Die approbierte Originalversion dieser Diplom-/Masterarbeit ist an der Hauptbibliothek der Technischen Universität Wien aufgestellt (http://www.ub.tuwien.ac.at).

The approved original version of this diploma or master thesis is available at the main library of the Vienna University of Technology (http://www.ub.tuwien.ac.at/englweb/).

DIPLOMARBEIT

Data Race Tests für Software eines weltraumtauglichen GPS-Empfängers

ausgeführt am Institut für Computertechnik
der Technischen Universität Wien
unter der Anleitung von

Univ. Prof. Dipl.-Ing. Dr. Hermann Kaindl

und Dipl.-Ing. Dr. Stephan Grünfelder

durch

Klaus Zandl

Matr. Nr. 9825926

Wiener Straße 173
2103 Langenzersdorf

Wien, 29.12.2008

_____

Kurzfassung

Data Races sind sporadisch auftretende Softwarefehler, die durch ungeordnete, das heißt „gleichzeitige“, Zugriffe auf gemeinsame Daten entstehen, wenn mindestens einer der Zugriffe die Daten verändert. Fehler aufgrund von Data Races sind schwer reproduzierbar, daher wurden und werden Verfahren und Werkzeuge entwickelt, die Data Races aufspüren.

Diese Diplomarbeit beschreibt die werkzeugunterstützte Suche nach Data Races in der Software eines weltraumtauglichen GPS-Empfängers. Die Software beinhaltet das Echtzeitbetriebssystem ARTOS und eine auf ARTOS aufbauende Applikationssoftware.

Zu Beginn wurde ARTOS unter Nutzung der Microsoft Windows-Thread-API im Intel Thread Checker untersucht und die Ergebnisse dieser Analysen bewertet. Ausgehend von den Untersuchungsergebnissen, erfolgte die Auswahl eines Testansatzes zur Untersuchung der Applikationssoftware mit dem Werkzeug Thread Analyzer der Firma Sun Microsystems. Dazu wurde der Testansatz für Solaris, ein UNIX-Derivat, portiert. Neben der Beschreibung der Testansätze selbst und deren Auswahl, zeigt die vorliegende Arbeit die notwendigen Schritte, um von der Windows-Thread-API auf die POSIX-Thread-API im Thread Analyzer umzustellen.

Die Diplomarbeit schließt mit einer Bewertung der 152 gefundenen potentiellen Data Races im Thread Analyzer.

Danksagung

Für ihre wertvolle Unterstützung schon während des Studiums und für ihre stets motivierenden und aufmunternden Worte und Taten, danke ich vor allem meiner Frau Verena. Dieser Dank gilt auch meinen Eltern, die mir die freie Wahl ließen, welchen Weg ich in meinem Leben einschlage und für ihren Beistand in jeglicher Art und Weise.

Außerdem möchte ich mich bei Dr. Stephan Grünfelder für die sehr gute Betreuung im Rahmen meiner Diplomarbeit bedanken und die dadurch entstandene Möglichkeit, den Großteil meiner Arbeit bei RUAG Aerospace Austria erledigen zu können.

1 Einleitung

1.1 Motivation

Als Anwender von Computerprogrammen und Betriebssystemen ist man sicher schon öfters auf den Begriff „Bug“, einen Programmfehler gestoßen. Bei diesen Fehlern kann es sich z. B. um einen logischen Fehler (falscher Algorithmus) handeln. Im Zuge der Entwicklung der letzten Jahrzehnte wird Software (Betriebssysteme, Programme, etc.) immer komplexer und die Anzahl der Codezeilen vergrößert sich von Version zu Version. Darum wird es auch immer herausfordernder entsprechende Tests zu kreieren, die möglichst viele – kein Test kann von sich behaupten alle Fehler zu finden – dieser Bugs finden, um diese anschließend beseitigen zu können. Eine besondere Herausforderung ist der Test von Software, die nicht bloß immer nur einen Thread abarbeitet, sondern wo viele dieser Tasks gleichzeitig ablaufen. Die parallele Ausführung mehrerer ausführbarer Programmeinheiten (Tasks, Threads) in einem linearen Adressraum birgt aber die Gefahr einer Inkonsistenz in der Datenstruktur, wenn mehr als ein Thread auf diese Daten zugreifen kann. Diese Inkonsistenz nennt man Data Race, den Zustand, in dem dieses Data Race auftritt, bezeichnet man als Race Condition.

Ein Beispiel für das Auftreten eines Data Race ist die Berechnung von Primzahlen entsprechend Listing 1. Dabei wird diese Berechnung von 2 Threads auf einer CPU durchgeführt, wobei jedes Mal, wenn eine Primzahl gefunden wird, der Wert der Variable `primes` inkrementiert wird. Das folgende Szenario könnte zu einem Data Race führen: Thread 1 aus Listing 1 der gerade eine Primzahl gefunden hat und gerade den Wert von `primes` aus dem Speicher gelesen hat, wird von Thread 2 unterbrochen und dieser bekommt die CPU. Thread 2 findet ebenfalls eine neue Primzahl, inkrementiert `primes` und schreibt den neuen Wert für `primes` in den Speicher zurück. Anschließend gibt Thread 2 die CPU ab und Thread 1 ist wieder an der Reihe. Allerdings addiert Thread 1 zum alten Wert von `primes` Eins dazu und die Zählung der Primzahlen ist somit falsch!

Zusammengefasst entsteht ein Data Race, wenn zwei unterschiedliche Threads „gleichzeitig“ auf eine Variable zugreifen, wobei zumindest einer davon den Wert der Variable verändert und es existiert kein Mechanismus, der diese Threads vom simultanen Zugriff abhält [4]. Im vorliegenden Single-Prozessor-System passiert der Zugriff natürlich nicht zum gleichen Zeitpunkt. Mit „gleichzeitig“ ist hier der zeitlich ungeordnete Zugriff von unterschiedlichen Threads auf gemeinsame Daten gemeint. Ein Data Race führt somit unbemerkt zu Störungen in der Datenstruktur, führt aber nicht gleich zu einem Absturz. Vielmehr ist es so, dass der Fehler oft erst später, nach vielen Arbeitsschritten, im Programm auftritt oder unerkannt bleibt [3].

Das Auffinden von Data Races (Race Detection) wird immer wichtiger, da der Trend hin zu Multiprocessing und Multithreading immer größer wird [10]. Dieser Trend zum Multiprocessing bedeutet verstärkten Einsatz von Multitasking-Betriebssystemen. Ein solches Betriebssystem ist ARTOS, entwickelt von RUAG Aerospace Austria GmbH (RAA). ARTOS wird insbesondere für Systeme mit hohen Ansprüchen an die Software-Integrität verwendet.

Nun gibt es schon Werkzeuge, die Data Races in Applikationssoftware finden. Diese Werkzeuge setzen auf APIs von weit verbreiteten Betriebssystemen auf. Eine besondere Herausforderung ist es aber jetzt, das Betriebssystem selbst auf Data Races zu testen.

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

extern int is_prime(int);

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

int primes = 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", primes);
   return 0;
}

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

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

Listing 1: Beispiel für ein Data Race. Die Anzahl der Primzahlen zwischen 1 und 20000 wird vermutlich meist korrekt berechnet. Das muss allerdings nicht immer der Fall sein.

1.2 Aufgabenstellung

Im Rahmen dieser Diplomarbeit war das ereignisgesteuerte Echtzeitbetriebssystem ARTOS und eine auf ARTOS aufbauende Applikationssoftware auf Data Races mit Toolunterstützung zu testen. ARTOS ist mit zwei Scheduler-Varianten ausgestattet; einem präemptiven Scheduler und einem Run-To-Completion (RTC) Scheduler. Beim präemptiven Ansatz kann der momentan laufende Thread bevorrechtigt, d. h. unterbrochen werden, wenn ein höherpriorer Thread lauffähig wird. Im Gegensatz dazu wird beim RTC Scheduler der gerade ausführende Thread nicht unterbrochen, sondern dieser läuft bis zum Ende und erst dann wird die CPU für andere Threads verfügbar [17]. Dabei waren die bei RAA entwickelten Testansätze – Thread Application Wrapper (TAW) und Duplikation der System-Calls (DuSCa) – zu prüfen und neue Tests mit neuen Werkzeugen durchzuführen.

Der Testansatz TAW benutzt Windows-Funktionen für die Erstellung eines Multithread-Environments. So wird jeder Aufruf an eine ARTOS-Funktion, auf einen Aufruf einer Operation der Windows-Thread-API (Application Programming Interface) abgebildet.

Beim Ansatz DuSCa exekutiert ein Windows-Thread den RTC Scheduler und alle seine administrierten ARTOS-Tasks. Zwei andere Windows-Threads exekutieren parallel dazu Aufrufe von ARTOS-Operationen. Wie auch bei TAW, erreicht man mit einer Testapplikation 100% Verzweigungsabdeckung (C1-Abdeckung) für alle ARTOS-Operationen.

Die beiden bestehenden Testansätze sollten mit dem Intel Thread Checker ausgeführt und alle Fehler, Warnungen, etc. erklärt werden. Es sollte überprüft werden, ob die vom Thread Checker angezeigten Diagnosen auch tatsächlich Fehler bzw. berechtigte Warnungen sind, oder ob es sich um ein False Positive, eine Falschmeldung durch das Tool handelt. Thread Checker ist ein Werkzeug zur Erkennung von Data Races, das die Windows-Thread-API unterstützt. Es läuft unter Microsoft Windows XP.

Nach dieser Analyse der Ergebnisse, die der Thread Checker lieferte, sollte entschieden werden, welcher der beiden Ansätze für das Werkzeug Thread Analyzer, ein Data Performance Tool von Sun um Deadlocks und Data Races zu finden, zu portieren ist, um in dieser Umgebung ARTOS erneut auf mögliche Data Races zu testen und schließlich eine auf ARTOS aufbauende GPS-Software zu testen. Thread Analyzer läuft unter Solaris. Unter dem gleichen Betriebssystem läuft eine simulierte Einsatzumgebung der GPS-Applikationssoftware, das sogenannte Referenzmodell. Thread Analyzer erlaubt somit einen Test der GPS-Software im angestammten Betriebssystem.

Das Referenzmodell ist eine mit MATLAB gekoppelte C-Bibliothek und läuft unter Solaris. Ein Testtreiber für konventionelle Tests der GPS-Software wurde von RAA zur Verfügung gestellt. Das Aufspüren von Data Races mit Hilfe dieses Testtreibers war Gegenstand der Diplomarbeit. Es sollte mit Tool-Unterstützung festgestellt werden, ob Data Races in der Software existieren, und allfällige False Positives des Tools sollten bewertet werden.

2 Verwandte Arbeiten – State of the Art

Data Races treten dann auf, wenn zwei gleichzeitig ablaufende Threads auf eine gemeinsame Variable zugreifen und zumindest einer davon einen Schreibzugriff darstellt. Außerdem benutzen die Threads keinen Mechanismus um den simultanen Zugriff zu verhindern [4]. Data Races sind schwer reproduzierbar, auch wenn der Programmablauf unverändert und wiederholt ausgeführt wird [11]. Das Auffinden bzw. Auftreten von Data Races kann außerdem durch das Einfügen von Debug-Informationen oder zusätzlichem Code verändert werden, da sich dadurch das Scheduling-Verhalten ändert [9]. Das Schwerwiegende an Data Races ist der Umstand, dass ein möglicher Fehler nicht sofort auffällt, sondern erst viel später, oder gar nicht, durch veränderte Daten zum Vorschein kommen kann [4]. Nicht jedes Data Race führt notwendigerweise zu einem Programmfehler oder zu einem gefährlichen Programmzustand.

In diesem Kapitel werden einige Ansätze genannt, die das automatisierte Auffinden von Data Races erlauben. Den Abschluss bilden ein Vergleich der beschriebenen Ansätze und ein kurzer Exkurs zum Thema Testen von Betriebssystemen.

2.1 Testansätze und Tools zum Auffinden von Data Races

Grundsätzlich kann man zwischen zwei Ansätzen (und den auf dem Markt befindlichen Tools, die diese verwenden) unterscheiden: statische Tools, die den Quellcode ohne tatsächlichen Programmdurchlauf untersuchen und dynamische Tools, die das Programm zur Laufzeit analysieren. Als mögliche dritte Variante können hier noch Post-Mortem-Techniken angegeben werden, die nach dem Absturz eines Programms aktiv werden und aufgrund von Auswertungen von Logfiles, Rückschlüsse auf die verursachende Stelle im Code machen können. Diese Technik wird im Rahmen dieser Arbeit aber nicht länger verfolgt.

2.2 Statische Tools

Statische Analyse-Tools führen eine semantische Überprüfung des Quellcodes eines Programmes auf Fehler durch. Das Programm wird dabei aber nicht ausgeführt.

Statische Analysen lassen sich in zwei Technologien einteilen: Abstract Interpretation und Theorem Proving [1].

(1) Eine Abstract Interpretation erstellt eine mathematische Darstellung der Programm-eigenschaften und damit kann sofort ein mögliches fehlerhaftes Verhalten bezüglich der vorgegebenen Eigenschaften anzeigt werden. Ein entscheidender Nachteil hier sind zahlreiche False Positives.

(2) Theorem Proving ist die Methode einer mathematischen Verifikation des Programms. Es muss der Beweis für gegebene Eigenschaften erbracht werden. Dieser Ansatz ist einwandfrei und vollständig. Das Problem ist die Komplexität des Modells bzw. des Tools und der menschliche Faktor bei der Bewältigung dieser Komplexität.

Als Beispiel einer Implementierung kann RacerX angeführt werden. RacerX [3] ist ein statisches Tool zum Aufspüren von Race Conditions und Deadlocks aus dem universitären Umfeld, das der ersten oben angeführten Diktion folgt. Es wird dem Programm entsprechend ein Kontrollflussgraph erstellt und durchlaufen. Mit Hilfe einer Tiefensuche (Depth First Search) werden Locks – je nach Bedarf –

entweder hinzugefügt oder entfernt und anschließend wird jedes Statement des Kontrollflussgraphen auf Data Races und Deadlocks überprüft [3].

Sun's LockLint ist ein frei verfügbares Werkzeug mit kommerziellem Hintergrund und verfolgt ebenfalls Diktion 1. Es wird zur Analyse von Mutual Exclusions und Locks verwendet, um Inkonsistenzen in den verwendeten Locking-Techniken zu entdecken.

Das Problem aller statischen Analysen sind die vielen zu treffenden Annahmen und Heuristiken, die zu verhältnismäßig vielen False Positives führen. In universitären Projekten werden statische Verfahren verwendet, um den Einsatz für dynamische Verfahren auf „sensible Programmregionen" zu beschränken und damit den Testvorgang zu beschleunigen [9]. Aufgrund der Tatsache, dass statische Analysen eine relativ große Anzahl von Falschalarmen liefern, entsteht so ein großer Aufwand, da zum Beispiel Critical Sections im Code extra kommentiert und die verwendeten Heuristiken parametrisiert werden müssen.

2.3 Dynamische Tools

2.3.1 Eraser

Eraser ermöglicht die dynamische Erkennung von Data Races und erfordert die Verwendung einer sogenannten Locking Discipline (eine Programmiervorschrift um Data Races zu verhindern), daher auch der Name Lockset Algorithmus: Eraser basiert auf der Annahme, dass jede potentiell gemeinsam genützte Variable mittels Locking[1] geschützt wird [5].

Die Untersuchung des Programms erfolgt „on-the-fly", also zur Laufzeit. Dabei wird der Code instrumentiert, d. h. jede Programmzeile wird mit Aufrufen von Eraser erweitert, um so den Lockset Algorithmus zu implementieren [4]. Abbildung 1 stellt den von Eraser verwendeten Algorithmus als Zustandsautomaten dar. Ausgehend vom Virgin State, also dem Zustand einer Variablen vor der Initialisierung, geht der Zustand über zu Exclusive. Was bedeutet, dass bis jetzt nur ein einziger Thread die Variable verändert hat. Solange der gleiche Thread den Wert der Variable ändert und alle anderen Threads diesen Wert lediglich auslesen, bleibt der Zustand entweder weiter in Exclusive oder geht über auf Shared. Beides sind Zustände, die ohne Data Races erreicht werden.

Sobald ein zusätzlicher Thread (zum Thread, der die Initialisierung durchgeführt hat) die Variable ändert, gibt es einen Übergang zu Shared-Modified. Hier gibt Eraser eine Warnung aus, dass eine mögliche Race Condition (Zustand, in dem möglicherweise ein Data Race auftritt) vorliegt, wenn kein einziges Lock, in allen auf die Variable zugreifenden Threads, aktiv ist.

[1] Ein Lock ist ein einfaches Synchronisationsobjekt, das für Mutual Exclusion verwendet wird. Es ist entweder für einen Thread *verfügbar* oder es wird von einem Thread *gehalten.* Die Funktion entspricht dabei der eines Semaphor, mit dem Unterschied, dass nur dem Lock Owner erlaubt ist, das Objekt wieder verfügbar zu machen [4].

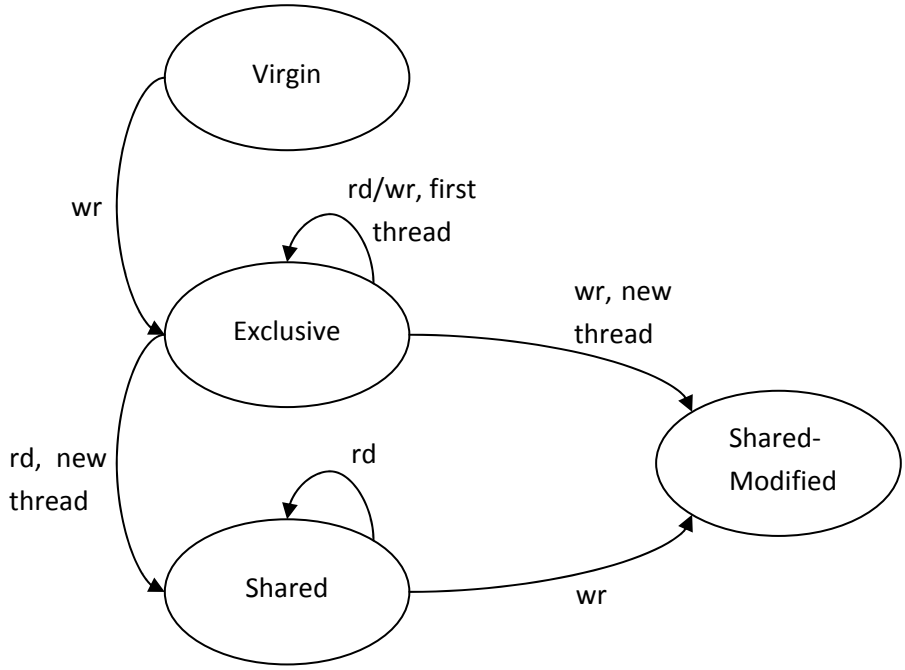

Abbildung 1: Der Eraser Lockset Algorithmus als Zustandsautomat. rd stellt einen Lesezugriff dar, wr steht für einen Schreibzugriff.

Würde Eraser sich nicht dieses Zustandsautomaten bedienen, sondern Warnungen nur dann ausgeben, wenn auf eine gemeinsame Variable ohne aktive Locks zugegriffen wird, würden auch für Initialisierungen von Variablen ohne Locking, Warnungen angegeben werden.

2.3.2 Implementierungen von Eraser

In [4] wird Eraser in optimierter Form dazu verwendet, den Webserver und Indexing Engine von AltaVista, den Vesta Cache Server und das Petal Distributed Disk System auf Race Conditions zu überprüfen. Dabei war eine gute Performance nicht das erklärte Ziel, wodurch eine Verringerung der Geschwindigkeit um einen Faktor 10 bis 30 bewusst in Kauf genommen wurde. Dass Eraser die Locking Discipline erfordert, kann diesem Algorithmus als Nachteil angerechnet werden. Die Autoren argumentieren, dass der damit verbundene Aufwand leicht zu rechtfertigen ist, wenn man bedenkt, dass Eraser einen gründlicheren Test nach Data Races ermöglicht, als andere dynamische Verfahren.

RaceTrack [11] wurde für den Test von multithreaded, objektorientierten Microsoft .NET-Programmen verwendet. Dabei kommt ein erweiterter Eraser-Algorithmus zum Einsatz. Alle Locks einer Variable x werden zu einem sogenannten Lockset (C_x) zusammengefasst. Die Menge von Threads, die auf diese Variable zugreifen, bilden ein sogenanntes Threadset (S_x). Sobald die Variable allokiert wird, wird C_x zu einem Set aller möglichen Locks initialisiert und S_x wird zu diesem Zeitpunkt als leere Menge ({}) definiert. In Abbildung 2 ist die Adaptive Threadset-Technik, die RaceTrack verwendet, dargestellt. Bei dieser Technik wird die Überprüfung des Threadsets S_x solange nicht in die Überprüfung auf mögliche Data Races mit einbezogen, bis das Lockset C_x ein Data Race meldet.

Die Zustände Virgin und Exclusive0 entsprechen Virgin und Exclusive des in Abbildung 1 dargestellten Automaten. Exclusive1 – in Abbildung 2 – wird erreicht, wenn ein weiterer Thread auf

die Variable x zugreift, aber diese Zugriffe finden nicht gleichzeitig statt ($|S_x| = 1$). Findet ein gleichzeitiger Zugriff statt ($|S_x| > 1$), geht der Zustand über auf Shared-Read oder Shared-Modify1. Auf Shared-Read, falls alle Zugriffe Lesezugriffe sind. Es wird daher nur das Lockset überprüft. In Shared-Modify1 finden auch gleichzeitige Schreibzugriffe statt. Wieder wird nur das Lockset analysiert, d. h. es wird überprüft, ob genügend Locks zum Schutz der Variable zur Verfügung stehen. Zu Exclusive2 wird übergegangen, sobald das Lockset nicht genügend Locks enthält. Hier wird nun das Threadset kontrolliert und man bleibt in diesem Zustand, solange $|S_x| = 1$ ist, d. h. Zugriffe durch Threads nicht gleichzeitig passieren. Shared-Modify2 wird erreicht, wenn gleichzeitige Lese- und Schreibzugriffe stattfinden. Es werden Threadset und Lockset analysiert.

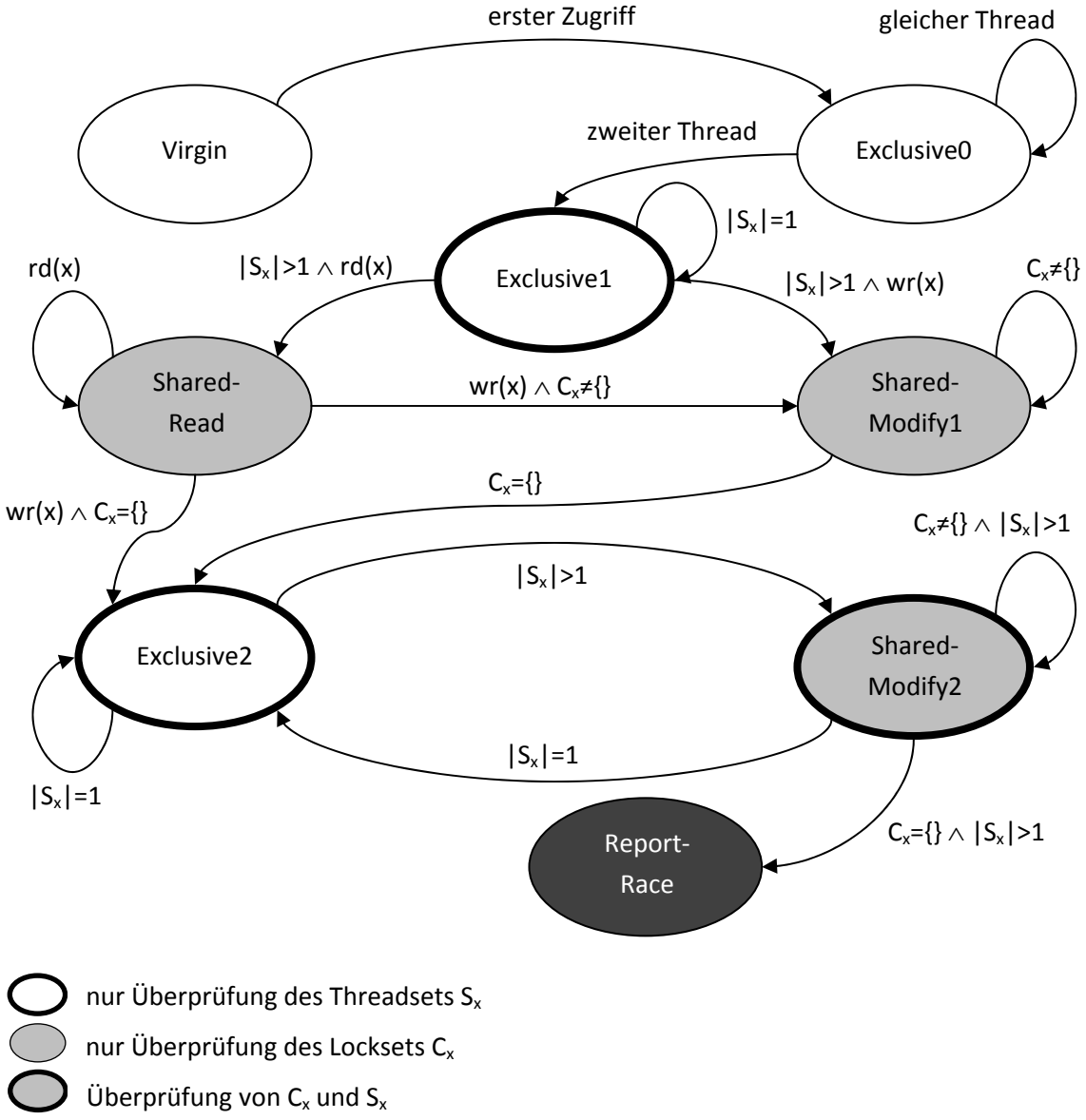

Abbildung 2: RaceTrack's Zustandsautomat. rd stellt einen Lesezugriff dar, wr steht für einen Schreibzugriff.

Falls die Zugriffe nicht mehr gleichzeitig erfolgen und genug Locks die Variable schützen, kommt es zum Übergang, zurück zu Exclusive2. Andernfalls, d. h. es stehen nicht ausreichend viele Locks zur

Verfügung (C_x = {})und es greifen mehrere Threads gleichzeitig zu ($|S_x| > 1$), wird ein potentielles Data Race gemeldet und es kommt zum Wechsel in den Zustand Report-Race.

Dank dieser Zusammenfassung von Threads zu einem Threadset und der getrennten Überprüfung von Lockset und Threadset verlangsamt RaceTrack die Programmausführung im Schnitt nur um einen Faktor 2.

Thread Analyzer [18] ist ein in Sun Studio integriertes Tool für die Erkennung und die Analyse von Data Races in einem Multithreading-Programm. Der Source-Code des Testobjekts wird während des Kompilierens instrumentiert, d. h. nur in diesen Code-Teilen können mögliche Data Races erkannt werden. Wie alle auf Eraser basierenden Werkzeuge, kann Thread Analyzer keine vom User implementierten Synchronisationsmechanismen erkennen, z. B. Spin Locks, die zur Synchronisation verwendet werden, und daher kann es zu False Positives kommen. Diese können aber durch geringfügige Veränderungen im Source-Code, durch Aufrufe von zur Verfügung gestellten Bibliotheksfunktionen, verhindert werden. Außerdem kann es durch die Ausführung des Thread Analyzers zu einem deutlichen Laufzeit-Overhead, abhängig von der Art der Applikation, kommen.

2.3.3 Die Happens-Before-Methode

Die Happens-Before-Relation wurde von Leslie Lamport formuliert und ist die Möglichkeit, die Kausalordnung von Ereignissen in asynchronen verteilten Systemen zu bestimmen [13]. Zwei Operationen sind gemäß Happens-Before geordnet,

- wenn sie in ein und demselben Thread ausgeführt werden und 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.

Abbildung 3 zeigt die mögliche Ordnung zweier Threads, die den gleichen Programmabschnitt ausführen. Die beiden Synchronisationsoperationen `lock` und `unlock` erlauben nur die dargestellte Reihenfolge, wenn sie auf das gleiche Synchronisationsobjekt mutex zugreifen. Diese Ordnung bezüglich Happens-Before ist durch die Pfeile in Abbildung 3 verdeutlicht. Happens-Before ist transitiv, d. h. aus der Tatsache, dass `unlock(mutex)` in Thread 1 vor `lock(mutex)` in Thread 2 ausgeführt wird, folgt, dass das Ändern von `x` in Thread 1 vor dem Erhöhen in Thread 2 geordnet ist. Daher ist dieser Programmablauf, und der damit verbundene Zugriff auf die gemeinsame Variable `x`, als unkritisch einzustufen.

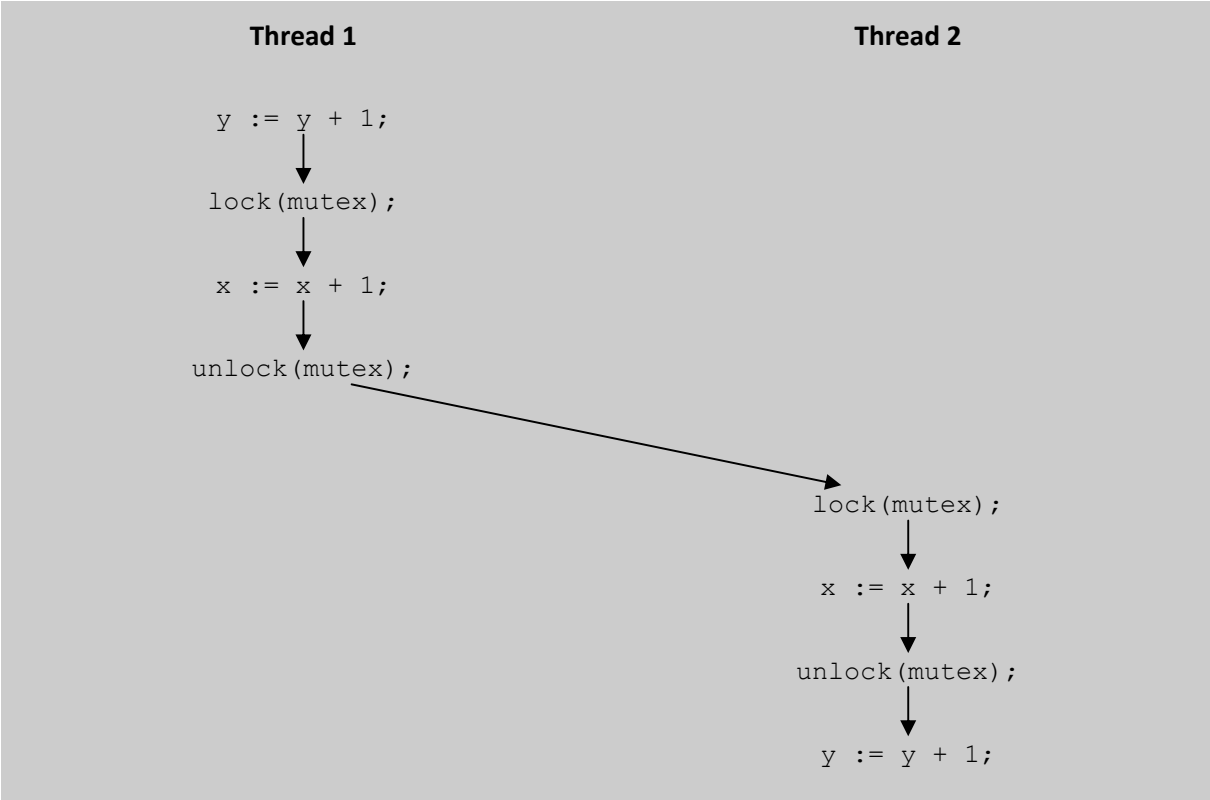

Abbildung 3: Lamport's Happens-Before-Relation ordnet die Events im gleichen Thread und in unterschiedlichen Threads in zeitlicher Reihenfolge. Bei ungünstigem Scheduling kann im gezeigten Code-Abschnitt jedoch ein mögliches Data Race auftreten.

Die für das Reporting von Data Races zugrundeliegende Annahme ist folgende: Greifen bei einem tatsächlich exekutierten Scheduling zwei Threads auf eine gemeinsame Variable zu, und diese Zugriffe sind nicht nach Happens-Before geordnet, kann es zu einem Data Race kommen und es wird eine Warnung ausgegeben.

Die Effektivität des Ansatzes hängt nämlich sehr stark vom Scheduling ab, denn dieses bestimmt, in welcher Reihenfolge die Code-Fragmente abgearbeitet werden. Es kann vorkommen, dass in einer analysierten Scheduling-Reihenfolge ein mögliches Data Race unentdeckt bleibt [4].

Um im ungeschützten Zugriff auf die Variable `y` in Abbildung 3 ein mögliches Data Race zu erkennen, ist ein unterschiedlicher Ablauf des Programms notwendig, wo der Kontext-Wechsel zwischen den beiden Threads so erfolgt, dass die beiden Zugriffe auf `y` nicht mehr gemäß Happens-Before geordnet sind. Das ist z. B. dann der Fall, wenn in Abbildung 3 Thread 2 vor Thread 1 ausgeführt werden würde. Diese Schwäche lässt sich durch eine große Anzahl von Testfällen (und damit Scheduling-Reihenfolgen) weitgehend vermeiden.

2.3.4 Implementierungen von Happens-Before

In der in [10] vorgestellten Variante wurde der Linux-Kernel auf Data Races, unter Verwendung von Happens-Before, getestet. Da Linux open source ist, haben sehr viele unterschiedliche Programmierer ebenso unterschiedliche Ansätze gewählt, um kritische Abschnitte im Code zu schützen. Daher ist es umso aufwändiger bzw. schwieriger die Tests zu gestalten, um nicht zu viele False Positives zu erhalten.

Goldilocks, beschrieben in [5], ist ein dynamischer Ansatz, der ausschließlich Happens-Before verwendet. Die Autoren geben an, dass Goldilocks genauso präzise wie Eraser arbeitet, aber eine bessere Performance bietet.

Der Intel Thread Checker [19] ist ein dynamisches Analyse-Tool, das Happens-Before verwendet, um Data Races bei der Verwendung von Threading- und Synchronisations-APIs zu erkennen. Der verwendete Local Conflict Detection Algorithm segmentiert den von den einzelnen Threads ausgeführten Code und erkennt garantiert alle lokalen Data Races innerhalb eines Segments. Die Praxistauglichkeit des Werkzeugs Thread Checker begründen die Autoren mit dem Kompromiss zwischen der Fähigkeit, in der überwiegenden Mehrheit von Situationen Data Races zu erkennen und der Notwendigkeit Speicherplatz zu sparen.

2.4 Vergleich von Race Detection Verfahren

Die folgende Auflistung soll einen kurzen Überblick über die Vor- und Nachteile der betrachteten Ansätze liefern:

- Dynamische Tools benötigen mehr Rechenleistung, verbrauchen mehr Testzeit und sind nicht auf Systemen mit strengen Zeitvorgaben einsetzbar [3].
- Außerdem können nur Fehler in tatsächlich ausgeführten Programmpfaden gefunden werden. Was bei Betriebssystemen mit zahlreichen Gerätetreibern zu einem größeren Problem führen kann, da es hier Pfade gibt, die nicht durchlaufen werden können [3, 11], wenn die Hardware nicht verfügbar ist, d. h. nicht angeschlossen ist.
- Dynamische Systeme, wie z. B. Eraser, sind schnell und leicht zu bedienen, können allerdings die Abwesenheit von Data Races mathematisch nicht beweisen und benötigen umfangreiche Testumgebungen [7].

Im Gegensatz dazu

- haben statische Analysen mit der Größe eines bestimmten Codes keinerlei Probleme, da der Code nicht instrumentiert werden muss. Sie können allerdings nicht so präzise Informationen zu Fehlern liefern, z. B. die genaue Angabe der Code-Zeile und der betroffenen Variable bzw. Funktion, wie ein Tool, das den Code zur Laufzeit überprüft [3].
- Statische Tools sind gründlicher, d. h. es werden mehr mögliche Race Conditions anzeigt, als ihre dynamischen Pendants, allerdings leiden sie mehr unter False Positives als dynamische Methoden [5].
- Einfache statische Tools, basierend auf Abstract Interpretation, können ein sehr nützlicher, erster Schritt für eine formale Verifikation sein [1].

Für alle Ansätze (egal ob dynamisch oder statisch) gilt, dass False Positives unvermeidbar sind. In vielen Fällen ist eine Kombination der verschiedenen Analyse-Tools und Technologien erfolgversprechender, als eine Technologie bis zu ihren Grenzen auszureizen [1].

Für die Wahl des Analyse-Tools im Rahmen dieser Diplomarbeit war vor allem die Größe von ARTOS ausschlaggebend. 3.187 Source Lines (davon 1.213 Zeilen Code und 1.639 Zeilen Kommentare) machen es absolut unnötig, sich wegen der Performance-Einbußen von Eraser, wie in [4] beschrieben, Gedanken zu machen. Außerdem rückte der sicherheitskritische Aspekt von ARTOS

die Entscheidung weiter in Richtung Eraser, da die Happens-Before-Relation, bei unpassendem Scheduling, Data Races übersehen kann.

2.5 Testen von Betriebssystemen

Beim Testen eines Betriebssystems, speziell eines Echtzeitbetriebssystems (RTOS), geht man analog vor wie bei jeder anderen Softwareentwicklung. Man führt Modultests (Unit Tests), Integrationstests und Systemtests durch. In [14] wird dazu der Design Approach for Real-Time Systems (DARTS) beschrieben. Hier wird das System zu Beginn in Tasks (gleichzeitig ablaufende Prozesse) und den dazugehörigen Interfaces zerlegt. Anschließend wird jeder dieser Tasks in Module inklusive deren Interfaces aufgeteilt. So kann für jedes Modul ein Unit Test und für jeden Task ein Integrationstest durchgeführt werden. Den Abschluss bildet ein Systemtest des gesamten RTOS.

Der gesamte Testvorgang wird aufgeteilt, d. h. das Unit Testing und Integration Testing wird auf dem Host System (Computer-System, das für die Entwicklung der Software verwendet wird) durchgeführt. Auf dem Target System (Computer-System, auf dem die entwickelte Software laufen soll) werden die Integrationstests abgeschlossen und der Systemtest ausgeführt.

Häufig ist ein Environment Simulator der einzige Weg, Tests auf dem Target System durchzuführen. Ein Environment Simulator simuliert das Verhalten einer RTOS-Umgebung, d. h. es verarbeitet alle Ein- und Ausgaben während des Tests. Folgende Vorteile ergeben sich dadurch [14]:

- Oft ist das reale Target System noch gar nicht verfügbar, um darauf testen zu können.
- Simulation ist die einzige Möglichkeit, um Rare Events, d. h. Ereignisse, die selten bis gar nicht auftreten, zu testen.
- Man hat mehr Kontrolle über die Testumgebung.
- Es ist einfacher Systemantworten zu erhalten, da die Simulationsumgebung einfacher instrumentiert werden kann.

Der Test eines Betriebssystems entspricht also, wie eingangs erwähnt, dem eines komplexen Softwareprojekts, mit Tests zu den jeweiligen Designphasen, analog zum klassischen Wasserfallmodell [15, 16]. Im Fall von ARTOS, führt der in Abschnitt 1.2 erwähnte Testtreiber, alle Unit Tests und Integrationstests durch.

Zum Thema Auffinden von Data Races in Betriebssystemen gibt es nur sehr wenig Literatur, diese wird im Folgenden besprochen.

2.5.1 Race Conditions im RTOS OSE

Als Anwendungsbeispiel für das Auffinden von Data Races wird in [8] ein Echtzeitbetriebssystem auf Race Conditions untersucht. Das verwendete Betriebssystem OSE, ein Echtzeitbetriebssystem von ENEA Embedded Technology AB, kommt vor allem auf Plattformen, wie z. B. PowerPC, ARM oder MIPS, zum Einsatz. Es wird häufig in Mobiltelefonen und – wie ARTOS – für sicherheitskritische Anwendungen verwendet.

Es wird eine adaptierte Form von Happens-Before verwendet. Der Test des RTOS läuft nicht auf der echten Hardware, sondern es kommt ein Simulator – Simics – zum Einsatz, der einen Standard-Computer simuliert. Der gesamte Race Detection Algorithmus ist separat geschrieben und in den Simulator integriert worden. Im Gegensatz zu diesem aufwändigen Ansatz, war es Ziel der

vorliegenden Diplomarbeit, ein fertiges Tool zu verwenden und für die Zwecke des Tests eines Echtzeitbetriebssystems, einen Adapter zu programmieren.

2.5.2 Dynamic Race Detection in Linux

In [10] wurde – wie schon in 2.3.4 erwähnt – der Linux-Kernel auf Data Races unter Verwendung von Happens-Before getestet. Die Herausforderung war der Test von mehr als einer Million Code-Zeilen, inklusive vieler userspezifischer Spin Locks. Die Autoren erklären, dass Interrupt Disabling (anstatt Locks zu verwenden) in Multiprozessor-Systemen keinen ausreichenden Schutz für Datenstrukturen bietet. Abhilfe würde die Erstellung von Kopien der einzelnen Datenstrukturen schaffen, da jedem Prozessor dann seine eigene Kopie zur Verfügung steht und eine Routine dann sicher Daten dieser Struktur verändern kann, wenn die Interrupts auf dem lokalen Prozessor deaktiviert sind.

Im Gegensatz dazu liegen bei ARTOS schon Systemtests mit 100%iger Verzweigungsabdeckung vor. Zusätzlich handelt es sich bei ARTOS um ein reines Single-Prozessor-System und daher ist auch die Methode, Interrupt Disable/Enable als Schutz von Critical Sections zu verwenden, zulässig.

3 Testobjekte und Testumgebung

Die im Rahmen dieser Arbeit untersuchten Testobjekte waren das Echtzeitbetriebssystem ARTOS und eine auf ARTOS aufbauende Applikationssoftware für einen GPS-Empfänger eines Forschungssatelliten. Das sogenannte Referenzmodell bildete die Simulationsumgebung für den Test der Applikationssoftware in Solaris. Dabei simuliert das Referenzmodell die GPS-Hardware, d. h. es werden der die Antenne bedienende ASIC (Application Specific Integrated Circuit) und die empfangenen GPS-Signale in Software modelliert. Sowohl ARTOS als auch das Referenzmodell sind in der Programmiersprache C codiert. Die Testdaten für die Stimulation der GPS-Applikation werden großteils offline, mit Hilfe von MATLAB, erzeugt.

3.1 Das Echtzeitbetriebssystem ARTOS

Das Software-Interface von ARTOS ist vollständig kompatibel zum open source Echtzeitbetriebssystem RTEMS (www.rtems.org). Dieses Interface führt alle Aufrufe von RTEMS-spezifischen Funktionen mit der Hilfe von Makros in die jeweiligen ARTOS-Funktionen über. Dabei handelt es sich um die Funktionen zur Behandlung von Message Queues, Semaphoren, Events, Interrupts, Partitions und Tasks.

ARTOS gibt es in zwei Varianten:

- ARTOS/n, mit Run-To-Completion (RTC) Scheduler, für einfache, missionskritische Anwendungen und
- ARTOS/p, mit präemptivem Scheduler.

Wie in der Aufgabenstellung in Abschnitt 1.2 schon kurz beschrieben, wird beim RTC Scheduler der gerade ausführende Thread nicht unterbrochen, sondern dieser läuft bis zum Ende und erst dann wird die CPU für andere Threads verfügbar. Im Gegensatz dazu kann beim präemptiven Scheduling der momentan laufende Thread bevorrechtigt, d. h. unterbrochen werden, wenn ein höherpriorer Thread lauffähig wird [17]. Threads sind beim präemptiven Scheduling daher typischerweise als Endlosschleifen implementiert. Beim RTC Scheduler wird der Context Switch, d. h. der Wechsel zwischen den einzelnen Threads, nur dann ausgeführt, wenn der laufende Thread terminiert oder blockiert. Die Auswahl der Scheduling-Strategie hat vor dem Kompilieren zu erfolgen, da sich die ARTOS-Module „Task“ und „Interrupt“ in den beiden Ausführungen voneinander unterscheiden. Das Interface beider Module ist allerdings bei ARTOS/n und ARTOS/p identisch.

Abbildung 4 zeigt die in ARTOS vorhandenen Module und deren Beziehungen untereinander. Im Vergleich der beiden ARTOS-Varianten, fällt bei ARTOS/n das Modul Context weg, da durch den RTC Scheduler in ARTOS/n kein Context Switch während der Laufzeit eines Threads möglich ist. In ARTOS/p hingegen, ist das Modul Context für Context Switching, also das Retten und Wiederherstellen von Registerzuständen, vor und nach dem Wechsel zwischen zwei Threads, zuständig.

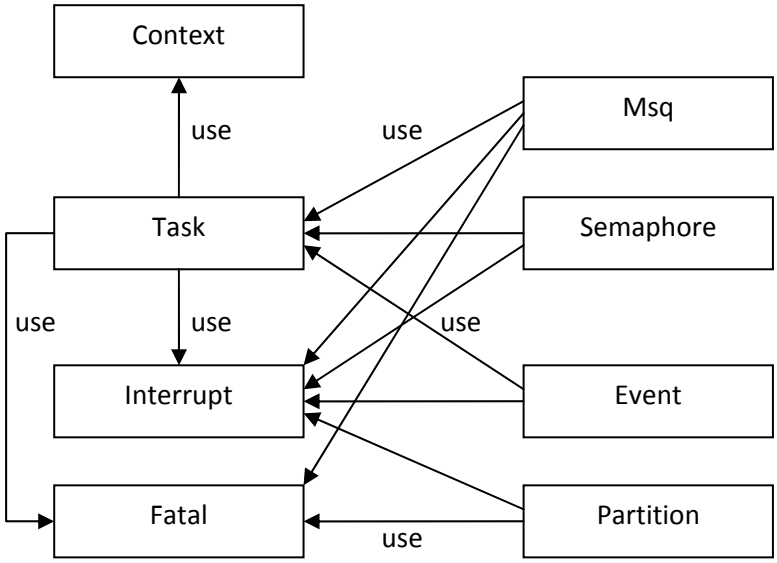

Abbildung 4: Module in ARTOS.

Kritische Abschnitte im Code (Critical Sections) werden durch das Deaktivieren und anschließende Reaktivieren von Interrupts geschützt. Da es sich beim beschriebenen System um ein Single-Prozessor-System handelt, ist diese Methode ausreichend.

3.2 Das Referenzmodell

Das Referenzmodell bildet eine Simulationsumgebung für den Test der Applikationssoftware. Abbildung 5 zeigt den Ablauf des Referenzmodells als Flussdiagramm. Die Applikationssoftware steuert den ASIC, mit der Bezeichnung AGGA (Advanced GPS and GLONASS ASIC). Dieser ASIC wird durch das Referenzmodell nachgebildet. Die GPS-Empfangsdaten werden im Vorhinein, d. h. offline, generiert. Die Adressen des verwendeten ASIC werden emuliert und die Testdaten werden an diese, durch das Modell simulierten Adressen, weitergeleitet. Außerdem erfolgt die Initialisierung aller Module des Referenzmodells. Nach der erfolgreichen Initialisierung werden die ARTOS-Tasks (es kommt ARTOS/n zur Anwendung) durch einen System-Call in der Applikationssoftware gestartet. Das bedeutet, dass die entsprechenden Ressourcen, z. B. Memory Partitions, Message Queues, Semaphore, für die zu startenden Tasks bereitgestellt werden. Außerdem werden sämtliche Interrupt Service Routinen (ISR) für alle definierten Interrupt Requests (IRQ) freigegeben. Jeder IRQ ist einer entsprechenden ISR zugeordnet. Im Normalbetrieb, ohne Referenzmodell, wird, sobald ein IRQ auftritt, die dazugehörende ISR der Applikationssoftware ausgeführt und nach deren Beendigung die Programmausführung fortgesetzt.

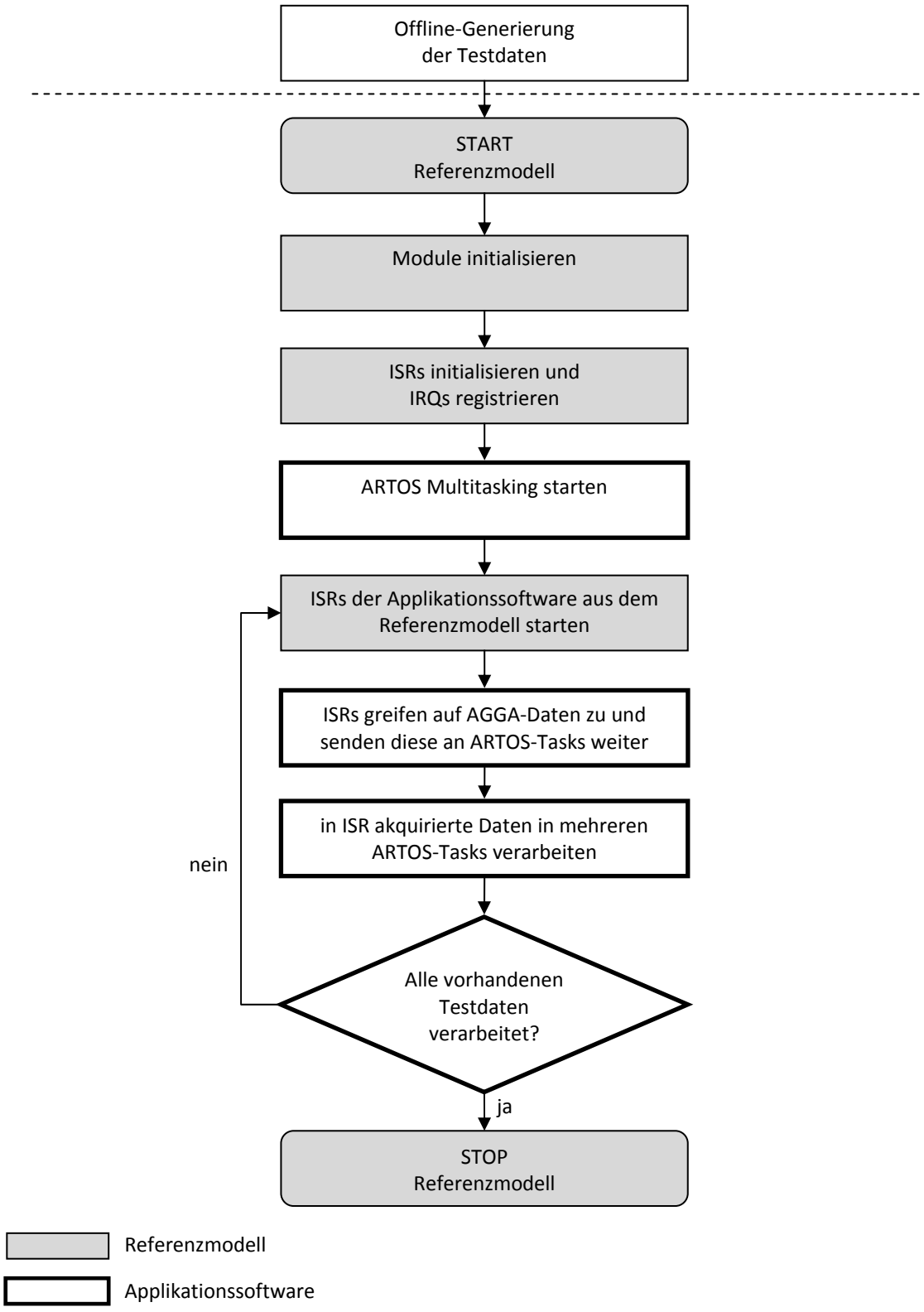

Abbildung 5: Ablauf des Referenzmodells und der Applikationssoftware.

Wie schon zu Beginn dieses Abschnitts kurz dargestellt, wird das Verhalten des AGGA durch das Referenzmodell simuliert. Die Applikationssoftware versorgt den AGGA mit Testdaten. Darunter versteht man die Ausführung der ISRs, die Daten vom AGGA lesen. Diese werden im Testbetrieb durch das Referenzmodell angestoßen. Die ISRs lesen die Daten aus den emulierten AGGA-Adressen aus und leiten sie über Message Queues oder Event Handling an die wartenden Threads der Applikationssoftware weiter.

Nach der Registrierung der IRQs erfolgt der System-Call zum Start des Multitaskings von ARTOS. Die Applikationssoftware wartet auf Messages aus den ISRs. Über eine Schleife – siehe Abbildung 5 – werden vom Referenzmodell die ISRs aufgerufen und die durch die ISRs akquirierten Daten in den ARTOS-Tasks verarbeitet. D. h. nach dem erfolgten Start des Multitaskings werden wieder alle, in der Zwischenzeit möglichen IRQs, behandelt und die Testdaten verarbeitet. Diese Schleife wird solange durchlaufen, wie Testdaten vorhanden sind. Nachdem alle Testdaten des Referenzmodells verarbeitet sind und alle ARTOS-Tasks beendet wurden, wird das gesamte Testszenario gestoppt.

Der RTC Scheduler in ARTOS/n läuft als Endlosschleife und ruft den Task-Dispatcher auf, sobald der soeben laufende Task terminiert/blockiert. Der Task-Dispatcher bestimmt den höchstprioren lauffähigen Task. Lässt sich so ein Task bestimmen, wird dieser Task aufgerufen.

Damit das Referenzmodell die ISRs der Applikationssoftware aufrufen kann, umgeht die Simulation den originalen RTC Scheduler von ARTOS/n und ruft direkt den Task-Dispatcher auf. D. h. das gesamte Referenzmodell und alle ISRs laufen, aus der Sicht von Solaris, als ein einziger Thread, obwohl dieser sequentiell mehrere ARTOS/n-Tasks abarbeitet. Dieser Solaris-Thread behandelt also alle Interrupts, verarbeitet die Testdaten und steuert somit den Ablauf von ARTOS.

3.3 Die Applikationssoftware

ARTOS/n stellt die von der Applikationssoftware benötigten Ressourcen, beispielsweise Partitionen (Partitions), zur Verfügung. Partitionen erlauben die Allokation und Freigabe von Speicherplatz fixer Größe aus einem statischen Pufferspeicher. In ARTOS müssen diese Partitionen ausschließlich vor dem Start des Multitaskings erstellt werden.

Diese Partitions haben den Vorteil einer besseren Performance, d. h. einer schnelleren Behandlung, im Vergleich zu den Funktionen `malloc` oder `free` der Standard-Bibliotheken, weil alle Partitions dieselbe, im Vorhinein festgelegte, Größe haben. Außerdem lassen sich so mögliche Fehler, z. B. Memory Leaks oder Null Pointer, die aufgrund der inkorrekten Verwendung von `malloc` entstehen, leicht erkennen.

Die Applikationssoftware ist stark Interrupt-getrieben. Vom AGGA generierte Interrupts lesen Daten aus, diese werden an ARTOS-Tasks übergeben. Die Tasks verarbeiten die Daten und blockieren nach erfolgreicher Erledigung. Dieser Vorgang iteriert.

Ein besonderes Prinzip kommt beim Versand von Messages zur Anwendung. Denn es werden in der GPS-Software aus Performance-Gründen nicht Nutzdaten in Message Queues versendet, sondern Pointer auf Partitions, die die Nutzdaten enthalten. Die Synchronisation von ARTOS-Tasks erfolgt oftmals mit Hilfe solcher Message Queues.

4 Test des RTOS mit dem Intel Thread Checker

Im Rahmen der Entwicklung von ARTOS wurden seitens RAA zwei Testansätze für die Überprüfung von ARTOS auf mögliche Data Races entwickelt, TAW (Thread Application Wrapper) und DuSCa (Duplikation der System-Calls). Die Ergebnisse dieser Überprüfungen waren mit dem Intel Thread Checker zu untersuchen und einer der Ansätze sollte für die Portierung nach Solaris ausgewählt werden.

In den folgenden Abschnitten wird gezeigt wie mit dem Intel Thread Checker die beiden Ansätze umgesetzt wurden und es wird eine Analyse und ein Vergleich der beiden Ansätze angestellt. Dieser Vergleich ermöglichte im Folgenden auch die Auswahl des Testansatzes zur Portierung der bestehenden Testsoftware für den Thread Analyzer von Sun.

4.1 Thread Application Wrapper

Beim Ansatz Thread Application Wrapper (TAW) werden – wie unter Abschnitt 1.2 kurz erläutert – ARTOS-Tasks durch Windows-Threads ersetzt. Dadurch wird es möglich, eine ARTOS-Applikation (mit Ausnahme der Module Interrupt und Task) unter Microsoft Windows laufen zu lassen. Abbildung 6 zeigt den Zusammenhang der einzelnen Module in ARTOS und wie anstelle des ARTOS-Schedulers, der Scheduler des Host-Betriebssystems, im konkreten Fall Microsoft Windows, verwendet wird. Die ARTOS-Module im grauen Oval werden dabei durch Testfunktionen (Stubs [12]) ersetzt.

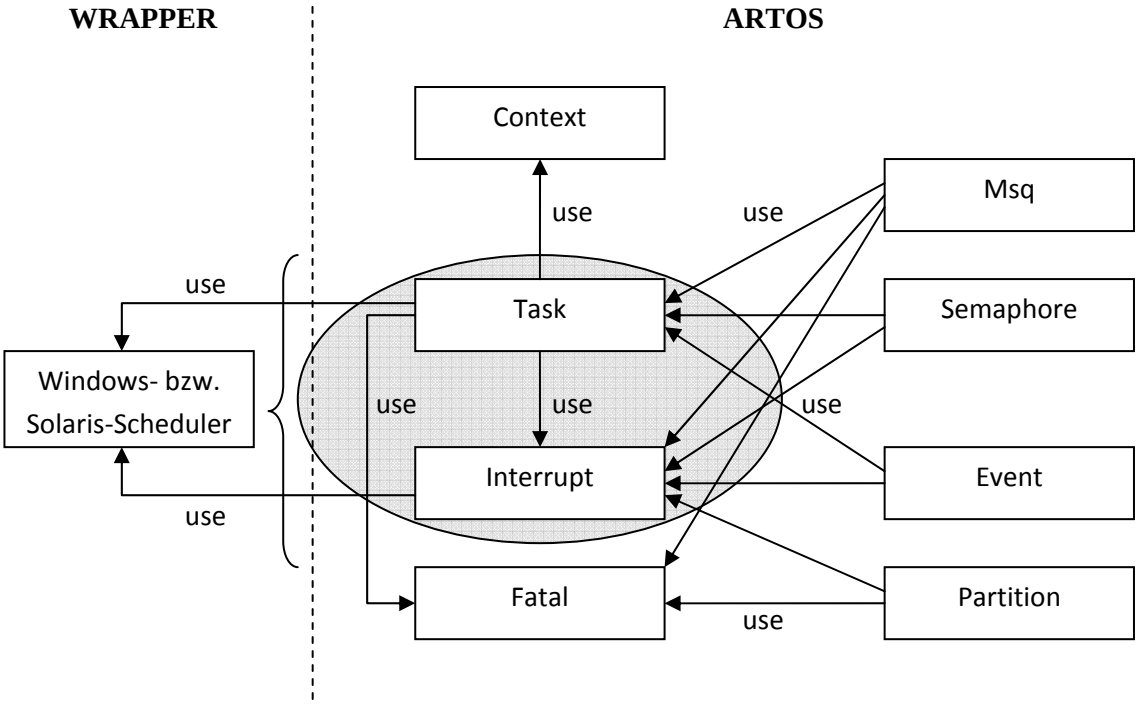

Abbildung 6: Module in ARTOS und das Wrapping des Schedulers.

Wie schon in Abschnitt 3.1 erwähnt, werden kritische Abschnitte im Code in ARTOS durch das Deaktivieren und anschließendes Reaktivieren von Interrupts geschützt. Da das Interrupt-Modul in TAW aber als Stub ausgeführt ist, muss das Schützen kritischer Abschnitte mit Mitteln des Host-

Betriebssystems passieren. Dies geschieht durch die Verwendung von Mutexes (Abkürzung für Mutual Exclusion Locks).

In den dafür benötigten Funktionen `INTERRUPT__Disable` und `INTERRUPT__Enable` werden mit Hilfe des Mutex `hArtosMutex`, Threads in ihrer Ausführung gehindert, bzw. durch anschießende Freigabe mit `ReleaseMutex`, blockierte Threads wieder lauffähig gemacht. Listings B3 und B5 (im Anhang B) illustrieren diese beiden Funktionen im Detail.

Listing 2 zeigt die Original-Funktion `TASK__StartMultiTasking` für das Starten des Multitaskings in ARTOS/n. Es wird hier in einer Endlosschleife der Task-Dispatcher aufgerufen, der den höchstprioren lauffähigen Task auswählt und ausführt.

```
void TASK__Dispatch(void)
{
    unsigned uiTaskIndex;

   /*
    * The task running lately has completed. Search for the task with
    * highest priority not blocked and ready.
    */

   ptThreadExecuting = &tTcbOfSystemIdleTask;

   for (uiTaskIndex = 0; uiTaskIndex < ARTOS__MAX_NOF_TASKS;
        uiTaskIndex++)
   {
      if (TASK_STATE__READY == atTcbPool[uiTaskIndex].eCurrentState)
      {
         /* we found the new task that we will run */
         ptThreadExecuting = &atTcbPool[uiTaskIndex];
         /* leave for loop */
         break;
      }
   }

   /* run the non-blocking task and wait until it returns */
   (ptThreadExecuting->pfnTaskEntryPoint)(0);
   return;
} /* TASK__Dispatch() */



void TASK__StartMultiTasking(void)
{
   FATAL__ASSERT(false == bMultiTaskingStarted, ARTOS_MODULE_ID__TASK);

   bMultiTaskingStarted = true;

   for (;;)
   {
      TASK__Dispatch();
   }

} /* TASK__StartMultiTasking() */
```

Listing 2: Der Task-Dispatcher und das Multitasking in ARTOS/n.

Nachdem der Task seinen Lauf beendet hat, kehrt die Funktion `TASK__Dispatch` zu `TASK__StartMultiTasking` zurück und die Suche und Ausführung des nächsten Tasks beginnt erneut.

Diese Funktionen aus Listing 2 werden nun im Ansatz TAW mit Hilfe von Windows-Funktionen gewrapped. Der Code-Ausschnitt in Listing 3 zeigt die Funktion `TASK__StartMultiTasking` in der für TAW modifizierten Version.

```
void TASK__StartMultiTasking(void)
{
   // pointer to the task control block (TCB) used to manage a task
   TASK__T_TaskControlBlock* ptMyThreadExecuting;
   DWORD dwThreadId;
   bMultiTaskingStarted = true;

   for (;;)
   {
      /* Pointer to the TCB of the task that could be running next */
      TASK__T_TaskControlBlock* ptAnyThread;

      /* Default: No new task is ready to run; either a deadlock
       * happened or we need to wait for an interrupt service routine to
       * execute. Run the system idle task.*/
      ptMyThreadExecuting = &tTcbOfSystemIdleTask;
      /* Loop through the TCBs to see if we find a new task to run */

      ...

      if (ptMyThreadExecuting == &tTcbOfSystemIdleTask)
      {
         /* selected the idle task */
         break;
      }

      ...

      ahThreads[uiWinThreads] = CreateThread(0, 0, ThreadStarter,
                                &ptMyThreadExecuting->uiPriority,
                                0, dwThreadId);
      if (ahThreads[uiWinThreads] == 0)
      {
         printf("Cannot start thread!\n");
         exit(1);
      }

      adwThreadIds[uiWinThreads] = dwThreadId;
      auiArtosTaskId[uiWinThreads] = ptMyThreadExecuting->uiObjectId;
      uiWinThreads++;
   }

   /* create IdleThread and set priority */
   ahThreads[uiWinThreads] = CreateThread(0, 0, IdleThread, 0, 0, 0);
   SetThreadPriority(ahThreads[uiWinThreads], THREAD_PRIORITY_IDLE);

   /* wait until threads complete */
   WaitForMultipleObjects(uiWinThreads, ahThreads, TRUE, INFINITE);

} /* TASK__StartMultiTasking() */
```

Listing 3: Aufruf der Windows-Funktionen im Rahmen des Multitaskings.

Nun läuft jeder ARTOS-Task in einem separaten Windows-Thread, in der gezeigten Funktion werden sämtliche Threads erzeugt. Bei Aufrufen der ARTOS-Funktion `TASK__Create` (entspricht in RTEMS der Funktion `rtems_task_create`) wurden Task-Eigenschaften in ARTOS/n-spezifischen Task Control Blocks abgelegt. Nun werden die in den Task Control Blocks abgelegten Daten verwendet und mit dem Aufruf der Windows-Funktion `CreateThread` Windows-Threads erzeugt. Nachdem alle Threads laufen, wird der IdleThread gestartet. Diesem wird mit der Windows-Funktion `SetThreadPriority` die niedrigste Priorität zugewiesen.

Durch den Aufruf der Windows-Funktion `WaitForMultipleObjects` wird nun solange gewartet, d. h. die ARTOS-Funktion `TASK__StartMultiTasking` wird solange ausgeführt, bis mindestens ein Thread abgeschlossen ist.

Das Wrapping der Funktion `TASK__UnBlock` (macht einen Task lauffähig) geschieht mit Hilfe von Tabellen, die die IDs der ARTOS/n-Tasks und Windows-Threads enthalten (`ahThreads[]` und `adwThreadIds[]`). In der Neuversion von `TASK__UnBlock` wird in einer Schleife überprüft, welcher ARTOS/n-Task mit welchem Windows-Thread korrespondiert. Wird hier eine Übereinstimmung gefunden, ist so der nächste lauffähige Task bestimmt. Listing A1 in Anhang A zeigt die Details.

Der für diesen Test bei RAA entwickelte Testtreiber führt alle ARTOS System-Calls mit 100% Verzweigungsabdeckung aus. Das gesamte Projekt TAW, das wie DuSCa diesen Testtreiber verwendet, wurde in Microsoft Visual Studio 8.0 übersetzt und mit dem Intel Thread Checker auf mögliche Data Races getestet. Im Thread Checker wurden nur die tatsächlich für ARTOS relevanten Funktionen überprüft, das heißt die Funktionen der Testumgebung wurden nicht instrumentiert.

Eine vom Thread Checker bei jedem Testlauf gezeigte Warnung ist in Abbildung 7 zu sehen.

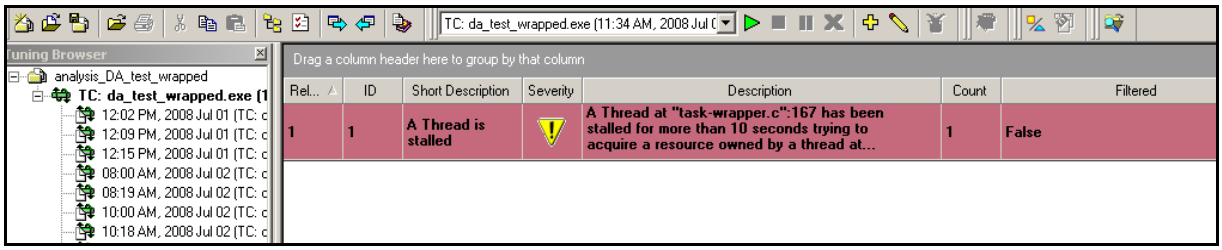

Abbildung 7: Ergebnis der Analyse von TAW im Intel Thread Checker.

Der genaue Wortlaut: *“A Thread at „task-wrapper.c“:168 has been stalled for more than 10 seconds trying to aquire a resource owned by a thread at “task-wrapper.c”:168”*, bezieht sich auf den Umstand, dass laut Thread Checker ein Thread für 10 Sekunden angehalten bzw. durch einen anderen Thread blockiert wurde. Die Zeile (168), die der Thread Checker hier angibt, befindet sich innerhalb der Funktion `TASK__GetNewTcb`. Diese Funktion akquiriert einen neuen Task Control Block und wird immer dann aufgerufen, wenn `TASK__Create` (`rtems_task_create`) ausgeführt wird, und dies geschieht im Testprogramm allerdings nur in der Startphase, und insgesamt nur fünf Mal.

Mit Hilfe des Debuggers in Visual Studio konnte nachgewiesen werden, dass ab dem Zeitpunkt, wo alle Threads laufen, die Funktion `TASK__GetNewTcb` nicht mehr aufgerufen wird. Daraus konnte geschlossen werden, dass die Zeilenangabe in dieser Warnung sich auf einen anderen Aufruf beziehen musste. Dabei kann es sich nur um die Ausführung der Funktion `WaitForMultipleObjects` handeln, da hier, wie eingangs beschrieben, tatsächlich ein Thread solange wartet, bis das Kriterium

für die Beendigung des Tests erfüllt ist. Somit konnte die angegebene Warnung als nicht korrekt und nicht relevant eingestuft werden.

Die einzige potentielle Race Condition, die im Testansatz TAW für ARTOS/n gemeldet wurde, betraf die momentane Anzahl von Nachrichten in der Message Queue. Das Ergebnis im Thread Checker zeigt Abbildung 8.

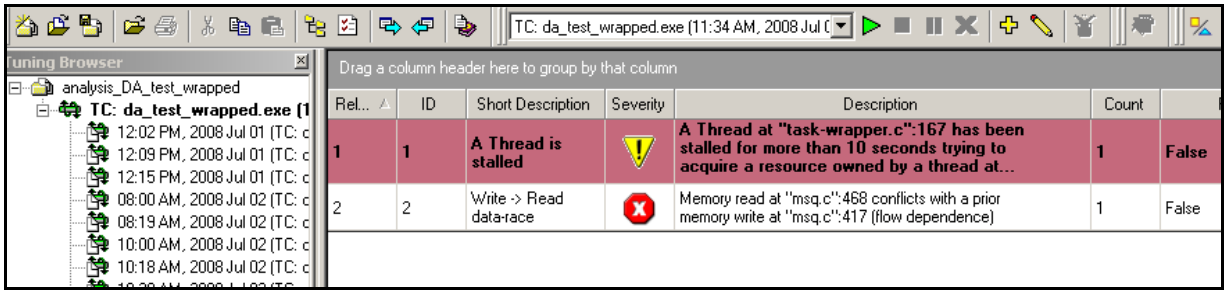

Abbildung 8: Potentielles Data Race im Testansatz TAW.

In der Funktion `MSQ__Urgent`, die einem Message Send mit hoher Priorität entspricht, wird die momentane Anzahl der in der Message Queue verweilenden Nachrichten inkrementiert (*memory write at „msq.c“:417*). In der Zwischenzeit kann dieser Wert allerdings mit Hilfe der Funktion `MSQ__GetNumberOfPendingMsg` ausgelesen worden sein (*memory read at „msq.c“:468*). Das bedeutet aber, dass die ausgelesene Anzahl der vorhandenen Nachrichten falsch ist.

Dieses korrekt angezeigte Data Race entsteht deswegen, da es dem User überlassen ist, ob diese Abfrage nach den Pending Messages in einer Critical Section ausgeführt wird oder nicht. Die Funktion `MSQ__GetNumberOfPendingMsg` sagt aus, wieviele Nachrichten sich *genau in diesem Moment* in der Queue befinden. Dieser *Moment* ist aber schon nach der Funktionsausführung wieder vorbei und nach diesem Aufruf, wenn dieser ungeschützt stattgefunden hat, kann man sich nicht mehr darauf verlassen, dass die Anzahl der vorhandenen Nachrichten unverändert ist. Das Data Race kann daher als unkritisch eingestuft werden.

Listing A1 in Anhang A veranschaulicht alle Funktionen des Wrapper-Moduls task-wrapper.c, das in TAW zum Einsatz kommt. Die darin angeführten Code-Abschnitte zeigen, wie mit Hilfe der Windows-Thread-API, die Funktionen zum Management der ARTOS-Tasks gewrapped werden.

4.2 Duplikation der System-Calls

Beim Testansatz durch Duplikation der System-Calls (DuSCa) wird ARTOS in drei Windows-Threads ausgeführt. Vor dem Start der Windows-Threads werden sämtliche ARTOS-Tasks initialisiert und gestartet. Danach erfolgt der Aufruf der Windows-Threads. Dabei läuft der ARTOS-Scheduler in einem einzelnen Windows-Thread und verwaltet drei ARTOS-Tasks, die auf Events, Semaphore und Message-Queue-Pakete warten. Zwei andere Windows-Threads laufen dazu parallel und versorgen die drei anderen ARTOS-Tasks mit Events, Semaphor-Updates und Messages.

Abbildung 9 stellt den Aufbau dar. Der Windows-Thread `Dispatcher` führt das Scheduling von ARTOS aus. Die beiden anderen Windows-Threads `SndTaskWrapper1` und `SndTaskWrapper2` führen parallel zueinander System-Calls durch. Beispielsweise `MSQ__Send`-, `EVENT__Send`- oder `SEM__Release`-Operationen, die an ARTOS-Tasks, die im Windows-Thread-Dispatcher laufen, gesendet werden. Das heißt, an die laufenden ARTOS-Tasks werden Events, Nachrichten, etc., von zwei unterschiedlichen Quellen gesendet.

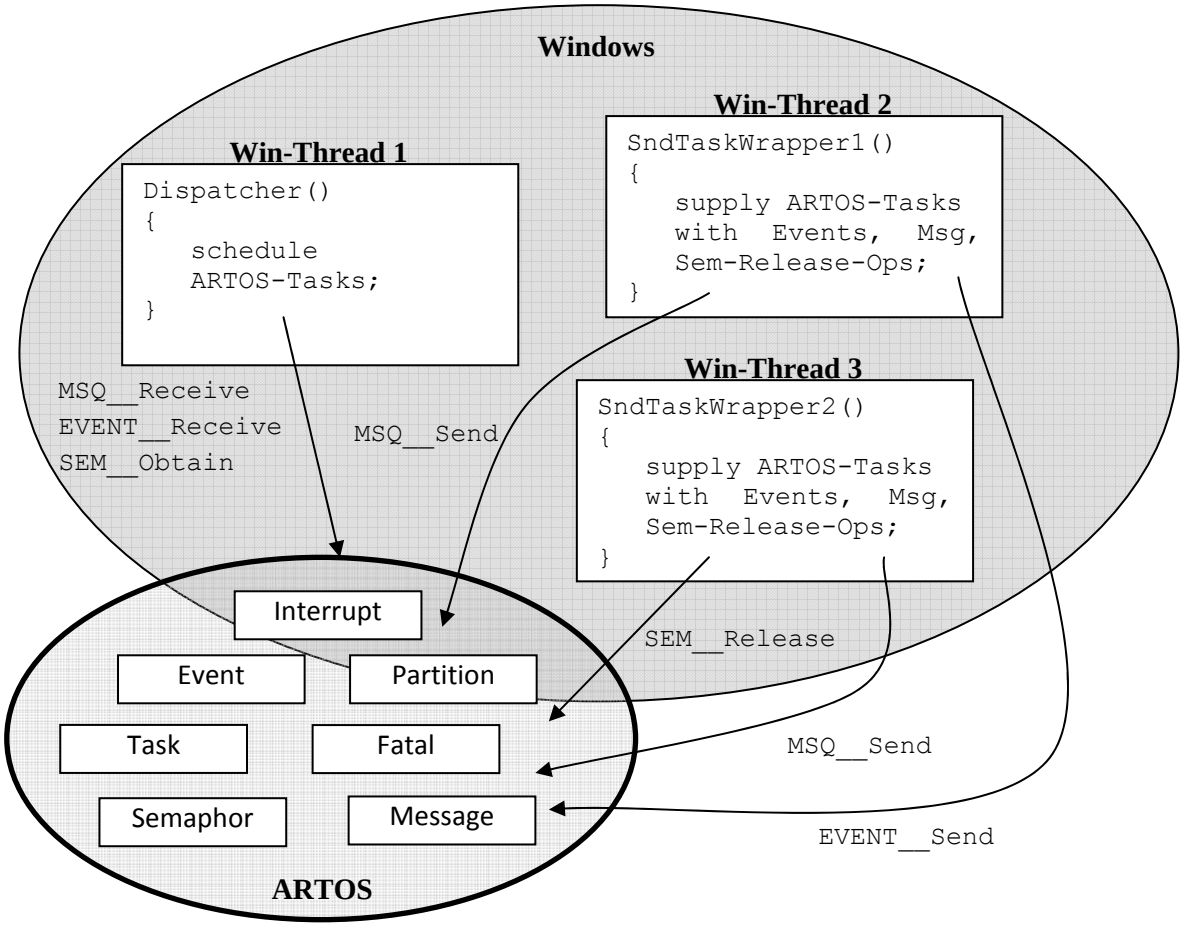

Abbildung 9: Aufbau des Testansatzes DuSCa. Die Funktion `Dispatcher()` führt das gesamte ARTOS-Scheduling aus, die beiden anderen Windows-Threads versorgen die ARTOS-Tasks mit Daten.

Wie beim Ansatz TAW ist auch hier das Interrupt-Modul durch einen Stub ersetzt. Die Interrupt-Funktionen zum De- und Reaktivieren werden, wie in Abschnitt 4.1 beschrieben, mit der Hilfe von Mutexes realisiert. Listings B3 und B7 in Anhang B zeigen die beiden Funktionen.

Die Funktionen der Testumgebung werden von der Überprüfung ausgeklammert und daher nicht instrumentiert. In Abbildung 10 ist das Ergebnis zu sehen, mit zwei[2] möglichen Race Conditions.

[2] Angezeigt werden in Abbildung 10 tatsächlich drei mögliche Data Races. Allerdings betreffen die Race Conditions ID 1 und ID 4 die gleichen Zeilen im Code, mit dem Unterschied, dass einmal ein Schreib-Lese-Zugriff und danach ein Lese-Schreib-Zugriff vorliegt. D. h. nur eine Umkehrung der Reihenfolge, die zum gleichen Ergebnis führt.

Rel...	ID	Short Description	Severity	Description	Count	Filtered
1	1	Write -> Read data-race		Memory read at "task-n.c":273 conflicts with a prior memory write at "task-n.c":564 (flow...	1	False
2	2	Write -> Read data-race		Memory read at "msq.c":468 conflicts with a prior memory write at "msq.c":341 (flow dependence)	2	False
3	3	Thread termination		Thread termination at "task-n.c":509 - includes stack allocation of 1 MB and use of 4 KB	1	False
4	4	Read -> Write data-race		Memory write at "task-n.c":564 conflicts with a prior memory read at "task-n.c":273 (anti dependence)	1	False
5	5	Thread termination		Thread termination at "task-n.c":509 - includes stack allocation of 1 MB and use of 4 KB	1	False

Abbildung 10: Erkannte Data Races mit Hilfe des Intel Thread Checkers im Testansatz DuSCa.

Die Einträge mit der ID 1 und 4 in Abbildung 10 beschreiben ein Data Race im ARTOS Task-Dispatcher, also der Funktion die den höchstprioren lauffähigen ARTOS-Task auswählt. Im realen Einsatz kann der Task-Dispatcher nicht parallel zu einer anderen ARTOS-Funktion ausgeführt werden. Im vorliegenden Testansatz laufen aber zwei Windows-Threads parallel, die die ARTOS-Tasks mit Events, Messages, etc. versorgen und daher kann eine ARTOS-Funktion zum Dispatcher parallel laufen. Somit war dieses gefundene Data Race zu vernachlässigen.

Das zweite erkannte mögliche Data Race (ID 2 in Abbildung 10) bezieht sich – wie schon bei den Ergebnissen beim Ansatz TAW – auf die Anzahl der momentan vorhandenen Messages (Pending Messages) in einer Message Queue. Mit Hilfe der Funktion `MSQ__GetNumberOfPendingMsg` wird diese Anzahl ausgelesen (*memory read at „msq.c“:468*). Allerdings wird hier, im Vergleich zum angezeigten Data Race in TAW, der gleiche Wert in der Funktion `MSQ__Send` inkrementiert (*memory write at „msq.c“:341*). Wie bei TAW ist die potentielle Race Condition, dass das Auslesen des Wertes für die Anzahl der vorhandenen Nachrichten in der Message Queue genau durch das Inkrementieren desselben unterbrochen wird.

Dieses angezeigte Data Race stellte – wie schon im Abschnitt 4.1 beschrieben – keine Gefährdung für die korrekte Programmausführung dar. Die Funktion zum Auslesen der momentan vorhandenen Nachrichten in der Message Queue dient Debug-Zwecken und es obliegt dem User diese Funktion zu schützen. Darum hat eine mögliche falsche Angabe keinen negativen Einfluss auf die allgemeine Funktionalität der Software. Dass im Ansatz TAW dieses Data Race in Verbindung mit der Funktion `MSQ__Urgent` und in DuSCa im Zusammenhang mit `MSQ__Send` auftritt, konnte nicht erklärt werden. Durchläufe der beiden Testansätze wurden mehrfach durchgeführt und brachten immer dasselbe Ergebnis. Auch sind die Testtreiber bei TAW und DuSCa identisch und somit konnte dieses Verhalten nur auf ein unterschiedliches Scheduling zurückgeführt werden.

Die angezeigten Informationen (ID 3 und 5 in Abbildung 10) bezogen sich lediglich auf den Umstand, dass für beendete Threads viel mehr Speicher allokiert wurde, als dann tatsächlich von diesen Threads benötigt worden ist.

4.3 Vergleich der Testansätze

Aus den beiden vorgestellten Testansätzen war einer auszuwählen und für Thread Analyzer unter Solaris zu portieren. Die Überlegungen für diese Auswahl waren wie folgt:

Beide Testansätze verwenden Windows-Funktionen, um das Multithreading von ARTOS zu simulieren. Der Ansatz DuSCa hat dabei den Nachteil, dass ARTOS-Funktionen parallel zum Task-Dispatcher laufen können, was in der Praxis nie der Fall sein wird. Dieser Nachteil beinhaltet allerdings den Vorteil von DuSCa gegenüber TAW, dass das Task-Management von ARTOS/n auch Teil des Testobjekts ist. Im Ansatz TAW werden die Tasks durch das jeweilige Gastbetriebssystem

gescheduled. Durch das Wrapping wird nicht der für das Task-Management zuständige Originalcode von ARTOS/n ausgeführt, sondern der Wrapper. Data Races im Task-Management von ARTOS/n können daher von TAW nicht erkannt werden.

Bei beiden Ansätzen wird die Behandlung der Interrupts durch einen Stub ersetzt. Alle ISRs müssen daher in Tests gesondert behandelt werden.

In punkto Testschärfe erzielen beide Ansätze die gleichen Ergebnisse. Beide finden das mögliche Data Race in Bezug auf die Anzahl der Nachrichten in der Message Queue. Die Data Races, die in DuSCa mit Hilfe des Thread Checkers zusätzlich noch gefunden werden, entstehen aufgrund der parallelen Versorgung der ARTOS-Tasks mit Daten im angesprochenen Testdesign und sind daher für den Vergleich bezüglich der Testschärfe nicht relevant.

Der große Vorteil von TAW im Vergleich zu DuSCa ist, dass TAW an der Schnittstelle zur Applikation nahezu eine 100%ige Imitation von ARTOS ist. Daher ist es auch möglich ARTOS in Verbindung mit einer Applikationssoftware zu testen, was bei DuSCa nicht machbar ist.

Der geringe Mehraufwand beim Test einer Applikation zu ARTOS im Testansatz TAW war dafür ausschlaggebend, den Ansatz TAW für die Portierung in das Betriebssystem Solaris auszuwählen.

5 Übernahme in die Solaris-Umgebung

Maßgeblich für die Notwendigkeit der Portierung des ausgewählten Testansatzes in die Solaris-Umgebung war der Umstand, dass das Referenzmodell ausschließlich in dieser Umgebung läuft.

Für die Bearbeitung des Source-Codes und der Durchführung der Data Race Tests wurde Sun Studio, eine integrierten Entwicklungsumgebung (IDE) mit C-Compiler für und unter Solaris, verwendet. In Sun Studio ist das Tool Thread Analyzer integriert, ein auf Eraser basierendes Werkzeug zum Auffinden von Data Races.

In den folgenden Abschnitten wird nun die Übernahme von TAW in das Betriebssystem Solaris beschrieben.

5.1 Threads der Betriebssysteme im Vergleich

Anstelle der vorher verwendeten Windows-Funktionen zur Thread-Behandlung, kommt nun die POSIX-Thread-API zum Einsatz. Die POSIX-Thread-API ist die auf UNIX-basierten Betriebsystemen übliche Applikationsschnittstelle für Multithreading. Die Portierung der Windows-Thread-API-Aufrufe auf die POSIX-Thread-API-Aufrufe wurde durch die Zuhilfenahme der Beispiele und Tutorials in [20] erleichtert. In [21] findet man eine gute Gegenüberstellung der Windows-Threads und der entsprechenden POSIX-Threads für die Verwendung in Solaris. In Tabelle 1 sind die wichtigsten Befehle der beiden Betriebssysteme zur Behandlung von Threads aufgelistet.

Tabelle 1: Windows- und POSIX-Threads im Vergleich.

Windows-Threads	**POSIX-Threads**
`CreateThread`	`pthread_create`
`GetCurrentThreadId`	`pthread_self`
`CreateMutex`	`pthread_mutex_init`
`WaitForSingleObject`	`pthread_mutex_lock`
`ReleaseMutex`	`pthread_mutex_unlock`
`WaitForMultipleObject`	`pthread_join`
`SetThreadPriortiy`	`pthread_attr_setschedparam` `pthread_attr_setschedpolicy`

Eine ausführliche Beschreibung der POSIX-Threads und detaillierte Anwendungsbeispiele findet man außerdem in [22]. Die in Tabelle 1 angeführte Gegenüberstellung soll aber nicht den Eindruck erwecken, dass die Überführung von TAW in das Betriebssystem Solaris 1:1 möglich ist. Im Zuge dieses praktischen Teiles der Diplomarbeit, waren einige Adaptierungen und zusätzliche Kunstgriffe, (z. B. in Form von Bibliotheken oder dem Wrapping von nicht unterstützten Funktionen) notwendig,

um die Funktionalität des portierten Ansatzes zu gewährleisten. Tabelle 1 soll nur einen kurzen Überblick bieten und erhebt keinen Anspruch auf Vollständigkeit.

Im Anhang B sind Listings zu finden, die Beispiele für die unterschiedliche Anwendung der POSIX-Thread-API im Vergleich zur Windows-Thread-API zeigen.

5.2 Thread-Prioritäten und Scheduling

Bei der Erstellung von Threads liegt der wesentlichste Unterschied, bei den beiden Betriebssystemen, in der Priorisierung. Während in der Windows-Variante die Prioritäten immer in einem Bereich von -15 (THREAD_PRIORITY_IDLE) bis +15 (THREAD_PRIORITY_TIME_CRITICAL) angesiedelt sind, hängen die Thread-Prioritäten in Solaris von der verwendeten Scheduling Policy ab. D. h. je nach ausgewähltem Scheduling verändern sich die Werte für den niedrigstprioren bzw. für den höchstprioren Thread. Die verfügbaren Policies für das Scheduling von POSIX-Threads sind in Tabelle 2 dargestellt.

Tabelle 2: Scheduling Policies für die Priorisierung von POSIX-Threads.

Scheduling Policy	**Prioritäten**
SCHED_OTHER (default)	-20 bis 14
SCHED_FIFO	0 bis 59
SCHED_RR	0 bis 59

SCHED_FIFO (First In, First Out) erlaubt einem Thread solange zu laufen, bis ein Thread mit einer höheren Priorität lauffähig (*ready*) ist, oder dieser Thread freiwillig die CPU abgibt. Wenn ein Thread in der SCHED_FIFO Policy *ready* wird, dann beginnt dieser sofort mit seiner Ausführung, außer es läuft bereits ein Thread mit gleicher oder höherer Priorität [22].

SCHED_RR (Round Robin) ist SCHED_FIFO sehr ähnlich, mit der Ausnahme, dass hier einem Thread in der Ausführung nur eine bestimmte Zeit (*time slice interval*) zur Verfügung steht. Wird diese Zeit überschritten und ist ein anderen Thread mit SCHED_FIFO oder SCHED_RR als verwendete Policy und derselben Priorität *ready*, dann wird der gerade laufende Thread angehalten und der neue, lauffähige Thread wird ausgeführt [22].

SCHED_OTHER ist genaugenommen keine Scheduling Policy. Die offizielle Erklärung ist, dass mit der Verwendung von SCHED_OTHER eine Portierung des Programms (wo SCHED_OTHER zum Einsatz kommt) möglich ist, aber SCHED_OTHER in Wahrheit nur angibt, dass dieses Programm kein Echtzeit-Scheduling benötigt. SCHED_OTHER kann als Ersatz von SCHED_FIFO oder SCHED_RR angesehen werden, oder als etwas komplett anderes. Es gibt keine klare Definition [22].

Werden Threads angehalten und anschließend wieder fortgesetzt, so werden die Threads bei SCHED_FIFO und SCHED_RR in der Reihenfolge ihrer Prioritäten aus der Warteschlange geholt. Das heißt der Thread mit der höchsten Priorität wird, unabhängig von seiner Position in der Warteschlange, wieder als Erster ausgeführt [22].

Das ist allerdings nicht das gewünschte Verhalten, wenn wir ARTOS-Tasks in Solaris Threads einbetten. Denn im Rahmen dieses Tests von ARTOS/n, müssen *bestimmte* Threads, die angehalten

wurden, explizit wieder fortgesetzt werden können, damit der Wrapper unter Zuhilfenahme von Tabellen (enthalten die ARTOS-Task IDs bzw. POSIX-Thread IDs) weiß, welcher Thread lauffähig wurde. Warten mehrere Threads vor einer Critical Section, dann kann hier im Modell von ARTOS/n einem ausgewählten Thread erlaubt werden, diese zu betreten. SCHED_FIFO oder SCHED_RR bieten, wie schon erwähnt, diese Funktionalität per se nicht an.

Darum kommt hier die POSIX-Funktion `pthread_kill` zum Einsatz, um ein Signal an einen ausgesuchten Thread senden zu können. So wird durch die Verwendung von userspezifischen Signalen ermöglicht, bestimmte Threads wieder lauffähig zu machen, beschrieben in Abschnitt 5.4. Dadurch wird das an sich korrekte Verhalten von SCHED_RR und SCHED_FIFO verhindert.

Beide Scheduling Policies weisen das gleiche Prioritäten-Intervall (siehe Tabelle 2) auf. Da aber im vorliegenden Modell, kein ARTOS-Task bzw. POSIX-Thread dieselbe Priorität aufweist, würden beide Scheduling Policies zum selben Ergebnis führen. Dieses Verhalten wurde in den Ergebnissen einiger Testläufe nachgewiesen. Die endgültige Auswahl von SCHED_RR hatte den Hintergrund, dass bei allfälligen Erweiterungen durch diese Policy, die Wahrscheinlichkeit für eine Dauerblockade von Threads kleiner ist.

Listing 4 zeigt die Funktion `TASK__StartMultiTasking` (Listing 3 stellt dieselbe Funktion in der Windows-Variante dar) für die weiteren Tests in Sun Studio, in der geänderten Form unter Verwendung der POSIX-Threads. Der Code-Ausschnitt zeigt aber lediglich den Teil der Priorisierung und der Erstellung der Threads.

```
void TASK__StartMultiTasking(void)
{
   ...

   for (;;)
   {
      ...

      /* change the priority of the scheduling struct to the priority
       * of the selected thread */
      param.sched_priority = ptMyThreadExecuting->uiPriority;
      // set the thread priority to the selected value
      rc = pthread_attr_setschedparam(&attr, &param);
      if (rc != 0)
         err_abort(status, "pthread_attr_setschedparam");
      rc = pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
      if (rc != 0)
         err_abort(status, "pthread_attr_setdetachstate");
      // set the selected policy (=2 => RR)
      rc = pthread_attr_setschedpolicy(&attr, policy);
      if (rc != 0)
         err_abort(status, "pthread_attr_setschedpolicy");
      /* set the inheritsched attribute or the policy and priority
       * which was set will be ignored */
      rc = pthread_attr_setinheritsched(&attr, PTHREAD_EXPLICIT_SCHED);
      if (rc != 0)
         err_abort(status, "pthread_attr_setinheritsched");
```

```
      // create the thread with the adjusted parameters
      iPThreads[uiPThreads] = pthread_create(&tids[uiPThreads], &attr,
                              ThreadStarter,
                             (void*) &ptMyThreadExecuting->uiPriority);

        ...
    }

  ...

} /* TASK__StartMultiTasking() */
```

Listing 4: Erstellung der POSIX-Threads in `TASK__StartMultiTasking`. Mit Hilfe spezieller Attribute werden die Scheduling Policy und die Priorität angegeben.

Wie in Listing 4 zu sehen ist, unterscheidet sich der Aufruf zum Erstellen der einzelnen Threads erheblich von der Windows-Variante. Die Angabe der Priorität muss vor der eigentlichen Ausführung von `pthread_create` explizit über die Angabe von Attributen erfolgen. Außerdem wird in den Attributen zusätzlich festgelegt, dass die Threads „joinable“ erstellt werden. D. h. nachdem alle Threads laufen, kann mittels `pthread_join` analog zur Windows-Funktion `WaitForMultipleObjects` auf die Beendigung der Threads gewartet werden. Wichtig ist auch die Verwendung von PTHREAD_EXPLICIT_SCHED. Ohne diese Verwendung würden alle Threads die erstellt werden, die Scheduling Policy und die Priorität vom ausführenden Thread, d. h. von jenem Thread der alle anderen erstellt, erben.

5.3 Interrupt-Behandlung

Eine weitere Abweichung im Zuge der Portierung von Windows nach Solaris ergab sich bei der Ersetzung der Windows-Funktion `WaitForSingleObject`, im Rahmen der Handhabung von Critical Sections, durch den POSIX-Befehl `pthread_mutex_lock`. Wie schon in Abschnitt 4.1 beschrieben, werden Mutexes verwendet, um kritische Abschnitte mit Mitteln des Host-Betriebssystems zu schützen. Eine Critical Section wird so durch das Erstellen eines Locks vor weiteren Zugriffen durch andere Threads geschützt und kann erst nach Freigabe des Locks (kann nur durch den Thread erfolgen, der das Lock hält), von einem anderen Thread betreten werden.

`WaitForSingleObject` und `pthread_mutex_lock` werden im Stub von `INTERRUPT__Disable` benutzt, um Critical Sections zu schützen, siehe Listings B3 und B4 im Anhang B. Ruft nun der Thread, der dieses Lock hält, diese Funktion erneut auf, so hat dies in der Windows-Variante keine Konsequenzen. D. h. `WaitForSingleObject` kann fortlaufend einige Male vom selben Thread aufgerufen werden, ohne zu blockieren. In Solaris hat ein nochmaliger Aufruf von `pthread_mutex_lock` durch den laufenden Thread allerdings zur Folge, dass sich dieser Thread selbst sperrt, d. h. es entsteht ein Deadlock. Der folgende Absatz zeigt die implementierte Lösung des Problems.

Der mehrmalige Aufruf von `INTERRUPT__Disable` passiert zum Beispiel dann, wenn eine Critical Section innerhalb einer Funktion betreten wird und in dieser eine andere Funktion mit einer weiteren Critical Section aufgerufen wird. Somit kommt es zu einer erneuten Ausführung und zu einer Erhöhung der Verschachtelungstiefe (Nesting Level) von `INTERRUPT__Disable` (Variable `uiLockNestingLevel`, siehe Listing B3 im Anhang B). Dieser Level gibt an, wie oft dieser Thread `INTERRUPT__Disable` aufgerufen hat, ohne ein korrespondierendes `INTERRUPT__Enable`. Ruft dieser Thread nun `INTERRUPT__Enable` auf, dann wird der Nesting Level wieder dekrementiert.

Solange dieser Level nicht auf Null ist, wird das Lock nicht freigegeben. Erst wenn der ausführende Thread die äußerste Critical Section verlassen hat und mit der Ausführung von `INTERRUPT__Enable` der Nesting Level Null wird, erhält ein neuer Thread die Möglichkeit zu laufen zu beginnen.

5.4 Suspending und Resuming von Threads

Ebenfalls eine Herausforderung war das gezielte Anhalten und Fortsetzen von bestimmten Threads. Die Windows-API stellt hier zwei Funktionen zur Verfügung, `SuspendThread` und `ResumeThread`, mit denen man einen ausgewählten Thread solange anhalten kann, bis man diesen wieder explizit fortsetzt. Diese Funktionalität wird im Testansatz TAW benötigt. Denn hier läuft der Task-Dispatcher von ARTOS nicht, der erkennt, wann ein Thread suspendiert werden muss, bzw. wieder lauffähig ist. Daher muss in TAW der Wrapper für den Scheduler, mit Hilfe der oben genannten Funktionen, den Scheduler des Host-Betriebssystems dazu bringen, dass die Threads in ihrer Ausführung unterbrochen und anschließend wieder fortgesetzt werden.

Anhang C zeigt die Funktionen, mit deren Hilfe es gelang, dass ein POSIX-Thread sich selbst oder einen anderen POSIX-Thread anhält und wieder fortsetzt. Dabei kann mit Hilfe des POSIX-Befehls `pthread_kill` und unter Verwendung von zwei userspezifischen Signalen (SIGUSR1 und SIGUSR2) ein bestimmter Thread in seiner Ausführung angehalten und anschließend wieder fortgesetzt werden. Die ursprüngliche Form dieser Funktionen aus [22] musste so angepasst werden, damit es auch möglich war, dass sich Threads selber anhalten können. Diese werden dann von anderen, lauffähigen Threads, wieder fortgesetzt.

Die Verwendung der POSIX-Funktionen `pthread_cond_wait` (Thread wartet auf ein Signal, um fortsetzen zu können) und `pthread_cond_signal` (versenden des Signals, auf das einer oder mehrere Threads warten) war nicht möglich, denn so kann kein *ausgewählter* Thread reaktiviert werden. Denn sobald mehrere Threads auf das Signal zum Weitermachen warten, würde der höchstpriore Thread in der Warteschlange zum Zug kommen.

6 Tests mit Thread Analyzer

Mit Hilfe des in Sun Studio integrierten Tools Thread Analyzer, kann die Ausführung eines Multithreading-Programms analysiert werden. D. h. es können mögliche Data Races und Deadlocks (deren Analyse wurde im Rahmen dieser Diplomarbeit nicht berücksichtigt) in einem Programm, das POSIX-Threads verwendet, entdeckt werden. Um diese Überprüfung möglich zu machen, wird der Code im Zuge des Kompilierens instrumentiert.

In Abschnitt 6.1 wird die Untersuchung von ARTOS mit dem portierten Testansatz TAW im Thread Analyzer beschrieben und die Ergebnisse werden bewertet. Danach erfolgt im nächsten Abschnitt die Beschreibung der Analyse der Gesamtsoftware, d. h. die Applikationssoftware wird im Referenzmodell, das die Simulationsumgebung für diese auf ARTOS aufbauende Software bildet, im Thread Analyzer einer Überprüfung auf mögliche Race Conditions unterzogen. Das Verhalten von ARTOS/n wurde hier wieder durch die Verwendung von TAW simuliert. Alle angezeigten Data Races werden hinsichtlich ihrer potentiellen Gefährlichkeit bewertet. Falschmeldungen durch das Tool werden üblicherweise als „False Positives“ bezeichnet.

6.1 Analyse von ARTOS/n mit TAW im Thread Analyzer

Im Vergleich zu den Ergebnissen des Intel Thread Checkers (siehe Abschnitt 4.1), meldete der Sun Thread Analyzer, zusätzlich zu allen bisher bekannten Meldungen, noch weitere mögliche Race Conditions. Abbildung 11 zeigt einen Auszug der Auflistung, der mit dem Thread Analyzer durchgeführten Überprüfung.

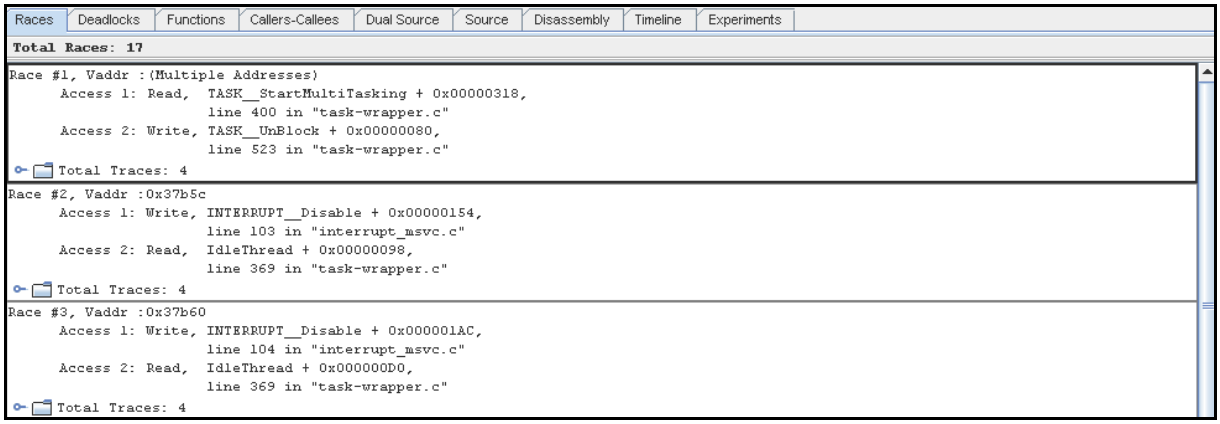

Abbildung 11: Ergebnisse der Untersuchung des Testansatzes TAW auf potentielle Data Races im Thread Analyzer.

Thread Analyzer bietet in den Ansichten *Functions* und *Callers-Callees* die Möglichkeit, alle Funktionen die bis zum Auftritt des Data Races aufgerufen und ausgeführt werden, aufzulisten. D. h. man kann die Entstehung des möglichen Fehlverhaltens von Anfang bis Ende nachverfolgen. *Dual Source* zeigt die betroffenen Befehlszeilen im Source-Code der Module, wo die Ausführung der beiden unterschiedlichen Threads zum Data Race führt.

Insgesamt wurden 17 mögliche Data Races gefunden. 14 dieser angezeigten Treffer bezogen sich auf den Wrapper selbst und waren daher für die Bewertung nicht relevant. Zwei der gefundenen Data Races entsprachen jenen, die schon mit Hilfe des Thread Checkers entdeckt wurden. Dabei handelte es sich, wie in den Abschnitten 4.1 und 4.2 erläutert, um die mögliche falsche Anzahl, der momentan in

einer Message Queue befindlichen Nachrichten, bei der Ausführung der Funktionen `MSQ__Send` bzw. `MSQ__Urgent`.

Ein bislang unbekanntes, mögliches Data Race fand der Thread Analyzer im Modul Event. Abbildung 12 zeigt die Dual Source-Ansicht der ausführenden Threads in den Funktionen `EVENT__Receive` und `EVENT__Send`.

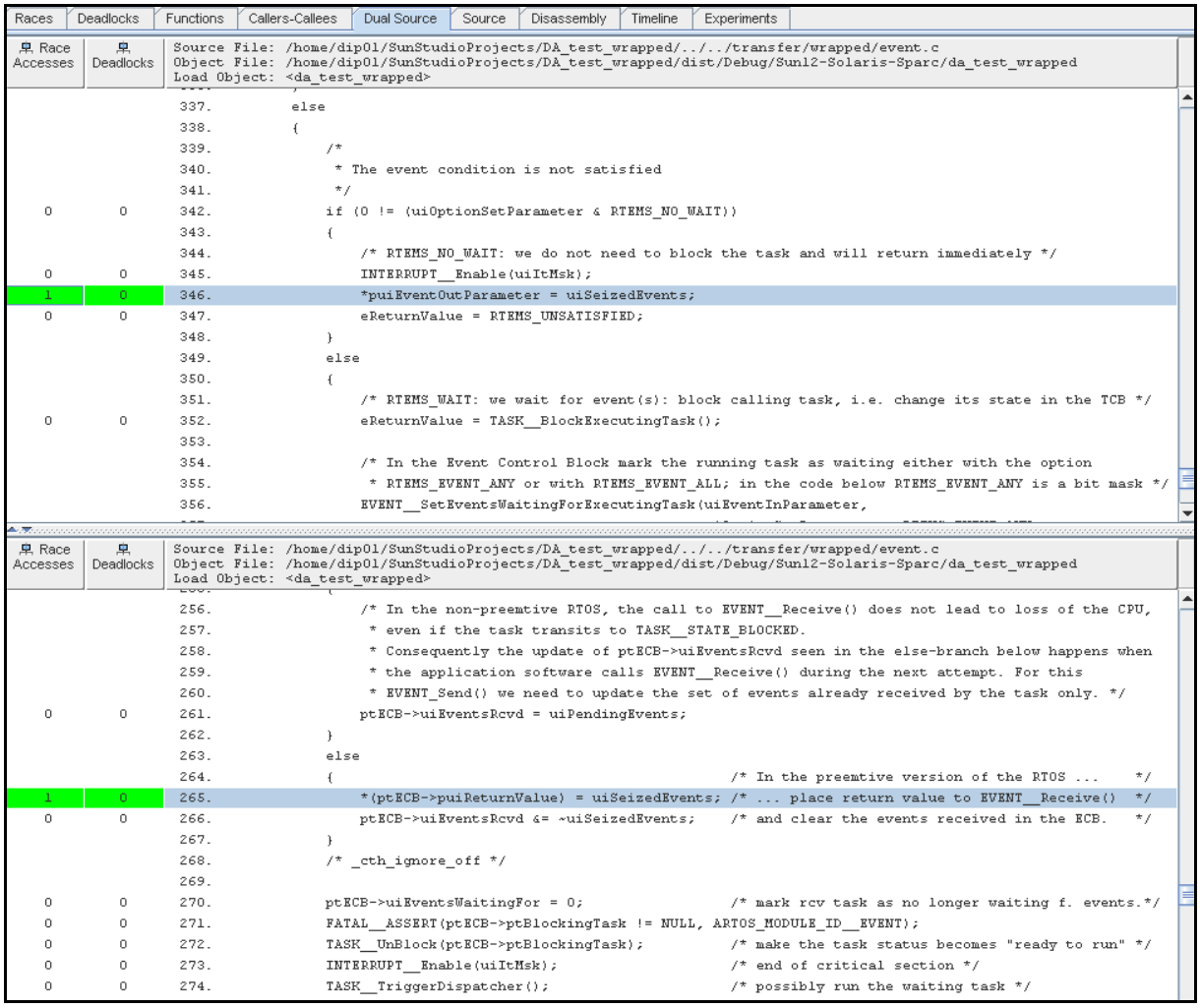

Abbildung 12: Potentielles Data Race im Modul Event. Dual Source-Ansicht im Thread Analyzer.

Die Funktion `EVENT__Send` führt ein Update des Event Sets, d. h. die Menge aller Events, auf die ein bestimmter Task wartet, aufgrund von neu empfangenen Events, durch. `EVENT__Receive` blockiert den Task solange, bis alle Events, auf die der Task wartet, eingetroffen sind.

Listing 5 und Listing 6 zeigen die beiden Funktionsprototypen von `EVENT__Send` und `EVENT__Receive`. Der Parameter `uiTaskID` in `EVENT__Send` in Listing 5 gibt die ID des Tasks an, an den Events gesendet werden sollen. `uiEvents` ist ein Bitmuster, das definiert, um welche Events es sich handelt.

```
/* _____ P u r p o s e _____
 * This routine sends an event set uiEvents to the task with ID
 * uiTaskID. If a blocked task's input event condition is satisfied by
 * this action, then it will be made ready. If its input event condition
 * is not satisfied, then the events sent by this call will be
 * remembered. Identical events sent to a task are not queued.
 * In ARTOS/p the calling task will get preempted if a task of higher
 * priority is unlocked as a result of this directive.
 *
 * ______ I n p u t _____
 * uiTaskID  - The ID of the task the event shall be sent to;
 * uiEvents  - The bit set of event(s) sent to the task; a set bit
 *             corresponds to an event flagged; see macros
 *             RTEMS_EVENT_0, RTEMS_EVENT_1, ...
 *
 * ______ O u t p u t _____
 * -
 *
 * ______ R e t u r n _____
 * RTEMS_SUCCESSFUL  - when the event could be sent
 * RTEMS_INVALID_ID  - if uiTaskID does not belong to an existing task
 */
RTEMS__E_StatusCode EVENT__Send(
   unsigned32 uiTaskID /*in*/,
   unsigned32 uiEvents /*in*/);
```

Listing 5: Funktionsprototyp von EVENT__Send.

Der Parameter `uiEventIn` der Funktion `EVENT__Receive` in Listing 6 stellt ebenfalls ein Bitmuster dar. Ein gesetztes Bit zeigt hier ein Event an, das Teil des Event Sets ist, auf das der Task wartet. Der Parameter `uiOptionSet` bietet zwei Varianten an. Hier kann angegeben werden, ob ein Aufruf dieser Funktion eine Blockierung des Tasks zur Folge hat, oder nicht. In `*puiEventOut` wird dem Aufrufer mitgeteilt welche Events eingetroffen sind. Jedes gesetzte Bit in `*puiEventOut` kennzeichnet somit ein eingetroffenes Event.

In Abbildung 12 ist der Fall angegeben, dass `EVENT__Receive` noch auf Events wartet. Wird in diesem Moment diese Ausführung durch `EVENT__Send` unterbrochen, kann es dazu kommen, dass ein Event zum bestehenden Event Set eines bestimmten Tasks hinzukommt.

Dieses Data Race konnte aber als unkritisch eingestuft werden, da dieses neu hinzugekommene Event durch den lesenden Zugriff nicht verloren geht und beim nächsten Aufruf von `EVENT__Receive` abgearbeitet werden kann.

Die Ergebnisse im Thread Analyzer zeigten, dass der Lockset Algorithmus Eraser im Vergleich zum Thread Checker, der Happens-Before verwendet, eine gründlichere Analyse der Software bezüglich möglicher Data Races liefert. Allerdings zeigte dieses Ergebnis auch, dass ein Einsatz von Happens-Before den Arbeitsaufwand bezüglich der Bewertung der potentiellen Data Races verringern würde, da diese unkritischen Data Races in diesem Fall – korrekterweise – gar nicht angezeigt werden würden.

```
/* _____ P u r p o s e _____
 * This directive attemtps to receive one or all the events specified in
 * uiEventIn. If uiEventIn equals EVENT__GET_PENDING_EVENTS, then the
 * currently pending events are returned in *uiEventOut. In all other
 * cases the function attempts to seize an event sent with
 * EVENT__Send(). The RTEMS_WAIT and RTEMS_NO_WAIT options in
 * uiOptionSet parameter are used to specify whether or not the task is
 * willing to wait for the event condition to be satisfied. The event
 * condition is specified by uiEventIn and the options RTEMS_EVENT_ANY
 * and RTEMS_EVENT_ALL, which specify if a singly event or the complete
 * event set is necessary to satisfy the event condition.
 *
 * _____ I n p u t _____
 * uiEventIn   - A bit set of events; a set bit indicates an event
 *               being part of the event set wanted; if zero
 *               (EVENT__GET_PENDING_EVENTS), the pending events are
 *               returned in *puiEventOut.
 * uiOptionSet - The two options RTEMS_WAIT or RTEMS_NO_WAIT are
 *               accepted; that means that the call to this function
 *               shall be blocking or non-blocking. Note, that the non-
 *               preemtive implementation, ARTOS/n, does not even block
 *               when RTEMS_WAIT was selected. In this case a call, that
 *               normaly would block, leads to the return of the special
 *               status code RTEMS_COOPERATIVE_RETURN. Other options
 *               that can be bitwise ORed to one of the two options
 *               described above are either RTEMS_EVENT_ANY or
 *               RTEMS_EVENT_ALL, specifying if the function shall wait
 *               for any of the events or for all of the events to
 *               occur.
 *               There is no check if the user of the function
 *               selects something senseless like RTEMS_EVENT_ANY |
 *               RTEMS_EVENT_ALL, because the definitions of RTEMS
 *               4.6.2, which disallow such a check, are taken over to
 *               ARTOS.
 *
 * _____ O u t p u t _____
 * *puiEventOut - This parameter holds the events of uiEventIn that were
 *                satisfied. If more than one event was sent to this
 *                task and RTEMS_EVENT_ANY  was chosen, then this
 *                function call clears all events that were sent AND
 *                selected by uiEventIn.
 *
 * _____ R e t u r n _____
 * RTEMS_SUCCESSFUL          - event(s) received as specified
 * RTEMS_UNSATISFIED         - the option RTEMS_NO_WAIT was selected and
 *                             the event condition was not satisfied,
 *                             i.e. "the event(s) did not happen"
 * RTEMS_INVALID_ADDRESS     - uiEventOut is a NULL pointer
 * RTEMS_COOPERATIVE_RETURN  - RTEMS_WAIT was selected and the event
 *                             condition was not satisfied in ARTOS/n.
 */
RTEMS__E_StatusCode EVENT__Receive(
   unsigned32 uiEventIn /*in*/,
   unsigned32 uiOptionSet /*in*/,
   unsigned32 *puiEventOut /*out*/);
```

Listing 6: Funktionsprototyp von `EVENT__Receive`.

6.2 Analyse der Gesamtsoftware mit TAW im Thread Analyzer

Um die Applikationssoftware im Thread Analyzer testen zu können, waren einige Adaptierungen notwendig. Einige Beispiele für diese Änderungen werden in Folge kurz beschrieben. Mit Hilfe des Referenzmodells – beschrieben in Abschnitt 3.2 – erfolgt jetzt die Analyse der Gesamtsoftware.

6.2.1 Adaption des Task Handling

Da auch bei der Analyse der Applikationssoftware der Testansatz TAW zur Anwendung kam, musste die Funktion `TASK__StartMultiTasking` ebenfalls geringfügig abgeändert werden. Das Referenzmodell weist im Umgang mit ARTOS nämlich eine Besonderheit auf: Wie im Abschnitt 3.2 beschrieben, läuft das Referenzmodell nach der Bearbeitung der AGGA-Daten durch die Applikation und bereitet neue AGGA-Daten auf. D.h. nachdem das Multitasking bereits gestartet wurde, muss das Referenzmodell noch weiter Aufgaben erledigen. Die Programmausführung muss also aus der Funktion `TASK__StartMultiTasking` zurückkommen, damit das Referenzmodell die Möglichkeit bekommt, die Schleife – wie in Abbildung 5 in Abschnitt 3.2 zu sehen – abzuarbeiten. Listing 7 zeigt den angesprochenen Abschnitt nach der Modifikation zum Zwecke des Tests mit dem Referenzmodell.

```
void TASK__StartMultiTasking(void)
{
   ...
   // create the thread with the adjusted parameters
   iPThreads[uiPThreads] = pthread_create(&tids[uiPThreads], &attr,
                           ThreadStarter,
                           (void*) &ptMyThreadExecuting->uiPriority);

   ...

   /* after all threads are created, the IdleThread is created */

   ...

/* The following section shall not be executed, because the ARTOS-Tasks
 * must be running, so that data acquired by the ISRs can be handled
 *
 * for (i = 0; i < ARTOS__MAX_NOF_TASKS-1; i++)
 * {
 *    /* pthread_join waits for termination of the undetached sibling
 *     * thread designated by thread and retrieves the exit status of
 *     * the terminated thread *
 *    pthread_join(tids[i], NULL);
 * }
 */

   pthread_attr_destroy(&attr);    // release resources

} /* TASK__StartMultiTasking() */
```

Listing 7: Mittels `pthread_join` wird in `TASK__StartMultiTasking` auf die Terminierung der Threads gewartet. Hier ist dieser Abschnitt im Code auskommentiert.

Damit die Abarbeitung der Testdaten weiter fortgeführt werden kann, darf das Warten der Threads mittels `pthread_join` hier nicht stattfinden.

6.2.2 Probleme mit im Betriebssystem verborgenen Critical Sections

Critical Sections werden in der Original-Version der Applikationssoftware, durch das De- und Reaktivieren von Interrupts geschützt. Diese Funktionalität bietet der Aufruf der ARTOS-Routinen `INTERRUPT__Enable` und `INTERRUPT__Disable` ebenfalls, allerdings werden diese Funktionen von der Applikationssoftware nicht verwendet. Daher kam es bei den ersten Testläufen zu vielen False Positives, weil es Funktionen gab, die ihre Critical Sections nicht im Host-Betriebssystem geschützt hatten. Die von der Applikationssoftware verwendeten Routinen, `SYSCALL__EnableIRQ` und `SYSCALL__DisableIRQ`, mussten daher genauso, wie die Interrupt-Funktionen von ARTOS, durch Stubs ersetzt werden. Ein einfacher Aufruf von `INTERRUPT__Enable` im sonst leeren Rumpf von `SYSCALL__EnableIRQ` und analog, ein `INTERRUPT__Disable` im Rumpf von `SYSCALL__DisableIRQ`, lösten das Problem.

6.2.3 Ergebnisse

Für die Durchführung der Analyse im Thread Analyzer wurden ausschließlich nur jene Module instrumentiert, die zu ARTOS und der Applikationssoftware gehören. Damit waren Fehlermeldungen für das Referenzmodells ausgeschlossen.

Die Untersuchung aller instrumentierten Module im Thread Analyzer führte zum Ergebnis von 152 möglichen Data Races. Diese hohe Anzahl war auf die Missachtung der Locking Discipline beim Zugriff auf Puffer von Partitionen zurückzuführen, wenn der Zugriff durch den Transport eines Zeigers auf die Partition in einer Message synchronisiert wird. Das wird vom Algorithmus Eraser gemeldet und deshalb wurden allein dafür 125 Race Conditions angezeigt.

Diese Methode, dass nicht Daten, sondern Pointer auf Daten verschickt werden, kann vom Thread Analyzer nicht als Schutz für Critical Sections bzw. Schutz vor möglichen Data Races erkannt werden. In Wahrheit wird aber hier ein Data Race verhindert. Die Message Queue sorgt dafür, dass die Pointer-Information in geregelter Reihenfolge an ihr Ziel weitergeleitet wird.

Ein zu Recht angezeigtes Data Race illustriert Abbildung 13. Diese Race Conditions traten in der Datei WCET.c auf, ein Modul zur Berechnung der Worst Case Execution Time (WCET). Hier werden Berechnungen zur WCET für alle Tasks und ISRs durchgeführt, d. h. es werden alle verbrauchten Clock Ticks während der Ausführung eines Tasks gezählt.

Die Funktion `WCET__Start` startet den Timer zur Berechnung der WCET, `WCET__Stop` beendet diesen. `WCET__NewPeriod` speichert die berechnete Zeit und setzt den Timer zurück. Die Funktionen zum Starten und Beenden des Timers laufen in einer Critical Section, d. h. die von Eraser geforderte Locking Discipline ist daher erfüllt. Allerdings wird dieser Schutz bei `WCET__NewPeriod` verabsäumt. Somit ergeben sich in diesem Zusammenhang insgesamt 5 Race Conditions. Denn die mögliche gleichzeitige Ausführung von `WCET__NewPeriod` aus einer der beiden Funktionen (`WCET__Start, WCET__Stop`) kann dazu führen, dass die Anzahl der gezählten Clock Ticks von `WCET__NewPeriod` zurückgesetzt wird und dadurch der Zählerstand falsch ist.

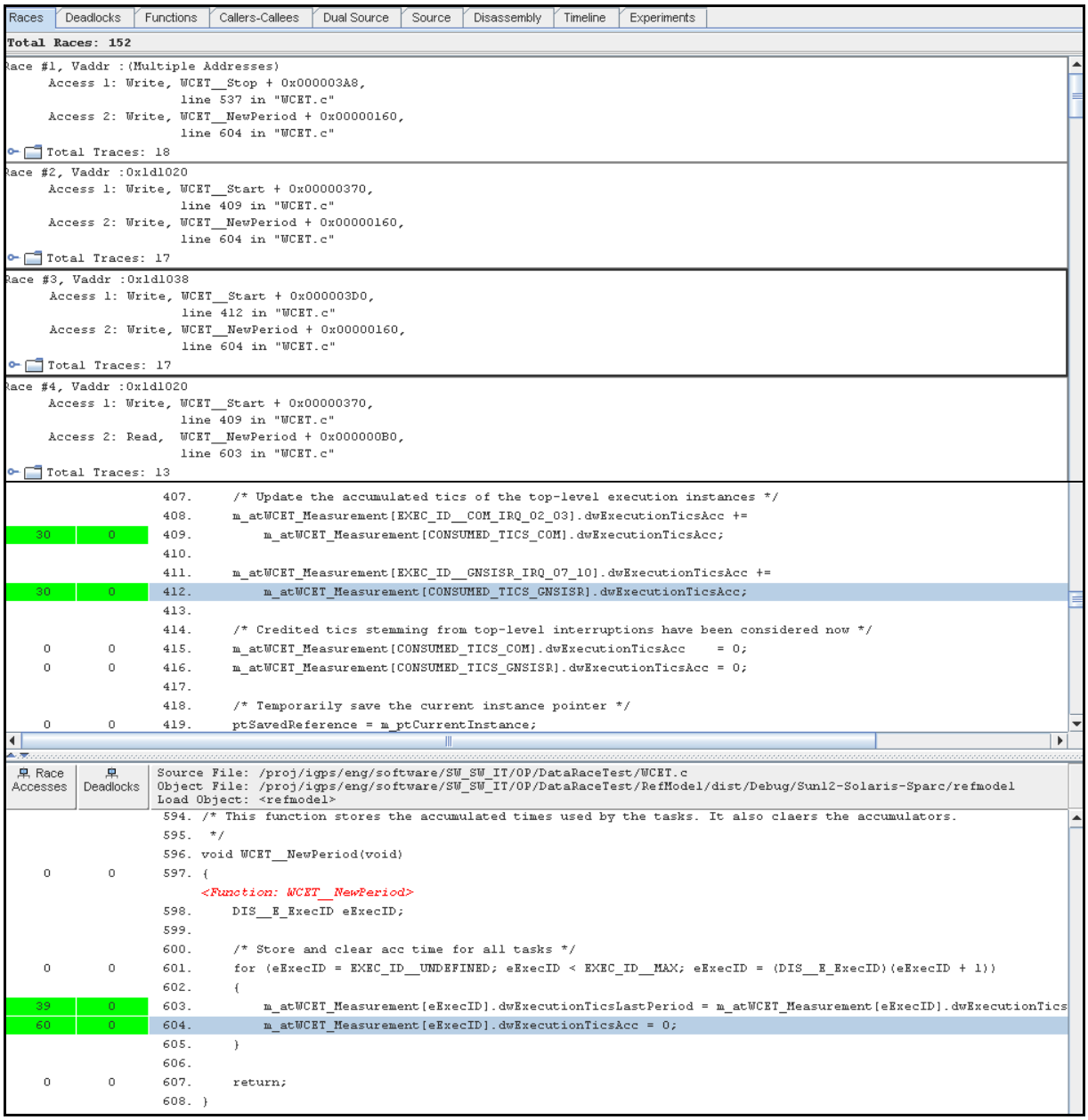

Abbildung 13: Data Races im Zusammenhang der Berechnung der Worst Case Execution Time (WCET). Anzeige der Funktionsaufrufe und des Bereichs des Source Codes beider Zugriffe.

Dieses Data Race hatte keine negativen Auswirkungen auf den operativen Code, da hier keine GPS-Daten verarbeitet werden. Das Modul WCET hat die Funktion eines integrierten Software-Tests. Die Testergebnisse, die das Modul WCET liefert, können aber durch ein Data Race verfälscht und somit ungültig sein.

Ein weiteres korrekt angezeigtes Data Race fand der Thread Analyzer im Modul SVSDB (Space Vehicle Status Database). Die beteiligte Funktion `SVSDB__PutParameter` transferiert eine spezifische Anzahl von Datensätzen von einem Parameter-Paket in eine Datenbank, während die Funktion `SVSDB__CheckRecordDeadline` überprüft, ob der Zeitstempel des Parameter-Pakets noch in einem gültigen Bereich liegt, oder schon eine bestimmte Deadline überschritten ist. In Abbildung 14 sind die Funktionsaufrufe und die betroffenen Zeilen im Code angegeben.

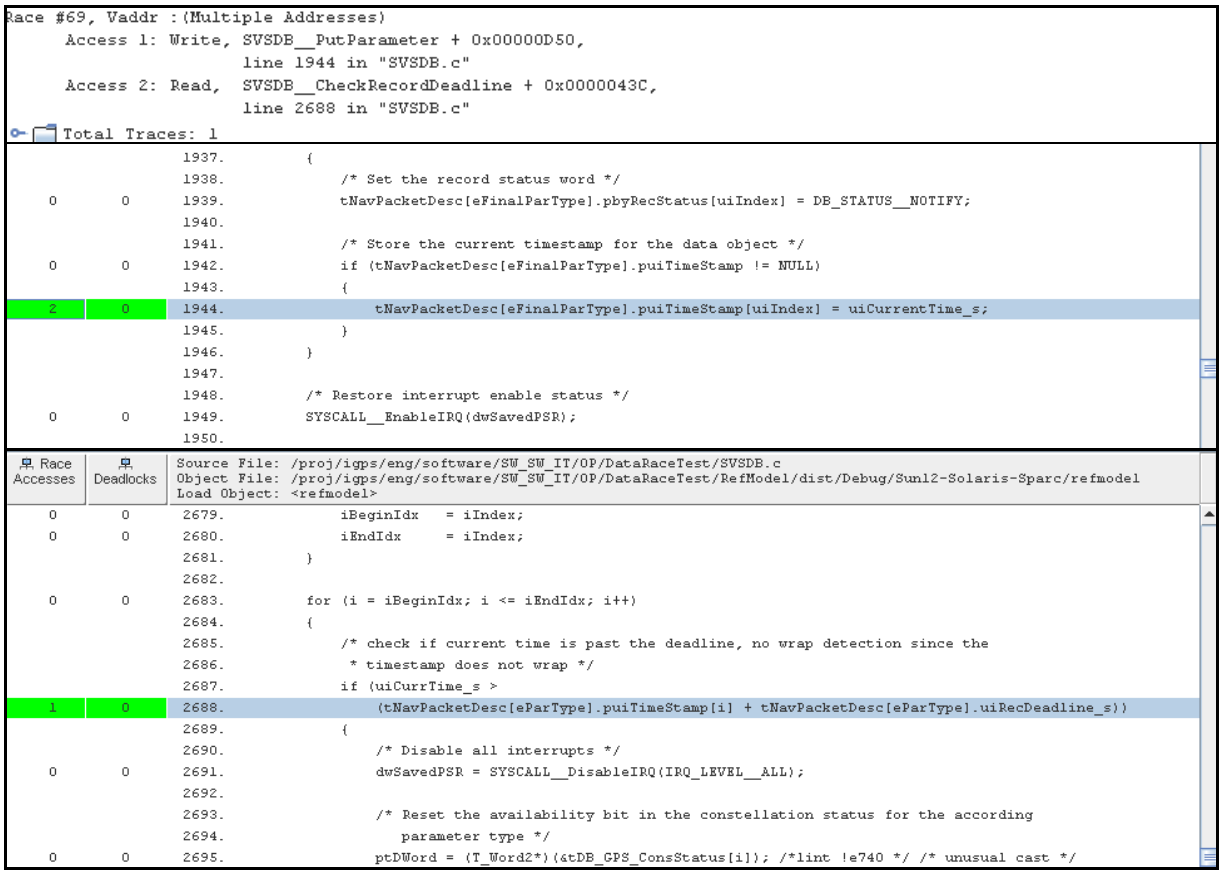

Abbildung 14: Data Race im Modul SVSDB durch ein Read-Before-Lock, also der Zugriff auf eine Variable unmittelbar vor dem Eintritt in eine Critical Section.

Das mögliche Data Race war hier, dass das Lesen des Zeitstempels eines Daten-Pakets und das Beschreiben desselbigen nicht synchronisiert sind. Das kann aber aufgrund äußerer Umstände nicht eintreffen.

Alle anderen gefundenen Data Races waren schon zuvor bekannt und wurden bewusst in Kauf genommen, d. h. im Code wird in Kommentaren explizit auf deren Ungefährlichkeit hingewiesen.

6.3 Diskussion der Ergebnisse

Die Überprüfung des Gesamtmodells im Thread Analyzer zeigte sehr gut die Wirkungsweise der von Eraser geforderten Locking Discipline: Sobald eine Critical Section nicht im Sinne von Eraser geschützt ist, wird eine Warnung ausgegeben. Die Missachtung dieser Vorgabe von Eraser, beim Zugriff auf Partitionen, erklärt die hohe Anzahl an Warnungen bei der Analyse der Gesamtsoftware.

Der größte Teil der Warnungen waren aber Falschmeldungen, weil Zugriffe auf die Puffer der Partitionen über Message Queues synchronisiert werden, die von Eraser nicht erkannt werden. Es ist zu erwarten, dass ein Test mit dem Intel Thread Checker hier nicht gewarnt hätte, weil die Zugriffe immer über die Lock/Unlock-Routinen in den Critical Sections von `MSQ__Send` und `MSQ__Receive` gemäß Happens-Before geordnet sind.

Sechs der Warnungen zeigten auf 2 potentielle Data Races, wovon eines durch äußere Umstände nicht zum Problemfall werden kann, das zweite aber ein Programmierfehler war, der behoben wurde.

7 Zusammenfassung und Ausblick

Die Ergebnisse der Tests im Intel Thread Checker illustrierten, dass einige Warnungen kritisch hinterfragt werden müssen, da diese nicht den tatsächlich betroffenen Bereich im Programm anzeigten. Außerdem zeigten sie, dass ein gutes Verständnis für den zu untersuchenden Source-Code notwendig ist, um die Meldungen der potentiellen Data Races richtig bewerten zu können. Die unterschiedlichen Ergebnisse der Analyse von TAW und DuSCa mit dem Intel Thread Checker werden auf das jeweilige Scheduling des zuständigen Betriebssystems zurückgeführt.

Die in TAW für die Analyse im Thread Checker verwendete Windows-Thread-API wurde durch die POSIX-Thread-API abgelöst. Die Funktionen ließen sich selbstverständlich nicht 1:1 vom einen in das andere Betriebssystem übernehmen. Doch nur so war es möglich, den abschließenden Test der GPS-Applikationssoftware inklusive ARTOS durchzuführen, da die dafür notwendige Simulationsumgebung ausschließlich unter Solaris läuft. Der Thread Analyzer von Sun stellte in dieser Umgebung das Tool für die Suche nach Data Races dar.

Der Vergleich mit den Ergebnissen, die der Intel Thread Checker beim Test der beiden Ansätze lieferte, zeigt die Abhängigkeit vom Scheduler bei Happens-Before. Denn Eraser erkannte beide Data Races im Zusammenhang mit `MSQ__Send` und `MSQ__Urgent` problemlos. Andererseits bewiesen die Ergebnisse im Thread Analyzer auch, dass Eraser tendenziell zu viele Warnungen ausgibt.

Tabelle 3 liefert eine Auflistung aller Arten der gefundenen Data Races beim Test der Applikationssoftware.

Tabelle 3: Ergebnis der Analyse der Gesamtsoftware mit TAW im Thread Analyzer.

Potentielle Data Races	**Anzahl der Warnungen**
Gemeldete Data Races im Zusammenhang mit der Synchronisation durch Message Queues; allesamt vernachlässigbar.	125
Zu vernachlässigendes Data Race aufgrund von Hinweisen im Code.	16
Ungeschütztes Lesen einer Variable, die in einer Critical Section verändert wird; als unkritisch bewertet.	3
Zugriff auf Teile des Codes des Referenzmodells; unkritisch da Referenzmodell nicht Teil der Untersuchung.	2
Korrekt erkanntes Data Race im Modul SVSDB.	1
Ebenfalls echtes Data Race im Modul WCET.	5
Gesamtanzahl der Warnungen	**152**

Bei der Bewertung der Ergebnisse bieten sowohl der Thread Analyzer, als auch der Thread Checker hohen Bedienungskomfort. Die Bewertung wurde vor allem durch die von beiden Werkzeugen

angebotene Dual Source-Ansicht unterstützt. Als weitere nützliche Bewertungshilfe erwies sich die Funktion zur Darstellung der Hierarchie der Funktionsaufrufe (Trace), die in Folge zu einem Data Race führen. Diese ermöglicht eine schnelle und genaue Einteilung in False Positives oder tatsächliche Race Conditions.

Werkzeuge zum Auffinden und zur Diagnose von Data Races gewinnen sicherlich in Zukunft an Bedeutung. Der vermehrte Einsatz von Mehrkern-CPUs gibt multithreaded Programmen Laufzeitvorteile gegenüber sequenziellen Programmen. Daher werden auch Data Race Tests vermutlich immer wichtiger.

Voraussetzung bei der Anwendung dieser Werkzeuge ist, dass diese, die vom zu untersuchenden Programm verwendete Thread-API, unterstützen. Erst die aufwändigen Ansätze TAW und DuSCa, ließen den Test einer Embedded-Anwendung (ARTOS), die eine solche API nicht verwendet, zu. Auch der Portierungsaufwand von der Windows-Thread-API zur POSIX-Thread-API war erheblich, obwohl nur Locks als Synchronisationsmechanismus verwendet wurden. Für die Implementierung der Testansätze war ein Personenmonat nötig, ebenso für die Portierung.

Abkürzungsverzeichnis

AGGA	Advanced GPS and GLONASS ASIC
API	Application Programming Interface
ARM	Acorn Risc Machine
ARTOS	Austrian Real-Time Operating System
ASIC	Application Specific Integrated Circuit
CPU	Central Processing Unit
DARTS	Design Approach for Real-Time Systems
DuSCa	Duplikation der System-Calls
GLONASS	Global Navigation Satellite System
GPS	Global Positioning System
IDE	Integrated Development Environment
IRQ	Interrupt Request
ISR	Interrupt Service Routine
MIPS	Microprocessor without interlocked pipeline stages
POSIX	Portable Operating System Interface
RAA	RUAG Aerospace Austria GmbH
RTC	Run-To-Completion
RTEMS	Real-Time Executive for Multiprocessor Systems
RTOS	Real-Time Operating System
TAW	Thread Application Wrapper
WCET	Worst Case Execution Time

Glossar

Prozess

Im Rahmen dieser Diplomarbeit steht der Begriff Prozess für einen Ablauf, d. h. für die Bedeutung des Begriffs Prozess im ganz allgemeinen Sinn, nicht für eine informationstechnische Interpretierung.

Task

Ein Task stellt die Ausführung eines sequentiellen Programms dar, für Aufgaben, die unabhängig voneinander abgearbeitet werden. Der Begriff Task wird oft als Synonym für Prozess verwendet, andererseits auch als Synonym für Thread.

Hier in dieser Arbeit ist ein Task immer einem Thread gleichzusetzen.

Thread

Threads können parallel laufen und verwenden eine gemeinsame Ressource. Die wichtigsten Zustände die ein Thread durchlaufen kann, sind Running, Ready und Blocked.

In dieser Diplomarbeit steht Thread bzw. Task für einen einzelnen, unabhängigen Ausführungsstrang, der parallel zu anderen Threads laufen kann und sich mit diesen einen Adressraum teilt.

Anhang A

Wrapper-Modul task-wrapper.c

Listing A1 zeigt alle Funktionen und Code-Abschnitte aus dem Wrapper-Modul task-wrapper.c zur Implementierung des Ansatzes TAW mit Hilfe der Windows-Thread-API.

Die 3 Tabellen zu Beginn von Listing A1 ermöglichen, dass ARTOS-Tasks durch Windows-Threads ersetzt werden können. In `TASK__StartMultiTasking` wird das Multitasking von ARTOS modelliert. Hier werden die Windows-Threads gestartet. Mit Hilfe der Funktion `TASK__GetExecutingTask`, wird der gerade ausführende Thread bestimmt. Dies geschieht, indem in einer Schleife die IDs der laufenden ARTOS-Tasks mit jenen der Windows-Threads verglichen werden. Bei einer Übereinstimmung, ist so der gesuchte Thread bestimmt. In `TASK__GetPriorityOfCallingTask` kommt ebenfalls diese Schleife zur Anwendung, um hier die Priorität eines bestimmten Threads auslesen zu können. Das Wrapping der Funktion `TASK__UnBlock` (macht einen Task lauffähig) geschieht in gleicher Weise, mit Hilfe der zu Beginn angesprochenen Tabellen. Dasselbe Prinzip wird auch in `TASK__GetIdOfExecutingTask` angewandt.

Um einen ausgewählten Thread zu blockieren, wird in `TASK__BlockExecutingTask` die Funktion `INTERRUPT__BlockThreadWhenLeavingCriticalSection` aufgerufen. Diese ermöglicht, dass ein bestimmter Thread, nachdem dieser eine Critical Section verlassen hat, blockiert werden kann.

```
/* WIN API WRAP BEGIN */
/* these three tables  allow to match a Windows thread to
 * an ARTOS task */
static unsigned uiWinThreads = 0;
HANDLE ahThreads[ARTOS__MAX_NOF_TASKS+1];
DWORD  adwThreadIds[ARTOS__MAX_NOF_TASKS];
unsigned auiArtosTaskId[ARTOS__MAX_NOF_TASKS];
/* WIN API WRAP END */



/* WIN API WRAP BEGIN */
void TASK__StartMultiTasking(void)
{
   TASK__T_TaskControlBlock* ptMyThreadExecuting;
   DWORD dwThreadId;
   FATAL__ASSERT(false == bMultiTaskingStarted, ARTOS_MODULE_ID__TASK);
   bMultiTaskingStarted = true;

   for (;;)
   {
      /* Pointer to the TCB of the task that could be running next */
      TASK__T_TaskControlBlock* ptAnyThread;

      /* Default: No new task is ready to run; either a deadlock
       * happened or we need to wait for an interrupt service routine to
       * execute. Run the system idle task. */
      ptMyThreadExecuting = &tTcbOfSystemIdleTask;
```

```
      /* Loop through the TCBs to see if the default state can be
       * overwritten. Loop taken from TASK__Dispatch() in ARTOS. */
      for (ptAnyThread = &atTcbPool[0];
         (size_t) ptAnyThread < sizeof(atTcbPool) + (size_t) atTcbPool;
         ptAnyThread++)
      {
         if (TASK_STATE__READY == ptAnyThread->eCurrentState)
         {
            /* we found the new task that we will run */
            ptMyThreadExecuting = ptAnyThread;
            /* leave for loop of former TASK__Dispatch() */
            break;
         }
      } /* for ptAnyThread */

      if (ptMyThreadExecuting == &tTcbOfSystemIdleTask)
      {
         /* selected the idle task */
         break;
      }

      /* need to cancel READY for next call to TASK__Dispatch() */
      ptMyThreadExecuting->eCurrentState = TASK_STATE__RUNNING;

      ahThreads[uiWinThreads] = CreateThread(0, 0, ThreadStarter,
                                &ptMyThreadExecuting->uiPriority, 0,
                                &dwThreadId);
      if (ahThreads[uiWinThreads] == 0)
      {
         printf("Cannot start thread!\n");
         exit(1);
      }

      adwThreadIds[uiWinThreads] = dwThreadId;
      auiArtosTaskId[uiWinThreads] = ptMyThreadExecuting->uiObjectId;
      uiWinThreads++;
   }
   ahThreads[uiWinThreads] = CreateThread(0, 0, IdleThread, 0, 0, 0);
   SetThreadPriority(ahThreads[uiWinThreads], THREAD_PRIORITY_IDLE);

   /* wait until threads complete */
   WaitForMultipleObjects(uiWinThreads, ahThreads, TRUE, INFINITE);

} /* TASK__StartMultiTasking() */
```

```
TASK__T_TaskControlBlock* TASK__GetExecutingTask(void)
{
   unsigned n, uiIndex;
   DWORD dwWinId;

   dwWinId = GetCurrentThreadId();
   for (n = 0; n < ARTOS__MAX_NOF_TASKS; n++)
   {
      /* obtain the index of this thread within auiArtosTaskId[] */
      if (adwThreadIds[n] == dwWinId) break;
   }
   FATAL__ASSERT(n < ARTOS__MAX_NOF_TASKS, ARTOS_MODULE_ID__TASK);
   uiIndex = auiArtosTaskId[n] & OBJECT__INDEX_MASK;

   Printf("TASK__GetExecutingTask(): thread %x, idx: %d\n",
          dwWinId, uiIndex);
   return &atTcbPool[uiIndex];
}
/* WIN API WRAP END */



unsigned32 TASK__GetPriorityOfCallingTask(void)
{
   register unsigned32 uiPriority;

   if (bMultiTaskingStarted == false)
   {
      uiPriority = 0; /* no task is scheduled */
      printf("WARNING: TASK__GetPriorityOfCallingTask() called before
              multitasking started!\n");
   }
   else
   {
      /* WIN API WRAP BEGIN */
      DWORD dwTheadId;
      unsigned n, uiIndex;
      dwTheadId = GetCurrentThreadId();
      for (n = 0; n < ARTOS__MAX_NOF_TASKS; n++)
      {
         /* obtain the index of this thread within auiArtosTaskId[] */
         if (adwThreadIds[n] == dwTheadId) break;
      }

      FATAL__ASSERT(n < ARTOS__MAX_NOF_TASKS, ARTOS_MODULE_ID__TASK);
      uiIndex = auiArtosTaskId[n] & OBJECT__INDEX_MASK;
      uiPriority = atTcbPool[uiIndex].uiPriority;
      /* WIN API WRAP END */
   }

    return uiPriority;
}
```

```
void TASK__UnBlock(TASK__T_TaskControlBlock* ptThisTask)
{
   /* This function is called for tasks in blocked state, only.
    * The transition BLOCKED->RUNNING is allowed. */
   /* modified for the WIN API wrapping */
   tThisTask->eCurrentState = TASK_STATE__RUNNING;
   FATAL__ASSERT(ptThisTask->uiMagicNumber == TCB_MAGIC_NUMBER,
                 ARTOS_MODULE_ID__TASK);

   /* WIN API WRAP BEGIN */
   {
      DWORD dwErrorCode;
      unsigned n;
      for (n = 0; n < ARTOS__MAX_NOF_TASKS; n++)
      {
         /* find out which adwThreadIds[n] holds the corresponding
          * Windows thread id */
          if (auiArtosTaskId[n] == ptThisTask->uiObjectId) break;
      }
      FATAL__ASSERT(n < ARTOS__MAX_NOF_TASKS, ARTOS_MODULE_ID__TASK);
      Printf("Resuming thread %x\n", adwThreadIds[n]);

      dwErrorCode = ResumeThread(ahThreads[n]);
      FATAL__ASSERT(dwErrorCode != -1, ARTOS_MODULE_ID__TASK);
   }
   /* WIN API WRAP END */

   return;
}



unsigned32 TASK__GetIdOfExecutingTask(void)
{
   /* WIN API WRAP BEGIN */
   unsigned n;
   DWORD dwWinId;

   dwWinId = GetCurrentThreadId();
   for (n = 0; n < ARTOS__MAX_NOF_TASKS; n++)
   {
      /* obtain the index of this thread within auiArtosTaskId[] */
      if (adwThreadIds[n] == dwWinId) break;
   }
   FATAL__ASSERT(n < ARTOS__MAX_NOF_TASKS, ARTOS_MODULE_ID__TASK);

   if (auiArtosTaskId[n] == 0)
   {
      // something runs really wrong ...
      printf("GetCurrentThreadId(): %x\n", GetCurrentThreadId());
      for (n = 0; n < ARTOS__MAX_NOF_TASKS; n++)
      {
         printf("Win: %x  Artos: %x\n" ,
                 adwThreadIds[n], auiArtosTaskId[n]);
      }
   }
   return auiArtosTaskId[n];
   /* WIN API WRAP END */
}
```

```
RTEMS__E_StatusCode TASK__BlockExecutingTask(void)
{
   if (bMultiTaskingStarted)
   {
      /* WIN API WRAP BEGIN */
      INTERRUPT__BlockThreadWhenLeavingCriticalSection();
     /* WIN API WRAP END */
   }
   else
   {
      printf("WARNING: TASK__BlockExecutingTask() called before
             multitasking started!\n");
   }

   /* this return value is specific to the non-preemptive RTOS */
   return RTEMS_COOPERATIVE_RETURN;
}
```

Listing A1: Funktionen des Wrapper-Moduls task-wrapper.c, die die Windows-Thread-API verwenden, zur Implementierung des Testansatzes TAW.

Anhang B

Interrupt-Initialise, -Disable und -Enable (Realisierung mit Mutexes)

Listing B1 zeigt die Funktion `INTERRUPT__Initialise` in der POSIX-Variante, Listing B2 dieselbe Funktion unter Verwendung der Windows-Thread-API. Listings B3 und B4 illustrieren die Funktion `INTERRUPT__Disable` in der Windows- und POSIX-Variante, die Listings B5 und B6 zeigen dies für die Funktion `INTERRUPT__Enable`. Listing B7 stellt ebenfalls diese Funktion dar, allerdings für den Ansatz DuSCa.

```
void INTERRUPT__Initialise(void)
{
   uiLockNestingLevel = 0;
   pthread_mutexattr_init(&mutexAttr);

   // Set the mutex as a normal "fast" mutex
   iArtosMutex = pthread_mutexattr_settype(&mutexAttr,
                                           PTHREAD_MUTEX_NORMAL);
   if (iArtosMutex != 0)
   {
      printf("pthread_mutexattr_settype failed at
              INTERRUPT__Initialise!\n");
   }
   // Create the mutex with the attributes set
   iArtosMutex = pthread_mutex_init(&mutex, &mutexAttr);
   if (iArtosMutex != 0)
   {
      printf("pthread_mutex_init failed at INTERRUPT__Initialise!\n");
   }
   // destroy the attribute
   pthread_mutexattr_destroy(&mutexAttr);

   if (iArtosMutex != 0)
   {
      printf("pthread_mutex_init() failed! Error Condition: %d\n",
              iArtosMutex);
   }
   /* create a thread-specific data key visible to all threads in the
    * process */
   gdwTlsIndex = pthread_key_create(&key_1, NULL);

   if (gdwTlsIndex != 0)
   {
      printf("SCRIPT ERROR: pthread_key_create() failed\n");
      exit(1);
   }

   /* pass the key index to gdwTlsIndex:
    * gdwTlsIndex is further used by INTERRUPT__Enable and
    * INTERRUPT__BlockThreadWhenLeavingCriticalSection to get the key
    * index of the TLS (Thread Local Storage) used by running thread
    */
   gdwTlsIndex = key_1;
}
```

Listing B1: Verwendung der POSIX-Thread-API in der Funktion INTERRUPT__Initialise.

```
void INTERRUPT__Initialise(void)
{
   uiLockNestingLevel = 0;
   hArtosMutex = CreateMutex(0, FALSE, 0);

   if ((hArtosMutex == 0))
   {
      printf("Mutex creation failed\n");
      exit(0);
   }

   /* create TLS - Thread Local Storage,
    * see http://support.microsoft.com/kb/163449 */
   gdwTlsIndex = TlsAlloc();
   if (gdwTlsIndex == TLS_OUT_OF_INDEXES)
   {
      printf("SCRIPT ERROR: TlsAlloc() failed\n");
      exit(1);
   }
}
```

Listing B2: Die Funktion INTERRUPT__Initialise unter Verwendung der Windows-Thread-API.

```
unsigned32 INTERRUPT__Disable(void)
{
   /* Note: WaitForSingleObject() can be called consecutively several
    * times by the same thread without blocking, i.e. if the thread
    * running has already called INTERRUPT__Disable() before, it won't
    * block when it calls it a second time
    */
   if (WaitForSingleObject(hArtosMutex, 5000L) != 0)
   {
      printf("ARTOS Mutex waiting failed\n");
      exit(1);
   }

   if (uiLockNestingLevel == 0)
   {
      uiTaskHoldingLock = TASK__GetIdOfExecutingTask();
      uiThreadHoldingLock = GetCurrentThreadId();
      Printf("Task obtaining lock: %x == Thread %x\n",
             uiTaskHoldingLock, uiThreadHoldingLock);
   }
   else
   {
      Printf("Thread %x increased lock nesting level to %d\n",
             GetCurrentThreadId(), uiLockNestingLevel + 1);
   }
   uiLockNestingLevel++;

   return 0;
}
```

Listing B3: Verwendung des Synchronisationsobjekts hArtosMutex, um das Verhalten von INTERRUPT__Disable in den Testansätzen TAW und DuSCa zu imitieren (Windows-Thread-API).

```
unsigned32 INTERRUPT__Disable(void)
{
   int rc;

   rc = pthread_mutex_lock(&mutex);
   if (rc != 0)
   {
      printf("ARTOS Mutex waiting failed! Error Condition: %d\n", rc);
      exit(1);
   }

   if (uiLockNestingLevel == 0)
   {
      uiTaskHoldingLock = TASK__GetIdOfExecutingTask();
      uiThreadHoldingLock = pthread_self();
      printf("Task obtaining lock: %x == Thread %x\n",
             uiTaskHoldingLock, uiThreadHoldingLock);
   }
   else
   {
      printf("Thread %x increased lock nesting level to %d\n",
             pthread_self(), uiLockNestingLevel + 1);
   }
   uiLockNestingLevel++;

   return 0;
} // INTERRUPT__Disable()
```

Listing B4: INTERRUPT__Disable im Testansatz TAW (POSIX-Thread-API).

```
void INTERRUPT__Enable(unsigned32 uiLevel)
{
   int *piBuffer;
   unsigned uiLocalLockNestingLevel;

   piBuffer = TlsGetValue(gdwTlsIndex);
   if (NULL == piBuffer)
   {
      printf("SCRIPT ERROR: TlsGetValue() returned NULL\n");
      Printf("Task releasing lock: %x == Thread %x\n",
             TASK__GetIdOfExecutingTask(), GetCurrentThreadId());
      exit(7);
    }

   /* no need to protect this no two threads can enter the critical
    * section at the same time */
   uiLockNestingLevel--;
   uiLocalLockNestingLevel = uiLockNestingLevel;

   if (uiLocalLockNestingLevel == 0)
   {
      Printf("Task releasing lock: %x == Thread %x\n",
             TASK__GetIdOfExecutingTask(), GetCurrentThreadId());
      uiTaskHoldingLock = 0;
      uiThreadHoldingLock = 0;
```

```
      if (*piBuffer)  /* thread should block */
      {
         DWORD dwErrorCode;
         int iActualPriority;

         *piBuffer = 0;  // should not block again
         TlsSetValue(gdwTlsIndex, piBuffer);

         /* the release of the mutex and the suspending of the thread
          * should be atomic since this is a lot of effort to program,
          * we try to get the same effect by increasing the priority of
          * the thread for a short time
          */
         Printf("Suspending Thread %x\n", GetCurrentThreadId());
         iActualPriority = GetThreadPriority(GetCurrentThread());
         SetThreadPriority(GetCurrentThread(),
                           THREAD_PRIORITY_TIME_CRITICAL);
         ReleaseMutex(hArtosMutex);

         dwErrorCode = SuspendThread(GetCurrentThread());
         if (dwErrorCode == -1)
         {
            printf("SuspendThread() failed.\n");
            exit(1);
         }
         SetThreadPriority(GetCurrentThread(), iActualPriority);
      }

      else
      {
         // release the critical section without suspending the thread
         ReleaseMutex(hArtosMutex);
      }
   }

   else
   {
      printf("Thread %x decreased lock nesting level to %d\n",
             GetCurrentThreadId(), uiLockNestingLevel);
   }

   return;
}
```

Listing B5: In INTERRUPT__Enable (im Testansatz TAW unter Verwendung der Windows-Thread-API) erfolgt die Freigabe der Critical Section, die zuvor in INTERRUPT__Disable geschützt wurde.

```
void INTERRUPT__Enable(unsigned32 uiLevel)
{
   pthread_once_t once = PTHREAD_ONCE_INIT;
   int *piBuffer;
   unsigned uiLocalLockNestingLevel;
   int tt, i, ret = 0;
   struct sched_param param_newPrio, param_prevPrio;

   piBuffer = (int*)pthread_getspecific(gdwTlsIndex);
   if (piBuffer == NULL)
   {
      printf("SCRIPT ERROR: pthread_getspecific() returned %x\n",
             piBuffer);
      printf("Task releasing lock: %x == Thread %x\n",
             TASK__GetIdOfExecutingTask(), pthread_self());
      exit(7);
    }

   /* no need to protect this no two threads can enter the critical
    * section at the same time */
   uiLockNestingLevel--;
   uiLocalLockNestingLevel = uiLockNestingLevel;

   if (uiLocalLockNestingLevel == 0)
   {
      printf("Task releasing lock: %x == Thread %x\n",
             TASK__GetIdOfExecutingTask(), pthread_self());
      uiTaskHoldingLock = 0;
      uiThreadHoldingLock = 0;

      if (*piBuffer)  /* thread should block */
      {
         // init the signalhandler for suspending and resuming threads
         ret = pthread_once(&once, suspend_init_routine);
         if (ret != 0)
         {
            printf("Initialization of the signalhandler failed.\n");
            exit(1);
         }

         unsigned dwErrorCode;
         int iActualPriority;

         *piBuffer = 0;  // should not block again
         pthread_setspecific(gdwTlsIndex, piBuffer);

         /* the release of the mutex and the suspending of the thread
          * should be atomic since this is a lot of effort to program,
          * we try to get the same effect by increasing the priority of
          * the thread for a short time
          */

         printf("Suspending Thread %x\n", pthread_self());
         iActualPriority = pthread_getschedparam(pthread_self(),
                           &policy, &param_prevPrio);

         /* all Threads are set to the same max. priority */
         // get the maximum priority level for SCHED_RR (59)
         ret = sched get priority max(policy);
```

```
         // set the current thread to the maximum priority level
         param_newPrio.sched_priority = ret;
         iActualPriority = pthread_setschedparam(pthread_self(),
                           policy, &param_newPrio);
         iArtosMutex = pthread_mutex_unlock(&mutex);

         // send the thread to suspend the userspecific signal
         dwErrorCode = pthread_kill(pthread_self(), SIGUSR1);

         if (dwErrorCode != 0)
         {
            printf("SuspendThread() failed.\n");
            exit(1);
         }

         pthread_setschedparam(pthread_self(), policy, &param_prevPrio);
      }
      else
      {
         // release the critical section without suspending the thread
         iArtosMutex = pthread_mutex_unlock(&mutex);
      }
   }
   else
   {
      printf("Thread %x decreased lock nesting level to %d\n",
             pthread_self(), uiLockNestingLevel);
   }

   return;
} // INTERRUPT__Enable()
```

Listing B6: INTERRUPT__Enable im Testansatz TAW unter Verwendung der POSIX-Thread-API.

```
void INTERRUPT__Enable(unsigned32 uiLevel)
{
   unsigned uiLocalLockNestingLevel;

   /* no need to protect this no two threads can enter the critical
    * section at the same time */
   uiLockNestingLevel--;
   uiLocalLockNestingLevel = uiLockNestingLevel;

   if (uiLocalLockNestingLevel == 0)
   {
      Printf("Task releasing lock: %x == Thread %x\n",
             TASK__GetIdOfExecutingTask(), GetCurrentThreadId());
      uiTaskHoldingLock = 0;
      uiThreadHoldingLock = 0;

      // release the critical section
      ReleaseMutex(hArtosMutex);
   }
   else
   {
      printf("Thread %x decreased lock nesting level to %d\n",
             GetCurrentThreadId(), uiLockNestingLevel);
    }

   return;
}
```

Listing B7: INTERRUPT__Enable im Testansatz DuSCa (Windows-Thread-API).

Anhang C

Gezieltes Suspending und Resuming von Threads

In Listing C1 wird das Modul susp.c gezeigt. Es wird verwendet, um mit Hilfe von userspezifischen Signalen, einen ausgewählten Thread anzuhalten und anschließend wieder fortzusetzen.

```
#include <pthread.h>
#include <signal.h>
#include "errors.h"
#include "artos.h"

// to indicate, that the signal handler has been initialized once
int iSuspInitRoutine = 0;
/*
 * Handle SIGUSR1 in the target thread, to suspend it until
 * receiving SIGUSR2 (resume).
 */

void suspend_signal_handler(int sig)
{
   sigset_t signal_set;

   /*
    * Block all signals except SIGUSR2 while suspended.
    */
   sigfillset(&signal_set);
   sigdelset(&signal_set, SIGUSR2);
   sigsuspend(&signal_set);

   /*
    * Once I'm here, I've been resumed, and the resume signal
    * handler has been run to completion.
    */

   return;
}

/*
 * Handle SIGUSR2 in the target thread, to resume it. Note that
 * the signal handler does nothing. It exists only because we need
 * to cause sigsuspend() to return.
 */
void resume_signal_handler(int sig)
{
   return;
}

/*
 * Dynamically initialize the "suspend package" when first used
 * (called by pthread_once).*/
void suspend_init_routine(void)
{
   int status;
   struct sigaction sigusr1, sigusr2;
```

```
   /* Install the signal handlers for suspend/resume. */
   sigusr1.sa_flags = 0;
   sigusr1.sa_handler = suspend_signal_handler;
   sigemptyset(&sigusr1.sa_mask);
   sigaddset(&sigusr1.sa_mask, SIGUSR2);
   sigusr2.sa_flags = 0;
   sigusr2.sa_handler = resume_signal_handler;
   sigemptyset(&sigusr2.sa_mask);

   status = sigaction(SIGUSR1, &sigusr1, NULL);
   if (status == -1)
   {
      errno_abort("Installing suspend handler");
   }

   status = sigaction(SIGUSR2, &sigusr2, NULL);
   if (status == -1)
   {
      errno_abort("Installing resume handler");
   }
   /* first call initialized the routine, set iSuspInitRoutine
    * to indicate, that no further calls are needed */
   iSuspInitRoutine = 1;

   return;
}
```

Listing C1: susp.c dient im Ansatz TAW in Solaris, um userspezifische Signale an ausgewählte Threads zu senden und diese so entweder zu unterbrechen, oder wieder fortzusetzen.

Anhang D

Verwendete Verzeichnisse im RAA-System

Folgende Verzeichnisse wurden im Rahmen der Data Race Tests verwendet:

ARTOS_DuSCa	Source Files des Testansatzes DuSCa zum Test von ARTOS/n und die Ergebnisse der Analyse im Intel Thread Checker
ARTOS_Eraser	Source Files des portierten Testansatzes TAW zum Test von ARTOS/n in Solaris und die Ergebnisse der Analyse im Thread Analyzer
ARTOS_TAW	Source Files des Testansatzes TAW zum Test von ARTOS/n und die Ergebnisse der Analyse im Intel Thread Checker
DOCS	Papers, Dokumentation zu den verwendeten Werkzeugen, Diplomarbeits-tagebuch
GPSR_Eraser	Source Files des portierten Testansatzes TAW zum Test von ARTOS/n, der adaptierten Files des Referenzmodells, der Applikationssoftware in Solaris und die Ergebnisse der Analyse im Thread Analyzer

Das Arbeitsverzeichnis findet man unter \\uranus\proj\sgps\eng\software\SW_SW_IT\OP\ im RAA-System. Die Datei 00readme.txt im selben Verzeichnis, enthält alle notwendigen Informationen zu den Projekten.

Literatur

[1] Cyrille Artho.
Combining Static and Dynamic Analysis to Find Multi-threading Faults Beyond Data Races. *Diss. ETH No. 1602,* April 2005.

[2] Cormac Flanagan and Stephen N. Freund.
Atomizer: A Dynamic Atomicity Checker For Multithreaded Programs. *Principles of Programming Languages (POPL04),* Venice, Italy, January 14-16, 2004.

[3] 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.

[4] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro and 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.

[5] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran.
Goldilocks: Efficiently Computing the Happens-Before Relation Using Locksets. *First Combined International Workshops, FATES 2006 and RV 2006,* Seattle, WA, USA, August 15-16, 2006, p. 193-207.

[6] Cyrille Artho, Klaus Havelund, and Armin Biere.
High-level Data Races. *The First International Workshop on Verification and Validation of Enterprise Information Systems (VVEIS'03),* Angers, France, April 2003.

[7] Polyvios Pratikakis, Jeffrey S. Foster, Michael Hicks.
LOCKSMITH: Context-Sensitive Correlation Analysis for Race Detection. *Programming Language Design and Implementation (PLDI'06),* Ottawa, Canada, June 2006, S. 320-331.

[8] Mathias Thore.
Automatic detection of race conditions in OSE using vector clocks. *Master's Thesis Information Technology, Uppsala University,* Uppsala, June 2005.
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.88.2847&rep=repl&type=pdf [abgerufen am 21.08.2008]

[9] Eli Pozniansky and Assaf Schuster.
Efficient On-the-Fly Data Race Detection in Multithreaded C++Programs. *Ninth ACM SIGPLAN symposium on Principles and practice of parallel programming,* San Diego, CA, USA, June 11-13, 2003.

[10] Karl Marklund and Lars Albertsson.
Dynamic Race Detection for Operating Systems. *Fifth Conference on Software Engineering Research and Practice,* Sweden, 2005

[11] Yuan Yu, Tom Rodeheffer, and Wei Chen.
RaceTrack: Efficient Detection of Data Race Conditions via Adaptive Tracking. *20th ACM Symposium on Operating Systems Principles 2005, SOSP 2005,* Brighton, UK, October 23-26, 2005.

[12] Wikipedia, the free encyclopedioa: Method stub [Online].
http://en.wikipedia.org/wiki/Method_stub [abgerufen am 07.07.2008]

[13] Wikipedia, the free encyclopedioa: Happened-before [Online].
http://en.wikipedia.org/wiki/Happened-before [abgerufen am 13.08.2008]

[14] Werner Schütz.
The Testability of Distributed Real-Time Systems
Diss.TU Wien, No. 527440 II, Februar 1992.

[15] S. Loveland, G. Miller, R. Prewitt, M. Shannon.
Testing z/OS: The premier operating system for IBM's zSeries server
IBM Systems Journal, Vol 41, No 1, 2002.

[16] European Space Agency, board for software standardisation and control.
ESA Software Engineering Stantards, ESA PSS-05-0 Issue 2, Februar 1991.

[17] Hermann Kopetz.
REAL-TIME SYSTEMS Design Principles for Distributed Embedded Applications.
Kluwer Academic Publishers, ISBN 0-7923-9894-7, 1997.

[18] Sun Studio 12: Thread Analyzer User's Guide - Sun Microsystems [Online].
http://docs.sun.com/app/docs/doc/820-0619 [abgerufen am 08.09.2008]

[19] Utpal Banerjee, Brian Bliss, Zhiqiang Ma, Paul Petersen.
A Theory of Data Race Detection. *Parallel and Distributed Systems: Testing and Debuggging. PADTAD-IV,* July 17, 2006, Portland, Maine, USA.

[20] Linux Tutorial: POSIX Threads [Online].
http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html
[abgerufen am 24.07.2008]

[21] developerWorks : Linux : Technical library view. Port Windows IPC apps to Linux, Part 1 – 3 [Online].
http://www.ibm.com/developerworks/views/linux/libraryview.jsp?search_by=port+windows+ipc+apps+linux [abgerufen am 25.07.2008]

[22] David R. Butenhof.
Programming with POSIX® Threads.
Addison-Wesley Professional, ISBN 0-2016-3392-2, 1997.