Chapter 4

Deadlocks in executions

When running a program, we want to be able to determine, if a deadlock occurred in a program execution E.

To define deadlocks, we classify all *active* routines (meaning routines that have not finished yet) by the following terms. We call routines that are running or runnable, or routines that are *waiting* on an operations, that will be conclude without being awoken from the outside (such as timers), *running* routines. Routines that are currently waiting on a concurrency operation like a channel or mutex are called *waiting* routines. If the operation, that blocks a *waiting* has no chance of being woken up again, we call it a *dead* routine. We define

$$\mathcal{G}_R = \{G_i \mid \mathit{running}(G_i)\} \tag{4.1}$$
$$\mathcal{G}_W = \{G_i \mid \mathit{waiting}(G_i)\} \tag{4.2}$$
$$\mathcal{G}_D = \{G_i \mid \mathit{dead}(G_i)\} \tag{4.3}$$
$$\tag{4.4}$$

It is easy to see, that

$$\mathcal{G}_D \subseteq \mathcal{G}_W \tag{4.5}$$
$$\mathcal{G}_A = \mathcal{G}_R \cup \mathcal{G}_W, \tag{4.6}$$

where $\mathcal{G}_A$ is the set of all *active* routines.

4.1 Total deadlock

A total deadlock is a situation, where

$$\mathcal{G}_R = \emptyset, \tag{4.7}$$

Since a *waiting* routine can only be woken up by a *running* routine, it follows that

$$\mathcal{G}_R = \emptyset \Rightarrow \mathcal{G}_A = \mathcal{G}_W = \mathcal{G}_D, \tag{4.8}$$

or in other words, if there are no *running* routines, then all *active* routines are dead. This will necessary cause the program to get stuck and to never terminate. An example for this can be seen in code 4.1. The program consists of the main routine (G_0) and one other routine ($G1$). The trace depicts an execution, where the first lock operations in each routine can acquire the lock (indicated by !), but then it tries but is not able to acquire the second lock (indicated by ?), since it is held by the other routine. Since all routines, and especially the main routine G_0 are involved in this deadlock, the program will never terminate and we have created a total deadlock.

```go
func main() { // G0
  m := sync.Mutex{}

  go func() { // G1
    m.Lock()
    n.Lock()
    n.Unlock()
    m.Unlock()
  }()

  n.Lock()
  m.Lock()
  m.Unlock()
  n.Unlock()
}
```

	G_0	$G1$
1	$fork(G1)$	
2	$lock(m)!$	
3		$lock(n)!$
4	$lock(n)?$	
5		$lock(m)?$

Code 4.1: Example for program with total deadlock

Detecting total deadlocks in go is easy and already automatically done by the scheduler in the Go Runtime. Since the runtime can only run functions in $\mathcal{G}_R$, it knows a total deadlock has occurred, if it is not able to schedule any routine.

In this case, the program is terminated automatically.

4.2 Partial deadlock

Partial deadlocks, only affect a subset of *active* routines. In this case, the program will terminate as soon as the main routine is finished. For this, an example can be found in code 4.2. The code is almost the same as in code 4.1, the only difference is, that the second block of lock and unlock is not longer in the main routine, but in its own routine. While the locks still create a deadlock, this block does not effect the main routine, which can simply return, making this kind of deadlock much more difficult to detect.

Partial deadlocks are situations, where

$$\mathcal{G}_D \neq \emptyset \wedge \mathcal{G}_R \neq \emptyset, \tag{4.9}$$

meaning there are *dead* routines, but not all routines are *dead*.

While total deadlocks are the more important type of bug to find, since only it can cause the program to get stuck, we also want to detect partial deadlocks. Since partial deadlocks are generally not a situation a developer intentionally adds to the code, it generally indicates an unwanted situation and may result in the program not performing as expected.

4.2.1 Idea

Unfortunately partial deadlocks are much more difficult to detect compared to total deadlocks, since we do not only need to find the routines that are waiting, but als need to determine, if there is a possibility of them waking up again. While for a total deadlock we only need to distinguish between *running* and *waiting* routines, we now have to additionally divide the *waiting* routines into those that are *dead* and those that are not.

The idea we implement is based on [3] but differs especially in its implementation.

The general idea is to determine for each *waiting* routine, if the blocking element is also referenced in a non-*dead* routine. For this, we can look at code 4.3. In both examples, the code blocks on the send on c. In code 4.1a, there is no

```go
func main() {
  m := sync.Mutex{}

  go func() { // G1
    m.Lock()
    n.Lock() // blocks
    n.Unlock()
    m.Unlock()
  }()

  go func() { // G2
    n.Lock()
    m.Lock() // blocks
    m.Unlock()
    n.Unlock()
  }()
}
```

	G_0	$G1$	$G2$
1	$fork(G1)$		
1	$fork(G2)$		
3		$lock(m)!$	
4			$lock(n)!$
5		$lock(n)?$	
6			$lock(m)?$
7	$return$		

Code 4.2: Example for program with partial deadlock

other routine, that has a reference to this channel c. It is therefore impossible for another routine to wake up the blocked routines, and we have a deadlock. In code 4.1b on the other hand, we have a different situation. Lets assume, we call the deadlock detection, while the first routine is *waiting* on the channel send, but before the second routine has reached the corresponding receive. In this case, there is a reference to the blocking element in an *running* routine. The first routine is therefore not part of a deadlock.

Lets assume routine G_i is *waiting*. We call $\mathcal{E}_{Wi}$ the set of elements, G_i is waiting on (with elements we mean the underlying element, e.g. a specific mutex or channel, not the trace elements/operation). In most cases, this set contains exactly one element. The number of elements can only be greater than one for selects without a default case and multiple non-default cases. Let Ref_i be the set of routines, that have a reference to any element $e \in \mathcal{E}_{Wi}$, not including G_i. For this, we use the same idea of reference as the Go garbage collector [4]. Let a routine G_i have the property *suspect*, if we can determine, that our analysis cannot determine if the routine is *dead* or not. We define the property *waiting′*

of a routine as

$$waiting'(G_i) \equiv waiting(G_i) \wedge \neg dead(G_i) \wedge \neg suspect(G_i) \tag{4.10}$$
$$\mathcal{G}_W{}' = \{G_i \mid waiting'(G_i)\} \tag{4.11}$$

We can than determine if a blocked routine is part of a partial deadlock, by applying the following rules:

$$\text{NoReference:} \quad \frac{waiting'(G_i) \qquad Ref_i = \emptyset}{dead(G_i)} \tag{4.12}$$

$$\text{AliveReference:} \quad \frac{waiting'(G_i) \qquad \exists G_j \in Ref_i : running(G_j)}{suspect(G_i)} \tag{4.13}$$

$$\text{DeadReference:} \quad \frac{waiting'(G_i) \qquad \forall G_j \in Ref_i : dead(G_j)}{dead(G_i)} \tag{4.14}$$

$$\text{SuspectReference:} \quad \frac{waiting'(G_i) \qquad \exists G_j \in Ref_i : suspect(G_j)}{suspect(G_i)} \tag{4.15}$$

$$\text{NoOtherRule:} \quad \frac{waiting'(G_i) \qquad \neg\exists G_j \in \mathcal{G}_W : \text{other rule can be applied to } G_j}{dead(G_i)} \tag{4.16}$$

NoReference describes situations, where no other routines have a reference to a blocking element. This is the situation shown in code 4.1a. Since no other routine has a reference to channel c and can therefore wake up the blocking send in line 5, the routine $G1$ is *dead*and therefore part of a (local) deadlock.

AliveReference states, that if a blocking element in a *waiting* routine G_i has a reference in another routine G_j that is still running, G_i is not dead, since G_j could end the block in G_i. An example for this is in code 4.1b. Here the send on the channel c blocks routine G_1. If we run the analysis while G_2 is in the *doSomething* function, G_2 is in *running* and in Ref_i. This means, it can wake up the blocked routines, by calling the receive of c. While we cannot conclude from such a situation, that a *waiting* routine is in a deadlock, we can also not guarantee, that it is not. This is especially the case, if the functions on the blocked elements are not compatible with the blocked operations. If the receive would be a send, the routine would be dead. Since we cannot distinguish between the two situations based on the given information, we cannot determine the routine to be *dead* or not *dead*, meaning it is *suspect*.

DeadReference is the situation, where all other *active* routines, that also hold a reference to a blocking element are dead. We can see this in code 4.1c. Lets assume, G_1 has acquired the routine and then terminated. It is therefore not an *active* routine anymore and can be ignored for further analysis. G_2 not blocks on the mutex lock. There is an *active* reference in G_3, but the routine would first need to run the send on the channel. By NoReference we can conclude, that G_3 is dead, since no other *active* routine contains a reference to the channel. Given this, we call apply DeadReference to conclude, that G_2 is also dead, since all *active* routines holding a reference to the mutex (not including G_2 itself) are dead.

SuspectReference is a transitive rule. To determine if a waiting routine could at some point be woken up, we need to determine if there is a routine that could wake it up, meaning a routine that holds a reference to blocking element. This is the case, if a routine is either *running* or if it is currently waiting, but could be woken up in the future. This is directly marked by a routine being *suspect*. For this reason we can conclude, that a routine where the blocked element is referenced in a routine that is marked *suspect* must also be *suspect*. An example for this can be seen in code 4.1d. Here, we assume, that G_1 is blocked by the send on c, G_2 is blocked by the send on d and G_3 is in the *doSomething* function. We want to determine the status of G_1. The blocking element in G_1 is also referenced in G_2. Since G_3 is running, and therefore *running*, we can use AliveReference to determine, that G_2 is *suspect*. This means it could be *dead*, but could also be woken up again at some point. If it is woken up again, this would also release G_1, meaning we cannot determine that G_1 is dead. But if G_2 would be *dead*, or if the operations in G_2 would not be compatible with the *send* in G_2 (we only know that there is a reference, but not if it is the required receive or a send or closed), G_1 would be dead. We therefore classify the routine as *suspect*.

NoOtherRule is a default rule. If we get to a point, where there are still routines G_i with $waiting'(G_i)$, but we cannot apply any other rule to any routine, we can conclude that all routines that have not been classified as either *dead* or *suspect* must be *dead*. This mainly covers the case for cyclic deadlocks. If we use code 4.2 as an example we can see why this is required. We cannot use NoReference, since for each blocking element, the other routine contains a reference. We cannot apply AliveReference since non of the routines containing references to the blocked elements are *running*. Since NoReference and AliveReference cannot be used, we cannot classify any of the routines as *dead* or *suspect*. It is therefore not possible to apply DeadReference or SuspectReference. Since there is therefore no other rule we can apply, we apply NoOtherRule to determine, that both routines are dead, meaning they form a deadlock. Since for all not-dead routines we must have been able to call one AliveReference or SuspectReference at some point during the analysis, by doing so removing them from $\mathcal{G}_W{}'$, NoOtherRule cannot be applied, meaning this rule cannot lead to a false positive.

4.2.2 Implementation

We implement the detection of local deadlocks during runtime inside the patch Go runtime[1]. We run a background routine, that runs the analysis once per second and when the program terminates.

The analysis contains three phases. First we determine all currently waiting routines and for each of them, which element is blocking. Then we use the garbage collector to determine all routines that have a reference to any of the blocking elements. At last, we use the rules explained above, to determine if there are any dead routines. If we find a routine, we report it as a deadlock.

Before we can determine if routines are *dead* or not, we first need to find all routines in $\mathcal{G}_W$. Fortunately, the go runtime allows us the easily detect such routines. Each routine is internally represented by an object called g. From this g, we can read the current state of the routine, e.g. *running*, *runnable* or *waiting*. A routine, that is blocked by a concurrency element is always in the *waiting* state. If the routine is in a *waiting* state, we can also query an additional field in g called *waitreason*, witch tells us, why the routine is waiting. If this vale is *waitReasonChanSend*, *waitReasonSyncMutexLock*, or similar, we known that the routine is currently blocking on one of the relevant concurrency operations and therefore *waiting*.

To detect if there are any *dead* routines in $\mathcal{G}_W$, we need to find out, if the blocking element could be released in the future. For this we need to determine, on which operations the routine blocks and if another routine could release it. While g does not directly contain a field pointing to the element it is *waiting* on, it is easy to add such an element for all relevant operations, adding an additional

[1]goPatch/src/runtime/advocate_local_deadlock.go

field and always setting it, before the relevant *gopark* command is run, which causes the routine to switch to the *waiting* state.

Using the internal $forEachG$ function provided by the go runtime, we can iterate over the g structures all *active* routines. Using the internal state of the routine represented by g and the *waitreason*, we can filter out the set $\mathcal{G}_W$ all routines that are currently *waiting* and for each of these routines the set of waiting elements $\mathcal{E}_{Wi}$.

In the second phase we adapt and run the Go garbage collector (GC) [4]. The GC finds all "roots", in our case, meaning variables accessible from the stack, and marks every object that can be reached from these roots. By propagating this process through references, like pointers, slices, maps or channels, it marks the accessible memory. We can use this functionality to our advantage. By slightly modifying the implementation, we can determine for each marking, from which routine the propagation was started and therefore, which routines has a reference to an object at a given memory address. When marking a cell, we check if the memory address is equal to the address of one of the elements at least one routine is waiting and if so, store which routine has marked a reference to the element[2]. Doing this, we can for each blocking element find all active routines, that have a reference to the same element.

Allocating memory while the GC is running can lead to problems. For this reason, we need to avoid this by allocating all required memory before the GC is started. This is mainly required when storing which routines reference each blocked element.

In the third phase, we iterate over all routines in $\mathcal{G}_W$ and try to apply any of the rules outlined in eq. (4.12)-(4.15). We repeat this, until none of the rules can be applied to any of the *waiting* routines and then report all *waiting* routines, that are marked as *dead* as well as all *waiting* routines that have not been marked *suspect*, in accordance with NoOtherRule (eq. (4.16)).

[2]goPatch/src/runtime/mgcmark.go, line 1431–1437

```go
var c = make(chan int)

func main() {
  go func() { // G1
    c <- 1
  }()

  go func() { // G2
    doSomething()
  }()
}
```

(a) NoReference

```go
var c = make(chan int)

func main() {
  go func() { // G1
    c <- 1
  }()

  go func() { // G2
    doSomething()
    <-c
  }()
}
```

(b) AliveReference

```go
var c = make(chan int)
var m = sync.Mutex{}

func main() {
  go func() {  // G1
    m.Lock()
  }

  go func() {  // G2
    m.Lock()
  }

  go func() {  // G3
    c <- 1
    m.Unlock()
  }()

}
```

(c) DeadReference

```go
var c = make(chan int)
var d = make(chan int)

func main() {
  go func() { // G1
    c <- 1
  }()

  go func() { // G2
    d <- 1
    <-c
  }

  go func() { // G3
    doSomething()
    <-d
  }
}
```

(d) SuspectReference

Code 4.3: Examples for rules