Bachelorprojekt

Dynamic Deadlock Detection in Go

Erik Daniel Kassubek

Datum

Albert-Ludwigs-Universität Freiburg im Breisgau
Technische Fakultät
Institut für Informatik

Betreuer
Prof. Dr. Thiemann, Albert-Ludwigs-Universität Freiburg
Prof. Dr. Sulzmann, Hochschule Karlsruhe

TODO: write abstract

Inhaltsverzeichnis

1 Einführung

2 Theoretischer Hintergrund

2.1 Deadlocks

Ein Deadlock ist ein Zustand in einem nebenläufigen Programm, indem alle laufenden Tread zyklisch auf die Freigabe von Ressourcen warten.
Im großen und ganzen lassen sich Deadlocks in zwei Gruppen einteilen: Ressourcen-Deadlocks und Kommunikations-Deadlocks [1]. Im folgenden sollen jedoch nur durch Locks erzeugte Ressourcen-Deadlocks betrachtet werden.
Locks gehören zu den am weitesten verbreiteten Mechanismen um sicher zu stellen, dass sich in einem nebenläufigen Programm immer nur ein Thread in einem kritischen Bereich aufhalten kann [1], bzw. dass immer nur maximal ein Tread gleichzeitig auf eine gemeinsame Resource, wie z.B. eine Variable zugreifen kann. Möchte ein Thread T_1 nun in einen Bereich eintreten, der durch ein Lock, welches bereits von einem anderen Thread T_0 beansprucht wird, geschützt ist, muss es so lange vor dem Lock warten, bis dieses von T_0 wieder frei gegeben wird.
Ein Deadlock kann nun entstehen, wenn all Treads vor einem solchen Lock warten müssen, wobei die Locks immer von einem anderen Tread gehalten werden. Im folgende bezeichnet $acq_i(l)$, dass das Lock l von Prozess T_i beansprucht und $rel_i(l)$, dass l von T_i wieder freigegeben wird. Man betrachte nun das folgende Beispiel [2] mit den Treads T_1 und T_2:

	T_1	T_2
1.	$acq_1(y)$	
2.	$acq_1(x)$	
3.	$rel_1(x)$	
4.	$rel_1(y)$	
5.		$acq_2(x)$
6.		$acq_2(y)$
7.		$rel_2(y)$
8.		$rel_2(x)$

Da T_1 und T_2 gleichzeitig ablaufen ist folgender Ablauf möglich:

	T_1	T_2
1.	$acq_1(y)$	
5.		$acq_2(x)$
2.	$B - acq_1(x)$	
6.		$B - acq_2(y)$

Dabei impliziert $B - acq_i(l)$, dass $acq_i(l)$ nicht ausgeführt werden konnte, bzw. dass der Thread T_i vor dem Lock halten muss, da das Lock bereits von einem anderen Tread beansprucht wird. In diesem Beispiel wartet nun T_1 darauf, dass das Lock x geöffnet wird und T_2 wartet darauf, dass Lock y geöffnet wird. Da nun aller Thread warten müssen, bis ein Lock freigegeben wird, allerdings keiner der Threads weiter laufen kann um ein Lock zu öffnen, kommt es zum Stillstand. Dieser Zustand wird als Deadlock bezeichnet.

2.2 Deadlocks-Detection

Deadlocks in Programmen sind Fehler, die oft den vollständige Abbruch eines Programmes zu Folge haben, wenn keine zusätzliche Routine zur Auflösung von Deadlocks implementiert ist. Aus diesem Grund möchte man bereits bei der Implementierung eines Programmes verhindern, dass ein solcher Deadlock auftreten kann. Unglücklicherweise kann es ohne zusätzliche Hilfsmittel schwierig sein, einen solchen Deadlock zu erkennen, da das Auftreten eines Deadlock von dem genauen Ablauf der verschiedenen Threads abhängt.
Um dennoch Deadlocks detektieren zu können, können Lock-Graphen oder Lock-Bäume verwendet werden.

2.2.1 Lock-Graphen

Ein Lock-Graph ist ein gerichteter Graph $G = (L, E)$, Dabei ist L die Menge aller Locks $E \subseteq L \times L$ ist definiert als $(l_1, l_2) \in E$ genau dann wenn es einen Tread T gibt, welcher $acq(l_2)$ ausgeführt, während er bereits das Lock l_1 hält [3]. Mathematisch ausgedrückt gilt also

$$(l_1, l_2) \in E \Leftrightarrow \exists t_1, t_3 \nexists t_2 \ ((t_1 < t_2 < t_3) \wedge aqr(l_1)[t_1] \wedge rel(l_1)[t_2] \wedge aqr(l_2)[t_3])$$

wobei $aqr(l_1)[t_i]$ bedeutet, dass $aqr(l_1)$ zum Zeitpunkt t_i ausgeführt wird und equivalent für $rel(l_1)[t_i]$.
Ein Deadlock kann nun auftreten wenn es innerhalb dieses Graphen einen Kreis gibt. Um zu verhindern, dass ein false positive dadurch ausgelöst wird, dass alle Kanten in einem solchen Kreis aus dem selben Thread kommen (wodurch kein Deadlock entstehen kann), können die Kanten noch zusätzlich mit einem Label versehen werden, welche den Thread identifiziert, durch welchen die Kante in den Graphen eingefügt wurde. Bei dem Testen nach Zyklen muss nun beachtet werden, dass nicht alle Kanten in dem Kreis das selbe Label haben [3].

2.2.2 Lock-Bäume

Anders als bei Lock-Graphen, speichert bei Lock-Bäumen jede Routine seine seine eigenen Abhängigkeiten. Dies bedeutet, dass der Lock-Baum b der Routine r genau dann die Kanet $x- > y$ besitzt, wenn in Routine r das Lock y beansprucht wird, während in r das Lock x bereits gehalten ist.
Diese Bäume haben mehrere Vorteile gegenüber Lock-Graphen. Da jede Routine seine eigene Datenstruktur hat, wird verhindert, dass es durch gleichzeitigen Zugriff verschiedener Routinen auf die selbe Datenstruktur zu Problemen kommen kann. Zudem muss die Zugehörigkeit der Abhängigkeiten zu verschiedenen Routinen nicht explizit gespeichert werden. Ein weiterer Vorteil besteht darin, dass Guard-Locks bei Lock0-Bäumen, anders alb ein Lock-Graphen nicht zu False-Positives kommt.

3 Review bestehender Tools

3.1 go-deadlock

Im folgenden soll eine Software zur Erkennung von Deadlocks analysiert werden. Dazu wird die Software "sasha-s/go-deadlock" [4], veröffentlicht auf Github, betrachtet.
Dieses verwendet Drop-In Replacements für die, in go standardmäßig implementierten, sync.Mutex Locks. Diese führen sowohl das eigentliche Locking aus und können, beim Durchlaufen des Programms Situationen erkennen, die zu einem Deadlock führen können. Dabei werden sowohl ein allgemeines Lock als auch Readers–writer-locks implementiert. Dies wirkt sich allerdings nur auf die Anwendung des eigentlichen Lockings aus, nicht aber auf die Detection von Deadlocks. Aus diesem Grund wird hierauf im folgenden nicht weiter eingegangen, und die Methoden für die Detection von Deadlocks bezieht sich sowohl auf die allgemeinen als auch die Readers–writer Locks.
Für die Erkennung werden drei verschiedene Fälle betrachtet:

- Rekursives Locking
- Zyklisches Locking
- Timeout

Diese sollen im folgenden genauer betrachtet werden.

3.1.1 Rekursives Locking

Unter rekursiven Locking versteht man, dass ein Lock von dem selben Thread mehrfach geschlossen wird, ohne zwischen diesen Schließungen wieder geöffnet zu werden. Ein Beispiel dazu ist das folgende Programm:

```
1 func recursiveLocking() {
2     var x deadlock.Mutex
3     x.Lock()
4     x.Lock()
5     x.Unlock()
6 }
```

Hierbei kann es schon, wie in dem Beispielcode gezeigt, bei nur einem Thread zu einem Deadlock kommen. Die Funktion beansprucht das Lock *mu* in Zeile 3 für sich. In Zeile 4 versucht sie erneut das Lock x zu beanspruchen. Da dieses aber schon in Zeile 3 beansprucht wurde, muss das Programm vor Zeile 4 warten, bis *mu* wieder freigegeben wird. Da dadurch aber die Freigabe in Zeile 3 niemals erreicht werden kann wartet das Programm für immer, befindet sich also in einem Deadlock.
Um solche Situation zu erkennen speichert das Programm ein Dictionary *cur* mit allen momentan gehaltenen Locks. Die Werte zu den jeweiligen Keys speichern sowohl, von welcher Go-Routine das Lock momentan gehalten wird, als auch in welcher Datei und Zeile der Befehl, der das Locking dieses Locks zu folge hatte, zu finden ist. Wird ein Lock wieder freigegeben, so wird der entsprechende Eintrag aus *cur* entfernt. Wird nun ein Lock neu beansprucht, überprüft das Programm, ob dieses Lock mit dem Thread, der das Lock beanspruchen möchte bereits in der Liste gehalten Locks auftaucht. Ist dies der Fall, dann nimmt das Programm an, dass es sich hierbei um ein mögliches Deadlock handelt und führt entsprechende Schritte aus, um den Nutzer zu warnen. Wird ein Lock wieder frei gegeben, wird der entsprechende Eintrag aus $l.loc$ entfernt.

3.1.2 Zyklisches Locking

Mit dieser Methode werden mögliche Deadlocks gefunden, die dadurch entstehen, dass alle Threads zyklisch auf die Freigabe eines Locks warten, welches von einem anderen Thread gehalten wird. Ein Beispiel dafür kann in folgender Funktion gesehen werden:

```
1  func circularLocking() {
2      var x Mutex
3      var y Mutex
4      ch := make(chan bool, 2)
5
6      go func() {
7          y.Lock()
8          x.Lock()
9          x.Unlock()
10         y.Unlock()
11         ch <- true
12     }()
13     go func() {
14         x.Lock()
15         y.Lock()
16         y.Unlock()
17         x.Unlock()
18         ch <- true
19     }()
20
21     <-ch
22     <-ch
23 }
```

Dieses Beispiel entspricht dem in Kap. 2.1 beschreiben Beispiel.
Für die Detektion verwendet das Programm zwei Dictionaries. Das erste ist *cur*, welches bereits in 3.1.1 betrachten wurde.
Das andere Dictionary *order*, definiert mit seinen Keys die Kanten eines Lock-Graphen. Die Keys bestehen dabei aus einer Struktur $beforeAfter$, die Referenzen zu den beiden Locks speicherte, welche von der Kante im Graphen verbunden werden. Wird ein neues Lock p beanspruchen, so wird für jedes Lock b, welches sich momentan in *cur* befindet ein neuer Eintrag $beforeAfter\{b, p\}$ in *order* hinzugefügt. Die Werte, die für jeden Key in *order* gespeichert werden, entsprechen den Information, die auch in *cur* für die beiden Locks gespeichert wird. Allerdings wird auf die Speicherung der ID der erzeugenden Go-Routine verzichtet, da sie nicht benötigt wird. Information aus *order* werden nur entfernt, wenn *order* eine festgelegte maximale Größe überschreitet.
Die Überprüfung, ob ein Lock p zu einem Deadlock führen kann, finden bereits statt, bevor das Lock in *cur* und *order* eingetragen wird. Dazu wird für jedes Lock b in *cur* überprüft, ob *order* einen Key $beforeAfter\{p, b\}$ besitzt, der Graph also die beiden Locks in umgekehrter Reihenfolge enthält. Existiert solch ein Key, und wurde b von einer anderen Routine als p in *cur* eingefügt, bedeutet dies einen Loop aus zwei Kanten im Lockgraphen und somit einen möglichen Deadlock.
Dies bedeutet aber auch, dass das Programm nicht in der Lage ist, ein Kreis in einem Lock-Graphen zu finden, wenn dieser aus drei oder mehr Kanten besteht. Soche Situationen können aber dennoch zu Deadlocks führen. Ein Beispiel dafür ist die folgende Funktion:

```
1  func threeEdgeLoop() {
2      var x Mutex
3      var y Mutex
4      var z Mutex
5      ch := make(chan bool, 3)
6
7      go func() {
8          // first routine
9          x.Lock()
10         y.Lock()
11         y.Unlock()
12         x.Unlock()
13         ch <- true
14     }()
15
16     go func() {
17         // second routine
18         y.Lock()
19         z.Lock()
20         z.Unlock()
21         y.Unlock()
22         ch <- true
23     }()
24
25     go func() {
26         // third routine
27         z.Lock()
28         x.Lock()
29         x.Unlock()
30         z.Unlock()
31         ch <- true
32     }()
33
34     <-ch
35     <-ch
36     <-ch
37 }
```

Führen die drei Routinen jeweils ihre erste Zeile gleichzeitig aus, muss jede Routine vor ihrer zweiten Zeile warten und es kommt zu einem Deadlock. Da diese Konstellation in einem Lockgraphen aber zu einem Kreis mit einer Länge von drei Kanten führen würde, kann das Programm den möglichen Deadlock nicht erkennen.
Ein weiteres Problem besteht darin, dass das Programm Einträge aus $l.order$ nicht löscht, wenn ein Lock wieder frei gegeben wird und auch nicht überprüft, das die Kanten in verschiedenen Go-Routinen erzeugt wurden. Sie werden nur entfernt, wenn das Dictionary eine bestimmte Größe überschreitet. Dies kann zu false-positives führen. Ein Beispiel dazu wäre die folgende Funktion:

```
1  func noDeletion() {
2      var x Mutex
3      var y Mutex
4
5      x.Lock()
6      y.Lock()
7      y.Unlock()
8      x.Unlock()
9
10     y.Lock()
11     x.Lock()
12     x.Unlock()
13     y.Unlock()
14 }
```

Es ist sehr einfach zu sehen, dass es in dieser Funktion nicht zu einem Deadlock kommen kann. Dennoch zeigt das Programm einen möglichen Deadlock an, da die Information über die Locks x und y aus Zeile $7-8$ in Zeile $10-11$ immer noch vorhanden ist, obwohl die beiden Locks in jedem Fall wieder freigegeben werden, bevor Zeile 10 erreicht wird.

3.1.3 Timeout

Neben diesen beiden Methoden, die vorausschauend nach möglichen Deadlocks Ausschau halten, versucht das Programm auch mit Timeouts um zu überprüfen ob sich das Programm bereits in einem Deadlock befindet. Möchte ein Thread ein Deadlock beanspruchen wird vorher eine go routine mit einem Counter gestartet. Sollte die Inanspruchnahme des Locks innerhalb der vorgegebenen Zeit (default: 30s) gelingen, wird die go routine beendet. Sollte es nicht gelingen, nimmt das Programm nach der festgelegten Zeit an, dass es zu einem Deadlock gekommen ist und gibt eine entsprechende Nachricht aus.
Diese Methode kann durchaus nützlich sein, um über Deadlocks informiert zu werden. Allerdings führt sie sehr leicht zu false-positives, wenn die Abarbeitung anderer Routinen und damit die Freigabe des Locks länger Dauer, als die festgelegte Timeout Zeit. Im folgenden Beispiel wird dies deutlich:

```
1  func falsePositive() {
2      var x deadlock.Mutex
3      finished := make(chan bool)
4
5      go func() {
6          // first go routine
7          x.Lock()
8          time.Sleep(40 * time.Second)
9          x.Unlock()
10     }()
11
12     go func() {
13         // second go routine
14         time.Sleep(2 * time.Second)
15         x.Lock()
16         x.Unlock()
17         finished <- true
18     }()
19
20     <-finished
21 }
```

Der Chanel finished wird lediglich verwendet um zu verhindern, dass das Programm beendet

wird, bevor die gol Routinen durchlaufen wurden. Er ist für die Deadlock Analyse also irrelevant. Das Programm startet zwei go-Routinen, die beide das selbe Lock x verwenden. Durch den time.Sleep(2 * time.Second) command wird sichergestellt, dass die erste go Routine zuerst auf das Lock zugreift. Dass Lock in Routine 2 muss also warten, bis es in Routine 1 wieder freigegeben wird. Dies geschieht in etwa $38s$ nachdem die zweite Routine mit dem warten auf die Freigabe von x beginnt. Da dies länger ist als die standardmäßig festgelegte maximale Wartezeit von $30s$ nimmt das Programm an, es sei in einen Deadlock gekommen, obwohl kein Solcher vorliegt, und auch kein Deadlock möglich ist.

3.1.4 False Negatives / Positives

Bei False Negatives or False Positives handelt es sich um Fälle, bei denen es zu einem Deadlock kommen kann, ohne dass dieser Detektiert wird, bz. Fälle die nicht zu einem Deadlock führen können, aber dennoch als Deadlock angezeigt werden. Neben den in den vorherigen Abschnitten gezeigten Fällen, kann es auch in weiteren Fällen zu solchem Verhalten kommen.
Ein Fall, bei dem ein Deadlock auftreten könnte, welcher von dem Programm aber nicht erkannt wird (False Negative), entsteht bei verschachtelten Routinen. Dabei erzeugt eine go-Routine eine weitere.

```
1  func nestedGoRoutines() {
2      var x deadlock.Mutex
3      var y deadlock.Mutex
4      ch := make(chan bool)
5
6      go func() {
7          y.Lock()
8          // nested routine
9          go func() {
10             x.Lock()
11             x.Unlock()
12             ch <- true
13         }()
14         <-ch
15         y.Unlock()
16     }()
17
18     go func() {
19         x.Lock()
20         y.Lock()
21         y.Lock()
22         x.Lock()
23     }()
24 }
```

Diese Funktion ist bezüglich ihres Ablaufs identisch zu der Funktion circularLocking in Kapitel 3.1.2. Dennoch ist es dem Programm aufgrund der Verschachtelten go-Routine nicht möglich, den möglichen Deadlock zu erkennen.
Ein Fall, bei dem es zu False Positives kommen kann, wird durch Guard-Locks ausgelöst. In dem folgenden Beispiel wird ein möglicher Deadlock der Locks x und y durch das Guard-Lock z verhindert.

```
1  func guardLocks() {
2      var x deadlock.Mutex
3      var y deadlock.Mutex
4      var z deadlock.Mutex
5
6      ch := make(chan bool, 2)
7
8      go func() {
9          z.Lock()
10         y.Lock()
11         x.Lock()
12         x.Unlock()
13         y.Unlock()
14         z.Unlock()
15         ch <- true
16     }()
17     go func() {
18         z.Lock()
19         x.Lock()
20         y.Lock()
21         y.Unlock()
22         x.Unlock()
23         z.Unlock()
24         ch <- true
25     }()
26
27     <-ch
28     <-ch
29 }
```

Da sich immer nur eine der beiden Routinen in dem Bereich aufhalten kann, der von z begrenzt wird, ist ein Deadlock in diesem Fall nicht möglich. Es wird aber dennoch ein möglicher Deadlock angezeigt, da das Programm nicht in der Lage ist, eine solches Guard-Lock zu erkennen.

3.2 Go Runtime Deadlock Detection

Go besitzt einen eigenen Detektor zur Erkennung von Deadlocks. Allerdings kann dieser, im Unterschied zu dem in 3.1 betrachteten Detector nur tatsächlich auftretende Deadlocks erkennen. Eine Erkennung ob in dem Code ein Deadlock möglich ist findet hierbei nicht statt. Um zu erkennen, ob ein Deadlock vorliegt zählt go die nicht blockierten Go-Routinen. Fällt dieser Wert auf 0, nimmt Go an, dass es zu einem Deadlock gekommen ist und bricht das Programm mit einer Fehlermeldung ab [5].

4 Implementierung eines Deadlock-Detectors

Im folgenden soll die Implementierung eines Deadlock-Detektors basierend auf dem Undead Algorithmus [1] beschrieben werden. Die eigentliche Implementierung in Go kann in [6] gefunden werden.
Der Detektor ist, wie auch der in 3.1 beschriebene Detektor, als Drop-In-Replacement ausgelegt. Das bedeutet, dass die Funktionen des Detektors direkt in den Programm-Code integriert werden. Ein Beispiel dazu ist das folgende Programm:

```
1  import (
2      deadlock "github.com/ErikKassubek/Deadlock-Go"
3  )
4
5  func potentialDeadlock() {
6      x := deadlock.NewLock()
7      y := deadlock.NewLock()
8      ch := make(chan bool, 2)
9
10     go func() {
11         deadlock.NewRoutine()
12
13         x.Lock()
14         y.Lock()
15         time.Sleep(time.Second)
16         y.Unlock()
17         x.Unlock()
18
19         ch <- true
20     }()
21
22     go func() {
23         deadlock.NewRoutine()
24
25         y.Lock()
26         x.Lock()
27         time.Sleep(time.Second)
28         x.Unlock()
29         y.Unlock()
30
31         ch <- true
32     }()
33
34     <-ch
35     <-ch
36 }
37
38 func main() {
39     deadlock.Initalize()
40     defer deadlock.FindPotentialDeadlocks()
41     potentialDeadlock()
42 }
```

Locks, die in Go normalerweise über sync.Mutex erzeugt und gehandhabt werden, werden dabei durch eigens definierte Locks ersetzt. Auf diesen können dann Lock, TryLock und Unlock Ope-

rationen ausgeführt werden.
Im folgende sollen die verschiedenen Phasen

- Optionen
- Initialisierung
- Logging
- Periodische Detektion
- Vollständige Detektion
- Meldung

des Detektors beschrieben werde.

4.1 Optionen

Die Funktionsweise des Detektors kann durch verschiedene Optionen verändert werden. Diese beinhalten die Aktivierung und Deaktivierung der periodischen und vollständigen Detektion, die Festlegung der Abstände, in denen die periodische Detektion aktiviert wird, die Entscheidung ob vollständige Call-Stacks oder nur vereinfachte Informationen gesammelt werden sowie ob Informationen über Single-level-Locks gesammelt werden sollen.
Zudem können maximale Werte für die Anzahl der Dependencies und Routinen, die maximale Größe der gesammelten Call-Stacks sowie die maximale Tiefe von der Lock-Bäume bestimmt werden. Diese Werte haben besonders Einfluss auf den benötigten Speicherplatz. Sind sie allerdings zu klein Gewählt, wird das Programm mit einer entsprechenden Fehlermeldung abgebrochen.
Diese Werte müssen, wenn sie anders als die default Werte gesetzt werden sollen vor der Initialisierung des Detektor gesetzt werden.

4.2 Initialisierung

Sowohl der eigentliche Detektor als auch jede neue go-Routine muss initialisiert werden.
Die Initialisierung, welche über “deadlock.Initialize()” gestartet wird, hat lediglich die Aufgabe, die periodische Detektion zu starten und in regelmäßigen Abständen auszuführen. Dazu wird eine go-Routine erzeugt, die über einen Ticker die periodische Detektion in festgelegten Abständen (default: 2s) started. Die eigentliche Funktionsweise der periodischen Detektion wird in Kapitel 4.4 genauer erklärt.
Jeder go-Routine, welche durch “deadlock.NewRoutine()” kreiert wird, wird ein Struct zugeordnet, welches neben dem Index der Routine auch alle Informationen speichert, die für den Aufbau des Lock-Baums nötig sind. Eine genauere Beschreibung dieser Informationen befindet sich im Kapitel 4.3. Diese Routinen werden in einem globalen Slice “routines” gespeichert. Zusätzlich wird noch ein Dictionary “mapIndex” angelegt. Dieses ordnet für jede Routine dem Index dieser Routine die Position in “routines” zu. Diese Aufteilung macht es möglich auf eine Routine ohne langes suchen zuzugreifen. Da jede Routine nur auf sein eigenes Element in “routines” zugreift, ist es so möglich, auf mehrere Elemente in “routines” gleichzeitig zuzugreifen, ohne die gesamte Liste für andere Routinen blocken zu müssen. Dies wäre nicht möglich, wenn die Routinen direkt in einem Dictionary gespeichert werden würden.

4.3 Logging-Phase

Der Detektor implementiert Drop-in-Replacements für Locks, auf denen Lock, TryLock und Unlock Operationen definiert sind. Diese führen das eigentliche Locking aus, und speichern gleichzeitig Informationen, die für die Detektion von Deadlocks benötigt werden.
Ein Lock besteht aus einem “sync.Mutex” Objekt, mit dem das eigentliche Locking im Hintergrund ausgeführt wird, sowie einer Liste von “callerInfo” Objekten. Mit diesen werden Informationen darüber gespeichert, wo die Locks erzeugt wurden, und wo Logging Operationen

ausgeführt wurden. Diese Informationen werden verwendet wenn ein tatsächlich auftretendes oder potentielles Deadlock gefunden wird, um die in diesen Deadlocks involvierten Locks in der Fehlermeldung zu identifizieren. Neben der Information, ob es sich um eine Lock-Initialisierung oder einen Lock-Vorgang handelt, werden je nachdem wie die Optionen (vgl. 4.1) gesetzt sind entweder ein vollständiger Call-Stack oder lediglich Informationen über die Datei und Zeile, in denen der Vorgang im Programmcode vorkommt gespeichert.
Im folgenden sollen die drei auf Locks definierten Operationen genauer betrachtet werden.

4.3.1 Lock

Der Lock-Vorgang besteht aus zwei Schritten. Zum einen wird das tatsächlich sync.Mutex-Lock beansprucht. Zum anderen wird die Datenstruktur, welche später für die Detektion von Deadlocks verwendet wird aktualisiert. Dies geschieht allerdings nur, wenn momentan mehr als eine Routine läuft. UNDEAD geht davon aus, dass Deadlock hauptsächlich durch die Kombination von mehreren gleichzeitig laufenden Routinen entsteht. Solange demnach nur eine Routine läuft kann es nicht zu einem solchen Deadlock kommen, und die entsprechenden Datenstrukturen werden nicht aktualisiert, um die Performance des Programmes zu verbessern. Die einige Möglichkeit, bei der ein Deadlock auftreten kann, wenn insgesamt nur eine Routine läuft, besteht in doppeltem Locking, wenn also ein Lock von der selben Routine mehrfach beansprucht wird, ohne zwischendurch frei gegeben zu werden. Allerdings ist UNDEAD auch in Situationen, in dehnen mehrere Routinen laufen nicht in der Lage, solche Deadlocks zu erkennen (s. 4.4), auch wenn die Detektion nicht besonders kompliziert wäre.
Eine andere Situation, bei der dies zu einem False-Negativ führen könnte, ist folgende. Man betrachte eine Situation mit zwei Routinen R1 und R2, welche gleichzeitig starten. Man nehme nun an, dass in einem Durchlauf R2 bereits beendet wurde, während R1 noch läuft. In diesem Fall werden Lock-Operationen, in R1 nicht mehr betrachtet. Wenn R2 in einem erneuten Durchlauf nun allerdings länger benötigt als in dem ersten Durchlauf, können diese zuvor ignorierten Locks dennoch zu einem Deadlock führen, der im ersten Durchlauf aber nicht erkannt werden konnte.
Angenommen es laufen mehr als eine Routine und Deadlock-Detektion ist in den Optionen nicht vollständig deaktiviert, wird die Datenstruktur zur Deadlock-Detektion aktualisiert. Jede Routine verfügt dabei, wie bereits in 4.2 beschrieben, über eine eigene Datenstruktur, über die die entsprechenden Lock-Bäume aufgebaut werden. Eine Dependency wird dabei über ein Objekt dargestellt, welches eine Referenz zu einem Lock l, sowie eine Liste “holdingSet” aller Locks, welche von der Routine bereits gehalten wurden, als l erfolgreich beansprucht wurde.
Die Datenstruktur beinhaltet die folgenden Variablen:

- index: Jeder Routine wird ein Index zugewiesen.
- holdingCount: Diese Variable entspricht der Anzahl der momentan von der Routine gehaltenen Locks
- holdingSet: Dieses Array beinhaltet Referenzen zu allen von dieser Routinen momentan gehaltenen Locks.
- depCount: Dieser Integer beinhaltet die Anzahl der in dem Lock-Baum gehaltenen Dependencies, also die Anzahl der Kanten in dem Lock-Baum.
- dependencies: Über dieses Array wird der eigentliche Lock-Baum gehalten. Er besteht aus einer Liste aller in dem Baum gespeicherten Dependencies.
- dependencyMap: Dieses Dictionary wird verwendet, um möglichst effizient abzuschätzen, ob die momentane Situation bereits aufgetreten ist, z.B. wenn sich das Lock in einer Schleife befindet. Als Werte wird eine Liste mit Referenzen zu Dependencies gespeichert, die bereits betrachtet worden sind.
- curDep: Diese Variable beinhaltet eine Referenz zu der letzten eingefügten Dependency.
- collectedSingleLevelLocks: Dieses Dictionary beinhaltet callerInfo für singel-level Locks.

Eine Aktualisierung der Datenstruktur ist nur notwendig, wenn “holdingCount” größer 0 ist,

da es sonst keine neuen Dependencies geben kann. Ist eine Aktualisierung notwendig, so wird zuerst der Schlüssel für "dependencyMap" berechnet, der der aktuellen Situation entspricht. Der Schlüssel berechnet sich über eine xOr Operation zwischen der Speicherposition des momentanen Locks sowie des letzten in "currentHolding" eingefügten Locks. Existiert dieser Schlüssel bereits, wird betrachtet, ob eine Dependency mit dem selben Lock existiert, bei dem das "holdingSet" der Dependency dem momentanen "holdingSet" der Routine entspricht. Wenn diese nicht übereinstimmen, oder der Schlüssel nicht gefunden wurde, geht das Programm davon aus, dass die momentan auftretenden Dependencies zum ersten mal auftreten. Es wird dementsprechend eine neue Dependencies in "dependencies" eingefügt, deren Lock aus dem Lock, auf dem die Lock-Operation ausgeführt wird entspricht, und deren "holdingSet" eine Kopie des "holdingSet" der Routine ist. Zudem wird die neue Dependencies in die Liste der Dependencies der "dependencyMap" hinzugefügt, und "curDep" auf die neu erzeugte Dependency gesetzt. Ist dies der Fall, dann wird eine Variable isNew auf true gesetzt, um zu Signalisieren, dass diese Situation noch nicht bekannt ist. Dadurch kann sicher gestellt werden, dass für dieses Lock nicht erneut ein Call-Stack bzw. Caller Informationen gespeichert werden.
Es ist möglich in den Optionen einzustellen, dass Informationen über Lock Vorgänge auf single-level Locks gesammelt werden. Da auf single-level Locks keine Dependencies definiert werden, kann nicht über "depMap" entschieden werden, ob der Lock-Vorgang schon einmal betrachtet worden ist. Um zu verhindern, dass der entsprechende Call-Stacks bzw. Caller-Informationen mehrfach gespeichert werden, werden Caller-Informationen (Datei und Zeile in denen die Lock Operation ausgeführt wird) in einem Dictionary gespeichert. Bei einem solchen Aufruf wird nun überprüft, ob das Lock an der entsprechenden Stelle bereits einmal gelockt wurde. Ist dies der Fall, so wird isNew auf true gesetzt. Andernfalls wird der neue Lock Vorgang in das Dictionary eingefügt.
Wenn isNew auf true gesetzt wurde und entweder der holdingCount größer 0 ist oder die Sammlung von Caller-Informationen für single-Level Locks aktiviert ist, wird entweder der ganze CallStack oder lediglich die Datei und Zeilennummer der Lock Operation in der Liste der "callerInfo" Objekte in dem Lock eingefügt. Anschließend wird das Lock in "currentHolding", also die Liste der momentan gehaltenen Locks eingefügt und "holdingCount" entsprechend angepasst.

4.3.2 TryLock

TryLock versuchen ein Lock zu beanspruchen. Wenn es von keinem anderen Lock gehalten wird, geben sie true zurück und beanspruchen das entsprechende Lock. Ist es nicht möglich, dann geben sie false zurück. Anders als Lock warted das Programm in diesem Fall allerdings nicht, bis das Lock freigegeben wurde, sondern führt das Programm weiter aus, ohne das Lock beansprucht zu haben.
TryLocks werden für die Detektion nur betrachtet, wenn die Beanspruchung erfolgreich war. In diesem Fall wird das Lock lediglich in "holdingSet" eingefügt. Es werden allerdings nicht betrachtet, welche Locks bereits von der entsprechenden Routine gehalten werden und somit Dependencies erzeugen würden.

4.3.3 UnLock

In der Unlock Operation wird das Lock lediglich aus "holdingSet" entfernt, und "holdingCount" entsprechend angepasst. Anschließend wird das entsprechende sync.Mutex-Lock freigegeben.

4.4 Periodische Detektion

Die Aufgabe der periodischen Detektion besteht darin, tatsächlich auftretende Deadlocks zu erkennen, und das Programm in diesem Fall abzubrechen. Sie wird, wie in 4.2 bereits beschrieben in regelmäßigen zeitigen Abständen automatisch gestartet, solange sie nicht über die Optionen

deaktiviert wird. Sie wird außerdem nur dann durchgeführt, wenn mindestens 2 Routinen laufen. Der periodische Detektor besitzt ein Array "lastHolding", in dem für jede Routine, das Letzte erfolgreich beanspruchte Lock gespeichert wird. Bevor die eigentliche Detektion beginnt, werden die Werte dieser Liste für jede Routine mit den momentanen Werten für das zuletzt erfolgreich beanspruchte Lock verglichen. Gibt es keine Abweichungen, so nimmt der Detektor an, dass sich die situation seit der letzten periodischen Detektion nicht verändert hat, und führt keine erneute Detektion aus. Gibt es hingegen Abweichungen, so wird "lastHolding" aktualisiert und die eigentliche Detektion gestartet.
Diese läuft nicht auf den Dependencies der eigentlichen Lock-Bäumen ab, sondern nur auf den zuletzt in die Lock-Bäume eingefügten Dependencies "curDep". Diese sind zur Erkennung von Deadlocks ausreichend, die Betrachtung ist aber effizienter, da ihre Anzahl in der Regel geringer ist, als die Menge aller Dependencies. In diesen "curDep's" wird nun nach Zyklen gesucht, welche auf ein Deadlock hinweisen würden. Dazu wird ein Stack verwendet. Zuerst wird das "curDep" c_1 einer der Routine r auf den Stack gelegt. Das ganze wird insgesamt für alle Routinen als Startroutine wiederholt. Im nächsten Schritt werden alle Routinen betrachtet, deren Index größer ist, als der Index der Routine, dessen "curDep" c_1 an oberster Position im Stapel liegt. Es wird nun überprüft, ob sich mit dem "curDep" c_2 dieser Routinen eine Kette bilden lässt, also ob das Lock von c_1 in dem "holdingSet" von c_2 vorkommt. Wird ein solches "curDep" gefunden, wird überprüft, ob die gebildete Kette einen Zyklus bildet. Dies ist genau dann der Fall, wenn das Lock von c_2 in dem "holdingSet" der "curDep" auftaucht, welche an unterster Position im Stack liegt. Ist die Kette kein Zyklus, dann wird c_2 auf den Stapel gelegt und der Suchvorgang rekursiv wiederholt, nun aber mit c_2 als neuem obersten "curDep". Dies wird solange fortgesetzt, bis entweder ein Zyklus gefunden wurde, oder alle Routinen betrachtet wurden. Dabei wird darauf geachtet, dass nur Routinen betrachtet werden, deren "curDep" noch nicht auf dem Stapel liegen. Wenn ein Zyklus gefunden wurde geht der Detektor davon aus, dass sich das Programm in ein Deadlock befindet. Um die Wahrscheinlichkeit von False-Positives zu verringern wird nun noch einmal für alle in dem Zyklus auftauchenden Routinen überprüft, ob sich das letzte Lock in dem "holdingSet" der Routine verändert hat. Ist dies der Fall, dann geht der Detektor von einem falschen Alarm aus. Andernfalls wird eine Fehlermeldung ausgegeben, die auf den Deadlock hinweist und die gesammelten "callerInfo's" zu den entsprechenden Locks ausgibt. Anschließend wird noch die vollständige Detektion, wie in 4.5 beschrieben ausgeführt und das Programm anschließend abgebrochen.

4.5 Vollständige Detektion

Die vollständige Detektion wird nach Beendigung des Programms ausgeführt. Die Aufgabe dieses Teils besteht darin, den Lock-Baum, der in der Logging-Phase aufgebaut wurde nach potentiellen Deadlocks zu durchsuchen. Da UNDEAD sowieso nicht in der Lage ist Deadlocks zu erkennen, die in nur einer Routine passieren (z.B. double locking), wird der Detektor nur ausgeführt, wenn in dem Programm mindestens zwei Routinen aufgetaucht sind. Außerdem zählt der Detektor, wie viele einzigartige Dependencies es in der Menge aller Lock-Bäume ist. Wenn es weniger als zwei einzigartige Routinen gibt, ist es ausgeschlossen, das der Detektor dort ein potentielles Deadlock findet. In diesem Fall wird die Suche abgebrochen.
Die Suche nach potentiellen Deadlocks läuft nahezu identisch ab wie die periodische Detektion in 4.4. Allerdings werden hierbei nicht nur die zuletzt eingefügten, sondern alle in dem Lock-Baum enthaltenen Dependencies betrachtet. Außerdem wird die Suche nicht bei dem ersten gefundenen Deadlock abgebrochen, sondern fortgeführt bis alle möglichen Pfade betrachtet wurden.

4.6 Meldung

Wird ein tatsächlicher oder potentieller Deadlock gefunden, wird dem Nutzer eine entsprechende Nachricht ausgegeben. Beide Methoden nutzen einen Stack um nach Zyklen zu suchen. Wenn ein solcher Kreis auftritt, sind die Locks, der in dem Stack gerade gehaltenen Dependencies gerade die Locks, die in dem Deadlock involviert sind. Da die Caller-Informationen bzw. Stack-Traces direkt in den Locks gespeichert werden, können diese Informationen nun ausgegeben werden, um dem Nutzer die Erkennung der (potentiellen) Deadlocks zu ermöglichen. Dabei wird zuerst ein Liste der Initialisierungen der Locks ausgegeben. Anschließend werden nach Locks sortiert die gesammelten Informationen über die Lock-Beanspruchungen ausgegeben.

5 Analyse der Implementierung und Vergleich zwischen 3.1

Im folgenden soll der in 4 beschriebene und in [6] implementierte Detektor analysiert und anschließend mit dem in 3.1 betrachteten Detektor verglichen werden.

Literatur

[1] J. Zhou, S. Silvestro, H. Liu, Y. Cai und T. Liu, „UNDEAD: Detecting and preventing deadlocks in production software," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Los Alamitos, CA, USA: IEEE Computer Society, Nov. 2017, S. 729–740. DOI: 10.1109/ASE.2017.8115684.

[2] M. Sulzmann, *Dynamic deadlock prediction*, https://sulzmann.github.io/AutonomeSysteme/lec-deadlock.html. (besucht am 03. 05. 2022).

[3] S. Bensalem und K. Havelund, „Dynamic Deadlock Analysis of Multi-threaded Programs," in *Hardware and Software, Verification and Testing*, S. Ur, E. Bin und Y. Wolfsthal, Hrsg., Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, S. 208–223, ISBN: 978-3-540-32605-2. DOI: 10.1007/11678779_15.

[4] sasha-s, *go-deadlock*, https://github.com/sasha-s/go-deadlock, 2018. (besucht am 03. 05. 2022).

[5] The Go Authors, *Go Runtime Library src/runtime/proc.go*, https://go.dev/src/runtime/proc.go#L4935. (besucht am 10. 05. 2022).

[6] E. Kassubek, *Deadlock-Go*, https://github.com/ErikKassubek/Deadlock-Go, 2021. (besucht am 08. 06. 2022).