Effizientes Auffinden von Nebenläufigkeitsfehlern durch statische Codeanalyse

Die zunehmende Verbreitung moderner Multicore-Prozessoren ist mit der Forderung nach nebenläufiger Software verbunden, die derartige Hardwarearchitekturen optimal auszunutzen versteht. In nicht unerheblichem Maße gilt das auch im embedded Umfeld. Softwareentwickler und –Tester müssen sich neuen Herausforderungen stellen. Kann die Softwarequalität von single-threaded Applikationen durch Code-Review und dynamische Testverfahren noch hinreichend gesichert werden, reichen diese Testmethoden für multithreaded Applikationen meist allein nicht mehr aus.

Was sind Nebenläufigkeitsfehler (Concurrency Bugs)?

Bei der Implementierung von nebenläufigen Applikationen bedürfen Ressourcen, die von verschiedenen Threads gemeinsam genutzt werden, der besonderen Aufmerksamkeit. Zu unerwartetem Laufzeitverhalten kann es sowohl bei einer fehlenden Koordinierung der Zugriffe (Synchronisierung) als auch bei einer fehlerhaft ausgeführten Zugriffssteuerung kommen.
Fehlt die Synchronisierung, kann das eine „Race Condition“ zur Folge haben. Fehlerbehaftete „Locking“-Mechanismen vermögen oft Race Conditons nicht wirksam zu verhindern und bergen zudem das Risiko von „Deadlocks“.

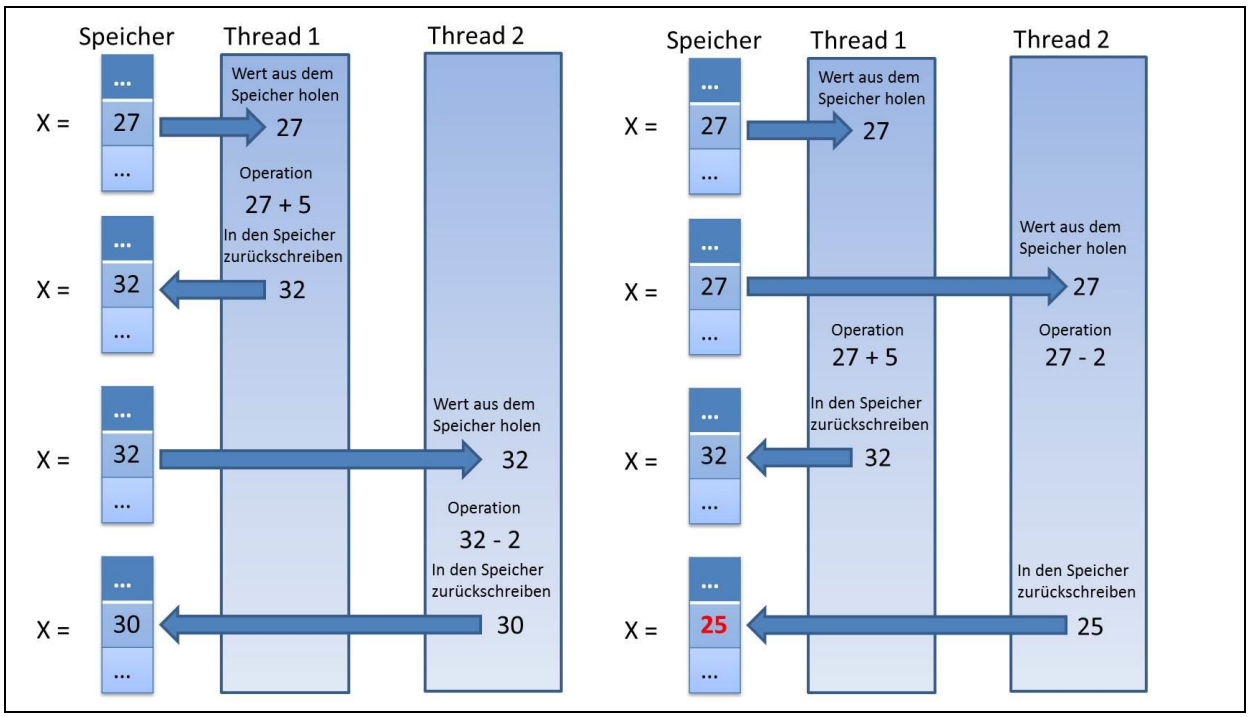

Abb. 1: Data Race

Abb. 1 erläutert das Auftreten einer Race Condition: Thread 1 hat die Aufgabe, den Wert der Variablen X um 5 zu erhöhen. Thread 2 dagegen dekrementiert X um 2. Links ist der vom Programmierer gewünschte Ablauf dargestellt. Thread 1 holt sich den Wert für X aus dem Speicher, inkrementiert diesen um 5 und schreibt das Ergebnis (hier 32) wieder in den Speicher zurück. Im Anschluss liest Thread 2 den Wert für X um diesen dann zu dekrementieren und im Speicher abzulegen. Im Beispiel ist das erwartete Ergebnis 30.

Werden keine Maßnahmen getroffen, um die oben beschriebene Reihenfolge der Speicherzugriffe zu gewährleisten, ist ein Ablauf wie rechts in der Abbildung dargestellt möglich:
Thread 1 und Thread 2 greifen beide lesend auf den Speicher zu und arbeiten beide mit dem gleichen Wert X = 27. Thread 1 inkrementiert den Wert und schreibt als Ergebnis X = 32 in den Speicher. Thread 2 dekrementiert den Wert für X um 2 und überschreibt mit dem Ergebnis 25 den von Thread 1 bereits berechneten Wert im Speicher. Die von Thread 1 durchgeführte Operation ist also praktisch wirkungslos und das Ergebnis weicht mit 25 vom erwarteten Wert 30 ab. Der Fehler ist die Folge einer Race Condition.

Die Lösung für das Problem heißt Synchronisation. Oft wird dafür ein Locking-Mechanismus verwendet. Ein Thread, der auf eine von mehreren Threads gemeinsam genutzte Ressource zugreift, belegt diese mit einer Sperre für alle anderen Threads und löst sie erst wieder, wenn das Rückschreiben in den Speicher abgeschlossen ist.
Problematisch sind stets verschachtelte Locks. Wurde deren Reihenfolge unbedacht gewählt, können sich Threads gegenseitig derart blockieren, dass ein Deadlock entsteht.

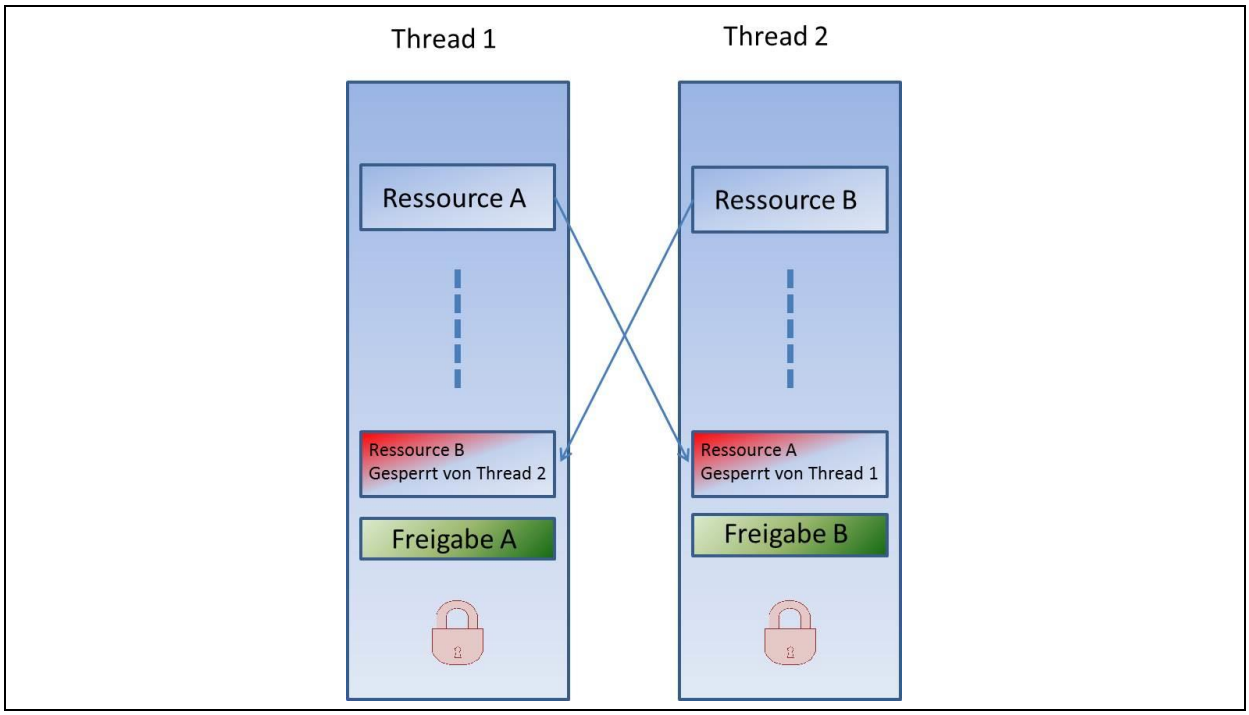

Abb. 2: Deadlock

Abb. 2 zeigt eine typische Deadlock-Situation. Thread 1 beansprucht die Ressource A und sperrt diese für Thread 2. Thread 2 hat seinerseits Ressource B für Thread 1 gesperrt. Beide Threads warten

jeweils vergeblich auf die gegenseitige Freigabe der Ressourcen um weiterarbeiten zu können. Die notwendigen Freigabebefehle dazu bleiben aber unerreichbar.
Ein Codebeispiel für einen Deadlock findet sich später im Verlauf dieser Ausführungen.

Nicht immer sind die Softwareentwickler Schuld an der Entstehung von Concurrency Bugs. Nebenläufigkeitsfehler können auch durch unbedachte Compilerwahl oder -Einstellungen entstehen. So wurde Multithreading für die Programmiersprache C erst offiziell mit dem Standard ISO/IEC 9899:2011 [1] eingeführt. Ein Build mit einem noch auf dem ISO-Standard ISO/IEC 9899:1999 [2] basierenden C-Compiler, der ein multithreaded Programm auf einen singlethreaded Ablauf hin optimiert, kann sich unter Umständen katastrophal auswirken.

Warum sind Nebenläufigkeitsfehler so gefürchtet?

Nebenläufigkeitsfehler sind überwiegend schwer aufzuspüren. Eine Ursache dafür ist, dass der Ablauf einzelner Threads durch ein von einer Vielzahl von Parametern abhängiges Scheduling gesteuert wird und derartige Fehler daher oft nicht deterministisch auftreten und somit dann auch nicht reproduzierbar sind.
Nicht selten läuft eine Applikation über Wochen stabil und fehlerfrei, liefert plötzlich ein fehlerhaftes Ergebnis, um dann wieder für eine lange Zeit unauffällig zu bleiben.

Erschwerend kommt hinzu, dass die meisten dynamischen Testverfahren, die bei single-threaded Applikationen stets gute Dienste leisten, hier hoffnungslos versagen. Invasives Vorgehen, wie z.B. in den Code eingefügte „printf()“ -Anweisungen oder auch nur an den Prozess angebundene Debugger sowie Analysetools ändern das Zeitverhalten derart, dass sich ein sporadisch auftretender Fehler schlimmstenfalls gar nicht mehr zeigt.
Aus diesem Grund werden solche Fehler auch in Anlehnung an die Heisenberg’sche Unschärferelation als „Heisenbugs“ bezeichnet.

Kostenrisiko Nebenläufigkeitsfehler

Abgesehen davon, dass Softwarefehler im Produktivbetrieb sogar Menschenleben gefährden können, sind die Kosten zur Beseitigung von Fehlern in dieser Phase des Softwarelebenszyklus am höchsten. Die relativen Kosten zur Fehlerbeseitigung in Abhängigkeit zur Projektphase folgen der im Qualitätsmanagement bekannten „Zehnerregel“. Eingehend behandelt wird der Zusammenhang z. B. von B. Boehm [3].
Je später ein Softwarefehler im Entwicklungsprozessverlauf beseitigt wird, desto höher sind die entstehenden Kosten.

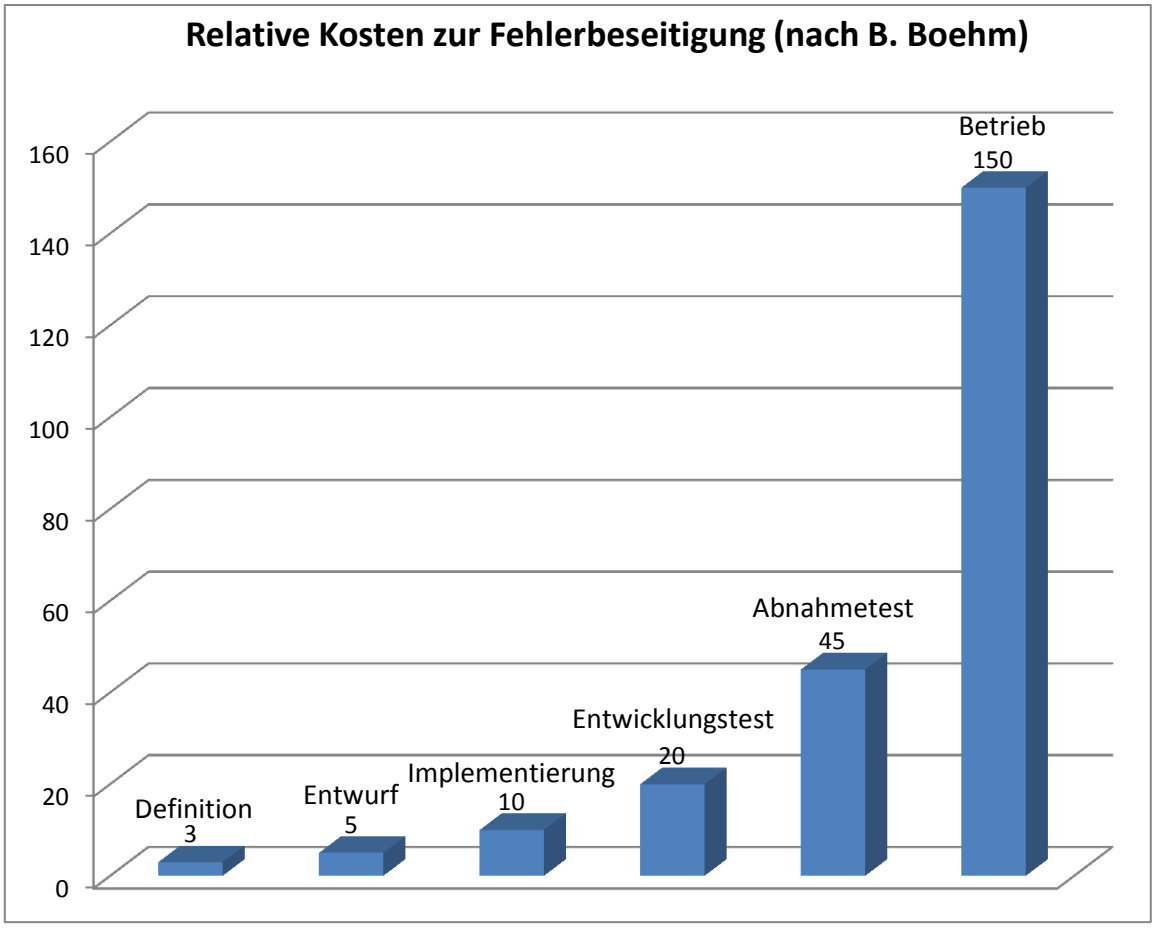

Abb. 3: Relative Kosten zur Fehlerbeseitigung für ein Projekt mittlerer Größe

Insbesondere Race Conditions sind aus bereits beschriebenen Gründen nur schwer aufzufinden und werden deshalb in der Praxis überwiegend erst spät offenbar. Sie zählen daher zu den großen Risikofaktoren eines jeden Softwareprojektes, das Nebenläufigkeit implementiert.

Welche Werkzeuge und Vorgehensweisen bieten die größten Erfolgsaussichten zum Auffinden von Nebenläufigkeitsfehlern?

Wie bereits erwähnt, sind Nebenläufigkeitsfehler mit Debuggern nicht immer zu triggern. Am erfolgversprechendsten bei sporadisch auftretenden Fehlern sind hier noch „Unattended Parallel Debugger“, die sich über eine lange Zeit an einen Prozess anbinden lassen und im Fehlerfall ein Protokoll erstellen. Oftmals ist es darüber hinaus nötig den Verlauf der Debug-Session aufzuzeichnen, um im Fehlerfall dessen Ursache in der Vergangenheit ausmachen zu können. Stellt sich der Fehler allerdings als Heisenbug dar, müssen all diese Bemühungen erfolglos bleiben.

Aussicht auf Erfolg kann eine genaue Sichtung des Quellcodes bieten. Allerdings sind nebenläufige Applikationen meist sehr komplex und logisch zusammengehörige Codeabschnitte häufig über mehrere Dateien verteilt. Im Rahmen eines Code-Reviews z.B. potentielle Race Conditions oder „Deadlocks“ nicht zu übersehen, ist auch für erfahrene Tester nicht einfach.

Wurden ausnahmslos Locking-Mechanismen zur Synchronisierung verwendet, können im Rahmen einer dynamischen Analyse Eraser-Algorithmen zur Anwendung kommen. Diese können überprüfen,

ob Zugriffe auf eine Ressource, die von mehreren Threads gemeinsam genutzt wird, stets durch Locks synchronisiert sind [4].

Als besonders effizient hat sich die statische Codeanalyse erwiesen, auf die im Folgenden noch genauer eingegangen wird.

Auffinden von Nebenläufigkeitsfehlern im Rahmen der statischen Codeanalyse

Werkzeuge zur statischen Codeanalyse sind bereits sehr frühzeitig im Softwareentwicklungsprozess einsetzbar und besonders leicht anzuwenden. Je nach Qualität vermögen sie eine Vielzahl von unterschiedlichen Fehlern aufzudecken. Im Gegensatz zu dynamischen Analysetools muss die Applikation nicht zum Ablauf gebracht werden und das aufwändige Erstellen von Testfällen entfällt. Es reicht aus, wenn das Programm fehlerfrei kompilierbar ist.
Der Quellcode wird lediglich durch einen Parser gelesen, um ihn dann in einer Art Simulation symbolisch zum Ablauf zu bringen und unter unterschiedlichen Kriterien untersuchen zu können. Statische Analyseläufe können daher vom Programmierer bereits während der Codierung zur Untersuchung von Teilimplementierungen einer größeren, noch unfertigen Applikation herangezogen werden. Die Aufwände zur Fehlerbeseitigung verschieben sich damit in den kostengünstigen, frühen Bereich.
Insbesondere diese Eigenschaft zeichnet die statische Analyse im Hinblick auf das Auffinden von Nebenläufigkeitsfehlern aus.
Nicht verschwiegen werden darf an dieser Stelle allerdings die Tatsache, dass es gravierende Unterschiede in den verwendeten Laufzeitmodellen der verschiedenen, auf dem Markt erhältlichen Tools gibt. Sind diese zu optimistisch, werden viele Nebenläufigkeitsfehler nicht entdeckt (False Negatives). Umgekehrt sorgen zu pessimistische Ansätze für viele „Falschmeldungen“ (False Positives). Hier gilt es die richtige Balance zu finden.

Der Einsatz eines hochwertigen Werkzeuges zur statischen Codeanalyse lohnt sich für die professionelle Softwareentwicklung praktisch immer und ist für die Entwicklung kritischer Applikationen unverzichtbar. Ist z.B. eine Zertifizierung der Applikation nach ISO 26262-6 (Automotive) gefordert, so schreibt diese die statische Codeanalyse sogar zwingend vor.

Beispiel zur Auffindung einer Race Condition mittels statischer Codeanalyse

Das Listing in Abb. 4 zeigt ein „Data Race“ als typisches Beispiel für eine Race Condition. Berechnet wird die Kreiszahl π iterativ nach der Trapezregel. Die Arbeit teilen sich zwei Threads. Auf ein Locking der Variablen „sum“ wurde hier verzichtet (die Zeilen zur Synchronisierung sind auskommentiert). Sporadisch auftretende, fehlerhafte Ergebnisse sind die Folge. Wird die Anzahl der Iterationen hinreichend klein gewählt, kann das bedingt durch die Zeit, die zur Erstellung eines Threads benötigt wird, zu einer sequentiellen Abarbeitung führen. Thread 1 hat dann die Arbeit bereits beendet, bevor Thread 2 seine aufnimmt. Eine wirkliche Nebenläufigkeit findet nicht statt. Der Fehler bleibt so womöglich über eine lange Zeit verborgen.

Zur statischen Codeanalyse wird ein leistungsfähiges Tool eingesetzt, das im Hinblick auf das gewählte, einfache Beispiel sicherlich unterfordert ist [5].

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>


pthread_mutex_t mutex0 = PTHREAD_MUTEX_INITIALIZER;

double sum = 0.;



void *calculate(void *params){
        double d, w;
        unsigned long l;

        unsigned long* part = (unsigned long*) params;
        d = 1.0 / part[2];
        for (l = part[0]; l < part[1]; ++l){
                w = d * (l + 0.5);
/*              pthread_mutex_lock(&mutex0); */
                        sum += 4.0 / (1.0 + w * w);
/*              pthread_mutex_unlock(&mutex0); */
        }
}



int main(void){

        pthread_t pth1, pth2;
        unsigned long segment_info0[3] = {0, 5000000, 10000000};
        unsigned long segment_info1[3] = {5000001, 10000000, 10000000};

        if (pthread_create(&pth1, NULL, calculate, (void*) &segment_info0) != 0){
                printf("Thread creation failed!\n");
                exit (EXIT_FAILURE);
        }

        if (pthread_create(&pth2, NULL, calculate, (void*) &segment_info1) != 0){
                printf("Thread creation failed!\n");
                exit (EXIT_FAILURE);
        }

        pthread_join(pth1, NULL);
        pthread_join(pth2, NULL);

        printf("PI = %25.20lf\n", sum / segment_info0[2]);

        return 0;
}
```

Abb. 4: Race Condition (Data Race)

Erwartungsgemäß deckt das Analysewerkzeug die Race Condition zuverlässig auf. Die Abb. 5 zeigt einen Auszug aus der entsprechend ausgegebenen Warnung.

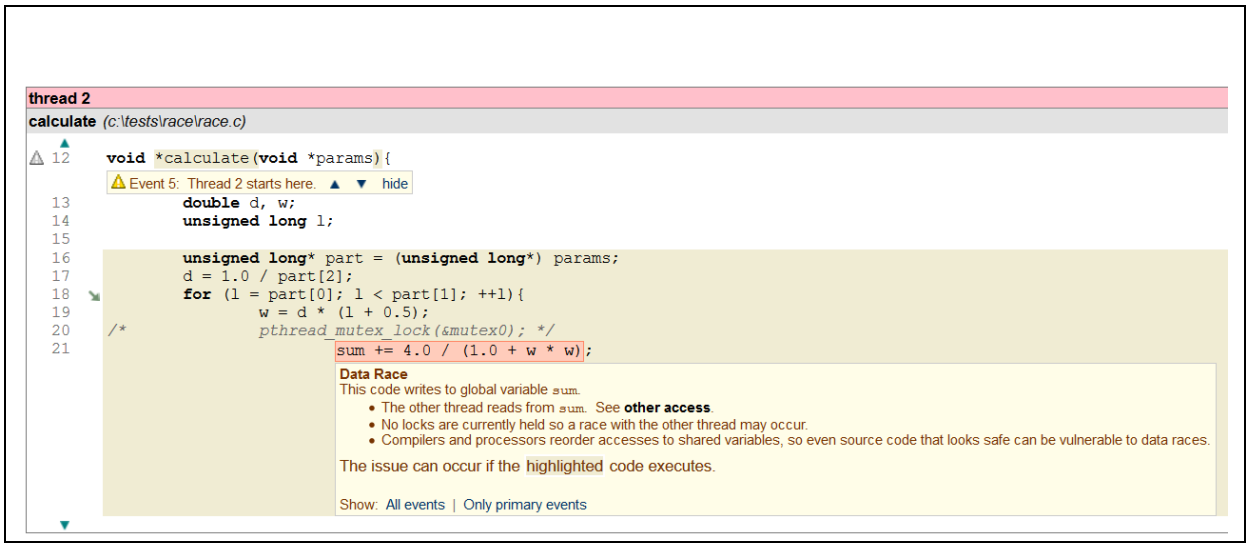

Abb. 5: Race Condition, aufgedeckt durch statische Codeanalyse

Auffinden eines Deadlocks

Zur Demonstration eines Deadlocks wird das kleine „Data Race - Beispiel“ etwas abgeändert.

```
void *calculate1(void *params){
        double d, w;
        unsigned long l;

        unsigned long* part = (unsigned long*)params;
        d = 1.0 / part[2];
        for (l = part[0]; l < part[1]; ++l){
                w = d * (l + 0.5);
                pthread_mutex_lock(&mutex0);
                pthread_mutex_lock(&mutex1);
                sum += 4.0 / (1.0 + w * w);
                pthread_mutex_unlock(&mutex1);
                pthread_mutex_unlock(&mutex0);
        }
}


void *calculate2(void *params){
        double d, w;
        unsigned long l;

        unsigned long* part = (unsigned long*)params;
        d = 1.0 / part[2];
        for (l = part[0]; l < part[1]; ++l){
                w = d * (l + 0.5);
                pthread_mutex_lock(&mutex1);
                pthread_mutex_lock(&mutex0);
                sum += 4.0 / (1.0 + w * w);
                pthread_mutex_unlock(&mutex0);
                pthread_mutex_unlock(&mutex1);
        }
}
```

Abb. 6: Klassischer Deadlock

Abb. 6 zeigt eine (zugegeben wenig sinnvolle) Modifikation des in Abb. 4 gezeigten Beispielprogrammes. Jeder Thread erhält seine eigene „calculate()“-Funktion. Der Zugriff auf die gemeinsame Variable „sum“ wird mittels zweier Mutex-Verriegelungen synchronisiert. Die Reihenfolge der Lock/Unlock-Verschachtelung ist klassisch für einen möglichen Deadlock.

Abb. 7: Warnung vor Deadlock als Ergebnis der statischen Codeanalyse

Abb. 7 zeigt, dass das Werkzeug durch statische Codeanalyse die Gefahr eines Deadlocks bei gefährlicher Mutex-Verschachtelung zu erkennen vermag.

Fazit

Der Einsatz eines leistungsfähigen Werkzeuges zur statischen Codeanalyse hilft Nebenläufigkeitsfehler schon früh in der Entwicklungsphase aufzufinden und trägt damit in erheblichem Maße dazu bei, Softwareprojekte risikominimiert und kosteneffizient abzuschließen.

Über den Autor:
Dipl.-Ing. (FH) Royd Lüdtke ist als Director Static Analysis Tools für die Verifysoft Technology GmbH tätig.

Quellenverzeichnis

[1] ISO/IEC 9899:2011 Information - Technology – Programming Languages – C
Lieferbar im Adobe PDF Format von der International Organization for Standardization, Genf, Schweiz
[2] ISO/IEC 9899:1999 Information - Technology – Programming Languages – C
Lieferbar im Adobe PDF Format von der International Organization for Standardization, Genf, Schweiz
[3] Boehm, B. W. (1981), Software Engineering Economics, Englewood Cliffs, N.J.: Prentice Hall
[4] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, T. Anderson (1997), Eraser, A Dynamic Data Race Detector for Multithreaded Programs: ACM Transactions on Computer Systems, Vol. 15, No. 4
[5] CodeSonar® User Guide and Technical Reference, GrammaTech, Inc., 531 Esty Street, Ithaca, NY 14850