Sound Dynamic Deadlock Prediction in Linear Time

Florian Rudaj

Agenda

- Einleitung
- Die Vorhersage von Deadlocks
- Dynamische Deadlock-Analyse
- Sync-preserving Deadlocks
- Fazit

Einleitung

- In nebenläufigen Programmen müssen Ressourcen geteilt werden
- Gleichzeitige Zugriffe auf geteilte Ressourcen können zu Inkonsistenzen führen
- Mutex und Locks dieser als Lösung

Einleitung

- Deadlocks sind schwer zu verhindern
- Aber daraus entstehende Problematik: Deadlocks
- Deadlocks sind schwer reproduzierbar und damit schwer zu debuggen

Einleitung

- Programm enthält Deadlock
- Tritt auf wenn beide Threads den jeweils ersten Mutex locken
- Der jeweils zweite Lock kann nicht reserviert werden, da er bereits belegt ist

```
func simple_deadlock() {
    x := 0
    y := 0
    var xMutex sync.Mutex
    var yMutex sync.Mutex
    go func() {
        xMutex.Lock()
        yMutex.Lock()
        x, y = doWork(x, y)
        yMutex.Unlock()
        xMutex.Unlock()
    }()
    yMutex.Lock()
    xMutex.Lock()
    x, y = doOtherWork(x, y)
    xMutex.Unlock()
    yMutex.Unlock()
}
```

Die Vorhersage von Deadlocks

- Deadlocks müssen vorhergesagt werden, ansonsten können Threads stecken bleiben
- Dynamische Deadlock-Analyse
  - Effizient
  - Liefert wenige oder keine False-Positives

Dynamische Deadlock-Analyse

- Grundlage: Trace
- Trace ist Aufzeichnung der abgelaufenen Operationen im Programm
- Dazu gehört: Acquire- und Release von Locks
- Je nach Methode auch Reads/Writes oder auch Forks/Joins relevant

Dynamische Deadlock-Analyse

```go
func simple_deadlock() {
    x := 0
    y := 0
    var xMutex sync.Mutex
    var yMutex sync.Mutex
    go func() {
        xMutex.Lock()
        yMutex.Lock()
        x, y = doWork(x, y)
        yMutex.Unlock()
        xMutex.Unlock()
    }()
    yMutex.Lock()
    xMutex.Lock()
    x, y = doOtherWork(x, y)
    xMutex.Unlock()
    yMutex.Unlock()
}
```

	T1	T2
1		
2	acq(y)	
3	acq(x)	
4	rel(x)	
5	rel(y)	
6		acq(x)
7		acq(y)
8		rel(y)
9		rel(x)

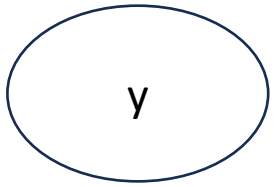
Dynamische Deadlock-Analyse

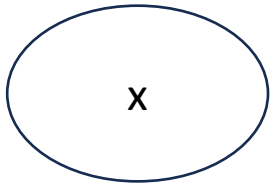
- Analyse durch Lock-Graphen sehr einfach
- Schritt 1: Erstelle Graph aus Acquire- und Releaseoperationen:
  - Locks sind Knoten
  - Kanten zwischen Knoten entstehen, wenn ein Thread einen Lock hält und den nächsten reservieren will
- Schritt 2:
  - Überprüfe den Graphen auf Zyklen
  - Sage Deadlock vorher, wenn Zyklus im Graph existiert

Dynamische Deadlock-Analyse

	T1	T2
1		
2	acq(y)	
3	acq(x)	
4	rel(x)	
5	rel(y)	
6		acq(x)
7		acq(y)
8		rel(y)
9		rel(x)

Es gibt die Locks x und y im Trace -> Knoten x und y im Graph

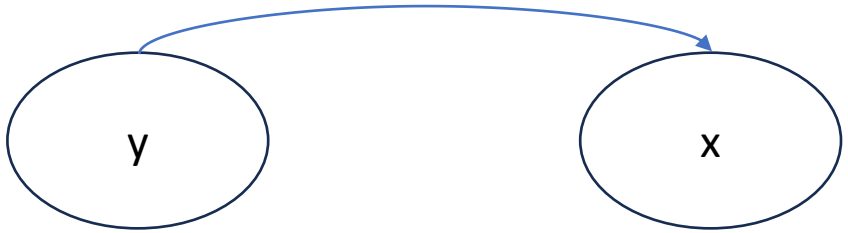




Dynamische Deadlock-Analyse

```
1        T1              T2
2    acq(y)
3    acq(x)
4    rel(x)
5    rel(y)
6                    acq(x)
7                    acq(y)
8                    rel(y)
9                    rel(x)
```

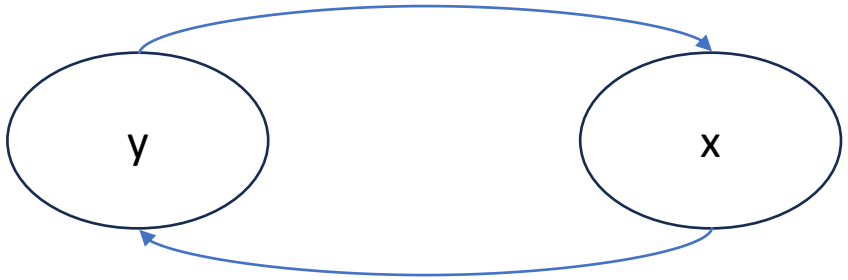
Es gibt die Locks x und y im Trace
➔ Knoten x und y im Graph
In T1 wird nach Lock y Lock x reserviert
➔ Kante von y nach x



Dynamische Deadlock-Analyse

	T1	T2
1		
2	acq(y)	
3	acq(x)	
4	rel(x)	
5	rel(y)	
6		acq(x)
7		acq(y)
8		rel(y)
9		rel(x)

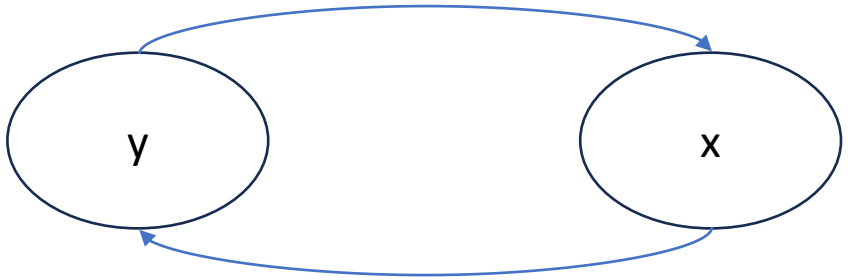
Es gibt die Locks x und y im Trace
➔ Knoten x und y im Graph
In T1 wird nach Lock y Lock x reserviert
➔ Kante von y nach x
In T2 wird nach Lock x Lock y reserviert
➔ Kante von x nach y



Dynamische Deadlock-Analyse

```
1        T1              T2
2     acq(y)
3     acq(x)
4     rel(x)
5     rel(y)
6                   acq(x)
7                   acq(y)
8                   rel(y)
9                   rel(x)
```

Es gibt die Locks x und y im Trace
➔ Knoten x und y im Graph
In T1 wird nach Lock y Lock x reserviert
➔ Kante von y nach x
In T2 wird nach Lock x Lock y reserviert
➔ Kante von x nach y

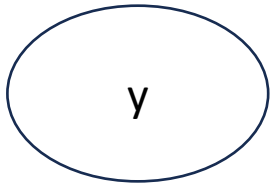


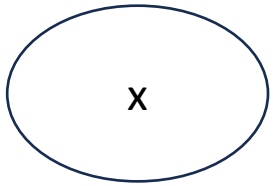
➔ Der resultierende Graph enthält einen Zyklus, also wird ein Deadlock vorhergesagt.

Dynamische Deadlock-Analyse

	T1	T2
1		
2	acq(y)	
3	acq(x)	
4	rel(x)	
5	rel(y)	
6	acq(x)	
7	acq(y)	
8	rel(y)	
9	rel(x)	

Es gibt die Locks x und y im Trace
➔ Knoten x und y im Graph

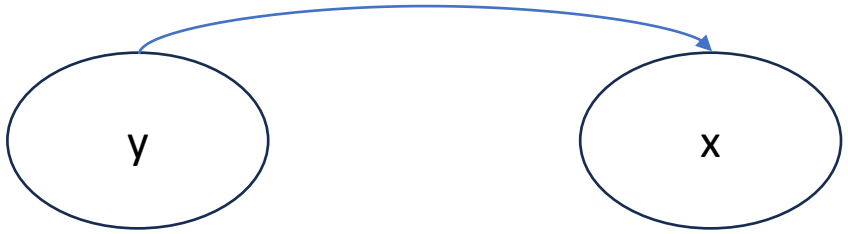




Dynamische Deadlock-Analyse

	T1	T2
1		
2	acq(y)	
3	acq(x)	
4	rel(x)	
5	rel(y)	
6	acq(x)	
7	acq(y)	
8	rel(y)	
9	rel(x)	

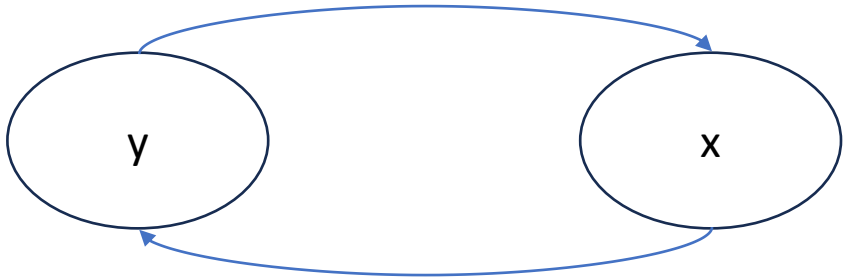
Es gibt die Locks x und y im Trace
➔ Knoten x und y im Graph
In T1 wird nach Lock y Lock x reserviert
➔ Kante von y nach x



Dynamische Deadlock-Analyse

	T1	T2
1		
2	acq(y)	
3	acq(x)	
4	rel(x)	
5	rel(y)	
6	acq(x)	
7	acq(y)	
8	rel(y)	
9	rel(x)	

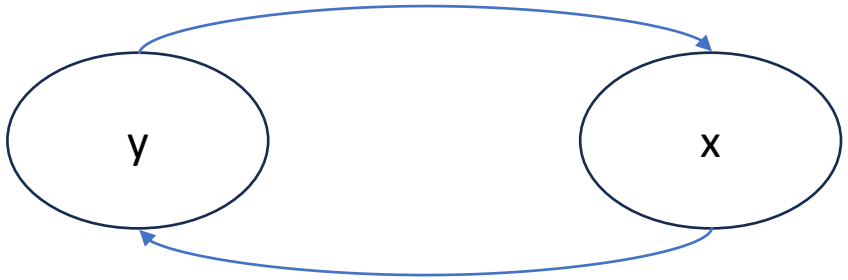
Es gibt die Locks x und y im Trace
➔ Knoten x und y im Graph
In T1 wird nach Lock y Lock x reserviert
➔ Kante von y nach x
In T1 wird nach Lock x Lock y reserviert
➔ Kante von x nach y



Dynamische Deadlock-Analyse

	T1	T2
1		
2	acq(y)	
3	acq(x)	
4	rel(x)	
5	rel(y)	
6	acq(x)	
7	acq(y)	
8	rel(y)	
9	rel(x)	

Es gibt die Locks x und y im Trace
➔ Knoten x und y im Graph
In T1 wird nach Lock y Lock x reserviert
➔ Kante von y nach x
In T1 wird nach Lock x Lock y reserviert
➔ Kante von x nach y



➔ Der resultierende Graph enthält einen Zyklus, also wird ein Deadlock vorhergesagt.
➔ False-Positive, da alle Events im gleichen Trace

Sync-preserving Deadlocks

- False-Positives sind wegen des daraus folgenden Aufwands unerwünscht
- Sync-preserving Deadlocks sind Untermenge aller Deadlocks
- Meiste in der Praxis vorkommenden Deadlocks sind Sync-preserving Deadlocks
- Lassen sich ohne False-Positives vorhersagen

Sync-preserving Deadlocks

- Für Vorhersage: Umordnung der Events im Original-Trace
- Dafür Sync-preserving Correct Reordering

Sync-preserving Deadlocks

Deadlock-Patterns:

Sind Sequenzen $D = < e_0, e_1, \dots, e_{k-1} >$ mit k eigenständigen Threads $t_0, \dots, t_{k-1}$ und k eigenständigen Locks $l_0, \dots, l_{k-1}$, sodass:

- thread(e_i) = t_i
- op(e_i) = acq(l_i)
- $l_i \in HeldLks_\delta(e_{(i+1)\%k})$
- $HeldLks_\delta(e_i) \cap HeldLks_\delta(e_j) = \emptyset$

Ein Deadlock-Pattern ist eine notwendige aber keine hinreichende Bedingung für einen tatsächlichen Deadlock

Sync-preserving Deadlocks

- Es werden nur wohlgeformte Traces betrachet
- Wohlgeformte Traces unterliegen der Lock-Semantik, die sich wie folgt definiert:

  Wenn ein Lock l bei einem Event e von Thread t reserviert wird, dann muss jede spätere Reservierung von einem Event e' desselben Locks l ein Vorgängerevent e'' haben, welches den Lock l in Thread t zwischen e und e' freigibt. Wenn e'' das frühste Release-Event ist sind e und e'' passende Acquire- und Release-Events. Dies wird durch $e = match_{\delta}(e'')$ bezeichnet.

Sync-preserving Deadlocks

Eine Umordnung ρ vom Trace δ ist ein Correct Reordering wenn folgende Regeln eingehalten werden:

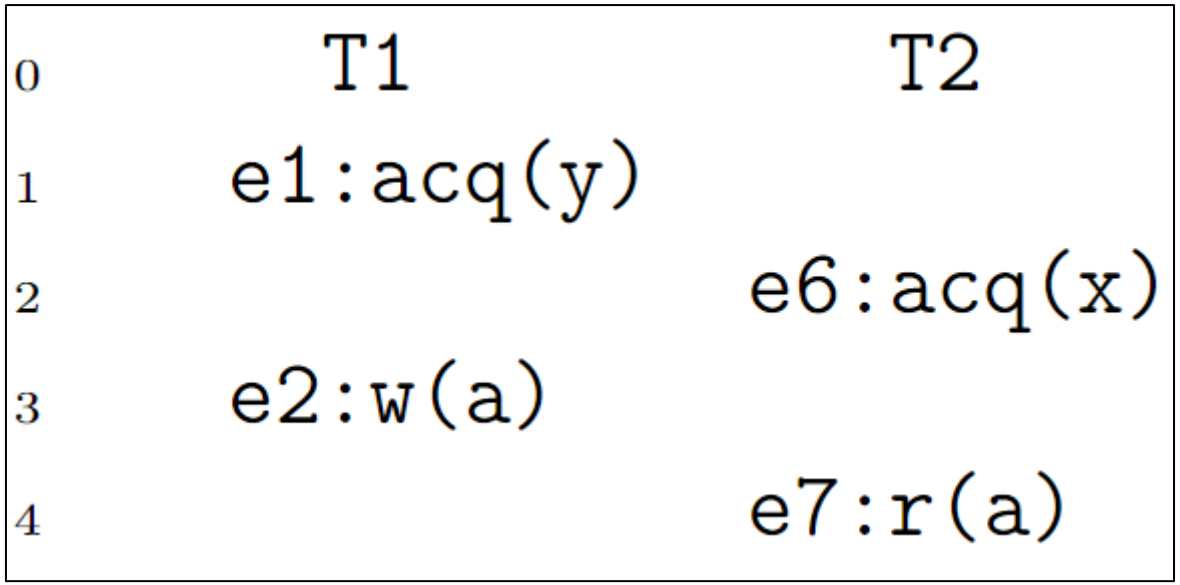
- **Subset**: $Events_\rho \subseteq Events_\delta$
- **Thread-Order**: Für jede $e, f \in Events_\delta$ mit $e \leq_{TO}^{\delta} f$ wenn $f \in Events_\rho$, dann $e \in Events_\rho$ und $e \leq_{TO}^{\rho} f$
- **Last-Write**: Für jedes Read-Event $r \in Events_\rho$ haben wir rf(r)$\in$ $Events_\rho$ und rfρ(r) = rfδ(r)

Sync-preserving Deadlocks

- Deadlock-Pattern: $< e_3, e_8 >$
- Lock-Semantik besteht

	T1	T2
0		
1	acq(y)	
2	w(a)	
3	acq(x)	
4	rel(x)	
5	rel(y)	
6		acq(x)
7		r(a)
8		acq(y)
9		rel(y)
10		rel(x)

Sync-preserving Deadlocks



- **Subset**: $Events_\rho \subseteq Events_\delta$
- **Thread-Order**: Für jede $e, f \in Events_\delta$ mit $e \leq^{\delta}_{TO} f$ wenn $f \in Events_\rho$, dann $e \in Events_\rho$ und $e \leq^{\rho}_{TO} f$
- **Last-Write**: Für jedes Read-Event $r \in Events_\rho$ haben wir $rf(r) \in Events_\rho$ und $rf_\rho(r) = rf_\delta(r)$

Sync-preserving Deadlocks

- Damit ein Correct Reordering Sync-preserving ist, müssen alle Acquire-Events auf denselben Lock in der gleichen Reihenfolge sein wie im Original-Trace.

Sync-preserving Deadlocks

	T1	T2
0		
1	acq(y)	
2	w(a)	
3	acq(x)	
4	rel(x)	
5	rel(y)	
6		acq(x)
7		r(a)
8		acq(y)
9		rel(y)
10		rel(x)

	T1	T2
0		
1	e1:acq(y)	
2		e6:acq(x)
3	e2:w(a)	
4		e7:r(a)

➔ Das Correct Reordering ist auch sync-preserving!

Sync-preserving Deadlocks

	T1	T2
0		
1	e1:acq(y)	
2		e6:acq(x)
3	e2:w(a)	
4		e7:r(a)
5	e3:acq(x)	
6		e8:acq(y)

Sync-preserving Deadlocks

- Wie findet man nun ein Sync-preserving Correct Reordering für ein Deadlock Pattern?
- Problem: Events und deren Anordnung müssen gefunden werden
- Sync-preserving Closure reduziert dieses Problem auf die Überprüfung, ob eine wohldefinierte Menge an Events ein Event aus dem Deadlock-Pattern enthält

Sync-preserving Deadlocks

Definition Sync-preserving Closure:

Zu einem Trace δ und einem $S \subseteq Events_\delta$ ist die Sync-preserving Closure die kleinste Menge S' ,sodass:

a) $S \subseteq S'$

b) Für jedes $e, e' \in Events_\delta$ gilt, dass $e \leq_{TO}^{\delta} e'$ oder $e = rf_\delta(e')$, wenn $e' \in S'$, dann $e \in S'$

c) Für jeden Lock l jede zwei eigenständige Events $e, e' \in S'$ mit op(e) = op(e') = acq(l), wenn $e \leq_{tr}^{\delta} e'$ dann $match_\delta(e) \in S'$s

Sync-preserving Deadlocks

	T0	T1	T2
e1.	fork(T2)		
e2.			acq(z)
e3.			acq(y)
e4.			acq(x)
e5.			rel(x)
e6.			rel(y)
e7.			rel(z)
e8.	acq(z)		
e9.	fork(T1)		
e10.		acq(x)	
e11.		acq(y)	
e12.		rel(y)	
e13.		rel(x)	
e14.	join(T1)		
e15.	rel(z)		

Deadlock-Pattern: <e4, e11>
S = {e3, e10}
S‘ = SPClosure(S)
S‘ = {e3, e10}
S‘ = {e3, e2, e10, e9, e8} wegen Thread-Order
➔ Zwei Acq-Events für Lock z e2 und e8
➔ Nach Lock-Semantik muss e7 hinzugefügt werden
S‘ = {e3, e2, e10, e9, e8, e7}
S‘ = {e3, e2, e10, e9, e8, e7, e6, e5, e4} wegen Thread-Order
➔ e4 aus dem Deadlock Pattern ist in S‘
➔ Kein Sync-preserving Deadlock

Fazit

- Die Vorhersage von Sync-preserving Deadlocks kann ohne False-Positives geschehen
- Sync-preserving Deadlocks decken die meisten in der Praxis auftretenden Deadlocks ab
- False-Negatives können dennoch auftreten