 = std::async([&]() {
    std::cout << "str: " << str5 << std::endl;
});
```

In this example, the `str` variable is accessed by the lambda function executed by `std::async` as a reference.

Returning values

When `std:async` is called, it immediately returns a future that will hold the value that the function or callable object will compute, as we saw in the previous chapter when using promises and futures.

In the previous examples, we didn´t use the returned object from `std::async` at all. Let's rewrite the last example from the *Packaged tasks* section in *Chapter 6*, where we used a `std::packaged_task` object to compute the power of two values. But in this case, we will spawn several asynchronous tasks using `std::async` to compute these values, wait for the tasks to finish, store the results, and finally, show them in the console:

```
#include <chrono>
#include <cmath>
#include <future>
#include <iostream>
#include <thread>
#include <vector>
#include <syncstream>

#define sync_cout std::osyncstream(std::cout)

using namespace std::chrono_literals;

int compute(unsigned taskId, int x, int y) {
    std::this_thread::sleep_for(std::chrono::milliseconds(
                                rand() % 200));
    sync_cout << "Running task " << taskId << '\n';
    return std::pow(x, y);
}

int main() {
    std::vector<std::future<int>> futVec;
    for (int i = 0; i <= 10; i++)
        futVec.emplace_back(std::async(compute,
                            i+1, 2, i));

    sync_cout << "Waiting in main thread\n";
    std::this_thread::sleep_for(1s);

    std::vector<int> results;
    for (auto& fut : futVec)
        results.push_back(fut.get());

    for (auto& res : results)
```

```
        std::cout << res << ' ';
    std::cout << std::endl;
    return 0;
}
```

The `compute()` function simply gets two numbers, `x` and `y`, and computes x^y. It also gets a number representing the task identifier and waits for up to two seconds before printing a message in the console and computing the result.

In the `main()` function, the main thread launches several tasks computing a sequence of power-of-two values. The futures returned by calling `std::async` are stored in the `futVec` vector. Then, the main thread waits for one second, emulating some work. Finally, we traverse the `futVec` vector and call the `get()` function in each future element, thus waiting for that specific task to finish and return a value, and we store the returned value in another vector called `results`. Then, we print the content of the `results` vector before exiting the program.

This is the output when running that program:

```
Waiting in main thread
Running task 11
Running task 9
Running task 2
Running task 8
Running task 4
Running task 6
Running task 10
Running task 3
Running task 1
Running task 7
Running task 5
1 2 4 8 16 32 64 128 256 512 1024
```

As we can see, each task took a different amount of time to complete, thus the output is not ordered by task identifier. But as we traverse the `futVec` vector in order when getting the results, these are shown as in order.

Now that we have seen how to launch asynchronous tasks and pass arguments and return values, let's learn how to use launch policies to control the methods of execution.

Launch policies

Apart from specifying the function or callable object as an argument when using the `std::async` function, we can also specify the **launch policy**. Launch policies control how `std::async` schedules the execution of asynchronous task. These are defined in the `<future>` library.

The launch policy must be specified as the first argument when calling `std::async`. This argument is of the type `std::launch`, a bitmask value where its bits control the allowed methods of execution, which can be one or more of the following enumeration constants:

- `std::launch::async`: The task is executed in a separate thread.
- `std::launch::deferred`: Enables lazy evaluation by executing the task in the calling thread the first time its result is requested via the future `get()` or `wait()` method. All further accesses to the same `std::future` will return the result immediately. That means that the task will only be executed when the result is explicitly requested, which can lead to unexpected delays.

If not defined, by default the launch policy will be `std::launch::async | std::launch::deferred`. Also, implementations can provide additional launch policies.

Therefore, by default the C++ standard states that `std::async` can run in either asynchronous or deferred mode.

Note that when more than one flag is specified, the behavior is implementation-defined, so depending on the compiler we are using. The standard recommends using available concurrency and deferring the task if the default launch policy is specified.

Let's implement the following example to test the different launch policy behaviors. First, we define the `square()` function, which will serve as the asynchronous task:

```
#include <chrono>
#include <future>
#include <iostream>
#include <string>
#include <syncstream>

#define sync_cout std::osyncstream(std::cout)

using namespace std::chrono_literals;

int square(const std::string& task_name, int x) {
    sync_cout << "Launching " << task_name
              << « task...\n»;
    return x * x;
}
```

Then, in the `main()` function, the program starts by launching three different asynchronous tasks, one using the `std::launch::async` launch policy, another task using the `std::launch::deferred` launch policy, and a third task using the default launch policy:

```
sync_cout << "Starting main thread...\n";
auto fut_async = std::async(std::launch::async,
```

```
                              square, «async_policy", 2);
auto fut_deferred = std::async(std::launch::deferred,
                               square, «deferred_policy", 3);
auto fut_default = std::async(square,
                              «default_policy", 4);
```

As mentioned in the previous chapter, `wait_for()` returns a `std::future_status` object indicating whether the future is ready, deferred, or has timed out. Therefore, we can use that function to check whether any of the returned futures are deferred. We do that by using a lambda function, `is_deferred()`, that returns `true` in that case. At least one future object, `fut_deferred`, is expected to return `true`:

```
auto is_deferred = [](std::future<int>& fut) {
    return (fut.wait_for(0s) ==
            std::future_status::deferred);
};

sync_cout << "Checking if deferred:\n";
sync_cout << "  fut_async: " << std::boolalpha
          << is_deferred(fut_async) << '\n';
sync_cout << "  fut_deferred: " << std::boolalpha
          << is_deferred(fut_deferred) << '\n';
sync_cout << "  fut_default: " << std::boolalpha
          << is_deferred(fut_default) << '\n';
```

Then, the main program waits for one second, emulating some processing, and finally retrieves the results from the asynchronous tasks and prints their value:

```
sync_cout << "Waiting in main thread...\n";
std::this_thread::sleep_for(1s);
sync_cout << "Wait in main thread finished.\n";

sync_cout << "Getting result from "
          << "async policy task...\n";
int val_async = fut_async.get();
sync_cout << "Result from async policy task: "
          << val_async << '\n';

sync_cout << "Getting result from "
          << "deferred policy task...\n";
int val_deferred = fut_deferred.get();
sync_cout << "Result from deferred policy task: "
          << val_deferred << '\n';
```

```
sync_cout << "Getting result from "
          << "default policy task...\n";
int val_default = fut_default.get();
sync_cout << "Result from default policy task: "
          << val_default << '\n';
```

This is the output from running the preceding code:

```
Starting main thread...
Launching async_policy task...
Launching default_policy task...
Checking if deferred:
  fut_async: false
  fut_deferred: true
  fut_default: false
Waiting in main thread...
Wait in main thread finished.
Getting result from async policy task...
Result from async policy task: 4
Getting result from deferred policy task...
Launching deferred_policy task...
Result from deferred policy task: 9
Getting result from default policy task...
Result from default policy task: 16
```

Note how the tasks with the default and `std::launch::async` launch policies are executed while the main thread is sleeping. Therefore, the task is started as soon as it can be scheduled. Also note how the deferred task, using the `std::launch::deferred` launch policy, starts executing once the value is requested.

Next, let's learn how to handle exceptions happening in the asynchronous task.

Handling exceptions

Exception propagation from the asynchronous task to the main thread is not supported when using `std::async`. To enable exception propagation, we might need a promise object to store the exception that later can be accessed by the future returned when calling `std::async`. But that promise object is not accessible or provided by `std::async`.

One feasible way to achieve this is to use a `std::packaged_task` object wrapping the asynchronous task. But if that is the case, we should directly use a packaged task as described in the previous chapter.

We could also use nested exceptions, available since C++11, by using `std::nested_exception`, a polymorphic mixin class that can capture and store the current exception, allowing nested exceptions of arbitrary types. From a `std::nested_exception` object, we can retrieve the stored exception by using the `nested_ptr()` method or rethrow it by calling `rethrow_nested()`.

To create a nested exception, we can throw an exception using the `std::throw_with_nested()` method. If we want to rethrow an exception only if it's nested, we can use `std::rethrow_if_nested()`. All these functions are defined in the `<exception>` header.

Using all these functions, we can implement the following example, where an asynchronous task throws a `std::runtime_error` exception, which is caught in the main body of the asynchronous task and rethrown as a nested exception. This nested exception object is then caught again in the main function and the sequence of exceptions is printed out, as shown in the following code:

```
#include <exception>
#include <future>
#include <iostream>
#include <stdexcept>
#include <string>

void print_exceptions(const std::exception& e,
                      int level = 1) {
    auto indent = std::string(2 * level, ‹ ›);
    std::cerr << indent << e.what() << '\n';
    try {
        std::rethrow_if_nested(e);
    } catch (const std::exception& nestedException) {
        print_exceptions(nestedException, level + 1);
    } catch (...) { }
}

void func_throwing() {
    throw std::runtime_error(
               «Exception in func_throwing");
}

int main() {
    auto fut = std::async([]() {
        try {
            func_throwing();
        } catch (...) {
            std::throw_with_nested(
                 std::runtime_error(
                      "Exception in async task."));
        }
    });

    try {
        fut.get();
```

```
    } catch (const std::exception& e) {
        std::cerr << "Caught exceptions:\n";
        print_exceptions(e);
    }

    return 0;
}
```

As we can see in the example, an asynchronous task is created that executes the `func_throwing()` function inside a `try-catch` block. This function simply throws a `std::runtime_error` exception, which is caught and then rethrown as part of a `std::nested_exception` class by using the `std::throw_with_nested()` function. Later, in the main thread, when we try to retrieve the result from the `fut` future object by calling its `get()` method, the nested exception is thrown and captured again in the main try-catch block, where the `print_exceptions()` function is called with the captured nested exception as an argument.

The `print_exceptions()` function prints the reason for the current exception (`e.what()`) and rethrows the exception if nested, thus catching it again and recursively printing exception reasons with indentation by nesting level.

As each asynchronous task has its own future, the program can handle exceptions from multiple tasks separately.

Exceptions when calling std::async

Apart from exceptions happening in the asynchronous task, there are also some cases when `std::async` might throw an exception. These exceptions are as follows:

- `std::bad_alloc`: If there is not enough memory to store internal data structures needed by `std::async`.
- `std:system_error`: If a new thread cannot be started when using `std::launch::async` as the launch policy. In this case, the error condition will be `std::errc::resource_unavailable_try_again`. Depending on the implementation, if the policy is the default one, it might fall back to deferred invocation or implementation-defined policies.

Most of the time, these exceptions are thrown out due to resource exhaustion. A solution can be retrying later when some asynchronous tasks currently working have finished and released their resources. Another, more reliable, solution is to limit the number of asynchronous tasks running at a given time. We will implement this solution shortly, but first, let's understand the futures returned by `std::async` and how to achieve better performance when dealing with them.

Async futures and performance

Futures returned by `std::async` behave differently from the ones obtained from promises when their destructors are called. When these futures are destroyed, their `~future` destructor is called where the `wait()` function is executed, causing the thread that was spawned at creation to join.

That would impact the program performance by adding some overhead if the thread used by `std::async` has not already been joined, therefore we need to understand when the future object will go out of scope and thus its destructor will be called.

Let's see, with several short examples, how these futures behave and some recommendations on how to use them.

We start by defining a task, `func`, that simply multiplies its input value by 2 and also waits for some time, emulating a costly operation:

```
#include <chrono>
#include <functional>
#include <future>
#include <iostream>
#include <thread>

#define sync_cout std::osyncstream(std::cout)

using namespace std::chrono_literals;

unsigned func(unsigned x) {
    std::this_thread::sleep_for(10ms);
    return 2 * x;
}
```

To measure the performance of a block of code, we will asynchronously run several tasks (in this example, `NUM_TASKS = 32`) and measure the running time using the steady clock from the `<chrono>` library. To do that, we simply record a time point representing the current point in time when the task starts by using the following command:

```
auto start = std::chrono::high_resolution_clock::now();
```

We can define in the `main()` function the following lambda function to be called when the task finishes to obtain the duration in milliseconds:

```
auto duration_from = [](auto start) {
    auto dur = std::chrono::high_resolution_clock::now()
               - start;
    return std::chrono::duration_cast
               <std::chrono::milliseconds>(dur).count();
};
```

With that code in place, we can start measuring different approaches to how futures can be used.

Let's start by running several asynchronous tasks but discarding the future returned by `std::async`:

```
constexpr unsigned NUM_TASKS = 32;

auto start = std::chrono::high_resolution_clock::now();

for (unsigned i = 0; i < NUM_TASKS; i++) {
    std::async(std::launch::async, func, i);
}

std::cout << "Discarding futures: "
          << duration_from(start) << '\n';
```

The duration of this test is 334 ms on my PC, a Pentium i7 4790K at 4 GHz with four cores and eight threads.

For the next test, let's store the returned future, but don't wait for the result to be ready. Obviously, this is not the right way of using computer power by spawning asynchronous tasks as consuming resources and not processing the results, but we are doing this for testing and learning purposes:

```
start = std::chrono::high_resolution_clock::now();

for (unsigned i = 0; i < NUM_TASKS; i++) {
    auto fut = std::async(std::launch::async, func, i);
}

std::cout << "In-place futures: "
          << duration_from(start) << '\n';
```

In this case, the duration is still 334 ms. In both cases, a future is created, and when going out of scope at the end of each loop iteration, it must wait for the thread spawn by `std::async` to finish and join.

As you can see, we are launching 32 tasks, each one consuming at least 10 ms. That totals 320 ms, a value equivalent to 334 ms obtained in these tests. The remaining performance cost comes from starting threads, checking the `for` loop variable, storing the time points when using the steady clock, and so on.

To avoid creating a new future object each time `std::async` is called, and waiting for its destructor to be called, let's reuse the future object as shown in the following code. Again, this is not the proper way as we are discarding access to the results of previous tasks:

```
std::future<unsigned> fut;
start = std::chrono::high_resolution_clock::now();

for (unsigned i = 0; i < NUM_TASKS; i++) {
```

```
    fut = std::async(std::launch::async, func, i);
}

std::cout << "Reusing future: "
          << duration_from(start) << '\n';
```

Now the duration is 166 ms. The reductions are due to not waiting for each future, as they are not destroyed.

But this is not ideal as we might be interested in knowing the result of the asynchronous tasks. Therefore, we need to store the results in a vector. Let's modify the previous example by using the `res` vector to store the results from each task:

```
std::vector<unsigned> res;

start = std::chrono::high_resolution_clock::now();

for (unsigned i = 0; i < NUM_TASKS; i++) {
    auto fut = std::async(std::launch::async, func, i);
    res.push_back(fut.get());
}

std::cout << "Reused future and storing results: "
          << duration_from(start) << '\n';
```

In this case, the duration goes back to 334 ms. This is because we are again waiting for the results after spawning each task by calling `fut.get()` before launching another asynchronous task. We are serializing the tasks' execution.

A solution could be to store the futures returned by `std::async` in a vector, and later traverse that vector and get the results. The following code illustrates how to do this:

```
std::vector<unsigned> res;
std::vector<std::future<unsigned>> futsVec;

start = std::chrono::high_resolution_clock::now();

for (unsigned i = 0; i < NUM_TASKS; i++) {
    futsVec.emplace_back(std::async(std::launch::async,
                         func, i));
}

for (unsigned i = 0; i < NUM_TASKS; i++) {
    res.push_back( futsVec[i].get() );
}
```

```
std::cout << "Futures vector and storing results: "
          << duration_from(start) << '\n';
```

Now the duration is only 22 ms! But why is that possible?

Now all tasks are truly running asynchronously. The first loop launches all tasks and stores the futures in the `futsVec` vector. There is no longer any waiting period due to future destructors being called.

The second loop traverses `futsVec`, retrieves each result, and stores them in the results vector, `res`. The time to execute the second loop will be approximately the time needed to traverse the `res` vector plus the time used by the slowest task to be scheduled and executed.

If the system where the tests were running had enough threads to run all asynchronous tasks at once, the runtime could be halved. There are systems that can automatically manage several asynchronous tasks under the hood by letting the scheduler decide what tasks to run. In other systems, when trying to launch many threads at once, they might complain by raising an exception. In the next section, we implement a thread limiter by using semaphores.

Limiting the number of threads

As we saw earlier, if there are not enough threads to run several `std::async` calls, a `std::runtime_system` exception can be thrown and indicate resource exhaustion.

We can implement a simple solution by creating a thread limiter using counting semaphores (`std::counting_semaphore`), a multithreading synchronization mechanism explained in *Chapter 4*.

The idea is to use a `std::counting_semaphore` object, setting its initial value to the maximum concurrent tasks that the system allows, which can be retrieved by calling the `std::thread::hardware_concurrency()` function, as learned in *Chapter 2*, and then use that semaphore in the task function to block if the total number of asynchronous tasks exceed the maximum concurrent tasks.

The following snippet implements this idea:

```
#include <chrono>
#include <future>
#include <iostream>
#include <semaphore>
#include <syncstream>
#include <vector>

#define sync_cout std::osyncstream(std::cout)

using namespace std::chrono_literals;
```

```
void task(int id, std::counting_semaphore<>& sem) {
    sem.acquire();

    sync_cout << "Running task " << id << "...\n";
    std::this_thread::sleep_for(1s);

    sem.release();
}

int main() {
    const int total_tasks = 20;

    const int max_concurrent_tasks =
              std::thread::hardware_concurrency();
    std::counting_semaphore<> sem(max_concurrent_tasks);

    sync_cout << "Allowing only "
              << max_concurrent_tasks
              << " concurrent tasks to run "
              << total_tasks << " tasks.\n";

    std::vector<std::future<void>> futures;
    for (int i = 0; i < total_tasks; ++i) {
        futures.push_back(
                std::async(std::launch::async,
                           task, i, std::ref(sem)));
    }

    for (auto& fut : futures) {
        fut.get();
    }
    std::cout << "All tasks completed." << std::endl;
    return 0;
}
```

The program starts by setting the total number of tasks that will be launched. Then, it creates a counting semaphore, `sem`, setting its initial value to the hardware concurrency value, as explained earlier. Finally, it just launches all tasks and waits for their futures to be ready, as usual.

The key point in this example is that each task, before performing its job, acquires the semaphore, thus decrementing the internal counter or blocking until the counter can be decremented. When the job is done, the semaphore is released, which increments the internal counter and unblocks other tasks that try to acquire the semaphore at that time. That means that a task will launch only if there

is a free hardware thread to be used for that task. Otherwise, it will be blocked until another task releases the semaphore.

Before exploring some real-life scenarios, let's first understand some drawbacks of using `std::async`.

When not to use std::async

As we have seen during this chapter, `std::async` does not provide direct control over the number of threads used or access to the thread objects themselves. We know now how to limit the number of asynchronous tasks by using counting semaphores, but there might be some applications where this is not the optimal solution and fine-grained control is required.

Also, the automatic management of threads can reduce performance by introducing overhead, especially when many small tasks are launched, leading to excessive context switching and resource contention.

The implementation imposes some limit on the number of concurrent threads that can be used, which can degrade performance or even throw exceptions. As `std::async` and the available `std::launch` policies are implementation-dependent, the performance is not uniform across different compilers and platforms.

Finally, in this chapter, we didn't mention how to cancel an asynchronous task started by `std::async` as there is no standard way of doing so before completion.

Practical examples

Now it's time to implement some examples to tackle real-life scenarios using `std::async`. We will learn how to do the following:

- Perform parallel computation and aggregation
- Asynchronously search across different containers or a large dataset
- Asynchronously multiply two matrices
- Chain asynchronous operations
- Improve the pipeline example from the last chapter

Parallel computation and aggregation

Data aggregation is the process of collecting raw data from multiple sources and organizing, processing, and providing a summary of the data for easy consumption. This process is useful in many fields, such as business reporting, financial services, healthcare, social media monitoring, research, and academia.

As a naive example, let's compute the result of squaring all numbers between *1* and *n* and obtaining their average value. We know that using the following formula to compute the sum of square values

would be much quicker and require less computer power. Also, the task could be more meaningful, but the purpose of this example is to understand the relationship between the tasks, not the task itself.

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)\frac{n(n+1)}{6}(2n+1)}{6}$$

The `average_squares()` function in the following example launches an asynchronous task per value between *1* and *n* to compute the square value. The resulting future objects are stored in the `futsVec` vector, which is later used by the `sum_results()` function to compute the sum of the squared values. The result is then divided by `n` to obtain the average value:

```
#include <future>
#include <iomanip>
#include <iostream>
#include <vector>

int square(int x) {
    return x * x;
}

int sum_results(std::vector<std::future<int>>& futsVec) {
    int sum = 0;
    for (auto& fut : futsVec) {
        sum += fut.get();
    }
    return sum;
}

int average_squares(int n) {
    std::vector<std::future<int>> futsVec;
    for (int i = 1; i <= n; ++i) {
        futsVec.push_back(std::async(
                std::launch::async, square, i));
    }
    return double(sum_results(futures)) / n;
}

int main() {
    int N = 100;
    std::cout << std::fixed << std::setprecision(2);
    std::cout << "Sum of squares for N = " << N
              << « is « << average_squares(N) << '\n';
    return 0;
}
```

For example, for $n = 100$, we can check that the value will be the same as the one returned by the function divided by *n*, *3,383.50*.

This example can easily be modified to implement a solution using the **MapReduce** programming model to handle large datasets efficiently. MapReduce works by dividing the data processing into two phases; the Map phase, where independent chunks of data are filtered, sorted, and processed in parallel across multiple computers, and the Reduce phase where results from the Map phase are aggregated, summarizing the data. This is like what we just implemented, using the `square()` function in the Map phase, and the `average_squares()` and `sum_results()` functions in the Reduce phase.

Asynchronous searches

One way to speed up searching a target value into large containers is to parallelize the search. Next, we will present two examples. The first one involves searching across different containers by using one task per container, while the second one involves searching across a large container, dividing it into smaller segments, and using a task per segment.

Searching across different containers

In this example, we need to search for a `target` value in different containers of diverse types (`vector`, `list`, `forward_list`, and `set`) containing names of animals:

```
#include <algorithm>
#include <forward_list>
#include <future>
#include <iostream>
#include <list>
#include <set>
#include <string>
#include <vector>

int main() {
    std::vector<std::string> africanAnimals =
              {"elephant", "giraffe", "lion", "zebra"};
    std::list<std::string> americanAnimals =
              {"alligator", "bear", "eagle", "puma"};
    std::forward_list<std::string> asianAnimals =
              {"orangutan", "panda", "tapir", "tiger"};
    std::set<std::string> europeanAnimals =
              {«deer», «hedgehog», «linx", "wolf"};

    std::string target = «elephant»;
    /* .... */
}
```

To search for the target value, we launch an asynchronous task for each container, using the `search()` template function, which simply calls the `std::find` function in a container and returns `true` if the target value is found, or `false` otherwise:

```
template <typename C>
bool search(const C& container, const std::string& target) {
    return std::find(container.begin(), container.end(),
                     target) != container.end();
}
```

These asynchronous tasks are launched using the `std::async` function with the `std::launch::async` launch policy:

```
int main() {
    /* .... */
    auto fut1 = std::async(std::launch::async,
                   search<std::vector<std::string>>,
                   africanAnimals, target);
    auto fut2 = std::async(std::launch::async,
                   search<std::list<std::string>>,
                   americanAnimals, target);
    auto fut3 = std::async(std::launch::async,
                   search<std::forward_list<std::string>>,
                   asianAnimals, target);
    auto fut4 = std::async(std::launch::async,
                  search<std::set<std::string>>,
                  europeanAnimals, target);
    /* .... */
```

Finally, we simply retrieve all return values from the futures created when using `std::async` and bitwise OR them:

```
int main() {
    /* .... */
    bool found = fut1.get() || fut2.get() ||
                 fut3.get() || fut4.get();
    if (found) {
        std::cout << target
                  << " found in one of the containers.\n";
    } else {
        std::cout << target
                  << " not found in any of "
                  << "the containers.\n";
    }
```

```
    return 0;
}
```

This example also shows the power of the **Standard Template Library** (**STL**) as it provides generic and reusable algorithms that can be applied to different containers and data types.

Searching in a large container

In the next example, we will implement a solution to find a target value in a large vector containing 5 million integer values.

To generate the vector, we use a random number generator with a uniform integer distribution:

```
#include <cmath>
#include <iostream>
#include <vector>
#include <future>
#include <algorithm>
#include <random>

// Generate a large vector of random integers using a uniform
distribution
std::vector<int> generate_vector(size_t size) {
    std::vector<int> vec(size);
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> dist(1, size);
    std::generate(vec.begin(), vec.end(), [&]() {
        return dist(gen);
    });
    return vec;
}
```

To search for a target value in a segment of a vector, we can use the `std::find` function with `begin` and `end` iterators pointing to the segment limits:

```
bool search_segment(const std::vector<int>& vec, int target, size_t
begin, size_t end) {
    auto begin_it = vec.begin() + begin;
    auto end_it = vec.begin() + end;
    return std::find(begin_it, end_it, target) != end_it;
}
```

In the `main()` function, we start by generating the large vector using the `generate_vector()` function, then defining the `target` value to find and the number of segments (`num_segments`) which the vector will be split for parallel searches:

```
const int target = 100;

std::vector<int> vec = generate_vector(5000000);
auto vec_size = vec.size();

size_t num_segments = 16;
size_t segment_size = vec.size() / num_segments;
```

Then, for each segment, we define its `begin` and `end` iterators and launch an asynchronous task to search for the target value in that segment. Thus, we execute `search_segment` asynchronously in a separate thread by using `std::async` with the `std::launch::async` launch policy. To avoid copying the large vector when passing it as an input argument of `search_segment`, we use a constant reference, `std::cref`. The futures returned by `std::async` are stored in the `futs` vector:

```
std::vector<std::future<bool>> futs;
for (size_t i = 0; i < num_segments; ++i) {
    auto begin = std::min(i * segment_size, vec_size);
    auto end = std::min((i + 1) * segment_size, vec_size);
    futs.push_back( std::async(std::launch::async,
                               search_segment,
                               std::cref(vec),
                               target, begin, end) );
}
```

Note that the vector size is not always a multiple of the segment size, thus the last segment might be shorter than the others. To deal with this situation and avoid issues when accessing out-of-bounds memory when checking the last segment, we need to properly set the `begin` and `end` indexes for each segment. For that, we use `std::min` to get the minimum value between the size of the vector and the hypothetical index of the last element in the current segment.

Finally, we check all results by calling `get()` on each future and print a message to the console if the target value was found in any of the segments:

```
bool found = false;
for (auto& fut : futs) {
    if (fut.get()) {
        found = true;
        break;
    }
}
```

```
if (found) {
    std::cout << "Target " << target
              << " found in the large vector.\n";
} else {
    std::cout << "Target " << target
              << " not found in the large vector.\n";
}
```

This solution can be used as the base for more advanced solutions dealing with huge datasets in distributed systems where each asynchronous task tries to find a target value in a specific machine or cluster.

Asynchronous matrix multiplication

Matrix multiplication is one of the most relevant operations in computer science, used in many domains, such as computer graphics, computer vision, machine learning, and scientific computing.

In the following example, we will implement a parallel computing solution by distributing the computation across multiple threads.

Let's start by defining a matrix type, `matrix_t`, as a vector of vectors holding integer values:

```
#include <cmath>
#include <exception>
#include <future>
#include <iostream>
#include <vector>

using matrix_t = std::vector<std::vector<int>>;
```

Then, we implement the `matrix_multiply` function, which accepts two matrices, `A` and `B`, passing them as constant references, and returns their multiplication. We know that if `A` is a matrix *mxn* (*m* stands for rows and *n* for columns) and `B` is a matrix *pxq*, we can multiply `A` and `B` if $n = p$, and the resulting matrix will be of dimensions *mxq* (*m* rows and *q* columns).

The `matrix_multiply` function just starts by reserving some space to the result matrix, `res`. Then, it loops over the matrix by extracting column `j` from `B` and multiplying it by row `i` from `A`:

```
matrix_t matrix_multiply(const matrix_t& A,
                         const matrix_t& B) {
    if (A[0].size() != B.size()) {
        throw new std::runtime_error(
                  «Wrong matrices dimmensions.");
    }
    size_t rows = A.size();
```

```
    size_t cols = B[0].size();
    size_t inner_dim = B.size();
    matrix_t res(rows, std::vector<int>(cols, 0));

    std::vector<std::future<int>> futs;
    for (auto i = 0; i < rows; ++i) {
        for (auto j = 0; j < cols; ++j) {
            std::vector<int> column(inner_dim);
            for (size_t k = 0; k < inner_dim; ++k) {
                column[k] = B[k][j];
            }

            futs.push_back(std::async(std::launch::async,
                                      dot_product,
                                      A[i], column));
        }
    }
    for (auto i = 0; i < rows; ++i) {
        for (auto j = 0; j < cols; ++j) {
            res[i][j] = futs[i * cols + j].get();
        }
    }
    return res;
}
```

The multiplication is done asynchronously by using `std::async` with the `std::launch::async` launch policy, running the `dot_product` function. Each returned future from `std::async` is stored in the `futs` vector. The `dot_product` function computes the dot product of vectors `a` and `b`, representing a row from `A` and a column from `B`, by multiplying element by element and returning the sum of these products:

```
int dot_product(const std::vector<int>& a,
                const std::vector<int>& b) {
    int sum = 0;
    for (size_t i = 0; i < a.size(); ++i) {
        sum += a[i] * b[i];
    }
    return sum;
}
```

As the `dot_product` function expects two vectors, we need to extract each column from `B` before launching each asynchronous task. This also enhances the overall performance as the vectors might be stored in contiguous blocks of memory, thus being more cache-friendly during computation.

In the `main()` function, we just define two matrices, `A` and `B`, and use the `matrix_multipy` function to compute their product. All matrices are printed into the console using the `show_matrix` lambda function:

```
int main() {
    auto show_matrix = [](const std::string& name,
                          matrix_t& mtx) {
        std::cout << name << '\n';
        for (const auto& row : mtx) {
            for (const auto& elem : row) {
                std::cout << elem << " ";
            }
            std::cout << '\n';
        }
        std::cout << std::endl;
    };

    matrix_t A = {{1, 2, 3},
                  {4, 5, 6}};

    matrix_t B = {{7, 8, 9},
                  {10, 11, 12},
                  {13, 14, 15}};

    auto res = matrix_multiply(A, B);

    show_matrix("A", A);
    show_matrix("B", B);
    show_matrix("Result", res);

    return 0;
}
```

This is the output from running this example:

```
A
1 2 3
4 5 6

B
7 8 9
10 11 12
13 14 15

Result
```

```
66 72 78
156 171 186
```

Using contiguous memory blocks improves performance when traversing vectors as many of their elements can be read at once into the cache. Using contiguous memory allocation is not guaranteed when using `std::vector`, therefore it might be better to use `new` or `malloc`.

Chain asynchronous operations

In this example, we will implement a simple pipeline composed of three stages where each stage takes the result from the previous stage and computes a value.

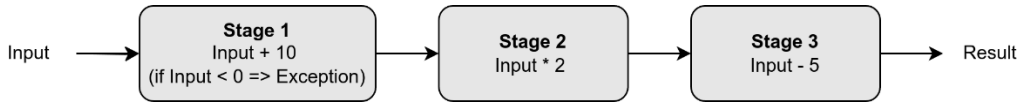


Figure 7.1 – Simple pipeline example

The first stage only accepts positive integers as input, otherwise it raises an exception, and adds 10 to that value before returning the result. The second stage multiplies its input by 2, and the third subtracts 5 from its input:

```
#include <future>
#include <iostream>
#include <stdexcept>

int stage1(int x) {
    if (x < 0) throw std::runtime_error(
                        "Negative input not allowed");
    return x + 10;
}

int stage2(int x) {
    return x * 2;
}

int stage3(int x) {
    return x - 5;
}
```

In the `main()` function, for the intermediate and final stages, we define the pipeline by using as input the futures generated by the previous stages. These futures are passed by reference to the lambda expression running the asynchronous code, where their `get()` function is used to get their result.

To retrieve the result from the pipeline, we just need to call the `get()` function from the future returned by the last stage. If an exception happens, for example, when `input_value` is negative, it is caught by the try-catch block:

```
int main() {
    int input_value = 5;

    try {
        auto fut1 = std::async(std::launch::async,
                               stage1, input_value);

        auto fut2 = std::async(std::launch::async,
                               [&fut1]() {
                                  return stage2(fut1.get()); });

        auto fut3 = std::async(std::launch::async,
                               [&fut2]() {
                                  return stage3(fut2.get()); });

        int final_result = fut3.get();
        std::cout << "Final Result: "
                  << final_result << std::endl;

    } catch (const std::exception &ex) {
        std::cerr << "Exception caught: "
                  << ex.what() << std::endl;
    }
    return 0;
}
```

The pipeline defined in this example is a simple one where each stage uses the future from the previous stage to get the input value and produce its result. In the next example, we will rewrite the pipeline implemented in the previous chapter using `std:async` with deferred launch policies to only execute the stages that are needed.

Asynchronous pipeline

As promised, in the last chapter, when we were implementing a pipeline, we mentioned that the different tasks could be kept switched off until needed by using futures with deferred execution. As also mentioned, this is useful in scenarios where the computation cost is high, but the result may not always be needed. As futures with deferred status can only be created by using `std::async`, now it's time to see how to do that.

We will implement the same pipeline described in the previous chapter, which follows the next task graph:

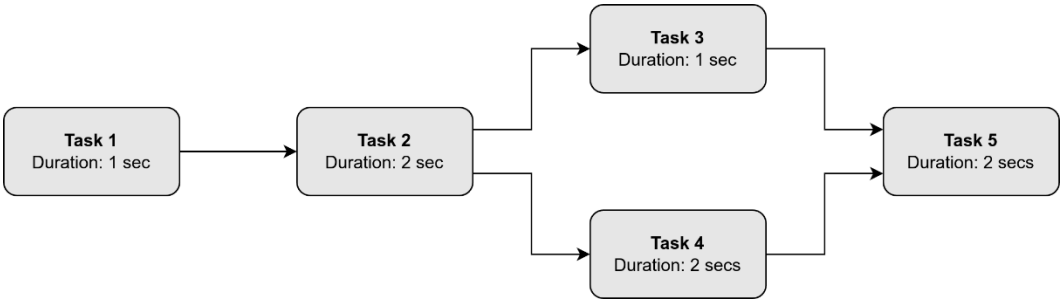


Figure 7.2 – Pipeline example

We start by defining the `Task` class. This class is like the one implemented in an example in the previous chapter but using the `std::async` function and storing the returned future instead of the promise used previously. Here, we will only comment on the relevant code changes from that example, so please look at that example for a full explanation of the `Task` class or check it out in the GitHub repository.

`Task` constructors store the task identifier (`id_`), the function to launch (`func_`), and whether the task has dependencies (`has_dependency_`). It also starts the asynchronous task in deferred launch mode by using `std::async` with the `std::launch::deferred` launch policy, meaning that the task is created but not started until needed. The returned future is stored in the `fut_` variable:

```
template <typename Func>
class Task {
   public:
    Task(int id, Func& func)
        : id_(id), func_(func), has_dependency_(false) {
        sync_cout << "Task " << id
                  << " constructed without dependencies.\n";
        fut_ = std::async(std::launch::deferred,
                          [this](){ (*this)(); });
    }

    template <typename... Futures>
    Task(int id, Func& func, Futures&&... futures)
        : id_(id), func_(func), has_dependency_(true) {
        sync_cout << "Task " << id
                   << " constructed with dependencies.\n";
        fut_ = std::async(std::launch::deferred,
                          [this](){ (*this)(); });
        add_dependencies(std::forward<Futures>
                                      (futures)...);
```

```
    }

   private:
    int id_;
    Func& func_;
    std::future<void> fut_;
    std::vector<std::shared_future<void>> deps_;
    bool has_dependency_;
};
```

The asynchronous tasks started by `std::async` call the `operator()` of their own instance (the `this` object). When that happens, `wait_completion()` is called, checking whether all futures in the shared future vector, `deps_`, storing dependent tasks are valid by calling their `valid()` function, and if so, waiting for them to finish by calling the `get()` function. When all dependent tasks are complete, the `func_` function is called:

```
public:
void operator()() {
    sync_cout << "Starting task " << id_ << std::endl;

    wait_completion();

    sync_cout << "Running task " << id_ << std::endl;
    func_();
}

private:
void wait_completion() {
    sync_cout << "Waiting completion for task "
              << id_ << std::endl;
    if (!deps_.empty()) {
        for (auto& fut : deps_) {
            if (fut.valid()) {
                sync_cout << "Fut valid so getting "
                          << "value in task " << id_
                          << std::endl;
                fut.get();
            }
        }
    }
}
```

There is also a new member function, `start()`, that waits for the `fut_` future created during the task construction when calling `std::async`. This will be used to trigger the pipeline by requesting the result of the last task:

```
public:
void start() {
    fut_.get();
}
```

As in the example in the previous chapter, we also define a member function called `get_dependency()` that returns a shared future constructed from `fut_`:

```
std::shared_future<void> get_dependency() {
    sync_cout << "Getting future from task "
              << id_ << std::endl;
    return fut_;
}
```

In the `main()` function, we define the pipeline by chaining task objects and setting their dependencies and the lambda function to run, `sleep1s` or `sleep2s`, following the diagram shown in *Figure 7.2*:

```
int main() {
    auto sleep1s = [](){
        std::this_thread::sleep_for(1s);
    };
    auto sleep2s = [](){
        std::this_thread::sleep_for(2s);
    };

    Task task1(1, sleep1s);
    Task task2(2, sleep2s, task1.get_dependency());
    Task task3(3, sleep1s, task2.get_dependency());
    Task task4(4, sleep2s, task2.get_dependency());
    Task task5(5, sleep2s, task3.get_dependency(),
               task4.get_dependency());

    sync_cout << "Starting the pipeline..." << std::endl;
    task5.start();

    sync_cout << "All done!" << std::endl;
    return 0;
}
```

Starting the pipeline is as simple as getting the result from the last task's future. We can do that by calling the `start()` method of `task5`. This will recursively call their dependency tasks by using the dependency vector and start the deferred asynchronous tasks.

This is the output of executing the preceding code:

```
Task 1 constructed without dependencies.
Getting future from task 1
Task 2 constructed with dependencies.
Getting future from task 2
Task 3 constructed with dependencies.
Getting future from task 2
Task 4 constructed with dependencies.
Getting future from task 4
Getting future from task 3
Task 5 constructed with dependencies.
Starting the pipeline...
Starting task 5
Waiting completion for task 5
Fut valid so getting value in task 5
Starting task 3
Waiting completion for task 3
Fut valid so getting value in task 3
Starting task 2
Waiting completion for task 2
Fut valid so getting value in task 2
Starting task 1
Waiting completion for task 1
Running task 1
Running task 2
Running task 3
Fut valid so getting value in task 5
Starting task 4
Waiting completion for task 4
Running task 4
Running task 5
All done!
```

We can see how the pipeline is created by calling each task's constructor and getting the futures from previous dependent tasks.

Then, when the pipeline is triggered, `task5` is started, starting `task3`, `task2`, and `task1` recursively. As `task1` has no dependencies, it doesn't need to wait for any other task to run its work, so it completes, allowing `task2` to complete, and later `task3`.

Next, `task5` continues checking its dependent tasks, so it's now `task4`'s turn to run. Since all `task4`'s dependent tasks are complete, `task4` just executes, allowing `task5` to run afterward, thus completing the pipeline.

This example can be improved by performing real computations and transferring results between tasks. Also, instead of deferred tasks, we could also think of stages with several parallel steps that can be computed in separate threads. Feel free to implement these improvements as an additional exercise.

Summary

In this chapter, we learned about `std::async`, how to use this function to execute asynchronous tasks, how to define its behavior by using launch policies, and how to handle exceptions.

We also now understand how the futures returned by the async function can impact performance and how to use them wisely. Also, we saw how to limit the number of asynchronous tasks by the number of available threads in the system by using counting semaphores.

We also mentioned some scenarios where `std::async` might not be the best tool for the job.

Finally, we implemented several examples covering real-life scenarios, which is useful for parallelizing many common tasks.

With all the knowledge acquired in this chapter, now we know when (and when not) to use the `std::async` function to run asynchronous tasks in parallel, improving the overall performance of applications, achieving better computer resource usage, and reducing resource exhaustion.

In the next chapter, we will learn how to achieve asynchronous execution by using coroutines, which have been available since C++20.

Further reading

- *Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14*, Scott Meyers, O'Reilly Media, Inc., 1st Edition – Chapter 7, Item 35 and Item 36
- `std::async`: `https://en.cppreference.com/w/cpp/thread/async`
- `std::launch`: `https://en.cppreference.com/w/cpp/thread/launch`
- Strassen algorithm: `https://en.wikipedia.org/wiki/Strassen_algorithm`
- Karatsuba algorithm: `https://en.wikipedia.org/wiki/Karatsuba_algorithm`
- OpenBLAS: `https://www.openblas.net`
- BLIS library: `https://github.com/flame/blis`
- MapReduce: `https://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html`

8

Asynchronous Programming Using Coroutines

In previous chapters, we saw different methods of writing asynchronous code in C++. We used threads, the basic units of execution, and some higher-level asynchronous code mechanisms, such as futures and promises and `std::async`. We will look at the Boost.Asio library in the next chapter. All these methods often use several system threads, created and managed by the kernel.

For example, the main thread of our program may need to access a database. This access may be slow, so we read the data in a different thread so our main thread can go on doing some other tasks. Another example is the producer-consumer model, where one or more threads generate data items to be processed, and one or more threads process those items in a fully asynchronous way.

Both of the preceding examples use threads, also called system (kernel) threads, and require different units of execution, one per thread.

In this chapter, we are going to study a different way to write asynchronous code – coroutines. Coroutines are an old concept from the late 1950s that was added to C++ only recently, since C++20. They don't need a separate thread (of course, we can have different threads running coroutines). Coroutines are a mechanism that allows us, among other things, to perform multiple tasks in a single thread.

In this chapter, we will cover the following main topics:

- What are coroutines and how are they implemented and supported by C++?
- Implementing basic coroutines to see what the requirements of a C++ coroutine are
- Generator coroutines and the new C++23 `std::generator`
- A string parser to parse integers
- Exceptions in coroutines

This chapter is about C++ coroutines implemented without using any third-party libraries. This way of writing coroutines is quite low level and we need to write code to support the compiler.

Technical requirements

For this chapter, you will need a C++20 compiler. For the generator examples, you will need a C++23 compiler. We have tested the examples with GCC `14.1`. The code is platform-independent, so even though we have a Linux focus in this book, all the examples should work on macOS and Windows. Please note that Visual Studio `17.11` doesn't support the C++23 `std::generator` yet.

The code for this chapter can be found in the book's GitHub repository: `https://github.com/PacktPublishing/Asynchronous-Programming-with-CPP`.

Coroutines

Before we start implementing coroutines in C++, we will introduce coroutines conceptually and see how they can be useful in our programs.

Let's start with a definition. A **coroutine** is a function that can suspend itself. Coroutines suspend themselves while waiting for an input value (while they are suspended, they don't execute) or after yielding a value such as the result of a computation. Once the input value is available or the caller requests another value, the coroutine resumes execution. We will come back to coroutines in C++ soon, but let's see with a real-life example how a coroutine works.

Imagine someone working as an assistant. They start the day reading emails.

One of the emails is a request for a report. After reading the email, they start writing the requested document. Once they have written the introductory paragraphs, they notice that they need another report from a colleague to get some accounting results from the previous quarter. They stop writing the report, write an email to their colleague requesting the needed information, and read the next email, which is a request to book a room for an important meeting in the afternoon. They open a special application the company has developed for booking meeting rooms automatically to optimize their use and book the room.

After a while, they receive the required accounting data from their colleague and resume writing the report.

The assistant is always busy working on their tasks. Writing the report is a good example of a coroutine: they start writing the report, then suspend the writing while they wait for the required information, and once the information arrives, they resume their writing. Of course, the assistant doesn't want to waste their time, and while they wait, they go on doing other tasks. Their colleague can be seen as another coroutine if they wait for requests and then send the appropriate response.

Now let's go back to software. Let's assume that we need to write a function that stores data in a database after processing some input information.

If the data comes all at once, we can implement just a function. The function will read the input, perform the required processing on it, and finally, write the result to a database. But what if the data

to be processed arrives in blocks and processing each block requires the result from the previous block processing (we can assume for the sake of this example that the first block processing needs only some default value)?

A possible solution to our problem would be to make the function wait for each data block, process it, store the result in the database, and then wait for the next one, and so on. But if we do that, we could potentially waste a lot of time while waiting for each block of data to arrive.

After reading the previous chapters, you may be thinking about different potential solutions: we could create a thread to read the data, copy the blocks to a queue, and a second thread (maybe the main thread) will process the data. This is an acceptable solution but using multiple threads may be overkill.

Another solution could be implementing a function to process only one block. The caller will wait for the input to be passed to the function and will keep the result of the previous block processing required to process each data block. In this solution, we must keep the state required by the data processing function in another function. It may be acceptable for a simple example, but once the processing gets more complicated (for example, requiring several steps with different intermediate results to be kept), the code might be difficult to understand and maintain.

We can solve the problem with a coroutine. Let's see some possible pseudocode for a coroutine that processes data in blocks and keeps intermediate results:

```
processing_result process_data(data_block data) {
    while (do_processing == true) {
        result_type result{ 0 };
        result = process_data_block(previous_result);
        update_database();
        yield result;
    }
}
```

The preceding coroutine receives a data block from the caller, performs all the processing, updates a database, and keeps the result required to process the next block. After yielding the result to the caller (more on yielding later), it suspends itself. Its execution will resume when the coroutine is called again by the caller requesting the processing of a new data block.

A coroutine such as this simplifies state management because it can keep the state between calls.

After this conceptual introduction to coroutines, we are going to start implementing them in C++20.

C++ coroutines

As we have seen, coroutines are just functions, but they are not like the functions we are used to. They have special properties that we will study in this chapter. In this section, we will focus on coroutines in C++.

A function starts executing when it's called and normally terminates with a return sentence or just when the function's end is reached.

A function runs from beginning to end. It may call another function (or even itself if it is recursive), and it may throw exceptions or have different return points. But it always runs from beginning to end.

A coroutine is different. A coroutine is a function that can suspend itself. The flow for a coroutine may be like the following pseudocode:

```
 void coroutine() {
    do_something();
    co_yield;
    do_something_else();
    co_yield;
    do_more_work();
    co_return;
}
```

We will see what those terms with the `co_` prefix mean soon.

For a coroutine, we need a mechanism to keep the execution state to be able to suspend/resume the coroutine. This is done for us by the compiler, but we must write some *helping* code to let the compiler help us back.

Coroutines in C++ are stackless. This means that the state we need to store to be able to suspend/resume the coroutine is stored in the heap calling `new`/`delete` to allocate/free dynamic memory. These calls are created by the compiler.

New keywords

Because a coroutine is essentially a function (with some special properties, but a function nonetheless), the compiler needs some way to know whether a given function is a coroutine. C++20 introduced three new keywords: `co_yield`, `co_await`, and `co_return`. If a function uses at least one of those three keywords, then the compiler knows it is a coroutine.

The following table summarizes the functionality of the new keywords:

Keyword	Input/Output	Coroutine State
`co_yield`	Output	Suspended
`co_await`	Input	Suspended
`co_return`	Output	Terminated

Table 8.1: New coroutine keywords

In the preceding table, we see that after `co_yield` and `co_await`, the coroutine suspends itself, and after `co_return`, it is terminated (`co_return` is the equivalent of the `return` statement in a C++ function). A coroutine cannot have a `return` statement; it must always use `co_return`. If the coroutine doesn't return any value and any of the other two coroutine keywords are used, the `co_return` statement can be omitted.

Coroutines restrictions

We have said that coroutines are functions using the new coroutines keywords. But coroutines have the following restrictions:

- Functions with a variable number of arguments using `varargs` can't be coroutines (a variadic function template can be a coroutine)
- A class constructor or destructor cannot be a coroutine
- The `constexpr` and `consteval` functions cannot be coroutines
- A function returning `auto` cannot be a coroutine but `auto` with a trailing return type can be
- The `main()` function cannot be a coroutine
- Lambdas can be coroutines

After studying the restrictions of coroutines (basically what kind of C++ functions cannot be coroutines), we are going to start implementing coroutines in the next section.

Implementing basic coroutines

In the previous section, we studied the basics of coroutines, what they are, and some use cases.

In this section, we will implement three simple coroutines to illustrate the basics of implementing and working with them:

- The simplest coroutine that just returns
- A coroutine sending values back to the caller
- A coroutine getting values from the caller

The simplest coroutine

We know that a coroutine is a function that can suspend itself and can be resumed by the caller. We also know that the compiler identifies a function as a coroutine if it uses at least one `co_yield`, `co_await`, or `co_return` expression.

The compiler will transform the coroutine source code and create some data structures and functions to make the coroutine functional and capable of being suspended and resumed. This is required to keep the coroutine state and be able to communicate with the coroutine.

The compiler will take care of all those details but bear in mind that C++ support for coroutines is quite low level. There are some libraries to make our lives easier when working with coroutines in C++. Some of them are **Lewis Baker's cppcoro** and **Boost.Cobalt**. The **Boost.Asio** library has support for coroutines too. These libraries are the subject of the next two chapters.

Let's start from scratch. And we mean by absolute scratch. We will write some code and be guided by both compiler errors and the C++ reference to write a basic but fully functional coroutine.

The following code is the simplest implementation of a coroutine:

```
void coro_func() {
    co_return;
}
int main() {
    coro_func();
}
```

Simple, isn't it? Our first coroutine will just return nothing. It will not do anything else. Sadly, the preceding code is too simple for a functional coroutine and will not compile. When compiling with GCC `14.1`, we get the following error:

```
error: coroutines require a traits template; cannot find
'std::coroutine_traits'
```

We also get the following note:

```
note: perhaps '#include <coroutine>' is missing
```

The compiler is giving us a hint: we may have missed including a required file. Let's include the `<coroutine>` header file. We'll deal with the error about the traits template in a minute:

```
#include <coroutine>
void coro_func() {
    co_return;
}
int main() {
    coro_func();
}
```

When compiling the preceding code, we get the following error:

```
 error: unable to find the promise type for this coroutine
```

The first version of our coroutine gave us a compiler error saying that the type `std::coroutine_traits` template couldn't be found. Now we get an error related to something called the *promise type*.

Looking at the C++ reference, we see that the `std::coroutine_traits` template determines the return type and parameter types of a coroutine. The reference also states that the return type of a coroutine must define a type named `promise_type`. Following the reference advice, we can write a new version of our coroutine:

```
#include <coroutine>
struct return_type {
    struct promise_type {
    };
};
template<>
struct std::coroutine_traits<return_type> {
    using promise_type = return_type::promise_type;
};
return_type coro_func() {
    co_return;
}
int main() {
    coro_func();
}
```

Please note that the return type of a coroutine can have any name (we have called it `return_type` here because is convenient for this simple example).

Compiling the preceding code again gives us some errors (they are edited for clarity). All the errors are about missing functions in the `promise_type` structure:

```
error: no member named 'return_void' in 'std::__n4861::coroutine_
traits<return_type>::promise_type'
error: no member named 'initial_suspend' in 'std::__n4861::coroutine_
traits<return_type>::promise_type'
error: no member named 'unhandled_exception' in 'std::__
n4861::coroutine_traits<return_type>::promise_type'
error: no member named 'final_suspend' in 'std::__n4861::coroutine_
traits<return_type>::promise_type'
error: no member named 'get_return_object' in 'std::__
n4861::coroutine_traits<return_type>::promise_type'
```

All the compiler errors we have seen until now are related to missing features in our code. Writing coroutines in C++ requires following some rules and helping the compiler to make its generated code functional.

The following is the final version of the simplest coroutine:

```
#include <coroutine>

struct return_type {
    struct promise_type {
        return_type get_return_object() noexcept {
            return return_type{ *this };
        }

        void return_void() noexcept {}

        std::suspend_always initial_suspend() noexcept {
            return {};
        }

        std::suspend_always final_suspend() noexcept {
            return {};
        }

        void unhandled_exception() noexcept {}
    };

    explicit return_type(promise_type&) {
    }

    ~return_type() noexcept {
    }
};

return_type coro_func() {
    co_return;
}

int main() {
    coro_func();
}
```

You may have noticed that we have removed the `std::coroutine_traits` template. Implementing the return and promise types is enough.

The preceding code compiles without any errors and you can run it. It does… nothing! But it's our first coroutine and we have learned that we need to supply some code required by the compiler to create the coroutine.

The promise type

The **promise type** is required by the compiler. We need to always have this type defined (it can be either a class or a struct), it must be named `promise_type`, and it must implement some functions specified in the C++ reference. We have seen that if we don't do that, the compiler will complain and give us errors.

The promise type must be defined inside the type returned by the coroutine, otherwise the code will not compile. The returned type (sometimes also called the **wrapper type** because it wraps `promise_type`) can be named arbitrarily.

A yielding coroutine

A do-nothing coroutine is good for illustrating some basic concepts. We will now implement another coroutine that can send data back to the caller.

In this second example, we will implement a coroutine that produces a message. It will be the "hello world" of coroutines. The coroutine will say hello and the caller function will print the message received from the coroutine.

To implement that functionality, we need to establish a communication channel from the coroutine to the caller. This channel is the mechanism that allows the coroutine to pass values to the caller and receive information from it. This channel is established through the coroutine's **promise type** and **handle**, which manages the state of the coroutine.

The communication channel works in the following way:

- **Coroutine frame**: When a coroutine is called, it creates a **coroutine frame**, which contains all the state information needed to suspend and resume its execution. This includes local variables, the promise type, and any internal state.
- **Promise type**: Each coroutine has an associated **promise type**, which is responsible for managing the coroutine's interaction with the caller functions. The promise is where the coroutine's return value is stored, and it provides functions to control the coroutine's behavior. We are going to see these functions in this chapter's examples. The promise is the interface through which the caller interacts with the coroutine.
- **Coroutine handle**: The **coroutine handle** is a type that gives access to the coroutine frame (the coroutine's internal state) and allows the caller to resume or destroy the coroutine. The handle is what the caller can use to resume the coroutine after it has been suspended (for example, after `co_await` or `co_yield`). The handle can also be used to check whether the coroutine is done or to clean up its resources.
- **Suspend and resume mechanism**: When a coroutine yields a value (`co_yield`) or awaits an asynchronous operation (`co_await`), it suspends its execution, saving its state in the coroutine frame. The caller can then resume the coroutine at a later point, retrieving the yielded or awaited value through the coroutine handle and continuing the execution.

We are going to see, in the following examples, that this communication channel requires a considerable amount of code on our side to help the compiler generate all the code required for a coroutine to be functional.

The following code is the new version of both the caller function and the coroutine:

```
return_type coro_func() {
    co_yield "Hello from the coroutine\n"s;

    co_return;
}

int main() {
    auto rt = coro_func();
    std::cout << rt.get() << std::endl;
    return 0;
}
```

The changes are as follows:

- `[1]`: The coroutine *yields* and sends some data (in this case, a `std::string` object) to the caller
- `[2]`: The caller reads that data and prints it

The required communication mechanism is implemented in the promise type and in the return type (which is a promise type wrapper).

When the compiler reads the `co_yield` expression, it will generate a call to the `yield_value` function defined in the promise type.

The following code is the implementation for our version of that function that generates (or yields) a `std::string` object:

```
std::suspend_always yield_value(std::string msg) noexcept {
    output_data = std::move(msg);
    return {};
}
```

The function gets a `std::string` object and moves it to the `output_data` member variable of the promise type. But this just keeps the data inside the promise type. We need a mechanism to get that string out of the coroutine.

The handle type

Once we require a communication channel to and from a coroutine, we need a way to refer to a suspended or executing coroutine. The C++ standard library implements such a mechanism in what is called a **coroutine handle**. Its type is `std::coroutine_handle` and it's a member variable of the return type. This structure is also responsible for the full life cycle of the handle, creating and destroying it.

The following code snippet is the functionality we added to our return type to manage a coroutine handle:

```
std::coroutine_handle<promise_type> handle{};

explicit return_type(promise_type& promise) : handle{ std::coroutine_
handle<promise_type>::from_promise(promise)} {
}

~return_type() noexcept {
    if (handle) {
        handle.destroy();
    }
}
```

The preceding code declares a coroutine handle of type `std::coroutine_handle<promise_type>` and creates the handle in the return type constructor. The handle is destroyed in the return type destructor.

Now, back to our yielding coroutine. The only missing bit is the `get()` function for the caller function to be able to access the string generated by the coroutine:

```
std::string get() {
    if (!handle.done()) {
        handle.resume();
    }
    return std::move(handle.promise().output_data);
}
```

The `get()` function resumes the coroutine if it is not terminated and then returns the string object.

The following is the full code for our second coroutine:

```
#include <coroutine>
#include <iostream>
#include <string>

using namespace std::string_literals;
```

```
struct return_type {
    struct promise_type {
        std::string output_data { };

        return_type get_return_object() noexcept {
            std::cout << "get_return_object\n";
            return return_type{ *this };
        }

        void return_void() noexcept {
            std::cout << "return_void\n";
        }

        std::suspend_always yield_value(
                        std::string msg) noexcept {
            std::cout << "yield_value\n";
            output_data = std::move(msg);
            return {};
        }

        std::suspend_always initial_suspend() noexcept {
            std::cout << "initial_suspend\n";
            return {};
        }

        std::suspend_always final_suspend() noexcept {
            std::cout << "final_suspend\n";
            return {};
        }

        void unhandled_exception() noexcept {
            std::cout << "unhandled_exception\n";
        }
    };

    std::coroutine_handle<promise_type> handle{};

    explicit return_type(promise_type& promise)
       : handle{ std::coroutine_handle<
                 promise_type>::from_promise(promise)}{
        std::cout << "return_type()\n";
    }
```

```
    ~return_type() noexcept {
        if (handle) {
            handle.destroy();
        }
        std::cout << "~return_type()\n";
    }

    std::string get() {
        std::cout << "get()\n";
        if (!handle.done()) {
            handle.resume();
        }
        return std::move(handle.promise().output_data);
    }
};

return_type coro_func() {
    co_yield "Hello from the coroutine\n"s;
    co_return;
}

int main() {
    auto rt = coro_func();
    std::cout << rt.get() << std::endl;
    return 0;
}
```

Running the preceding code prints the following messages:

```
get_return_object
return_type()
initial_suspend
get()
yield_value
Hello from the coroutine
~return_type()
```

This output shows us what is happening during the coroutine execution:

1. The `return_type` object is created after a call to `get_return_object`
2. The coroutine is initially suspended
3. The caller wants to get the message from the coroutine, so `get()` is called

4. `yield_value` is called and the coroutine is resumed and the message is copied to a member variable in the promise
5. Finally, the message is printed by the caller function, and the coroutine returns

Note that the promise (and promise type) have nothing to do with the C++ Standard Library `std::promise` type explained in *Chapter 6*.

A waiting coroutine

In the previous example, we saw how to implement a coroutine that can communicate back to the caller by sending it a `std::string` object. Now, we are going to implement a coroutine that can wait for input data sent by the caller. In our example, the coroutine will wait until it gets a `std::string` object and then print it. When we say that the coroutine "waits," we mean it is suspended (that is, not executing) until the data is received.

Let's start with changes to both the coroutine and the caller function:

```
return_type coro_func() {
    std::cout << co_await std::string{ };
    co_return;
}

int main() {
    auto rt = coro_func();
    rt.put("Hello from main\n"s);
    return 0;
}
```

In the preceding code, the caller function calls the `put()` function (a method in the return type structure) and the coroutine calls `co_await` to wait for a `std::string` object from the caller.

The changes to the return type are simple, that is, just adding the `put()` function:

```
void put(std::string msg) {
    handle.promise().input_data = std::move(msg);
    if (!handle.done()) {
        handle.resume();
    }
}
```

We need to add the `input_data` variable to the promise structure. But just with those changes to our first example (we take it as the starting point for the rest of the examples in this chapter because it's the minimum code to implement a coroutine) and the coroutine handle from the previous example, the code cannot be compiled. The compiler gives us the following error:

```
error: no member named 'await_ready' in 'std::string' {aka 'std::__
cxx11::basic_string<char>'}
```

Going back to the C++ reference, we see that when the coroutine calls `co_await`, the compiler will generate code to call a function in the promise object called `await_transform`, which has a parameter of the same type as the data the coroutine is waiting for. As its name implies, `await_transform` is a function that transforms any object (in our example, `std::string`) into an awaitable object. `std::string` is not awaitable, hence the previous compiler error.

`await_transform` must return an **awaiter** object. This is just a simple struct implementing a required interface for the awaiter to be usable by the compiler.

The following code shows our implementation of the `await_transform` function and the `awaiter` struct:

```
auto await_transform(std::string) noexcept {
    struct awaiter {
        promise_type& promise;

        bool await_ready() const noexcept {
            return true;
        }

        std::string await_resume() const noexcept {
            return std::move(promise.input_data);
        }

        void await_suspend(std::coroutine_handle<
                           promise_type>) const noexcept {

        }
    };
    return awaiter(*this);
}
```

The `promise_type` function `await_transform` is required by the compiler. We cannot use a different identifier for this function. The parameter type must be the same as the object the coroutine is waiting for. The `awaiter` struct can be named with any name. We used `awaiter` here because is descriptive. The `awaiter` struct must implement three functions:

- `await_ready`: This is called to check whether the coroutine is suspended. If that is the case, it returns `false`. In our example, it always returns `true` to indicate the coroutine is not suspended.
- `await_resume`: This resumes the coroutine and generates the result of the `co_await` expression.
- `await_suspend`: In our simple awaiter, this returns `void`, meaning the control is passed to the caller and the coroutine is suspended. It's also possible for `await_suspend` to return a Boolean. Returning `true` in this case is like returning `void`. Returning `false` means the coroutine is resumed.

This is the code for the full example of the waiting coroutine:

```
#include <coroutine>
#include <iostream>
#include <string>

using namespace std::string_literals;

struct return_type {
    struct promise_type {
        std::string input_data { };

        return_type get_return_object() noexcept {
            return return_type{ *this };
        }

        void return_void() noexcept {
        }

        std::suspend_always initial_suspend() noexcept {
            return {};
        }

        std::suspend_always final_suspend() noexcept {
            return {};
        }

        void unhandled_exception() noexcept {
        }
```

```
        auto await_transform(std::string) noexcept {
            struct awaiter {
                promise_type& promise;

                bool await_ready() const noexcept {
                    return true;
                }

                std::string await_resume() const noexcept {
                    return std::move(promise.input_data);
                }

                void await_suspend(std::coroutine_handle<
                                   promise_type>) const noexcept {
                }
            };

            return awaiter(*this);
        }
    };

    std::coroutine_handle<promise_type> handle{};

    explicit return_type(promise_type& promise)
      : handle{ std::coroutine_handle<
                        promise_type>::from_promise(promise)} {
    }

    ~return_type() noexcept {
        if (handle) {
            handle.destroy();
        }
    }

    void put(std::string msg) {
        handle.promise().input_data = std::move(msg);
        if (!handle.done()) {
            handle.resume();
        }
    }
};
```

```
return_type coro_func() {
    std::cout << co_await std::string{ };
    co_return;
}

int main() {
    auto rt = coro_func();
    rt.put("Hello from main\n"s);
    return 0;
}
```

In this section, we have seen three basic examples of coroutines. We have implemented the simplest coroutine and then coroutines with communication channels to both generate data for the caller (`co_yield`) and wait for data from the caller (`co_await`).

In the next section, we will implement a type of coroutine called a generator and generate sequences of numbers.

Coroutine generators

A **generator** is a coroutine that generates a sequence of elements by repeatedly resuming itself from the point that it was suspended.

A generator can be seen as an *infinite* sequence because it can generate an arbitrary number of elements. The caller function can get as many new elements from the generator as it needs.

When we say infinite, we mean in theory. A generator coroutine will yield elements without a definite last element (it is possible to implement generators with a limited range) but, in practice, we must deal with issues such as overflow in the case of numerical sequences.

Let's implement a generator from scratch, applying the knowledge we have gained in the previous sections of this chapter.

Fibonacci sequence generator

Imagine we are implementing an application and we need to use the Fibonacci sequence. As you probably already know, the **Fibonacci sequence** is a sequence in which each number is the sum of the two preceding ones. The first element is 0, the second element is 1, and then we apply the definition and generate element after element.

Fibonacci sequence: $F(n) = F(n-2) + F(n-1); \qquad F(0) = 0,\ F(1) = 1$

We can always generate these numbers with a `for` loop. But if we need to generate them at different points in our program, we need to implement a way to store the state of the sequence. We need to keep somewhere in our program what the last element we generated was. Was it the fifth or maybe the tenth element?

A coroutine is a very good solution for this problem; it will keep the required state itself and it will suspend until we request the next number in the sequence.

Here's the code using a generator coroutine:

```
int main() {
    sequence_generator<int64_t> fib = fibonacci();

    std::cout << "Generate ten Fibonacci numbers\n"s;

    for (int i = 0; i < 10; ++i) {
        fib.next();
        std::cout << fib.value() << " ";
    }
    std::cout << std::endl;

    std::cout << "Generate ten more\n"s;

    for (int i = 0; i < 10; ++i) {
        fib.next();
        std::cout << fib.value() << " ";
    }
    std::cout << std::endl;

    std::cout << "Let's do five more\n"s;

    for (int i = 0; i < 5; ++i) {
        fib.next();
        std::cout << fib.value() << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

As you can see in the preceding code, we generate the numbers we need without worrying about what the last element was. The sequence is generated by the coroutine.

Note that even though in theory the sequence is infinite, our program must be aware of potential overflow for very big Fibonacci numbers.

To implement the generator coroutine, we follow the principles explained previously in this chapter.

First, we implement the coroutine function:

```
sequence_generator<int64_t> fibonacci() {
    int64_t a{ 0 };
    int64_t b{ 1 };
    int64_t c{ 0 };

    while (true) {
        co_yield a;
        c = a + b;
        a = b;
        b = c;
    }
}
```

The coroutine just generates the next element in the Fibonacci sequence by applying the formula. The elements are generated in an infinite loop, but the coroutine suspends itself after `co_yield`.

The return type is the `sequence_generator` struct (we use a template to be able to use either 32- or 64-bit integers). It contains a promise type, pretty much like the one in the yielding coroutine we saw in a previous section.

In the `sequence_generator` struct, we added two functions that are useful when implementing a sequence generator.

```
void next() {
    if (!handle.done()) {
        handle.resume();
    }
}
```

The `next()` function resumes the coroutine for a new Fibonacci number in the sequence to be generated.

```
int64_t value() {
    return handle.promise().output_data;
}
```

The `value()` function returns the last generated Fibonacci number.

This way, we decouple element generation and its retrieval Qvalue.

Please find the full code for this example in the book's accompanying GitHub repository.

C++23 std::generator

We have seen that implementing even the most basic coroutine in C++ requires a certain amount of code. This may change in C++26 with more support for coroutines in the C++ Standard Library, which will allow us to write coroutines much more easily.

C++23 introduced the `std::generator` template class. By using it, we can write coroutine-based generators without writing any of the required code, such as the promise type, the return type, and all their functions. To run this example, you will need a C++23 compiler. We have used GCC 14.1. `std::generator` is not available in Clang.

Let's see the Fibonacci sequence generator using the new C++23 Standard Library features:

```
#include <generator>
#include <iostream>

std::generator<int> fibonacci_generator() {
    int a{ };
    int b{ 1 };

    while (true) {
        co_yield a;
        int c = a + b;
        a = b;
        b = c;
    }
}
auto fib = fibonacci_generator();

int main() {
    int i = 0;
    for (auto f = fib.begin(); f != fib.end(); ++f) {
        if (i == 10) {
            break;
        }
        std::cout << *f << " ";
        ++i;
    }
    std::cout << std::endl;
}
```

The first step is to include the `<generator>` header file. Then, we just write the coroutine because all the rest of the required code has been written for us. In the preceding code, we access the generated elements with an iterator (which is provided by the C++ Standard Library). This allows us to use a range-for loop, algorithms, and ranges.

It is also possible to write a version of the Fibonacci generator to generate a certain number of elements instead of an infinite series:

```
std::generator<int> fibonacci_generator(int limit) {
    int a{ };
    int b{ 1 };

    while (limit--) {
        co_yield a;
        int c = a + b;
        a = b;
        b = c;
    }
}
```

The code changes are very simple: just pass the number of elements we want the generator to generate and use it as the termination condition in the `while` loop.

In this section, we have implemented one of the most common coroutine types – a generator. We have implemented generators both from scratch and using the C++23 `std::generator` class template.

We will implement a simple string parser coroutine in the next section.

Simple coroutine string parser

In this section, we will implement our last example: a simple string parser. The coroutine will wait for the input, a `std::string` object, and will yield the output, a number, after parsing the input string. To simplify the example, we will assume that the string representation of the number doesn't have any errors and that the end of a number is represented by the hash character, `#`. We will also assume that the number type is `int64_t` and that the string won't contain any values out of that integer type range.

The parsing algorithm

Let's see how to convert a string representing an integer into a number. For example, the string `"-12321#"` represents the number -12321. To convert the string into a number, we can write a function like this:

```
int64_t parse_string(const std::string& str) {
    int64_t num{ 0 };
    int64_t sign { 1 };

    std::size_t c = 0;
    while (c < str.size()) {
```

```
        if (str[c] == '-') {
            sign = -1;
        }
        else if (std::isdigit(str[c])) {
            num = num * 10 + (str[c] - '0');
        }
        else if (str[c] == '#') {
            break;
        }
        ++c;
    }

    return num * sign;
}
```

The code is quite simple because of the assumption that the string is well formed. If we read the minus sign, `-`, then we change the sign to -1 (by default, we assume positive numbers, and if there is a + sign, it is simply ignored). Then, the digits are read one by one, and the number value is calculated as follows.

The initial value of `num` is `0`. We read the first digit and add its numeric value to the current `num` value multiplied by 10. This is the way we read numbers: the leftmost digit will be multiplied by 10 as many times as the number of digits to its right.

When we use characters to represent digits, they have some value according to the ASCII representation (we assume no wide characters or any other character type is used). The characters *0* to *9* have consecutive ASCII codes, so we can easily convert them to numbers by just subtracting *0*.

Even if for the preceding code the last character check is not necessary, we have included it here. When the parser routine finds the # character, it terminates the parsing loop and returns the final number value.

We can use this function to parse any string and get the number value, but we need the full string to convert it into a number.

Let's think about this scenario: the string is being received from a network connection and we need to parse it and convert it into a number. We may save the characters to a temporary string and then call the preceding function.

But there is another issue: what if the characters arrive slowly, such as once every few seconds, because that's the way they are transmitted? We want to keep our CPU busy and, if possible, do some other task (or tasks) while waiting for each character to arrive.

There are different approaches to solving this problem. We can create a thread and process the string concurrently, but this can be costly in computer time for such a simple task. We can use `std::async` too.

The parsing coroutine

We are working with coroutines in this chapter, so we will implement the string parser using C++ coroutines. We don't need an extra thread, and because of the asynchronous nature of coroutines, it will be quite easy to perform any other processing while the characters arrive.

The boilerplate code we will need for the parsing coroutine is pretty much the same as the code we have already seen in the previous examples. The parser itself is quite different. See the following code:

```
async_parse<int64_t, char> parse_string() {
    while (true) {
        char c = co_await char{ };
        int64_t number { };
        int64_t sign { 1 };

        if (c != '-' && c != '+' && !std::isdigit(c)) {
            continue;
        }

        if (c == '-') {
            sign = -1;
        }
        else if (std::isdigit(c)) {
            number = number * 10 + c - '0';
        }

        while (true) {
            c = co_await char{};
            if (std::isdigit(c)) {
                number = number * 10 + c - '0';
            }
            else {
                break;
            }
        }

        co_yield number * sign;
    }
}
```

I think that you can now easily recognize the return type (`async_parse<int64_t, char>`) and that the parser coroutine suspends itself waiting for an input character. The coroutine will suspend itself after yielding the number once the parsing has been done.

But you can see too that the preceding code is not as simple as our first attempt at parsing a string into a number.

First, the parser coroutine parses one character after another. It doesn't get the full string to parse, hence the infinite `while (true)` loops. We don't know how many characters there are in the full string, so we need to keep on receiving and parsing them.

The outer loop means that the coroutine will parse numbers, one after another, as the characters arrive – forever. But remember that it suspends itself to wait for the characters, so we don't waste CPU time.

Now, one character arrives. The first check is to see whether it is a valid character for our number. If the character is not either the minus sign, `-`, the plus sign, `+`, or a digit, then the parser waits for the next character.

If the next character is a valid one, then the following apply:

- If it is the minus sign, we change the sign value to -1
- If it is the plus sign, we ignore it
- If it is a digit, we parse it into the number, updating the current number value using the same method as we saw in the first version of the parser

After the first valid character, we enter a new loop to receive the rest of the characters, either digits or the separator character (`#`). Note here that when we say valid character, we mean good for numerical conversion. We are still assuming that the input characters form a valid number that is correctly terminated.

Once the number has been converted, it is yielded by the coroutine and the outer loop executes again. The terminating character is needed here because the input character stream is, in theory, endless, and it can contain many numbers.

The code for the rest of the coroutine can be found in the GitHub repo. It follows the same convention as any other coroutine. First, we define the return type:

```
template <typename Out, typename In>
struct async_parse {
// …
};
```

We use a template for flexibility because it allows us to parameterize both the input and output data types. In this case, these types are `int64_t` and `char`, respectively.

The input and output data items are the following:

```
std::optional<In> input_data { };
Out output_data { };
```

For input, we are using `std::optional<In>` because we need a way to know whether we have received a character. We use the `put()` function to send a character to the parser:

```
 void put(char c) {
    handle.promise().input_data = c;
    if (!handle.done()) {
        handle.resume();
    }
}
```

This function just assigns a value to the `std::optional input_data` variable. To manage the waiting for the characters, we implement the following awaiter type:

```
auto await_transform(char) noexcept {
    struct awaiter {
        promise_type& promise;

        [[nodiscard]] bool await_ready() const noexcept {
            return promise.input_data.has_value();
        }

        [[nodiscard]] char await_resume() const noexcept {
            assert (promise.input_data.has_value());
            return *std::exchange(
                           promise.input_data,
                           std::nullopt);
        }

        void await_suspend(std::coroutine_handle<
                           promise_type>) const noexcept {
        }
    };

    return awaiter(*this);

}
```

The `awaiter` struct implements two functions to handle the input data:

- `await_ready()`: Returns `true` if the optional `input_data` variable contains a valid value. It returns `false` otherwise.
- `await_resume()`: Returns the value stored in the optional `input_data` variable and *empties* it, assigning it to `std::nullopt`.

In this section, we have seen how to implement a simple parser using C++ coroutines. This is our last example, illustrating a very basic stream processing function using coroutines. In the next section, we will see exceptions in coroutines.

Coroutines and exceptions

In the previous sections, we implemented a few basic examples to learn the main C++ coroutines concepts. We implemented a very basic coroutine first to understand what the compiler required from us: the return type (sometimes called the wrapper type because it wraps the promise type) and the promise type.

Even for such a simple coroutine, we had to implement some functions we explained while we wrote the examples. But one function has not been explained yet:

```
void unhandled_exception() noexcept {}
```

We assumed then that coroutines couldn't throw exceptions, but the truth is they can. We can add the functionality to handle exceptions in the body of the `unhandled_exception()` function.

Exceptions in coroutines can happen while the return type or the promise type object is created and while the coroutine is executed (as in a normal function, coroutines can throw exceptions).

The difference is that if the exception is thrown before the coroutine is executed, the code creating the coroutine must handle the exception, while if the exception is thrown when the coroutine is executed, then `unhandled_exception()` is called.

The first case is just the usual exception handling with no special functions called. We can put the coroutine creation inside a `try-catch` block and handle the possible exceptions as we normally do in our code.

If, on the other hand, `unhandled_exception()` is called (inside the promise type), we must implement the exception-handling functionality inside that function.

There are different strategies to handle such exceptions. Among them are the following:

- Rethrow the exception so we can handle it outside the promise type (that is, in our code).
- Terminate the program (for example, calling `std::terminate`).
- Leave the function empty. In this case, the coroutine will crash and it will very likely crash the program too.

Because we have implemented very simple coroutines, we have left the function empty.

In this last section, we have introduced the exception-handling mechanism for coroutines. It is very important to handle exceptions properly. For example, if you know that after an exception occurs inside a coroutine, it won't be able to recover; then, it may be better to let the coroutine crash and handle the exception from another part of the program (usually from the caller function).

Summary

In this chapter, we have seen coroutines, a recently introduced feature in C++ that allows us to write asynchronous code without creating new threads. We have implemented a few simple coroutines to explain the basic requirements of a C++ coroutine. Additionally, we have learned how to implement generators and a string parser. Finally, we have seen exceptions in coroutines.

Coroutines are important in asynchronous programming because they let the program suspend execution at specific points and resume later, allowing other tasks to run in the meantime, all running in the same thread. They allow better resource utilization, reduce waiting time, and improve the scalability of applications.

In the next chapter, we will introduce Boost.Asio – a very powerful library for writing asynchronous code in C++.

Further reading

- *C++ Coroutines for Beginners*, Andreas Fertig, Meeting C++ Online, 2024
- *Deciphering Coroutines*, Andreas Weiss, CppCon 2022

Part 4: Advanced Asynchronous Programming with Boost Libraries

In this part, we will learn about advanced asynchronous programming techniques using powerful Boost libraries, enabling us to efficiently manage tasks that interact with external resources and system-level services. We will explore the **Boost.Asio** and **Boost.Cobalt** libraries, learning how they simplify the development of asynchronous applications while offering fine-grained control over complex processes such as task management and coroutine execution. Through hands-on examples, we will see how Boost.Asio handles asynchronous I/O operations in both single-threaded and multithreaded environments, and how Boost.Cobalt abstracts away the complexities of C++20 coroutines, allowing us to focus on functionality instead of low-level coroutine management.

This part has the following chapters:

- *Chapter 9*, *Asynchronous Programming Using Boost.Asio*
- *Chapter 10*, *Coroutines with Boost.Cobalt*

9

Asynchronous Programming Using Boost.Asio

Boost.Asio is a C++ library included in the well-known Boost libraries family that simplifies the development of solutions dealing with asynchronous **input/output** (**I/O**) tasks managed by the **operating system** (**OS**), making it easier to develop asynchronous software that deals with internal and external resources, such as network communications services or file operations.

For that purpose, Boost.Asio defines OS services (services belonging to and managed by the OS), I/O objects (providing interfaces to OS services), and the I/O execution context object (an object that behaves as a services registry and proxy).

In the following pages, we will introduce Boost.Asio, describe its main building blocks, and explain some common patterns to develop asynchronous software with this library, which are widely used in the industry.

In this chapter, we're going to cover the following main topics:

- What Boost.Asio is and how it simplifies asynchronous programming with external resources
- What I/O objects and I/O execution contexts are, and how they interact with OS services and between each other
- What the Proactor and Reactor design patterns are, and how they are related to Boost.Asio
- How to keep the program thread-safe and how to serialize tasks using strands
- How to efficiently pass data to asynchronous tasks using buffers
- How to cancel asynchronous operations
- Examples of common practices with timers and networking applications

Technical requirements

For this chapter, we will need to install the Boost C++ libraries. The most recent version when writing this book is Boost 1.85.0. Here are the release notes:

`https://www.boost.org/users/history/version_1_85_0.html`

For installation instructions in Unix variants systems (Linux, macOS), check out the following link:

`https://www.boost.org/doc/libs/1_85_0/more/getting_started/unix-variants.html`

For Windows systems, check out this link:

`https://www.boost.org/doc/libs/1_85_0/more/getting_started/windows.html`

Also, depending on the project we want to develop, we might need to configure **Boost.Asio** or install dependencies:

`https://www.boost.org/doc/libs/1_85_0/doc/html/boost_asio/using.html`

All code shown during this chapter will be supported by the C++20 version. Please check the technical requirements section in *Chapter 3* with some guidance on how to install GCC 13 and Clang 8 compilers.

You can find the complete code in the following GitHub repository:

`https://github.com/PacktPublishing/Asynchronous-Programming-with-CPP`

The examples for this chapter are located under the `Chapter_09` folder. All source code files can be compiled using CMake as follows:

```
cmake . && cmake —build .
```

Executable binaries will be generated under the *bin* directory.

What is Boost.Asio?

Boost.Asio is a cross-platform C++ library created by Chris Kohlhoff that provides a portable network and low-level I/O programming, including sockets, timers, hostname resolution, socket iostreams, serial ports, file descriptors, and Windows HANDLEs, providing a consistent asynchronous model. It also provides coroutine support, but as we learned in the previous chapter, they are now available in C++20, so we will only introduce them briefly in this chapter.

Boost.Asio allows the program to manage long-running operations without the explicit usage of threads and locks. Also, as it implements a layer on top of the OS services, it allows portability, efficiency, ease of use, and scalability, using the most appropriate underlying OS mechanisms to achieve these goals, for example, scatter-gather I/O operations or moving data across while minimizing costly copies.

Let's start by learning about the basic Boost.Asio blocks, I/O objects, and I/O execution context objects.

I/O objects

Sometimes, an application needs to access OS services, run asynchronous tasks on them, and collect the results or errors. **Boost.Asio** provides a mechanism composed of I/O objects and I/O execution context objects to allow this functionality.

I/O objects are task-oriented objects representing the actual entities performing I/O operations. As we can see in *Figure 9.1*, Boost.Asio provides core classes to manage concurrency, streams, buffers, or other core functionality to the library and also includes portable networking classes for network communications via **Transmission Control Protocol/Internet Protocol** (**TCP/IP**), **User Datagram Protocol** (**UDP**), or **Internet Control Message Protocol** (**ICMP**), classes to define the security layer, the transmission protocol and serial port, among other tasks, and also platform-specific classes to deal with specific settings depending on the underlying OS.

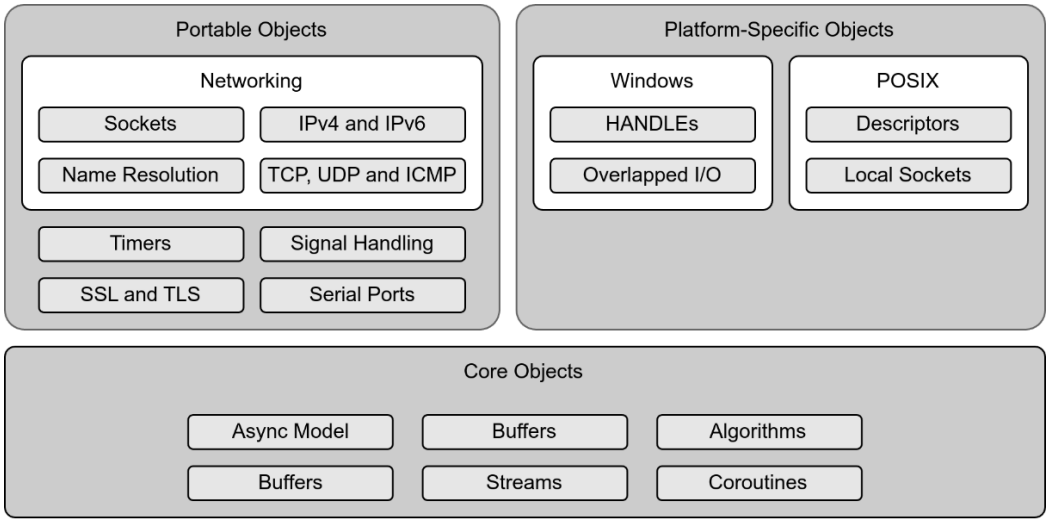


Figure 9.1 – I/O objects

The I/O objects do not directly execute their tasks in the OS. They need to communicate with the OS via an I/O execution context object. An instance of a context object is passed as the first argument in the I/O object constructors. Here, we are defining an I/O object (a timer with an expiration time of three seconds) and passing an I/O execution context object (`io_context`) via its constructor:

```
#include <boost/asio.hpp>
#include <chrono>

using namespace std::chrono_literals;
```

```
boost::asio::io_context io_context;
boost::asio::steady_timer timer(io_context, 3s);
```

Most I/O objects have methods whose name starts with `async_`. These methods trigger asynchronous operations, which will call a completion handler, a callable object passed as an argument to the method when the operation completes. These methods return immediately, not blocking the program flow. The current thread can continue performing other tasks while the task is not complete. Once completed, the completion handler will be called and executed, dealing with the result or error of the asynchronous task.

I/O objects also provide the blocking counterpart methods, which will block until completion. These methods do not need to receive a handler as a parameter.

As mentioned before, note that the I/O objects don't interact directly with the OS; they need an I/O execution context object. Let's learn about this class of objects.

I/O execution context objects

To access the I/O services, the program uses at least one I/O execution context object that represents the gateway to the OS I/O services. It's implemented with the `boost::asio::io_context` class, providing the core I/O functionality of OS services to I/O objects. In Windows, `boost::asio::io_context` is based in **I/O completion ports** (**IOCP**), on Linux, it is based in **epoll**, and on FreeBSD/macOS, it is based in **kqueue**.

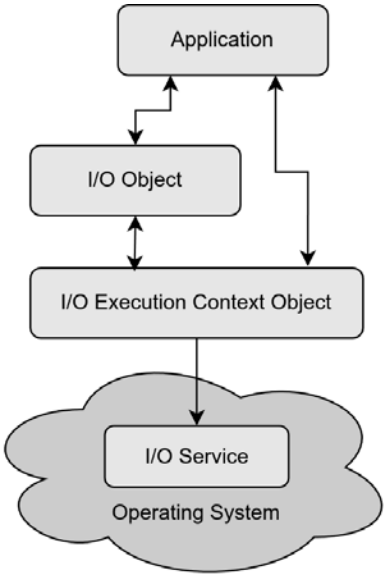


Figure 9.2 – Boost.Asio architecture

`boost::asio::io_context` is a subclass of `boost::asio::execution_context`, a base class for function object execution also inherited by other execution context objects, such as `boost::asio::thread_pool` or `boost::asio::system_context`. In this chapter, we will be using `boost::asio::io_context` as our execution context object.

The `boost::asio::io_context` class has been a replacement for the `boost::asio::io_service` class since version 1.66.0, embracing more modern features and practices from C++. `boost::asio::io_service` is still available for backward compatibility.

As described earlier, Boost.Asio objects can schedule asynchronous operations using methods starting with `async_`. When all the asynchronous tasks are scheduled, the program needs to call the `boost::asio::io_context::run()` function to execute an event processing loop, allowing the OS to deal with the tasks and pass to the program the results and trigger the handlers.

Coming back to our previous example, we will now set up the completion handler, `on_timeout()`, a callable object (in this case a function) that we pass as a parameter when calling the asynchronous `async_wait()` function. Here is the code example:

```
#include <boost/asio.hpp>
#include <iostream>

void on_timeout(const boost::system::error_code& ec) {
    if (!ec) {
        std::cout << "Timer expired.\n" << std::endl;
    } else {
        std::cerr << "Error: " << ec.message() << '\n';
    }
}

int main() {
    boost::asio::io_context io_context;
    boost::asio::steady_timer timer(io_context,
                                    std::chrono::seconds(3));
    timer.async_wait(&on_timeout);
    io_context.run();
    return 0;
}
```

Running this code, we should see the message `Timer expired.` in the console after three seconds, or an error message if the asynchronous call fails for any reason.

`boost::io_context::run()` is a blocking call. This is intended to keep the event loop running, allow the asynchronous operations to run, and prevent the program from exiting. Obviously, this function can be called in a new thread and leave the main thread unblocked to carry on with other tasks, as we have seen in previous chapters.

When there are no pending asynchronous operations, `boost::io_context::run()` will return. There is a template class, `boost::asio::executor_work_guard`, that can keep `io_context` busy and avoid it exiting if needed. Let's see how it works with an example.

Let's start by defining a background task that will wait for two seconds before posting some work through `io_context` using the `boost::asio::io_context::post()` function:

```
#include <boost/asio.hpp>
#include <chrono>
#include <iostream>
#include <thread>

using namespace std::chrono_literals;

void background_task(boost::asio::io_context& io_context) {
    std::this_thread::sleep_for(2s);
    std::cout << "Posting a background task.\n";
    io_context.post([]() {
        std::cout << "Background task completed!\n";
    });
}
```

In the `main()` function, the `io_context` object is created, and a `work_guard` object is constructed using that `io_context` object.

Then, two threads are created, `io_thread`, where `io_context` runs, and `worker`, where `background_task()` will run. We also pass `io_context` as a reference to the background task to post work, as explained earlier.

With that in place, the main thread does some work (waiting for five seconds) and then removes the work guard by calling its `reset()` function, letting `io_context` exit its `run()` function, and joins both threads before exiting, as shown here:

```
int main() {
    boost::asio::io_context io_context;
    auto work_guard = boost::asio::make_work_guard(
                      io_context);

    std::thread io_thread([&io_context]() {
        std::cout << "Running io_context.\n";
        io_context.run();
        std::cout << "io_context stopped.\n";
    });

    std::thread worker(background_task,
```

```
                        std::ref(io_context));

    // Main thread doing some work.
    std::this_thread::sleep_for(5s);
    std::cout << "Removing work_guard." << std::endl;
    work_guard.reset();
    worker.join();
    io_thread.join();
    return 0;
}
```

If we run the previous code, this is the output:

```
Running io_context.
Posting a background task.
Background task completed!
Removing work_guard.
io_context stopped.
```

We can see how the background thread posts a background task correctly, and this is completed before the work guard is removed and the I/O context object stops its execution.

Another way to keep the `io_context` object alive and servicing requests is to provide asynchronous tasks by continuously calling `async_` functions or posting work from the completion handlers. This is a common pattern when reading or writing to sockets or streams:

```
#include <boost/asio.hpp>
#include <chrono>
#include <functional>
#include <iostream>

using namespace std::chrono_literals;

int main() {
    boost::asio::io_context io_context;
    boost::asio::steady_timer timer(io_context, 3s);

    std::function<void(const boost::system::error_code&)>
                  timer_handler;

    timer_handler = [&timer, &timer_handler](
                     const boost::system::error_code& ec) {
        if (!ec) {
            std::cout << "Handler: Timer expired.\n";
            timer.expires_after(1s);
```

```
            timer.async_wait(timer_handler);
        } else {
            std::cerr << "Handler error: "
                      << ec.message() << std::endl;
        }
    };
    timer.async_wait(timer_handler);
    io_context.run();
    return 0;
}
```

In this case, `timer_handler` is the completion handler defined as a lambda function that captures the timer and itself. Every second, when the timer expires, it prints the `Handler: Timer expired.` message and restarts itself by enqueueing a new asynchronous task (using the `async_wait()` function) into the `io_context` object via the `timer` object.

As we have already seen, the `io_context` object can run from any thread. By default, this object is thread-safe, but in some scenarios where we want better performance, we might want to avoid this safety. This can be adjusted during its construction, as we will see in the next section.

Concurrency hints

The `io_context` constructor accepts as an argument a concurrency hint, suggesting to the implementation the number of active threads that should be used for running completion handlers.

By default, this value is `BOOST_ASIO_CONCURRENCY_HINT_SAFE` (value *1*), indicating that the `io_context` object will run from a single thread, enabling several optimizations due to this fact. That doesn't mean that `io_context` can only be used from one thread; it still provides thread safety, and it can use I/O objects from many threads.

Other values that can be specified are as follows:

- `BOOST_ASIO_CONCURRENCY_HINT_UNSAFE`: Disables locking so all operations on `io_context` or I/O objects must occur in the same thread.
- `BOOST_ASIO_CONCURRENCY_HINT_UNSAFE_IO`: Disables locking in the reactor but keeps it in the scheduler, so all operations in the `io_context` object can use different threads apart from the `run()` function and the other methods related to executing the event processing loop. We will learn about schedulers and reactors when explaining the design principles behind the library.

Let's now learn about what the event processing loop is and how to manage it.

The event processing loop

Using the `boost::asio::io_context::run()` method, `io_context` blocks and keeps processing I/O asynchronous tasks until all have been completed and the completion handlers have been notified. This I/O requests processing is done in an internal event processing loop.

There are other methods to control the event loop and avoid blocking until all asynchronous events are processed. These are as follows:

- `poll`: Run the event processing loop to execute ready handlers
- `poll_one`: Run the event processing loop to execute one ready handler
- `run_for`: Run the event processing loop for a specified duration
- `run_until`: Same as the previous one but only until a specified time
- `run_one`: Run the event processing loop to execute at most one handler
- `run_one_for`: Same as the previous one but only for a specified duration
- `run_one_until`: Same as the previous one but only until a specified time

The event loop can also be stopped by calling the `boost::asio::io_context::stop()` method or checking if its status is stopped by calling `boost:asio::io_context::stopped()`.

When the event loop is not running, tasks already being scheduled will continue executing. Other tasks will remain pending. Pending tasks can be resumed and pending results collected by starting the event loop with one of the methods mentioned previously again.

In previous examples, the application sent some work to `io_context` by calling asynchronous methods or by using the `post()` function. Let's learn now about `dispatch()` and its differences with `post()`.

Giving some work to the io_context

Apart from sending work to `io_context` via the asynchronous methods from the different I/O objects or by using `executor_work_guard` (explained below), we can also use the `boost::asio::post()` and `boost::asio::dispatch()` template methods. Both functions are used to schedule some work into an `io_context` object.

The `post()` function guarantees that the task will be executed. It places its completion handler in the execution queue and eventually, it will be executed:

```
boost::asio::io_context io_context;
io_context.post([] {
    std::cout << "This will always run asynchronously.\n";
});
```

On the other hand, `dispatch()` may execute the task immediately if `io_context` or strand (more on strands later in this chapter) are in the same thread where the task is being dispatched, or otherwise placed in the queue for asynchronous execution:

```
boost::asio::io_context io_context;
io_context.dispatch([] {
    std::cout << "This might run immediately or be queued.\n";
});
```

Therefore, using `dispatch()`, we can optimize performance by reducing context switching or queuing delays.

Dispatched events can execute directly from the current worker thread even if there are other pending events queued up. The posted events must always need to be managed by the I/O execution context, waiting until other handlers complete before being allowed to be executed.

Now that we have already learned about some basic concepts, let's learn how synchronous and asynchronous operations work under the hood.

Interacting with the OS

Boost.Asio can interact with I/O services using synchronous and asynchronous operations. Let's learn how they behave and what the main differences are.

Synchronous operations

If the program wants to use an I/O service in a synchronous way, usually, it will create an I/O object and use its synchronous operation method:

```
boost::asio::io_context io_context;
boost::asio::steady_timer timer(io_context, 3s);
timer.wait();
```

When calling `timer.wait()`, the request is sent to the I/O execution context object (`io_context`), which calls the OS to perform the operation. Once the OS finishes with the task, it returns the result to `io_context`, which then translates the result, or an error if anything went wrong, back to the I/O object (`timer`). Errors are of type `boost::system::error_code`. If an error occurs, an exception is thrown.

If we don't want exceptions to be thrown, we can pass an error object by reference to the synchronous method to capture the status of the operation and check it afterward:

```
boost::system::error_code ec;
Timer.wait(server_endpoint, ec);
```

Asynchronous operations

In the case of asynchronous operations, we need to also pass a completion handler to the asynchronous method. This completion handler is a callable object that will be invoked by the I/O context object when the asynchronous operation finishes, notifying the program about the result or operation error. Its signature is as follows:

```
void completion_handler(
     const boost::system::error_code& ec);
```

Continuing with the timer example, now, we need to call the asynchronous operation:

```
socket.async_wait(completion_handler);
```

Again, the I/O object (`timer`) forwards the request to the I/O execution context object (`io_context`). `io_context` requests to the OS to start the asynchronous operation.

When the operation is finished, the OS places the result in a queue, where `io_context` is listening. Then, `io_context` dequeues the result, translates the error into an error code object, and triggers the completion handler to notify the program about the completion of the task and the result.

To allow `io_context` to follow these steps, the program must execute `boost::asio::io_context::run()` (or similar functions introduced earlier that manage the event processing loop) and block the current thread while processing any unfinished asynchronous operation. As already commented, if there are no pending asynchronous operations, `boost::asio::io_context::run()` exits.

Completion handlers are required to be copy-constructible, meaning that a copy-constructor must be available. If a temporary resource is needed (such as memory, thread, or file descriptor), this resource is released before calling the completion handler. That allows us to call the same operation without overlapping resource usage, avoiding increasing the peak resource usage in the system.

Error handling

As mentioned previously, **Boost.Asio** allows users to handle errors in two different ways: by using error codes or throwing exceptions. If we pass a reference to a `boost::system::error_code` object when calling an I/O object method, the implementation will pass errors through that variable; otherwise, an exception will be thrown.

We already implemented some examples following the first approach by checking the error codes. Let's now see how to catch exceptions.

The following example creates a timer with an expiration period of three seconds. The `io_context` object is running from the background thread, `io_thread`. When the timer starts the asynchronous task by calling its `async_wait()` function, it passes the `boost::asio::use_future` argument so the function returns a future object, `fut`, that later is used inside a try-catch block to call its `get()` function and retrieve the stored result or exception, as we learned in *Chapter 6*. After starting the

asynchronous operation, the main thread waits for one second and the timer cancels the operation by calling its `cancel()` function. As this happens before its expiration time (three seconds), an exception is thrown:

```
#include <boost/asio.hpp>
#include <chrono>
#include <future>
#include <iostream>
#include <thread>

using namespace std::chrono_literals;

int main() {
    boost::asio::io_context io_context;
    boost::asio::steady_timer timer(io_context, 1s);

    auto fut = timer.async_wait(
                    boost::asio::use_future);

    std::thread io_thread([&io_context]() {
                        io_context.run();
    });

    std::this_thread::sleep_for(3s);

    timer.cancel();

    try {
        fut.get();
        std::cout << "Timer expired successfully!\n";
    } catch (const boost::system::system_error& e) {
        std::cout << "Timer failed: "
                  << e.code().message() << '\n';
    }
    io_thread.join();
    return 0;
}
```

The exception of type `boost::system::system_error` is caught, and its message is printed. If the timer cancels its operation after the asynchronous operation completes (in this example, by sleeping the main thread for more than three seconds), the timer expires successfully, and no exception is thrown.

Now that we have seen the main building blocks of Boost.Asio and how they interact together, let's recap and understand the design patterns behind its implementation.

The Reactor and Proactor design patterns

When using event handling applications, we can follow two approaches to designing the concurrent solution: the Reactor and Proactor design patterns.

These patterns describe the mechanisms followed to process events, indicating how these are initiated, received, demultiplexed, and dispatched. As the system collects and queues the I/O events coming from different resources, demultiplexing these events means separating them to be dispatched to their correct handlers.

The **Reactor pattern** demultiplexes and dispatches synchronously and serially service requests. It usually follows a non-blocking synchronous I/O strategy, returning the result if the operation can be executed, or an error if the system has no resources to complete the operation.

On the other hand, the **Proactor pattern** allows demultiplexing and dispatching service requests in an efficient asynchronous way by immediately returning the control to the caller, indicating that the operation has been initiated. Then, the called system will notify the caller when the operation is complete.

Thus, the Proactor pattern distributes responsibilities among two tasks: the long-duration operations that are executed asynchronously and the completion handlers that process the results and usually invoke other asynchronous operations.

Boost.Asio implements the Proactor design pattern by using the following elements:

- **Initiator**: An I/O object that initiates the asynchronous operation.
- **Asynchronous operation**: A task to run asynchronously by the OS.
- **Asynchronous operation processor**: This executes the asynchronous operation and queues results in the completion event queue.
- **Completion event queue**: An event queue where the asynchronous operation processor pushes events, and the asynchronous event dequeues them.
- **Asynchronous event demultiplexer**: This blocks the I/O context, waiting for events, and returning completed events to the caller.
- **Completion handler**: A callable object that will process the results of the asynchronous operation.
- **Proactor**: This calls the asynchronous event demultiplexer to dequeue events and dispatch them to the completion handler. This is what the I/O execution context does.

Figure 9.3 clearly shows the relationship between all these elements:

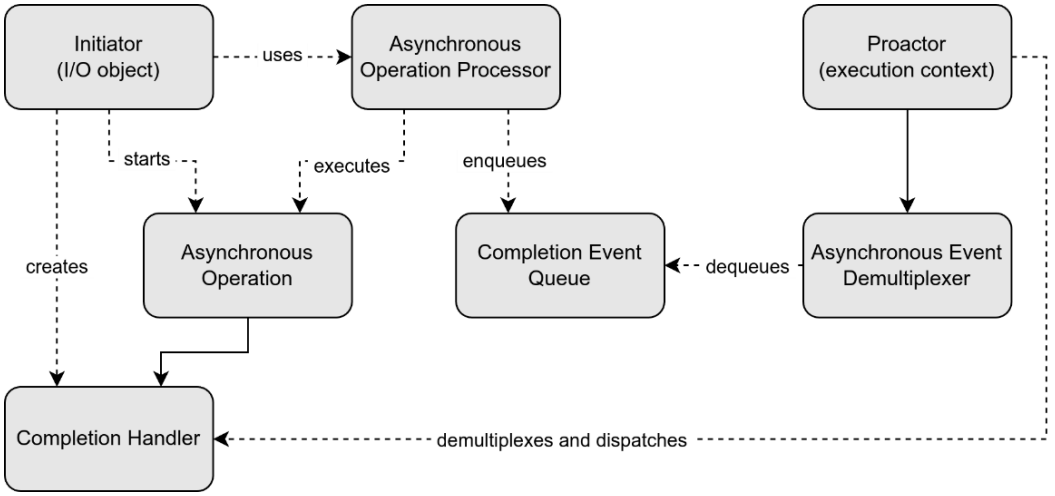


Figure 9.3 – Proactor design pattern

The Proactor pattern increases the separation of concerns at the same time as encapsulating concurrency mechanisms, simplifying application synchronization, and increasing performance.

On the other hand, we have no control over how or when the asynchronous operations are scheduled or how efficiently the OS will perform these operations. Also, there is an increase in memory usage due to the completion event queue and increased complexity in debugging and testing.

Another aspect of the design of Boost.Asio is the thread safety of the execution context objects. Let's now dig into how threading works with Boost.Asio.

Threading with Boost.Asio

I/O execution context objects are thread-safe; their methods can be called from different threads safely. That means that we can use a separate thread to run the blocking `io_context.run()` method and leave the main thread unblocked to carry on with other unrelated tasks.

Let's now explain the different ways to configure the asynchronous application in terms of how to use threads.

Single-threaded approach

The starting point and preferred solution for any **Boost.Asio** application should follow a single-threaded approach where the I/O execution context object runs in the same thread where the completion handlers are being processed. These handlers must be short and non-blocking. Here is an example of a steady timer completion handler running in the same thread as the I/O context, the main thread:

```
#include <boost/asio.hpp>
#include <chrono>
#include <iostream>

using namespace std::chrono_literals;

void handle_timer_expiry(
            const boost::system::error_code& ec) {
    if (!ec) {
        std::cout << "Timer expired!\n";
    } else {
        std::cerr << "Error in timer: "
                  << ec.message() << std::endl;
    }
}

int main() {
    boost::asio::io_context io_context;
    boost::asio::steady_timer timer(io_context,
                                    std::chrono::seconds(1));
    timer.async_wait(&handle_timer_expiry);
    io_context.run();
    return 0;
}
```

As we can see, the `steady_timer` timer calls the asynchronous `async_wait()` function, setting up the `handle_timer_expiry()` completion handler, in the same thread that the `io_context.run()` function is being executed in. When the asynchronous function finishes, its completion handler will run in the same thread.

As the completion handler is running in the main thread, its execution should be quick to avoid freezing the main thread and other relevant tasks that the program should perform. In the next section, we will learn how to deal with long-running tasks or completion handlers and keep the main thread responsive.

Threaded long-running tasks

For long-running tasks, we can keep the logic in the main thread but use other threads to pass work and get results back to the main thread:

```
#include <boost/asio.hpp>
#include <iostream>
#include <thread>

void long_running_task(boost::asio::io_context& io_context,
                       int task_duration) {
    std::cout << "Background task started: Duration = "
              << task_duration << " seconds.\n";
    std::this_thread::sleep_for(
                      std::chrono::seconds(task_duration));
    io_context.post([&io_context]() {
        std::cout << "Background task completed.\n";
        io_context.stop();
    });
}

int main() {
    boost::asio::io_context io_context;

    auto work_guard = boost::asio::make_work_guard
                                        (io_context);

    io_context.post([&io_context]() {
        std::thread t(long_running_task,
                      std::ref(io_context), 2);
        std::cout << "Detaching thread" << std::endl;
        t.detach();
    });

    std::cout << "Running io_context...\n";
    io_context.run();
    std::cout << "io_context exit.\n";
    return 0;
}
```

In this example, after `io_context` is created, a work guard is used to avoid the `io_context.run()` function to immediately return before any work is posted.

The posted work consists of a `t` thread being created to run the `long_running_task()` function in the background. That `t` thread is detached before the lambda function exits; otherwise, the program would terminate.

In the background task function, the current thread sleeps for a given period and then posts another task into the `io_context` object to print a message and stop `io_context` itself. If we don't call `io_context.stop()`, the event processing loop will continue running forever and the program will not finish, as `io_context.run()` will continue blocking due to the work guard.

Multiple I/O execution context objects, one per thread

This approach is like the single-threaded one, where each thread has its own `io_context` object and processes short and non-blocking completion handlers:

```
#include <boost/asio.hpp>
#include <chrono>
#include <iostream>
#include <syncstream>
#include <thread>

#define sync_cout std::osyncstream(std::cout)

using namespace std::chrono_literals;

void background_task(int i) {
    sync_cout << "Thread " << i << ": Starting...\n";
    boost::asio::io_context io_context;
    auto work_guard =
              boost::asio::make_work_guard(io_context);

    sync_cout << "Thread " << i << ": Setup timer...\n";
    boost::asio::steady_timer timer(io_context, 1s);
    timer.async_wait(
        [&](const boost::system::error_code& ec) {
            if (!ec) {
                sync_cout << "Timer expired successfully!"
                          << std::endl;
            } else {
                sync_cout << "Timer error: "
                          << ec.message() << ,\n';
        }
        work_guard.reset();
    });
```

```
    sync_cout << "Thread " << i << ": Running
                      io_context...\n";
    io_context.run();
}

int main() {
    const int num_threads = 4;
    std::vector<std::jthread> threads;

    for (auto i = 0; i < num_threads; ++i) {
        threads.emplace_back(background_task, i);
    }
    return 0;
}
```

In this example, four threads are created, each one running the `background_task()` function where an `io_context` object is created, and a timer is set up to timeout after one second together with its completion handler.

Multiple threads with a single I/O execution context object

Now, there is only one `io_context` object but it is starting the asynchronous tasks from different I/O objects from different threads. In this case, the completion handlers can be called from any of those threads. Here is an example:

```
#include <boost/asio.hpp>
#include <chrono>
#include <iostream>
#include <syncstream>
#include <thread>
#include <vector>

#define sync_cout std::osyncstream(std::cout)

using namespace std::chrono_literals;

void background_task(int task_id) {
    boost::asio::post([task_id]() {
        sync_cout << "Task " << task_id
                  << " is being handled in thread "
                  << std::this_thread::get_id()
                  << std::endl;
        std::this_thread::sleep_for(2s);
        sync_cout << "Task " << task_id
```

```
                    << " complete.\n";
    });
}

int main() {
    boost::asio::io_context io_context;
    auto work_guard = boost::asio::make_work_guard(
                                    io_context);

    std::jthread io_context_thread([&io_context]() {
        io_context.run();
    });

    const int num_threads = 4;
    std::vector<std::jthread> threads;
    for (int i = 0; i < num_threads; ++i) {
        background_task(i);
    }

    std::this_thread::sleep_for(5s);
    work_guard.reset();

    return 0;
}
```

In this example, only one `io_context` object is created and run in a separate thread, `io_context_thread`. Then, an additional four background threads are created, where work is posted into the `io_context` object. Finally, the main thread waits for five seconds to let all threads finish their work and resets the work guard, letting the `io_context.run()` function return if there is no more pending work. When the program exits, all threads automatically join, as they are instances of `std::jthread`.

Parallelizing work done by one I/O execution context

In the previous example, a unique I/O execution context object was used with its `run()` function being called from different threads. Then, each thread posted some work that completion handlers were executing in available threads at the time of completion.

This is a common way to parallelize work done by one I/O execution context, by calling its `run()` function from multiple threads, distributing the processing of asynchronous operations across those threads. This is possible because the `io_context` object provides a thread-safe event dispatching system.

Here is another example where a pool of threads is created, with each thread running `io_context.run()`, making these threads compete to pull tasks from the queue and execute them. In this case, only one asynchronous task is created using a timer that expires in two seconds. One of the threads will pick up the task and execute it:

```
#include <boost/asio.hpp>
#include <iostream>
#include <thread>
#include <vector>

using namespace std::chrono_literals;

int main() {
    boost::asio::io_context io_context;

    boost::asio::steady_timer timer(io_context, 2s);
    timer.async_wait(
        [](const boost::system::error_code& /*ec*/) {
            std::cout << "Timer expired!\n";
    });

    const std::size_t num_threads =
                std::thread::hardware_concurrency();
    std::vector<std::thread> threads;
    for (std::size_t i = 0;
         i < std::thread::hardware_concurrency(); ++i) {
            threads.emplace_back([&io_context]() {
                io_context.run();
            });
    }
    for (auto& t : threads) {
        t.join();
    }
    return 0;
}
```

This technique improves scalability, as the application better utilizes multiple cores, and reduces latency by handling asynchronous tasks concurrently. Also, contention can be reduced and throughput increased by reducing bottlenecks generated when single-threaded code processes many simultaneous I/O operations.

Note that the completion handlers also must use synchronization primitives and be thread-safe if they are shared across different threads or modify shared resources.

Also, there is no guarantee in the order the completion handlers will be executed. As many threads can run simultaneously, any of them can complete earlier and call its associated completion handler.

As threads are competing to pull tasks from the queue, there might be potential lock contention or context-switching overhead if the thread pool size is not optimal, ideally matching the number of hardware threads, as done in this example.

Now, it's time to understand how the objects' lifetime can affect the stability of our asynchronous programs developed with Boost.Asio.

Managing objects' lifetime

One of the main disastrous issues that can happen with asynchronous operations is that, when the operation takes place, some of the required objects have been destroyed. Therefore, managing objects' lifetimes is crucial.

In C++, an object's lifetime begins when the constructor ends and ends when the destructor begins.

A common pattern used to keep objects alive is to let the object create a shared pointer instance to itself, ensuring that the object remains valid as long as there are shared pointers pointing to it, meaning that there are ongoing asynchronous operations needing that object.

This technique is called `shared-from-this` and uses the `std::enable_shared_from_this` template base class, available since C++11, which provides the `shared_from_this()` method used by the object to obtain a shared pointer to itself.

Implementing an echo server – an example

Let's see how it works by creating an echo server. At the same time, we will be discussing this technique, we will also be learning about how to use Boost.Asio for networking.

Transmission of data over a network can take a long time to complete, and several errors can occur. That makes network I/O services a special good case to be dealt with by Boost.Asio. Network I/O services were the first services to be included in the library.

The main common usage of Boost.Asio in the industry is to develop networking applications due to its support for the internet protocols TCP, UDP, and ICMP. The library also provides a socket interface based on the **Berkeley Software Distribution** (**BSD**) socket API to allow the development of efficient and scalable applications using a low-level interface.

However, as, in this book, we are interested in asynchronous programming, let's focus on implementing an echo server using a high-level interface.

An echo server is a program that listens to a specific address and port and writes back everything that it reads from that port. For that purpose, we will create a TCP server.

The main program will simply create an `io_context` object, set up the `EchoServer` object by passing the `io_context` object and a port number to listen from, and call `io_context.run()` to start the event processing loop:

```
#include <boost/asio.hpp>
#include <memory>

constexpr int port = 1234;

int main() {
    try {
        boost::asio::io_context io_context;
        EchoServer server(io_context, port);
        io_context.run();
    } catch (std::exception& e) {
        std::cerr << "Exception: " << e.what() << "\n";
    }
    return 0;
}
```

When `EchoServer` initializes, it will start listening for incoming connections. It does that by using a `boost::asio::tcp::acceptor` object. This object accepts via its constructor an `io_context` object (as usual for I/O objects) and a `boost::asio::tcp::endpoint` object, which indicates the connection protocol and port number used for listening. As a `boost::asio::tcp::v4()` object is used to initialize the endpoint object, the protocol that the `EchoServer` will use is IPv4. The IP address is not specified to the endpoint constructor, therefore the endpoint IP address will be *any address* (`INADDR_ANY` for IPv4 or `in6addr_any` for IPv6). Next, the code implementing the `EchoServer` constructor is as follows:

```
using boost::asio::ip::tcp;

class EchoServer {
   public:
    EchoServer(boost::asio::io_context& io_context,
               short port)
        : acceptor_(io_context,
                    tcp::endpoint(tcp::v4(),
                    port)) {
        do_accept();
    }

   private:
    void do_accept() {
        acceptor_.async_accept([this](
```

```
                    boost::system::error_code ec,
                    tcp::socket socket) {
            if (!ec) {
                std::make_shared<Session>(
                    std::move(socket))->start();
            }
            do_accept();
        });
    }

    tcp::acceptor acceptor_;
};
```

The `EchoServer` constructor calls the `do_accept()` function after setting up the acceptor object. The `do_accept()` function calls the `async_accept()` function waiting for incoming connections. When a client connects to the server, the OS returns the connection's socket (`boost::asio::tcp::socket`) or an error via the `io_context` object.

If there is no error and a connection is established, a shared pointer of a `Session` object is created, moving the socket into the `Session` object. Then, the `Session` object runs the `start()` function.

The `Session` object encapsulates the state of a particular connection, in this case, the `socket_` object and the `data_` buffer. It also manages asynchronous reads and writes into that buffer by using `do_read()` and `do_write()`, which we will implement in a moment. But before this, comment that `Session` inherits from `std::enable_shared_from_this<Session>`, allowing `Session` objects to create shared pointers to themselves, ensuring that the session objects remain alive throughout the lifetime of asynchronous operations needing them, as long as there is at least one shared pointer pointing to a `Session` instance managing that connection. This shared pointer is the one created in the `do_accept()` function in the `EchoServer` object when the connection was established. Here is the implementation of the `Session` class:

```
class Session
    : public std::enable_shared_from_this<Session>
{
   public:
    Session(tcp::socket socket)
        : socket_(std::move(socket)) {}

    void start() { do_read(); }

   private:
    static const size_t max_length = 1024;

    void do_read();
```

```
    void do_write(std::size_t length);

    tcp::socket socket_;
    char data_[max_length];
};
```

Using a `Session` class allows us to separate the logic that manages the connection from the one that manages the server. `EchoServer` just needs to accept connections and create a `Session` object per connection. That way, a server can manage multiple clients, keeping their connections independent and asynchronously managed.

`Session` is the one that manages the behavior of that connection using the `do_read()` and `do_write()` functions. When `Session` starts, its `start()` function calls the `do_read()` function, as shown here:

```
void Session::do_read() {
    auto self(shared_from_this());
    socket_.async_read_some(boost::asio::buffer(data_,
                                                max_length),
        [this, self](boost::system::error_code ec,
                     std::size_t length) {
            if (!ec) {
                do_write(length);
            }
        });
}
```

The `do_read()` function creates a shared pointer to the current session object (`self`) and uses the socket's `async_read_some()` asynchronous function to read some data into the `data_` buffer. If successful, this operation returns the data copied into the `data_` buffer and the number of read bytes in the `length` variable.

Then, `do_write()` is called with that `length` variable, asynchronously writing the content of the `data_` buffer into the socket by using the `async_write()` function. When this asynchronous operation succeeds, it restarts the cycle by calling again the `do_read()` function, as shown here:

```
void Session::do_write(std::size_t length) {
    auto self(shared_from_this());
    boost::asio::async_write(socket_,
                             boost::asio::buffer(data_,
                                                 length),
        [this, self](boost::system::error_code ec,
                     std::size_t length) {
            if (!ec) {
                do_read();
```

```
            }
        });
}
```

You might wonder why it is that `self` is defined but is not being used. It looks like `self` is redundant, but as the lambda function is capturing it by value, a copy is being created, increasing the reference count of the shared pointer to the `this` object, ensuring that the session will not be destroyed if the lambda is active. The `this` object is captured to provide access to its members into the lambda function.

As an exercise, try to implement a `stop()` function that breaks the cycle between `do_read()` and `do_write()`. Once all asynchronous operations are complete and the lambda functions exit, the `self` objects will be destroyed and there will be no other shared pointers pointing to the `Session` object, thus the session will be destroyed.

This pattern ensures robust and safe management of objects' lifetimes during asynchronous operations, avoiding dangling pointers or early destruction, which would lead to undesired behavior or crashes.

To test this server, just start the server, open a new terminal, and use the `telnet` command to connect to the server and send data to it. As arguments, we can pass the `localhost` address, indicating that we are connecting to a server running on the same machine (IP address of `127.0.0.1`) and the port, in this case, `1234`.

The `telnet` command will start and show some information about the connection and indicate that we need to hit the *Ctrl + }* keys to close the connection.

Typing anything and hitting the *Enter* key will send that entered line to the echo server, which will listen and send back the same content; in this example, it will be `Hello world!`.

Just close the connection and exit `telnet` by using the `quit` command to exit back to the terminal:

```
$ telnet localhost 1234
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Hello world!
Hello world!
telnet> quit
Connection closed.
```

In this example, we have already used a buffer. Let's learn a bit more about them in the next section.

Transferring data using buffers

Buffers are contiguous regions of memory used during I/O operations to transfer data.

Boost.Asio defines two types of buffers: **mutable buffers** (`boost::asio::mutable_buffer`), where data can be written, and **constant buffers** (`boost::asio::const_buffers`), which are used to create read-only buffers. Mutable buffers can be converted into constant buffers, but not the opposite. Both types of buffers provide protection against overruns.

There is also the `boost::buffer` function to help with the creation of mutable or constant buffers from different data types (a pointer to raw memory and size, a string (`std::string`), or an array or vector of **plain old data** (**POD**) structures (meaning a type, a structure, or a class that has no user-defined copy assignment operator or destructor, and without private or protected non-static data members). For example, to create a buffer from an array of chars, we can use the following:

```
char data[1024];
mutable_buffer buffer = buffer(data, sizeof(data));
```

Also, note that the buffer's ownership and lifetime are the responsibility of the program, not the Boost.Asio library.

Scatter-gather operations

Buffers can be used efficiently by using scatter-gather operations where multiple buffers are used together to receive data (scatter-read) or to send data (gather-write).

Scatter-read is the process of reading data from a unique source into different non-contiguous memory buffers.

Gather-write is the opposite process; data is gathered from different non-contiguous memory buffers and written into a single destination.

These techniques increase efficiency and performance as they reduce the number of system calls or data copying. They are not only used for I/O operations but also in other use cases, such as data processing, machine learning, or parallel algorithms, such as sorting or matrix multiplication.

To allow scatter-gather operations, several buffers can be passed together to the asynchronous operation inside a container (`std::vector`, `std::list`, `std::array`, or `boost::array`).

Here is an example of scatter-read where a socket reads some data asynchronously into both the `buf1` and `buf2` buffers:

```
std::array<char, 128> buf1, buf2;
std::vector<boost::asio::mutable_buffer> buffers = {
    boost::asio::buffer(buf1),
    boost::asio::buffer(buf2)
};
socket.async_read_some(buffers, handler);
```

Here is how to achieve a gather-read:

```
std::array<char, 128> buf1, buf2;
std::vector<boost::asio::const_buffer> buffers = {
    boost::asio::buffer(buf1),
    boost::asio::buffer(buf2)
};
socket.async_write_some(buffers, handler);
```

Now, the socket does the opposite operation, writing some data from both buffers into the socket buffer for asynchronous sending.

Stream buffers

We can also use stream buffers to manage data. **Stream buffers** are defined by the `boost::asio::basic_streambuf` class, based in the `std::basic_streambuf` C++ class and defined in the `<streambuf>` header file. It allows a dynamic buffer where its size can adapt to the amount of data being transferred.

Let's see in the following example how stream buffers work together with scatter-gather operations. In this case, we are implementing a TCP server that listens and accepts clients' connections from a given port, reads the messages sent by the clients into two stream buffers, and prints their content to the console. As we are interested in understanding stream buffers and scatter-gather operations, let's simplify the example by using synchronous operations.

As in the previous example, in the `main()` function, we use a `boost::asio::ip::tcp::acceptor` object to set up the protocol and port that the TCP server will use to accept connections. Then, in an infinite loop, the server uses that acceptor object to attach a TCP socket (`boost::asio::ip::tcp::socket`) and call the `handle_client()` function:

```
#include <array>
#include <iostream>
#include <boost/asio.hpp>
#include <boost/asio/streambuf.hpp>

using boost::asio::ip::tcp;

constexpr int port = 1234;

int main() {
    try {
        boost::asio::io_context io_context;
        tcp::acceptor acceptor(io_context,
                               tcp::endpoint(tcp::v4(), port));
```

```
        std::cout << "Server is running on port "
                  << port << "...\n";

        while (true) {
            tcp::socket socket(io_context);
            acceptor.accept(socket);
            std::cout << "Client connected...\n";

            handle_client(socket);
            std::cout << "Client disconnected...\n";
        }
    } catch (std::exception& e) {
        std::cerr << "Exception: " << e.what() << '\n';
    }
    return 0;
}
```

The `handle_client()` function creates two stream buffers: `buf1` and `buf2`, and adds them to a container, in this case, `std::array`, to be used in scatter-gather operations.

Then, the synchronous `read_some()` function from the socket is called. This function returns the number of bytes read from the socket and copies them into the buffers. If anything goes wrong with the socket connection, an error will be returned in the error code object, `ec`. In that case, the server will print the error message and exit.

Here is the implementation:

```
void handle_client(tcp::socket& socket) {
    const size_t size_buffer = 5;
    boost::asio::streambuf buf1, buf2;

    std::array<boost::asio::mutable_buffer, 2> buffers = {
        buf1.prepare(size_buffer),
        buf2.prepare(size_buffer)
    };

    boost::system::error_code ec;
    size_t bytes_recv = socket.read_some(buffers, ec);
    if (ec) {
        std::cerr << "Error on receive: "
                  << ec.message() << '\n';
        return;
    }
```

```
    std::cout << "Received " << bytes_recv << " bytes\n";

    buf1.commit(5);
    buf2.commit(5);

    std::istream is1(&buf1);
    std::istream is2(&buf2);
    std::string data1, data2;
    is1 >> data1;
    is2 >> data2;

    std::cout << "Buffer 1: " << data1 << std::endl;
    std::cout << "Buffer 2: " << data2 << std::endl;
}
```

If there are no errors, the stream buffers' `commit()` function is used to transfer five bytes to each of the stream buffers, `buf1` and `buf2`. The contents of these buffers are extracted by using `std::istream` objects and printed to the console.

To execute this example, we need to open two terminals. In one terminal, we execute the server, and in the other, the `telnet` command, as shown earlier. In the `telnet` terminal, we can type a message (for example, *Hello World*). This message is sent to the server. The server terminal will then show the following:

```
Server is running on port 1234...
Client connected...
Received 10 bytes
Buffer 1: Hello
Buffer 2: Worl
Client disconnected...
```

As we can see, only 10 bytes are processed and distributed into the two buffers. The space character between the two words is processed but discarded when parsing the input by the `iostream` objects.

Stream buffers are useful when the size of the incoming data is variable and unknown in advance. These types of buffers can be used together with fixed-sized buffers.

Signal handling

Signal handling allows us to catch signals sent by the OS and gracefully shut down the application before the OS decides to kill the application's process.

Boost.Asio provides the `boost::asio::signal_set` class for this purpose, which starts an asynchronous wait for one or more signals to occur.

This is an example of how to handle the `SIGINT` and `SIGTERM` signals:

```
#include <boost/asio.hpp>
#include <iostream>

int main() {
    try {
        boost::asio::io_context io_context;
        boost::asio::signal_set signals(io_context,
                                    SIGINT, SIGTERM);

        auto handle_signal = [&](
                const boost::system::error_code& ec,
                int signal) {
            if (!ec) {
                std::cout << "Signal received: "
                          << signal << std::endl;

                // Code to perform cleanup or shutdown.
                io_context.stop();
            }
        };

        signals.async_wait(handle_signal);
        std::cout << "Application is running. "
                  << "Press Ctrl+C to stop...\n";

        io_context.run();
        std::cout << "Application has exited cleanly.\n";
    } catch (std::exception& e) {
        std::cerr << "Exception: " << e.what() << '\n';
    }
    return 0;
}
```

The `signals` object is `signal_set`, listing the signals that the program waits for, `SIGINT` and `SIGTERM`. This object has an `async_wait()` method that asynchronously waits for any of those signals to happen and triggers the completion handler, `handle_signal()`.

As usual in completion handlers, `handle_signal()` checks the error code, `ec`, and if there is no error, some cleanup code might execute to cleanly and gracefully exit the program. In this example, we just stop the event processing loop by calling `io_context.stop()`.

We could also wait synchronously for signals by using the `signals.wait()` method.

If the application is multithreaded, the signals event handler must run in the same thread as the `io_context` object, typically being the main thread.

In the next section, we will learn how to cancel operations.

Canceling operations

Some I/O objects, such as sockets or timers, have object-wide cancellation of outstanding asynchronous operations by calling their `close()` or `cancel()` methods. If an asynchronous operation is canceled, the completion handler will receive an error with the `boost::asio::error::operation_aborted` code.

In the following example, a timer is created, and its timeout period is set to five seconds. But after sleeping the main thread for only two seconds, the timer is canceled by calling its `cancel()` method, making the completion handler be called with a `boost::asio::error::operation_aborted` error code:

```
#include <boost/asio.hpp>
#include <chrono>
#include <iostream>
#include <thread>

using namespace std::chrono_literals;

void handle_timeout(const boost::system::error_code& ec) {
    if (ec == boost::asio::error::operation_aborted) {
        std::cout << "Timer canceled.\n";
    } else if (!ec) {
        std::cout << "Timer expired.\n";
    } else {
        std::cout << "Error: " << ec.message()
                  << std::endl;
    }
}

int main() {
    boost::asio::io_context io_context;
    boost::asio::steady_timer timer(io_context, 5s);

    timer.async_wait(handle_timeout);

    std::this_thread::sleep_for(2s);
    timer.cancel();
```

```
    io_context.run();
    return 0;
}
```

But if we want a per-operation cancellation, we need to set up a cancellation slot that will be triggered when a cancellation signal is emitted. This cancellation signal/slot pair composes a lightweight channel to communicate cancellation operations, like the ones created between promises and futures explained in *Chapter 6*. The cancellation framework has been available in Boost.Asio since version 1.75.

This approach enables a more flexible cancellation mechanism where multiple operations can be canceled using the same signal, and it integrates seamlessly with Boost.Asio's asynchronous operations. Synchronous operations can only be canceled by using the `cancel()` or `close()` methods described earlier; they are not supported by the cancellation slots mechanism.

Let's modify the previous example and use a cancellation signal/slot to cancel the timer. We only need to modify the way the timer is canceled in the `main()` function. Now, when the asynchronous `async_wait()` operation is executed, a cancellation slot is created by binding a slot from the cancellation signal and the completion handler using the `boost::asio::bind_cancellation_slot()` function.

As before, the timer has an expiration period of five seconds, and again, the main thread only sleeps for two seconds. This time, a cancellation signal is emitted by calling the `cancel_signal.emit()` function. The signal will trigger the counterpart cancellation slot and execute the completion handler with a `boost::asio::error::operation_aborted` error code, printing in the console the `Timer canceled.` message; see the following:

```
int main() {
    boost::asio::io_context io_context;
    boost::asio::steady_timer timer(io_context, 5s);

    boost::asio::cancellation_signal cancel_signal;

    timer.async_wait(boost::asio::bind_cancellation_slot(
        cancel_signal.slot(),
        handle_timeout
    ));

    std::this_thread::sleep_for(2s);
    cancel_signal.emit(
        boost::asio::cancellation_type::all);

    io_context.run();
    return 0;
}
```

When the signal is emitted, a cancellation type must be specified, letting the target operation know what the application requires and the operation guarantees, thus controlling the scope and behavior of the cancellation.

The various categories of cancellation are as follows:

- `None`: No cancellation is performed. It can be useful if we want to test if a cancellation should occur.
- `Terminal`: The operation has unspecified side effects so the only safe way to cancel the operation is to close or destroy the I/O object, being its result final, for example, completing a task or transaction.
- `Partial`: The operation has well-defined side effects so the completion handler can take the required actions to resolve the issue, meaning that the operation is partially completed and can be resumed or retried.
- `Total` or `All`: The operation has no side effects. Cancels both terminal and partial operations, enabling a comprehensive cancellation by stopping all ongoing asynchronous operations.

If the cancellation type is not supported by the asynchronous operation, the cancellation request is discarded. For example, timer operations support all categories of cancellation, but sockets only support `Total` and `All`, meaning that if we try to cancel a socket asynchronous operation with a `Partial` cancellation, this cancellation will be ignored. This prevents undefined behavior if an I/O system tries to handle an unsupported cancellation request.

Also, cancellation requests made after the operation is initiated but before it starts, or after its completion, have no effect.

Sometimes, we need to run some work sequentially. Next, we will introduce how we can achieve this by using strands.

Serializing workload with strands

A **strand** is a strict sequential and non-concurrent invocation of completion handlers. Using strands, asynchronous operations can be sequenced without explicit locking by using mutexes or other synchronization mechanisms seen earlier in this book. Strands can be implicit or explicit.

As shown earlier in this chapter, if we execute `boost::asio::io_context::run()` from only one thread, all event handlers will execute in an implicit strand, as they will be sequentially queued one by one and triggered from the I/O execution context.

Another implicit strand happens when there are chained asynchronous operations where one asynchronous operation schedules the next asynchronous operation, and so on. Some previous examples in this chapter already used this technique, but here there is another one.

In this case, if there are no errors, the timer keeps restarting itself in the `handle_timer_expiry()` event handler by recursively setting up the expiration time and calling the `async_wait()` method:

```
#include <boost/asio.hpp>
#include <chrono>
#include <iostream>

using namespace std::chrono_literals;

void handle_timer_expiry(boost::asio::steady_timer& timer,
                         int count) {
    std::cout << "Timer expired. Count: " << count
              << std::endl;
    timer.expires_after(1s);
    timer.async_wait([&timer, count](
                const boost::system::error_code& ec) {
        if (!ec) {
            handle_timer_expiry(timer, count + 1);
        } else {
            std::cerr << „Error: „ << ec.message()
                      << std::endl;
        }
    });
}

int main() {
    boost::asio::io_context io_context;
    boost::asio::steady_timer timer(io_context, 1s);
    int count = 0;
    timer.async_wait([&](
                const boost::system::error_code& ec) {
        if (!ec) {
            handle_timer_expiry(timer, count);
        } else {
            std::cerr << "Error: " << ec.message()
                      << std::endl;
        }
    });
    io_context.run();
    return 0;
}
```

Running this example would print the `Timer expired. Count: <number>` line every second with the counter increasing on each line.

In case some work needs to be serialized but these approaches are not appropriate, we can use explicit strands by using `boost::asio::strand` or its specialization for I/O context execution objects, `boost::asio::io_context::strand`. Posted work using these strand objects will serialize their handler execution in the order they enter the I/O execution context queue.

In the following example, we will create a logger that serializes writing operations into a single log file from several threads. We will be logging messages from four threads, writing five messages from each. We expect the output to be correct, but this time without using any mutex or other synchronization mechanism.

Let's start by defining the `Logger` class:

```
#include <boost/asio.hpp>
#include <chrono>
#include <fstream>
#include <iostream>
#include <memory>
#include <string>
#include <thread>
#include <vector>

using namespace std::chrono_literals;

class Logger {
   public:
    Logger(boost::asio::io_context& io_context,
           const std::string& filename)
        : strand_(io_context), file_(filename
        , std::ios::out | std::ios::app)
    {
        if (!file_.is_open()) {
            throw std::runtime_error(
                      "Failed to open log file");
        }
    }

    void log(const std::string message) {
        strand_.post([this, message](){
            do_log(message);
        });
    }

   private:
```

```
    void do_log(const std::string message) {
        file_ << message << std::endl;
    }

    boost::asio::io_context::strand strand_;
    std::ofstream file_;
};
```

The `Logger` constructor accepts an I/O context object, used to create a strand object (`boost::asio::io_context::strand`), and `std::string`, specifying a log filename that is used to open the log file or create it if it does not exist. The log file is open for appending new content. If the file is not open before the constructor finishes, meaning that there was an issue when accessing or creating the file, the constructor throws an exception.

The logger also provides the public `log()` function that accepts `std::string`, specifying a message as a parameter. This function uses the strand to post new work into the `io_context` object. It does that by using a lambda function, capturing by value the logger instance (the object `this`) and the message, and calls the private `do_log()` function, where a `std::fstream` object is used to write the message into the output file.

There will be only one instance of the `Logger` class in the program, shared by all threads. That way, the threads will write to the same file.

Let's define a `worker()` function that each thread will run to write `num_messages_per_thread` messages into the output file:

```
void worker(std::shared_ptr<Logger> logger, int id) {
    for (unsigned i=0; i < num_messages_per_thread; ++i) {
        std::ostringstream oss;
        oss << "Thread " << id << " logging message " << i;
        logger->log(oss.str());
        std::this_thread::sleep_for(100ms);
    }
}
```

This function accepts a shared pointer to the `Logger` object and a thread identifier. It prints all the messages using the `Logger`'s public `log()` function explained earlier.

To interleave the threads executions and rigorously test how the strands work, each thread will sleep for 100 ms after writing each message.

Finally, in the `main()` function, we start an `io_context` object and a work guard to avoid an early exit from the `io_context`. Then, a shared pointer to a `Logger` instance is created, passing the necessary parameters explained earlier.

A thread pool (vector of `std::jthread` objects) is created by using the `worker()` function and passing the shared pointer to the logger and a unique identifier for each thread. Also, a thread running the `io_context.run()` function is added to the thread pool.

In the following example, as we know that all messages will be printed out in less than two seconds, we make `io_context` run for only that period, using `io_context.run_for(2s)`.

When the `run_for()` function exits, the program prints `Done!` to the console and finishes:

```
const std::string log_filename = "log.txt";
const unsigned num_threads = 4;
const unsigned num_messages_per_thread = 5;

int main() {
    try {
        boost::asio::io_context io_context;
        auto work_guard = boost::asio::make_work_guard(
                                  io_context);
        std::shared_ptr<Logger> logger =
             std::make_shared<Logger>(
                  io_context, log_filename);

        std::cout << "Logging "
                  << num_messages_per_thread
                  << " messages from " << num_threads
                  << " threads\n";

        std::vector<std::jthread> threads;
        for (unsigned i = 0; i < num_threads; ++i) {
            threads.emplace_back(worker, logger, i);
        }

        threads.emplace_back([&]() {
            io_context.run_for(2s);
        });

    } catch (std::exception& e) {
        std::cerr << "Exception: " << e.what() << '\n';
    }

    std::cout << "Done!" << std::endl;
    return 0;
}
```

Running this example will show the following output:

```
Logging 5 messages from 4 threads
Done!
```

This is the content of the generated `log.txt` log file. As the sleep time for each thread is the same, all threads and messages are sequentially ordered:

```
Thread 0 logging message 0
Thread 1 logging message 0
Thread 2 logging message 0
Thread 3 logging message 0
Thread 0 logging message 1
Thread 1 logging message 1
Thread 2 logging message 1
Thread 3 logging message 1
Thread 0 logging message 2
Thread 1 logging message 2
Thread 2 logging message 2
Thread 3 logging message 2
Thread 0 logging message 3
Thread 1 logging message 3
Thread 2 logging message 3
Thread 3 logging message 3
Thread 0 logging message 4
Thread 1 logging message 4
Thread 2 logging message 4
Thread 3 logging message 4
```

If we remove the work guard, the log file only has the following content:

```
Thread 0 logging message 0
Thread 1 logging message 0
Thread 2 logging message 0
Thread 3 logging message 0
```

This happens because the first batch of work is promptly posted and queued into `io_object` from each thread, but `io_object` exits after finishing dispatching the work guard and notifying the completion handlers before the second batch of messages is posted.

If we also remove the `sleep_for()` instruction in the worker thread, now, the log file content is as follows:

```
Thread 0 logging message 0
Thread 0 logging message 1
Thread 0 logging message 2
Thread 0 logging message 3
Thread 0 logging message 4
Thread 1 logging message 0
Thread 1 logging message 1
Thread 1 logging message 2
Thread 1 logging message 3
Thread 1 logging message 4
Thread 2 logging message 0
Thread 2 logging message 1
Thread 2 logging message 2
Thread 2 logging message 3
Thread 2 logging message 4
Thread 3 logging message 0
Thread 3 logging message 1
Thread 3 logging message 2
Thread 3 logging message 3
Thread 3 logging message 4
```

Earlier, the content was sorted by message identifier, and now it's by thread identifier. This is because now, when a thread starts and runs the `worker()` function, it posts all messages at once, without any delay. Therefore, the first thread (thread `0`) enqueues all its work before the second thread has the chance to do that, and so on.

Continuing with further experiments, when we posted content into the strand, we captured the logger instance and the message by value, by using the following instruction:

```
strand_.post([this, message]() { do_log(message); });
```

Capturing by value allows the lambda function running `do_log()` to use a copy of the needed objects, keeping them alive, as commented earlier in this chapter when we discussed object lifetimes.

Say, for some reason, we decided to capture by reference using the following instruction:

```
strand_.post([&]() { do_log(message); });
```

Then, the resulting log file will have incomplete log messages and even incorrect characters because the logger is printing from memory areas that belonged to a message object that no longer exists when the `do_log()` function executes.

Therefore, always assume asynchronous changes; the OS might perform some changes out of our control, so always know what is under our control and, most importantly, what's not.

Finally, instead of using a lambda expression and capturing the `this` and `message` objects by value, we could also use `std::bind` as follows:

```
strand_.post(std::bind(&Logger::do_log, this, message));
```

Let's learn now how we can simplify the echo server we implemented earlier by using coroutines and improving it by adding a command to exit the connection from the client's side.

Coroutines

Boost.Asio has also included support for coroutines since version 1.56.0 and supported native coroutines since version 1.75.0.

As we have learned in the previous chapter, using coroutines simplifies how the program is written as there is no need to add completion handlers and split the flow of the program into different asynchronous functions and callbacks. Instead, with coroutines, the program follows a sequential structure where an asynchronous operation call pauses the execution of the coroutine. When the asynchronous operation completes, the coroutine is resumed, letting the program continue its execution from where it was previously paused.

With newer versions (newer than 1.75.0), we can use native C++ coroutines via `co_await`, to wait for asynchronous operations within a coroutine, `boost::asio::co_spawn` to launch a coroutine, and `boost::asio::use_awaitable` to let Boost.Asio know that an asynchronous operation will use coroutines. With earlier versions (from 1.56.0), coroutines were available using `boost::asio::spawn()` and `yield` contexts. As the newer approach is preferred, not only because it supports native C++20 coroutines, but the code is also more modern, clean, and readable, we will focus on this approach in this section.

Let's implement again the echo server, but this time using Boost.Asio's awaitable interface and coroutines. We will also add some improvements, such as support to close the connection from the client's side when sending the `QUIT` command, showing how to process data or commands on the server side, and stopping handling connections and exiting if any exception is thrown.

Let's start by implementing the `main()` function. The program starts by using `boost::asio::co_spawn` to create a new coroutine-based thread. This function accepts as parameters an execution context (`io_context`, but can also use a strand), a function with the `boost::asio::awaitable<R,E>` return type, which will be used as the coroutine's entry point (the `listener()` function that we will implement and explain next), and a completion token that will be called when the thread has completed. If we want to run the coroutine without being notified of its completion, we can pass the `boost::asio::detached` token.

Finally, we start processing asynchronous events by calling `io_context.run()`.

In case there is any exception, it will be caught by the try-catch block, and the event processing loop will be stopped by calling `io_context.stop()`:

```
#include <boost/asio.hpp>
#include <iostream>
#include <sstream>
#include <string>

using boost::asio::ip::tcp;

int main() {
    boost::asio::io_context io_context;

    try {
        boost::asio::co_spawn(io_context,
                    listener(io_context, 12345),
                    boost::asio::detached);
        io_context.run();

    } catch (std::exception& e) {
        std::cerr << "Error: " << e.what() << std::endl;
        io_context.stop();
    }

    return 0;
}
```

The `listener()` function receives as parameters an `io_context` object and the port number that the listener will accept connections from, using an `acceptor` object as explained earlier. It also must have a return type of `boost::asio::awaitable<R,E>`, where `R` is the return type of the coroutine and `E` is the exception type that might be thrown. In this example, `E` is set as default, so not explicitly specified.

The connection is accepted by calling the `async_accept` acceptor function. As we are now using a coroutine, we need to specify `boost::asio::use_awaitable` to the asynchronous function and use `co_await` to stop the coroutine execution until is resumed when the asynchronous task completes.

When the listener coroutine task resumes, `acceptor.async_accept()` returns a socket object. The coroutine continues by spawning a new thread, using the `boost::asio::co_spawn` function, executing the `echo()` function, and passing the `socket` object to it:

```
boost::asio::awaitable<void> listener(boost::asio::io_context& io_
context, unsigned short port) {
    tcp::acceptor acceptor(io_context,
                           tcp::endpoint(tcp::v4(), port));
    while (true) {
        std::cout << "Accepting connections...\n";
        tcp::socket socket = co_await
                acceptor.async_accept(
                    boost::asio::use_awaitable);

        std::cout << "Starting an Echo "
                  << "connection handler...\n";
        boost::asio::co_spawn(io_context,
                              echo(std::move(socket)),
                              boost::asio::detached);
    }
}
```

The `echo()` function is responsible for handling a single client connection. It must follow a similar signature as the `listener()` function; it needs a return type of `boost::asio::awaitable<R,E>`. As commented earlier, the `socket` object is moved into this function from the listener.

The function asynchronously reads content from the socket and writes it back in an infinite loop that only finishes if it receives the `QUIT` command or an exception is thrown.

Asynchronous reads are done by using the `socket.async_read_some()` function, which reads data into the data buffer using `boost::asio::buffer` and returns the number of bytes read (`bytes_read`). As the asynchronous task is managed by a coroutine, `boost::asio::use_awaitable` is passed to the asynchronous operation. Then, `co_wait` just instructs the coroutine engine to pause the execution until the asynchronous operation finishes.

Once some data is received, the execution of the coroutine resumes, checking if there is really some data to process, otherwise, it finishes the connection by exiting the loop, thus the `echo()` function as well.

If data is read, it converts it into `std::string` for easy manipulation. It removes the `\r\n` ending, if present, and compares the string against `QUIT`.

If `QUIT` is present, it performs an asynchronous write, sends the `Good bye!` message, and exits the loop. Otherwise, it sends the received data back to the client. In both cases, an asynchronous write operation is performed by using the `boost::asio::async_write()` function, passing the socket, `boost:asio::buffer` wrapping the data buffer to send, and `boost::asio::use_awaitable` as with the asynchronous read operation.

Then, `co_await` is used again to suspend the execution of the coroutine while the operation is performed. Once completed, the coroutine will resume and repeat these steps in a new loop iteration:

```
boost::asio::awaitable<void> echo(tcp::socket socket) {
    char data[1024];

    while (true) {
        std::cout << "Reading data from socket...\n";
        std::size_t bytes_read = co_await
                 socket.async_read_some(
                        boost::asio::buffer(data),
                        boost::asio::use_awaitable);

        if (bytes_read == 0) {
            std::cout << "No data. Exiting loop...\n";
            break;
        }

        std::string str(data, bytes_read);
        if (!str.empty() && str.back() == '\n') {
            str.pop_back();
        }
        if (!str.empty() && str.back() == '\r') {
            str.pop_back();
        }

        if (str == "QUIT") {
            std::string bye("Good bye!\n");
            co_await boost::asio::async_write(socket,
                         boost::asio::buffer(bye),
                         boost::asio::use_awaitable);
            break;
        }

        std::cout << "Writing '" << str
                  << "' back into the socket...\n";
        co_await boost::asio::async_write(socket,
                    boost::asio::buffer(data,
                                        bytes_read),
                    boost::asio::use_awaitable);
    }
}
```

The coroutine loops until no data is read, happening when the client closes the connection, when the `QUIT` command is received, or when an exception occurs.

Asynchronous operations are used throughout to ensure the server remains responsive, even when handling multiple clients simultaneously.

Summary

In this chapter, we learned about Boost.Asio and how to use this library to manage asynchronous tasks that deal with external resources managed by the OS.

For that purpose, we introduced the I/O objects and I/O execution context objects, with an in-depth explanation of how they work and interact together, how they access and communicate with OS services, what the design principles are behind them, and how to use them properly in single-threaded and multi-threaded applications.

We also showed different techniques available in Boost.Asio to serialize work using strands, to manage the objects' lifetimes used by asynchronous operations, how to start, interrupt, or cancel tasks, how to manage the event processing loop that the library uses, and how to handle signals sent by the OS.

Other concepts related to networking and coroutines were also introduced, and we also implemented some useful examples using this powerful library.

All these concepts and examples allow us to acquire a deeper knowledge of how to manage asynchronous tasks in C++ and how an extensively used library works under the hood to achieve this goal.

In the next chapter, we will learn about another Boost library, Boost.Cobalt, that provides a rich and high-level interface to develop asynchronous software based in coroutines.

Further reading

- Boost.Asio official site: `https://www.boost.org/doc/libs/1_85_0/doc/html/boost_asio.html`
- Boost.Asio reference: `https://www.boost.org/doc/libs/1_85_0/doc/html/boost_asio/reference.html`
- Boost.Asio revision history: `https://www.boost.org/doc/libs/1_85_0/doc/html/boost_asio/history.html`
- Boost.Asio BSD socket API: `https://www.boost.org/doc/libs/1_85_0/doc/html/boost_asio/overview/networking/bsd_sockets.html`
- BSD socket API: `https://web.mit.edu/macdev/Development/MITSupportLib/SocketsLib/Documentation/sockets.html`

- *The Boost C++ Libraries, Boris Schälig*: `https://theboostcpplibraries.com/boost.asio`
- *Thinking Asynchronously: Designing Applications with Boost.Asio, Christopher Kohlhoff*: `https://www.youtube.com/watch?v=D-lTwGJRx0o`
- *CppCon 2016: Asynchronous IO with Boost.Asio, Michael Caisse*: `https://www.youtube.com/watch?v=rwOv_tw2eA4`
- *Pattern-Oriented Software Architecture – Patterns for Concurrent and Networked Objects*, *Volume 2*, D. Schmidt et al, Wiley, 2000
- *Boost.Asio C++ Network Programming Cookbook*, Dmytro Radchuk, Packt Publishing, 2016
- *Proactor: An Object Behavioral Pattern for Demultiplexing and Dispatching handlers for Asynchronous events*, Irfan Pyarali, Tim Harrison, Douglas C Schmidt, Thomas D Jordan. 1997
- *Reactor: An Object Behavioral Pattern for Demultiplexing and Dispatching Handlers for Synchronous events*, Douglas C Schmidt, 1995
- *Input/Output Completion Port*: `https://en.wikipedia.org/wiki/Input/output_completion_port`
- *kqueue*: `https://en.wikipedia.org/wiki/Kqueue`
- *epoll*: `https://en.wikipedia.org/wiki/Epoll`

10
Coroutines with Boost.Cobalt

The previous chapters introduced C++20 coroutines and the Boost.Asio library, which is the foundation for writing asynchronous **input/output** (**I/O**) operations using Boost. In this chapter, we will explore Boost.Cobalt, a high-level abstraction based on Boost.Asio that simplifies asynchronous programming with coroutines.

Boost.Cobalt allows you to write clear, maintainable asynchronous code while avoiding the complexities of manually implementing coroutines in C++ (as covered in *Chapter 8*). Boost.Cobalt is fully compatible with Boost.Asio, allowing you to seamlessly combine both libraries in your projects. By using Boost.Cobalt, you can focus on building your application without worrying about the low-level details of coroutines.

In this chapter, we will cover the following Boost.Cobalt topics:

- Introducing the Boost.Cobalt library
- Boost.Cobalt generators
- Boost.Cobalt tasks and promises
- Boost.Cobalt channels
- Boost.Cobalt synchronization functions

Technical requirements

To build and execute the code examples from this chapter, a compiler that supports C++20 is required. We have used both Clang `18` and GCC `14.2`.

Make sure you use Boost version 1.84 or newer and that your Boost library was compiled with C++20 support. At the time of writing this book, Cobalt support is rather fresh in Boost and not all precompiled distributions may provide this component. The situation will generally improve by the time of reading this book. If, for any reason, the Boost library in your system does not meet these requirements, you have to build it from its source. Compiling with an earlier version, such as C++17, won't include Boost.Cobalt since it relies heavily on C++20 coroutines.

You can find the complete code in the following GitHub repository:

`https://github.com/PacktPublishing/Asynchronous-Programming-with-CPP`

The examples for this chapter are located under the `Chapter_10` folder.

Introducing the Boost.Cobalt library

We introduced how C++20 supports coroutines in *Chapter 8*. It was made obvious that writing coroutines is not an easy task due to two main reasons:

- Writing coroutines in C++ requires a certain amount of code to make the coroutine work but is not related to the functionality we want to implement. For example, the coroutine we wrote to generate the Fibonacci sequence was quite simple, but we had to implement the wrapper type, the promise, and all the functions required for it to be usable.
- The development of plain C++20 coroutines requires a good knowledge of the low-level aspects of how coroutines are implemented in C++, how the compiler transforms our code to implement all the mechanisms necessary to keep the coroutine state, and details about how the functions we must implement are called and when.

Asynchronous programming is difficult enough without all those many details. It would be much better if we could focus on our program and be isolated from the lower-level concepts and code. We saw how C++23 introduced `std::generator` precisely to achieve this. Let us write just the generator code, and let the C++ Standard Library and compiler take care of the rest. It is expected that this coroutine support will improve in the next C++ version.

Boost.Cobalt, one of the libraries included in the Boost C++ libraries, allows us to do just that – avoid the coroutines details. Boost.Cobalt was introduced in Boost 1.84 and requires C++20 because it relies on the language coroutines features. It is based on Boost.Asio, and we can use both libraries in our programs.

The goal of Boost.Cobalt is to allow us to write simple single-threaded asynchronous code using coroutines – applications that can do multiple things simultaneously in a single thread. Of course, by simultaneously, we mean concurrently, not in parallel, because there is only one thread. By using Boost.Asio multithreading features, we can execute coroutines in different threads, but in this chapter, we will focus on single-threaded applications.

Eager and lazy coroutines

Before we introduce the coroutine types implemented by Boost.Cobalt, we need to define the two kinds of coroutines:

- **Eager coroutines**: An eager coroutine begins execution as soon as it is called. This means that the coroutine logic starts running immediately, and it progresses through its sequence until it hits a suspension point (such as `co_await` or `co_yield`). The creation of the coroutine effectively starts its processing, and any side effects in its body will execute right away.

  Eager coroutines are beneficial when you want the coroutine to initiate its work immediately upon being created, such as starting asynchronous network operations or preparing data.

- **Lazy coroutines**: A lazy coroutine defers its execution until is explicitly awaited or used. The coroutine object can be created without any of its body running until the caller decides to interact with it (usually by awaiting it with `co_await`).

  Lazy coroutines are useful when you want to set up a coroutine but delay its execution until a certain condition is met or when you need to coordinate its execution with other tasks.

After defining eager and lazy coroutines we will describe the different types of coroutines implemented in Boost.Cobalt.

Boost.Cobalt coroutine types

Boost.Cobalt implements four types of coroutines. We will introduce them in this section, and then see some examples later in the chapter:

- **Promise**: This is the main coroutine type in Boost.Cobalt. It is used to implement asynchronous operations that return a single value (calling `co_return`). It is an eager coroutine. It supports `co_await`, allowing asynchronous suspension and continuation. For example, a promise can be used to execute a network call that, when complete, will return its result without blocking other operations.
- **Task**: Task is the lazy version of the promise. It will not begin execution until is explicitly awaited. It provides more flexibility in controlling when and how a coroutine runs. When awaited, the task starts execution, allowing for delayed processing of asynchronous operations.
- **Generator**: In Boost.Cobalt, a generator is the only type of coroutine that can yield values. Each value is yielded individually using `co_yield.` Its functionality is like `std::generator` in C++23 but it allows waiting with `co_await` (`std::generator` doesn't).
- **Detached**: This is an eager coroutine that can use `co_await` but not `co_return` values. It cannot be resumed and usually is not awaited.

So far, we introduced Boost.Cobalt. We defined what eager and lazy coroutines are and then we defined the four main types of coroutines in the library.

In the next section, we will dive into one of the most important topics related to Boost.Cobalt – generators. We will also implement a few simple examples of generators.

Boost.Cobalt generators

As discussed in *Chapter 8*, **generator coroutines** are specialized coroutines designed to yield values incrementally. After each value is yielded, the coroutine suspends itself until the caller requests the next value. In Boost.Cobalt, generators work in the same way. They are the only coroutine type that can yield values. This makes generators essential when you need a coroutine to produce multiple values over time.

One key characteristic of Boost.Cobalt generators is that they are eager by default, meaning they start execution immediately after being called. Additionally, these generators are asynchronous, allowing them to use `co_await`, an important difference from `std::generator` introduced in C++23, which is lazy and doesn't support `co_await`.

Looking at a basic example

Let's begin with the simplest Boost.Cobalt program. This example is not that of a generator, but we will explain some important details with its help:

```
#include <iostream>

#include <boost/cobalt.hpp>

boost::cobalt::main co_main(int argc, char* argv[]) {
    std::cout << "Hello Boost.Cobalt\n";
    co_return 0;
}
```

In the preceding code, we observe the following:

- To use Boost.Cobalt, the `<boost/cobalt.hpp>` header file must be included.
- You also must link the Boost.Cobalt library to your application. We supply a `CMakeLists.txt` file to do just that, not only for Boost.Cobalt but for all the required Boost libraries. To link Boost.Cobalt explicitly (that is, not all the required Boost libraries), just add the following line to your `CMakeLists.txt` file:

  ```
  target_link_libraries(${EXEC_NAME} Boost::cobalt)
  ```

- Use of the `co_main` function. Instead of the usual `main` function, Boost.Cobalt introduces a coroutine-based entry point called `co_main`. This function can use coroutine-specific keywords such as `co_return`. Boost.Cobalt implements the required `main` function internally.

Using `co_main` will let you implement the `main` function (entry point) of your program as a coroutine, thus being able to call `co_await` and `co_return`. Remember from *Chapter 8* that the `main` function cannot be a coroutine.

If you cannot change your current `main` function, it is possible to use Boost.Cobalt. You just need to call a function, which will be the top-level function of your asynchronous code using Boost.Cobalt, from `main`. In fact, this is what Boost.Cobalt is doing: it implements a `main` function, which is the entry point of the program, and that (hidden to you) `main` function calls `co_main`.

The easiest way to use your own `main` function would be something like this:

```
cobalt::task<int> async_task() {
    // your code here
    // …
    return 0;
}
int main() {
    // main function code
    // …
    return cobalt::run(async_code();
}
```

The example simply prints a greeting message and then returns 0 calling `co_await`. In all future examples, we will follow this pattern: including the `<boost/cobalt.hpp>` header file and using `co_main` instead of `main`.

Boost.Cobalt simple generators

Armed with the knowledge from our previous basic example, we will implement a very simple generator coroutine:

```
#include <chrono>
#include <iostream>

#include <boost/cobalt.hpp>

using namespace std::chrono_literals;

using namespace boost;

cobalt::generator<int> basic_generator()
{
    std::this_thread::sleep_for(1s);
    co_yield 1;
```

```
    std::this_thread::sleep_for(1s);
    co_return 0;
}

cobalt::main co_main(int argc, char* argv[]) {
    auto g = basic_generator();
    std::cout << co_await g << std::endl;
    std::cout << co_await g << std::endl;
    co_return 0;
}
```

The preceding code shows a simple generator that yields an integer value (using `co_yield`) and returns another one (using `co_return`).

`cobalt::generator` is a `struct` template:

```
template<typename Yield, typename Push = void>
struct generator
```

The two parameter types are as follows:

- `Yield`: The generated object type
- `Push`: The input parameter type (defaults to `void`)

The `co_main` function prints both numbers after getting them using `co_await` (the caller waits for the values to be available). We have introduced some delays to simulate the processing a generator must do to generate the numbers.

Our second generator will yield the square of an integer:

```
#include <chrono>
#include <iostream>

#include <boost/cobalt.hpp>

using namespace std::chrono_literals;

using namespace boost;

cobalt::generator<int, int> square_generator(int x){
    while (x != 0) {
        x = co_yield x * x;
    }

    co_return 0;
```

```
}

cobalt::main co_main(int argc, char* argv[]){
    auto g = square_generator(10);

    std::cout << co_await g(4) << std::endl;
    std::cout << co_await g(12) << std::endl;
    std::cout << co_await g(0) << std::endl;

    co_return 0;
}
```

In this example, `square_generator` yields the square of the `x` parameter. This shows how we can *push* values to a Boost.Cobalt generator. In Boost.Cobalt, pushing values to a generator means passing parameters (in the preceding example, the passed parameters are integers).

The generator in this example, though correct, can be confusing. Take a look at the following line of code:

```
auto g = square_generator(10);
```

This creates the generator object with `10` as the initial value. Then, look at the following line of code:

```
std::cout << co_await g(4) << std::endl;
```

This will print the square of `10` and will push `4` to the generator. As you can see, the printed values are not the squares of the values passed to the generator. This is because the generator is initialized with one value (in this example, `10`), and it generates the squared value when the caller calls `co_await` to pass another value. The generator will yield `100` when receiving the new value, `4`, then it will yield `16` when receiving the value of `12`, and so on.

We said that Boost.Cobalt generators are eager, but it is possible to make them wait (`co_await`) as soon as they start executing. The following example shows how to do it:

```
#include <iostream>
#include <boost/cobalt.hpp>

boost::cobalt::generator<int, int> square_generator() {
    auto x = co_await boost::cobalt::this_coro::initial;
    while (x != 0) {
        x = co_yield x * x;
    }

    co_return 0;
}
```

```
boost::cobalt::main co_main(int, char*[]) {
    auto g = square_generator();

    std::cout << co_await g(4) << std::endl;
    std::cout << co_await g(10) << std::endl;
    std::cout << co_await g(12) << std::endl;
    std::cout << co_await g(0) << std::endl;

    co_return 0;
}
```

The code is very similar to the previous example, but there are some differences:

- We create the generator without any parameter being passed to it:

  ```
  auto g = square_generator();
  ```

- Take a look at the first line of the generator's code:

  ```
  auto x = co_await boost::cobalt::this_coro::initial;
  ```

  This makes the generator wait for the first pushed integer. This behaves as a lazy generator (in fact, it starts executing immediately because the generator is eager, but the first thing it does is wait for an integer).

- The yielded values are what we would expect from the code:

  ```
  std::cout << co_await g(10) << std::endl;
  ```

  This will print `100` instead of the square of the previously pushed integer.

Let's summarize here what the example does: the `co_main` function calls the `square_generator` coroutine to generate the square of an integer value. The generator coroutine suspends itself at the beginning waiting for the first integer and it suspends itself after yielding each square. The example is easy on purpose, just to illustrate how to write a generator using Boost.Cobalt.

An important feature of the preceding program is that it runs in a single thread. This means that `co_main` and the generator coroutine run one after another.

A Fibonacci sequence generator

In this section, we will implement a Fibonacci sequence generator like the one we implemented in *Chapter 8*. This will let us see how much easier is writing generator coroutines with Boost.Cobalt than with pure C++20 without using any coroutines library.

We have written two versions of the generator. The first one calculates an arbitrary term of the Fibonacci sequence. We push the term we want to generate, and we get it. This generator uses a lambda as a Fibonacci calculator:

```
boost::cobalt::generator<int, int> fibonacci_term() {
    auto fibonacci = [](int n) {
        if (n < 2) {
            return n;
        }

        int f0 = 0;
        int f1 = 1;
        int f;

        for (int i = 2; i <= n; ++i) {
            f = f0 + f1;
            f0 = f1;
            f1 = f;
        }

        return f;
    };

    auto x = co_await boost::cobalt::this_coro::initial;
    while (x != -1) {
        x = co_yield fibonacci(x);
    }

    co_return 0;
 }
```

In the preceding code, we see that this generator is very similar to the one we implemented in the previous section to calculate the square of a number. At the beginning of the coroutine, we have the following:

```
auto x = co_await boost::cobalt::this_coro::initial;
```

This line of code suspends the coroutine to wait for the first input value.

And then we have the following:

```
while (x != -1) {
        x = co_yield fibonacci(x);
    }
```

This generates the requested Fibonacci sequence term and suspends itself until the next term is requested. While the requested term is not equal to `-1`, we can go on requesting more values until pushing `-1` terminates the coroutine.

The next version of the Fibonacci generator will yield an infinite number of terms as they are requested. By *infinite* we mean *potentially infinite*. Think about this generator as always ready to yield one more Fibonacci sequence number:

```
boost::cobalt::generator<int> fibonacci_sequence() {
    int f0 = 0;
    int f1 = 1;
    int f = 0;

    while (true) {
        co_yield f0;

        f = f0 + f1;
        f0 = f1;
        f1 = f;
    }
}
```

The preceding code is easy to understand: the coroutine yields a value and suspends itself until another one is requested and the coroutine calculates the new value and yields it and suspends itself again in an infinite loop.

In this case, we can see the advantage of a coroutine: we can generate the terms of the Fibonacci sequence, one after another, whenever we need them. We don't need to keep any state to generate the next term because the state is kept in the coroutine.

Also note that even if the function executes an infinite loop, because it is a coroutine, it suspends and resumes again and again, avoiding blocking the current thread.

Boost.Cobalt tasks and promises

As we have already seen in this chapter, Boost.Cobalt promises are eager coroutines that return one value and Boost.Cobalt tasks are the lazy version of promises.

We can see them as just functions that don't yield multiple values like generators do. We can call a promise repeatedly to get more than one value, but the state won't be kept between calls (as in generators). Basically, a promise is a coroutine that can use `co_await` (it can use `co_return` too).

Different use cases of promises would be a socket listener to receive network packets, process them, make queries to a database, and then generate some results from the data. In general, their functionality

requires asynchronously waiting for some result and then performing some processing on that result (or maybe just returning it to the caller).

Our first example is a simple promise that generates one random number (this can be done with a generator too):

```
#include <iostream>
#include <random>

#include <boost/cobalt.hpp>

boost::cobalt::promise<int> random_number(int min, int max) {
    std::random_device rd;
    std::mt19937 gen(rd());

    std::uniform_int_distribution<> dist(min, max);
    co_return dist(gen);
}

boost::cobalt::promise<int> random(int min, int max) {
    int res = co_await random_number(min, max);
    co_return res;
}

boost::cobalt::main co_main(int, char*[]) {
    for (int i = 0; i < 10; ++i) {
        auto r = random(1, 100);
        std::cout << "random number between 1 and 100: "
                  << co_await r << std::endl;
    }
    co_return 0;
}
```

In the preceding code, we have written three coroutines:

- `co_main`: Remember that in Boost.Cobalt, `co_main` is a coroutine and it calls `co_return` to return a value.
- `random()`: This coroutine returns a random number to the caller. It calls `random()` with `co_await` to generate the random number. It asynchronously waits for the random number to be generated.
- `random_number()`: This coroutine generates a uniformly distributed random number between two values, `min` and `max`, and returns it to the caller. `random_number()` is also a promise.

The following coroutine returns a `std::vector<int>` of random numbers. `co_await random_number()` is called in a loop to generate a vector of `n` random numbers:

```
boost::cobalt::promise<std::vector<int>> random_vector(int min, int
max, int n) {
    std::vector<int> rv(n);
    for (int i = 0; i < n; ++i) {
        rv[i] = co_await random_number(min, max);
    }
    co_return rv;
}
```

The preceding function returns a promise of `std::vector<int>`. To access the vector, we need to call `get()`:

```
auto v = random_vector(1, 100, 20);
for (int n : v.get()) {
    std::cout << n << " ";
}
std::cout << std::endl;
```

The previous code prints the elements of the `v` vector. To access the vector, we need to call `v.get()`.

We are going to implement a second example to illustrate how the execution of promises and tasks differ:

```
#include <chrono>
#include <iostream>
#include <thread>

#include <boost/cobalt.hpp>

void sleep(){
    std::this_thread::sleep_for(std::chrono::seconds(2));
}

boost::cobalt::promise<int> eager_promise(){
    std::cout << "Eager promise started\n";
    sleep();
    std::cout << "Eager promise done\n";
    co_return 1;
}

boost::cobalt::task<int> lazy_task(){
    std::cout << "Lazy task started\n";
    sleep();
```

```
    std::cout << "Lazy task done\n";
    co_return 2;
}

boost::cobalt::main co_main(int, char*[]){
    std::cout << "Calling eager_promise...\n";
    auto promise_result = eager_promise();
    std::cout << "Promise called, but not yet awaited.\n";

    std::cout << "Calling lazy_task...\n";
    auto task_result = lazy_task();
    std::cout << "Task called, but not yet awaited.\n";

    std::cout << "Awaiting both results...\n";
    int promise_value = co_await promise_result;
    std::cout << "Promise value: " << promise_value
              << std::endl;

    int task_value = co_await task_result;
    std::cout << "Task value: " << task_value
              << std::endl;

    co_return 0;
}
```

In this example, we have implemented two coroutines: a promise and a task. As we have already said, the promise is eager and it starts executing as soon as it's called. The task is lazy and it's suspended after being called.

When we run the program, it prints all the messages, which let us know exactly how the coroutines execute.

After the first three lines of `co_main()` are executed, the printed output is the following:

```
Calling eager_promise...
Eager promise started
Eager promise done
Promise called, but not yet awaited.
```

From these messages, we know that the promise has been executed until the call to `co_return`.

After the next three lines of `co_main()` are executed, the printed output has these new messages:

```
Calling lazy_task...
Task called, but not yet awaited.
```

Here, we see that the task has not been executed. It is a lazy coroutine and, for this reason, it just suspends immediately after being called and no messages are printed by this coroutine just yet.

Three more lines of `co_main()` are executed, and these are the new messages in the program's output:

```
Awaiting both results...
Promise value: 1
```

The call to `co_await` on the promise gives us its result (in this case, set to `1`) and its execution ends.

Finally, we call `co_await` on the task, and it then executes and returns its value (which, in this case, is set to `2`). The output is the following:

```
Lazy task started
Lazy task done
Task value: 2
```

This example shows how tasks are lazy and start suspended and only resume executing when the caller calls `co_await` on them.

In this section, we have seen that, as in the case of generators, it is much easier to write promise and task coroutines using Boost.Cobalt than just using plain C++. We don't need to write all the support code that C++ requires to implement coroutines. We have also seen the main difference between tasks and promises.

In the next section, we will study an example of a channel, a communication mechanism between two coroutines in a producer/consumer model.

Boost.Cobalt channels

In Boost.Cobalt, channels provide a way for coroutines to communicate asynchronously, allowing data transfer between a producer and a consumer coroutine in a safe and efficient manner. They are inspired by Golang channels and allow communication through message passing, promoting a *share-memory-by-communicating* paradigm.

A **channel** is a mechanism through which values are asynchronously passed from one coroutine (the producer) to another (the consumer). This communication is non-blocking, which means that coroutines can suspend their execution when they wait for data to be available on the channel or when they write data to a channel that has limited capacity. Let's clarify this: both reading and writing operations may be blocking, depending on the buffer size if, by *blocking*, we mean coroutines are suspended, but on the other hand, from the point of view of threads, these operations don't block the thread.

If the buffer size is zero, a read and a write will need to occur at the same time and act as a rendezvous (synchronous communication). If the channel size is bigger than zero and the buffer is not full, the write operation will not suspend the coroutine. Likewise, if the buffer is not empty, the read operation will not suspend.

Similar to Golang channels, Boost.Cobalt channels are strongly typed. A channel is defined for a specific type, and only that type can be sent through it. For example, a channel of the `int` type (`boost::cobalt::channel<int>`) can only transmit integers.

Let's now see an example of a channel:

```
#include <iostream>
#include <boost/cobalt.hpp>
#include <boost/asio.hpp>
boost::cobalt::promise<void> producer(boost::cobalt::channel<int>& ch)
{
    for (int i = 1; i <= 10; ++i) {
        std::cout << "Producer waiting for request\n";
        co_await ch.write(i);
        std::cout << "Producing value " << i << std::endl;
    }
    std::cout << "Producer end\n";
    ch.close();
    co_return;
}
boost::cobalt::main co_main(int, char*[]) {
    boost::cobalt::channel<int> ch;
    auto p = producer(ch);
    while (ch.is_open()) {
        std::cout << "Consumer waiting for next number \n";
        std::this_thread::sleep_for(std::chrono::seconds(5));
        auto n = co_await ch.read();
        std::cout << "Consuming value " << n << std::endl;
        std::cout << n * n << std::endl;
    }
    co_await p;
    co_return 0;
}
```

In this example, we create a size `0` channel and two coroutines: the `producer` promise and `co_main()`, which acts as the consumer. The producer writes integers to the channel and the consumer reads them back and prints them squared.

We added `std::this_thread::sleep` to delay the program execution and, hence, be able to see what happens as the program runs. Let's see an excerpt of the example's output to see how it works:

```
Producer waiting for request
Consumer waiting for next number
Producing value 1
Producer waiting for request
```

```
Consuming value 1
1
Consumer waiting for next number
Producing value 2
Producer waiting for request
Consuming value 2
4
Consumer waiting for next number
Producing value 3
Producer waiting for request
Consuming value 3
9
Consumer waiting for next number
```

Both the consumer and the producer wait for the next action to happen. The producer will always wait for the consumer to request the next item. This is basically how generators work, and it is a very common pattern in asynchronous code using coroutines.

The consumer executes the following line of code:

```
auto n = co_await ch.read();
```

Then, the producer writes the next number to the channel and waits for the next request. This is done in the following line of code:

```
co_await ch.write(i);
```

You can see in the fourth line of the previous output excerpt how the producer goes back to waiting for the next request.

Boost.Cobalt channels make writing this kind of asynchronous code very clean and easy to understand.

The example shows both coroutines communicating through a channel.

That wraps up this section. The next one will introduce synchronization functions – mechanisms to wait for more than one coroutine.

Boost.Cobalt synchronization functions

Previously, we implemented coroutines, and, in every case in which we called `co_await`, we did it for just one coroutine. This means we waited for the result of only one coroutine. Boost.Cobalt has mechanisms that allow us to wait on more than one coroutine. These mechanisms are called **synchronization functions**.

There are four synchronization functions implemented in Boost.Cobalt:

- **race**: The `race` function waits for one coroutine out of a set to complete, but it does so in a pseudo-random manner. This mechanism helps avoid starvation of coroutines, ensuring that one coroutine doesn't dominate the execution flow over others. When you have multiple asynchronous operations and you want the first to finish to determine the flow, `race` will allow any coroutine that becomes ready to proceed in a non-deterministic order.

  When you have multiple tasks (tasks in the generic sense, not Boost.Cobalt tasks) and are interested in completing one first, without preference as to which one, but want to prevent one coroutine from always winning in situations where readiness is simultaneous, you will use `race`.

- **join**: The `join` function waits for all the coroutines in a given set to complete and return their results as values. If any of the coroutines throws an exception, `join` will propagate the exception to the caller. It's a way to gather results from multiple asynchronous operations that must all finish before proceeding.

  You will use `join` when you need the result of multiple asynchronous operations together and want to throw an error if any of them fail.

- **gather**: The `gather` function, like `join`, waits for a set of coroutines to complete, but it handles exceptions differently. Instead of throwing an exception immediately when one of the coroutines fails, `gather` captures each coroutine's result individually. This means that you can inspect the outcome (success or failure) of each coroutine independently.

  When you need all asynchronous operations to complete but you want to capture all results and exceptions individually to handle them separately, you will use `gather`.

- **left_race**: The `left_race` function is like `race` but with deterministic behavior. It evaluates the coroutines from left to right and waits for the first coroutine to become ready. This can be useful when the order of coroutine completion matters, and you want to ensure a predictable outcome based on the order in which they were provided.

  When you have multiple potential results and need to favor the first available coroutine in the order provided, making the behavior more predictable than `race`, you will use `left_race`.

In this section, we will explore examples of both `join` and `gather` functions. As we have seen, both functions wait for a set of coroutines to finish. The difference between them is that `join` throws an exception if any of the coroutines throw an exception, and `gather` always returns the results for all the awaited coroutines. In the case of the `gather` function, the result for each coroutine will either be an error (absent value) or a value. `join` returns a tuple of values or throws an exception; `gather` returns a tuple of optional values that have no value in the event of an exception (the optional variables are not initialized).

The full code for the following example is in the GitHub repo. We will focus here on the main sections.

We have defined a simple function to simulate data processing, which is just a delay. The function throws an exception if we pass a delay bigger than 5,000 milliseconds:

```
boost::cobalt::promise<std::chrono::milliseconds::rep>
process(std::chrono::milliseconds ms) {
    if (ms > std::chrono::milliseconds(5000)) {
        throw std::runtime_error("delay throw");
    }

    boost::asio::steady_timer tmr{ co_await boost::cobalt::this_
coro::executor, ms };
    co_await tmr.async_wait(boost::cobalt::use_op);
    co_return ms.count();
}
```

The function is a Boost.Cobalt promise.

Now, in the next section of the code, we will wait for three instances of this promise to run:

```
auto result = co_await boost::cobalt::join(process(100ms),
                                           process(200ms),
                                           process(300ms));

std::cout << "First coroutine finished in: "
          <<  std::get<0>(result) << "ms\n";
std::cout << "Second coroutine took finished in: "
          <<  std::get<1>(result) << "ms\n";
std::cout << "Third coroutine took finished in: "
         <<  std::get<2>(result) << "ms\n";
```

The preceding code calls `join` to wait for the completion of three coroutines and then prints the time they took. As you can see, the result is a tuple, and to make the code as simple as possible, we just call `std::get<i>(result)` for each element. In this case, all the processing times are inside the valid range and no exception is thrown, so we can get the result for all the executed coroutines.

If an exception is thrown, then we won't get any value:

```
try {
    auto result throw = co_await
    boost::cobalt::join(process(100ms),
                        process(20000ms),
                        process(300ms));
}
catch (...) {
    std::cout << "An exception was thrown\n";
}
```

The preceding code will throw an exception because the second coroutine receives a processing time outside of the valid range. It will print an error message.

When calling the `join` function, we want all the coroutines to be considered as part of processing and, in the event of an exception, the full processing fails.

If we need to get all the results for each coroutine, we will use the `gather` function:

```
try
    auto result throw =
    boost::cobalt::co_await lt::gather(process(100ms),
                                       process(20000ms),
                                       process(300ms));

    if (std::get<0>(result throw).has value()) {
        std::cout << "First coroutine took: "
                  <<  *std::get<0>(result throw)
                  << "msec\n";
    }
    else {
        std::cout << "First coroutine threw an exception\n";
    }
    if (std::get<1>(result throw).has value()) {
        std::cout << "Second coroutine took: "
                  <<  *std::get<1>(result throw)
                  << "msec\n";
    }
    else {
        std::cout << "Second coroutine threw an exception\n";
    }
    if (std::get<2>(result throw).has value()) {
        std::cout << "Third coroutine took: "
                  <<  *std::get<2>(result throw)
                  << "msec\n";
    }
    else {
        std::cout << "Third coroutine threw an exception\n";
    }
}
catch (...) {
    // this is never reached because gather doesn't throw exceptions
    std::cout << "An exception was thrown\n";
}
```

We have put the code inside a `try-catch` block but no exception is thrown. The `gather` function returns a tuple of optional values, and we need to check whether each coroutine returned a value or not (the optional has a value or not).

We use `gather` when we want the coroutines to return a value if they are executed successfully.

These examples of `join` and `gather` functions conclude our introduction to the Boost.Cobalt synchronization functions.

Summary

In this chapter, we saw how to implement coroutines using the Boost.Cobalt library. It was added to Boost only recently, and there is not much information about it. It simplifies the development of asynchronous code with coroutines, avoiding writing the low-level code necessary for C++20 coroutines.

We studied the main library concepts and developed some simple examples to understand them.

With Boost.Cobalt, writing asynchronous code using coroutines is simplified. All the low-level details of writing coroutines in C++ are implemented by the library and we can focus just on the functionality we want to implement in our programs.

In the next chapter, we will see how to debug asynchronous code.

Further reading

- Boost.Cobalt reference: *Boost.Cobalt reference guide* (`https://www.boost.org/doc/libs/1_86_0/libs/cobalt/doc/html/index.html#overview`)
- A YouTube video on Boost.Cobalt: *Using coroutines with Boost.Cobalt* (`https://www.youtube.com/watch?v=yElSdUqEvME`)

Part 5: Debugging, Testing, and Performance Optimization in Asynchronous Programming

In this final part, we focus on the essential practices of debugging, testing, and optimizing the performance of multithreaded and asynchronous programs. We will begin by using logging and advanced debugging tools and techniques, including reverse debugging and code sanitizers, to identify and resolve subtle bugs in asynchronous applications, such as crashes, deadlocks, race conditions, memory leaks, and thread safety issues, followed by testing strategies tailored for asynchronous code using the GoogleTest framework. Finally, we will dive into performance optimization, understanding key concepts such as cache sharing, false sharing, and how to mitigate performance bottlenecks. Mastering these techniques will provide us with a comprehensive toolkit for identifying, diagnosing, and improving the quality and performance of asynchronous applications.

This part has the following chapters:

- *Chapter 11*, *Logging and Debugging Asynchronous Software*
- *Chapter 12*, *Sanitizing and Testing Asynchronous Software*
- *Chapter 13*, *Improving Asynchronous Software Performance*

11

Logging and Debugging Asynchronous Software

There is no way to ensure that a software product is free from bugs, so from time to time, bugs can appear. This is when logging and debugging are indispensable.

Logging and debugging are essential for identifying and diagnosing issues in software systems. They provide visibility into the runtime behavior of code, helping developers trace errors, monitor performance, and understand the flow of execution. By using logging and debugging effectively, developers can detect bugs, resolve unexpected behavior, and improve overall system stability and maintainability.

While writing this chapter, we assume you are already familiar with using a debugger to debug C++ programs and know some basic debugger commands and terminology, such as breakpoints, watchers, frames, or stack traces. To brush up on that knowledge, you can refer to the references provided in the *Further reading* section at the end of the chapter.

In this chapter, we're going to cover the following main topics:

- How to use logging to spot bugs
- How to debug asynchronous software

Technical requirements

For this chapter, we will need to install third-party libraries to compile examples.

The `spdlog` and `{fmt}` libraries need to be installed to compile the example in the logging section. Please check their documentation (`spdlog`'s documentation is available at `https://github.com/gabime/spdlog` and `{fmt}`'s documentation is available at `https://github.com/fmtlib/fmt`) and follow the installation steps suitable for your platform.

Some examples need a compiler supporting C++20. Therefore, check the technical requirements section in *Chapter 3*, which has some guidance on how to install GCC 13 and Clang 8 compilers.

You can find all the complete code in the following GitHub repository:

`https://github.com/PacktPublishing/Asynchronous-Programming-with-CPP`

The examples for this chapter are located under the `Chapter_11` folder. All source code files can be compiled using CMake as follows:

```
$ cmake . && cmake —build .
```

Executable binaries will be generated under the `bin` directory.

How to use logging to spot bugs

Let's start with a trivial but useful method for understanding what a software program does while executing – logging.

Logging is the process of keeping a log of events that occur in a program, storing information by using messages to record how a program executes, tracking its flow, and helping with identifying issues and bugs.

Most Unix-based logging systems use the standard protocol, **syslog**, created by Eric Altman back in 1980 as part of the Sendmail project. This standard protocol defines the boundaries between the software generating the log messages, the system storing them, and the software reporting and analyzing these log events.

Each log message includes a facility code and a severity level. The facility code identifies the type of system that originated a specific log message (user-level, kernel, system, network, etc.), and the severity level describes the condition of the system, indicating the urgency of dealing with a specific issue, the severity levels being emergency, alert, critical, error, warning, notice, info, and debug.

Most logging systems or loggers provide various destinations or sinks for log messages: console, files that can later be opened and analyzed, remote syslog servers, or relays, among other destinations.

Logging is useful where debuggers are not, as we will see later, especially in distributed, multithreaded, real-time, scientific, or event-centric applications, where inspecting data or following the program flow using the debugger can become a tedious task.

Logging libraries usually also provide a thread-safe singleton class that allows multithreading and asynchronous writing to log files, helps with log rotation, avoids large log files by creating new ones on the fly without losing log events, and time stamping, for better tracking when a log event happens.

Instead of implementing our own multithreaded logging system, a better approach is to use some well-tested and documented production-ready libraries.

How to select a third-party library

When selecting a logging library (or any other library), we need to investigate the following points before integrating it into our software to avoid future issues:

- **Support**: Is the library updated and upgraded regularly? Is there a community or active ecosystem around the library that can help with any questions that can arise? Is the community happy using the library?
- **Quality**: Is there a public bugs report system? Are bug reports dealt with promptly, providing solutions and fixing bugs in the library? Does it support recent compiler versions and support latest C++ features?
- **Security**: Does the library, or any of its dependent libraries, have any vulnerabilities reported?
- **License**: Is the library license aligned with our development and product needs? Is the cost affordable?

For complex systems, it may be worth considering centralized systems to collect and generate logging reports or dashboards, such as **Sentry** (`https://sentry.io`) or **Logstash** (`https://www.elastic.co/logstash`), that can collect, parse, and transform logs, and can be integrated with other tools, such as **Graylog** (`https://graylog.org`), **Grafana** (`https://grafana.com`), or **Kibana** (`https://www.elastic.co/kibana`).

The next section describes some interesting logging libraries.

Some relevant logging libraries

There are many logging libraries in the market, each covering some specific software requirements. Depending on the program constraints and needs, one of the following libraries might be more suitable than others.

In *Chapter 9*, we explored **Boost.Asio**. Boost also provides another library, **Boost.Log** (`https://github.com/boostorg/log`), a powerful and configurable logging library.

Google also provides many open source libraries, including **glog**, the Google logging library (`https://github.com/google/glog`), which is a C++14 library that provides C++-style streams APIs and helper macros.

If the developer is familiar with Java, an excellent choice could be Apache **Log4cxx** (`https://logging.apache.org/log4cxx`), based on **Log4j** (`https://logging.apache.org/log4j`), a versatile, industrial-grade, Java logging framework.

Other logging libraries worth considering are as follows:

- **spdlog** (`https://github.com/gabime/spdlog`) is an interesting logging library that we can use with the `{fmt}` library. Also, the program can start logging messages and queuing them since startup, even before the log output file name is specified.

- **Quill** (`https://github.com/odygrd/quill`) is an asynchronous low-latency C++ logging library.
- **NanoLog** (`https://github.com/PlatformLab/NanoLog`) is a nanosecond scale logging system with `printf`-like APIs.
- **lwlog** (`https://github.com/ChristianPanov/lwlog`) is an amazingly fast asynchronous C++17 logging library.
- **XTR** (`https://github.com/choll/xtr`) is a fast and convenient C++ logging library for low-latency and real-time environments.
- **Reckless** (`https://github.com/mattiasflodin/reckless`) is a low-latency and high-throughput logging library.
- **uberlog** (`https://github.com/IMQS/uberlog`) is a cross-platform and multi-process C++ logging system.
- **Easylogging++** (`https://github.com/abumq/easyloggingpp`) is a single-header C++ logging library with the ability to write our own sinks and track performance.
- **tracetool** (`https://github.com/froglogic/tracetool`) is a logging and tracing shared library.

As a guideline, depending on the system to develop, we might choose one of the following libraries:

- **For low-latency or real-time systems**: Quill, XTR, or Reckless
- **For high performance at nanosecond scale logging**: NanoLog
- **For asynchronous logging**: Quill or `lwlog`
- **For cross-platform, multi-process applications**: `uberlog`
- **For simple and flexible logging**: Easylogging++ or glog
- **For familiarity with Java logging**: Log4cxx

All libraries have advantages but also disadvantages that need to be investigated prior to selecting a library to include in your system. The following table summarizes these points:

Library	Advantages	Disadvantages
`spdlog`	Easy integration, performance-focused, customizable	Lacks some advanced features for extreme low-latency needs
Quill	High performance in low-latency systems	More complex setup compared to simpler, synchronous loggers
NanoLog	Best in class for speed, optimized for performance	Limited in features; suited for specialized use cases

Library	Advantages	Disadvantages
`lwlog`	Lightweight, good for quick integration	Less mature and feature-rich than alternatives
XTR	Very efficient, user-friendly interface	More suited for specific real-time applications
Reckless	Highly optimized for throughput and low latency	Limited flexibility compared to more general-purpose loggers
`uberlog`	Great for multi-process and distributed systems	Not as fast as specialized low-latency loggers
Easylogging++	Easy to use, customizable output sinks	Less performance-optimized than some other libraries
`tracetool`	Combines logging and tracing in one library	Not focused on low-latency or high-throughput
Boost.Log	Versatile, integrates well with Boost libraries	Higher complexity; can be overkill for simple logging needs
glog	Simple to use, good for projects requiring easy APIs	Not as feature-rich for advanced customization
Log4cxx	Robust, time-tested, industrial-strength logging	More complex to set up, especially for smaller projects

Table 11.1: Advantages and disadvantages of various libraries

Please visit the logging libraries' websites to understand better what features they provide and compare performance between them.

As `spdlog` is the most forked and starred C++ logging library repository in GitHub, in the next section, we will implement an example of using this library to catch a race condition.

Logging a deadlock – an example

Before implementing this example, we need to install the `spdlog` and `{fmt}` libraries. `{fmt}` (`https://github.com/fmtlib/fmt`) is an open source formatting library providing a fast and safe alternative to C++ IOStreams.

Please check their documentation and follow the installation steps depending on your platform.

Let's implement an example where a deadlock is happening. As we learned in *Chapter 4*, a deadlock can happen when two or more threads need to acquire more than one mutex to perform their work. If mutexes are not acquired in the same order, a thread can acquire a mutex and wait forever for another mutex acquired by another thread.

In this example, two threads need to acquire two mutexes, `mtx1` and `mtx2`, to increase the value of the `counter1` and `counter2` counters and swap their values. As the mutexes are acquired in different order by the threads, a deadlock can happen.

Let's start by including the required libraries:

```
#include <fmt/core.h>
#include <spdlog/sinks/basic_file_sink.h>
#include <spdlog/sinks/stdout_color_sinks.h>
#include <spdlog/spdlog.h>

#include <chrono>
#include <iostream>
#include <mutex>
#include <thread>

using namespace std::chrono_literals;
```

In the `main()` function, we define the counters and mutexes:

```
uint32_t counter1{};
std::mutex mtx1;

uint32_t counter2{};
std::mutex mtx2;
```

Before spawning the threads, let's set up a **multi-sink logger**, a logger that can write log messages into the console and a log file simultaneously. We will also set up its log level to debug, making the logger publish all log messages with a severity level greater than debug, and the format for each log line consisting of the timestamp, the thread identifier, the log level, and the log message:

```
auto console_sink = std::make_shared<
         spdlog::sinks::stdout_color_sink_mt>();
console_sink->set_level(spdlog::level::debug);

auto file_sink = std::make_shared<
         spdlog::sinks::basic_file_sink_mt>("logging.log",
                                            true);
file_sink->set_level(spdlog::level::info);

spdlog::logger logger("multi_sink",
         {console_sink, file_sink});

logger.set_pattern(
         "%Y-%m-%d %H:%M:%S.%f - Thread %t [%l] : %v");
logger.set_level(spdlog::level::debug);
```

We also declare an `increase_and_swap` lambda function that increases the values of both counters and swaps them:

```
auto increase_and_swap = [&]() {
    logger.info("Incrementing both counters...");
    counter1++;
    counter2++;

    logger.info("Swapping counters...");
    std::swap(counter1, counter2);
};
```

Two worker lambda functions, `worker1` and `worker2`, acquire both mutexes and call `increase_and_swap()` before exiting. As lock guard (`std::lock_guard`) objects are used, the mutexes are released when leaving the worker lambda functions during their destruction:

```
auto worker1 = [&]() {
    logger.debug("Entering worker1");

    logger.info("Locking mtx1...");
    std::lock_guard<std::mutex> lock1(mtx1);
    logger.info("Mutex mtx1 locked");

    std::this_thread::sleep_for(100ms);

    logger.info("Locking mtx2...");
    std::lock_guard<std::mutex> lock2(mtx2);
    logger.info("Mutex mtx2 locked");

    increase_and_swap();

    logger.debug("Leaving worker1");
};

auto worker2 = [&]() {
    logger.debug("Entering worker2");

    logger.info("Locking mtx2...");
    std::lock_guard<std::mutex> lock2(mtx2);
    logger.info("Mutex mtx2 locked");

    std::this_thread::sleep_for(100ms);

    logger.info("Locking mtx1...");
```

```
    std::lock_guard<std::mutex> lock1(mtx1);
    logger.info("Mutex mtx1 locked");

    increase_and_swap();

    logger.debug("Leaving worker2");
};

logger.debug("Starting main function...");

std::thread t1(worker1);
std::thread t2(worker2);

t1.join();
t2.join();
```

Both worker lambda functions are similar but with a small difference: `worker1` acquires `mutex1` and then `mutex2`, and `worker2` follows the opposite order, first acquiring `mutex2` and then `mutex1`. There is a sleep period between both mutexes' acquisition to let the other thread acquire its mutex, therefore, provoking a deadlock as `worker1` will acquire `mutex1` and `worker2` will acquire `mutex2`.

Then, after sleeping, `worker1` will try to acquire `mutex2` and `worker2` will try the same with `mutex1`, but none of them will succeed, blocking forever in a deadlock.

The following is the output when running this code:

```
2024-09-04 23:39:54.484005 - Thread 38984 [debug] : Starting main
function...
2024-09-04 23:39:54.484106 - Thread 38985 [debug] : Entering worker1
2024-09-04 23:39:54.484116 - Thread 38985 [info] : Locking mtx1...
2024-09-04 23:39:54.484136 - Thread 38986 [debug] : Entering worker2
2024-09-04 23:39:54.484151 - Thread 38986 [info] : Locking mtx2...
2024-09-04 23:39:54.484160 - Thread 38986 [info] : Mutex mtx2 locked
2024-09-04 23:39:54.484146 - Thread 38985 [info] : Mutex mtx1 locked
2024-09-04 23:39:54.584250 - Thread 38986 [info] : Locking mtx1...
2024-09-04 23:39:54.584255 - Thread 38985 [info] : Locking mtx2...
```

The first symptom to note when inspecting the logs is that the program never finishes and therefore probably is deadlocked.

From the logger output, we can see that `t1` (thread `38985`) is running `worker1` and `t2` (thread `38986`) is running `worker2`. As soon as `t1` enters `worker1`, it acquires `mtx1`. The `mtx2` mutex is acquired by `t2` though, as soon as `worker2` starts. Then, both threads wait for 100 ms and try to acquire the other mutex, but none succeed, and the program remains blocked.

Logging is indispensable in production systems but imposes some performance penalty if abused, and most of the time requires human intervention to investigate an issue. As a compromise between log verbosity and performance penalty, one might choose to implement different logging levels and log only major events during normal operation, while still retaining the ability to provide extremely detailed logs if opted for, when the situation needs it. A more automated way to detect errors in code early in the development cycle is by using testing and code sanitizers, which we will learn about in the next chapter.

Not all bugs can be detected, so usually using a debugger is the way to track down and fix bugs in software. Let's learn next how to debug multithreading and asynchronous code.

How to debug asynchronous software

Debugging is the process of finding and fixing errors in computer programs.

In this section, we will explore several techniques to debug multithreading and asynchronous software. You must have some previous knowledge of how to use debuggers, such as **GDB** (the GNU project debugger) or **LLDB** (the LLVM low-level debugger), and the terminology of the debugging process, such as breakpoints, watchers, backtraces, frames, and crash reports.

Both GDB and LLDB are excellent debuggers with most of their commands being the same and only a few ones that differ. LLDB might be preferred if the program is being debugged on macOS or for large code bases. On the other hand, GDB has an established legacy, being familiar to many developers, and supporting a broader range of architectures and platforms. In this section, we will use GDB 15.1 just because it is part of the GNU framework and was designed to work alongside the `g++` compiler, but most commands shown later can also be used with LLDB when debugging programs compiled with `clang++`.

As some debugger features that deal with multithreading and asynchronous code are still in development, always update the debuggers to the latest versions to include up-to-date features and fixes.

Some useful GDB commands

Let's start with some GDB commands that are useful when debugging any kind of program and acquire a foundation for the next sections.

When debugging a program, we can start the debugger and pass the program as an argument. Extra arguments that the program might need can be passed with the `--args` option:

```
$ gdb <program> --args <args>
```

Or, we can attach the debugger to a running program by using its **process identifier** (**PID**):

```
$ gdb -p <PID>
```

Once inside the debugger, we can run the program (with the `run` command) or start it (with the `start` command). Running means that the program executes until reaching a breakpoint or finishing. `start` just places a temporary breakpoint at the beginning of the `main()` function and runs the program, stopping the execution at the beginning of the program.

If, for instance, we want to debug a program that has already crashed, we can use the core dump file that the crash generated, which might be stored in a specific location in the system (usually `/var/lib/apport/coredump/` on Linux systems, but please check for the exact location in your system by visiting the official documentation). Also, note that typically, core dumps are disabled by default, requiring the `ulimit -c unlimited` command to be run prior to, and in the same shell as, the program crashing. The `unlimited` argument can be changed to some arbitrary limit if we are dealing with an exceptionally large program or the system is short on disk space.

After the `coredump` file is generated, just copy it to the location where the program binary is located and use the following command:

```
$ gdb <program> <coredump>
```

Note that all binaries must have debugging symbols, thus compiled with the `-g` option. In production systems, release binaries usually have symbols stripped and stored in separate files. There are GDB commands to include those symbols and command-line tools to inspect them, but this topic is beyond the scope of this book.

Once the debugger starts, we can use GDB commands to navigate through the code or check variables. Some useful commands are as follows:

- `info args`: This shows information about arguments used to call the current function.
- `info locals`: This shows local variables in the current scope.
- `whatis`: This shows the type of the given variable or expression.
- `return`: This returns from the current function without executing the rest of the instructions. A return value can be specified.
- `backtrace`: This lists all stack frames in the current call stack.
- `frame`: This lets you change to a specific stack frame.
- `up`, `down`: This moves across the call stack, toward the caller (`up`) or the callee (`down`) of the current function.
- `print`: This evaluates and displays the value of an expression, being that expression a variable name, a class member, a pointer to a memory region, or directly a memory address. We can also define pretty printers to display our own classes.

Let's finish this section with one of the most basic but also used techniques for debugging programs. This technique is called `printf`. Every developer has used `printf` or alternative commands to print the content of variables along the code path to show their contents in strategic code locations. In GDB, the `dprintf` command helps to set `printf`-style breakpoints that print information when those breakpoints are hit but without stopping the program execution. This way, we can use print statements when debugging a program without the need for code modifications, recompilation, and program restarts.

Its syntax is as follows:

```
$ dprintf <location>, <format>, <args>
```

For example, if we want to set a `printf` statement at line 25 to print the content of the `x` variable but only if its value is greater than `5`, this is the command:

```
$ dprintf 25, "x = %d\n", x if x > 5
```

Now that we have some foundations, let's start by debugging a multithreaded program.

Debugging multithreaded programs

The example shown here will never finish as a deadlock will happen as two mutexes are locked in a different order by different threads, as already explained earlier in this chapter when introducing logging:

```
#include <chrono>
#include <mutex>
#include <thread>

using namespace std::chrono_literals;

int main() {
    std::mutex mtx1, mtx2;

    std::thread t1([&]() {
        std::lock_guard lock1(mtx1);
        std::this_thread::sleep_for(100ms);
        std::lock_guard lock2(mtx2);
    });

    std::thread t2([&]() {
        std::lock_guard lock2(mtx2);
        std::this_thread::sleep_for(100ms);
        std::lock_guard lock1(mtx1);
    });
```

```
    t1.join();
    t2.join();

    return 0;
}
```

First, let's compile this example using `g++` and add debug symbols (the `-g` option) and disallow code optimization (the `-O0` option), preventing the compiler from restructuring the binary code and making it more difficult for the debugger to find and show relevant information by using the `--fno-omit-frame-pointer` option.

The following command compiles the `test.cpp` source file and generates the `test` binary. We can also use `clang++` with the same options:

```
$ g++ -o test -g -O0 --fno-omit-frame-pointer test.cpp
```

If we run the resulting program, this will never finish:

```
$ ./test
```

To debug a program that is running, let's first retrieve its PID by using the `ps` Unix command:

```
$ ps aux | grep test
```

Then, attach the debugger by providing `pid` and start debugging the program:

```
$ gdb -p <pid>
```

Say the debugger starts with the following message:

```
ptrace: Operation not permitted.
```

Then, just run the following command:

```
$ sudo sysctl -w kernel.yama.ptrace_scope=0
```

Once GDB starts properly, you will be able to type commands into its prompt.

The first command we can execute is the next one to check what threads are running:

```
(gdb) info threads
  Id   Target Id                                    Frame
* 1    Thread 0x79d1f3883740 (LWP 14428) "test" 0x000079d1f3298d61 in
__futex_abstimed_wait_common64 (private=128, cancel=true, abstime=0x0,
op=265, expected=14429, futex_word=0x79d1f3000990)
    at ./nptl/futex-internal.c:57
  2    Thread 0x79d1f26006c0 (LWP 14430) "test" futex_wait (private=0,
expected=2, futex_word=0x7fff5e406b00) at ../sysdeps/nptl/futex-
```

```
internal.h:146
  3    Thread 0x79d1f30006c0 (LWP 14429) "test" futex_wait (private=0,
expected=2, futex_word=0x7fff5e406b30) at ../sysdeps/nptl/futex-
internal.h:146
```

The output shows that the `0x79d1f3883740` thread with GDB identifier `1` is the current one. If there are many threads and we are only interested in a specific subset, let's say threads 1 and 3, we can show information only for those threads by using the following command:

```
(gdb) info thread 1 3
```

Running a GDB command will affect the current thread. For example, running the `bt` command will show the backtrace for thread 1 (output simplified):

```
(gdb) bt
#0  0x000079d1f3298d61 in __futex_abstimed_wait_common64
(private=128, cancel=true, abstime=0x0, op=265, expected=14429, futex_
word=0x79d1f3000990) at ./nptl/futex-internal.c:57
#5  0x000061cbaf1174fd in main () at 11x18-debug_deadlock.cpp:22
```

To switch to another thread, for example, thread 2, we can use the `thread` command:

```
(gdb) thread 2
[Switching to thread 2 (Thread 0x79d1f26006c0 (LWP 14430))]
```

Now, the `bt` command will show the backtrace for thread 2 (output simplified):

```
(gdb) bt
#0  futex_wait (private=0, expected=2, futex_word=0x7fff5e406b00) at
../sysdeps/nptl/futex-internal.h:146
#2  0x000079d1f32a00f1 in lll_mutex_lock_optimized
(mutex=0x7fff5e406b00) at ./nptl/pthread_mutex_lock.c:48
#7  0x000061cbaf1173fa in operator() (__closure=0x61cbafd64418) at
11x18-debug_deadlock.cpp:19
```

To execute a command in different threads, just use the `thread apply` command, in this case, executing the `bt` command on threads 1 and 3:

```
(gdb) thread apply 1 3 bt
```

To execute a command in all threads, just use `thread apply all <command>`.

Note that when a breakpoint is reached in a multithreaded program, all threads of execution stop running, allowing the examination of the overall state of the program. When the execution is restarted by commands such as `continue`, `step`, or `next`, all threads resume. The current thread will move one statement forward, but that is not guaranteed for other threads, which could move forward several statements or even stop in the middle of a statement.

When the execution stops, the debugger will jump and show the context of the execution in the current thread. To avoid the debugger jumping between threads by locking the scheduler, we can use the following command:

```
(gdb) set scheduler-locking <on/off>
```

We can also use the following command to check the scheduler locking status:

```
(gdb) show scheduler-locking
```

Now that we have learned some new commands for multithreading debugging, let's check what is happening with the application we attached to the debugger.

If we retrieve the backtraces in threads 2 and 3, we can see the following (output simplified only showing the relevant parts):

```
(gdb) thread apply all bt
Thread 3 (Thread 0x79d1f30006c0 (LWP 14429) "test"):
#0  futex_wait (private=0, expected=2, futex_word=0x7fff5e406b30) at
../sysdeps/nptl/futex-internal.h:146
#5  0x000061cbaf117e20 in std::mutex::lock (this=0x7fff5e406b30) at /
usr/include/c++/14/bits/std_mutex.h:113
#7  0x000061cbaf117334 in operator() (__closure=0x61cbafd642b8) at
11x18-debug_deadlock.cpp:13
Thread 2 (Thread 0x79d1f26006c0 (LWP 14430) "test"):
#0  futex_wait (private=0, expected=2, futex_word=0x7fff5e406b00) at
../sysdeps/nptl/futex-internal.h:146
#5  0x000061cbaf117e20 in std::mutex::lock (this=0x7fff5e406b00) at /
usr/include/c++/14/bits/std_mutex.h:113
#7  0x000061cbaf1173fa in operator() (__closure=0x61cbafd64418) at
11x18-debug_deadlock.cpp:19
```

Note that, after running `std::mutex::lock()`, both threads are waiting at line 13 for thread 3 and at 19 for thread 2, which matches with `std::lock_guard lock2` in `std::thread t1` and `std::lock_guard lock1` in `std::thread t2`, respectively.

Therefore, we detected a deadlock happening in those threads at these code locations.

Let's now learn more about debugging multithreaded software by catching a race condition.

Debugging race conditions

Race conditions are one of the most difficult bugs to detect and debug because they usually occur sporadically and with different effects each time, and sometimes some expensive computation happens before the program reaches the point of failure.

This erratic behavior is not only caused by race conditions. Other issues related to incorrect memory allocation can cause similar symptoms, so it's not possible to classify a bug as a race condition until there is some investigation and we reach a root-cause diagnosis.

One way of debugging race conditions is to use watchpoints to manually inspect if a variable changes its value without any statement executed in the current thread modifying it, or placing breakpoints in strategic locations triggered by specific threads when reached, as shown here:

```
(gdb) break <linespec> thread <id> if <condition>
```

For example, see the following:

```
(gdb) break test.cpp:11 thread 2
```

Or, even using assertions and checking if the current value of any variable accessed by different threads has the expected value. This approach is followed in the next example:

```
#include <cassert>
#include <chrono>
#include <cmath>
#include <iostream>
#include <mutex>
#include <thread>

using namespace std::chrono_literals;

static int g_value = 0;
static std::mutex g_mutex;

void func1() {
    const std::lock_guard<std::mutex> lock(g_mutex);
    for (int i = 0; i < 10; ++i) {
        int old_value = g_value;
        int incr = (rand() % 10);
        g_value += incr;
        assert(g_value == old_value + incr);
        std::this_thread::sleep_for(10ms);
    }
}

void func2() {
    for (int i = 0; i < 10; ++i) {
        int old_value = g_value;
        int incr = (rand() % 10);
        g_value += (rand() % 10);
```

```
        assert(g_value == old_value + incr);
        std::this_thread::sleep_for(10ms);
    }
}

int main() {
    std::thread t1(func1);
    std::thread t2(func2);

    t1.join();
    t2.join();

    return 0;
}
```

Here, two threads, `t1` and `t2`, are running functions that increase the `g_value` global variable by a random value. Each time it is increased, `g_value` is compared with the expected value, and if they are not equal, the assert instruction will stop the program.

Compile this program and run the debugger as follows:

```
$ g++ -o test -g -O0 test
$ gdb ./test
```

After the debugger starts, run the program by using the `run` command. The program will run and, at some point, abort by receiving the `SIGABRT` signal, showing that the assertion was not met.

```
test: test.cpp:29: void func2(): Assertion `g_value == old_value +
incr' failed.
Thread 3 "test" received signal SIGABRT, Aborted.
```

With the program stopped, we can use the `backtrace` commands to check the backtrace at that point and change the source code at that point of failure to a specific `frame` or `list`.

This example is quite simple, so it's clear through checking the assertion output that something is going wrong with the `g_value` variable, and this is most probably a race condition.

But with a more elaborate program, this process of manually debugging issues is quite arduous, so let's focus on another technique called reverse debugging that can help with that.

Reverse debugging

Reverse debugging, also known as **time travel debugging**, allows a debugger to stop a program after failure and go back into the history of the execution of a program to investigate the reason for the failure. This functionality is achieved by logging (recording) the execution of each machine instruction

of the program being debugged together with each change in values of memory and registers, and afterward, using these records to replay and rewind the program at will.

On Linux, we can use GDB (since version 7.0), **rr** (originally developed by Mozilla, `https://rr-project.org`), or **Undo's time travel debugger** (**UDB**) (`https://docs.undo.io`). On Windows, we can use **Time Travel Debugging** (`https://learn.microsoft.com/en-us/windows-hardware/drivers/debuggercmds/time-travel-debugging-overview`).

Reverse debugging is only supported by a limited number of GDB targets, such as remote target Simics, **system integration and design** (**SID**) simulators, or the `process record and replay` target for native Linux (only for `i386`, `amd64`, `moxie-elf`, and `arm`). At the time of authoring this book, Clang's reverse debugging feature is still under development.

Therefore, due to these limitations, we decided to do a small showcase by using `rr`. Please follow the instructions on the project website to build and install the `rr` debugging tool: `https://github.com/rr-debugger/rr/wiki/Building-And-Installing`.

Once installed, to record and replay a program, use the following commands:

```
$ rr record <program> --args <args>
$ rr replay
```

For example, if we have a program called `test`, the command sequence will be as follows:

```
$ rr record test
rr: Saving execution to trace directory `/home/user/.local/share/rr/
test-1'.
```

Say the following fatal error is shown instead:

```
[FATAL src/PerfCounters.cc:349:start_counter()] rr needs /proc/sys/
kernel/perf_event_paranoid <= 3, but it is 4.
Change it to <= 3.
Consider putting 'kernel.perf_event_paranoid = 3' in /etc/
sysctl.d/10-rr.conf.
```

Then, use the following command to adjust the kernel variable, `kernel.perf_event_paranoid`:

```
$ sudo sysctl kernel.perf_event_paranoid=1
```

Once a record is available, use the `replay` command to start debugging the program:

```
$ rr replay
```

Or, if the program crashed and you want just to start debugging at the end of the recording, use the `-e` option:

```
$ rr replay -e
```

At this point, `rr` will use the GDB debugger to start the program and load its debug symbols. Then, you can use any of the following commands to reverse debugging:

- `reverse-continue`: Start executing the program in reverse. Execution will stop when a breakpoint is reached or due to a synchronous exception.
- `reverse-next`: Run backward to the beginning of the previous line executed in the current stack frame.
- `reverse-nexti`: This executes a single instruction in reverse, jumping those moving to inner stack frames.
- `reverse-step`: Run the program backward until the control reaches the start of a new source line.
- `reverse-stepi`: Execute in reverse one machine instruction.
- `reverse-finish`: This executes until the current function invocation, that is, the beginning of the current function.

We can also reverse the direction of debugging and use regular commands for forward debugging (such as `next`, `step`, `continue`, and so on) in the opposite direction by using the following command:

```
(rr) set exec-direction reverse
```

To set the execution direction back to forward, use the following command:

```
(rr) set exec-direction forward
```

As an exercise, install the `rr` debugger and try to debug the previous example using reverse debugging.

Let's now move on to how to debug coroutines, a challenging task due to their asynchronous nature.

Debugging coroutines

As we have seen so far, asynchronous code can be debugged as synchronous code, by using breakpoints in strategic places with specific conditions, using watchpoints to inspect variables, and stepping into or over to walk through the code. Also, using techniques described earlier to select specific threads and locking the scheduler helps to avoid unnecessary distractions when debugging.

As we have already learned, there are complexities in asynchronous code, such as what thread will be used when the asynchronous code is executed, which makes it more difficult to debug. For C++ coroutines, debugging is even harder to master due to their suspend/resume nature.

Clang compiles programs using coroutines in two steps: Semantic analysis is performed by Clang, and coroutine frames are constructed and optimized in the **LLVM** middle-end. As debug information is generated in the Clang frontend, there will be insufficient debug information as coroutine frames are generated later in the compilation process. GCC follows a similar approach.

Also, if the execution breaks inside a coroutine, the current frame will only have one variable, `frame_ptr`. In a coroutine, there are no pointer or function parameters. Coroutines store their state in the heap before suspending, and only using the stack during execution. `frame_ptr` is used to access all necessary information for the coroutine to run properly.

Let's debug the **Boost.Asio** coroutine example implemented in *Chapter 9*. Here, we only show the relevant instructions. Please visit the *Coroutines* section in *Chapter 9* to check the complete source code:

```
boost::asio::awaitable<void> echo(tcp::socket socket) {
    char data[1024];

    while (true) {
        std::cout << "Reading data from socket...\n";//L12
        std::size_t bytes_read = co_await
            socket.async_read_some(
                boost::asio::buffer(data),
                            boost::asio::use_awaitable);
        /* .... */

        co_await boost::asio::async_write(socket,
                boost::asio::buffer(data, bytes_read),
                boost::asio::use_awaitable);
    }
}

boost::asio::awaitable<void>
listener(boost::asio::io_context& io_context,
         unsigned short port) {
    tcp::acceptor acceptor(io_context,
                           tcp::endpoint(tcp::v4(), port));

    while (true) {
        std::cout << "Accepting connections...\n";  // L45
        tcp::socket socket = co_await
            acceptor.async_accept(
                boost::asio::use_awaitable);

        boost::asio::co_spawn(io_context,
            echo(std::move(socket)),
            boost::asio::detached);
    }
}

/* main function */
```

As we are using Boost, let's include the **Boost.System** library to add more symbols for debugging when compiling the source code:

```
$ g++ --std=c++20 -ggdb -O0 --fno-omit-frame-pointer -lboost_
system  test.cpp -o test
```

Then, we start the debugger with the generated program and set breakpoints in lines 12 and 45, which are the locations of the first instruction inside the while loops on each coroutine:

```
$ gdb -q ./test
(gdb) b 12
(gdb) b 45
```

We also enable the GDB built-in pretty printers to show readable output for Standard Template Library containers:

```
(gdb) set print pretty on
```

If now we run the program (the `run` command), it will reach the breakpoint at line 42 inside the coroutine listener before accepting connections. Using the `info locals` command, we can check the local variables.

Coroutines create a state machine with several internal fields, such as promise objects, with attached thread, address of caller object, pending exceptions, and so on. Also, they store `resume` and `destroy` callbacks. These structures are compiler-dependent, tied to the compiler's implementation, and accessible via `frame_ptr` if we are using Clang.

If we continue running the program (with the `continue` command), the server will be waiting for a client to connect. To exit the waiting status, we use `telnet`, as shown in *Chapter 9*, to connect a client to the server. At that point, the execution will be stopped because the breakpoint at line 12 inside the `echo()` coroutine is reached, and `info locals` show the variables used by each `echo` connection.

Using the `backtrace` command will show a call stack that might have some complexities due to the coroutines' suspend nature.

In pure C++ routines, described in *Chapter 8*, there are two expressions where setting breakpoints could be interesting:

- `co_await`: The execution is suspended until the awaited operation is complete. Breakpoints can be set at the point where the coroutine is resumed by inspecting the underlying `await_suspend`, `await_resume`, or custom awaitable code.
- `co_yield`: Suspends execution and yields a value. During debugging, step into the `co_yield` to observe how control flows between coroutines and their calling functions.

As coroutines are quite new in the C++ world and compilers are continuously evolving, we hope that soon debugging coroutines will be more straightforward.

Once we have found and debugged some bugs and can reproduce scenarios that lead to those specific bugs, it would be convenient to design some tests that cover those cases to avoid future changes in code that could lead to similar problems or incidents. Let's learn how to test multithreaded and asynchronous code in the next chapter.

Summary

In this chapter, we learned how to use logging and debug asynchronous programs.

We started by using logging to spot issues in running software, showing its usefulness in detecting deadlocks by using the `spdlog` logging library. Many other libraries were also discussed, describing their relevant features that might be suitable to specific scenarios.

However, not all bugs can be spotted by using logs, and some may only be detected later in the software development life cycle when some issues happen in production, even when dealing with program crashes and incidents. Debuggers are useful tools to inspect running or crashed programs, understand their code path, and find bugs. Several examples and debugger commands were introduced to deal with generic code, but also and especially with multithreaded and asynchronous software, race conditions, and coroutines. Also, the `rr` debugger was introduced, showing the potential of including reverse debugging in our developer toolbox.

In the next chapter, we will learn about using sanitizers and testing techniques to performance and optimization techniques that can be used to improve asynchronous programs' runtime and resource usage.

Further reading

- *Logging*: `https://en.wikipedia.org/wiki/Logging_(computing)`
- *Syslog*: `https://en.wikipedia.org/wiki/Syslog`
- *Google Logging Library*: `https://github.com/google/glog`
- *Apache Log4cxx*: `https://logging.apache.org/log4cxx`
- *spdlog*: `https://github.com/gabime/spdlog`
- *Quill*: `https://github.com/odygrd/quill`
- *xtr*: `https://github.com/choll/xtr`
- *lwlog*: `https://github.com/ChristianPanov/lwlog`
- *uberlog*: `https://github.com/IMQS/uberlog`
- *Easylogging++*: `https://github.com/abumq/easyloggingpp`
- *NanoLog*: `https://github.com/PlatformLab/NanoLog`

- *Reckless Logging Library*: https://github.com/mattiasflodin/reckless
- *tracetool*: https://github.com/froglogic/tracetool
- *Logback Project*: https://logback.qos.ch
- *Sentry*: https://sentry.io
- *Graylog*: https://graylog.org
- *Logstash*: https://www.elastic.co/logstash
- *Debugging with GDB*: https://sourceware.org/gdb/current/onlinedocs/gdb.html
- *LLDB tutorial*: https://lldb.llvm.org/use/tutorial.html
- *Clang Compiler User's Manual*: https://clang.llvm.org/docs/UsersManual.html
- *GDB: Running programs backward*: https://www.zeuthen.desy.de/dv/documentation/unixguide/infohtml/gdb/Reverse-Execution.html#Reverse-Execution
- *Reverse Debugging with GDB*: https://sourceware.org/gdb/wiki/ReverseDebug
- *Debugging C++ Coroutines*: https://clang.llvm.org/docs/DebuggingCoroutines.html
- *SID Simulator User's Guide*: https://sourceware.org/sid/sid-guide/book1.html
- *Intel Simics Simulator for Intel FPGAs: User Guide*: https://www.intel.com/content/www/us/en/docs/programmable/784383/24-1/about-this-document.html
- *IBM Support: How do I enable core dumps*: https://www.ibm.com/support/pages/how-do-i-enable-core-dumps
- *Core Dumps – How to enable them?*: https://medium.com/@sourabhedake/core-dumps-how-to-enable-them-73856a437711

12

Sanitizing and Testing Asynchronous Software

Testing is the process of evaluating and verifying that a software solution does what it's meant to do, validating its quality and ensuring that user requirements are met. With proper testing, we can prevent bugs from happening and improve performance.

In this chapter, we will explore several techniques to test asynchronous software, mainly using the **GoogleTest** library and sanitizers available from **GNU Compiler Collection** (**GCC**) and **Clang** compilers. Some prior knowledge in unit testing is required. In the *Further reading* section at the end of this chapter, you can find some references that could be useful to refresh and expand your knowledge in these areas.

In this chapter, we're going to cover the following main topics:

- Sanitizing code to analyze the software and find potential issues
- Testing asynchronous code

Technical requirements

For this chapter, we will need to install **GoogleTest** (`https://google.github.io/googletest`) to compile some of the examples.

Some examples need a compiler supporting C++20. Therefore, check the *Technical requirements* section in *Chapter 3*, as it includes some guidance on how to install GCC 13 and Clang 8 compilers.

You can find all the complete code in the following GitHub repository:

`https://github.com/PacktPublishing/Asynchronous-Programming-with-CPP`

The examples for this chapter are located under the `Chapter_12` folder. All source code files can be compiled using CMake as follows:

```
$ cmake . && cmake —build .
```

Executable binaries will be generated under the `bin` directory.

Sanitizing code to analyze the software and find potential issues

Sanitizers are tools, originally developed by Google, used to detect and prevent various types of issues or security vulnerabilities in code, helping developers catch bugs early in the development process, reducing the cost of issues being fixed late, and increasing software stability and security.

Sanitizers are usually integrated into development environments and are usually enabled during manual testing or when running unit tests, **continuous integration** (**CI**) pipelines, or code review pipelines.

C++ compilers, such as GCC and Clang, have compiler options to generate code when building the program to track the execution at runtime and report errors and vulnerabilities. They are implemented in Clang from version 3.1 and GCC from version 4.8.

As extra instructions are injected into the program's binary code, there is a performance penalty of around 1.5x to 4x slowdowns depending on the sanitizer type. Also, there is an overall memory overhead of 2x to 4x and a stack size increase of up to 3x. But note that slowdowns are much lower than the ones experienced when using other instrumentation frameworks or dynamic analysis tools, such as **Valgrind** (`https://valgrind.org`), which imposes a much higher slowdown of up to 50 times slower than production binaries. On the other hand, the benefit of using Valgrind is that no recompilation is needed. Both approaches only detect issues while the program is running and only on those code paths that the execution traverses. So, we need to ensure sufficient coverage.

There are also static analysis tools and linters, useful for detecting issues during compilation and checking all the code that is being included in the program. For example, compilers, such as GCC and Clang, can perform extra checks and provide useful information by enabling the `-Werror`, `-Wall`, and `-pedantic` options.

There are also open source alternatives, such as **Cppcheck** or **Flawfinder**, or commercial solutions that are free for open source projects, such as **PVS-Studio** or **Coverity Scan**. Other solutions, such as **SonarQube**, **CodeSonar**, or **OCLint**, can be used in **continuous integration / continuous delivery** (**CI/CD**) pipelines for ongoing quality tracking.

In this section, we will focus on sanitizers, which can be enabled by passing some special options to the compiler.

Compiler options

To enable sanitizers, we need to pass some compiler options when compiling the program.

The main option is `--fsanitize=sanitizer_name`, where `sanitizer_name` is one of the following options:

- `address`: This is for **AddressSanitizer** (**ASan**), to detect memory errors such as buffer overflows and use-after-free bugs
- `thread`: This is for **ThreadSanitizer** (**TSan**), to identify data races and other thread synchronization issues in multi-threaded programs by monitoring thread interactions
- `leak`: This is for **LeakSanitizer** (**LSan**), to spot memory leaks by tracking memory allocations and ensuring that all allocated memory is properly freed
- `memory`: This is for **MemorySanitizer** (**MSan**), to uncover the use of uninitialized memory
- `undefined`: This is for **UndefinedBehaviorSanitizer** (**UBSan**), to detect undefined behavior, such as integer overflows, invalid type casts, and other erroneous operations

Clang also includes `dataflow`, `cfi` (control flow integrity), `safe_stack`, and `realtime`.

GCC adds `kernel-address`, `hwaddress`, `kernel-hwaddress`, `pointer-compare`, `pointer-subtract`, and `shadow-call-stack`.

As this list and flag behavior can change over time, it's recommended to check the compilers' official documentation.

Additional flags might be needed:

- `-fno-omit-frame-pointer`: A **frame pointer** is a register used by compilers to track the current stack frame, containing, among other information, the base address of the current function. Omitting frame pointers might increase the performance of the program but at the cost of making debugging significantly harder; it makes it more difficult to locate local variables and reconstruct stack traces.
- `-g`: Include debug information and display filenames and line numbers in the warning messages. If the debugger GDB is used, the `-ggdb` option might be desirable as the compiler can produce more expressive symbols to be used when debugging. Also, a level can be specified by using `-g[level]`, with `[level]` being a value from `0` to `3`, adding more debug information at each level increase. The default level is `2`.
- `-fsanitize-recover`: These options cause the sanitizer to attempt to continue running the program as if no error was detected.
- `-fno-sanitize-recover`: The sanitizer will detect only the first error, and the program will exit with a non-zero exit code.

To keep a reasonable performance, we might need to adjust the optimization level by specifying the `-O[num]` option. Different sanitizers work best up to a certain level of optimization. It's best to start with `-O0` and, if the slowdown is significant, try to increase to `-O1`, `-O2`, and so on. Also, as different sanitizers and compilers recommend specific optimization levels, check their documentation.

When using Clang, to make stack traces easy to understand and let sanitizers convert addresses into source code locations, apart from using the flags mentioned earlier, we can also set the specific environment variable, `[X]SAN_SYMBOLIZER_PATH`, to the location of `llvm-symbolizer` (with `[X]` being `A` for AddressSatinizer, `L` for LSan, `M` for MSan, and so on). We can also include this location in the `PATH` environment variable. Here is an example of setting the `PATH` variables when using **AddressSatinizer**:

```
export ASAN_SYMBOLIZER_PATH=`which llvm-symbolizer`
export PATH=$ASAN_SYMBOLIZER_PATH:$PATH
```

Note that enabling `-Werror` with certain sanitizers can lead to false positives. Also, other compiler flags might be needed, but warning messages during execution will show that a problem is happening and will be evident that a flag is needed. Check the sanitizers' and compilers' documentation to find which flag to use in those cases.

Avoiding sanitizing part of the code

Sometimes, we may want to silence some sanitizer warning and skip sanitizing some functions due to the following reasons: it is a well-known issue, the function is correct, it's a false positive, this function needs to speed up, or it is an issue in a third-party library. In those cases, we can use suppression files or exclude the code area by using some macro instructions. There is also a blacklist mechanism, but as it is deprecated in favor of suppression files; we will not comment on it here.

With suppression files, we just need to create a text file listing the areas of the code where we don't want the sanitizer to run. Each line consists of a pattern following a specific format depending on the sanitizer, but typically, the structure is as follows:

```
type:location_pattern
```

Here, `type` indicates the type of suppression, for example, the `leak` and `race` values, and `location_pattern` is a regular expression matching the function or library name to suppress. Here is an example of a suppression file for an ASan, explained in the next section:

```
# Suppress known memory leaks in third-party function Func1 in library
Lib1
leak:Lib1::Func1
# Ignore false-positive from function Func2 in library Lib2
race:Lib2::Func2
# Suppress issue from libc
leak:/usr/lib/libc.so.*
```

Let's call this file `myasan.supp`. Then, compile and pass this suppression file to the sanitizer via `[X]SAN_OPTIONS` as follows:

```
$ clang++ -O0 -g -fsanitize=address -fno-omit-frame-pointer test.cpp
-o test
$ ASAN_OPTIONS=suppressions=myasan.supp ./test
```

We can also use macros in source code to exclude specific functions to be sanitized by using `__attribute__((no_sanitize("<sanitizer_name>")))` as follows:

```
#if defined(__clang__) || defined (__GNUC__)
# define ATTRIBUTE_NO_SANITIZE_ADDRESS __attribute__((no_sanitize_
address))
#else
# define ATTRIBUTE_NO_SANITIZE_ADDRESS
#endif
...
ATTRIBUTE_NO_SANITIZE_ADDRESS
void ThisFunctionWillNotBeInstrumented() {...}
```

This technique provides a fine-grained compile-time control over what should be instrumented by the sanitizer.

Let's now explore the most common types of code sanitizers, starting with one of the most relevant to check address misusages.

AddressSanitizer

The purpose of ASan is to detect memory-related errors happening due to buffer overflows (heap, stack, and global) during out-of-bounds accesses of arrays, using a block of memory after being released with free or delete operations, and other memory leaks.

Apart from setting `-fsanitize=address` and other flags recommended earlier, we can also use `–fsanitize-address-use-after-scope` to detect the memory used after moving out of scope or setting the `ASAN_OPTIONS=option detect_stack_use_after_return=1` environment variable to detect use after return.

`ASAN_OPTIONS` can also be used to instruct the ASan to print the stack trace or set a log file as follows:

```
ASAN_OPTIONS=detect_stack_use_after_return=1,print_stacktrace=1,log_
path=asan.log
```

Clang on Linux has full support for ASan, followed by GCC on Linux. By default, ASan is disabled as it adds extra runtime overhead.

Also, ASan processes all calls to `glibc` – the GNU C library providing the core libraries for GNU systems. However, this is not the case with other libraries, so it's recommended to recompile such libraries with the `-fsanitize=address` option. As commented earlier, with Valgrind, recompilation is not required.

ASan can be combined with UBSan, which we will see later, but It slows down the performance by around 50%.

If we want a more aggressive diagnostics sanitizing, we can use the following flag combination:

```
ASAN_OPTIONS=strict_string_checks=1:detect_stack_use_after_
return=1:check_initialization_order=1:strict_init_order=1
```

Let's see two examples of using ASan to detect common software issues, with memory being used after being freed and detecting buffer overflows.

Memory usage after being freed

One common issue in software is using memory after being freed. In this example, memory allocated in the heap is being used after being deleted:

```
#include <iostream>
#include <memory>

int main() {
  auto arr = new int[100];
  delete[] arr;
  std::cout << "arr[0] = " << arr[0] << '\n';
  return 0;
}
```

Let's suppose that the previous source code is in a file called `test.cpp`. To enable ASan, we just compile the file using the following command:

```
$ clang++ -fsanitize=address -fno-omit-frame-pointer -g -O0 –o test
test.cpp
```

Then, executing the resulting output `test` program, we obtain the following output (note that the output is simplified, only showing relevant content and might differ from different compiler versions and execution context):

```
ERROR: AddressSanitizer: heap-use-after-free on address 0x514000000040
at pc 0x63acc82a0bec bp 0x7fff2d096c60 sp 0x7fff2d096c58
READ of size 4 at 0x514000000040 thread T0
    #0 0x63acc82a0beb in main test.cpp:7:31
0x514000000040 is located 0 bytes inside of 400-byte region
[0x514000000040,0x5140000001d0)
```

```
freed by thread T0 here:
    #0 0x63acc829f161 in operator delete[](void*) (/
mnt/StorePCIE/Projects/Books/Packt/Book/Code/build/bin/
Chapter_11/11x02-ASAN_heap_use_after_free+0x106161) (BuildId:
7bf8fe6b1f86a8b587fbee39ae3a5ced3e866931)
previously allocated by thread T0 here:
    #0 0x63acc829e901 in operator new[](unsigned long) (/
mnt/StorePCIE/Projects/Books/Packt/Book/Code/build/bin/
Chapter_11/11x02-ASAN_heap_use_after_free+0x105901) (BuildId:
7bf8fe6b1f86a8b587fbee39ae3a5ced3e866931)
SUMMARY: AddressSanitizer: heap-use-after-free test.cpp:7:31 in main
```

The output shows that the ASan was applied and detected a heap-use-after-free error. This error is happening in the `T0` thread (main thread). The output also points to the code where that memory region was allocated, and later freed, and its size (400 bytes region).

These kinds of errors not only happen with heap memory but also with memory regions allocated in the stack or global area. ASan can be used to detect these kinds of issues, such as memory overflows.

Memory overflows

Memory overflows, also known as buffer overflows or overruns, happen when some data is written in a memory address past the allocated memory of a buffer.

The following example shows a heap memory overflow:

```
#include <iostream>

int main() {
  auto arr = new int[100];
  arr[0] = 0;
  int res = arr[100];
  std::cout << "res = " << res << '\n';
  delete[] arr;
  return 0;
}
```

After compiling and running the resulting program, this is the output:

```
ERROR: AddressSanitizer: heap-buffer-overflow on address
0x5140000001d0 at pc 0x582953d2ac07 bp 0x7ffde9d58910 sp
0x7ffde9d58908
READ of size 4 at 0x5140000001d0 thread T0
    #0 0x582953d2ac06 in main test.cpp:6:13
0x5140000001d0 is located 0 bytes after 400-byte region
[0x514000000040,0x5140000001d0)
allocated by thread T0 here:
```

```
    #0 0x582953d28901 in operator new[](unsigned long) (test+0x105901)
(BuildId: 82a16fc86e01bc81f6392d4cbcad0fe8f78422c0)
    #1 0x582953d2ab78 in main test.cpp:4:14
(test+0x2c374) (BuildId: 82a16fc86e01bc81f6392d4cbcad0fe8f78422c0)
SUMMARY: AddressSanitizer: heap-buffer-overflow test.cpp:6:13 in main
```

As we can see from the output, now ASan reports a heap-buffer-overflow error in the main thread (`T0`) when accessing a memory address beyond a 400-byte region (the `arr` variable).

A sanitizer that is integrated into ASan is LSan. Let's learn now how to detect memory leaks using this sanitizer.

LeakSanitizer

LSan is used to detect memory leaks happening when memory has been allocated but not properly freed.

LSan is integrated into ASan and enabled by default on Linux systems. It can be enabled on macOS by using `ASAN_OPTIONS=detect_leaks=1`. To disable it, just set `detect_leaks=0`.

If the `-fsanitize=leak` option is used, the program will link against a subset of the ASan supporting LSan, disabling compile-time instrumentation and reducing the ASan slowdown. Note that this mode is not as well tested as the default mode.

Let's see an example of memory leak:

```
#include <string.h>
#include <iostream>
#include <memory>

int main() {
    auto arr = new char[100];
    strcpy(arr, "Hello world!");
    std::cout << "String = " << arr << '\n';
    return 0;
}
```

In this example, 100 bytes are allocated (the `arr` variable) but never freed.

To enable LSan, we just compile the file using the following command:

```
$ clang++ -fsanitize=leak -fno-omit-frame-pointer -g -O2 –o test test.
cpp
```

Running the resulting test binary, we obtain the following result:

```
ERROR: LeakSanitizer: detected memory leaks
Direct leak of 100 byte(s) in 1 object(s) allocated from:
```

```
    #0 0x5560ba9a017c in operator new[](unsigned long) (test+0x3417c)
(BuildId: 2cc47a28bb898b4305d90c048c66fdeec440b621)
    #1 0x5560ba9a2564 in main test.cpp:6:16
SUMMARY: LeakSanitizer: 100 byte(s) leaked in 1 allocation(s).
```

LSan correctly reports that a memory region of 100 bytes was allocated by using the operator `new` but never deleted.

As this book explores multithreading and asynchronous programming, let's learn now about a sanitizer to detect data races and other thread issues: TSan.

ThreadSanitizer

TSan is used to detect threading issues, especially data races and synchronization issues. It cannot be combined with ASan or LSan. TSan is the sanitizer most aligned with the content of this book.

This sanitizer is enabled by specifying the `-fsanitize=thread` compiler option and its behavior can be modified by using the `TSAN_OPTIONS` environment variable. For example, if we want to stop after the first error, just use the following:

```
TSAN_OPTIONS=halt_on_error=1
```

Also, for a reasonable performance, use the compiler's `-O2` option.

TSan only reports race conditions happening at runtime, thus it won't alert on race conditions present in code paths not executed at runtime. Therefore, we need to design tests that provide good coverage and use a realistic workload.

Let's see some examples of TSan detecting data races. In the next example, we'll do this by using a global variable without protecting its access with a mutex:

```
#include <thread>

int globalVar{0};

void increase() {
  globalVar++;
}

void decrease() {
  globalVar--;
}

int main() {
  std::thread t1(increase);
```

```
  std::thread t2(decrease);

  t1.join();
  t2.join();
  return 0;
}
```

After compiling the program, use the following command to enable TSan:

```
$ clang++ -fsanitize=thread -fno-omit-frame-pointer -g -O2 –o test
test.cpp
```

Running the resulting program generates the following output:

```
WARNING: ThreadSanitizer: data race (pid=31692)
  Write of size 4 at 0x5932b0585ae8 by thread T2:
    #0 decrease() test.cpp:10:12 (test+0xe0b32) (BuildId:
895b75ef540c7b44daa517a874d99d06bd27c8f7)
  Previous write of size 4 at 0x5932b0585ae8 by thread T1:
    #0 increase() test.cpp:6:12 (test+0xe0af2) (BuildId:
895b75ef540c7b44daa517a874d99d06bd27c8f7)
  Thread T2 (tid=31695, running) created by main thread at:
    #0 pthread_create <null> (test+0x6062f) (BuildId:
895b75ef540c7b44daa517a874d99d06bd27c8f7)
  Thread T1 (tid=31694, finished) created by main thread at:
    #0 pthread_create <null> (test+0x6062f) (BuildId:
895b75ef540c7b44daa517a874d99d06bd27c8f7)
SUMMARY: ThreadSanitizer: data race test.cpp:10:12 in decrease()
ThreadSanitizer: reported 1 warnings
```

From the output, it's clear that there is a data race when accessing `globalVar` in the `increase()` and `decrease()` functions.

If we decide to use GCC instead of Clang, the following error can be reported when running the resulting program:

```
FATAL: ThreadSanitizer: unexpected memory mapping 0x603709d10000-
0x603709d11000
```

This memory mapping issue is caused by a security feature called **address space layout randomization** (**ASLR**), a memory-protection technique used by the OS to protect against buffer overflow attacks by randomizing the address space of processes.

One solution is to reduce ASLR by using the following command:

```
$ sudo sysctl vm.mmap_rnd_bits=30
```

The value passed to `vm.mmap_rnd_bits` (`30` in the preceding command) can be reduced further if the error is still happening. To check that the value is correctly set, just run the following:

```
$ sudo sysctl vm.mmap_rnd_bits
vm.mmap_rnd_bits = 30
```

Note that this change is not permanent. Therefore, when the machine reboots, its value will be set to the default one. To persist this change, add `m.mmap_rnd_bits=30` to `/etc/sysctl.conf`.

But that reduces the security of the system, so it might be preferable to temporarily disable ASLR for a particular program by using the following command:

```
$ setarch `uname -m` -R ./test
```

Running the preceding command will show a similar output to what was shown earlier when compiling with Clang.

Let's move to another example where a `std::map` object is accessed without a mutex. Even if the map is being accessed for different key values, as writing to a `std::map` invalidates their iterators, that can cause data races:

```
#include <map>
#include <thread>

std::map<int,int> m;

void Thread1() {
  m[123] = 1;
}

void Thread2() {
  m[345] = 0;
}

int main() {
  std::jthread t1(Thread1);
  std::jthread t2(Thread1);
  return 0;
}
```

Compiling and running the resulting binary generates a large output with three warnings. Here, we only show the most relevant lines of the first warning (other warnings are similar):

```
WARNING: ThreadSanitizer: data race (pid=8907)
  Read of size 4 at 0x720c00000020 by thread T2:
  Previous write of size 8 at 0x720c00000020 by thread T1:
```

```
  Location is heap block of size 40 at 0x720c00000000 allocated by
thread T1:
  Thread T2 (tid=8910, running) created by main thread at:
  Thread T1 (tid=8909, finished) created by main thread at:
SUMMARY: ThreadSanitizer: data race test.cpp:11:3 in Thread2()
```

The TSan warnings are flagged when both the `t1` and `t2` threads are writing into the map, `m`.

In the next example, there is only one auxiliary thread accessing the map via a pointer, but this thread is competing against the main thread to access and use the map. The `t` thread accesses the map, `m`, to change the value for the `foo` key; meanwhile, the main thread prints its value to the console:

```
#include <iostream>
#include <thread>
#include <map>
#include <string>

typedef std::map<std::string, std::string> map_t;

void *func(void *p) {
  map_t& m = *static_cast<map_t*>(p);
  m["foo"] = "bar";
  return 0;
}

int main() {
  map_t m;
  std::thread t(func, &m);
  std::cout << "foo = " << m["foo"] << '\n';
  t.join();
  return 0;
}
```

Compiling and running this example generates a massive output with seven TSan warnings. Here, we only show the first warning. Feel free to check the complete report by compiling and running the example in the GitHub repository:

```
WARNING: ThreadSanitizer: data race (pid=10505)
  Read of size 8 at 0x721800003028 by main thread:
    #8 main test.cpp:17:28 (test+0xe1d75) (BuildId:
8eef80df1b5c81ce996f7ef2c44a6c8a11a9304f)
  Previous write of size 8 at 0x721800003028 by thread T1:
    #0 operator new(unsigned long) <null> (test+0xe0c3b) (BuildId:
8eef80df1b5c81ce996f7ef2c44a6c8a11a9304f)
    #9 func(void*) test.cpp:10:3 (test+0xe1bb7) (BuildId:
```

```
8eef80df1b5c81ce996f7ef2c44a6c8a11a9304f)
  Location is heap block of size 96 at 0x721800003000 allocated by
thread T1:
    #0 operator new(unsigned long) <null> (test+0xe0c3b) (BuildId:
8eef80df1b5c81ce996f7ef2c44a6c8a11a9304f)
    #9 func(void*) test.cpp:10:3 (test+0xe1bb7) (BuildId:
8eef80df1b5c81ce996f7ef2c44a6c8a11a9304f)
  Thread T1 (tid=10507, finished) created by main thread at:
    #0 pthread_create <null> (test+0x616bf) (BuildId:
8eef80df1b5c81ce996f7ef2c44a6c8a11a9304f)
SUMMARY: ThreadSanitizer: data race test.cpp:17:28 in main
ThreadSanitizer: reported 7 warnings
```

From the output, TSan is warning about a data race when accessing a `std::map` object allocated in the heap. That object is the map `m`.

However, TSan can not only detect data races due to a lack of mutexes but can also report when a variable must be atomic.

The next example shows that scenario. The `RefCountedObject` class defines objects that can keep a reference count of how many objects of that class have been created. Smart pointers follow this idea to delete the underlying allocated memory on destruction when the counter reaches the value `0`. In this example, we are only showing the `Ref()` and `Unref()` functions that increment and decrement the reference count variable, `ref_`. To avoid issues in a multithreading environment, `ref_` must be an atomic variable. As here, this is not the case, and the `t1` and `t2` threads are modifying `ref_`, a possible data race can happen:

```
#include <iostream>
#include <thread>

class RefCountedObject {
   public:

    void Ref() {
        ++ref_;
    }

    void Unref() {
        --ref_;
    }

   private:
    // ref_ should be atomic to avoid synchronization issues
    int ref_{0};
};
```

```
int main() {
  RefCountedObject obj;
  std::jthread t1(&RefCountedObject::Ref, &obj);
  std::jthread t2(&RefCountedObject::Unref, &obj);
  return 0;
}
```

Compiling and running this example shows the following output:

```
WARNING: ThreadSanitizer: data race (pid=32574)
  Write of size 4 at 0x7ffffffcc04 by thread T2:
    #0 RefCountedObject::Unref() test.cpp:12:9 (test+0xe1dd0)
(BuildId: 448eb3f3d1602e21efa9b653e4760efe46b621e6)
  Previous write of size 4 at 0x7ffffffcc04 by thread T1:
    #0 RefCountedObject::Ref() test.cpp:8:9 (test+0xe1c00) (BuildId:
448eb3f3d1602e21efa9b653e4760efe46b621e6)
  Location is stack of main thread.
  Location is global '??' at 0x7fffffdd000 ([stack]+0x1fc04)
  Thread T2 (tid=32577, running) created by main thread at:
    #0 pthread_create <null> (test+0x6164f) (BuildId:
448eb3f3d1602e21efa9b653e4760efe46b621e6)
    #2 main test.cpp:23:16 (test+0xe1b94) (BuildId:
448eb3f3d1602e21efa9b653e4760efe46b621e6)
  Thread T1 (tid=32576, finished) created by main thread at:
    #0 pthread_create <null> (test+0x6164f) (BuildId:
448eb3f3d1602e21efa9b653e4760efe46b621e6)
    #2 main test.cpp:22:16 (test+0xe1b56) (BuildId:
448eb3f3d1602e21efa9b653e4760efe46b621e6)
SUMMARY: ThreadSanitizer: data race test.cpp:12:9 in
RefCountedObject::Unref()
ThreadSanitizer: reported 1 warnings
```

TSan output shows that there is a data race condition happening in the `Unref()` function when accessing a memory location previously modified by the `Ref()` function.

Data races can also happen in objects being initialized from several threads without any synchronization mechanism. In the following example, an object of type `MyObj` is being created in the `init_object()` function, and the global static pointer, `obj`, is assigned its address. As this pointer is not protected by a mutex, there is a data race happening when the `t1` and `t2` threads try to create an object and update the `obj` pointer from the `func1()` and `func2()` functions respectively:

```
#include <iostream>
#include <thread>

class MyObj {};
```

```
static MyObj *obj = nullptr;

void init_object() {
  if (!obj) {
    obj = new MyObj();
  }
}

void func1() {
  init_object();
}

void func2() {
  init_object();
}

int main() {
  std::thread t1(func1);
  std::thread t2(func2);

  t1.join();
  t2.join();
  return 0;
}
```

This is the output after compiling and running this example:

```
WARNING: ThreadSanitizer: data race (pid=32826)
  Read of size 1 at 0x5663912cbae8 by thread T2:
    #0 func2() test.cpp (test+0xe0b68) (BuildId:
12f32c1505033f9839d17802d271fc869b7a3e38)
  Previous write of size 1 at 0x5663912cbae8 by thread T1:
    #0 func1() test.cpp (test+0xe0b3d) (BuildId:
12f32c1505033f9839d17802d271fc869b7a3e38)
  Location is global 'obj (.init)' of size 1 at 0x5663912cbae8
(test+0x150cae8)
  Thread T2 (tid=32829, running) created by main thread at:
    #0 pthread_create <null> (test+0x6062f) (BuildId:
12f32c1505033f9839d17802d271fc869b7a3e38)
  Thread T1 (tid=32828, finished) created by main thread at:
    #0 pthread_create <null> (test+0x6062f) (BuildId:
12f32c1505033f9839d17802d271fc869b7a3e38)
SUMMARY: ThreadSanitizer: data race test.cpp in func2()
ThreadSanitizer: reported 1 warnings
```

The output shows what we described earlier, a data race happening due to access to the `obj` global variable from `func1()` and `func2()`.

As the C++11 standard has officially deemed data races as undefined behavior, let's see now how to use UBSan to detect undefined behavior issues in the program.

UndefinedBehaviorSanitizer

UBSan can detect undefined behavior in code, for example, when shifting bits by an excessive amount, integer overflows, or misuse of null pointers. It can be enabled by specifying the `-fsanitize=undefined` option. Its behavior can be modified at runtime by setting the `UBSAN_OPTIONS` variable.

Many errors that can be detected by UBSan are also detected by the compiler during compilation.

Let's see a simple example:

```
int main() {
  int val = 0x7fffffff;
  val += 1;
  return 0;
}
```

To compile the program and enable UBSan, use the following command:

```
$ clang++ -fsanitize=undefined -fno-omit-frame-pointer -g -O2 -o test
test.cpp
```

Running the resulting program generates the following output:

```
test.cpp:3:7: runtime error: signed integer overflow: 2147483647 + 1
cannot be represented in type 'int'
SUMMARY: UndefinedBehaviorSanitizer: undefined-behavior test.cpp:3:7
```

The output is quite simple and self-explanatory; there is a signed integer overflow operation.

Let's now learn about another useful C++ sanitizer to detect uninitialized memory and other memory usage issues: MSan.

MemorySanitizer

MSan can detect uninitialized memory usage, for example, when using variables or pointers before they have been assigned a value or address. It can also track uninitialized bits in a bitfield.

To enable MSan, use the following compiler flags:

```
-fsanitize=memory -fPIE -pie -fno-omit-frame-pointer
```

It can also track each uninitialized value to the memory allocation from where it was created by specifying the `-fsanitize-memory-track-origins` option.

GCC has no support for MSan, so the `-fsanitize=memory` flag is not valid when using this compiler.

In the following example, the `arr` integer array is created, but only its position `5` is initialized. The value at position `0` is used when printing the message to the console, but this value is still uninitialized:

```
#include <iostream>

int main() {
  auto arr = new int[10];
  arr[5] = 0;
  std::cout << "Value at position 0 = " << arr[0] << '\n';
  return 0;
}
```

To compile the program and enable MSan, use the following command:

```
$ clang++ -fsanitize=memory -fno-omit-frame-pointer -g -O2 -o test
test.cpp
```

Running the resulting program generates the following output:

```
==20932==WARNING: MemorySanitizer: use-of-uninitialized-value
    #0 0x5b9fa2bed38f in main test.cpp:6:41
    #3 0x5b9fa2b53324 in _start (test+0x32324) (BuildId:
c0a0d31f01272c3ed59d4ac66b8700e9f457629f)
SUMMARY: MemorySanitizer: use-of-uninitialized-value test.cpp:6:41 in
main
```

Again, the output shows clearly that an uninitialized value is being used at line 6 when reading the value at position `0` in the `arr` array.

Finally, let's summarize other sanitizers in the next section.

Other sanitizers

There are other sanitizers available that are useful when developing for certain systems, such as kernel or real-time development:

- **Hardware-assisted AddressSanitizers (HWASan)**: A new variant of ASan that consumes much less memory by using the hardware ability to ignore the top byte of a pointer. It can be enabled by specifying the `–fsanitize=hwaddress` option.
- **RealTimeSanitizer (RTSan)**: Real-time testing tool to detect real-time violations when calling methods that are not safe in functions with deterministic runtime requirements.

- **FuzzerSanitizer**: A sanitizer that detects potential vulnerabilities by feeding large volumes of random data into the program, checking if the program crashes, and looking for memory corruption or other security vulnerabilities.
- **Kernel-related sanitizers**: There are also sanitizers available to track issues by kernel developers. For the sake of curiosity, some of these are as follows:
  - **Kernel Address Sanitizer** (**KASAN**)
  - **Kernel Concurrency Sanitizer** (**KCSAN**)
  - **Kernel Electric-Fence** (**KFENCE**)
  - **Kernel Memory Sanitizer** (**KMSAN**)
  - **Kernel Thread Sanitizer** (**KTSAN**)

Sanitizers can automatically find many issues in our code. Once we have found and debugged some bugs and can reproduce scenarios that lead to those specific bugs, it would be convenient to design some tests that cover those cases to avoid future changes in code that could lead to similar problems or incidents.

Let's learn how to test multithreaded and asynchronous code in the next section.

Testing asynchronous code

Finally, let's explore some techniques to test asynchronous code. The examples shown in this section need **GoogleTest** and **GoogleTest Mock** (**gMock**) libraries to compile. If you are unfamiliar with these libraries, please check the official documentation on how to install and use them.

As we know, **unit testing** is the practice of writing small and isolated tests that verify the functionality and behavior of a single unit of code. Unit testing helps to find and fix bugs, refactor and improve your code quality, document and communicate the underlying code design, and facilitate collaboration and integration.

This section will not cover the best way to group tests into logical and descriptive suites, or when you should use assertions or expectations to verify the values of different variables and tested methods outcomes. The purpose of this section is to provide some guidelines on how to create unit tests to test asynchronous code. Therefore, some previous knowledge about unit testing or **test-driven development** (**TDD**) is desirable.

The main difficulty when dealing with asynchronous code is that it might execute in another thread, and usually without knowing when that will happen, or when it will complete.

The main approach to follow when testing asynchronous code is to try to separate the functionality from multithreading, meaning that we might want to test the asynchronous code in a synchronous way, trying to execute it in one specific thread, removing context switching, threads creation and

destruction, and other activities that might affect the result and timings on the tests. Sometimes, timers are also used, waiting for a callback to be invoked before timeout.

Testing a simple asynchronous function

Let's start with a small example of testing an asynchronous operation. This example shows a `asyncFunc()` function that is tested by running it asynchronously by using `std::async`, as shown in *Chapter 7*:

```
#include <gtest/gtest.h>
#include <chrono>
#include <future>

using namespace std::chrono_literals;

int asyncFunc() {
    std::this_thread::sleep_for(100ms);
    return 42;
}

TEST(AsyncTests, TestHandleAsyncOperation) {
    std::future<int> result = std::async(
                          std::launch::async,
                          asyncFunc);
    EXPECT_EQ(result.get(), 42);
}

int main(int argc, char **argv) {
    ::testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

`std::async` returns a future that is used to retrieve the computed value. In this case, `asyncFunc` just waits for `100ms` before returning the value `42`. If the asynchronous task runs properly, the test will pass as there is an expectation instruction checking that the returned value is in fact `42`.

There is only one test defined, using the `TEST()` macro, where its first parameter is the test suite name (in this example, `AsyncTests`) and the second parameter is the test name (`TestHandleAsyncOperation`).

In the `main()` function, the GoogleTest library is initialized by calling `::testing::InitGoogleTest()`. This function parses the command line for the flags that GoogleTest recognizes. Then, `RUN_ALL_TESTS()` is called, which collects and runs all tests and returns `0` if all tests are successful or `1` otherwise. This function originally was a macro, which is why its name is in uppercase.

Limiting test durations by using timeouts

One issue that could happen with this approach is that the asynchronous task can fail to be scheduled for any reason, take longer than expected to complete, or just not get completed for any reason. To deal with this situation, a timer can be used, setting its timeout period to a reasonable value to give enough time for the test to complete successfully. Therefore, if the timer times out, the test will fail. The following example shows that approach by using a timed waiting on the future returned by `std::async`:

```
#include <gtest/gtest.h>
#include <chrono>
#include <future>

using namespace std::chrono;
using namespace std::chrono_literals;

int asyncFunc() {
    std::this_thread::sleep_for(100ms);
    return 42;
}

TEST(AsyncTest, TestTimeOut) {
    auto start = steady_clock::now();
    std::future<int> result = std::async(
                          std::launch::async,
                          asyncFunc);

    if (result.wait_for(200ms) ==
                std::future_status::timeout) {
        FAIL() << "Test timed out!";
    }

    EXPECT_EQ(result.get(), 42);

    auto end = steady_clock::now();
    auto elapsed = duration_cast<milliseconds>(
                                 end - start);
    EXPECT_LT(elapsed.count(), 200);
}

int main(int argc, char** argv) {
    ::testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

Now, the `wait_for()` function of the future object `result` is called, waiting 200 ms for the asynchronous task to complete. As the task will be completed in 100 ms, the timeout will not expire. If for any reason `wait_for()` is called with a value lower than 100 ms, it would time out and the `FAIL()` macro will be called, making the test fail.

The test continues running and checks if the returned value is `42` as in the previous example, and then also checks if the time spent running the asynchronous task is less than the used timeout.

Testing callbacks

Testing callback is a relevant task, especially when implementing libraries and **application programming interfaces** (**APIs**). The following example shows how to test that a callback has been called and its result:

```
#include <gtest/gtest.h>
#include <chrono>
#include <functional>
#include <iostream>
#include <thread>

using namespace std::chrono_literals;

void asyncFunc(std::function<void(int)> callback) {
    std::thread([callback]() {
        std::this_thread::sleep_for(1s);
        callback(42);
    }).detach();
}

TEST(AsyncTest, TestCallback) {
    int result = 0;
    bool callback_called = false;

    auto callback = [&](int value) {
        callback_called = true;
        result = value;
    };

    asyncFunc(callback);

    std::this_thread::sleep_for(2s);
    EXPECT_TRUE(callback_called);
    EXPECT_EQ(result, 42);
}
```

```
int main(int argc, char** argv) {
    ::testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

The `TestCallback` test just defines a callback as a lambda function that accepts an argument. This lambda function captures by reference the `result` variable where the `value` argument is stored, and the `callback_called` Boolean variable that by default is `false` and set to `true` when the callback is called.

Then, the test calls the `asyncFunc()` function that spawns a thread that waits for one second before calling the callback and passing the value `42`. The test waits for two seconds before checking if the callback has been called by using the `EXPECT_TRUE` macro and checking the value of `callback_called`, and if `result` has the expected value of `42`.

Testing event-driven software

We saw in *Chapter 9* how to use **Boost.Asio** and its event queue to dispatch asynchronous tasks. In event-driven programming, typically, we also need to test callbacks, as in the previous example. We can set up the test to inject callbacks and validate the result after they are called. The following example shows how to test asynchronous tasks in a Boost.Asio program:

```
#include <gtest/gtest.h>
#include <boost/asio.hpp>
#include <chrono>
#include <thread>

using namespace std::chrono_literals;

void asyncFunc(boost::asio::io_context& io_context,
               std::function<void(int)> callback) {
    io_context.post([callback]() {
        std::this_thread::sleep_for(100ms);
        callback(42);
    });
}

TEST(AsyncTest, BoostAsio) {
    boost::asio::io_context io_context;

    int result = 0;
    asyncFunc(io_context, [&result](int value) {
        result = value;
    });
```

```
    std::jthread io_thread([&io_context]() {
        io_context.run();
    });

    std::this_thread::sleep_for(150ms);
    EXPECT_EQ(result, 42);
}

int main(int argc, char** argv) {
    ::testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

The `BoostAsio` test starts by creating an I/O execution context object, `io_context`, and passing it to the `asyncFunc()` function together with a lambda function implementing a task or callback to run in the background. This callback simply sets the value of the `result` variable, captured by the lambda function, to the value passed to it.

The `asyncFunc()` function just uses `io_context` to post a task that consists of a lambda function that calls the callback with the value `42` after waiting for 100 ms.

The test then just waits for 150 ms for the background task to finish and checks that the result value is `42` to mark the test as passed.

Mocking external resources

If the asynchronous code also depends on external resources, such as file access, network servers, timers, or other modules, we might need to mock them and avoid unwanted failures due to any resource issues translated into the tests. Mocking and stubbing are techniques used to replace or modify the behavior of a real object or function with a fake or simplified one, for testing purposes. This way, we can control the input and output of the asynchronous code and avoid side effects or interference from other factors.

For example, if the tested code depends on a server, the server can fail to connect or execute its task, making the test fail. In these cases, failures are due to resource issues, not due to the asynchronous code being tested, causing a false, and usually transient, failure. We can mock external resources by using our own mock classes that mimic their interfaces. Let's see an example of how to use a mock class and use dependency injection to use that class for testing.

In this example, there is an external resource, `AsyncTaskScheduler`, whose `runTask()` method is used to execute an asynchronous task. As we only want to test the asynchronous task and remove any undesired side effects that the asynchronous task scheduler could generate, we can use a mock class mimicking the `AsyncScheduler` interface. This class is `MockTaskScheduler`, which

inherits from `AsyncTaskScheduler` and implements its `runTask()` base class method, where the task is run synchronously:

```
#include <gtest/gtest.h>
#include <functional>

class AsyncTaskScheduler {
   public:
    virtual int runTask(std::function<int()> task) = 0;
};

class MockTaskScheduler : public AsyncTaskScheduler {
   public:
    int runTask(std::function<int()> task) override {
        return task();
    }
};

TEST(AsyncTests, TestDependencyInjection) {
    MockTaskScheduler scheduler;

    auto task = []() -> int {
        return 42;
    };

    int result = scheduler.runTask(task);
    EXPECT_EQ(result, 42);
}

int main(int argc, char** argv) {
    ::testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

The `TestDependencyInjection` test just creates a `MockTaskScheduler` object and a task in the form of a lambda function and uses the mock object to execute the task by running the `runTask()` function. Once the task runs, `result` will have the value `42`.

Instead of fully defining the mock class, we can also use the gMock library and mock only the needed methods. This example shows gMock in action:

```
#include <gmock/gmock.h>
#include <gtest/gtest.h>
#include <functional>
```

```
class AsyncTaskScheduler {
   public:
    virtual int runTask(std::function<int()> task) = 0;
};

class MockTaskScheduler : public AsyncTaskScheduler {
   public:
    MOCK_METHOD(int, runTask, (std::function<int()> task),
(override));
};

TEST(AsyncTests, TestDependencyInjection) {
    using namespace testing;

    MockTaskScheduler scheduler;

    auto task = []() -> int {
        return 42;
    };

    EXPECT_CALL(scheduler, runTask(_)).WillOnce(
        Invoke(task)
    );

    auto result = scheduler.runTask(task);
    EXPECT_EQ(result, 42);
}

int main(int argc, char** argv) {
    ::testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

Now, `MockTaskScheduler` also inherits from `AsyncTaskScheduler`, where the interface is defined, but instead of overriding its methods, the `MOCK_METHOD` macro is used instead, where the return type, the mocked method name, and its parameters are passed.

Then, the `TestMockMethod` test uses the `EXPECT_CALL` macro to define an expected call to the `runTask()` mocked method in `MockTaskScheduler`, which will happen only once and invoke the lambda function task, which returns the value `42`.

That call just happens in the next instruction where `scheduler.runTask()` is called, storing the returned value in the result. The test finishes by checking if `result` is the expected value of `42`.

Testing exceptions and failures

Asynchronous tasks do not always succeed and generate a valid result. Sometimes something can go wrong (network failures, timeouts, exceptions, etc.), and returning an error or throwing an exception is the way to let the user know about this situation. We should simulate failures to ensure that the code handles these gracefully.

Testing errors or exceptions can be done in the usual way, by using a try-catch block and using assertions or expectations to check if an error is thrown and make the test succeed or fail. GoogleTest also provides the `EXPECT_ANY_THROW()` macro that simplifies checking if an exception has happened. Both approaches are shown in the following example:

```
#include <gtest/gtest.h>
#include <chrono>
#include <future>
#include <iostream>
#include <stdexcept>

using namespace std::chrono_literals;

int asyncFunc(bool should_fail) {
    std::this_thread::sleep_for(100ms);
    if (should_fail) {
        throw std::runtime_error("Simulated failure");
    }
    return 42;
}

TEST(AsyncTest, TestAsyncFailure1) {
    try {
        std::future<int> result = std::async(
                              std::launch::async,
                              asyncFunc, true);
        result.get();
        FAIL() << "No expected exception thrown";
    } catch (const std::exception& e) {
        SUCCEED();
    }
}

TEST(AsyncTest, TestAsyncFailure2) {
    std::future<int> result = std::async(
                            std::launch::async,
                            asyncFunc, true);
```

```
    EXPECT_ANY_THROW(result.get());
}

int main(int argc, char** argv) {
    ::testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

Both the `TestAsyncFailure1` and `TestAsyncFailure2` tests are very similar. Both execute asynchronously the `asyncFunc()` function, which now accepts a `should_fail` Boolean argument indicating whether the task should succeed and return the value `42` or fail and throw an exception. Both tests make the task fail, with the difference being that `TestAsyncFailure1` uses the `FAIL()` macro if no exception is thrown, making the test fail, or `SUCCEED()` if an exception is caught by the try-catch block, and `TestAsyncFailure2` uses the `EXPECT_ANY_THROW()` macro to check if an exception happens when trying to retrieve the result from the future result by calling its `get()` method.

Testing multiple threads

When testing asynchronous software involving multiple threads in C++, one common and effective technique is using condition variables to synchronize the threads. As we have seen in *Chapter 4*, condition variables allow threads to wait for certain conditions to be met before proceeding, making them essential for managing inter-thread communication and coordination.

Next is an example where multiple threads perform some tasks while the main thread waits for all other threads to finish.

Let's start by defining some necessary global variables, such as the total number of threads (`num_threads`), `counter` as an atomic variable that will increase each time the asynchronous task is invoked, and the condition variable, `cv`, and its associated mutex, `mtx`, which will help to unblock the main thread once all asynchronous tasks have been completed:

```
#include <gtest/gtest.h>
#include <atomic>
#include <chrono>
#include <condition_variable>
#include <iostream>
#include <mutex>
#include <syncstream>
#include <thread>
#include <vector>

using namespace std::chrono_literals;
```

```
#define sync_cout std::osyncstream(std::cout)

std::condition_variable cv;
std::mutex mtx;

bool ready = false;
std::atomic<unsigned> counter = 0;

const std::size_t num_threads = 5;
```

The `asyncTask()` function will execute the asynchronous task (simply waiting for 100 ms in this example) before increasing the `counter` atomic variable and notifying via the `cv` condition variable to the main thread that its work is done:

```
void asyncTask(int id) {
    sync_cout << "Thread " << id << ": Starting work..."
              << std::endl;
    std::this_thread::sleep_for(100ms);
    sync_cout << "Thread " << id << ": Work finished."
              << std::endl;

    ++counter;
    cv.notify_one();
}
```

The `TestMultipleThreads` test will start by spawning a number of threads where each one will asynchronously run the `asyncTask()` task. Then, it will wait, using a condition variable that `counter` has the same value as the number of threads, meaning that all background tasks have finished their work. The condition variable sets a timeout of 150 ms using the `wait_for()` function to limit the time the test can run but gives some room for all background tasks to be completed successfully:

```
TEST(AsyncTest, TestMultipleThreads) {
    std::vector<std::jthread> threads;

    for (int i = 0; i < num_threads; ++i) {
        threads.emplace_back(asyncTask, i + 1);
    }

    {
        std::unique_lock<std::mutex> lock(mtx);
        cv.wait_for(lock, 150ms, [] {
            return counter == num_threads;
        });
        sync_cout << "All threads have finished."
```

```
                    << std::endl;
    }

    EXPECT_EQ(counter, num_threads);
}
```

The test finishes by checking that indeed `counter` has the same value as `num_threads`.

Finally, the `main()` function is implemented:

```
int main(int argc, char** argv) {
    ::testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

As explained earlier, the program starts by initializing the GoogleTest library by calling `::testing::InitGoogleTest()` and then calling `RUN_ALL_TESTS()` to collect and run all tests.

Testing coroutines

With C++20, coroutines provide a new way to write and manage asynchronous code. Coroutine-based code can be tested by using a similar approach to other asynchronous code, but with the subtle difference that coroutines can suspend and be resumed.

Let's see an example with a simple coroutine.

We have seen in *Chapter 8* that coroutines have some boilerplate code to define their promise type and awaitable methods. Let's start by implementing the `Task` structure that will define the coroutine. Please revisit *Chapter 8* to fully understand this code.

Let's start by defining the `Task` structure:

```
#include <gtest/gtest.h>
#include <coroutine>
#include <exception>
#include <iostream>

struct Task {
    struct promise_type;
    using handle_type =
              std::coroutine_handle<promise_type>;

    handle_type handle_;

    Task(handle_type h) : handle_(h) {}
```

```
    ~Task() {
        if (handle_) handle_.destroy();
    }

    // struct promise_type definition
    // and await methods
};
```

Inside `Task`, we define `promise_type`, which describes how the coroutine is managed. This type provides certain predefined methods (hooks) that control how the values are returned, how the coroutine is suspended, and how resources are managed once the coroutine is completed:

```
struct Task {
    // ...
    struct promise_type {
        int result_;
        std::exception_ptr exception_;

        Task get_return_object() {
            return Task(handle_type::from_promise(*this));
        }

        std::suspend_always initial_suspend() {
            return {};
        }

        std::suspend_always final_suspend() noexcept {
            return {};
        }

        void return_value(int value) {
            result_ = value;
        }

        void unhandled_exception() {
            exception_ = std::current_exception();
        }
    };
    // ....
};
```

Then, the methods used for controlling the suspension and resumption of the coroutine are implemented:

```
struct Task {
    // ...
    bool await_ready() const noexcept {
        return handle_.done();
    }

    void await_suspend(std::coroutine_handle<>
                           awaiting_handle) {
        handle_.resume();
        awaiting_handle.resume();
    }

    int await_resume() {
        if (handle_.promise().exception_) {
            std::rethrow_exception(
                handle_.promise().exception_);
        }
        return handle_.promise().result_;
    }

    int result() {
        if (handle_.promise().exception_) {
            std::rethrow_exception(
                    handle_.promise().exception_);
        }
        return handle_.promise().result_;
    }
    // ....
};
```

Having the `Task` structure in place, let's define two coroutines, one that computes a valid value and another that throws an exception:

```
Task asyncFunc(int x) {
    co_return 2 * x;
}

Task asyncFuncWithException() {
    throw std::runtime_error("Exception from coroutine");
    co_return 0;
}
```

As test functions inside the `TEST()` macro in GoogleTest cannot directly be coroutines because they don't have a `promise_type` structure associated with them, we need to define some helper functions:

```
Task testCoroutineHelper(int value) {
    co_return co_await asyncFunc(value);
}

Task testCoroutineWithExceptionHelper() {
    co_return co_await asyncFuncWithException();
}
```

With that in place, we can now implement the tests:

```
TEST(AsyncTest, TestCoroutine) {
    auto task = testCoroutineHelper(5);
    task.handle_.resume();
    EXPECT_EQ(task.result(), 10);
}

TEST(AsyncTest, TestCoroutineWithException) {
    auto task = testCoroutineWithExceptionHelper();
    EXPECT_THROW({
            task.handle_.resume();
            task.result();
        },
        std::runtime_error);
}

int main(int argc, char **argv) {
    ::testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

The `TestCoroutine` test defines a task using the `testCoroutineHelper()` helper function and passing the value `5`. When resuming the coroutine, it's expected that it will return the value doubled, thus the value `10`, which is tested using `EXPECT_EQ()`.

The `TestCoroutineWithException` test uses a similar approach, but now using the `testCoroutineWithExceptionHelper()` helper function, which will throw an exception when the coroutine is resumed. This is exactly what happens inside the `EXPECT_THROW()` assertion macro before checking that indeed the exception is of type `std::runtime_error`.

Stress testing

A race condition detector can be achieved by performing stress testing. For highly concurrent or multi-threaded asynchronous code, stress testing is crucial. We can simulate high load with multiple asynchronous tasks to check if the system behaves correctly under stress. Also, it's important to use random delays, thread interleaving, or stress-testing tools, to reduce deterministic conditions, increasing the test coverage.

The next example shows the implementation of a stress test that spawns 100 (`total_nums`) threads that execute the asynchronous task where the atomic variable counter is increased with each run after a random wait:

```
#include <gtest/gtest.h>
#include <atomic>
#include <chrono>
#include <iostream>
#include <thread>
#include <vector>

std::atomic<int> counter(0);
const std::size_t total_runs = 100;

void asyncIncrement() {
    std::this_thread::sleep_for(std::chrono::milliseconds(rand() %
100));
    counter.fetch_add(1);
}

TEST(AsyncTest, StressTest) {
    std::vector<std::thread> threads;

    for (std::size_t i = 0; i < total_runs; ++i) {
        threads.emplace_back(asyncIncrement);
    }
    for (auto& thread : threads) {
        thread.join();
    }
    EXPECT_EQ(counter, total_runs);
}

int main(int argc, char** argv) {
    ::testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

The test succeeds if the counter has the same value as the total number of threads.

Parallelizing tests

To run test suites quicker we can parallelize the tests running in different threads, but tests must be independent, each running in a specific thread as a synchronous single-threaded solution. Also, they need to set up and tear down any necessary objects without keeping the state from previous test runs.

When using CMake together with GoogleTest, we can run all detected tests in parallel by specifying the number of concurrent jobs we want to use with the following command:

```
$ ctest -j <num_jobs>
```

All the examples shown in this section are a small subset of what can be done for testing asynchronous code. We hope that these techniques provide enough insight and knowledge to develop further testing techniques that deal with specific scenarios you might face.

Summary

In this chapter, we learned about how to sanitize and test asynchronous programs.

We started by learning how to sanitize code using sanitizers to help find multithreaded and asynchronous issues, such as race conditions, memory leaks, and use-after-scope errors, among many other issues.

Then, some testing techniques designed to deal with asynchronous software were described, using GoogleTest as the testing library.

Using these tools and techniques helps detect and prevent undefined behavior, memory errors, and security vulnerabilities while ensuring that concurrent operations execute correctly, timing issues are handled properly, and code performs as expected under various conditions. This improves the overall program's reliability and stability.

In the next chapter, we will learn about performance and optimization techniques that can be used to improve asynchronous programs' runtime and resource usage.

Further reading

- Sanitizers: `https://github.com/google/sanitizers`
- Clang 20.0 ASan: `https://clang.llvm.org/docs/AddressSanitizer.html`
- Clang 20.0 hardware-assisted ASan: `https://clang.llvm.org/docs/HardwareAssistedAddressSanitizerDesign.html`
- Clang 20.0 TSan: `https://clang.llvm.org/docs/ThreadSanitizer.html`

- Clang 20.0 MSan: https://clang.llvm.org/docs/MemorySanitizer.html
- Clang 20.0 UBSan: https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html
- Clang 20.0 DataFlowSanitizer: https://clang.llvm.org/docs/DataFlowSanitizer.html
- Clang 20.0 LSan: https://clang.llvm.org/docs/LeakSanitizer.html
- Clang 20.0 RealtimeSanitizer: https://clang.llvm.org/docs/RealtimeSanitizer.html
- Clang 20.0 SanitizerCoverage: https://clang.llvm.org/docs/SanitizerCoverage.html
- Clang 20.0 SanitizerStats: https://clang.llvm.org/docs/SanitizerStats.html
- GCC: *Program Instrumentation Options*: https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html
- Apple Developer: *Diagnosing memory, thread, and crash issues early*: https://developer.apple.com/documentation/xcode/diagnosing-memory-thread-and-crash-issues-early
- GCC: *Options for Debugging Your Program*: https://gcc.gnu.org/onlinedocs/gcc/Debugging-Options.html
- OpenSSL: *Compiler Options Hardening Guide for C and C++*: https://best.openssf.org/Compiler-Hardening-Guides/Compiler-Options-Hardening-Guide-for-C-and-C++.html
- Memory error checking in C and C++: Comparing Sanitizers and Valgrind: https://developers.redhat.com/blog/2021/05/05/memory-error-checking-in-c-and-c-comparing-sanitizers-and-valgrind
- The GNU C Library: https://www.gnu.org/software/libc
- Sanitizers: Common flags: https://github.com/google/sanitizers/wiki/SanitizerCommonFlags
- AddressSanitizer flags: https://github.com/google/sanitizers/wiki/AddressSanitizerFlags
- AddressSanitizer: A Fast Address Sanity Checker: https://www.usenix.org/system/files/conference/atc12/atc12-final39.pdf
- MemorySanitizer: Fast detector of uninitialized memory use in C++: https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/43308.pdf

- Linux Kernel Sanitizers: https://github.com/google/kernel-sanitizers
- TSan flags: https://github.com/google/sanitizers/wiki/ThreadSanitizerFlags
- TSan: Popular data races: https://github.com/google/sanitizers/wiki/ThreadSanitizerPopularDataRaces
- TSan report format: https://github.com/google/sanitizers/wiki/ThreadSanitizerReportFormat
- TSan algorithm: https://github.com/google/sanitizers/wiki/ThreadSanitizerAlgorithm
- Address space layout randomization: https://en.wikipedia.org/wiki/Address_space_layout_randomization
- GoogleTest User's Guide: https://google.github.io/googletest

13

Improving Asynchronous Software Performance

In this chapter, we'll introduce the performance aspects of asynchronous code. Code performance and optimization is a deep and complex subject, and we can't cover everything in just one chapter. We aim to give you a good introduction to the subject with some examples of how to measure performance and optimize your code.

This chapter will cover the following key topics:

- Performance measurement tools with a focus on multithreaded applications
- What's false sharing, how to spot it, and how to fix/improve our code
- An introduction to modern CPUs' memory cache architecture
- A review of the **single-producer-single-consumer** (**SPSC**) lock-free queue we implemented in *Chapter 5*

Technical requirements

Like in the previous chapters, you'll need a modern C++ compiler that supports C++20. We'll be using GCC 13 and Clang 18. You'll also need a PC with an Intel/AMD multicore CPU running Linux. For this chapter, we used Ubuntu 24.04 LTS running on a workstation with a CPU AMD Ryzen Threadripper Pro 5975WX (32 cores). A CPU with 8 cores is ideal but 4 cores is enough to run the examples.

We'll also be using the Linux `perf` tool. We'll explain how to get and install these tools later in this book.

The examples for this chapter can be found in this book's GitHub repository: `https://github.com/PacktPublishing/Asynchronous-Programming-with-CPP`.

Performance measurement tools

To learn about the performance of our applications, we need to be able to measure it. If there's one key takeaway from this chapter, it's to *never estimate or guess your code performance*. To know whether your program meets its performance requirements (either latency or throughput), you need to measure, measure, and then measure again.

Once you have the data from your performance tests, you'll know the hotspots in your code. Maybe they're related to memory access patterns or thread contention (such as, for example, when multiple threads must wait to acquire a lock to access a resource). This is where the second most important takeaway comes into play: *set a goal when optimizing your application*. Don't aim to achieve the best performance possible because there always will be room for improvement. The right thing to do is to set a clear specification with targets such as maximum processing time for a transaction or the number of network packets processed per second.

With these two main ideas in mind, let's start with the different methods we can use to measure code performance.

In-code profiling

A very simple but useful way to start understanding the performance of our code is **in-code profiling**, which consists of adding some extra code to measure the execution time of some code sections. This method is good to use as a tool while we're writing the code (of course, we need to have access to the source code). This will allow us to find some performance issues in our code, as we'll see later in this chapter.

We're going to use `std::chrono` as our initial approach to profiling our code.

The following code snippet shows how we can use `std::chrono` to do some basic profiling of our code:

```
auto start = std::chrono::high_resolution_clock::now();
// processing to profile
auto end = std::chrono::high_resolution_clock::now();

auto duration = std::chrono::duration_
cast<std::chrono::milliseconds>(end - start);
std::cout < duration.count() << " milliseconds\n";
```

Here, we get two time samples that call `high_resolution_clock::now()` and print the time lapse converted into milliseconds. Depending on the time we estimate the processing is going to take, we could use either microseconds or seconds, for example. With this simple technique, we can easily get an idea of how long the processing takes and we can easily compare different options.

Here, `std::chrono::high_resolution_clock` is the clock type that offers the highest precision (smallest tick period provided by the implementation). The C++ Standard Library allows it to be an alias of either `std::chrono::system_clock` or `std::chrono::steady_clock`. libstdc++ has it aliased to `std::chrono::system_clock`, whereas libc++ uses `std::chrono::steady_clock`. For the examples in this chapter, we've used GCC and libstdc++. The clock resolution is 1 nanosecond:

```
/**
 *  @brief Highest-resolution clock
 *
 *  This is the clock "with the shortest tick period." Alias to
 *  std::system_clock until higher-than-nanosecond definitions
 *  become feasible.
 *  @ingroup chrono
*/
using high_resolution_clock = system_clock;
```

Now, let's see a full example of profiling two of the C++ Standard Library algorithms to sort vectors – `std::sort` and `std::stable_sort`:

```
#include <algorithm>
#include <chrono>
#include <iostream>
#include <random>
#include <utility>

int uniform_random_number(int min, int max) {
    static std::random_device rd;
    static std::mt19937 gen(rd());
    std::uniform_int_distribution dis(min, max);
    return dis(gen);
}

std::vector<int> random_vector(std::size_t n, int32_t min_val, int32_t
max_val) {
    std::vector<int> rv(n);
    std::ranges::generate(rv, [&] {
            return uniform_random_number(min_val, max_val);
        });
    return rv;
}

using namespace std::chrono;
```

```
int main() {
    constexpr uint32_t elements = 100000000;
    int32_t minval = 1;
    int32_t maxval = 1000000000;

    auto rv1 = random_vector(elements, minval, maxval);
    auto rv2 = rv1;

    auto start = high_resolution_clock::now();
    std::ranges::sort(rv1);
    auto end = high_resolution_clock::now();
    auto duration = duration_cast<milliseconds>(end - start);
    std::cout << "Time to std::sort "
              << elements << " elements with values in ["
              << minval << "," << maxval << "] "
              << duration.count() << " milliseconds\n";

    start = high_resolution_clock::now();
    std::ranges::stable_sort(rv2);
    end = high_resolution_clock::now();
    duration = duration_cast<milliseconds>(end - start);
    std::cout << "Time to std::stable_sort "
              << elements << " elements with values in ["
              << minval << "," << maxval << "] "
              << duration.count() << " milliseconds\n";

    return 0;
}
```

The preceding code generates a vector of normally distributed random numbers and then sorts the vector with both `std::sort()` and `std::stable_sort()`. Both functions sort the vector, but `std::sort()` uses a combination of quicksort and insertion sort algorithms called introsort, while `std::stable_sort()` uses merge sort. The sort is *stable* because equivalent keys have the same order in both the original and sorted vectors. For a vector of integers, this isn't important, but if the vector has three elements with the same value, after sorting the vector, the numbers will be in the same order.

After running the code, we get the following output:

```
Time to std::sort 100000000 elements with values in [1,1000000000]
6019 milliseconds
Time to std::stable_sort 100000000 elements with values in
[1,1000000000] 7342 milliseconds
```

In this example, `std::stable_sort()` is slower than `std::sort()`.

In this section, we learned about a simple way to measure the running time of sections of our code. This method is intrusive and requires that we modify the code; it's mostly used while we develop our applications. In the next section, we're going to introduce another way to measure execution time called micro-benchmarks.

Code micro-benchmarks

Sometimes, we just want to analyze a small section of code in isolation. We may need to run it more than once and then get the average running time or run it with different input data. In these cases, we can use a benchmark (also called a **micro-benchmark**) library to do just that – execute small parts of our code in different conditions.

Micro-benchmarks must be used as a guide. Bear in mind that the code runs in isolation, and this can give us very different results when we run all the code together due to the many complex interactions among different sections of our code. Use them carefully and be aware that micro-benchmarks can be misleading.

There are many libraries we can use to benchmark our code. We'll use *Google Benchmark*, a very good and well-known library.

Let's start by getting the code and compiling the library. To get the code, run the following commands:

```
git clone https://github.com/google/benchmark.git
cd benchmark
git clone https://github.com/google/googletest.git
```

Once we have the code for both the benchmark and Google Test libraries (the latter is required to compile the former), we'll build it.

Create a directory for the build:

```
mkdir build
cd build
```

With that, we've created the build directory inside the benchmark directory.

Next, we'll use CMake to configure the build and create all the necessary information for `make`:

```
cmake .. -DCMAKE_BUILD_TYPE=Release -DBUILD_SHARED_LIBRARIES=ON
-DCMAKE_INSTALL_PREFIX=/usr/lib/x86_64-linux-gnu/
```

Finally, run `make` to build and install the libraries:

```
make -j16
sudo make install
```

You also need to add the library to the `CmakeLists.txt` file. We've done that for you in the code for this book.

Once Google Benchmark has been installed, we can work on an example with a few benchmark functions to learn how to use the library for some basic benchmarking.

Note that both `std::chrono` and Google Benchmark aren't specific tools for working with asynchronous/multithreaded code and are more like generic tools.

This is our first example of using Google Benchmark:

```
#include <benchmark/benchmark.h>

#include <algorithm>
#include <chrono>
#include <iostream>
#include <random>
#include <thread>

void BM_vector_push_back(benchmark::State& state) {
    for (auto _ : state) {
        std::vector<int> vec;
        for (int i = 0; i < state.range(0); i++) {
            vec.push_back(i);
        }
    }
}

void BM_vector_emplace_back(benchmark::State& state) {
    for (auto _ : state) {
        std::vector<int> vec;
        for (int i = 0; i < state.range(0); i++) {
            vec.emplace_back(i);
        }
    }
}

void BM_vector_insert(benchmark::State& state) {
    for (auto _ : state) {
        std::vector<int> vec;
        for (int i = 0; i < state.range(0); i++) {
            vec.insert(vec.begin(), i);
        }
    }
}
```

```
BENCHMARK(BM_vector_push_back)->Range(1, 1000);
BENCHMARK(BM_vector_emplace_back)->Range(1, 1000);
BENCHMARK(BM_vector_insert)->Range(1, 1000);

int main(int argc, char** argv) {
    benchmark::Initialize(&argc, argv);
    benchmark::RunSpecifiedBenchmarks();

    return 0;
}
```

We need to include the library header:

```
#include <benchmark/benchmark.h>
```

All benchmark functions have the following signature:

```
void benchmark_function(benchmark::State& state);
```

This is a function with one parameter, `benchmark::State& state`, that returns `void`. The `benchmark::State` parameter has a dual purpose:

- **Controlling the iteration loop**: The `benchmark::State` object is used to control how many times a benchmarked function or piece of code should be executed. This helps measure the performance accurately by repeating the test enough times to minimize variability and collect meaningful data.
- **Measuring time and statistics**: The `state` object keeps track of how long the benchmarked code takes to run, and it provides mechanisms to report metrics such as elapsed time, iterations, and custom counters.

We've implemented three functions to benchmark adding elements to a `std::vector` sequence in different ways: the first function uses `std::vector::push_back`, the second uses `std::vector::emplace_back`, and the third uses `std::vector::insert`. The first two functions add elements at the end of the vector, while the third function adds elements at the beginning of the vector.

Once we've implemented the benchmark functions, we need to tell the library that they must be run as a benchmark:

```
BENCHMARK(BM_vector_push_back)->Range(1, 1000);
```

We use the `BENCHMARK` macro to do this. For the benchmarks in this example, we set the number of elements to be inserted into the vector in each iteration. The range goes from `1` to `1000` and each iteration will insert eight times the number of elements of the previous iteration until it reaches the maximum. In this case, it will insert 1, 8, 64, 512, and 1,000 elements.

When we run our first benchmark program, we get the following output:

```
2024-10-17T05:02:37+01:00
Running ./13x02-benchmark_vector
Run on (64 X 3600 MHz CPU s)
CPU Caches:
  L1 Data 32 KiB (x32)
  L1 Instruction 32 KiB (x32)
  L2 Unified 512 KiB (x32)
  L3 Unified 32768 KiB (x4)
Load Average: 0.00, 0.02, 0.16
-----------------------------------------------------------------------
Benchmark                             Time             CPU   Iterations
-----------------------------------------------------------------------
BM_vector_push_back/1              10.5 ns         10.5 ns     63107997
BM_vector_push_back/8              52.0 ns         52.0 ns     13450361
BM_vector_push_back/64              116 ns          116 ns      6021740
BM_vector_push_back/512             385 ns          385 ns      1819732
BM_vector_push_back/1000            641 ns          641 ns      1093474
BM_vector_emplace_back/1           10.8 ns         10.8 ns     64570848
BM_vector_emplace_back/8           53.3 ns         53.3 ns     13139191
BM_vector_emplace_back/64           108 ns          108 ns      6469997
BM_vector_emplace_back/512          364 ns          364 ns      1924992
BM_vector_emplace_back/1000         616 ns          616 ns      1138392
BM_vector_insert/1                 10.6 ns         10.6 ns     65966159
BM_vector_insert/8                 58.6 ns         58.6 ns     11933446
BM_vector_insert/64                 461 ns          461 ns      1485319
BM_vector_insert/512               7249 ns         7249 ns        96756
BM_vector_insert/1000             23352 ns        23348 ns        29742
```

First, the program prints information about the execution of the benchmark: the date and time, the name of the executable, and information about the CPU it's running on.

Take a look at the following line:

```
Load Average: 0.00, 0.02, 0.16
```

This line gives us an estimate of the CPU load: from 0.0 (no load at all or very low load) to 1.0 (fully loaded). The three numbers correspond to the CPU load for the last 5, 10, and 15 minutes, respectively.

After printing the CPU load information, the benchmark prints the results of each iteration. Here's an example:

```
BM_vector_push_back/64              116 ns          116 ns      6021740
```

This means that `BM_vector_push_back` was called 6,021,740 times (the number of iterations) while inserting 64 elements into the vector.

The `Time` and `CPU` columns give us the average time for each iteration:

- **Time**: This is the real time that's elapsed from the beginning to the end of each benchmark execution. It includes everything that happens during the benchmark: CPU computation, I/O operations, context switches, and more.
- **CPU time**: This is the amount of time the CPU spent processing the instructions of the benchmark. It can be smaller than or equal to `Time`.

In our benchmark, because the operations are simple, we can see that `Time` and `CPU` are mostly the same.

Looking at the results, we can come to the following conclusions:

- For simple objects such as 32-bit integers, both `push_back` and `emplace_back` take the same amount of time.
- Here, `insert` takes the same amount of time as `push_back`/`emplace_back` for a small number of elements but from 64 elements onwards, it takes considerably more time. This is because `insert` must copy all the elements after each insertion (we insert the elements at the beginning of the vector).

The following example also sorts a `std::vector` sequence, but this time, we'll use a micro-benchmark to measure execution time:

```
#include <benchmark/benchmark.h>

#include <algorithm>
#include <chrono>
#include <iostream>
#include <random>
#include <thread>

std::vector<int> rv1, rv2;

int uniform_random_number(int min, int max) {
    static std::random_device rd;
    static std::mt19937 gen(rd());
    std::uniform_int_distribution dis(min, max);
    return dis(gen);
}

std::vector<int> random_vector(std::size_t n, int32_t min_val, int32_t
```

```
max_val) {
    std::vector<int> rv(n);
    std::ranges::generate(rv, [&] {
        return uniform_random_number(min_val, max_val);
    });
    return rv;
}

static void BM_vector_sort(benchmark::State& state, std::vector<int>&
vec) {
    for (auto _ : state) {
        std::ranges::sort(vec);
    }
}

static void BM_vector_stable_sort(benchmark::State& state,
std::vector<int>& vec) {
    for (auto _ : state) {
        std::ranges::stable_sort(vec);
    }
}

BENCHMARK_CAPTURE(BM_vector_sort, vector, rv1)->Iterations(1)-
>Unit(benchmark::kMillisecond);
BENCHMARK_CAPTURE(BM_vector_stable_sort, vector, rv2)->Iterations(1)-
>Unit(benchmark::kMillisecond);

int main(int argc, char** argv) {
    constexpr uint32_t elements = 100000000;
    int32_t minval = 1;
    int32_t maxval = 1000000000;

    rv1 = random_vector(elements, minval, maxval);
    rv2 = rv1;
    benchmark::Initialize(&argc, argv);
    benchmark::RunSpecifiedBenchmarks();

    return 0;
}
```

The preceding code generates a vector of random numbers. Here, we run two benchmark functions to sort the vector: one using `std::sort` and another using `std::stable_sort`. Note that we use two copies of the same vector, so the input is the same for both functions.

The following line of code uses the `BENCHMARK_CAPTURE` macro. This macro allows us to pass parameters to our benchmark functions – in this case, a reference to `std::vector` (we pass by reference to avoid copying the vector and impacting the benchmark result).

We specify the results to be in milliseconds instead of nanoseconds:

```
BENCHMARK_CAPTURE(BM_vector_sort, vector, rv1)->Iterations(1)-
>Unit(benchmark::kMillisecond);
```

Here are the results of the benchmark:

```
-------------------------------------------------------------------------
----
Benchmark                             Time         CPU   Iterations
-------------------------------------------------------------------------
----
BM_vector_sort                        5877 ms      5876 ms            1
BM_vector_stable_sort.                7172 ms      7171 ms            1
```

The results are consistent with the ones we got measuring time using `std::chrono`.

For our last Google Benchmark example, we'll create a thread (`std::thread`):

```
#include <benchmark/benchmark.h>

#include <algorithm>
#include <chrono>
#include <iostream>
#include <random>
#include <thread>

static void BM_create_terminate_thread(benchmark::State& state) {
    for (auto _ : state) {
        std::thread thread([]{ return -1; });
        thread.join();
    }
}

BENCHMARK(BM_create_terminate_thread)->Iterations(2000);

int main(int argc, char** argv) {
    benchmark::Initialize(&argc, argv);
    benchmark::RunSpecifiedBenchmarks();

    return 0;
}
```

This example is simple: `BM_create_terminate_thread` creates a thread (doing nothing, just returning 0) and waits for it to end (`thread.join()`). We run `2000` iterations to get an estimation of the time it takes to create a thread.

The results are as follows:

```
-------------------------------------------------------------
----------
Benchmark                       Time             CPU
Iterations
-------------------------------------------------------------
----------
BM_create_terminate_thread.       32424 ns        21216 ns     2000
```

In this section, we learned how to use the Google Benchmark library to create micro-benchmarks to measure the execution time of some functions. Again, micro-benchmarks are just an approximation and due to the isolated nature of the code being benchmarked, they may be misleading. Use them carefully.

The Linux perf tool

Using `std::chrono` in our code or a micro-benchmark library such as Google Benchmark requires gaining access to the code to be profiled and also being able to modify it by either adding extra calls to measure the execution time of code sections or running small snippets as micro-benchmark functions.

With the Linux `perf` tool, we can analyze the execution of a program without changing any of its code.

The Linux `perf` tool is a powerful, flexible, and widely used performance analysis and profiling utility for Linux systems. It provides detailed insights into system performance at the kernel and user space levels.

Let's consider the main uses of `perf`.

First, we have **CPU profiling**. The `perf` tool allows you to capture the execution profile of a process, measuring which functions consume most of the CPU time. This can be very useful in helping to identify CPU-intensive parts of the code and bottlenecks.

The following command line will run `perf` on the small `13x07-thread_contention` program we wrote to illustrate the basics of the tool. The code for this application can be found in this book's GitHub repository:

```
perf record --call-graph dwarf ./13x07-thread_contention
```

The `--call-graph` option records the data of the function call hierarchy in a file called `perf.data`, while the `dwarf` option instructs `perf` to use the dwarf file format to debug symbols (to get the function names).

After the previous command, we must run the following:

```
perf script > out.perf
```

This will dump the recorded data (including the call stack) into a text file called `out.perf`.

Now, we need to convert the text file into a picture with the call graph. To do this, we can run the following command:

```
gprof2dot -f perf out.perf -o callgraph.dot
```

This will generate a file called `callgraph.dot` that can be visualized using Graphviz.

You may need to install `gprof2dot`. For this, you need Python installed on your PC. Run the following command to install `gprof2dot`:

```
pip install gprof2dot
```

Install Graphviz too. In Ubuntu, you can do this like so:

```
sudo apt-get install graphviz
```

Finally, you can generate the `callgraph.png` picture by running the following command:

```
dot -Tpng callgraph.dot -o callgraph.png
```

Another very common way to visualize the call graph of a program is by using a flame graph.

To generate a flame graph, clone the `FlameGraph` repository:

```
git clone https://github.com/brendangregg/FlameGraph.git
```

In the `FlameGraph` folder, you'll find the scripts to generate the flame graphs.

Run the following command:

```
FlameGraph/stackcollapse-perf.pl out.perf > out.folded
```

This command will collapse the stack traces into a format that can be used by the FlameGraph tool. Now, run the following command:

```
Flamegraph/flamegraph.pl out.folded > flamegraph.svg
```

You can visualize the flame graph with a web browser:

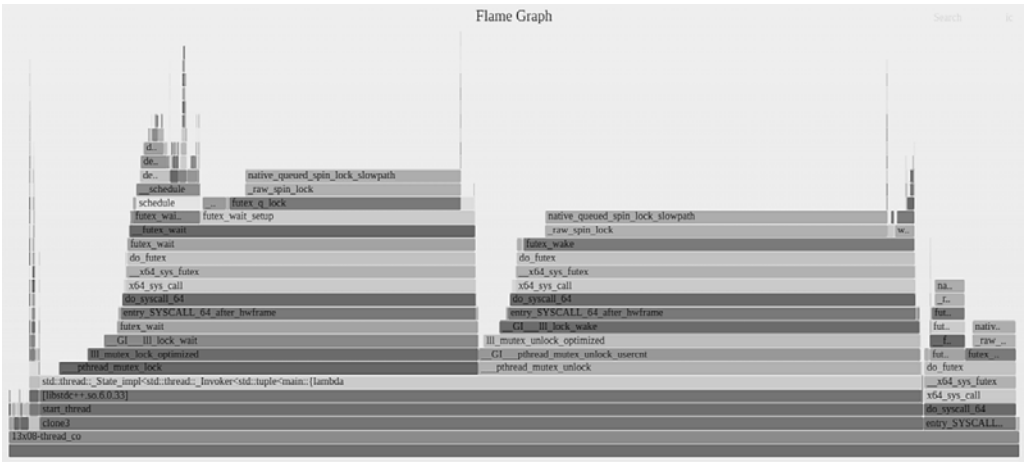


Figure 13.1: An overview of a flame graph

Now, let's learn how to gather the performance statistics of a program.

The following command will show the number of instructions that have been executed and CPU cycles that were used during the execution of `13x05-sort_perf`. The number of instructions per cycle is the average number of instructions the CPU executes in each clock cycle. This metric is only useful when we microbenchmark or measure short parts of the code. For this example, we can see that the CPU is executing one instruction per cycle, which is average for a modern CPU. In multithreaded code, we can get a much bigger number due to the parallel nature of the execution, but this metric is generally used to measure and optimize code executed in a single CPU core. The number must be interpreted as how busy we keep the CPU because it depends on many factors, such as the number of memory reads/writes, memory access patterns (linear consecutive/no linear), level of branching in the code, and so on:

```
perf stat -e instructions,cycles ./13x05-sort_perf
```

After running the preceding command, we get the following result:

```
Performance counter stats for './13x05-sort_perf':

    30,993,024,309      instructions                     #     1.03
             insn per cycle
    30,197,863,655      cycles

       6.657835162 seconds time elapsed

       6.502372000 seconds user
       0.155008000 seconds sys
```

Running the following command, you can get the list of all the predefined events you can analyze with `perf`:

```
perf list
```

Let's do a few more:

```
perf stat -e branches ./13x05-sort_perf
```

The previous command measures the number of branch instructions that have been executed. We get the following result:

```
Performance counter stats for './13x05-sort_perf':

     5,246,138,882      branches

       6.712285274 seconds time elapsed

       6.551799000 seconds user
       0.159970000 seconds sys
```

Here, we can see that a sixth of the instructions that were executed are branching instructions, which is expected in a program that sorts large vectors.

As mentioned previously, measuring the level of branching in our code is important, especially for short sections of the code (to avoid interactions that can impact what's being measured). A CPU will run instructions much faster if there are no branches or there are just a few. The main issue with branches is that the CPU may need to rebuild the pipeline and that can be costly, especially if branches are in inner/critical loops.

The following command will report the number of L1 cache data accesses (we will see the CPU cache in the next section):

```
perf stat -e all_data_cache_accesses ./13x05-sort_perf
```

We get the following result:

```
Performance counter stats for './13x05-sort_perf':

    21,286,061,764      all_data_cache_accesses

       6.718844368 seconds time elapsed

       6.561416000 seconds user
       0.157009000 seconds sys
```

Let's go back to our lock contention example and gather some useful statistics with `perf`.

Another benefit of using `perf` is **CPU migrations** – that is, the number of times a thread was moved from one CPU core to another. Thread migration between cores can degrade cache performance since threads lose the benefit of cached data when moving to a new core (more on caches in the next section).

Let's run the following command:

```
perf stat -e cpu-migrations ./13x07-thread_contention
```

This results in the following output:

```
Performance counter stats for './13x08-thread_contention':

                45      cpu-migrations

      50.476706194 seconds time elapsed

      57.333880000 seconds user
     262.123060000 seconds sys
```

Let's look at another advantage of using `perf`: **context switches**. It counts the number of context switches (how many times a thread is swapped out and another thread is scheduled) during the execution. High-context switching can indicate that too many threads are competing for CPU time, leading to performance degradation.

Let's run the following command:

```
perf stat -e context-switches ./13x07-thread_contention
```

This results in the following output:

```
Performance counter stats for './13x08-thread_contention':

        13,867,866      cs

      47.618283562 seconds time elapsed

      52.931213000 seconds user
     247.033479000 seconds sys
```

That's a wrap on this section. Here, we introduced the Linux `perf` tool and some of its applications. We'll study the CPU memory cache and false sharing in the next section.

False sharing

In this section, we'll study a common issue with multithreaded applications called **false sharing**.

We already know that the ideal implementation of a multithreaded application is minimizing the data that's shared among its different threads. Ideally, we should share data just for read access because in that case, we don't need to synchronize the threads to access the shared data and thus we don't need to pay the runtime cost and deal with issues such as deadlock and livelock.

Now, let's consider a simple example: four threads run in parallel, generate random numbers, and calculate their sum. Each thread works independently, generating random numbers and calculating the sum stored in a variable just written by itself. This is the ideal (though for this example, a bit contrived) application, with threads working independently without any shared data.

The following code is the full source for the example we're going to analyze in this section. You can refer to it while you read the explanations:

```
#include <chrono>
#include <iostream>
#include <random>
#include <thread>
#include <vector>

struct result_data {
    unsigned long result { 0 };
};

struct alignas(64) aligned_result_data {
    unsigned long result { 0 };
};

void set_affinity(int core) {
    if (core < 0) {
        return;
    }

    cpu_set_t cpuset;
    CPU_ZERO(&cpuset);
    CPU_SET(core, &cpuset);
    if (pthread_setaffinity_np(pthread_self(), sizeof(cpu_set_t),
&cpuset) != 0) {
        perror("pthread_setaffinity_np");
        exit(EXIT_FAILURE);
    }
}
```

```
template <typename T>
auto random_sum(T& data, const std::size_t seed, const unsigned long
iterations, const int core) {
    set_affinity(core);
    std::mt19937 gen(seed);
    std::uniform_int_distribution dist(1, 5);
    for (unsigned long i = 0; i < iterations; ++i) {
        data.result += dist(gen);
    }
}

using namespace std::chrono;

void sum_random_unaligned(int num_threads, uint32_t iterations) {
    auto* data = new(static_cast<std::align_val_t>(64)) result_
data[num_threads];

    auto start = high_resolution_clock::now();
    std::vector<std::thread> threads;
    for (std::size_t i = 0; i < num_threads; ++i) {
        set_affinity(i);
        threads.emplace_back(random_sum<result_data>,
std::ref(data[i]), i, iterations, i);
    }
    for (auto& thread : threads) {
        thread.join();
    }
    auto end = high_resolution_clock::now();
    auto duration = std::chrono::duration_cast<milliseconds>(end -
start);
    std::cout << "Non-aligned data: " << duration.count() << "
milliseconds" << std::endl;

    operator delete[] (data, static_cast<std::align_val_t>(64));
}

void sum_random_aligned(int num_threads, uint32_t iterations) {
    auto* aligned_data = new(static_cast<std::align_val_t>(64))
aligned_result_data[num_threads];
    auto start = high_resolution_clock::now();
    std::vector<std::thread> threads;
    for (std::size_t i = 0; i < num_threads; ++i) {
        set_affinity(i);
        threads.emplace_back(random_sum<aligned_result_data>,
```

```
std::ref(aligned_data[i]), i, iterations, i);
    }
    for (auto& thread : threads) {
        thread.join();
    }
    auto end = high_resolution_clock::now();
    auto duration = std::chrono::duration_cast<milliseconds>(end -
start);
    std::cout << "Aligned data: " << duration.count() << "
milliseconds" << std::endl;
    operator delete[] (aligned_data, static_cast<std::align_val_
t>(64));
}

int main() {
    constexpr unsigned long iterations{ 100000000 };
    constexpr unsigned int num_threads = 8;

    sum_random_unaligned(8, iterations);
    sum_random_aligned(8, iterations);

    return 0;
}
```

If you compile and run the previous code, you'll get an output similar to the following:

```
Non-aligned data: 4403 milliseconds
Aligned data: 160 milliseconds
```

The program just calls two functions: `sum_random_unaligned` and `sum_random_aligned`. Both functions do the same: they create eight threads, and each thread generates random numbers and calculates their sum. No data is shared among the threads. You can see that the functions are pretty much the same and the main difference is that `sum_random_unaligned` uses the following data structure to store the sum of the generated random numbers:

```
struct result_data {
    unsigned long result { 0 };
};
```

The `sum_random_aligned` function uses a slightly different one:

```
struct alignas(64) aligned_result_data {
    unsigned long result { 0 };
};
```

The only difference is the use of `alignas(64)` in informing the compiler that the data structure instances must be aligned at a 64-byte boundary.

We can see that the difference in performance is quite dramatic because the threads are performing the same tasks. Just aligning the variables written by each thread to a 64-byte boundary greatly improves performance.

To understand why this is happening, we need to consider a feature of modern CPUs – the memory cache.

CPU memory cache

Modern CPUs are very fast at computing and when we want to achieve maximum performance, memory access is the main bottleneck. A good estimate for memory access is about 150 nanoseconds. In that time, our 3.6 GHz CPU has gone through 540 clock cycles. As a rough estimate, if the CPU executes an instruction every two cycles, that's 270 instructions. For a normal application, memory access is an issue, even though the compiler may reorder the instructions it generates and the CPU may also reorder the instructions to optimize memory access and try to run as many instructions as possible.

Therefore, to improve the performance of modern CPUs, we have what's called a **CPU cache** or **memory cache**, which is memory in the chip to store both data and instructions. This memory is much faster than RAM and allows the CPU to retrieve data much faster, significantly boosting overall performance.

As an example of a real-life cache, think about a cook. They need some ingredients to make lunch for their restaurant clients. Now, imagine that they only buy those ingredients when a client comes to the restaurant and orders their food. That will be very slow. They can also go to the supermarket and buy ingredients for, say, a full day. Now, they can cook for all their clients and serve them their meals in a much shorter period.

CPU caches follow the same concept: when the CPU needs to access a variable, say a 4-byte integer, it reads 64 bytes (this size may be different, depending on the CPU, but most modern CPUs use that size) of contiguous memory *just in case* it may need to access more contiguous data.

Linear memory data structures such as `std::vector` will perform better from a memory access point of view because in these cases, the cache can be a big performance improvement. For other types of data structures, such as `std::list`, this won't be the case. Of course, this is just about optimizing cache use.

You may be wondering why if in-CPU cache memory is so good, isn't all memory like that? The answer is cost. Cache memory is very fast (much faster than RAM), but it's also very expensive.

Modern CPUs employ a hierarchical cache structure, typically consisting of three levels called L1, L2, and L3:

- **L1 cache** is the smallest and fastest. It's also the closest to the CPU, as well as the most expensive. It's often split into two parts: an instruction cache for storing instructions and a data cache for storing data. The typical sizes are 64 Kb split into 32 Kb for instructions and 32 Kb for data. The typical access time to the L1 cache is between 1 and 3 nanoseconds.

- **L2 cache** is larger and slightly slower than L1, but still much faster than RAM. Typical L2 cache sizes are between 128 Kb and 512 Kb (the CPU used to run the examples in this chapter has 512 Kb of L2 cache per core). Typical access times for L2 cache are about 3 to 5 nanoseconds.
- **L3 cache** is the largest and slowest of the three. The L1 and L2 caches are per core (each core has its own L1 and L2 cache), but L3 is shared by more than one core. Our CPU has 32 Mb of L3 cache shared by each group of eight cores. The typical access time is about 10 to 15 nanoseconds.

With that, let's turn our attention to another important concept related to memory cache.

Cache coherency

The CPU doesn't access RAM directly. This access is always done through the cache, and RAM is accessed only if the CPU doesn't find the data required in the cache. In multi-core systems, each core having its own cache means that one piece of RAM may be present in the cache of multiple cores at the same time. These copies need to be synchronized all the time; otherwise, computation results could be incorrect.

So far, we've seen that each core has its own L1 cache. Let's go back to our example and think about what happens when we run the function using non-aligned memory.

In this case, each instance of `result_data` is 8 bytes. We create an array of 8 instances of `result_data`, one for each thread. The total memory that's occupied will be 64 bytes and all the instances will be contiguous in memory. Every time a thread updates the sum of random numbers, it changes the value that's stored in the cache. Remember that the CPU will always read and write 64 bytes in one go (something called a **cache line** – you can think of it as the smallest memory access unit). All the variables are in the same cache line and even if the threads don't share them (each thread has its own variable – `sum`), the CPU doesn't know that and needs to make the changes visible for all the cores.

Here, we have 8 cores, and each core is running a thread. Each core has loaded 64 bytes of memory from RAM into the L1 cache. Since the threads only read the variables, everything is OK, but as soon as one thread modifies its variable, the contents of the cache line are invalidated.

Now, because the cache line is invalid in the remaining 7 cores, the CPU needs to propagate the changes to all the cores. As mentioned previously, even if the threads don't share the variables, the CPU can't possibly know that, and it updates all the cache lines for all the cores to keep the values consistent. This is called cache coherency. If the threads shared the variables, it would be incorrect not to propagate the changes to all the cores.

In our example, the cache coherency protocol generates quite a lot of traffic inside the CPU because all the threads *share the memory region where the variables reside*, even though they don't from the program's point of view. This is the reason we call it false sharing: the variables are shared because of the way the cache and the cache coherency protocol work.

When we align the data to a 64-byte boundary, each instance occupies 64 bytes. This guarantees that they are in their own cache line and no cache coherency traffic is necessary because in this case, there's no data sharing. In this second case, performance is much better.

Let's use `perf` to confirm that this is really happening.

First, we run `perf` while executing `sum_random_unaligned`. We want to see how many times the program accesses the cache and how many times there's a cache miss. Each time the cache needs to be updated because it contains data that's also in a cache line in another core counts as a cache miss:

```
perf stat -e cache-references,cache-misses ./13x07-false_sharing
```

We obtain these results:

```
Performance counter stats for './13x07-false_sharing':

       251,277,877      cache-references
       242,797,999      cache-misses
                        # 96.63% of all cache refs
```

Most of the cache references are cache misses. This is expected because of false sharing.

Now, if we run `sum_random_aligned`, the results are quite different:

```
Performance counter stats for './13x07-false_sharing':

           851,506      cache-references
           231,703      cache-misses
                        # 27.21% of all cache refs
```

The number of both cache references and cache misses is much smaller. This is because there's no need to constantly update the caches in all the cores to keep cache coherency.

In this section, we saw one of the most common performance issues of multithreaded code: false sharing. We saw a function example with and without false sharing and the negative impact false sharing has on performance.

In the next section, we'll go back to the SPSC lock-free queue we implemented in *Chapter 5* and improve its performance.

SPSC lock-free queue

In *Chapter 5*, we implemented an SPSC lock-free queue as an example of how to synchronize access to a data structure from two threads without using locks. This queue is accessed by just two threads: one producer pushing data to the queue and one consumer popping data from the queue. It's the easiest queue to synchronize.

We used two atomic variables to represent the head (buffer index to read) and tail (buffer index to write) of the queue:

```
std::atomic<std::size_t> head_ { 0 };
std::atomic<std::size_t> tail_ { 0 };
```

To avoid false sharing, we can change the code to the following:

```
alignas(64) std::atomic<std::size_t> head_ { 0 };
alignas(64) std::atomic<std::size_t> tail_ { 0 };
```

After this change, we can run the code we implemented to measure the number of operations per second (push/pop) performed by the producer and consumer threads. The code can be found in this book's GitHub repository.

Now, we can run `perf`:

```
perf stat -e cache-references,cache-misses ./13x09-spsc_lock_free_
queue
```

We'll get the following results:

```
101559149 ops/sec

 Performance counter stats for <./13x09-spsp_lock_free_queue>:

       532,295,487      cache-references
       219,861,054      cache-misses                     #   41.30% of
all cache refs

       9.848523651 seconds time elapsed
```

Here, we can see that the queue is capable of about 100 million operations per second. Also, there are roughly 41% cache misses.

Let's review how the queue works. Here, the producer is the only thread writing `tail_` and the consumer is the only thread writing `head_`. Still, both threads need to read `tail_` and `head_`. We've declared both atomic variables as `aligned(64)` so that they're guaranteed to be in different cache lines and there's no false sharing. However, there is true sharing. True sharing also generates cache coherency traffic.

True sharing means that both threads have shared access to both variables, even if each variable is just written by one thread (and always the same thread). In this case, to improve performance, we must reduce sharing, avoiding as much of the read access from each thread to both variables as we can. We can't avoid data sharing, but we can reduce it.

Let's focus on the producer (it's the same mechanism for the consumer):

```
bool push(const T &item) {
    std::size_t tail = tail_.load(std::memory_order_relaxed);
    std::size_t next_tail = (tail + 1) & (capacity_ - 1);
    if (next_tail == cache_head_) {
        cache_head_ = head_.load(std::memory_order_acquire);
        if (next_tail == cache_head_) {
            return false;
        }
    }

    buffer_[tail] = item;
    tail_.store(next_tail, std::memory_order_release);
    return true;
}
```

The `push()` function is only called by the producer.

Let's analyze what the function does:

- It atomically reads the last index where an item was stored in the ring buffer:

  ```
  std::size_t tail = tail_.load(std::memory_order_relaxed);
  ```

- It calculates the index where the item will be stored in the ring buffer:

  ```
  std::size_t next_tail = (tail + 1) & (capacity_ - 1);
  ```

- It checks whether the ring buffer is full. However, instead of reading `head_`, it reads the cached head value:

  ```
      if (next_tail == cache_head_) {
  ```

  Initially, both `cache_head_` and `cache_tail_` are set to zero. As mentioned previously, the goal of using these two variables is to minimize cache updates between cores. The cache variables technique works like so: every time `push` (or `pop`) is called, we atomically read `tail_` (which is written by the same thread, so no cache updates are required) and generate the next index where we'll store the item that's passed as a parameter to the `push` function. Now, instead of using `head_` to check whether the queue is full, we use `cache_head_`, which is only accessed by one thread (the producer thread), avoiding any cache coherency traffic. If the queue is "full," then we update `cache_head_` by atomically loading `head_`. After this update, we check again. If the second check results in the queue being full, then we return `false`.

  The advantage of using these local variables (`cache_head_` for the producer and `cache_tail_` for the consumer) is that they reduce true sharing – that is, accessing variables that may be updated in the cache of a different core. This will work better when the producer pushes several

items in the queue before the consumer tries to get them (same for the consumer). Say that the producer inserts 10 items in the queue and the consumer tries to get one item. In this case, the first check with the cache variable will tell us that the queue is empty but after updating with the real value, it will be OK. The consumer can get nine more items just by checking whether the queue is empty by only reading the `cache_tail_` variable.

- If the ring buffer is full, then update `cache_head_`:

```
head_.load(std::memory_order_acquire);
        if (next_tail == cache_head_) {
            return false;
        }
```

- If the buffer is full (not just that `cache_head_` needs to be updated), then return `false`. The producer can't push a new item to the queue.
- If the buffer isn't full, add the item to the ring buffer and return `true`:

```
buffer_[tail] = item;
    tail_.store(next_tail, std::memory_order_release);
    return true;
```

We've potentially reduced the number of times the producer thread will access `tail_` and hence reduced cache coherency traffic. Think about this case: the producer and the consumer use the queue and the producer calls `push()`. When `push()` updates `cache_head_`, it may be more than one slot ahead of `tail_`, which means we don't need to read `tail_`.

The same principle applies to the consumer and `pop()`.

Let's run `perf` again after modifying the code to reduce cache coherency traffic:

```
162493489 ops/sec

 Performance counter stats for <./13x09-spsp_lock_free_queue>:

       474,296,947      cache-references
       148,898,301      cache-misses                     #   31.39% of
all cache refs

       6.156437788 seconds time elapsed

      12.309295000 seconds user
       0.000999000 seconds sys
```

Here, we can that performance has improved by about 60% and that there is a smaller number of cache references and cache misses.

With that, we've learned how reducing access to shared data between two threads can improve performance.

Summary

In this chapter, we covered three methods you can use to profile your code: `std::chrono`, micro-benchmarking with the Google Benchmark library, and the Linux `perf` tool.

We also saw how to improve multithreaded programs' performance by both reducing/eliminating false sharing and reducing true sharing, reducing the cache coherency traffic.

This chapter provided a basic introduction to some profiling techniques that will be very useful as a starting point for further studies. As we said at the beginning of this chapter, performance is a complex subject and deserves its own book.

Further reading

- Fedor G. Pikus, *The Art of Writing Efficient Programs*, First Edition, Packt Publishing, 2021.
- Ulrich Drepper, *What Every Programmer Should Know About Memory*, 2007.
- Shivam Kunwar, *Optimizing Multithreading Performance* (`https://www.youtube.com/watch?v=yN7C3SO4Uj8`).

Index

A

B

C

D

E

F

G

H

I

N

O

P

Q

R

S

T

U

V

W

X

Y

packtpub.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

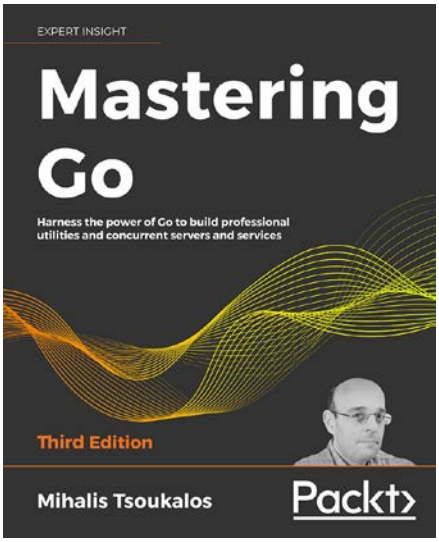
- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at packtpub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.packtpub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:



Mastering Go – Third Edition

Mihalis Tsoukalos

ISBN: 978-1-80107-931-0

- Use Go in production
- Write reliable, high-performance concurrent code
- Manipulate data structures including slices, arrays, maps, and pointers
- Develop reusable packages with reflection and interfaces
- Become familiar with generics for effective Go programming
- Create concurrent RESTful servers, and build gRPC clients and servers
- Define Go structures for working with JSON data

Go Programming - From Beginner to Professional

Samantha Coyle

ISBN: 978-1-80324-305-4

- Understand the Go syntax and apply it proficiently to handle data and write functions
- Debug your Go code to troubleshoot development problems
- Safely handle errors and recover from panics
- Implement polymorphism using interfaces and gain insight into generics
- Work with files and connect to popular external databases
- Create an HTTP client and server and work with a RESTful web API
- Use concurrency to design efficient software
- Use Go tools to simplify development and improve your code

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packtpub.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Share Your Thoughts

Now you've finished *Asynchronous Programming with C++*, we'd love to hear your thoughts! If you purchased the book from Amazon, please click here to go straight to the Amazon review page for this book and share your feedback or leave a review on the site that you purchased it from.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below

`https://packt.link/free-ebook/9781835884249`

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly