CS3211 - Parallel and Concurrent Programming

Sherisse Tan Jing Wen

May 6, 2024

Contents

Chapter 1

Introduction

Parallelism, or True Concurrency, is dictated by the number of hardware threads in the CPU.

Definition 1.0.1 – Concurrency Two or more execution flows make progress at the same time by **interleaving** their executions or by executing instructions (on the CPU) at the same time. Aka. Illusion of Concurrency

Definition 1.0.2 – Parallelism Tasks both make progress and execute simultaneously. Aka. True Concurrency

Generally, Parallelism can only occur in systems with multi-core processors. Otherwise, single-core processors are unable to execute multiple tasks at the same time without any interleaving due to the context switch required.

Note, outside the scope of this module, there exists a concept called Hyperthreading. Some single-core processors are able to support Hyperthreading such that they can have multiple threads actively running at the same time, where the Operating System treats this CPU as if it has two cores.

1.1 Types of Parallelism

There are two main types of Parallelism:

1. Task Parallelism: Doing the same amount of work, but faster
2. Data Parallelism: Doing more work in the same amount of time
3. Pipeline Parallelism: Breaking the entire work into sub-tasks which is each done by a separate thread

Task Parallelism divides the work into tasks to make the threads 'specialists', where each thread focuses on doing a specific task. The overall work can thus be divided by task type, which also helps to separate concerns. In Task Parallelism, we can divide a sequence of tasks such that each thread is responsible for a stage of the pipeline.

In contrast, Data Parallelism involves dividing the overall data into chunks, where the parallel execution of operations on these data is safe. Therefore, Data Parallelism is often used in embarrassingly parallel scenarios.

1.2 Processes and Threads

Concurrency can be achieved using either multiple processes, or multiple threads. However, each of these have their trade-offs.

Multiple processes have a lower performance as compared to using multiple threads, however, they are also safer.

There are 2 types of threads: User-Level Threads and Kernel Threads.

User-Level Threads are those defined by the user in the program, i.e., by using `thread::spawn()`.

In contrast, Kernel Threads refer to the hardware threads available in the CPU, and are thus, available for the OS and the scheduler to work with directly.

Processes do not share memory, reducing the risk of Race Conditions, and thus, Data Races unless there is a communication method set up between the processes which shares data.

However, due to the overhead of the requried system calls, and that all the data structures required must be allocated, initialised and copied, creating new processes is very costly. Communication between processes are also costly as they occur through the usage of system calls.

In contrast, threads share the same registers and stack information, reducing the overhead of context switching and communication. However, threads can cause Race Conditions and Data Races since the address space of the process they spawn from is shared. Thread generation is also faster since no copy of the address space is required.

With processes, there are two ways that the process can interact with the OS: Exceptions and Interrupts.

Exceptions occur synchronously, when errors are raised when executing programs. An Exception Handler is executed.

Interrupts occur asynchronously, when external events interrupt the execution of a program. Typically, interrupts are usually hardware related, for example, when the user presses `ctrl-c`. An Interrupt Handler is executed.

1.3 Building and Execution of Concurrent Programs (C/C++)

The flow for the Building of programs goes as follows: Preprocessor -> Compiler -> Assembler -> Linker

- Preprocessor: Replaces preprocessor directives, i.e., `#include, #define`
- Compiler: Parses the `C++` source code, converting it into assembly
- Assembler: Assembles the assembly code into machine code
- Linker: Produces the final compilation output from the object files produced by the Compiler

The flow for the Execution of built programs is as follows: Compilation and Linking -> Loading -> Execution

- Compilation and Linking: Done by the compiler
- Loading: The Loader is usually specific to the OS
- Execution: Coordinated by the OS; The Program gets access to CPU, Memory, Devices, etc. as required

Chapter 2

Concurrency Concepts

Correctly Synchronised Programs should behave as if:

- Memory operations are actually executed in an order that appears equivalent to some sequentially consistent interleaved execution of the memory operations of each thread in the source code
- Each write appears to be atomic and globally visible simultaneously to all processors

2.1 Synchronisation

2.1.1 Data Races and Race Conditions

When working with Concurrent Programs, there is a need to ensure a correct synchronisation between the processes / threads involved, to prevent any Data Races as a result of the Race Conditions everywhere in the code.

Definition 2.1.1 – Race Condition A flaw that occurs when the timing or ordering of events affect the program's correctness

When there are Race Conditions, the outcome depends on the relative ordering of the execution of operations, whereby the threads / processes *race* to perform their respective operations.

Definition 2.1.2 – Data Race The scenario whereby two or more memory accesses, at least one of which is a WRITE, to the same location are performed concurrently without any synchronisation

Generally, Data Races cause undefined behaviours, as we can no longer guarantee what is the outcome of the Data Race. Note that since Data Races, at least by definition, refers to memory accesses to the **same memory location**, the datatypes of the object matters.

```
struct dataRace {
  int x: 6;
  int y: 8;
}

struct noDataRace {
  int x: 6;
  char y;
}

dataRace obj1 = dataRace {25, 2};
void test1() { obj1.x = 5; } // Thread 1
void test2() { std::cout << obj1.y } // Thread 2

noDataRace obj2 = noDataRace {25, 'a'};
void test3() { obj2.x = 5; } // Thread 3
void test4() { std::cout << obj2.y } // Thread 4
```

Listing 2.1: Data Races based on Memory Location in C

In the above example, there is a data race for `obj1` in Threads 1 and 2, but no data race for `obj2` in Threads 3 and 4.

2.1.2 Synchronisation Mechanisms

Mechanisms to ensure synchronisation include:

- Locks: Primitive with minimal semantics; Usually used as a foundation to build the other mechanisms
- Semaphores: Basic Locks; Easy to get the hang of and understand, but difficult to actually program with
- Monitors: High-Level with implicit effect on operations; Requires language support
- Messages: A Simple model of communication and synchronisation based on the atomic transfer of data across channel(s); Has a direct application in Distributed Systems

Importantly, do not pass data, references or pointers to said data outside the scope of the lock as the data can then be accessed outside of the lock, causing Synchronisation Errors.

Additionally, when working with multiple mutexes within a single Critical Section, it is easy to get deadlocks. As such, where possible, do the following:

- Avoid nested locks
- Avoid calling user-supplied code while holding a lock
- Acquire locks in a fixed order

However, notice that acquiring locks is a busy waiting operation, meaning that the threads do not make progress, but still execute by checking if they can acquire the lock. We thus use Condition Variables to let threads sleep until a specific event occurs.

Condition Variables work by having some other working thread notify ≥ 1 threads waiting on the Condition Variable. The notified threads then wake up, and continue processing.

Condition Variables also use Spurious Wake, whereby the waiting thread reacquires the mutex and checks the condition, but not in direct response to a notification. This is in the event that there is no notification to the thread, but the condition has already been satisfied

2.2 Requirements for Critical Sections in Concurrency

Generally, Critical Sections refer to sections in the code where there are Race Conditions and Synchronisation is required. Critical Sections require the following properties:

- Safety Property: Nothing bad happens
- Liveness Property: Something good happens
- Performance Requirement

More specifically, the following is required:

1. Mutual Exclusion (Mutex): If one thread is in the Critical Section, then no other thread is
2. Progress: If some thread T is not in the Critical Section, then T cannot prevent another thread S from entering the Critical Section. Additionally, a thread in the Critical Section will eventually leave it.
3. Bounded Wait: If some thread T is waiting to enter the Critical Section, then T must eventually enter.
4. Performance: The overhead of entering and exiting the Critical Section is small w.r.t the work being done within it.

Each of these properties must hold for each run, although the performance can depend.

2.2.1 Deadlocks

Deadlocks exist **iff** the following four conditions hold simultaneously:

1. Mutual Exclusion: At least one resource must be held in a non-sharable mode
2. Hold and Wait
3. No preemption
4. Circular Wait

In layman's terms: Deadlock exists when the waiting process is still holding on to another resource that the first needs before it can finish. It is a problem that can arise when processes compete for access to limited resources, and when processes are incorrectly synchronised.

2.2.2 Livelocks

Livelocks are different from Deadlocks in the sense that it is in essence, a continuous back-and-forth Deadlock. Livelocks occur when two or more threads require the same resource which is protected by a Synchronisation Mechanism, i.e. a Lock. However, instead of proceeding and entering the Critical Section, the threads repeatedly release and acquire the Synchronisation Mechanism.

You can think of a Livelock as the meme with two people running around a pillar.

2.2.3 Starvation

Starvation violates the Bounded Wait Requirement of Critical Sections. As its name implies, it refers to the scenario where threads never get a chance to execute, whether it is because there is a Deadlock / Livelock and it never gets to enter the critical section, or because it is always preempted by another thread.

Overall, there is a lot of challenges with writing Concurrent Programs, despite the advantages it can offer us in terms of performance. Concurrency can also help with the separation of concrens within programs, especially if a task-based paradigm is followed (See: Go).

However, concurrent programs are typically buggier than other programs, due to the need for synchronisation, and they are also difficult to maintain due to the naturally more complex code required. It is also necessary to consider the overhead as a result of generating new processes and/or threads, and if the performance increase offered outweighs this overhead.

2.3 Task Dependency Graph

To identify the dependencies between different tasks in the concurrent program, and thus figure out what tasks can be done in parallel and what must be completed before another task can execute, we can model the tasks using a Graph.

A valid topological ordering for the task dependency graph would thus be a valid ordering for tasks to execute sequentially without errors.

2.4 Amdahl's Law

Amdahl's Law says that the Speedup of parallel execution is limited by the fraction of the algorithm that cannot be parallelized (f).

2.5 Ownership

The concept of Ownership refers to the idea that each object must have exactly 1 owner, exactly 1 main reference to it. This concept is especially important in Rust, whereby Ownership is employed to ensure safe concurrency, as well as to assist in garbage collection.

In C++, for objects placed in the free store (i.e., Heap, Dynamic Memory), the usage of `new()`, `malloc()` and `delete()`, `free()` are utilised to achieve this Ownership concept.

2.6 Lifetimes

Each object also has its own Lifetime, or Lifecycle, which refers to the time between its creation, and its destruction. The lifetime of an object is equal to, or nested within the lifetime of its storage, i.e., if an object is initialised in a function, it's lifetime ends when the function returns.

Note that, depending on how the object is initialised, it is also not guaranteed that the object's lifetime will extend beyond that of the function even if it is returned from the function.

It is crucial to be able to understand where an object's lifetime starts and ends, as it is possible for *dangling references*, whereby an operation is made on a reference to an object which is already destroyed.

2.7 Resource Acquisition and Initialisation (RAII)

RAII is an important concept in `C++` and `Rust`. In these languages, each object has a constructor and a destructor which are called at the start and end of an object's lifetime respectively.

2.8 Memory Model

The Memory Model exists to give us a guarantee that we can get the same behaviour on all architectures.

Typically, Processors actually reorder the operations in the source code to provide better performance, i.e., to reduce Cache Misses. However, agressive reordering gets complicated in concurrent programs as these optimisations are not visible to programmers.

The C++ Compiler follows the As-If Rule, which states that the compiler is permitted to perform any changes to the program as long as the following remains true:

- Accesses (both Read and Write) to volatile objects occur strictly according to the semantics of the expressions in which they occur. In particular, they are not reordered w.r.t. other volatile accesses on the same thread.
- At program termination, data written to files is exactly as if the program was executed as written
- Prompting text which is sent to interactive devices will be shown before the program waits for input

However, Programs with undefined behaviour (i.e., as a result of Data Races) are free from the above As-If Rule.

2.8.1 Atomics

Atomics are used to enforce a modification order across threads, allowing us to better reason with the program's correctness.

A Modification Order is composed of all writes to an object from all threads in the program. It varies between runs, but all threads must agree on the same modification order in the same run.

Once a thread has seen a particular entry in the modification order, the following must hold:

- Subsequent reads from that thread must return the same, or a later, value
- Subsequent writes from the that thread must occur later in the modification order

2.8.2 Relationships between Operations

Before we can look into the various methods of Memory Ordering, we first have to understand the various Relationships possible between Operations, so that we can better trace the Memory and Modification Order.

Sequenced-Before (sb)

Within the same thread, an evaluation A may be Sequenced-Before another evaluation B, indicating that A must finish before B starts.

Synchronises-With (sw)

The Synchronises-with r/s appears between Atomic load and store operations in separate threads. An atomic store operation, W in a thread A on a variable x Synchronises-With an atomic load operation R in thread B on x if R reads the value stored by W.

Happens-Before (HB)

Regardless of threads, A Happens-Before B if any of the following is true:

- A is Sequenced-Before B
- A Synchronises-With B
- A Happens-Before X and X Happens-Before B

There is also a stronger version, aka. Strongly Happens-Before which requires any of the following to be true:

- A is Sequenced-Before B
- A Synchronises-With B, and both A and B are Sequentially Consistent Atomic Operations
- A is Sequenced-Before X and X Happens-Before Y and Y is Sequenced-Before X
- A Strongly Happens-Before X and X Strongly Happens-Before B

Informally, if A Strongly Happens-Before B, then A appears to be evaluated before B in all contexts.

Additionally, note that Happens-Before is a Transitive Relationship.

Visible Side-Effects

The side-effect A on a scalar M is visible with respect to B on M if both of the following is true:

1. A Happens-Before B
2. There is no other side effect X to M where A Happens-Before X and X Happens-Before B

If A is visible with respect to B, then the longest contiguous subset of the side-effects to M, in modification order, where B does not Happens-Before it is known as the visible sequence of side-effects. Here, note that the value of M determined by B will be the value stored by one of these side-effects.

Inter-thread synchronisation boils down to establishing relationships between operations, and defining which side-effects become visible under what conditions.

```
atomic_int x, y;

void t1() { // Thread 1
  y = 1;
  x.store(1, std::memory_order_release);
}

void t2() { // Thread 2
  while (x.load(std::memory_order_acquire) != 1) {};
  std::cout << y;
}
```

Listing 2.2: Visible Side-Effects Example in C

In the above example, note that there is a Synchronises-With r/s between the two threads at the store and load, with release-acquire memory ordering, and thus, a Happens-Before r/s with `y=1` and `std::cout << y`. Thus, the Visible Side Effects that can be observed at Line 8 is:

- $y = 1$ from Thread 1
- $x = 1$ from Thread 1

2.8.3 Modification Order

All modifications to any particular Atomic variable occur in a total order specific to that Atomic variable. The following 4 requirements are guaranteed for all atomic operations:

1. Write-Write Coherence
2. Read-Read Coherence
3. Read-Write Coherence
4. Write-Read Coherence

Write-Write Coherence

If an evaluation A that modifies some Atomic variable M Happens-Before an evaluation B that also modifies M, then A appears earlier than B in M's Modification Order.

Read-Read Coherence

If a value computation A of some Atomic variable M Happens-Before another value computation B on M, and if the value of A comes from a write X on M, then the value of B must be either the value stored by X, or the value stored by a side effect Y on M that appears later than X in the Modification Order of M

Read-Write Coherence

If a value computation A of some Atomic variable M Happens-Before an operation B on M, then the value of A comes from a side effect X that appears earlier than B in the Modification Order of M.

Write-Read Coherence

If a side effect X on an Atomic object M Happens-Before a value computation B of M, then the evaluation of B shall take its value from X or a side effect Y that follows X in the Modification Order of M

2.8.4 Memory Order

There are 3 main types of Memory Ordering which is used to define how memory accesses, including regular, non-atomic memory accesses, are to be ordered around an atomic operation:

1. Sequentially Consistent Ordering
2. Non-Sequentially Consistent Ordering
   (a) Relaxed Ordering
   (b) Release-Acquire Ordering

Sequentially Consistent Ordering

Sequentially consistent ordering restricts the compiler from reordering operations. It implies that the behaviour of the program is consistent with a simple, sequential view of the world, which all threads agree on.

A Sequentially Consistent Store *synchronises-with* a Sequentially Consistent Load of the same variable. However, note that this typically results in a large performance penalty, especially on weakly-ordered machines with many cores.

Non-Sequentially Consistent Ordering

In constrast to Sequentially Consistent Ordering, with Non-Sequentially Consistent Orderings, there is no single global order of events. Different threads can see different views of the same operations, however they **must** agree on the Modification Order of each individual variable.

Relaxed Ordering

Operations on Atomics performed with Relaxed Ordering still guarantee Atomicity and Modification Order Consistency, but do not participate in Synchronises-With relationships. Regardless, within the same thread, operations on the same variable still obey the Happens-Before relationship, just that there is no guarantee on the ordering relative to other threads.

In order to enforce memory-ordering constraints without modifying data, we can use Fences. Fences are typically used with the Relaxed Memory Ordering, and puts a line in the code that certain operations can't cross.

These are constructed using `atomic_thread_fence` which, when used with `memory_order_release`, prevents all preceding read and writes from moving past subsequent stores.

Acquire-Release Ordering

Acquire-Release Ordering has no total Modification Order, but does introduce some synchronisation between threads. If an Atomic store in a thread A is tagged with `memory_order_release`, and an atomic load in thread B tagged with `memory_order_acquire` reads the value stored in A, then the Atomic store in A Synchronises-With the Atomic load in B.

Then, all memory writes that Happens-Before the Atomic store in A (from thread A's pov) become Visible Side Effects to B. Note however, that the Synchronises-With is only established between the two threads A and B. Other threads do not see this same memory order.

Chapter 3

Synchronisation Problems

Generally, there are a few classicial synchronisation problems that model problems we encounter when creating concurrent programs.

Synchronisation Problem	Computer Systems Problem
Barrier	Wait until threads / processes reach a certain point in the execution
Producer-Consumer	Model interactions between a processor and devices that interact through FIFO channels
Reader-Writer	Model access to shared memory
Dining Philosophers	Allocation of limited resources to a group of processes in a deadlock-free and starvation-free manner
Barbershop	Coordinate the execution of a processor
FIFO Semaphore	Needed to avoid starvation and increase fairness in the system
H2O	Allocation of specific resource to a process
Cigarette Smokers	The agent represents an Operating System that allocates resources, and the smookers represent applications that need resources

Table 3.1: Classical Synchronisation Problems and the Computer System Problems they model

3.1 Producer-Consumer

In the Producer-Consumer Problem, processes share a buffer (whether bounded or unbounded). Producers are then able to produce items to insert into the buffer, as long as it is not full, and Consumers are able to remove items from the buffer as long as the buffer is not empty.

3.2 Barrier

In the Barrier Problem, we aim to stop all threads / processes at a certain point and wait until all threads have reached said point, before allowing them to contiune. This appears in many collective routines as part of directive-based parallel languages, and is also the basis of implementation for `WaitGroup.Wait()` in `Go`.

3.3 Reader-Writer

In the Reader-Writer Problem, processes share a data structure, rather than a buffer as in the Producer-Consumer Problem.

The reader retrieves information from the data structure, while the writer modifies the information in the data structure. The writer must have exclusive access, that is, there can only be 1 active writer at a time, but multiple readers can access the data structure at the same time.

The solution for Reader-Writer Problem is to use a Turnstile:

```
// Writer
turnstile.wait()
roomEmpty.wait()
// CS for writer
turnStile.signal()
roomEmpty.signal()

// Reader
turnstile.wait()
turnstile.signal()
readSwitch.lock(roomEmpty)
// CS for reader
readSwitch.unlock(roomEmpty)
```

Listing 3.1: Turnstile Solution for Reader-Writer

The solution works such that as long as there is no writer, readers can continue to pass the turnstile. However, once a writer arrives, it calls `.wait()` on the turnstile, preventing other readers from continuing to pass through. After the current readers in the CS have exited, the writer enters its CS. Then, only after the writer exits, can more readers continue to pass through the turnstile. This prevents starvation of the writers.

3.4 Barbershop

The Barbershop problem consists of a waiting room with n chairs, excluding the barbar chair. When there are no customers, the barber goes to sleep. A customer wakes up the barber if he is sleeping. If the barber is busy, but there are chairs available, the customer sits in one of the free chairs. If a customer enters but all the chairs are occupied, the customer leaves.

The general solution for the barbershop problem is as follows:

```
// Customer
wait(mutex);
if (customers == n) {
  signal(mutex);
  exit();
}

customers += 1;
signal(mutex);
signal(barber);
getHairCut();
signal(customerDone);
wait(barberDone);
wait(mutex);
customers -= 1;
signal(mutex);


// Barber
while (TRUE) {
  wait(customer);
  signal(barber);
  cutHair();
  wait(customerDone);
  wait(barberDone);
}
```

Listing 3.2: Barbershop Solution Psuedocode

3.5 H2O Problem

In the H2O Problem, Hydrogen and Oxygen atoms arrive at the factory. We only want to bond 2 H with 1 O. Additionally, only 1 bonding can occur at a time, and other atoms block while bonding.

The solution (in `C`, `Go` and `Rust`) is as follows:

```
struct WaterFactory {
  std::barrier<> oBarrier(1), hSem{2}
  std::barrier<> barrier;
  WaterFactory() : barrier(3) {}
}

void oxygen(void (*bond)) {
  oBarrier.arrive_and_wait();
  barrier.arrive_and_wait();
  bond();
  oSem.release();
}

void hydrogen(void (*bond)) {
  barrier.arrive_and_wait();
  bond();
  hSem.release();
}
```

Listing 3.3: H2O Solution using Barriers and Semaphores in C

```
type WaterFactoryWithLeader struct {
  oxygenMutex chan struct{}
  precomH chan chan struct{}
}

func (wf *WaterFactoryWithLeader) hydrogen(bond func()) {
  commit := make(chan struct{})
  wf.precomH <- commit
  <- commit
  bond()
  commit <- struct{}{}
}

func (wf *WaterFactoryWithLeader) oxygen(bond func()) {
  <-wf.oxygenMutex // Using a channel as a mutex

  // Receive arrival requests from 2 hydrogen atoms
  h1 :=wf.precomH
  h2 :=wf.precomH

  h1 <- struct{}{}
  h2 <- struct{}{}
  bond()

  <-h1
  <-h2

  wf.oxygenMutex <- struct{}{}
}
```

Listing 3.4: H2O Solution using Barriers and Semaphores in Go

```rust
use futures::future::join_all;
use std::sync::Arc;
use tokio::sync::{Barrier, Semaphore}

fn bond(s: &str) {
  println!("bond {s}");
}

struct WaterFactory {
  o_sem: Semaphore,
  h_sem: Semaphore,
  barrier: Barrier,
}

impl WaterFactory {
  fn new() -> Self {
    Self {
      o_sem: Semaphore::new(1),
      h_sem: Semaphore::new(2),
      barrier: Barrier::new(3),
    }
  }

  async fn oxygen(&self, bond: impl FnOnce()) {
    let _ = self.o_sem.acquire().await.unwrap();
    self.barrier.wait().await;
    bond();
  }

  async fn hydrogen(&self, bond: impl FnOnce()) {
    let _ = self.h_sem.acquire().await.unwrap();
    self.barrier.wait().await;
    bond();
  }
}

#[tokio::main]
async fn main() {
  let n = 10;
  let f = Arc::new(WaterFactory::new());

  let hs = (0..n * 2).map(|i| {
    let f = f.clone(); // Clone shared pointer; increases ref count
    tokio::spawn((|| async move {
      f.hydrogen(|| bond(&format!("h{i}"))).await;
    })())
  });
  let os = (0..n).map(|i| {
    let f = f.clone(); // Clone shared pointer; increases ref count
    tokio::spawn((|| async move {
      f.oxygen(|| bond(&format("o{i}"))).await;
    })())
  });

  // Join all task handles to force them to execute concurrently
  join_all(Iterator::chain(hs, os)).await;
}
```

Listing 3.5: H2O Solution in Rust

Chapter 4

Concurrency Patterns

4.1 Confinement

Confinement refers to 'confining' each Goroutine (or thread) to operate on a chunk of data. This achieves safe operation on the set of data with good performance, since the data is protected by confinement.

There are two types of Confinement:

1. Ad-hoc Confinement: Not enforced; Done by the programmer and thus, easy to violate
2. Lexical Confinement: Restricts the access to shared locations

Note that for Lexical Confinement, it may be difficult to determine the best size of each data slice.

```
printData := func(wg *sync.WaitGroup, data []byte) {
  defer wg.Done()
  var buff bytes.Buffer
  for _, b := range data {
    fmt.Fprintf(&buff, "\%c", b)
  }
  fmt.Println(buff.String())
}

var wg sync.WaitGroup
wg.Add(2)
data := []byte("golang")
go printData(&wg, data[:3])
go printData(&wg, data[3:])
wg.Wait()
```

Listing 4.1: Lexical Confinement for data parallelism in Go

4.2 Pipeline

In Concurrent Programs, the Pipeline pattern separates the concerns of each stage such that each stage can be modified independently of each other. The Pipeline pattern also supports the fan-in and/or fan-out of various stages in the pipeline.

See Section 6.1 for more information on how the Pipeline Pattern works in Go specifically.

The Pipeline Design increases efficiency as work and resources can be divided among each stage such that they all take the same time to complete their tasks. However, it is not obviously faster than a task pool and tweaking is required to make pipelining more efficient. Nevertheless, Pipelining works better if there is a cap on a specific resource which is required by all tasks in the task pool (although at different times, not at the same time)

4.3 Fan-in, Fan-out

The Fan-in, Fan-out Patterns works when the stages in a Pipeline take different times to finish executing. Fan-out refers to multiple consumers to a single stage, and allows multiple goroutines to handle the input from the pipeline. After Fanning Out, we then use Fan-In, to combine all the results from the different goroutines into a single result in the next channel.

Typically, we Fan-Out a stage of the pipeline if:

- It doesn't rely on values the stage had calculated before (i.e., not persistent)
- It takes a long time to run

Note that there is no guarantee on the order which the different goroutines fanned-out run, nor in what order they return.

As a general rule of thumb, the number of goroutines fanned-out is `runtime.NumCPU()` by default, although the code can be profiled to make subsequent adjustments to enhance the performance.

When Fanning-In, consumers read from the multiplexed channel and a single goroutine is started for each incoming channel, transferring the information from the multiple streams into the single multiplexed channel.

```
done := make(chan interface{})
defer close(done)
start := time.Now()
rand := func() interface{} { return rand.Intn(500000000)}
randIntStream := toInt(done, repeatFn(done, rand))

numFinders := runtime.NumCPU()
fmt.Printf("Spinning up \%d prime finders.\n", numFinders)

finders := make([] <-chan interface{}, numFinders)
fmt.Println("Primes: ")

for i := 0; i < numFinders; i++ {
  finders[i] = primeFinder(done, randIntStream)
}

for prime := range take(done, fanIn(done, finders...), 10) {
  fmt.Printf("\t\%d\n", prime)
}

fmt.Printf("Search took: \%v", time.Since(start))
```

Listing 4.2: Fan-Out, Fan-In Example in Go

Chapter 5

Concurrency in C++

In C++, Concurrency is typically achieved by modelling the program in terms of threads, synchronising the access to shared resources between them and using thread pools to limit the number of threads that must be handled.

5.1 Threads

There is a large variety of methods to create threads in C++

```
void do_some_work();
std::thread thread1(do_some_work);

class background_task {
    public:
        void operator()() const {
            do_something();
            do_something_else();
        }
};

background_task f;
std::thread thread2(f);

std::thread thread3((background_task()));
std::thread thread4{background_task()};

std::thread thread5([] {
    do_something();
    do_something_else();
 });
```

Listing 5.1: Creating threads in C++

Additionally, we can pass arguments to thread functions in a variety of ways.

```
void f(int i, std::string const& s);
std::thread t(f, 3, "hello");

void f(int i, std::string const& s);
void oops(int some_param) {
    char buffer[1024];
    sprintf(buffer, "\%i", some_param);
    std::thread t(f, 3, buffer);
    t.detach();
}

void f(int i, std::string const& s);
void not_oops(int some_param) {
    char buffer[1024];
    sprintf(buffer, "\%i", some_param);
    std::thread t(f, 3, std::string(buffer));
    t.detach();
}
```

```
void update_data_for_widget(widget_id w, widget_data& data);
void oops_again(widget_id w) {
    widget_data data;
    std::thread t(update_data_for_widget, w, data);
    display_status();
    t.join();
    process_widget_data(data);
}

std::thread t(update_data_for_widget, w, std::ref(data));
```

Listing 5.2: Passing arguments to a thread function

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

void hello() {
    std::cout << "Hello World Concurrent";
}

int main() {
    std::thread t(hello);
    t.join();

    std::count << "Thread t finished executing";
}
```

Listing 5.3: Joining threads in C++

`t.join()` blocks and waits for the thread `t` to finish executing and stop, before continuing. Make sure to join the thread even if an exception occurs.

`t.detach()` does not wait for the thread to complete, but may result in the main thread finishing before the thread `t` does. Thus, "Hello World Concurrent" may not be printed at all. Additionally, once a thread has been detached, it cannot be joined again.

```
void some_function();
void some_other_function();
std::thread t1(some_function):
std::thread t2 = std::move(t1);
t1 = std::thread(some_other_function);
std::thread t3;
t3 = std::move(t2);
s1 = std::move(t3); // Does not work since t1 already has a thread


std::thread f() {
    void some_function();
    return std::thread(some_function);
}
std::thread g() {
    void some_other_function(int);
    std::thread t(some_other_function, 42);
    return t;
}


void f(std::thread t);
void g() {
    void some_other_function();
    f(std::thread(some_function));
    std::thread t(some_function);
    f(std::move(t));
}
```

Listing 5.4: Transferring Ownership of Threads

5.2 Synchronisation

Generally, Synchronisation in C++ uses Lock-based Mechanisms such as Mutexes. These are constructed using `std::Mutex`.

```
#include <mutex>

std::mutex some_mutex;
void increment(int new_value) {
    some_mutex.lock();
    new_value++;
    some_mutex.unlock();

    return new_value;
}
```

Listing 5.5: Using Mutexes in C++

However, notice that both `.lock()` and `.unlock()` has to be called. `.unlock()` also has to be called when there are exceptiions which result in errors. The alternative is to use `std::lock_guard` which, using RAII, locks the mutex on construction and unlocks it on destruction.

```
#include <mutex>

std::mutex some_mutex;
void increment(int new_value) {
    std::lock_guard<std::mutex> guard(some_mutex);
    new_value++;
    return new_value;
}
```

Listing 5.6: Mutexes with lockguard in C++

Other types of locks used include:

- `std::unique_lock`: Instance doesn't always own the mutex that it is associated with; Allows for locking the mutex later using `std::defer_lock`
- `std::lock()`: Lock ≥ 1 mutex(es) at once without risking deadlock
  - `std::adopt_lock` can be used to indicate to `std::lock_guard` objects that the mutex(es) are already locked
  - `std::lock_guard` will attempt to adopt the ownership of the existing lock on the mutex rather than attempt to lock the mutex
- `std::scoped_lock`: Instance accepts and locks a list of mutexes; Behaves similarly to `std::lock()`

```
class some_big_object;
void swap1(some_big_object& lhs, some_big_object& rhs);

class X {
    private:
        some_big_object some_detail;
        std::mutex m;
    public:
        X(some_big_object const& sd):some_detail(sd){}
        friend void swap(X& lhs, X& rhs) {
            if (& lhs == & rhs)
                return;
            std::lock(lhs.m, rhs.m);
            std::lock_guard<std::mutex> lock_a(lhs.m, std::adopt_lock);
            std::lock_guard<std::mutex> lock_b(rhs.m, std::adopt_lock);
            swap(lhs.some_detail, rhs.some_detail);
        }
};

void swap2(X& lhs, X& rhs) {
    if (& lhs == & rhs)
        return;
    std::scoped_lock<std::mutex, std::mutex> guard{lhs.m, rhs.m};
    swap(lhs.some_detail, rhs.some_detail);
}
```

Listing 5.7: Locking multiple mutexes at once

Condition Variables are constructed using either:

- `std::condition_variable`: Works with `std::mutex`; Is simpler, lightweight and has less overhead
- `std::condition_variable_any`: Works with anything mutex-like; Potentially has additional costs in terms of size, performance or OS resources

During a call to `.wait()` of a Condition Variable, it:

- checks the supplied condition any number of times
- checks the condition with the mutex locked
- returns immediately iff the function provided to test the condition returns true

```
std::mutex mut;
std::queue<data_chunk> data_queue;
std::condition_variable data_cond;

void data_preparation_thread() {
    while (more_data_to_prepare()) {
        data_chunk const data=prepare_data();
        {
            std::lock_guard<std::mutex> lk{mut};
            data_queue.push(data);
        }

        data_cond.notify_one();
    }
}

void data_processing_thread() {
    while (true) {
        std::unique_lock<std::mutex> lk{mut};
        data_cond.wait(lk, [] {
            return !data_queue.empty();
        });
        data_chunk data = data_queue.front();
        data_queue.pop();
        lk.unlock();
        process(data);
        if (is_last_chunk(data))
            break;
    }
}
```

Listing 5.8: Condition Variables in C++

Chapter 6

Concurrency with Go

In Go, concurrency is based on Communicating Sequential Processes (CSP). The program is modelled in terms of the tasks that need to be executed, synchronising the tasks by allowing for communication between them. Goroutines are spinned up to run independently of each other and they communicate through Channels.

Operation	Channel State	Result
Read	nil	Block
	Open and Not Empty	Value
	Open and Empty	Block
	Closed	<default_value>, false
	Write Only	Compilation Error
Write	nil	Block
	Open and Not Full	Write Value
	Open and Full	Block
	Closed	Panic
	Receive Only	Compilation Error
close	nil	Panic
	Open	Closes Channel; reads succeed until channel is drained, then reads default value
	Closed	Panic
	Receive Only	Compilation Error

Table 6.1: Go Channel States

Goroutines are functions that run independently. They share the same address space as other go routines, and are cheaper than threads. They are a special class of coroutines (concurrent subroutine), s.t. when a goroutine blocks, that thread blocks without affecting other goroutines. They can also be preempted for the runtime to suspend them.

```
var wg sync.WaitGroup

for _, salutation := range []string{"hello", "greetings", "good day"} {
  wg.Add(1) // Adds 1 to the count of the number of goroutines spinned up
  go func(salutation string) {
    defer wg.Done() // Defers .Done() until end of scope; Indicates completed goroutine
    fmt.Println(salutation)
  }(salutation)
}
wg.Wait() // Waits for all the go routines to complete
```

Listing 6.1: Goroutine usage in Go

Additionally, note that goroutines are not garbage collected by default and the programmer has the responsibility of preventing leaks. In order to prevent goroutines from leaking, ensure the proper termination of the goroutine when:

- It has completed its work
- It cannot continue its work due to some unrecoverable error
- It's told to stop working

Generally, if a goroutine A creates another goroutine B, it is the responsibility of A to ensure that it can stop B.

```
doWork := func(strings <-chan string) <-chan interface{} {
  completed := make(chan interface{})
  go func() {
    defer fmt.Println("doWork exited")
    defer close(completed)
    for s := range strings { // Forever blocks attempting to read
      fmt.Println(s)
    }
  }()
  return completed
}
doWork(nil) // nil channels block when attempting to read or write
fmt.Println("Done")
```

Listing 6.2: Leaking Goroutine Example in Go

Channels serve as conduits for a stream of information. Values may be read into the channel, and read somewhere down stream. This helps to ensure separation of concerns as no knowledge is required about the other parts of the program, just what data is and can be read from the channel.

A channel is a reference to a place in memory where the channel resides, and the references to this channel can be passed around.

Generally, the owner of the channel is the goroutine that instantiates, writes and closes the channel. For unidirectional channels, owners have a write-access view into the channel, whereas consumers, or utilizers, only have a read access view into the channel. As such, consumers have to responsibly handle closed channels, or blocking for any reason.

```
  var dataStream chan interface{}
  datastream = make(chan interface{})

  var receiveChan <-chan interface{}
  var sendChan chan<- interface{}
  dataStream := make(chan interface{})
  receiveChan = dataStream
  sendChan = dataStream
```

Listing 6.3: Creating channels in Go

```
  intStream := make(chan int)
  go func() {
    defer close(intStream)
    for i := 1; i <= 5; i++ {
      intStream <- i // Send i into the channel intStream
    }
  }()
  for integer := range intStream {
    fmt.Printf("\%v ", integer)
  }


  begin := make(chan interface{})
  var wg sync.WaitGroup
  for i := 0; i < 5; i++ {
    wg.Add(1)
    go func(i int) {
      defer wg.Done()
      <-begin
      fmt.Printf("\%v has begun\n", i)
    }(i)
  }
  fmt.Println("Unblocking goroutines..")
  close(begin)
  wg.Wait()
```

Listing 6.4: Synchronising using channels in Go

Note that in the Go Memory Model (Section 2.8), a Send on a channel is Synchronised-Before the completion of the corresponding receive from that channel. Additionally, the receive from an unbuffered channel is Synchronised-Before the send on that channel completes. In layman's terms, the Send and Receive must occur at the same time.

This is part of the guarantee that when data is passed into a channel, it is always read by something else. However, this also causes some potential deadlock issues, especially when using `WaitGroup`.

```
package main
import (
  "fmt"
  "sync"
)

func main() {
  ch := make(chan int) // make an unbuffered channel
  var wg sync.WaitGroup
  for i := 0; i < 1000; i++ {
      wg.Add(1)
      go func() {
        defer wg.Done() // defer until end of goroutine
        count := <-ch   // read from channel; blocking dequeue
        count++         // safely add 1 as the exclusive owner
        ch <- count     // write back to channel; blocking enqueue
      }()
  }
  ch <- 0 // send initial value; blocking enqueue

  wg.Wait() // Wait for all goroutines to complete
  fmt.Println("Count: ", <-ch) // Read final result
}
```

Listing 6.5: WaitGroup Deadlock Example in Go

Notice that in the above example, we reach a deadlock in the final iteration as the remaining write to channel operation in the last goroutine to finish execute needs, by definition, to be executed at the same time as another read operation which occurs in the main goroutine. However, that last read operation is blocked by `wg.Wait()` as `defer` is only called after the goroutine goes out of scope.

In this scenario, we can solve it by using a buffered channel, however, note that buffered channels can more easily result in data races and provide greater challenges in ensuring safe concurrency.

`select` statements are used to compose multiple channels together, locally, within a single function or type, and globally, at the intersection of two or more components in a system, to form larger abstractions.

When working with `select`, it is important that we can handle the following scenarios:

- Multiple channels have something to read: By default, the Go runtime will perform any ready cases with equal probability
- There are never any channels that are ready
- We want to perform some operation, but none of the channels are currently ready.

In `select`, all channel reads and writes are considered simultaneously to see if any are ready. The entire statement blocks if none of the channels are ready, and does not execute a default case instead. We can however, define an empty default case, which when used in conjunction with a `for` loop, allows us to do other work while waiting. Aka. `For-select loop`

Typically, the `For-select loop` pattern is used when sending iteration variables out on a channel, or when looping indefinitely and waiting to be stopped (i.e., when the channel is closed).

```
done := make(chan interface{})
go func() {
  time.Sleep(5*time.Second)
  close(done)
}()
workCounter := 0
loop:
for {
  select {
  case <-done: // signals end of work for the done channel
    break loop
  default:
  }
  // Simulate Work while waiting
  workCounter++
  time.Sleep(1*time.Second)
}
fmt.Printf("\%v cycles of work done.\n", workCounter)
```

Listing 6.6: For-Select Loop in Go

6.1 Pipeline Pattern in Go

In Go, each stage is a (group of) goroutines running the same function, and each (series of) stages is connected by channels.

In each stage, the goroutines:

- receive values from upstream via inbound channels
- perform some operation(s) on that data
- send values downstream via outbound channels

Additionally, each stage can have any number of inbound and outbound channels. The exception would be the first and last stages which can only have 1 source / producer and 1 sink / consumer respectively.

Chapter 7

Concurrency in Rust

Generally, Rust provides stronger safety guarantees without compromising on performance similar to that of `C/C++`. Rust does not have a Garbage Collector and a Runtime (like Go). It does this by preventing simulataneous mutation and aliasing, by enforcing ownership and borrowing, which, while it exists in other languages like `C++` and `Go`, is not strictly enforced.

In Rust, giving ownership is the default and deep copies of data is explicity by using `clone()`. Ownership thus prevents double-free as we are guaranteed that only the owner frees, and Borrowing prevents use-after-free

There are two types of borrows in Rust:

- Immutable / Shared Borrow: Allows Aliasing, but not mutation
- Mutable Borrow: Allows mutation, but not aliasing

```
fn main() {
  let mut book = Vec::new();
  book.push();
  book.push();
  publish(book); // ownership of 'book' passed to publish()
  publish(book); // Compilation Error; main func no longer owner of 'book'
}

fn publish(book: Vec<String>) {
  ...
}
```

Listing 7.1: Ownership in Rust

```
fn main() {
  let mut book = Vec::new();
  book.push();
  {
    let r = &book; // Shared Borrow of book; No 'mut'
    book.push(); // Compilation Error
    r.push(); // Compilation Error
  }
  book.push(); // Success; Book can be mutated
}
```

Listing 7.2: Immutable / Shared Borrows in Rust

```
fn main() {
  let mut book = Vec::new();
  book.push();
  {
    let r = &mut book; // Mutable borrow of book
    book.len(); // Compilation Error
    r.push(); // Success; Reference can be mutated
  } // Mutable Borrow ends
  book.push(); // Success; Book can be mutated
}
```

Listing 7.3: Mutable Borrows in Rust

Rust's main Concurrency Paradigm is that of Async Programming, which is usually for handling blocking read-/write operations, typically I/O ones. As such, Rust heavily utilises Futures to allow for non-blocking I/O where possible. It is also important to note that Futures should thus not be blocking.

Futures aid in State Management by keeping track of in-progress operations along with the associated states in a single package. Futures represent values that will exist sometime in the future, but for which the calculation has not occurred or completed yet. The Event loop, runtime for Futures, keep polling the Future until it is ready, and runs all the other code that it can run in the meantime.

An executor loops over Futures that can currently make progress and calls `.poll()` on them to get their status, i.e., Ready or Pending. If no Future can make progress, the executor goes to sleep until ≥ 1 Future(s) calls `.wake()`. Executors can be single threaded or multi-threaded and, when running on multi-core machines, Futures can truly run in parallel.

A popular executor in the `Rust` ecosystem is `Tokio` which, by default, is multi-threaded.

An `async` function is a function that returns a Future. Any Futures used in said function are chained together by the compiler.

`.await` waits for a Future to complete and gets its value. As such, `.await` can only be called in an async function or block of code.

Async functions have no stack, which is why they are sometimes also referred to as stackless coroutines. However, the executor thread still has a stack which is used to run normal synchronous functions, but this stack isn't used to store states when switching between async tasks. Instead, all states are self contained in the generated Future.

Additionally, we cannot recurse on async functions as the Future returned by them need to have a fixed size known at compile-time in order to allocate the appropriate memory space for them.

Chapter 8

Debugging Concurrent Programs

As expected, one of the main challenges when writing Concurrent Program is debugging them when something goes wrong. Typically, concurrent related bugs can range from things like Unwanted Blocking (i.e., Deadlock, Livelock, etc.), to Race Conditions, and even to Early Termination of Programs.

Some other common errors are:

- Dangling Pointers
- Double Free
- Use after Free
- Lifetime Issues

8.1 Techniques

Generally, techniques for locating bugs are:

- Looking at the code (lol)
- Testing

Reviewing code by eye-power

Some questions to consider when reviewing the code written:

- Which data needs to be protected from concurrent access?
- How do you ensure that the data is protected?
- Where in the code could other threads be at this time?
- Which mutexes does this, or other threads hold?
- Are there any ordering requirements between the operations done in these threads? Are they enforced?
- Is data loaded by this thread still valid? Could it have been modified by other threads?
- If we assume another thread is modifying this data, what would that mean and can we ensure this never happens?

Testing multithreaded code

Note that testing multithreaded code is, in itself, difficult as the code may not always fail and may only fail sometimes. It is also difficult to reproduce the problem even if we see a failing test.

Generally, when writing the test for a piece of code, follow these guidelines:

- Run the smallest amount of code that could potentially demonstrate a problem
- Eliminate concurrency from the test to verify if the problem is concurrency-related first
- Run the multithreaded code on both multicore and singlecore processors

Additionally, we also want to test the performance of our concurrent program as we expect a (significant) speedup.

8.2 Debugging Tools

There are a variety of Debugging Tools that can be used as well (in C++). Generally, they work by keeping track of the memory state, computing the Happens-Before relationship to find data races.

Valgrind

Valgrind is a heavy-weight binary tool which has a larger overhead (about 20x) as it shadows the system memory. It tracks and stores information on the memory usage by the program during its execution, and detects incorrect memory access.

Helgrind

Helgrind is a Valgrind tool which instead helps detect data races and potential deadlocks. It does this by building a directed graph indicating the order in which locks have been acquired and, if there is a cycle in the graph, reports a potential deadlock.

To identify data races, it also builds a directed graph, but instead one which is acyclic and represents the collective Happens-Before dependencies.

It results in a much larger slowdown of about 100x.

Sanitizers

Sanitizers utilise a more compilation-based approach to detect issues, resulting in a lower overhead (of about 5-10x) as compared to Valgrind tools. There are a variety of Sanitizer tools, i.e. `AddressSanitizer`, `ThreadSanitizer`, `MemorySanitizer` and more.

An Action Plan for debugging (using these tools) is as follows:

1. Valgrind memcheck
2. ThreadSanitizer (TSan)
3. AddressSanitizer (ASan)
4. Helgrind

8.3 Formal Model Checking

Formal Model Checking refers to mathematically proving that the code is correct. This is one of the safest and most comprehensive methods of reasoning about and checking programs for correctness but is not widely adopted in the industry due to the high costs involved and the expertise required to perform such a method.

For example, the Formal Specification created for the program may be faulty, resulting in incorrect deductions as to the program correctness. Additionally, the specification created is usually difficult to prove that it is both correct, and that the code follows the specification correctly. This is not even accounting for the time required to come up with the complete specification in the first place.

Generally, the steps for Formal Model Checking is as follows:

1. Build the model using a special Domain Specific Language; Involves writing a format specification, usually in a new language, for the program
2. Check the model
    - Are all the constraints met?
    - Does anything unexpected happen?
    - Does it deadlock?

Formal Model Checking allows us to:

1. Check things make sense before starting any implementation which may be costly especially at a larger scale
2. Prove certain properties for existing code
3. Allow for aggressive optimization without compromising the correctness of the program

Some existing model checkers for concurrent programs are: TLA+, Coq Proof Assistant and Alloy

```
#include <cstdint>
#include <iostream>
#include <thread>
#include <shared_mutex>
#include "io.hpp"
#include "engine.hpp"

void Engine::accept(ClientConnection connection) {
  auto thread = std::thread(&Engine::connection_thread, this, std::move(connection));
  thread.detach();
}

void Engine::connection_thread(ClientConnection connection) {
  while(true) {
    ClientCommand input {};
    switch(connection.readInput(input)) {
      case ReadResult::Error: SyncCerr {} << "Error reading input" << std::endl;
      case ReadResult::EndOfFile: return;
      case ReadResult::Success: break;
    }

    if (input.type == input_buy) {
        beforeExecute(input);

        std::shared_lock lock(sellMut);

        uint32_t current_count = input.count;
        if (sellOrders.contains(input.instrument)) {
            for (auto &[key, value] : sellOrders[input.instrument]) {
                std::scoped_lock<std::mutex> orderLock(value.mutex);

                if (key.price > input.price) {
                    break;
                }

                if (value.isDeleted) {
                    continue;
                }

                value.executionId += 1;

                if (value.count == current_count) {
                    Output::OrderExecuted(key.orderId, input.order_id, value.executionId,
    key.price, value.count, getCurrentTimestamp());
                    value.isDeleted = true;
                    current_count = 0;
                }
                else if (value.count > current_count) {
                    Output::OrderExecuted(key.orderId, input.order_id, value.executionId,
    key.price, current_count, getCurrentTimestamp());
                    value.count -= current_count;
                    current_count = 0;
                }
                else {
                    Output::OrderExecuted(key.orderId, input.order_id, value.executionId,
    key.price, value.count, getCurrentTimestamp());
                    current_count -= value.count;
                    value.isDeleted = true;
                }

                if (current_count == 0) {
                    break;
                }
            }
        }
        lock.unlock();

        if (current_count > 0) {
            std::unique_lock addLock(buyMut);

            if (!buyOrders.contains(input.instrument)) {
```

```
            buyOrders.insert({input.instrument, std::map<BuyOrderKey, Order>()});
        }

        auto output_time = getCurrentTimestamp();
        BuyOrderKey key = BuyOrderKey(input.order_id, input.type, input.price,
input.instrument, output_time);
        Order order = Order(0, current_count, false);
        buyOrders[input.instrument].insert({key, order});

        Output::OrderAdded(input.order_id, input.instrument, input.price,
current_count, false, output_time);

        addLock.unlock();
    }

    afterExecute(input);
}
else if (input.type == input_sell) {
    beforeExecute(input);

    std::shared_lock lock(buyMut);

    uint32_t current_count = input.count;
    if (buyOrders.contains(input.instrument)) {
        for (auto &[key, value] : buyOrders[input.instrument]) {
            std::scoped_lock<std::mutex> orderLock(value.mutex);

            if (key.price < input.price) {
                break;
            }

            if (value.isDeleted) {
                continue;
            }

            value.executionId += 1;

            if (value.count == current_count) {
                Output::OrderExecuted(key.orderId, input.order_id, value.executionId,
key.price, value.count, getCurrentTimestamp());
                value.isDeleted = true;
                current_count = 0;

            }
            else if (value.count > current_count) {
                Output::OrderExecuted(key.orderId, input.order_id, value.executionId,
key.price, current_count, getCurrentTimestamp());
                value.count -= current_count;
                current_count = 0;
            }
            else {
                Output::OrderExecuted(key.orderId, input.order_id, value.executionId,
key.price, value.count, getCurrentTimestamp());
                current_count -= value.count;
                value.isDeleted = true;
            }

            if (current_count == 0) {
                break;
            }
        }
    }
    lock.unlock();

    if (current_count > 0) {
        std::unique_lock addLock(sellMut);

        if (!sellOrders.contains(input.instrument)) {
            sellOrders.insert({input.instrument, std::map<SellOrderKey, Order>()});
        }
```

```
            auto output_time = getCurrentTimestamp();
            SellOrderKey key = SellOrderKey(input.order_id, input.type, input.price,
    input.instrument, output_time);
            Order order = Order(0, current_count, false);
            sellOrders[input.instrument].insert({key, order});

            Output::OrderAdded(input.order_id, input.instrument, input.price,
    current_count, true, output_time);

            addLock.unlock();
        }

        afterExecute(input);
    }
    else if (input.type == input_cancel) {
        bool found = false;

        std::shared_lock buyLock(buyMut);
        for (auto &[instrument, buyOrderPerInstrument] : buyOrders)
        {
            for (auto &[key, value] : buyOrderPerInstrument)
            {
                if (key.orderId == input.order_id)
                {
                    found = true;

                    std::lock_guard<std::mutex> orderGuard(value.mutex);
                    if (value.isDeleted)
                    {
                        Output::OrderDeleted(input.order_id, false, getCurrentTimestamp());
                    }
                    else
                    {
                        value.isDeleted = true;
                        Output::OrderDeleted(input.order_id, true, getCurrentTimestamp());
                    }

                    break;
                }
            }
        }
        buyLock.unlock();

        std::shared_lock sellLock(sellMut);
        for (auto &[instrument, sellOrderPerInstrument] : sellOrders)
        {
            for (auto &[key, value] : sellOrderPerInstrument)
            {
                if (key.orderId == input.order_id)
                {
                    found = true;

                    std::lock_guard<std::mutex> orderGuard(value.mutex);
                    if (value.isDeleted)
                    {
                        Output::OrderDeleted(input.order_id, false, getCurrentTimestamp());
                    }
                    else
                    {
                        value.isDeleted = true;
                        Output::OrderDeleted(input.order_id, true, getCurrentTimestamp());
                    }
                    break;
                }
            }
        }
        sellLock.unlock();

        if (!found)
        {
            Output::OrderDeleted(input.order_id, false, getCurrentTimestamp());
```

```
        }
    }
    else {
        SyncCerr {}
            << "Got order: " << static_cast<char>(input.type) << " " << input.instrument <<
    " x " << input.count << " @ "
            << input.price << " ID: " << input.order_id << std::endl;

    }

  }
}

void Engine::beforeExecute(ClientCommand input) {
    std::shared_lock phaseLock(phaseMut);

    if (phase.contains(input.instrument)) {
        phaseLock.unlock();
    } else {
        phaseLock.unlock();
        std::unique_lock uniqueLock(phaseMut);
        phase.insert({input.instrument, PhaseValue()});
    }

    phaseLock.lock();
    PhaseValue &phaseValue = phase[input.instrument];
    phaseLock.unlock();

    if (input.type == input_sell) {
        std::unique_lock uniqueLock(phaseValue.mut);
        while (phaseValue.val > 0) phaseValue.cond.wait(uniqueLock);
        phaseValue.val--;
    } else {
        std::unique_lock uniqueLock(phaseValue.mut);
        while (phaseValue.val < 0) phaseValue.cond.wait(uniqueLock);
        phaseValue.val++;
    }
}

void Engine::afterExecute(ClientCommand input) {
    std::shared_lock lock(phaseMut);
    PhaseValue &ins_val = phase[input.instrument];
    lock.unlock();

    if (input.type == input_sell) {
        std::unique_lock uniqueLock(ins_val.mut);
        ins_val.val++;
        if (ins_val.val <= 0) {
            uniqueLock.unlock();
            ins_val.cond.notify_all();
        }
    } else {
        std::unique_lock uniqueLock(ins_val.mut);
        ins_val.val--;
        if (ins_val.val >= 0) {
            uniqueLock.unlock();
            ins_val.cond.notify_all();
        }
    }
}
```

Listing 8.1: Assignment 1 - Engine.cpp

```
package main

import "C"
import (
  "container/heap"
  "context"
  "fmt"
  "io"
  "net"
  "os"
  "time"
)

type Engine struct {
  mainGoRoutine    MainGoRoutine
  channelSemaphore ChannelSemaphore
}

type MainGoRoutine struct {
  instrumentChans       map[string](chan input)
  done                  chan int
  idToInstrumentHashmap map[uint32]string
}

func (m *MainGoRoutine) checkInstrument(in input) {
  if in.orderType == inputBuy || in.orderType == inputSell {
    m.idToInstrumentHashmap[in.orderId] = in.instrument
  } else {
    in.instrument = m.idToInstrumentHashmap[in.orderId]
  }
  instrumentChan, exists := m.instrumentChans[in.instrument]
  if !exists {
    instrumentChan = make(chan input)
    go func() {
      sellOrderPQ := make(SellOrderPQ, 0)
      heap.Init(&sellOrderPQ)
      buyOrderPQ := make(BuyOrderPQ, 0)
      heap.Init(&buyOrderPQ)

      instrumentGoRoutine := InstrumentGoRoutine{
        sellOrders:  make(SellOrderPQ, 0),
        buyOrders:   make(BuyOrderPQ, 0),
        isCancelled: make(map[uint32]bool),
      }
      for {
        select {
        case input := <-instrumentChan:
          instrumentGoRoutine.processOrder(&input)
        case <-m.done:
          return
        }
      }
    }()
    m.instrumentChans[in.instrument] = instrumentChan
  }
  instrumentChan <- in
}

type ChannelSemaphore struct {
  ch chan int
}

func (e *Engine) acquire() {
  e.channelSemaphore.ch <- 1
}

func (e *Engine) release() {
  <-e.channelSemaphore.ch
  // check if its the last client. if it its, close the mainGoRoutine
  if len(e.channelSemaphore.ch) <= 0 {
    close(e.mainGoRoutine.done)
```

```
  }
}

type InstrumentGoRoutine struct {
  sellOrders  SellOrderPQ
  buyOrders   BuyOrderPQ
  isCancelled map[uint32]bool
}

// ---------------------------------------------------------------------

func (e *Engine) accept(ctx context.Context, conn net.Conn, mainChannel chan input) {
  go func() {
    <-ctx.Done()
    conn.Close()
  }()
  go handleConn(conn, mainChannel, e)
}

func handleConn(
  conn net.Conn,
  mainChannel chan input,
  e *Engine) {
  defer conn.Close()
  defer e.release()
  e.acquire()

  clientDone := make(chan int)
  for {
    in, err := readInput(conn)
    if err != nil {
      if err != io.EOF {
        _, _ = fmt.Fprintf(os.Stderr, "Error reading input: \%v\n", err)
      }
      return
    }
    in.clientDone = clientDone
    mainChannel <- in
    <-clientDone

  }
}

// Match, Resolve and Log
func (inst *InstrumentGoRoutine) processOrder(in *input) {
  if in.orderType == inputCancel {
    _, ok := inst.isCancelled[in.orderId]
    if ok {
      outputOrderDeleted(*in, false, GetCurrentTimestamp())
    } else {
      inst.isCancelled[in.orderId] = true
      outputOrderDeleted(*in, true, GetCurrentTimestamp())
    }
  } else if in.orderType == inputBuy {
    current_count := in.count
    for inst.sellOrders.Len() > 0 && current_count > 0 {
      item := heap.Pop(&inst.sellOrders)
      sellOrderItem, ok := item.(*SellOrderItem)
      if !ok {
        fmt.Fprintf(os.Stderr, "Error popping from SellOrderPQ")
      }

      resting := sellOrderItem.value

      if resting.price > in.price {
        curItem := SellOrderItem{
          value: resting,
        }
        heap.Push(&inst.sellOrders, &curItem)
        break
      }
```

```
      _, cancelled := inst.isCancelled[resting.orderId]
      if cancelled {
        continue
      }
      resting.executionId++

      if resting.count == current_count {
        outputOrderExecuted(
          resting.orderId,
          in.orderId,
          uint32(resting.executionId),
          resting.price,
          resting.count,
          GetCurrentTimestamp(),
        )
        inst.isCancelled[resting.orderId] = true
        inst.isCancelled[in.orderId] = true
        current_count = 0
      } else if resting.count > current_count {
        outputOrderExecuted(
          resting.orderId,
          in.orderId,
          uint32(resting.executionId),
          resting.price,
          current_count,
          GetCurrentTimestamp(),
        )
        resting.count -= current_count

        curItem := SellOrderItem{
          value: resting,
        }

        heap.Push(&inst.sellOrders, &curItem)
        inst.isCancelled[in.orderId] = true
        current_count = 0
      } else {
        outputOrderExecuted(
          resting.orderId,
          in.orderId,
          uint32(resting.executionId),
          resting.price,
          resting.count,
          GetCurrentTimestamp(),
        )
        current_count -= resting.count
        inst.isCancelled[resting.orderId] = true
      }
      if current_count == 0 {
        break
      }

    }
    if current_count > 0 {
      timeStamp := GetCurrentTimestamp()
      buyOrder := BuyOrder{
        orderType:    in.orderType,
        orderId:      in.orderId,
        price:        in.price,
        count:        current_count,
        instrument:   in.instrument,
        timestamp:    timeStamp,
        executionId: 0,
        isDeleted:    false,
      }
      curItem := BuyOrderItem{
        value: buyOrder,
      }
      heap.Push(&inst.buyOrders, &curItem)
      in.count = current_count
      outputOrderAdded(*in, timeStamp)
```

```
    }
  } else if in.orderType == inputSell {
    current_count := in.count
    for inst.buyOrders.Len() > 0 && current_count > 0 {
      item := heap.Pop(&inst.buyOrders)
      buyOrderItem, ok := item.(*BuyOrderItem)
      if !ok {
        fmt.Fprintf(os.Stderr, "Error popping from BuyOrderPQ")
      }

      resting := buyOrderItem.value

      if resting.price < in.price {
        curItem := BuyOrderItem{
          value: resting,
        }
        heap.Push(&inst.buyOrders, &curItem)
        break
      }
      _, cancelled := inst.isCancelled[resting.orderId]
      if cancelled {
        continue
      }
      resting.executionId++

      if resting.count == current_count {
        outputOrderExecuted(
          resting.orderId,
          in.orderId,
          uint32(resting.executionId),
          resting.price,
          resting.count,
          GetCurrentTimestamp(),
        )
        inst.isCancelled[resting.orderId] = true
        inst.isCancelled[in.orderId] = true
        current_count = 0
      } else if resting.count > current_count {
        outputOrderExecuted(
          resting.orderId,
          in.orderId,
          uint32(resting.executionId),
          resting.price,
          current_count,
          GetCurrentTimestamp(),
        )
        resting.count -= current_count

        curItem := BuyOrderItem{
          value: resting,
        }

        heap.Push(&inst.buyOrders, &curItem)
        inst.isCancelled[in.orderId] = true
        current_count = 0
      } else {
        outputOrderExecuted(
          resting.orderId,
          in.orderId,
          uint32(resting.executionId),
          resting.price,
          resting.count,
          GetCurrentTimestamp(),
        )
        current_count -= resting.count
        inst.isCancelled[resting.orderId] = true
      }
      if current_count == 0 {
        break
      }
    }
```

```
    if current_count > 0 {
      timeStamp := GetCurrentTimestamp()
      order := SellOrder{
        orderType:   in.orderType,
        orderId:     in.orderId,
        price:       in.price,
        count:       current_count,
        instrument:  in.instrument,
        timestamp:   timeStamp,
        executionId: 0,
        isDeleted:   false,
      }

      curItem := SellOrderItem{
        value: order,
      }

      heap.Push(&inst.sellOrders, &curItem)
      in.count = current_count
      outputOrderAdded(*in, timeStamp)
    }

  }

  in.clientDone <- 1
}

func GetCurrentTimestamp() int64 {
  return time.Now().UnixNano()
}
```

Listing 8.2: Assignment 2 - Engine.go

```rust
use std::error::Error;
use std::sync::Arc;
use std::sync::mpsc;

use tokio::io::{AsyncBufReadExt, AsyncWriteExt, BufReader};
use tokio::net::{TcpListener, TcpStream};
use tokio::sync::Semaphore;

use crate::task::{Task, TaskType};

pub trait ServerTrait {
    async fn start_server(
        &self,
        address: String,
        tx: mpsc::Sender<Result<(), Box<dyn Error + Send>>>,
    ) -> std::io::Result<()>;
}

pub struct Server;

impl ServerTrait for Server {
    async fn start_server(
        &self,
        address: String,
        tx: mpsc::Sender<Result<(), Box<dyn Error + Send>>>,
    ) -> std::io::Result<()> {
        println!("Starting the server");
        let listener_res = TcpListener::bind(address).await;

        match listener_res {
            Ok(listener) => {
                tx.send(Ok(())).unwrap();
                let task_semaphore = Arc::new(Semaphore::new(40));

                loop {
                    match listener.accept().await {
                        Ok((stream, addr)) => {
                            let sem_clone = task_semaphore.clone();
                            // spawn new thread
                            tokio::spawn(async {Self::handle_connection(stream,
    sem_clone).await;});
                        }
                        Err(e) => {
                            eprintln!("Error accepting connection: {}", e);
                        }
                    }
                }
            },
            Err(e) => {
                println!("here {}", e);
                tx.send(Err(Box::new(e))).unwrap()
            }
        }
        Ok(())
    }
}

impl Server {
    async fn handle_connection(mut stream: TcpStream, task_semaphore: Arc<Semaphore>) ->
    std::io::Result<()> {
        let mut buf_reader = BufReader::new(stream);

        loop {
            let mut line = String::new();
            let mut buf: Vec<u8> = Vec::new();
            match buf_reader.read_until(b'\n', &mut buf).await {
                Ok(0) => {
                    return Ok(());
                }
                Ok(_) => {
                    let line = String::from_utf8(buf).unwrap();
```

```
                    let sem_clone = task_semaphore.clone();
                    let task_response = tokio::spawn(async move {
                        Self::get_task_value(line, sem_clone).await
                    });
                    if let Ok(Some(r)) = task_response.await {
                        buf_reader.get_mut().write_all(&[r]).await?;
                    }
                }
                Err(e) => {
                    eprintln!("Unable to get command due to: {}", e);
                }
            }
        }
    }

    async fn get_task_value(buf: String, task_semaphore: Arc<Semaphore>) -> Option<u8> {
        async fn try_parse(buf: String, task_semaphore: Arc<Semaphore>) -> Result<u8,
    Box<dyn std::error::Error>> {
            let numbers: Vec<&str> = buf.trim().split(':').collect();
            let task_type = numbers.first().unwrap().parse::<u8>()?;
            let seed = numbers.last().unwrap().parse::<u64>()?;

            match TaskType::from_u8(task_type).unwrap() {
                TaskType::CpuIntensiveTask => {
                    let permit = Arc::clone(&task_semaphore).acquire_owned().await.unwrap();
                    // execute async
                    let result = Task::execute_async(task_type, seed).await;
                    drop(permit);
                    Ok(result)
                },
                TaskType::IOIntensiveTask => {
                    // execute async
                    let result = Task::execute_async(task_type, seed).await;
                    Ok(result)
                }
            }
        }

        match try_parse(buf, task_semaphore).await {
            Ok(r) => Some(r),
            Err(_) => None
        }
    }
}
```

Listing 8.3: Assignment 3 - Server.rs