9
Multithreading

Jörn Loviscach

Versionsstand: 21. Juli 2015, 11:50

Die nummerierten Felder sind absichtlich leer, zum Ausfüllen beim Ansehen der Videos:
http://www.j3L7h.de/videos.html

Bitte hier notieren, was beim Bearbeiten unklar geblieben ist

1 Idee des Multithreading

In den vergangenen Jahren haben sich die Taktgeschwindigkeiten der Prozessoren praktisch kaum erhöht; statt dessen sitzen auf dem selben Chip immer mehr eigenständige „Kerne“ [Cores], die parallel arbeiten. Hochleistungsrechner und viele Server haben sogar mehrere Prozessor-Chips, die jeweils mehrere Kerne enthalten. Um die Recheneinheiten jedes Kerns gut auszulasten, tun die größeren modernen Prozessoren so, als ob jeder echte Kern zwei Kernen wären (Hyperthreading). Für einen Quad-Core-Prozessor sieht die Software dann acht Kerne. Weit darüber liegen Grafikkarten mit inzwischen mehr als 1000 Kernen. Diese sind allerdings bei weitem nicht nicht so unabhängig voneinander und nicht so vielseitig wie die des Hauptprozessors.

Statt dass man einen Arbeiter Akkord schuften lassen muss, hat man also mehrere Arbeiter, die parallel arbeiten können. Das setzt aber voraus, dass die Arbeit auch verteilt wird: Mehrere Arbeiter müssen gleichzeitig tätig sein.

[1]

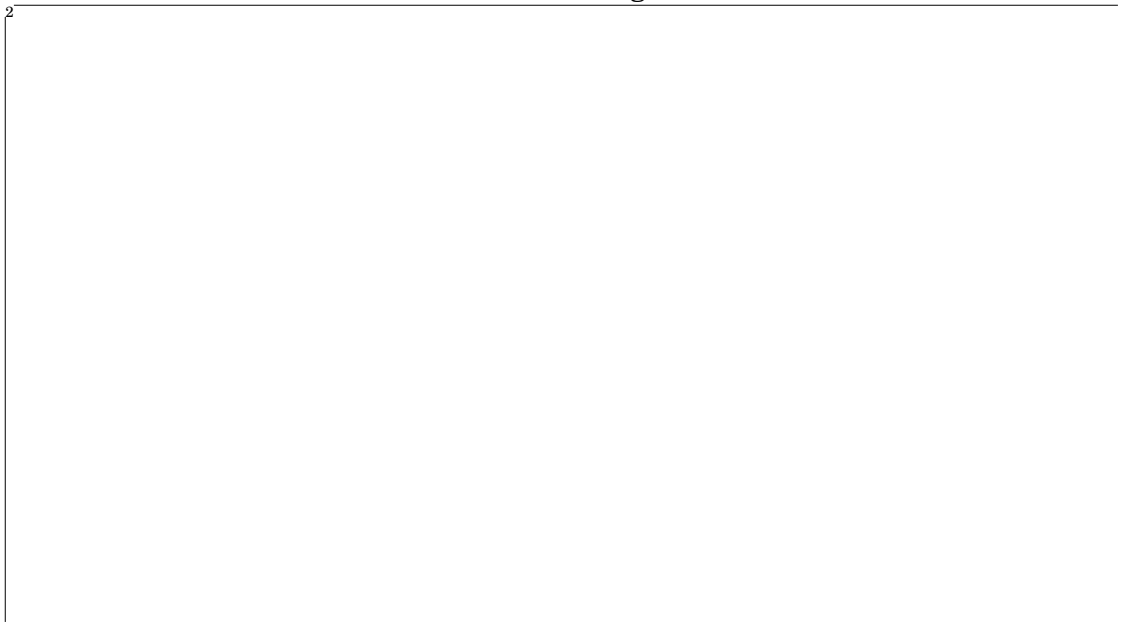
Eine einfache Art, mehrere Kerne zu nutzen, ist, mehrere Programme („Prozesse“) gleichzeitig laufen zu lassen. Gibt es nur einen Kern, muss das Betriebssystem schnell zwischen allen Programm hin und her schalten, damit jedes läuft – mit entsprechend verringerter Geschwindigkeit. Mit mehreren Kernen ist das Umschalten seltener oder gar nicht mehr nötig. Demo im Windows Task Manager: CPU-Auslastung, Priorität und Zugehörigkeit.

Aber auch ein einziges Programm kann mehrere Kerne benutzen: durch Multithreading. Das Programm ist dann zu einem gegebenen Zeitpunkt nicht an genau einer Stelle im Code, sondern an mehreren Stellen gleichzeitig. Es ist in Threads („Fäden“) zerfasert; diese können jeweils auf verschiedenen Kernen laufen. Das Betriebssystem verteilt die Threads automatisch auf die Kerne. Threads können verschiedene Prioritätsstufen haben: So muss das Betriebssystem eine Tonwiedergabe sehr regelmäßig mit Arbeitskraft versorgen; ein Virenscanner darf dagegen auch mal eine Sekunde Zwangspause einlegen, wenn die Rechenleistung knapp wird. Demo im Process Explorer (Download-Link): Zahl und Aktivität von Threads in Programmen.

2 Threads, volatile

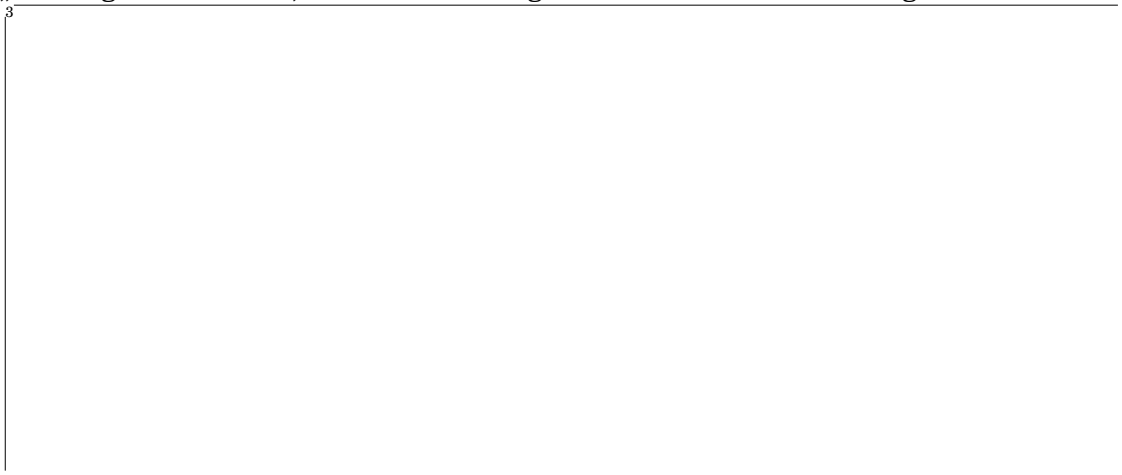
Jedes Programm startet mit einem einzigen Thread. Man kann aber weitere Threads anlegen, denen Namen geben (zum Debuggen!), deren Priorität einstellen

und ihnen vor allem Methoden zum Arbeiten geben:

[2]


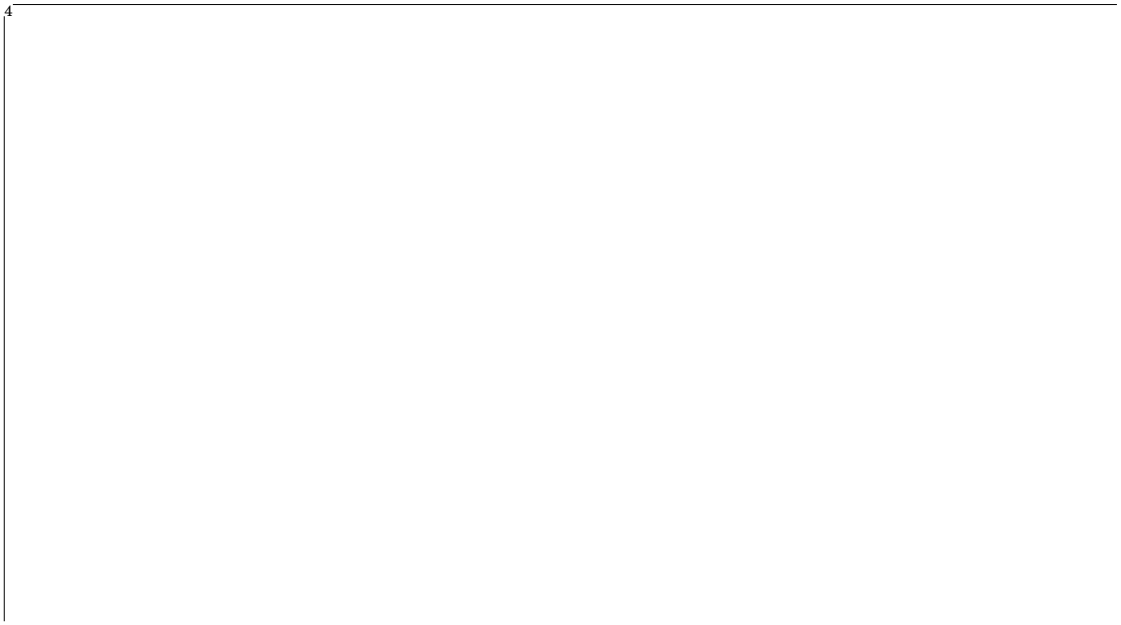
Beim Programmhalt zeigt der Debugger von Microsoft Visual Studio im Fenster „Threads“, welcher Thread gerade im Einzelschrittdebugger sichtbar ist. Mit Mausklick rechts auf den Thread kann man zu einem anderen Thread wechseln.

Ist die Methode beendet, die ein Thread ausführt, endet auch der Thread. Oft hat man in solchen Methoden aber Endlosschleifen, zum Beispiel, um dauerhaft auf Nachrichten aus dem Internet zu lauschen. Dann kann man den Thread auf „Hintergrund“ stellen, damit er am Programmende automatisch abgebrochen wird:

[3]


Sicherer ist, die Endlosschleife mit einer Abfrage zu versehen, ob der Thread enden soll. Dann kann man die Arbeit geordnet beenden:

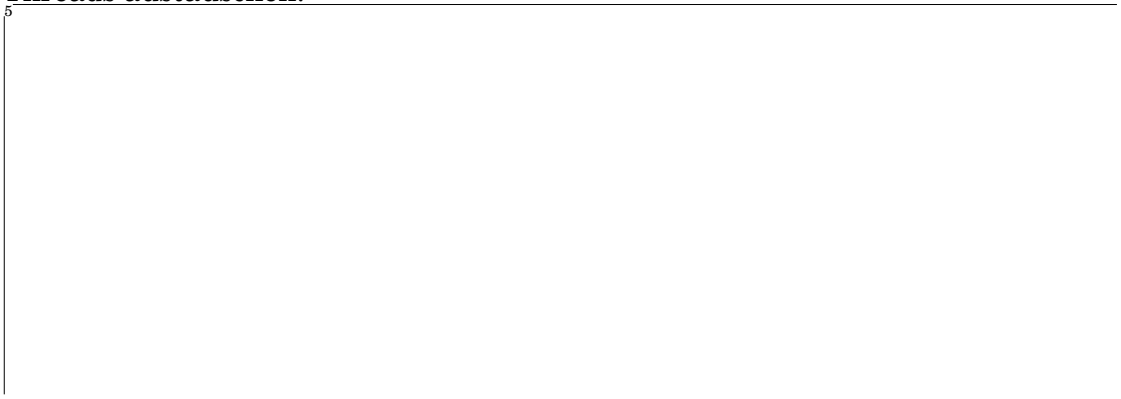
4



Hier steht mit `volatile` („flüchtig“) ein neues Schlüsselwort. Das verbietet, dass der Compiler die Reihenfolge der Befehle großzügig optimiert, und erzwingt, dass der Wert im Speicher aktuell gehalten wird, statt – was effizienter wäre – nur auf dem jeweiligen Kern. Letzteres kann dazu führen, dass jeder Kern einen anderen Wert der Variablen sieht.

Mit Hilfe von `volatile` kann man auch andere einfache Datentypen zwischen Threads austauschen:

5



Vorsicht: Das funktioniert meist auch ohne `volatile`, aber eben nur *meist*. Ein vergessenes `volatile` verursacht Fehler, die nur auf manchen Systemen und dort auch nur unter unklaren Bedingungen auftreten.

Auch in C, C++ und Java gibt es das Schlüsselwort `volatile`. In C und C++ hat es aber eine andere Bedeutung. Für Mikroprozessoren, die in C oder C++ programmiert werden, ist `volatile` wichtig, wenn Register gelesen werden, die sich ohne Zutun des Programms ändern können, und wenn es um Interrupt-Routinen geht, also Routinen, die zum Beispiel durch einen Zeitgeber oder durch ein von außen angelegtes Signal ausgelöst werden. In C++11 hat `std::atomic< >` die Bedeutung von `volatile` in Java und C#.

In Java und in der neuesten Version C++11 von C++ werden Threads ähnlich behandelt wie in C#.

3 Parallelisierte for-Schleife

Es gibt viele Versuche, den Umgang mit Threads zu vereinfachen. Auf vielen Systemen findet sich eine Variante der `for`-Schleife, in der mehrere Schleifendurchgänge gleichzeitig ablaufen – in verschiedenen Threads. Microsoft hat das und einige andere Funktionen als „Task Parallel Library“ in .NET 4.0 eingebaut:[6]

4 Synchronisation von Threads

So lange jeder Thread getrennt von allen anderen für sich alleine läuft, hat man wenig Ärger. Das Heikelste an der Programmierung mit Threads ist das Zusammenspiel zwischen mehreren Threads. Die dürfen sich nicht ins Gehege kommen.

Das einfachste Beispiel dafür ist, dass mehrere Threads eine gemeinsame Variable zum Zählen nutzen – etwa, um die Gesamtzahl von Suchtreffern auf verschiedenenen Webseiten zu bestimmen. Bei einem Treffer liest der jeweilige Thread den alten Wert der Variable ein, erhöht den um eins und schreibt den erhöhten Wert in die Variable zurück. So sollte das passieren:[7]

Man hat allerdings hier einen kritischen Wettlauf [race condition], der schief gehen kann:

[8]

Solche Situationen sorgen für schwer nachvollziehbare Fehler. Ein großer Teil der aktuellen Informatik-Forschung befasst sich mit Methoden, Fehler dieser Art zu finden – oder zu vermeiden.

So sieht das in C# aus:

[9]

Die übliche Lösung besteht darin, dafür zu sorgen, dass in „kritischen“ Codeblöcken nur ein einziger Thread sein darf. Will ein weiterer Thread einen solchen Codeblock ausführen, wird er am Anfang angehalten, bis der bisherige Thread den Codeblock verlassen hat. In C# erzeugt man dazu ein Dummy-Objekt, das sich merkt, ob irgendein Thread einen der damit gesicherten Codeblöcke betreten hat:

[10]

Wenn alle Zugriffe auf die gemeinsame Variable immer innerhalb von `lock` stehen, muss die nicht mehr als `volatile` deklariert werden, denn auch `lock` verhindert die entsprechenden Optimierungen.

In Java schreibt man `synchronized` statt `lock`.

Zum sicheren Erhöhen einer Variablen und für ähnliche einfache Aufgaben kommt man eigentlich ohne `lock` aus: Dafür gibt es vorgefertige Methoden in `System.Threading.Interlocked.Increment`. Das `lock` sieht man eher, wenn zum Beispiel eine Liste aus verschiedenen Threads gefüllt wird. Es dürfen dann nicht mehrere Aufrufe von `Add` gleichzeitig passieren.

Mit der Synchronisation kann man race conditions und die daraus entstehenden, schwer zu findenden Fehler vermeiden. Gleichzeitig handelt man sich neue, schwer zu findende Fehler ein: Ein Thread, der innerhalb eines mit `lock` markierten Abschnitts ist, lässt andere Threads vor dem `lock` warten. Mit Pech wartet Thread A auf Thread B, aber gleichzeitig an anderer Stelle Thread B auf Thread A – oder noch komplizierter mit mehreren Threads, die reihum aufeinander warten. Dann geht nichts mehr: Deadlock. Solche Sitationen kann man zum Beispiel mit Petri-Netzen analysieren.