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

$$(l_1, l_2) \in E \Leftrightarrow \exists t_1, t_3 \not\exists t_2 \ ((t_1 < t_2 < t_3) \wedge aqr(l_1)[t_1] \wedge rel(l_1)[t_2] \wedge aqr(l_2)[t_3])$$

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

Funktionsweis von Lock-Bäumen

2.2.3 UNDEAD

Funktionsweise UNDEAD

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 Methoden angewendet, die im folgenden betrachtet werden sollen.

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 eine Liste $l.cur$ mit allen momentan gehaltenen Locks. Dabei wird neben der Information, welches Lock gehalten wird auch gespeichert, von welchem Thread das Lock gehalten wird. 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 Inkonsistentes Locking

Mit dieser Methode werden möchte 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.
Die Detection funktioniert über einen Lockgraphen. Dieser wird über ein Dictionary gehandhabt. Wird ein neuer neues Lock p beanspruchen, so wird für jedes Lock b, welches bereits gehalten wird ein neuer Eintrag $beforeAfter\{b, p\}$ in das Dictionary $l.order$ hinzugefügt. Der Key gibt somit die beiden Locks an und speichert auch, welches der Beiden Locks zuerst beansprucht wurde. Die Werte, die dem Key zugeordnet sind, beinhalten Information darüber, über welche Funktion, bzw. Routine das Lock beansprucht wurde. Um zu überprüfen ob ein neues Lock zu einem Deadlock führen kann, wird überprüft, ob $l.order$ einen Key $beforeAfter\{p, b\}$ besitzt, der die beiden Locks somit in umgekehrter Reihenfolge enthält. Existiert solch ein Key, 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. 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

Neben den bereits in den vorherigen Abschnitten betrachteten Fällen, bei denen ein möglicher Deadlock nicht erkannt wurde, gibt es noch andere Fälle, bei denen dies der Fall ist. Ein Fall, bei dem ein Deadlock auftreten könnte, welcher von dem Programm aber nicht erkannt wird, 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.

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

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).