Automatisches Erkennen von Data Races

Stephan Grünfelder
Friedersdorf 4
3643 Maria Laach
http://www.methver.webs.com

Race Conditions sind eine besonders unangenehme Art von Programmfehlern. Sie können überall dort auftreten, wo Software nicht nur rein sequenziell abgearbeitet wird, sondern zumindest zum Teil parallel. Das betrifft also fast alle Anwendungen mit Betriebssystemen und auch Software, die mit Interrupt Service Routinen arbeitet. Race Conditions treten nur sporadisch auf und sind besonders schwer zu erkennen. Dieser Artikel zeigt wie Race Conditions durch Software-Werkzeuge erkannt werden können und demonstriert den Einsatz eines solchen Werkzeugs beim Test eines Echtzeit-Betriebssystems.

Race Conditions, haben zurecht den Ruf sehr heimtückisch und unangenehm zu sein. Sie treten nur selten auf und sind mit traditionellen Testmethoden [1,2,3,4] gar nicht oder nur durch Zufall zu finden. Und selbst wenn so ein Test eine Race Condition erzeugt, kann es noch immer sehr schwer sein nachzuweisen, dass sie jemals auftrat. Ebenso schwer ist es dann die Ursache für eine Race Condition zu finden. Denn ein geänderter Compiler-Schalter, ein neues `printf` zu Debug-Zwecken oder ein geändertes Speichermodell verändern das Laufzeitverhalten des Systems und können den so schwer zu rekonstruierenden Fehler unsichtbar machen.

Wie Data Races aussehen, was sie machen

Eine *Race Condition* ist eine Art von Fehler in einem System oder Prozess, wo der Ausgang des Prozesses unerwartet von der Reihenfolge oder Dauer von anderen Ereignissen abhängt [7]. Der Begriff entstammt der Vorstellung, dass zwei Signale in einem Wettlauf versuchen die Systemantwort jeweils zuerst zu beeinflussen. Race Conditions können in schlecht entworfenen Schaltungen und schlecht entworfener Software auftreten. Bei Software speziell dann, wenn das Programm durch mehrere parallel laufende Prozesse ausgeführt wird (*multi threaded execution*).

Listing 1 zeigt den häufigsten Vertreter unter den Race Conditions, ein sogenanntes *Data Race*. So genannt, weil zwei parallel ausgeführte Programmpfade durch den Zugriff auf gemeinsame Daten (die gemeinsame Variable) in eine ungewollte zeitliche Abhängigkeit zueinander geraten. Sie „laufen“ beim Zugriff auf die Variable „um die Wette“ und dieser Wettlauf kann zu einem Fehler führen. Nehmen wir ein System mit einer CPU an. Thread 1 aus Listing 1 wird unterbrochen, als er gerade den Wert für `primzahlen` aus dem Hauptspeicher in ein Register lädt, kurz nachdem er eine neue Primzahl gefunden hat. Nun bekommt aber Thread 2 die CPU. Auch er findet eine neue Primzahl lädt den gleichen Wert für `primzahlen` aus dem Hauptspeicher in ein Register, inkrementiert ihn und schreibt ihn in den Speicher zurück. Nun bekommt Thread 1 wieder die CPU. Dabei wird zum veralteten Wert für `primzahlen` ebenfalls Eins dazuaddiert und dann in den Hauptspeicher zurückgeschrieben. Die Zählung der Primzahlen ist somit falsch, die zwei gefundenen Primzahlen wurden nur als eine Primzahl gezählt. Laufen die beiden Threads auf verschiedenen CPUs, so kann der Fehler ebenfalls auftreten, wenn eine der beiden CPUs die

Variable aus dem Speicher liest bevor die andere CPU den neuen Wert in den Speicher zurückgeschrieben hat.

```
#include <rtos.h>
#include <stdio.h>

extern int is_prime(int);

void zaehler1(void);  /* Thread 1 */
void zaehler2(void);  /* Thread 2 */

int primzahlen = 0;   /* so viele Primzahlen wurden gefunden */


int main(void)
{
   rtos_id id_thread1, id_thread2;

   rtos_thread_create(ROUND_ROBIN_SCHEDULING, &id_thread1);
   rtos_thread_create(ROUND_ROBIN_SCHEDULING, &id_thread2);

   rtos_thread_start(id_thread1, zaehler1);  /* starte Thread 1 */
   rtos_thread_start(id_thread2, zaehler2);  /* starte Thread 2 */

   printf("Es gibt %d Primzahlen zwischen 1 und 20000",
          primzahlen);

   return 0;
}


void zaehler1(void)
{
   int i;
   for (i = 1; i < 10000; i++)
   {
      if (is_prime(i)) primzahlen = primzahlen + 1;
   }
}


void zaehler2(void)
{
   int i;
   for (i = 10000; i < 20000; i++)
   {
      if (is_prime(i)) primzahlen++;
      /* auch der Operator ++ garantiert keine
       * Atomizität des Inkrements */
   }
}
```

Listing 1: Beispiel eines Data Races. Die Anzahl der Primzahlen zwischen 1 und 20000 wird vermutlich meist korrekt berechnet. Muss aber nicht unbedingt so sein.

Ob es tatsächlich zum beschriebenen Fehlerszenario auf einem Ein- oder Mehrprozessorsystem kommt, ist großteils Zufallssache. Es sind Fälle von auf Data Races zurückzuführenden Software-Fehlern bekannt, die jahrelang unsichtbar blieben, ehe sie zu einem plötzlichen folgenschweren Systemausfall führten. Denn nicht alle Data Races sind so

leicht zu sehen, wie es in Listing 1 der Fall ist. Listing 2 zeigt, wie sich Data Races auch besser verstecken können.

```
#include <string.h>

int a[100];

void thread1(void)
{
   int i, *q;
   char *s;
   ...
   a[i] += 1;     /* Data Race, wenn thread1.i == thread2.j */
   *q++;          /* Data Race, wenn thread1.q == thread2.p */
   strtok(s,';'); /* Oft ein Data Race, wenn die C-lib nicht
                   * "thread-safe" ist, weil dort eine
                   * Variable verwendet wird um einen Zustand
                   * zu speichern. */
}

void thread2(void)
{
   int j, *p;
   char *s;
   ...
   a[j] += 2;
   *p++;
   strtok(s,';');
}
```

Listing 2: Beispiele für schwerer erkennbare Data Races.

Dieser Artikel beschäftigt sich ausschließlich mit der Erkennung von Data Races und lässt andere Arten von Race Conditions außer acht. Zur Erkennung stehen eine Reihe von Werkzeugen, teils kommerziell, teils Freeware, zur Verfügung. Auf den folgenden Seiten wird deren Funktion vereinfacht dargestellt. Ziel des Artikels ist es, Sie mit dem nötigen Know-How zu versorgen, das für Ihre Anwendung richtige Werkzeug auszusuchen. Zunächst wird aber noch ein für das Verständnis der Funktion wichtiger Mechanismus vorgestellt: *Locking*.

Locking als Waffe gegen Data Races

Ein möglicher Fix für das Problem in Listing 1 ist den Zugriff auf die gemeinsame Variable `primzahlen` exklusiv zu machen. Das ist in Listing 3 geschehen. Eine Service des Betriebssystems garantiert, dass solange ein Thread auf die gemeinsame Variable zugreift, der andere Thread dies nicht kann. Mit den beiden Betriebssystem-Direktiven `rtos_lock()` und `rtos_unlock()` wird eine *Critical Section* definiert. Betritt ein Thread die Critical Section, so kann sie der andere erst betreten, wenn sie der erste Thread wieder verlassen hat. Anders formuliert: diese Direktiven definieren eine virtuelle Zutrittsberechtigung, ein *Lock*. Ein Lock ist ein Objekt zur Thread-Synchronisation, es ist entweder verfügbar oder im Besitz eines Threads.

Ein Data Race um eine Variable *x* zwischen zwei oder mehreren Threads kann nur dann vorkommen wenn

- zumindest ein Zugriff auf *x* schreibend ist,
- die Threads keinen Mechanismus verwenden, der den gleichzeitigen Zugriff auf *x* verhindert.

```
#include <rtos.h>
#include <stdio.h>

extern int is_prime(int);

void zaehler1(void);  /* Thread 1 */
void zaehler2(void);  /* Thread 2 */

int primzahlen = 0;   /* so viele Primzahlen wurden gefunden */
rtos_id id_lock;      /* Schutz f. d. Zugriff auf primzahlen *
                       * dies ist eines von potentiell vielen
                       * "Locks" */

int main(void)
{
   /* Start der Threads u.s.w. */
}


void zaehler1(void)
{
   int i;
   for (i = 1; i < 10000; i++)
   {
      if (is_prime(i))
      {
         rtos_lock(id_lock);
         primzahlen = primzahlen + 1;
         rtos_unlock(id_lock);
      }
   }
}


void zaehler2(void)
{
   int i;
   for (i = 10000; i < 20000; i++)
   {
      if (is_prime(i))
      {
         rtos_lock(id_lock);
         primzahlen++;
         rtos_unlock(id_lock);
      }
   }
}
```

Listing 3: Fix für das Data Race aus Listing 1. Nun wird der Zugriff auf die gemeinsame Variable geschützt.

Eraser

Nehmen wir nun an, dass uns Software vorliegt, die exklusiv Locking verwendet um den exklusiven Zugriff auf gemeinsam verwendete Variablen zu erwirken. Können wir da nicht automatisch feststellen, ob wir Data Races in der Software haben? Können wir. Der Algorithmus *Eraser*, oft auch *Lockset Algorithmus* genannt, kann das für uns tun [5]. Implementierungen von Eraser findet man in Open Source Tools und in kommerziellen Werkzeugen.

Eraser verifiziert, dass Zugriffe auf von verschiedenen Threads gemeinsam verwendete Variablen immer mit Locks geschützt werden. Die zugrundeliegende Idee ist ganz einfach:

(1) Für jede gemeinsam verwendete Variable *v* definiere eine Menge von potentiell darauf angewandten Locks *C(v)*.

(2) Starte die Software.

(3) Sei *locks_aktiv(t)* die Bezeichnung für die momentan von Thread *t* gehaltenen Locks. Für jeden Zugriff auf eine Varialbe *v* durch einen Thread *t* setze $C(v) := C(v) \cap locks\_aktiv(t)$.

(4) Wenn $C(v) = \{\}$, dann gib eine Warnung aus.

Bild 1 zeigt ein sehr einfaches Beispiel, wo Eraser ein Data Race erkennt, wenn das im Bild links gezeigte Programm auf einem Doppelprozessor-System läuft. Wir sehen, dass die Zugriffe auf die Variable *v* in dem dargestellten Zeitraster gar nicht zu einem Fehler führen und trotzdem ein Data Race erkannt wird.

Ohne Verfeinerung des Algorithmus wäre Eraser aber zu strikt und würde daher viele Fehler melden, die keine sind. Die Entwickler von Eraser haben daher eine Methode vorgesehen drei gängige Programmszenarios zu erkennen und auszunehmen:

(1) Die Initialisierung von gemeinsam verwendeten Variablen ohne Lock ist zulässig.

(2) Variablen, die nur einmal während der Initialisierung geschrieben werden und später von mehreren Threads gelesen werden, benötigen kein Locking.

(3) Die Verwendung von Read-Write Locks, die mehreren lesenden Threads gleichzeitig das Betreten der Critical Section erlauben, aber nur einem schreibenden Thread, wird erkannt.

Thread 1, CPU-1	**Thread 2, CPU-2**	***Locks_aktiv***	***C(v)***
		{}	{mtx1, mtx2}
lock(mtx1);	Irgendwas();	{mtx1}	{mtx1, mtx2}
v = v + 1;	Irgendwas();	{mtx1}	{mtx1}
unlock(mtx1);	Irgendwas();	{}	{mtx1}
Irgendwas();	lock(mtx2);	{mtx2}	{mtx1}
Irgendwas();	v = v + 1;	{mtx2}	**{}**
Irgendwas();	unlock(mtx2};	{}	{}

Bild 1: Die gemeinsame Variable v wird durch verschiedene Locks geschützt. Für den notwendigen exklusiven Zugriff auf die Variable wäre es aber notwendig bei beiden Zugriffen den gleichen Lock zu halten. Auch wenn der simultane Zugriff auf v in der dargestellten Sequenz nicht auftritt, wird das Data Race erkannt.

Erkennen von harmlosen Szenarien

In vielen Systemen ist es nicht unüblich, dass ein Init-Thread zunächst einmal einige Variablen oder Objekte initialisiert bevor andere Threads ihre Arbeit aufnehmen und dann ggf. auf diese Variablen zugreifen. Um falschen Alarm für diese Fälle zu vermeiden, darf Eraser erst mit der Reduzierung der Locksets *C(v)* beginnen, wenn die Initialisierung abgeschlossen ist. Nun „weiß“ der Algorithmus aber nicht, wann die Initialisierung zu Ende ist und trifft daher die Annahme, das dies mit dem ersten Zugriff eines zweiten Threads erledigt ist. Solange Zugriffe auf eine Variable *v* nur durch einen einzigen Thread erfolgen, bleibt *C(v)* unverändert.

Nachdem das simultane Lesen einer gemeinsam verwendeten Variable durch verschiedene Threads kein Data Race ist, besteht auch keine Notwendigkeit die Variable zu schützen, wenn sie nur „read only“ ist. Um diesem Umstand Rechnung zu tragen, gibt Eraser nur dann eine Warnung aus, wenn eine initialisierte Variable das erste Mal durch einen anderen Thread beschrieben wird.

Damit Eraser die beiden eben beschriebenen Szenarien berücksichtigen kann, wird für jede Variable ein in Bild 2 gezeigter Zustandsautomat definiert. Zunächst ist der Automat im Zustand *Virgin*. Nach einem ersten Zugriff auf die assoziierte Variable wechselt er in den Zustand *Exclusive*. Dieser Zustand bedeutet, dass bislang nur ein Thread auf die Variable zugreift, es ist anzunehmen, dass dieser Thread die Variable initialisiert. Zugriffe aus diesem Thread ändern nichts am Zustand des Automaten und ändern auch *C(v)* nicht, Szenario (1) führt somit nicht zum Falschalarm.

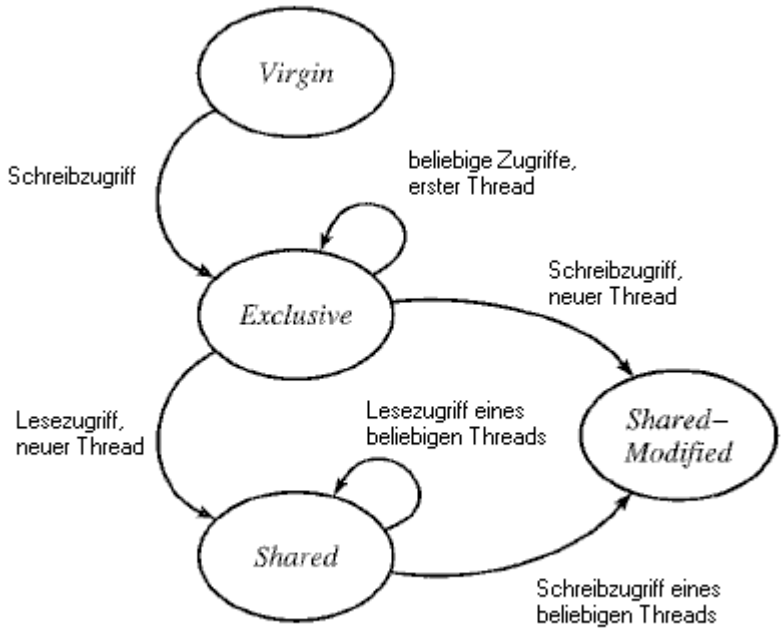

Bild 2: Eraser bedient für jede Variable einen hier gezeigten Zustandsautomaten. Data Races werden nur gemeldet, wenn sich der Automat im Zustand *Shared-Modified* befindet.

Ein Lesezugriff eines anderen Threads ändert den Zustand in *Shared*. In diesem Zustand wird *C(v)* aktualisiert, wie vorher beschrieben, aber um gegen Szenario (2) immun zu sein wird

keine Data-Race-Warnung ausgegeben, selbst wenn *C(v)* leer ist. Ein Schreibzugriff eines neuen Threads bringt einen Zustandswechsel zu *Shared-Modified*. In diesem Zustand wird *C(v)* immer aktualisiert und wenn *C(v)* leer ist wird ein Data Race gemeldet.

Die Beschreibung des Umgangs mit Szenario (3), das waren die Read-Write-Locks, ist in [5] nachzulesen. Auch das Erkennen dieser drei Szenarien kann nicht verhindern, dass Eraser manchmal vor einem Data-Race warnt, wenn es nicht notwendig wäre. Für diese Fälle kann der Benutzer mit Hilfe von speziellen Kommentaren im Quellcode (*source code annotations*) die überflüssige Warnung verhindern.

Eraser im Vergleich

Was sind nun die Stärken und Schwächen von Eraser? Zunächst muss Eraser die Software instrumentieren um jede Variable mit einem Zustandsautomaten assoziieren zu können. Instrumentieren heißt, dass der Quellcode oder der Binärcode automatisch verändert werden muss, damit Eraser arbeiten kann. Des weiteren muss Eraser das instrumentierte Programm ausführen können. Das könnte nicht immer so leicht sein. Denken Sie an Systeme mit knappem Speicher. Die Instrumentierung macht das Programm größer und langsamer.

Eraser findet nur Fehler in Programmteilen, die auch tatsächlich ausgeführt wurden. Es obliegt dem Benutzer sich darum zu kümmern, dass alle Programmteile, die zu Data Races führen könnten auch tatsächlich im Testlauf ausgeführt werden. Eraser kann nicht mit anderen Synchronisationsmechanismen außer Locks umgehen. Bedient sich die zu prüfende Software aber keiner anderen Mechanismen außer Locks, so findet Eraser Data Races nahezu garantiert, selbst dann, wenn diese im Testlauf gar nicht eingetreten sind und auch dann, wenn die betroffenen Programmteile nur ein einziges Mal durchlaufen wurden.

Bild 3 zeigt ein auf Eraser basierendes Tool in Aktion. Eine von mehreren Threads aufgerufene C-Funktion erhöht eine globale Variable. Der potentielle Fehler wird über die mehrfach angesprochene Adresse erkannt.

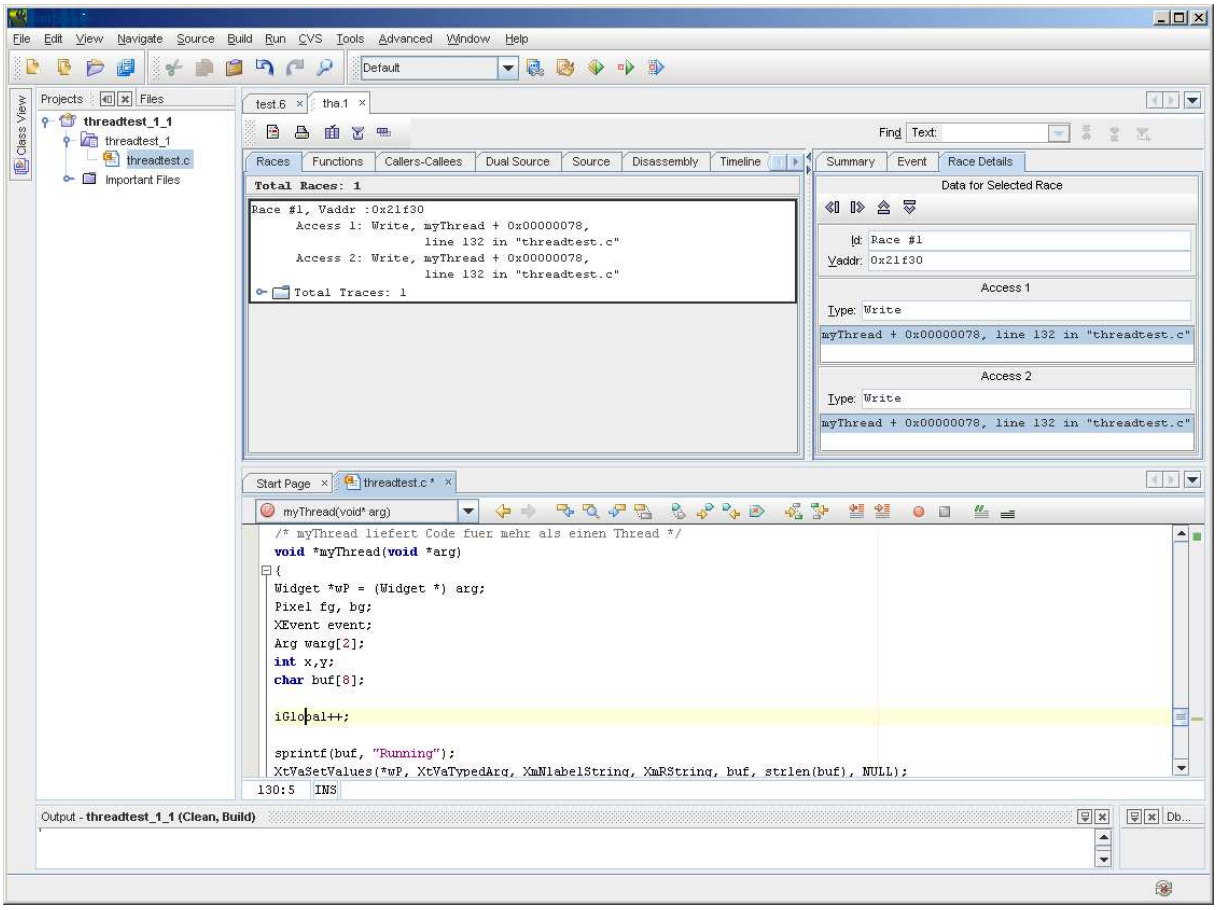

Bild 3: Das finden von Data Races ist ein leichtes Spiel für Eraser, wenn eine von mehreren Threads genutzte Variable ungeschützt beschrieben wird.

Happens Before

Eine andere Familie von Algorithmen zum Erkennen von Data Races basiert auf der sogenannten *Happens-Before-Relation*. Mit Hilfe dieser transitiven Halbordnung wird festgestellt ob die während eines Testlaufs protokollierten Zugriffe auf gemeinsame Variablen potentiell simultan erfolgen können. Das ist dann der Fall, wenn diese Zugriffe nicht gemäß der Halbordnung geordnet sind. Zwei Operationen sind gemäß Happens-Before geordnet,

- wenn sie in ein und demselben Thread ausgeführt werden so definitiv eine Operation vor einer anderen ausgeführt wird, oder
- wenn sie Synchronisationsoperationen sind und die Semantik dieser Operationen eine andere zeitliche Reihenfolge nicht gestattet.

Bild 4 illustriert diese Relation. Die Semantik der Synchronisationsoperationen `unlock()` und `lock()` erlaubt keine andere zeitliche Reihenfolge, wenn sie auf das gleiche Objekt `mutex` zugreifen, daher sind diese beiden Operationen bezüglich Happens-Before geordnet. Im Bild dargestellt durch einen Pfeil. Die anderen eingezeichneten Ordnungen ergeben sich durch Ausführung im jeweils selben Thread. Happens-Before is transitiv. Somit ist das Erhöhen von `x` in Thread 1 geordnet vor dem Erhöhen von `x` in Thread 2. Der Zugriff auf diese gemeinsam verwendete Variable wird für diesen Programmdurchlauf somit als unkritisch bewertet.

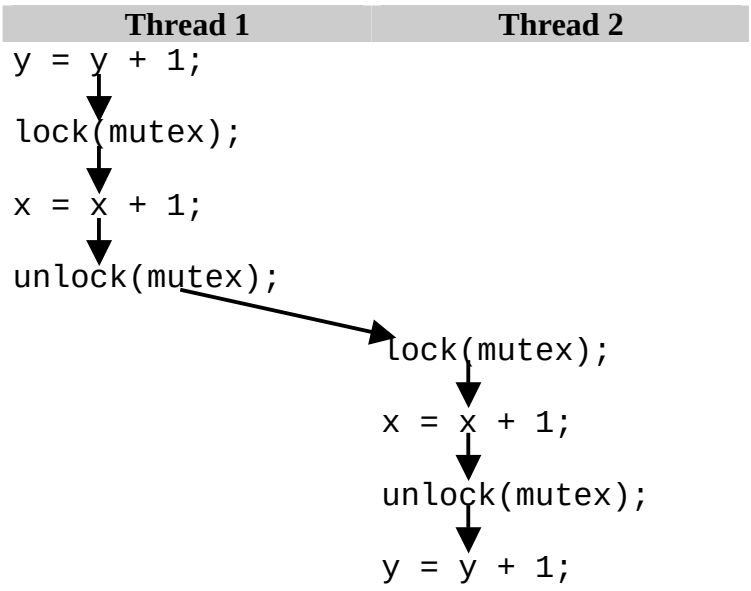

Bild 4: Der ungeschützte Zugriff auf die Variable y ist gemäß Happens-Before geordnet und kann daher bei dieser unglücklichen Reihung des Schedulers nicht über die fehlende Relation erkannt werden. Eraser hätte keine Probleme.

Wie man sieht ist auch der ungeschützte Zugriff auf `y` in diesem Programmdurchlauf durch Happens-Before geordnet. Um dieses Data Race zu erkennen, ist ein (anderer) Programmlauf nötig, wo der Kontext-Wechsel zwischen den beiden Threads so zustande kommt, dass die beiden Zugriffe auf `y` nicht mehr geordnet sind. Beispielsweise, wenn die im Bild 4 gezeigten Operationen von Thread 2 vor denen von Thread 1 ausgeführt werden.

Ob zwei Operationen bezüglich Happens-Before geordnet sind oder nicht, kann sehr effizient mit dem sogenannten *Vector Clock Algorithmus* errechnet werden. Die Beschreibung dieser Technik sprengt aber den Rahmen dieses Artikels und wird zum Beispiel in [6] sehr verständlich erklärt.

Auch Happens-Before-basierende Werkzeuge müssen Lesezugriffe von Schreibzugriffen unterscheiden um die Wahrscheinlichkeit von falschen Alarmen zu reduzieren. Es wird dann Alarm gegeben, wenn ein Abschnitt eines Threads eine Adresse liest und beschreibt, die ein nicht über die Relation geordneter Abschnitt eines anderen Threads beschreibt.

Happens Before im Vergleich

Auch unter Verwendung der Happens-Before-Technik muss der Code zuerst instrumentiert und dann ausgeführt werden um Race Conditions zu erkennen. Während Eraser auch in einem einzigen Programmdurchlauf alle ungeschützten Zugriffe auf gemeinsame Variablen erkennt, können Algorithmen, die mit Happens-Before arbeiten, nicht garantieren ein Data Race in jedem Fall zu erkennen. Sie sind darauf angewiesen, dass die zu untersuchenden Programmteile oftmals durchlaufen werden, sodass zumindest eine Scheduling-Sequenz den potentiell simultanen (Schreib-)Zugriff auf eine Variable aufdeckt. Das kann natürlich einiges

an Zeit kosten. Happens-Before-Methoden können im schlimmsten Fall Data Races verpassen, die Eraser leicht finden würde, siehe Zugriff auf Variable y in Bild 4.

Ist auf der anderen Seite so ein simultaner Zugriff aus bestimmten Gründen nicht möglich, etwa weil auch selbstgeschriebene Thread-Synchronisation eingesetzt wird oder weil externe Ereignisse dem Thread-Scheduling immer eine gewisse Reihenfolge aufzwingen, dann wird ein Happens-Before-Tool auch nicht Alarm schlagen, während Eraser das unnötigerweise tut.

Es werden nur Fehler in Programmteilen gefunden, die auch tatsächlich ausgeführt wurden. So wie bei Eraser muss der Fehler dazu aber nicht unbedingt auftreten um erkannt zu werden. Tools, die auf Happens-Before basieren, können typischerweise auch mit anderen Synchronisationsmethoden als nur mit Locks umgehen und sind daher universeller einsetzbar als auf Eraser basierende Tools.

Statische Analyse

Eine alternative zu den beiden vorgestellten dynamischen Methoden, bei denen die zu testenden Software instrumentiert und ausgeführt werden muss, ist die statische Analyse durch ein intelligentes Software-Tool. Hier muss die Software nicht ausgeführt werden. Das Tool analysiert den Quellcode der Software und gibt dann mehr oder weniger qualifizierte Warnungen aus. Publikationen über die Arbeitsweise dieser Tools erfordern meist vertiefende Kenntnis von formaler Logik und sind entsprechend schwierig zu lesen. Eine löbliche Ausnahme ist da die Arbeit von Dawson Engler and Ken Ashcraft [8], die Dank teilweisen Verzichts auf Kopierrechte am WWW zu finden ist.

Statische Tools ermitteln in einem ersten Schritt selbst die möglichen Programmpfade eines Threads aufgrund der Analyse des Quellcodes. Im zweiten Schritt wird dann verfahren wie in den dynamischen Verfahren.

Im Gegensatz zu den dynamischen Verfahren ist es daher nicht notwendig Testszenarios zu erzeugen. Beim Test eines großen Betriebssystems ist das ein großer Vorteil. Denken Sie an die unzähligen Hardware-Treiber, die etwa Linux beherbergt. Für dynamische Tests müsste immer die jeweilige Hardware parat sein!

Auf der anderen Seite müssen statische Tools öfters auf Heuristiken zurückgreifen, wenn es darum geht festzustellen, ob ein Zugriff auf eine Variable in einer Critical Section sein müsste oder nicht. Besonders oft kommen Heuristiken zum Einsatz, wenn Zeigervariablen verwendet werden. Funktions-Pointer erschweren festzustellen welche Teile des Quellcodes im Kontext eines Threads laufen. Zeiger auf Variablen erlauben der Quellcode-Analyse nicht festzustellen auf welche Adresse zur Laufzeit gezeigt wird. Alleine der Typ des Zeigers lässt in geringem Maße Rückschlüsse zu.

Die Folge dieser Unsicherheit ist eine relativ große Anzahl von Falschalarmen und damit entweder Bedarf an ebenfalls vergleichsweise vielen Spezial-Kommentaren im Code oder großer Aufwand bei der Parametrisierung der Heuristiken.

Statische Analyse im Vergleich

Nachdem der Code zur Analyse nicht instrumentiert werden muss, spielen Platz- oder Laufzeitprobleme keine Rolle. Auch wenn man Millionen von Programmzeilen auf Data Races überprüfen will, ist statische Analyse ein heißer Tipp: es müssen keine Testfälle erstellt werden. Die Ersparnis der Erstellung der Testdurchläufe wird aber teilweise wieder durch den

hohen Aufwand bei der Beurteilung von falschen Alarmen und beim Schreiben von Spezialkommentaren aufgefressen.

Die Verwendung von Zeigern im Testobjekt kann einige Probleme bereiten. Sie erfordern die Verwendung von Heuristiken bzw. intensive Interaktion mit dem Benutzer

In der akademischen Forschung wurde statische Analyse erfolgreich mit dynamischen Ansätzen kombiniert: Im ersten Schritt untersucht ein übervorsichtiger statischer Algorithmus in welchen Programmteilen es überhaupt zu Data Races kommen kann. Im zweiten Schritt werden dann nur für diese Programmteile dynamische Verfahren angewandt und damit Zeit für die Erstellung der dynamischen Tests gespart.

Anwendungsbeispiel – Deeply Embedded

Ein Anwendungsbeispiel für Data Race Tests ist der Test des Echtzeitbetriebssystemkerns ARTOS. ARTOS ist ein von RUAG Space Austria entwickelter Micro Kernel, der speziell für Systeme mit sicherheitsrelevanter Funktion geschrieben wurde und gratis unter den Lizenzbestimmungen der GPL beziehbar ist. ARTOS unterstützt Semaphore (Locks), Events, Message Queues und Memory Partitions.

Die Operationen des Betriebssystems können durch mehrere Threads parallel aufgerufen werden. Daher muss ARTOS sicherstellen, dass Daten, auf die in den Operationen geschrieben wird, entsprechend geschützt werden. Um den Control Block eines Semaphors zu schützen, kann der RTOS-Kern nicht selbst einen Semaphor verwenden. Daher werden im Kern Interrupts vor dem Zugriff auf eine zu schützende Variable über die Funktionen `INTERRUPT__Disable()` ausgeschalten und danach über `INTERRUPT__Enable()` wieder eingeschalten. Nur im Rahmen der Abarbeitung eines Interrupts könnte der Scheduler einem anderen Task die CPU zuweisen und somit potentiell einem zweiten Thread Zugriff auf die zu schützende Variable gewähren. Das kurzzeitige Abschalten der Interrupts verhindert das.

Soweit so gut. Wie können wir dieses System aber testen? Die am Markt erhältlichen Data Race Test-Tools erkennen standardisierte Schnittstellen, wie zum Beispiel die Microsoft Windows Thread API oder die pthread API, und können ohne Kunstgriff im gegenständlichen Fall nicht helfen. Der Kunstgriff ist in den folgenden zwei Absätzen beschrieben.

Tool-Entscheidung und -Einsatz

Als Betriebssystem für hohe Sicherheitsansprüche liegen für ARTOS bereits Modultests vor, die alle Statements und Verzweigungen des Kernels ausführen. Es ist also naheliegend diese Tests für die Zwecke des Data Race Testing wiederzuverwenden. Die Wahl fällt daher eindeutig auf ein dynamisches Testverfahren. Nachdem das Ein- und Ausschalten von Interrupts einem einzigen globalen Lock-Mechanismus gleicht, ist Eraser das best geeignetste Verfahren.

Nun muss man dem Tool aber noch beibringen die Funktionen `INTERRUPT__Disable()`und `INTERRUPT__Enable()` als Lock zu erkennen. Das macht man zum Beispiel, indem diese Funktionen zum Zweck des Tests umgeschrieben werden. Die Disable-Funktion ruft dabei eine dem Tool bekannte Lock-Funktion auf, die Enable-Funktion eine dem Tool bekannte Unlock-Funktion. Für den Test werden danach in der Thread-Umgebung des Tools mehrere Threads gestartet, die die zu testenden Operationen aufrufen.

Verwandte Anwendungsfälle

Ganz so wie Data Races lassen sich auch potentielle Deadlocks automatisch erkennen, ohne, dass sie jemals auftreten. Diese Techniken werden im Buch (Arbeitstitel) „Software-Test für Eingebettete Systeme“, beschrieben, das voraussichtlich Mitte 2013 im dPunkt-Verlag erscheint. Autor des Buchs ist Stephan Grünfelder.

Literatur

[1] S. Grünfelder: Den Fehlern auf der Spur, Teil 1: Das Handwerk des Testens will gelernt sein. *Elektronik*, Heft 22/2004, S. 60-72.

[2] S. Grünfelder, N. Langmead: Den Fehlern auf der Spur, Teil 2: Modultests; Isolationstests, Testdesign und die Frage der Testumgebung. *Elektronik*, Heft 23/2004, S. 66-75.

[3] S. Grünfelder: Den Fehlern auf der Spur, Teil 3: Der Integrationstest, das ungeliebte Stiefkind des Testprozesses. *Elektronik*, Heft 13/2005.

[4] S. Grünfelder: Den Fehlern auf der Spur, Teil 4: Systemtests – die letzte Teststufe ist alles andere als eine exakte Wissenschaft. *Elektronik*, Heft 14/2006. S. 45-51.

[5] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, T. Anderson: Eraser, A Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computer Systems*, Vol. 15, No. 4, November 1997, S. 391-411.

[6] Mathias Thore, *Automatic detection of race conditions in OSE using vector clocks*. Diplomarbeit an der Universität Uppsala, Juni 2005, erhältlich über www.virtutech.com.

[7] www.wikipedia.org/wiki/Race_Condition

[8] D. Engler and K. Ashcraft: RacerX: Effective, Static Detection of Race Conditions and Deadlocks. *ACM Symposium on Operating Systems and Principles 2003*, Bolton Landing, New York, USA.