Was ist eigentlich Nebenläufigkeit?

Moritz Strübe, MATHEMA GmbH

Jeder, der mit Interrupts, Threads oder Coroutinen arbeitet oder auch nur externe Hardware ansteuert, kommt mit Nebenläufigkeit (Concurrency) in Berührung. Und während viele Entwickler eine Vorstellung von dem Begriff haben, so fällt es doch vielen schwer diesen klar und prägnant zu definieren. Eine klare Definition hilft nicht nur bei der Kommunikation, sondern auch beim Erkennen und Lösen von Problemen, die durch Nebenläufigkeit entstehen.

Zielsetzung

In diesem Beitrag soll vor allem das Bewusstsein für die Problematik von Nebenläufigkeit und daraus resultierenden Problemen gestärkt werden. Ist ein Nebenläufigkeitsproblem erkannt und verstanden, ist das Problem schon fast gelöst. Auch die klassischen Methoden wie man Nebenläufigkeitsprobleme löst werden andiskutiert.

Was ist eigentlich Nebenläufigkeit?

Eine Recherche im Netz bringt unzählige Definitionen zutage. Die Auswahl zeigt das Spektrum auf:

1. Wenn viele Dinge gleichzeitig passieren, nennen wir ein interagierendes System nebenläufig. (https://openbook.rheinwerk-verlag.de/javainsel/16_001.html)
2. Nebenläufigkeit ist gegeben, wenn auf einer Strukturebene zu einem Zeitpunkt mehrere, in ihrer Art vergleichbare Aufgaben, ausgeführt werden können. (https://tu-dres-den.de/zih/ressourcen/dateien/lehre/ss2012/tevpr_content/V1_Einfuehrung_4_on_1.pdf)
3. Die Nebenläufigkeit, mitunter auch Parallelität (englisch concurrency) genannt, ist in der Informatik die Eigenschaft eines Systems, mehrere Aufgaben, Berechnungen, Anweisungen oder Befehle gleichzeitig ausführen zu können. (https://de.wikipedia.org/wiki/Nebenl%C3%A4ufigkeit)
4. Mit Nebenläufigkeit bezeichnet man die Fähigkeit eines Systems, zwei oder mehr Vorgänge gleichzeitig oder quasi-gleichzeitig ausführen zu können. (https://dbs.cs.uni-duesseldorf.de/lehre/docs/java/javabuch/html/k100141.html)

Während die erste, flapsige Definition aus „Java ist eine Insel“, im Buch noch klargestellt wird, fehlt ihr die Prägnanz. Die zweite ist falsch. Die dritte, aus der Wikipedia, ist unpräzise, denn nur die vierte geht auf Quasiparallelität ein. Quasiparallelität ist, wenn sich das System verhält, als würde es eine parallele Ausführung unterstützen. Zum Beispiel, wenn mehrere Threads auf nur einen Kern laufen. Wenn man sich Nebenläufigkeitsprobleme betrachtet merkt man schnell, dass es für diese egal ist, ob die Nebenläufigkeit durch echte- oder quasi-parallelität entsteht.

Wichtig ist, insbesondere in Kontext der hardwarenahen Softwareentwicklung, dass Nebenläufigkeit nicht nur auf dem CPU-Kern passiert, sondern auch in Kontext der angeschlossenen Peripherie. Zum Beispiel der angeschlossene ADC-Wandler der parallel zur Programmausführung Register mit Messwerten aktualisiert.

Nebenläufigkeitsprobleme

Nebenläufigkeit an sich ist kein Problem. Sie erlaubt es Dinge (Qausi-)Parallel auszuführen und so das System reaktiver zu machen. Die Probleme entstehen in dem Moment, in dem auf gemeinsame Daten zugegriffen wird.

Ich möchte dies im Folgenden anhand eines einfachen Beispiels demonstrieren: Auf einem 8-Bit Microkontroller wird ein Zähler (`foo`) mit 1 initialisiert und in einem Interrupt hochgezählt, wobei sichergestellt ist, dass dieser nicht überläuft. Im Hauptpro-gramm wird geprüft, ob der Zahler 0 ist und in diesem Fall in einen Fehlerfall (`panic()`) gewechselt, da dies nie vorkommen sollte. Obwohl der Zähler nie `0` ist, kann es dennoch passieren, dass der Fehlerfall eintritt. Im Folgenden ist so ein Fehlerfall aufgezeigt:

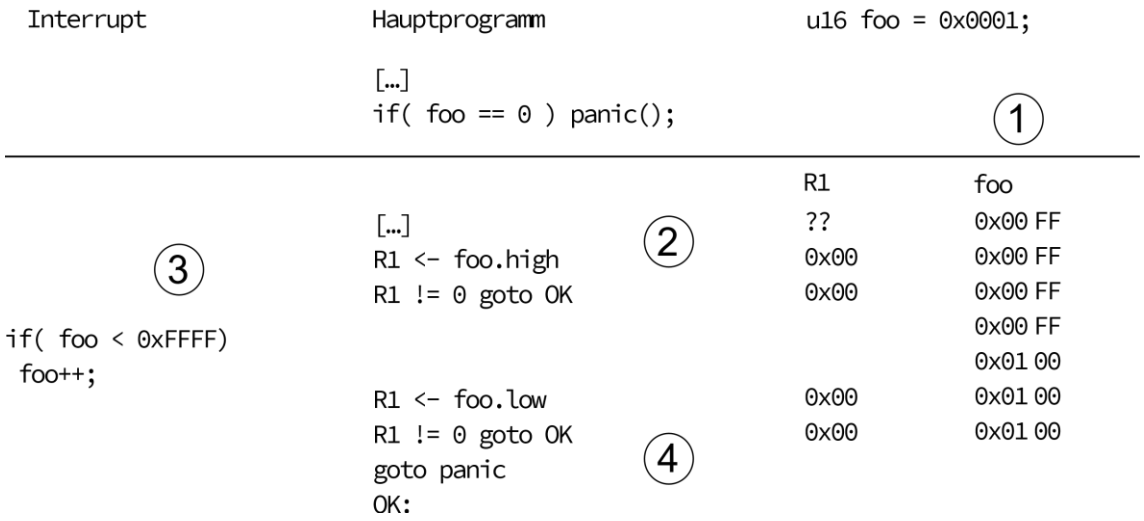

Abbildung 1Beispiel eines Nebenläufigkeitsproblems auf einem 8 Bit Mikrocontroller

In Abbildung 1 steht links der Code des Interrupts. Da dieser nicht unterbrochen wird, ist der Assemblercode in diesem Fall uninteressant. In der Mitte der code des Haupt-programms, oben der C-Code und unten eine mögliche Übersetzung in Assembler-Pseudocode. Rechts daneben der aktuelle Wert des CPU-Registers R1 und der Wert der Variablen im RAM.

1) Spannend ist in diesem Beispiel der Übergang von `foo` von `0x00FF` zu `0x0100`, daher wird davon ausgegangen das `foo` den Wert `0x00FF` hat.

2) Da `foo` 16 Bit groß ist, der Adressbus aber nur 8 Bit breit ist, wird zunächst das höherwertige Byte geladen. Ist dieses ungleich `0`, kann der Wert nicht `0` sein, und die Prüfung des zweiten Bytes kann entfallen. Es wird daher zu OK gesprungen. In unserem Beispiel ist der Wert `0` und der Code-Fluss wird fortgesetzt.

3) Wir gehen davon aus, dass der Interrupt genau in diesem Moment zuschlägt und den Zähler erhöht.

4) Nun wird das zweite Byte geladen. Inzwischen ist das niederwerte Byte `0` und wird in das Register geladen. Da auch dieses Null ist, wird nun die Fehlerbahandlung angesprungen.

Unabhängig davon, ob der Compiler den Code wie im Beispiel übersetzt oder nicht, es müssen immer zwei Byte geladen werden und der Interrupt kann zwischen den beiden Ladevorgängen auftreten. Hier hilft auch kein volatile, da dieses Verhalten nicht im Zusammenhang mit einer Compileroptimierung steht Das heißt jedoch nicht, dass je nach Implementierung kein volatile benötigt wird.

Auch wenn das Problem hier beispielhaft für einen 16 Bit Wert und einen 8 Bit Mikrocontroller demonstriert wurde, so trifft es natürlich alle Datenstrukturen, die die CPU nicht atomar mit einer Instruktion laden kann.

Was auch klar wird, ist das dieser Fehler sehr perfide ist. Denn wie hoch ist die Wahrscheinlichkeit, dass das Interrupt genau zu diesem Zeitpunkt zuschlägt? Dies kann Tage, Wochen, Monate oder Jahre dauern. Und dann wird es auch praktisch unmöglich diesen zuverlässig zu reproduzieren.

Lösungen

Ist das Problem erkannt, so muss eine Lösung gefunden werden.

Programmiert man bare metal ist Interrupts abschalten, bei einfachen Operationen, wie in diesem Beispiel, eine einfache und wirkungsvolle Lösung. Der Interrupt wird um ein paar wenige Takte verzögert und das Problem vermieden.

Hat man ein Betriebssystem mit mehreren Threads wird meist auf Locks zurückgegriffen. Hierbei stellt eine bereitgestellte Funktion sicher, dass immer nur genau einer das Lock besitzen darf. Die Entwickler der zugehörigen Funktionen haben sollten dann sichergestellt haben, dass es nicht zu Nebenläufigkeitsproblemen kommt. Dies kann das Abschalten der Interrupts sein. Der Vorteil hier ist jedoch, dass dies für einen gemacht wird.

Aber Locks sind kein Allheilmittel. Nehmen wir das Beispiel mit dem Interrupt: Was soll der Interrupt machen, wenn das Hauptprogramm das Lock hält? Er kann nicht ein-fach warten, bis das Hauptprogramm diesen wieder frei gibt. In diesem Fall hätte man ein sogenanntes Dead-Lock: Das Programm steckt wegen des Interrupts fest und der Interrupt wartet auf die Freigabe des Locks.

Locks haben auch eine Vielzahl anderer Probleme, die den Rahmen hier sprengen würde und in der einschlägigen Literatur nachzulesen sind. Was sie aber meist gemein haben: Sie sind, wie die meisten Nebenläufigkeitsprobleme, sporadisch und zum Teil schwer zu reproduzieren.

Lock-Frei Sychronisieren

Robuster sind sogenannte Lock-Freie Synchronisationsmechanismen. Diese erlauben es verschiedene Ablauffäden ohne Blockaden zu Synchronisieren.

In dem aufgezeigten Beispiel findet die Modifikation des Interrupts Atomar statt. In einem solchen Fall kann man den Wert so auf aus dem Speicher auslesen, bis man zwei Mal hintereinander dieselbe Variable ausliest. Dies setzt natürlich voraus, dass der Interrupt nicht zwei Mal zu genau dem richtigen Zeitpunkt zuschlägt und die Daten so modifiziert, dass wieder ein falscher Wert ausgelesen werden kann. Meist kann man dies mit einer einfachen Systemanalyse ausschließen. Zum Beispiel, wenn man sich sicher ist, dass zwischen zwei Interrupts genug Zeit vergeht.

Das funktioniert allerdings nur, weil es genau einen Schreiber, den Interrupt, gibt. Gibt es mehrere Schreiber, so muss man sicherstellen, dass nicht ein anderer Schreiber den Wert seit dem letzten Auslesen modifiziert hat. Die meisten 32-Bit Mikrocontroller haben spezielle Instruktionen, die den Speicher unter bestimmten Bedingungen atomar modifizieren. Der bekannteste ist das CAS (Compare and Swap). Hierbei wird der Wert nur dann modifiziert, wenn er einen erwarteten Wert hat. Wenn man sich den Mechanismus genauer anschaut, werden die drei Instruktionen (Laden, vergleichen, bedingtes schreiben des neuen Wertes) auch hier unter einem Lock ausgeführt. Allerdings handelt es sich um ein Hardware-Lock auf dem Speicherbus, welches dafür sorgt, dass die Instruktionen atomar ausgeführt werden können.

Außer dem CAS gibt es noch viele weitere Mechanismen, die dieses Verhalten zum Teil mit weiteren Vorteilen verknüpfen. Hier lohnt es sich etwas Zeit mit dem Handbuch zu verbringen, oder auf entsprechende Bibliotheken zurückzugreifen.

Alles nicht so einfach

Die Problematik der Synchronisation wurde hier nur angerissen. Befasst man sich näher damit so tauchen immer neue Aspekte auf, die einem das Leben schwer machen. So dürfen C und C++-Compiler bei der Optimierung davon ausgehen, dass es keine Neben-läufigkeit gibt – außer man sagt es ihnen explizit. Dies hat zur Folge, dass sie Speicherzugriffe nach Belieben verschieben und/oder neu anordnen dürfen. Ein so ein Hinweis ist volatile. Auf solche Variablen muss immer und in der richtigen Reihenfolge zugegriffen werden. Unter diesen Umständen ist bereits die Implementierung eines einfachen Ringpuffers mit einem Leser und einem Schreiber alles andere als einfach. Im Zweifel sollte man immer noch einen Blick in den Assemblercode werfen.

Zum Glück gibt es aber auch reichlich Unterstützung in Form von Bibliotheken. Sowohl ab C++11 als auch C11 gibt es eine atomic-Bibliothek, die einen unterstützt daten Atomar zu modifizieren. Die meisten Betriebssysteme, und sein es auch nur einfache Bibliotheksbetriebsysteme, bieten meist verschiedene Möglichkeiten Informationen von einem Thread zum anderen zu schieben. Hat man kein Betriebssystem lohnt sich fast immer ein Blick in die vorhandenen: Hier bietet sich unter anderem Contiki-OS und Contiki-NG (beide BSD) oder das arm MBED OS (Apache 2.0) an. Insbesondere Contiki ist noch sehr einfach, so dass man hier – natürlich unter Beachtung der Lizenz – für die eigenen Bare-Metal-Projekte gut räubern kann. Andere bekannte OS wie freeRTOS (MIT) oder Zephyr OS (Apache 2.0), und zum Teil auch schon das arm MBED OS, haben häufig das „Problem“, dass die Synchronisationsmechanismen zu sehr mit den Betriebssystem-kern verbandelt sind.

Autor

Moritz Strübe entwickelt seit über 15 Jahren Software für eingebettete Systeme. Sein Schwerpunkt liegt bei der Entwicklung mit C/C++ für Mikrocontroller, Architektur und der Qualitätssicherung durch die Verwendung von verschiedenen Werkzeugen in der CI. Seit sieben Jahren unterstützt er bei der Mathema Kunden bei der Softwareentwicklung. Hierbei stößt er immer wieder auf die Themen Echtzeit, Nebenläufigkeit und Security.

Kontakt

Internet: www.mathema.de

Email: moritz.struebe@mathema.de