University of Warsaw
Faculty of Mathematics, Informatics and Mechanics

Natalia Kucharczuk
Student no. 406686

Michał Radwański
Student no. 395415

Alicja Ziarko
Student no. 406665

Antoni Żewierżejew
Student no. 406582

Deadlock detection in Seastar applications

Bachelor's thesis
in COMPUTER SCIENCE

Supervisor:
dr Janina Mincer-Daszkiewicz
Institute of Informatics

Warsaw, June 2021

Abstract

Seastar is a C++ framework based on promises and futures. It enables writing asynchronous and highly efficient applications and implements custom user-space scheduler and threading primitives.

However, Seastar is not resistant to concurrency bugs such as deadlocks which gives a motivation for adding a deadlock detection feature. The goal of this thesis is to implement that functionality. As of today there exist multiple deadlock detection algorithms on which we could base. In this thesis, we investigate examples of such algorithms together with some modifications and then decide which are the most applicable for Seastar. Moreover, we evaluate the efficiency and accuracy of the implemented solutions.

Keywords

concurrent programming, deadlock, deadlock detection, semaphore, future, promise, Seastar, wait-for graphs, happens-before relation

Thesis domain (Socrates-Erasmus subject area codes)

11.3 Informatics, Computer Science

Subject classification

Computing methodologies - Concurrent algorithms

Tytuł pracy w języku polskim

Wykrywanie zakleszczeń w Seastarze

Contents

Introduction

Mistakes in programming are very common and can result in a failure as well as in an outage and loss of efficiency. Moreover, there are varieties of mistakes that are not easily found. One kind of them are bugs causing a decline in program performance (or stopping it altogether), without necessarily affecting the results. Among such bugs are deadlocks which are the focus of this thesis.

There are quite a few potential deadlock detection approaches. These include static analysis, model checking, dynamic analysis and run time monitoring. They can also be combined. Choosing the best combination depends on the implementation of synchronisation mechanisms. Since our work concerns Seastar, we focus on its concurrency model. Seastar is based on futures, promises and continuations. Because of that, there are no explicit threads as usually computations consist of chains of small tasks. That is a complication — most existing algorithms have to be properly adapted first. We also have to take into consideration a share-nothing approach to memory in Seastar, which means that each of the physical threads has separate memory resources assigned.

Deadlock detection can be approached in two ways. One way is to detect deadlocks that happened and give the programmer some feedback about where they occurred. The other, more difficult one, is to find a deadlock that could have happened if the tasks had interleaved in a different order. There are some algorithms for that, but their focus is most often on finding a deadlock that happened among locks.

In this thesis, we implement a feature in Seastar that detects potential deadlocks occurring between operations on semaphores. Moreover, we develop some tools which can support programmers in finding the source of the deadlock.

In the next chapters, we give some background about Seastar. We describe the considered algorithms and our modifications made in order to adjust them to Seastar. We also present challenges we encountered and implementation choices. In the end, we validate the developed tool and evaluate its performance.

Chapter 1

Preliminaries

In this chapter, we provide the necessary background and definitions concerning both concurrent programming and the Seastar framework.

One of the ideas of concurrent programming is to shorten the time of program execution by running some of its multiple tasks in parallel. In many cases, this approach requires synchronisation and communication between the concurrent tasks. Most of the programming languages and libraries have built-in synchronisation mechanisms which may also be called synchronisation primitives. In particular, Seastar provides some of the standard synchronisation primitives [17], such as:

- `rwlock`, used for preventing simultaneous reading and writing;
- `shared mutex`, which is basically a semaphore with one permit;
- `semaphore`, which stores permits that can be acquired by using wait method; the semaphore can also be given a permit by using the signal method;
- `condition variable`, that blocks execution until some condition is satisfied (implemented using semaphore);
- `spinlock`, that waits in a loop until some boolean condition is satisfied;
- `pipe`, used for transferring data between two endpoints — the first one producing it and the second one consuming;
- `submit to`, that makes it possible to submit functions for execution to another core;
- `gate`, which is used to stop new requests from coming in and checking if the existing ones are complete, it is needed when there is a service running and it needs to be shut down.

The synchronisation between tasks may easily lead to bugs. One of them is deadlock which is the main focus of this thesis. **Deadlock** in the program occurs if there is a group of processes in which each member waits for another one, or for itself, to take some action. For example, process A waits for the semaphore permits which are currently acquired by process B. At the same time, process B waits for the semaphore permits acquired by process A. As a result, none of the processes A and B can finish their work and they are deadlocked. As one may expect, this phenomenon is highly undesirable as it leads to program stalls, high latency, or other negative effects. This definition shall be adapted to Seastar needs, which, prominently, doesn't use the abstraction of processes.

Seastar is a C++ framework that provides an asynchronous interface in otherwise thread-based language. It is based on the assumption that by default data should not be shared between different CPU cores. Instead, Seastar supports multicore servers by the use of sharding. Each logical core runs a separate event loop (**reactor**), with its own memory allocator, TCP/IP stack, and other services. **Shards** communicate by explicit message passing, rather than using locks and condition variables, contrary to what is common in traditional threaded programming.

The crucial concepts that are used by Seastar are promises and futures. A **future** is an object that represents a value that will be provided at some point in the future (or is already available, but this distinction is abstracted out by implementation). A **promise** is an object that can be used to give a future some value. This is similar to a queue with producer (promise) and consumer (future) which can take in total one unit of resource. Each Seastar promise object has methods `get_future()` and `set_value(value)` and they are used in the following way:

```
seastar::promise<int> pr;
    seastar::future<int> fut = pr.get_future();
    assert(!fut.available());
    pr.set_value(4);
    assert(fut.available());
    // This is valid only if the future is available.
    assert(fut.get() == 4);
    // This however is invalid - the result of a future
    // can be accessed only once.
    // fut.get();
```

The futures are non-copyable objects and use the C++ move semantics.

Seastar scheduler can be told to run a function after a successful computation of a value (materializing of a future). This is called a **continuation** and the return value of such a continuation is another future. So for example, there is a function

```
    seastar::future<> seastar::sleep(time)
```

which is a future that does not hold any data. This can be chained:

```
    seastar::future<int> some_future = seastar::sleep(100ms).then([] {
        std::cout << "We waited 100 ms.";
        return 5;
    });
```

The type of this expression is however `seastar::future<int>`. Any chain of tasks, that are created with help of continuations is called a **fiber**.

Seastar in itself provides a cooperative scheduler. Creating a future object puts the appropriate task onto the task queue (or on the IO completion queue, if that is applicable), however, that task will not be scheduled right away — this will happen only after the currently running task has exited. Seastar guarantees that each task that has been scheduled from a single CPU core will be run on the same CPU core. This provides immediate benefits. Unless some data is processed on many cores (their tasks are put on queues of separate reactors, which are the event loop of the shard), there is often no need to use traditional exclusion mechanisms, such as mutexes or atomic variables. One of the examples of the gains achieved is the built-in `seastar::shared_ptr<T>`, which behaves similarly to the C++ STL `std::shared_ptr<T>`, however, operates under the assumption that modifying it from different reactors is an undefined behaviour. This enables the use of a simple machine `long` for reference counting, without any further synchronisation of access.

The `future<>` returned by `seastar::sleep` is certainly not available right after creation (even if the sleep time is 0 seconds). It might happen however that the future value is already available. This behavior can be seen in the following example:

```
    seastar::promise<int> pr;
    seastar::future<int> fut = pr.get_future();
    pr.set_value(4);
    return fut.then([] (int x) {
        std::cout << "Got " << x << "\n";
    });
```

In that case, the resulting continuation is run straight away, without the need to wait in the queue. This might potentially introduce infinite loops, so after a limited number of continuations that were scheduled right away, the scheduling is transferred back to the waiting task queue.

Apart from having the option to wait just on one of the futures, there are several helper functions included in Seastar, for example:

- `seastar::repeat` — executes a given lambda a specified number of times,
- `seastar::do_for_each` — is a classical range-based for loop,
- `seastar::parallel_for_each` — schedules the lambda for each element from the range given by iterators, thus not giving any guarantees on the order of execution,
- `seastar::when_all` — takes several future values and exits only after each of them is available.

Some of the mentioned functions are not implemented based on the public interface of promises and futures but are using the reactor methods directly.

As mentioned before, Seastar provides semaphores, which operate in FIFO manner (this is however detail of implementation). Their usage is based on futures as well. It is to be noted, that just as the `seastar::shared_ptr<T>`, Seastar's semaphores are not thread-safe and can be used just from one shard. An example of usage is shown below:

```
seastar::semaphore sem(1); // This is the initial count.
sem.wait(1).then([&sem] {
    sem.signal(1); // This does not return anything.
});
```

Chapter 2

Project motivation and scope

Seastar is not just a neat framework. The driving factor for creating Seastar was the development of Scylla [16], a fast, scalable NoSQL database, which to this day is the most prominent example of Seastar deployment. Databases naturally are highly concurrent, so it is no surprise that deadlocks occur from time to time. Big systems are especially prone to such non-obvious bugs. This work was done in cooperation with ScyllaDB, the parent company of Scylla. Therefore, many of our decisions were made while taking into consideration the best interests of ScyllaDB.

Seastar is a well-developed framework with many embedded features, hence it is not possible to support each use case in the time frame of this project. We decided to narrow the scope and focus on detecting deadlocks that happen due to misuse of semaphores. There are several reasons. First of all, semaphores are a bit similar to ubiquitous locks, which gives hope for related work in this area. Moreover, comparing with other mechanisms, semaphores are indeed often used as a method of preventing execution (a good sign that a deadlock might occur), contrary to gates, which are used for the finalisation of operations. Comparing to `sumbmit_to`, semaphore is used relatively often — this is due to Scyllas' shared-nothing approach (`submit_to` shall occur as rarely as possible). Condition variable in Seastar is implemented using semaphore. The other options are simply not used very often. Focusing once again on Scylla, we counted how often specific structures occurred in Scylla and within Seastar code itself, and found that the other synchronisation primitives are almost nonexistent:

- `spinlock` — exactly one usage, for delivery of SIGSEGV and SIGABRT signals (this is the only thread-safe structure),
- `pipe` — one usage, for log collection utility,
- `shared_mutex` — one usage, for guarding access to files on disk,
- `rwlock` — three usages, all non-remarkable.

On the other hand, there are more than 200 usages of semaphore, which makes this synchronisation primitive the obvious target for our work.

Chapter 3

Existing deadlock detection approaches

In Chapter 1, the definition of deadlock is given. It is also mentioned that one can try either to detect deadlocks that happened or such ones that might potentially happen. This second problem is clearly more difficult than the first one — if we can detect potential deadlocks we can also detect actual ones. In this chapter, we focus on potential deadlock detection, which is also the problem we try to solve.

There are at least two ways by which deadlock detection algorithms can be categorized — by the synchronisation primitives and by the way the analysis is performed.

In the first group, there are two main categories — dynamic analysis and static analysis. Dynamic analysis means searching for deadlock based on how the program executed, while static analysis means searching for deadlock based on the program code.

In the second group, there are many categories, but the most popular one is looking for deadlocks that occur due to misuse of locks (mutexes). One can also search for deadlocks involving semaphores or condition variables or any combination of (potentially other) synchronisation primitives.

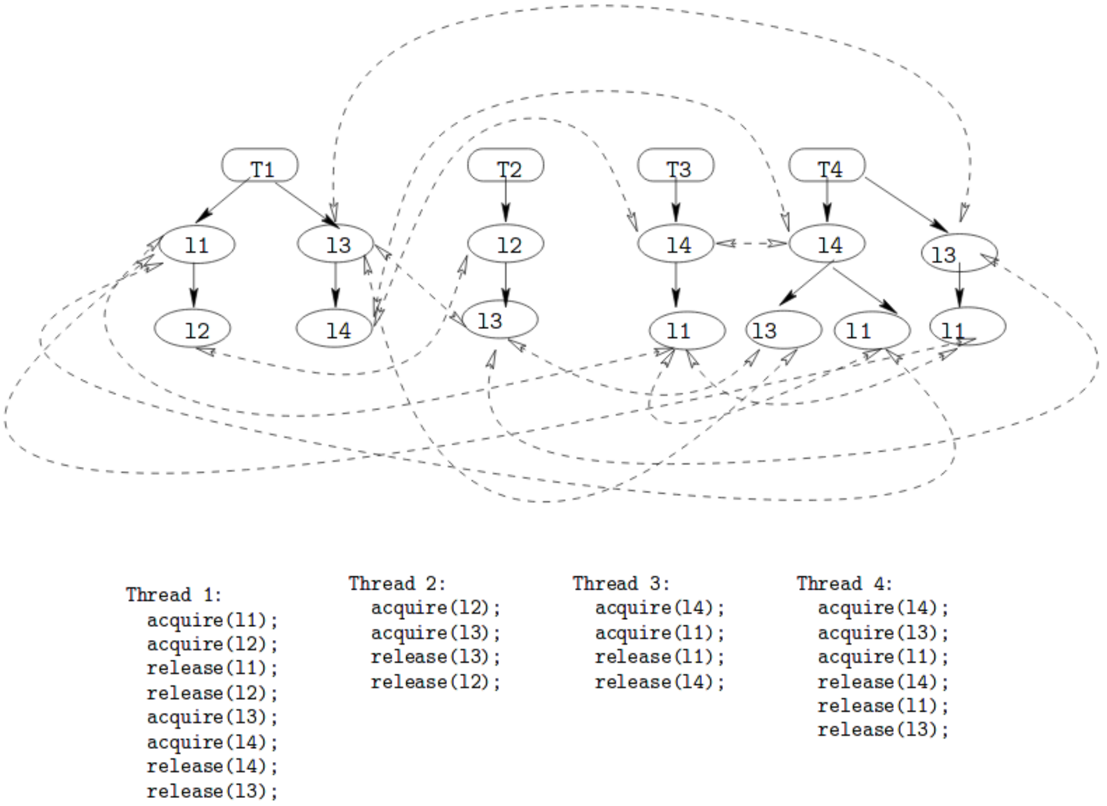
We will present three algorithms that show some of the very important ideas and are the most relevant to the Seastar framework. The static analysis in C++ would be difficult to implement and not time efficient (this is potentially an uncomputable problem). Usually, such tools require the programmer to manually work out how to attach certain annotations into their own code. This is a clear problem, from a standpoint of a framework programmer, as the code being run is not known. Another problem is also that there is almost no mature open-source tooling (lots of industrially provided software for model checking is either commercial or at best freeware, which is a no-go for an open-source Apache-licensed project). This is why all of the algorithms in question are based on dynamic analysis. Two of them detect deadlocks between locks only, and the third one can be used for a wide range of synchronisation primitives.

A lock is a special case of a semaphore. It has the value of 0 or 1, and it can be released only by the same thread as the one which acquired it, which also holds the obligation to release it.

The first algorithm we consider is `lockdep` [11], the deadlock detection from the Linux kernel. It is aimed to be lightweight and as foolproof as possible but assumes a small number of locks. Basically, all the locks are divided into classes, so multiple instances of the same type of locks are considered the same. Then, while executing, the most important rule is that locks have to always be acquired in the same order. This means that locks create a hierarchical structure (an acyclic graph) and we can only acquire a new lock if it is lower in the hierarchy than all that are currently acquired. Since acquiring the same locks with enabled interrupt requests (irqs) and inside of irq would result in deadlock, there are also rules for reacquiring locks and acquiring locks in the context of irqs. These rules ensure that there is no cycle in the lock hierarchy itself, so there can never be a cycle of threads waiting for each other. Overall it provides robust deadlock protection but is pretty restrictive. Firstly it only works with (read-write) locks, secondly, it prevents multiple locking scenarios without actual deadlock.

Our second algorithm, presented by Agarwal, Wang and Stoller at [2], also works while the program is running but the way it works is a bit different. It also gives fewer false positives than the previous algorithm. The idea here is to build a tree of locks for each of the running threads. The tree is built in the following way:

1. The root of the tree is labelled with some identifier of the thread.
2. The root has one child for each lock that it acquired.
3. The node that responds to the lock l has a child responding to the lock l' if and only if l' was acquired why l was being held.
4. All the nodes at level one that have no child are deleted from the tree.



```
Thread 1:
  acquire(l1);
  acquire(l2);
  release(l1);
  release(l2);
  acquire(l3);
  acquire(l4);
  release(l4);
  release(l3);
```

```
Thread 2:
  acquire(l2);
  acquire(l3);
  release(l3);
  release(l2);
```

```
Thread 3:
  acquire(l4);
  acquire(l1);
  release(l1);
  release(l4);
```

```
Thread 4:
  acquire(l4);
  acquire(l3);
  acquire(l1);
  release(l4);
  release(l1);
  release(l3);
```

Figure 3.1: Constructed tree of lock dependency (upper part of the figure) and shortened execution of threads (lower part of the figure), from [3]

When we already have a tree for each thread (see figure (3.1)), we can build a run-time lock graph by adding edges between the nodes in different trees, which correspond to the same locks. After that step, we just need to search for a particular kind of cycle in the graph. That means a cycle satisfying the following conditions:

1. Denoting the cycle's vertices as $v_1, v_2, ..., v_n$, where $v_n = v_1$ there cannot exist indices i, j, k such that $i < j < k < n$ and v_j is in a different thread tree than v_i, but v_k is in the same one as v_j.
2. Denoting the cycle's vertices as $v_1, v_2, ..., v_n$, where $v_n = v_1$, there does not exist i such that $i, i+1, i+2$ are all from different trees.

This cycle can be found in polynomial complexity, using the algorithm described at [3].

The third approach, also described by Agarwal, Wang and Stoller at [3], can be used for a lot of synchronisation primitives. It uses happens-before orderings — a relation on the subset of all operations in a program, such that if operations a and b are in a happens-before relation, then a must happen before b does. For example for semaphores it can be defined in the following way: if a thread t is waiting on semaphore s, the operation unblocking s is in the happens-before relation with every operation in thread t after the current wait. So for example with code:

```
s1(3)
s2(5)
Thread 1:                          Thread 2:
s1.acquire(4)                      s2.acquire(2)
s2.acquire(3)                      s1.signal(1)
s2.signal(1)                       s2.acquire(3)
s1.signal(5)
```

The happens-before relation let us make the acyclic graph shown in Figure 3.2.

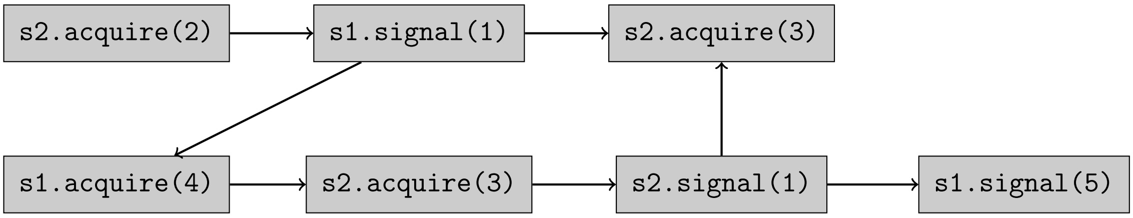


Figure 3.2: Happens-before graph.

When we have the graph we need to check all of its topological sorts for deadlock. Similarly, we can define the happens-before relation for any other synchronisation-primitive.

There is a lot of other work on deadlock detection in literature, although not much of it is directly applicable to Seastar. Programming language Go has its own static deadlock detection feature [13]. There are also multiple tools and mechanisms detecting deadlocks such as Pulse [10], UnDead [19] and Dreadlocks [9]. There is also an algorithm that is able to detect deadlocks in large-scale applications extremely efficiently [4] (unfortunately only between locks). Agrawal, Carey and DeWitt at [1] proposed some reasonable assumptions, under which deadlocks can be detected very cheaply. Much theoretical work on the subject of deadlock detection was made by Fajstrup and Raußen at [5]. Feitelson at [6] presented a solution that does not use any graph structures but processes' counters. Recently, Voss and Sarkar at [18] proposed an ownership semantics for promises, and provided a lock-free algorithm for runtime deadlock detection, which however doesn't take into considerations semaphores whatsoever. They also note, that the concept of a deadlock for promises isn't currently well-defined.

Chapter 4

Algorithm selection process

Investigation of the existing deadlock detection algorithms gave us many ideas on how the deadlock detection process can look like and what is worth a try. However, none of the algorithms could be directly implemented in the Seastar framework. The main reason for that is that it is not so easy to say who is the owner of the resource in Seastar programs. It is more complex than just acquiring resources and returning them, and many of the existing approaches are based on this assumption. We explain the passing of ownership in Seastar programs in detail further in this chapter and in Chapter 6.

The other reason for not settling on one particular deadlock detection algorithm is that most of them focus on a single synchronisation primitive. A useful deadlock detector should detect deadlocks that can happen using all synchronisation primitives and as our time might not allow us to implement deadlock detection for all primitives, we should at least build a tool that allows future developers to scale it easily.

Taking all of that into consideration, we decided to combine the gathered knowledge and create an algorithm as simple as possible, which could be easily upgraded later on. The majority of non-static deadlock detection algorithms were based on happens-before and wait-for graphs. They can be created for many synchronisation primitives and there are multiple algorithms that can be applied to them. Also, they store information that can be used in debugging messages such as where exactly the deadlock appeared and what led to it.

Our first approach was to create a wait-for graph but it turned out to be very challenging in Seastar. For instance, at the time of calling `wait` on a semaphore, the task that will be scheduled due to this operation hasn't yet been created. It became an obstacle that was very hard to work around and it drove us to search for another more suited approach.

In the end, we decided to create a graph based on the happens-before relation. Because of Seastar's continuations, explained in Chapter 1, the ownership of resource can be passed from one future to another or even multiple futures (by the use of for example `seastar::parallel_for_each`). This brings a lot of complications in tracking the order of events and it is explained in detail in Chapter 6. Therefore we decided to build a happens-before graph for Seastar in the following way. The vertices symbolise the core Seastar entities - promises, futures and tasks. We have an edge from vertex A to B if and only if A must have happened before B. The events we record are (primarily) one of the following:

- creating/destroying a semaphore,
- wait on a semaphore,
- signal on a semaphore,

- creating/destroying a task, promise, or future.

Gray at [8] noticed that deadlocks with semaphores are very unlikely to form cycles of length greater than two and that most deadlocks consist of only two deadlocked semaphores. Even though this paper was written a long time ago, we do believe it still should be true to some extent — meaning long cycles are not likely. Taking into account the possibility, that the way software is designed today differs from the old ways, in our algorithm we generate all subsets of semaphores of size at most p, where p is some small parameter given by the user. Then for each subset, we create a new happens-before graph consisting of events relevant to selected semaphores only. For example, assume that we have 3 fibers running the given Philosopher code as in the dining philosophers problem.

```
seastar::future<> Philosopher(int id, seastar::semaphore& left,
                              seastar::semaphore& right) {
        return left.wait(1).then([&left, &right, id] {
                return right.wait(1).then([&left, &right, id] {
                        std::cerr << "Philosopher eats." << std::endl;
                        return seastar::make_ready_future<>();
                });
        });
}
```

We have 3 semaphores `sem1, sem2, sem3` with one resource each. Philosopher number `id` gets `sem{id}` for its left semaphore and `sem{id+1 mod 3}` for its right semaphore. One of the possible happens-before graphs for this problem is shown in the Figure 4.1.

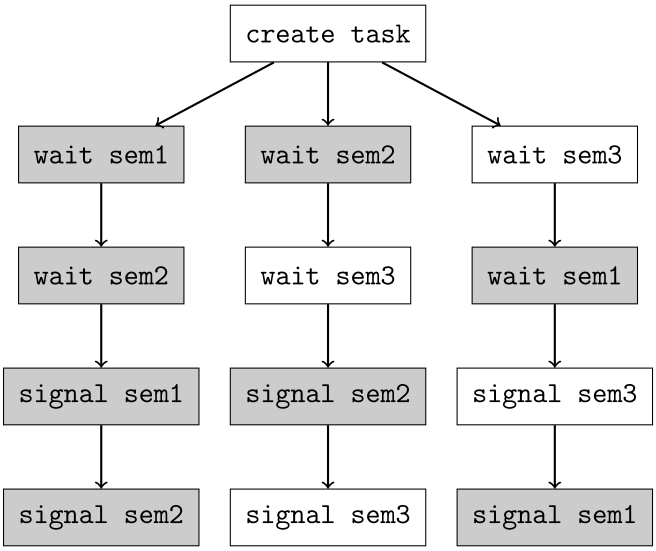


Figure 4.1: Happens-before graph for 3 Philosophers.

Now we can create a reduced graph for any subset of semaphores of size at most $p = 3$. Let us take a subset `{sem1, sem2}`. The reduced graph is shown in the Figure 4.2

This reduction minimizes the size of the obtained graph and preserves all relationships between semaphore events. In general, this reduction will significantly reduce the size of the graph, so we can look for deadlocks in those small graphs instead of the original ones.

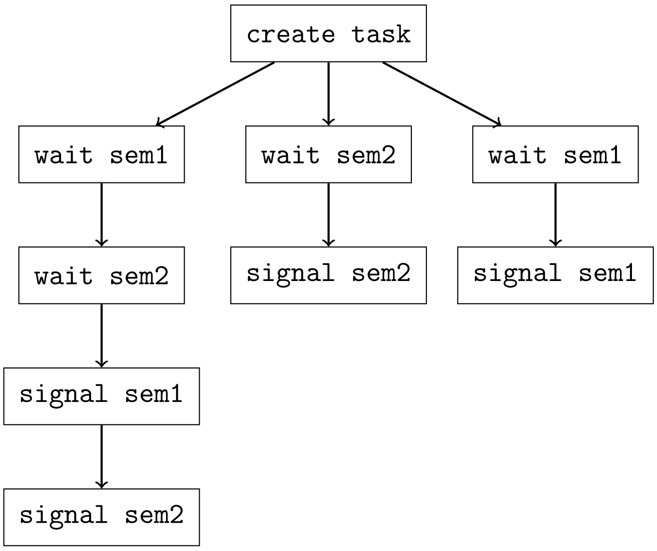


Figure 4.2: Reduced happens-before graph for 3 Philosophers and 2 semaphores.

This brings up the question of how do we look for potential deadlock in happens-before graph. Let us look into what this graph shows. It sets the order of some events but not all of them. For example, if event A happens before event B and event C but we have no information regarding the order of B and C themselves then it could be any of B and C happening first. Therefore we consider all possible orders of events and those are all topological sorts of a graph. We call each such order the execution chain since it is one of the possible ways the program can be executed. For each of these execution chains, we check if it leads to a deadlock, i.e. whether we encounter deadlock while simulating the program execution according to this executions chain. Going back to our example we can have multiple topological orders of the non-reduced graph. Two of them are shown in Figure 4.3. While the execution chain shown on the left side of the figure does not lead to a deadlock, the execution chain on the right does — when we do wait on all three semaphores, there is no further operation in this chain that can be executed. Therefore we predict a deadlock.

It is a brute force solution but we optimised it in multiple ways, which we describe in the next chapter.

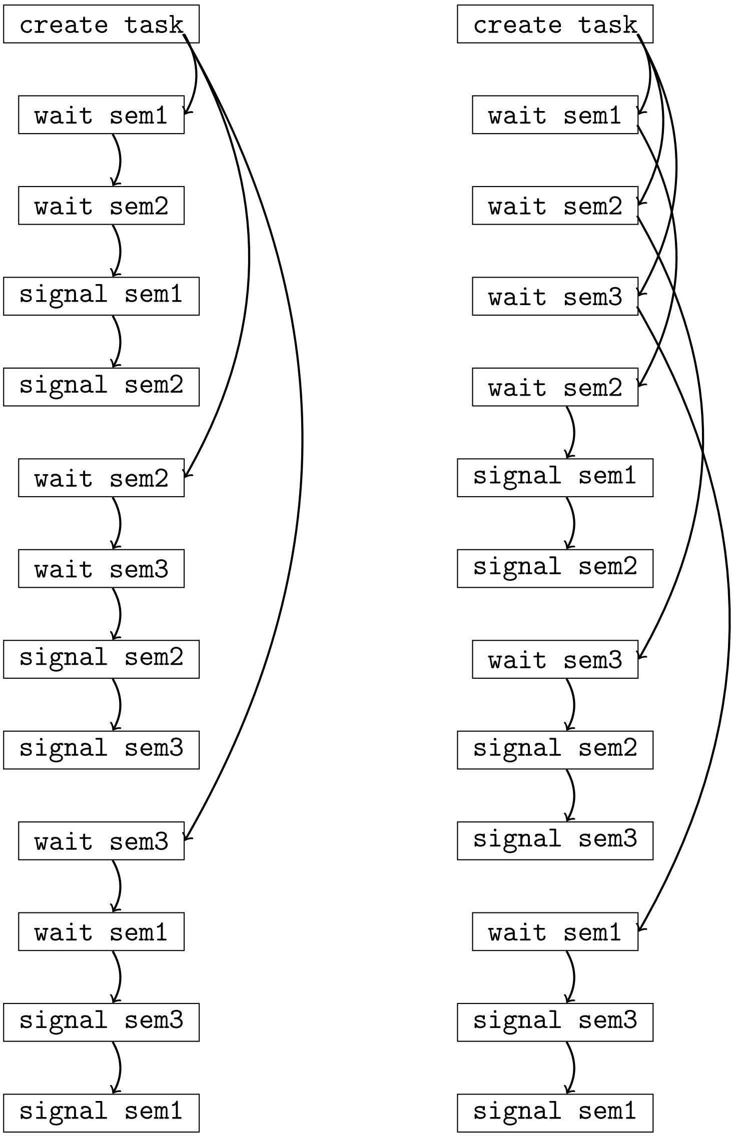


Figure 4.3: Two execution chains of the dining philosophers problem

Chapter 5

Optimisations

The algorithm as it is described in the previous chapter would not be efficient enough to search for deadlock in most of the problems. Therefore we need to optimize it. Observe, that there are some topological sorts that are not valid orders of execution. To see what we mean by that, let us focus again on the dining philosophers problem (for 3 of them). Any topological sort with the prefix in Figure 5.1 is not a valid order of execution, since all the semaphores have one permission at the beginning of execution, and in this topological sort, there are two waits on semaphore 1 before any signal on this semaphore.

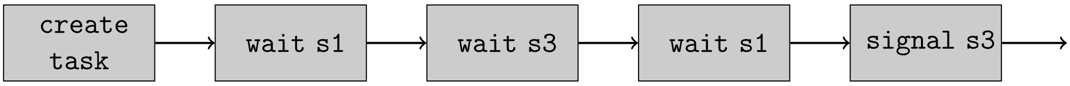


Figure 5.1: Invalid order of execution

We want to only check the topological sorts that represent valid orders of execution. Hence we perform backtracking with memoization. The pseudocode of this algorithm is as follows:

```
global variables
    $Safe = \emptyset$ - states that are safe
end global variables
procedure FindDeadlock($Graph$ - happens-before graph, $State$ - current execution state, $Ready$ - vertices which are ready to be executed)
    if $Ready$ is empty then
        return False
    end if
    if $Ready \in Safe$ then
        return False
    end if
    $CanProgress \leftarrow$False
    for $v \in Ready$ do
        if $v$ can be executed in the current $State$ then
            $TempState \leftarrow State$ with executed $v$
            $TempReady \leftarrow Ready/\{v\}$
            for $w \in Graph$ that became executable do
                $TempReady \leftarrow TempReady \cup \{w\}$
            end for
            if FindDeadlock(Graph, TempState, TempReady) then
```

```
                return True
            end if
            $CanProgress$ ←True
        end if
    end for
    if !$CanProgress$ then
        return True                                        ▷ This state is in deadlock.
    end if
    $Safe \leftarrow Safe \cup \{Ready\}$
    return False
end procedure
```

This is a simplified version of the algorithm, in which we return no information about the found deadlock.

To clarify, what is happening in this code:

- *State* is a state of the program (for example it has all the semaphores together with their queues, number of their permits, etc.),
- *Ready* is a set of operations that are ready to be executed, an operation can be one of `create task`, `wait` and `signal`,
- An operation can be executed if executing it won't cause any semaphore to go below zero,
- An operation a is ready to be executed if all the operations in graph *Graph* that must happen before a were executed.

We implemented this algorithm in C++. The tricky part was checking if *Ready* was in $Safe$ and inserting it — $Safe$ is a set of sets, hence we need an efficient way to:

- compare two sets from $Safe$,
- insert a set into $Safe$,
- insert into *Ready*.

For that reason we use our own set. We use hashes in order not to compare $Safe$ elements in linear time. Moreover in our implementation, inserting or erasing an element updates the set's hash in a constant time.

In the case where the sum of units from waits on the semaphore is smaller than the initial value of the semaphore, said semaphore can't participate in a deadlock. In the case of Scylla, this simple optimisation allows us to get rid of $\frac{3}{4}$ of all operations.

Another observation is that we only need to search for deadlock in certain semaphore groups.

Theorem: We say that a set of semaphores A makes a deadlock if after changing all the operations that are not on the semaphores from A to nops in the happens-before graph a potential deadlock still exists in such a graph. Let now graph G be a graph with vertices being semaphores, in which there exists an edge from semaphore v to semaphore w if and only if there exists a wait on semaphore v in happens-before relation with a signal on semaphore w. Then if a set of semaphores S makes a deadlock and no proper subset of S does, then S is a strongly connected component of G.

Lemma: If the state in which the deadlock happened doesn't contain a wait on semaphore i, that is ready to be executed but there aren't enough permits on the semaphore i then this semaphore will not participate in the deadlock.

Lemma Proof: Quick argument is that if we change the operations on semaphore i to nops (which is equivalent to ignoring a semaphore) then in the same (still reachable) state there still would be no operations that can be executed.

Theorem Proof: We will prove this by contradiction. Let's assume that we have a set of semaphores S that is not a strongly connected subgraph of G but it makes a deadlock. Let's look at the strongly connected components of S. It is known that strongly connected components of a graph make a DAG and we can take the "last" or the "smallest" vertex in this DAG, meaning one that doesn't have any outgoing edges. Let's take this strongly connected component and call it S'. From our assumptions, we know it's strictly contained in S and nonempty. Let's ignore this subset meaning we look at the set $S - S'$. Equivalently we change the operations on semaphores from S' to nops. Some operations would now become possible to execute, but they are in a happens-before relation with a vertex previously signifying a wait on one of the semaphores in S'. Thus because there are no edges in S coming out from S' there is no signal operation on semaphores from $S - S'$ that could become ready to execute, but after changing the operations to nops won't. Hence the permit count on semaphores from $S - S'$ will stay the same or be lower. But we had a state in which there exist wait operations on those semaphores that can't be executed because of not having enough permits, so in any state which comes after that, they still can't be executed for the same reason. So we now see that the semaphores from $S - S'$ make a deadlock and that is a contradiction. □

Having this result, in our program we search for all strongly connected components in graph G and only run detection on such groups. We search for them by finding non-strict cycles meaning cycles with repeated vertices.

Another optimisation is running the detection on different semaphore groups in parallel — the subproblems don't depend on each other.

Chapter 6

Implementation details — tracing

In this chapter, we explain the technical details of our solution.

In order to build the execution graph, we decided to modify Seastar, so that it includes our tracing stubs inside operations that we need to trace. While parts of the task could have been achieved with instrumentation frameworks, like Frida [15], there are many problems (such as functions being inline), which would lead to complicated and fragile implementation. Therefore, a more traditional yet more reliable approach is taken.

6.1. Happens-before relation in Seastar

The first part is tracing data. Since Seastar is asynchronous with promises and futures, deciding what to trace is not trivial. In a standard and synchronous C++ program, the activity of each thread is enough to determine happens-before relationship between events. But in the case of Seastar, there is nothing we could call a continuous thread. Recall that continuation is a subtype of task. We have a multitude of tasks synchronised with futures and promises. These three will be called vertices. We trace happens-before relationship between vertices, most notably:

- future → task, when a task is a continuation of a future;
- task → promise, when a task completes this promise;
- promise → task when a promise causes a task to start;
- promise → future, when a promise makes a future ready.

Some of those relationships can be traced by simple instrumentation, knowing only the parameters some functions were called with (for instance, a continuation created by `then` call). However, for tracing task-related info, we need to remember the task that is currently running, which is not a part of function parameters.

There are some additional challenges to that. Seastar is heavily optimised so in some places there are "shortcuts" that improve performance. For example let us look at this snippet:

```
semaphore.wait(1).then([&semaphore] {
    std::cout << "Critical section" << std::endl;
    semaphore.signal(1);
});
```

We would expect to observe the following chain:
`promise(wait)->task(lambda#1)->promise(then)->future(then)`
But if the semaphore has permits then `wait` will return a ready future. The subsequent `then` will not create an actual task but execute function associated with continuation immediately. That means that the code can be executed for task or pseudo-task which is an available future. That is why we update the current task in a stack-like manner.

Similar optimisations appear in several places, and oftentimes we are faced with C++'s move constructors, which leave some questions about what is the expected behaviour. For instance, can we assume that a moved future object is never accessed afterwards? This is tricky since we identify each such object by its address in memory. In this particular case, it turns out that they are invalid after the move. Nonetheless, Seastar's implementation has many subtle details that require careful thought. The modifications we made explain why specific code locations shall generate event or why not.

Tracing semaphore operations is a matter of simple instrumentation as well.

In order to better understand the gathered data, we decided to enrich our logs with debugging information, as otherwise, all we know is the memory address of some objects, without any hope for knowing which part of our code is responsible for it. Since continuations are common in Seastar applications, and C++ standard guarantees that the type of each anonymous function is distinct, the most important piece of debugging info that we are dumping are the types of functions passed to Seastar functions, mainly the type of continuation created by `then`.

6.2. Architecture of the tracer

The next question to answer is how to log large amounts of data efficiently. We have to trace small but really frequent events (there can be multiple generated events per single function call).

Let's first look at optimising the data size itself. We used Google's Protocol Buffers (protobuf) [14] as it is quite space-efficient and can be easily constructed in place (uses a minimal number of heap allocations). That allows for almost zero-copy tracing, with protobuf serializing directly to the output buffer. Another advantage is a highly optimized implementation, with parsers in multiple programming languages, which comes in handy while actually processing the logs we gather. If each event that holds some debugging information describing the type of a function is passed as-is, this quickly becomes a major bottleneck in tracing, as the types, due to the C++ templates, are often very long. To optimize this case, we notice that usually, lots of the debugging strings repeat multiple times (they are embedded in binary and not constructed in run time). Therefore, we map them to small identifiers and thus reduce footprint.

To better illustrate the used format, let us take a look at the protocol definition:

```
message deadlock_trace {
  enum event_type {
    EDGE = 0; // pre, vertex(post), value(speculative)
    VERTEX_CTOR = 1; // vertex
    VERTEX_DTOR = 2; // vertex
    VERTEX_MOVE = 3; // pre, vertex(post)
    SEM_CTOR = 4; // sem, value(count)
    SEM_DTOR = 5; // sem, value(count)
    SEM_MOVE = 6; // pre, sem
    STRING_ID = 7; // value(id), extra(string)
    FUNC_TYPE = 8; // vertex, value(func_type id), extra(file:line)
    SEM_SIGNAL = 9; // sem, value(count), vertex
    SEM_WAIT = 10; // sem, pre, vertex(post), value(count)
    SEM_WAIT_CMPL = 11; // sem, vertex(post)
  }
  required event_type type = 1;
  required uint64 timestamp = 2;
  optional uint64 value = 3;
  message typed_address {
    required uint64 address = 1;
    optional uint64 type_id = 2;
  }
  optional typed_address vertex = 4;
  optional typed_address pre = 5;
  optional uint64 sem = 6;
  optional string extra = 7;
}
```

The second thing is the tracing itself. To bring flexibility for the parser of our dumped format, we decided to simply write to files on storage. Due to the share-nothing approach of Seastar, we could not use just one global file for all the tracing data, as the synchronisation between threads would cause massive overheads. There also is not any way to make it more granular than per one shard.

That is why we use one file per shard. Outputs from each shard are combined by timestamps of each event, which is collected with nanosecond precision, to avoid invalid ordering in our logs. The biggest challenge is using Seastar's DMA output. Writes to a single file cannot be parallel so it is required, that at each moment, at most one task is writing. To assure this in the Seastar initialisation process, we start a looping task that writes data (which we hold in an internal buffer) to disk. We do not want to stop execution while the output operation is ongoing, but the output buffer cannot be deallocated or overwritten while an operation is ongoing. Our solution is to allocate two buffers (`std::vectors`) of data. One is written to and from the other one operation can be ongoing. In the output loop, the buffers are just swapped (there cannot be parallel append, because of Seastar cooperative scheduling). With this, we achieve (amortized) zero-copy and zero allocation output.

There were some challenges that we needed to tackle to receive data in full, without omissions and in the correct order. One of them is that if tracing on each of the shards finishes without proper synchronisation, we risk seeing references to a vertex (pointer) on shard 2, which was created on shard 1, only after tracing on shard 1 has been already disabled. This is something which cannot be easily accommodated from the parser program (ignoring such events implies that we lose much credibility in our modifications — after all, assuming the tracing starts and finishes at the same exact time on each of the shards, each edge from a previously not seen vertex signals for us that there is a bug). To synchronize these, we use an atomic variable. There is a similar problem for the start of tracing.

Chapter 7

Implementation details — deadlock detection

For our integration and optimisation tests, we used Scylla database, one of the biggest applications written in Seastar.

After defining and implementing the happens-before relation (and the rest of the tracing infrastructure) we started the tests. The first challenge was the tracing of data. Solutions we found are discussed in chapter 6. They allowed us to achieve less than 10% overhead when writing to ramdisk. When writing to the same hard drive that the database uses, performance degrades much more because drive's bandwidth is highly contested. In our tests we have up to 100 MiB/s data which can easily impact slower drives.

The next task is to parse the traces. We check the correctness of the traces when possible, most significantly that the constructors and destructors are paired correctly, which means that there aren't two constructors or destructors for the same address in a row. We also check that every referenced vertex and semaphore has been constructed. As explained, we combine multiple traces using the timestamp (nanosecond precision) of each event and in our tests, we never got a trace in which the order by timestamps was different from the actual one.

As we said, the traces can be rather big (even 2GB for 10s) so a parser that reads all data to memory would require a lot of it, or wouldn't succeed at all. For those reasons, we decide to read data using input iterators that only store a constant amount of data.

After parsing the traces we have to create the happens-before graph. For the effectiveness of later algorithms, we decide to give each vertex or semaphore a unique integer id and use that rather than smart pointers. One challenge here is that in certain cases we have cycles in created graph for example in following case:

```
future<> computation1 = foo();
if (computation1.available()) {
    // In this example code takes this branch.
    return bar();
}
return computation1.then([] {
    return bar();
});
```

This would give the graph seen in Figure 7.1.

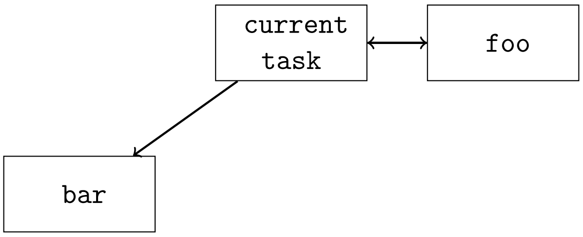


Figure 7.1: Invalid order of execution

We cannot use that as a happens-before relation because it has cycles. Because of that, we remove the cycles by adding timing information to the graph. We interpret a vertex as multiple connected vertices where each represents one part in time. So in the following scenario, we actually would receive graph seen in Figure 7.2.

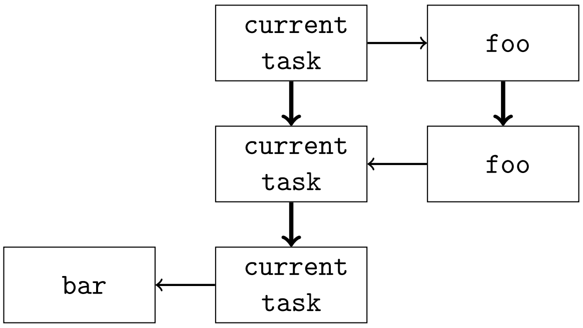


Figure 7.2: Invalid order of execution

However in this scenario, it's unnecessarily big, so we only add a new vertex part when it's needed. This is when one of following conditions is met:

- adding incoming edge to a vertex with semaphore operation,
- adding incoming edge to a vertex with outgoing edges.

The actual small form can be seen in Figure 7.3.

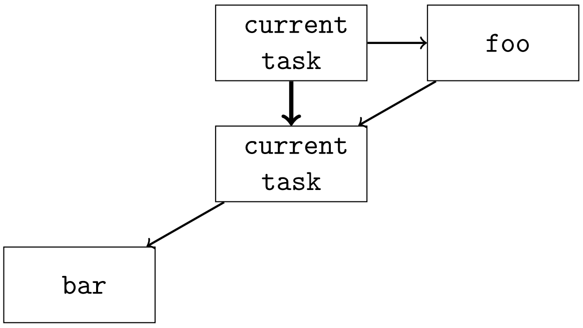


Figure 7.3: Invalid order of execution

But the total number of vertices in such a graph could possibly be not much smaller than the total number of traced events, which would cause the graph to be big and take up a lot of memory. As discussed, there is no need to remember vertices without semaphore operation as long as we remember happens-before relations implied by its edges. Therefore, while parsing, we delete these vertices. This means that the memory used while parsing is the number of actually existing vertices at a given point in traces plus the number of semaphore operations. This is a huge reduction because the total number of vertices can easily take up more memory than the traced program had available.

When deleting a vertex in the graph we have to restore happens-before relation to be equivalent. This can be easily done in the following way when deleting vertex v:

```
**for** $u_1$ such exists edge $u_1 \to v$ **do**
    **for** $u_2$ such exists edge $v \to u_2$ **do**
        $add\_edge(u_1, u_2)$
    **end for**
**end for**
```

This has a small problem — it generates a lot of edges that are transitive closures of existing edges, growing the graph with unnecessary information. Erasing all such edges wouldn't be possible without increasing the complexity of the algorithm. The following change reduces a lot of these edges, while still being efficient:

```
**for** $u_1$ such exists edge $u_1 \to v$ **do**
    **for** $u_2$ such exists edge $v \to u_2$ **do**
        **if** there isn't $w \neq v$ such edges $u_1 \to w$ and $w \to u_2$ exist **then**
            $add\_edge(u_1, u_2)$
        **end if**
    **end for**
**end for**
```

Another challenge is the visualization of data. Big graphs are difficult to represent in text form. For each vertex, we prepare its attributes in a format understood by graphviz [7] — we describe the look and content of the vertex. This is where the type names come in handy. Then we can dump it at any stage of processing and have a graphical representation. The reduced graph for the dining philosophers problem can be seen in Figure 7.4.

Next task is finding the semaphore groups. We achieve this with the following algorithm:

```
$Groups \leftarrow \emptyset$
$Checked \leftarrow \emptyset$
$Queue \leftarrow \{(\{v\}, v) : v \in E(G)\}$
for $Queue$ is not empty do
    $(Set, v) \leftarrow Queue.pop()$
    if $(Set, v)$ in $Checked$ then
        continue
    end if
    $Checked \leftarrow (Checked \cup \{(Set, v)\})$
    for $u$ such exists edge $v \rightarrow u$ do
        if $v = u$ then
            $Groups \leftarrow (Groups \cup \{Set\})$
        end if
        $NewSet \leftarrow (Set \cup \{u\})$
        $Queue.push((NewSet, u))$
    end for
end for
```

Finally, to represent deadlock we generate a graph with overlayed info which can be seen in Figure 7.5.

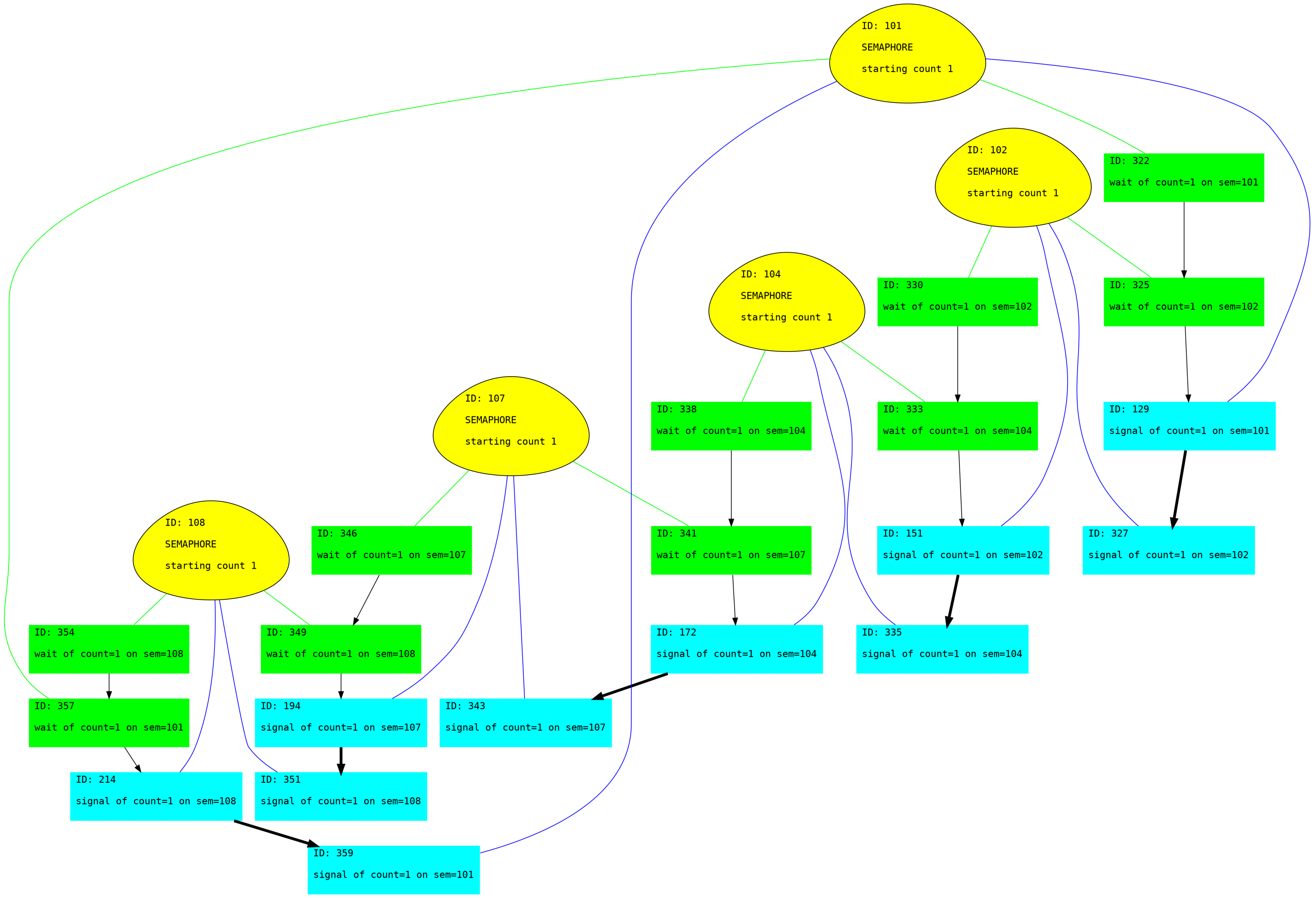

ID: 101
SEMAPHORE
starting count 1
ID: 102
SEMAPHORE
starting count 1
ID: 322
wait of count=1 on sem=101
ID: 104
SEMAPHORE
starting count 1
ID: 330
wait of count=1 on sem=102
ID: 325
wait of count=1 on sem=102
ID: 107
SEMAPHORE
starting count 1
ID: 338
wait of count=1 on sem=104
ID: 333
wait of count=1 on sem=104
ID: 129
signal of count=1 on sem=101
ID: 108
SEMAPHORE
starting count 1
ID: 346
wait of count=1 on sem=107
ID: 341
wait of count=1 on sem=107
ID: 151
signal of count=1 on sem=102
ID: 327
signal of count=1 on sem=102
ID: 354
wait of count=1 on sem=108
ID: 349
wait of count=1 on sem=108
ID: 172
signal of count=1 on sem=104
ID: 335
signal of count=1 on sem=104
ID: 357
wait of count=1 on sem=101
ID: 194
signal of count=1 on sem=107
ID: 343
signal of count=1 on sem=107
ID: 214
signal of count=1 on sem=108
ID: 351
signal of count=1 on sem=108
ID: 359
signal of count=1 on sem=101

Figure 7.4: Graph constructed from traces from the dining philosophers problem (5 philosophers) with reduction (yellow nodes represent semaphores)

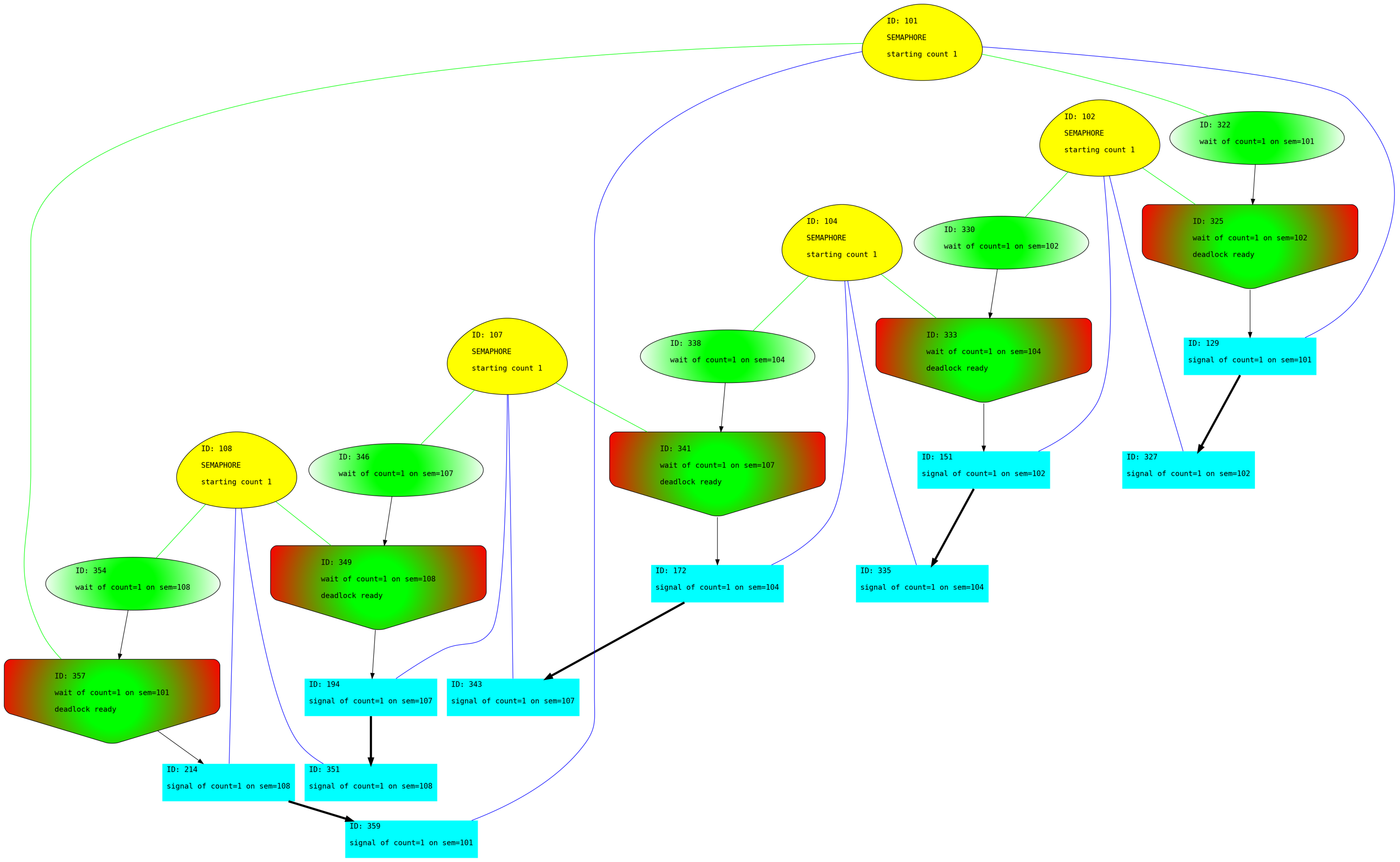

ID: 101
SEMAPHORE
starting count 1
ID: 102
SEMAPHORE
starting count 1
ID: 322
wait of count=1 on sem=101
ID: 325
wait of count=1 on sem=102
deadlock ready
ID: 104
SEMAPHORE
starting count 1
ID: 330
wait of count=1 on sem=102
ID: 333
wait of count=1 on sem=104
deadlock ready
ID: 129
signal of count=1 on sem=101
ID: 107
SEMAPHORE
starting count 1
ID: 338
wait of count=1 on sem=104
ID: 341
wait of count=1 on sem=107
deadlock ready
ID: 151
signal of count=1 on sem=102
ID: 327
signal of count=1 on sem=102
ID: 108
SEMAPHORE
starting count 1
ID: 346
wait of count=1 on sem=107
ID: 349
wait of count=1 on sem=108
deadlock ready
ID: 172
signal of count=1 on sem=104
ID: 335
signal of count=1 on sem=104
ID: 354
wait of count=1 on sem=108
ID: 357
wait of count=1 on sem=101
deadlock ready
ID: 194
signal of count=1 on sem=107
ID: 343
signal of count=1 on sem=107
ID: 214
signal of count=1 on sem=108
ID: 351
signal of count=1 on sem=108
ID: 359
signal of count=1 on sem=101

Figure 7.5: Deadlock graph constructed from traces from the dining philosophers problem (5 philosophers) with reduction (yellow nodes represent semaphores). In the deadlocked state vertices with white outline have already been executed and vertices with red outline could be executed but are stopped because of not enough permits on semaphore.

Chapter 8

Work done

Let's summarize who was responsible for which tasks in the project.

- Examining the existing research, filtering it by reading the introductions, presenting the more interesting results to the whole group, debating about whether it can be useful – Alicja, Antoni, Michał and Natalia.
- Examining and understanding Seastar concurrency model – Antoni and Michał.
- Defining happens-before relationship for Seastar primitives – Antoni and Michał.
- Implementing and optimizing tracing module – Antoni and Michał.
- Developing the detection algorithm – Alicja, Antoni and Natalia.
- Implementing the detection algorithm – Alicja and Natalia.
- Writing the parser from some intermediate form – Alicja and Natalia.
- Graph visualisation – Michał.
- Creating tests for deadlock detection – Alicja, Antoni, Michał and Natalia.
- Rewriting code from Python to C++ after prototyping phase – Antoni.
- Developing and adding the optimizations to the detection algorithm – Alicja and Antoni.
- Improving and adding features to C++ parser – Antoni and Michał.
- Testing tracing efficiency on Scylla – Michał.
- Testing detection efficiency on Scylla – Antoni.
- Writing blogpost about our project for Scylla – Alicja and Michał.
- Writing this thesis – Alicja, Antoni, Michał and Natalia.

Chapter 9

Summary

The goal of the project was to detect deadlocks in programs written in Seastar framework. It ended up with a partial success. We managed to create a tool for almost automatic deadlock detection in arbitrary Seastar programs, however not without caveats.

We succeeded in defining what is a happens-before graph for an asynchronous concurrency model with promises and futures, without explicit threads. This has a broader scope than only Seastar, as a similar model is used in multiple other programming languages and frameworks, most notably in ECMAScript (commonly known as JavaScript) [12]. Our tool works well for all of our test programs. Some of them are very simple, some are somewhat more complicated, in all they are meant to use all the popular constructs of Seastar (meaning different kinds of parallel loops for example). We are also able to parse the logs from Scylla and run the detection. We probably also would be able to find deadlocks there if we came up with a way to deal with the false positives. The tool for generating the graphs of execution can as well be used for other purposes than deadlock detection. It could be used as a debugging tool in a complex Seastar applications, especially such that doesn't use common programming patterns (say, non-obvious control flow, heavy continuation-passing style, or other deviations from the norm).

In our project, some areas require more work. Firstly, perhaps the detection could be optimized more, because in the end, we didn't manage to overcome the exponential computational complexity. For example, maybe some randomized algorithms could be used. Moreover, some Scylla specific optimisations could be added — maybe there are some observations about the structure of its happens-before graph that we were unable to make. Secondly, as mentioned before, we only search for deadlock among the semaphore operations. It could be useful to include other synchronisation primitives, even though they are less widely used than semaphores. Moreover in our solution, we need to assume that the happens-before graph is not influenced by the actual order of execution and this creates some false positives. Maybe it would be possible to somehow connect our approach with some static analysis to avoid them.

Seastar is used in a multitude of projects and hence its repository needs to contain only the features that are updated and well maintained. Seastar is continuously changing so someone making changes would need to understand, what is our code doing in each of the at least 50 places in which we inserted it. Since our tool is only good enough to run on rather small-scaled applications, having our code in their main repository is not worth the effort for ScyllaDB and so, our code is not getting merged into upstream Seastar. However, our tool is in a public repository at `https://github.com/haaawk/seastar/tree/master` and if someone wants to try using it, it is easily available.

Bibliography

[1] R. Agrawal, M. Carey, D. DeWitt. *Deadlock Detection is cheap*, ERL MEMORANDUM NO. MS3/5, January 1983

[2] R. Agrawal, L. Wang, S. D. Stoller. *Detecting potential deadlocks with static analysis and run time monitoring.* In Proceedings of the Parallel and Distributed Systems: Testing and Debugging (PADTAD) Track of the 2005 IBM Verication Conference, volume 3875 of Lecture Notes in Computer Science, pages 191-207 Springer-Verlag, 2006

[3] R. Agrawal, L. Wang, S. D. Stoller. *Detecting potential deadlocks with static analysis and run time monitoring.* Technical Report DAR-05-25, Computer Science Department, SUNY at Stony Brook, Sept, 2005.

[4] Y. Cai, W. K. Chan. [IEEE 2012 34th International Conference on Software Engineering (ICSE) - Zurich, Switzerland (2012.06.2-2012.06.9)] 2012 34th International Conference on Software Engineering (ICSE) - MagicFuzzer: Scalable deadlock detection for large-scale applications.

[5] L. Fajstrup, M. Raußen. *Detecting Deadlocks in Concurrent Systems* Basic Research in Computer Science RS-96-16, May 1996

[6] D. G. Feitelson. *Deadlock detection without wait-for graphs*, Parallel Computing, Volume 17, Issue 12, pp. 1377-1383, 1991

[7] GraphViz is a converter from a text description of a graph to an image. `https://graphviz.org/`, accessed 2021-06-13

[8] J.N. Gray, P. Homan, H. Korth, R. Obermarck. *A Straw Man Analysis of the Probability of Waiting and Deadlock in a Database System*, Rep. RJ3066, IBM Research Lab., San Jose, California, Feb. 1981

[9] E. Koskinen, M. Herlihy. *Dreadlocks: Efficient Deadlock Detection* SPAA 2008: Proceedings of the 20th Annual ACM Symposium on Parallelism in Algorithms and Architectures, Munich, Germany, June 2008

[10] T, Li, C. S. Ellis, A. R. Lebeck, D. J. Sorin. *Pulse: A Dynamic Deadlock Detection Mechanism Using Speculative Execution.* 2005 USENIX Annual Technical Conference, April 2005

[11] I. Molnar, A. van de Ven. *Runtime locking correctness validator*, 2006. `https://www.kernel.org/doc/Documentation/locking/lockdep-design.txt`

[12] Mozilla Developer Network. Concurrency model in JavaScript via event loop. `https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop`, accessed 2021-06-13

[13] N. Ng, N. Yoshida. *Static Deadlock Detection for Concurrent Go by Global Session Graph Synthesis.* CC 2016: Proceedings of the 25th International Conference on Compiler Construction, pp. 174–184, March 2016

[14] Protocol Buffers. Google's data interchange format. `https://github.com/protocolbuffers/protobuf`, accessed 2021-06-13

[15] O. A. V. Ravnås. Frida - A world-class dynamic instrumentation framework. `https://frida.re/`, accessed 2021-06-13

[16] Source code of Scylla Database. `https://github.com/scylladb/scylla`, accessed 2021-06-13

[17] Seastar framework documentation. `http://docs.seastar.io/master/index.html`, accessed 2021-06-13

[18] C. Voss, V. Sarkar. *An Ownership Policy and Deadlock Detector for Promises* Principles and Practice of Parallel Programming, ACM, Pages 348-361, 2021

[19] J. Zhou, S. Silvestro, H. Liu, Y. Cai, T. Liu. *UNDEAD: Detecting and preventing deadlocks in production software*, 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 729-740, 2017