Parallele Programmierung ohne Spinlocks

Wie man Deadlocks vermeiden kann

Jens Harnisch, Li Lin, Albrecht Mayer, Gerhard Wirrer
Infineon Technologies AG

Um die Leistungsfähigkeit moderner Mehrkernprozessoren nutzen zu können, sind je nach Anwendung eine gewisse Synchronisation, zum Beispiel durch Barrieren, und ein Schutz von Ressourcen, zum Beispiel durch Spinlocks, notwendig. Dadurch können Deadlocks entstehen, welche natürlich besonders für sicherheitskritische Systeme unerwünscht sind. Ein alternatives Programmiermuster sind lockfreie Algorithmen. Diese müssen für die von mehreren Kernen gemeinsam genutzten Datenstrukturen speziell angepasst werden. Es wird eine unbegrenzte Queue vorgestellt, bewertet und in Bezug zu anderen Herangehensweisen gesetzt.

Motivation: Spinlocks und ihre Probleme

Mutexe, Semaphore und Events werden zur Synchronisation paralleler Prozesse und zum Schutz gemeinsam genutzter Ressourcen, z.B. Datenstrukturen, von Betriebssystemen bereitgestellt.

Zur Implementierung werden atomare Instruktionen der jeweiligen Rechnerarchitektur genutzt. Eine Pseudonotierung für ein Spinlock wird im Folgenden angegeben:

```
inline void get_spinlock(volatile unsigned long* spinlock_ptr)
{
      while (swap(spinlock_ptr, BUSY) == BUSY)
      ;
}
```

Die Funktion swap, innerhalb welcher dann der atomare Maschinenbefehl swap Befehl genutzt wird, liest den aktuellen Wert von der Adresse spinlock_ptr, und schreibt den Wert BUSY an die Adresse von spinlock_ptr. Die Ausführung des Maschinenbefehls an sich wird ohne Blockade funktionieren. Aber wenn der Spinlock nicht verfügbar wird, dann wird die while Schleife nicht verlassen, es entsteht ein Deadlock. So könnte es zum Beispiel sein, dass die Software auf einem anderen Kern abgestürzt ist, und daher der Spinlock nicht mehr freigegeben wird. Spinlocks können auch an Prioritätsinversion beteiligt sein. Bei mehreren Spinlocks im Design wächst die Komplexität. In der ersten Version von Linux für Mehrkernsysteme gab es daher nur den einen Big Kernel Lock. Allerdings wird damit teilweise die Leistungsfähigkeit eines Mehrkernsystems auf nur einen Kern reduziert.

Blockadefreie Programmierung: Prinzipien, Vor- und Nachteile

Programmierung ohne Blockade (Lock Free Programming) stellt eine seit vielen Jahren bekannte Alternative zur Arbeit mit Mutexen und Spinlocks dar [1]. Deadlocks können damit nicht mehr entstehen. Algorithmen ohne Locks müssen auf die zu schützende Datenstruktur angepasst werden, zum Beispiel eine Queue. Oft sind es zwar nur wenige Zeilen Quelltext, aber das Design ist komplex. Weiterhin ver-

schenkt man mit diesen Algorithmen potenziell Rechenleistung, durch die spekulative Ausführung. Zumindest dieses Problem relativiert sich bei Verfügbarkeit vieler Kerne. Das Prinzip lockfreier Programmierung ist gewöhnlich folgendes:

1. Anlegen einer Kopie der sensiblen Datenstruktur
2. Ausführung der Operation auf der Kopie der Datenstruktur
3. Vor Zurückschreiben des Ergebnisses überprüfen, ob ein anderer Prozess die Struktur mittlerweile verändert hat. Wenn ja, muss die Operation wiederholt werden, auf einer neuen Kopie der Struktur (zurück zu 1.). Falls die Datenstruktur zwischenzeitlich nicht verändert wurde, kann die berechnete Datenstruktur zurückgeschrieben werden.

Für die Überprüfung, ob die Datenstruktur zwischenzeitlich verändert wurde, kann man zum Beispiel einen Pointer nutzen. Zeigt der Pointer noch auf die gleiche Adresse wie beim Anlegen der Kopie, dann hat zwischendurch scheinbar kein anderer Prozess auf der Struktur gearbeitet. Bekannt ist aber das „ABA“ Problem. Dabei zeigt der Pointer beim Anlegen der Kopie auf Adresse A, und auch bei der Überprüfung nach Ausführung der Rechnung. Das kann aber Zufall sein. In Wahrheit zeigte der Pointer zwischendurch auf die Adresse B, ein weiterer Prozess hat die Struktur zwischenzeitlich manipuliert und sein Ergebnis zurückgeschrieben. Wenn wir dieses Problem nicht beachten, überschreiben wir das Ergebnis des weiteren Prozesses auf der sensiblen Datenstruktur. Zur Lösung des Problems kann zum Beispiel ein Zähler eingeführt werden, der bei jedem Zurückschreiben der Struktur erhöht wird.

Beispiel: Eine unbegrenzte Queue

Im Gegensatz zur Arbeit mit Mutexen oder Spinlocks, welche recht generisch eingesetzt werden können, muss lockfreie Programmierung an die jeweilige Datenstruktur angepasst werden, daher zum Beispiel an verlinkte Listen, Stacks oder Warteschlangen (Queues). Eine lockfreie unbegrenzte Queue, implementiert in C, wird im Folgenden vorgestellt. Die Software wurde auf einem AURIX 2nd Generation Microcontroller mit 6 Kernen in Betrieb genommen.

Kernelement der Queue ist eine Node, mit dem aktuellen Wert und einem Verweis auf den nachfolgenden Node:

```
struct Node
{
      int value;
      struct Node* next;
};
```

Das Design der Queue unterstützt unbegrenzt viele Elemente in der Queue. Eine unbegrenzte Anzahl von Elementen ist natürlich unrealistisch für ein eingebettetes System, aber die Unterstützung einer variablen Anzahl von Elementen, z.B. erkannter Objekte in einer Szene für autonomes Fahren, ist ein realistisches Szenario. Einfügen und Löschen eines Elements wird im Folgenden erklärt.

Einfügen eines Elements in die Queue (Enqueue): An Hand des Knotens mit dem Zeiger auf Ende (s. Bild 1) wird der letzte wirkliche Knoten ermittelt. Im Enqueue Schritt 1 wird das aktuelle Ende der Queue ermittelt (Knoten B), und der Zeiger next

dieses Knotens B wird umgesetzt von NULL auf die Adresse von Knoten A. Knoten A soll ein neuer Knoten in der Queue werden. Die Queue ist jetzt in sich konsistent, damit kann auch der Zeiger next des Knotens Tail auf die Adresse von Knoten A umgesetzt werden.

Löschen eines Elementes aus der Queue (Dequeue): Mit Hilfe des Knotens Head wird der aktuelle erste Knoten ermittelt, am Anfang ist das der Wächterknoten, auch genannt Sentinel. Der Knoten Sentinel zeigt auf den Knoten D, den eigentlichen letzten Knoten. In Dequeue Schritt 1 wird der eigentliche Wert des Knotens gelesen. In Schritt 2 wird der Zeiger next des Knoten Head umgesetzt auf Knoten D.

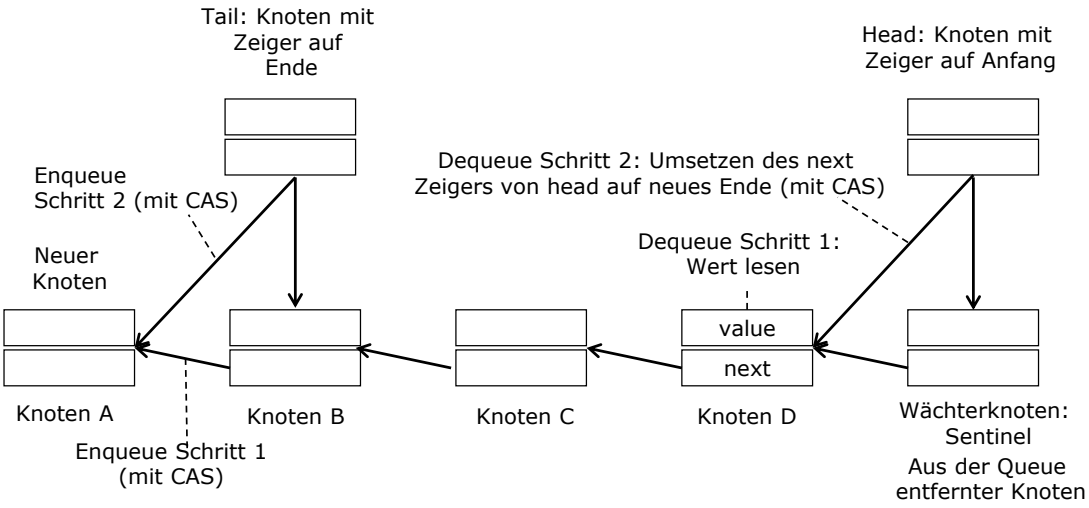

Bild 1: Einfügen eines neuen Knotens am Ende der Queue, und Lesen und Löschen eines Knotens am Anfang der Queue

Drei der ausgeführten Schritte müssen mit einer atomaren Compare-und-Swap Instruktion ausgeführt werden, im Bild mit CAS bezeichnet. Die Ausführung der atomaren Instruktion kann nicht blockiert werden. Man kann die obige Implementierung der Queue vergleichen mit herkömmlichen Implementierungen für eine Queue. Dort wird entweder ein Spinlock für die gesamte Queue verwendet, egal ob man ein Element löscht oder eines hinzufügt. Oder zwei Spinlocks werden verwendet, jeweils für Einfügen bzw. Löschen eines Elements. Aber alle diese herkömmlichen Implementierungen können potenziell blockieren.

Analyse und Test von nicht blockierenden Algorithmen sind eher anspruchsvoller als von blockierenden Algorithmen. Gleichzeitigkeit wird ja bei nicht blockierenden Algorithmen explizit unterstützt, und nicht zeitweise ausgeschlossen, wie bei Implementierungen mit Spinlocks. Auf der anderen Seite handelt es sich um ein weitgehend nichtsynchronisiertes Programmiermuster, und nicht um ein Programmiermuster mit Synchronisation, so wie bei Logical Execution Time. Programmierung mit nicht blockierenden Algorithmen ähnelt letztendlich eher Transactional Memory, bei welchem Transaktionen spekulativ ausgeführt werden, aber bei Konflikten wieder rückgängig gemacht werden. Die Komplexität nicht blockierender Algorithmen muss auch beim Testen berücksichtigt werden. Der entwickelte Algorithmus wurde nicht formal verifiziert sondern nur analysiert. Für diese Analyse war die Verfügbarkeit des nichtintrusiven Tracesystems MCDS (Multi-Core Debug Solution) auf der Zielplattform, dem AURIX 2nd Generation Microcontroller TC39x, sehr

hilfreich. So konnten verschiedene Szenarien der Gleichzeitigkeit genau nachverfolgt werden, z. B. die Sequenz der Zugriffe durch verschiedene Kerne für eine Korrektheitsanalyse aber auch das exakte Zeitverhalten für eine Performanzanalyse. Beides wird durch besondere MCDS Eigenschaften ermöglicht. Dazu zählt der parallele und zeitlich ausgerichtete Trace an verschiedenen Beobachtungspunkten mit sehr hoher zeitlicher Auflösung.

Fazit

Da mit lockfreier Programmierung Deadlocks schon per Design ausgeschlossen sind, kann die Nutzung vor allem in sicherheitskritischer Software vorteilhaft sein. In [1] wurde dokumentiert, wie durch lockfreie Programmierung sogar eine höhere Performanz als mit Spinlocks erreicht wurde. Auf der anderen Seite muss lockfreie Programmierung immer speziell zugeschnitten werden auf die zu schützende Datenstruktur, und die Implementierungen sind schwerer verständlich als die Nutzung eines Spinlocks. Obwohl lockfreie Programmierung schon seit vielen Jahren bekannt ist [1], ist die Verbreitung eher beschränkt. Die Unterstützung lockfreier Programmierung in der C++ STL durch Execution Policies wird die Verbreitung sicher erhöhen. Da C++ auch im Embeeded Bereich zunehmend akzeptiert wird, einschließlich eingeschränkter Nutzung der STL, wird lockfreie Programmierung sicher auch dort ihren Platz finden, auch in sicherheitskritischen Systemen mit Anforderungen für harte Echtzeit. Um eine konkrete Implementierung zu überprüfen, ist hardwareunterstütztes Tracing auf dem Microcontroller das Mittel der Wahl.

Literatur- und Quellenverzeichnis

[1] Henry Massalin and Calton Pu. A Lock-Free Multiprocessor OS Kernel. Technical Report CUCS-005-91, Columbia University, 1991.

Autoren

Jens Harnisch ist Tool Partner Manager bei Infineon und arbeitet im Application und Concept Engineering an Softwarekonzepten zur Nutzung aktueller und zukünftiger Microcontroller.
jens.harnisch@infineon.com

Dr. Albrecht Mayer ist bei Infineon als Senior Principal für die AURIX On-Chip Debug und Trace Architektur verantwortlich.
albrecht.mayer@infineon.com

Li Lin hat bei Infineon an seiner Dissertation auf dem Gebiet von Mehrkernmikrokontrollern gearbeitet und ist mittlerweile im Application und Concept Engineering im Bereich Debug und Trace tätig.
li.lin@infineon.com

Gerhard Wirrer ist Lead Principal Software Architect Automotive Microcontrollers bei Infineon Technologies.
gerhard.wirrer@infineon.com