Echtzeitsysteme – Nicht-blockierende Synchronisationsverfahren

Benedikt Dremel
Friedrich-Alexander-Universität Erlangen-Nürnberg
Benedikt.Dremel@e-technik.stud.uni-erlangen.de

Zusammenfassung

Die Anforderungen an Echtzeitsysteme werden immer komplexer und erfordern daher eine steigende Performanz. Dadurch werden Mehrkernprozessoren auch in diesem Bereich immer interessanter.
Dabei stellt der Zugriff verschiedener Fäden auf gemeinsame Ressourcen eine besondere Herausforderung dar: Hier kann es zu unter anderem zu Verklemmungen oder unbegrenzten Prioritätsverletzungen kommen. Nicht-blockierende Synchronisation ist dabei eine Realisierungsmöglichkeit der Kommunikation zwischen Fäden. Im Folgenden sollen drei Verfahren aufgezeigt werden, die eine solche nicht-blockierende Synchronisation erlauben. Dabei wird gezeigt, dass zwei der Verfahren auf für Echtzeitsysteme bedingt geeignet sind, der Ansatz, mittels eines sogenannten CAS-Befehls jedoch nicht.

1. Einleitung

Echtzeitsysteme finden und fanden bereits Eingang in viele Bereiche der Technik, so ist in jedem modernen Auto eine Vielzahl an Steuer- und Regelungsaufgaben in digitaler Form umgesetzt. Dabei werden die einzelnen Aufgaben immer mehr integriert und dadurch versucht, Synergien zu nutzen.
Deshalb und aus Kostengründen wird häufig versucht, Aufgaben auf möglichst wenige Recheneinheiten zu binden, wobei die Performanz von Einprozessorsystemen inzwischen oft nichtmehr ausreicht. Daher scheinen Mehrprozessorsysteme eine erfolgsversprechende Möglichkeit, die kostensparend und trotzdem hinreichend performant ist.

In diesem Kontext stellt die Kommunikation der Fäden eine Herausforderung dar. Der Zugriff auf gemeinsame Datenstrukturen kann zu Verzögerungen in Folge von Blockierung führen. Somit kann keine maximale Ausführungszeit (***WCET: worst-case-execution-time***) bestimmt werden und die Einhaltung von Terminen (***Deadlines***) ist nicht garantierbar. Das Verpassen von Terminen ist – zumindest für harte Echtzeitsysteme – nicht tolerierbar.
Dabei werden traditionell blockierende Verfahren zur Synchronisation eingesetzt. Blockierende Verfahren zeichnen sich dadurch aus, dass sie kritische Abschnitte definieren, die nur ein Faden exklusiv zu einem Zeitpunkt betreten darf. Alle anderen Fäden, die zur selben Zeit diesen Abschnitt betreten wollen, werden blockiert. Dabei sind binäre und zählende Semaphore die mithin bekanntesten Verfahren.
Bei diesen Varianten kann es zu Verklemmungen (***Deadlocks***) kommen, wenn es eine zyklische Kopplung zwischen Sperren gibt. Auch kann es zu unbestimmte Prioritätenumkehr kommen, wenn ein niederpriorer Faden einen höherprioren Faden beim Zugriff auf gemeinsame Daten „ausbremst“ (vergleiche dazu Kapitel 3). Insgesamt besteht die Gefahr, durch viele blockierende Programmteile Effizienzverluste akzeptieren zu müssen, da Parallelität an diesen Stellen verhindert wird.
Daher sind alternative Verfahren von Interesse, die diese Probleme umgehen. Eine Variante sind sogenannte nicht-blockierenden Synchronisationsverfahren, die auf das klassische Blockieren von Fäden verzichten.

Im ersten Kapitel wird die Problematik genauer erläutert und die Praxisrelevanz solcher Verfahren aufgezeigt.
Im darauf folgenden wird auf einige Grundlagen, wie die CAS-Instruktion oder auch unbestimmte Prioritätenumkehr, eingegangen, die für das Verständnis des Restlichen nötig sind.
Im dritten Kapitel werden drei Algorithmen erklärt und analysiert, die solche nicht-blockierende Synchronisation ermöglichen. Das sind sowohl die Variante mit CAS-Instruktion als auch das „Helping mit Wartestapel“ sowie der Single-Server.
Daraufhin werden diese Verfahren hinsichtlich ihrer Echtzeitfähigkeit und Multiprozessorkompatibilität geprüft.
Abschließend wird ein Fazit gezogen.

2. Grundlagen

Im folgenden sollen einige Grundlagen erklärt werden, die für das Verständnis der Algorithmen und deren Analyse wichtig sind.

2.1 Der CAS-Befehl

Bei nicht-blockierenden Synchronisationsverfahren werden oft spezielle Maschinenbefehle verwendet. Dabei ist der compare-and-swap-Befehl (CAS) von Bedeutung:
Dieser Befehl vergleicht eine Speicherstelle mit einem Wert und überschreibt diese, wenn der Vergleich positiv ausfällt. Wenn der Vergleich fehlschlägt, wird der Wert der Speicherzelle nicht verändert. Dieser Maschinenbefehl wird atomar ausgeführt. Somit ist ein atomares Vergleichen und Umsetzen z.B. eines Zeigers möglich. Eine mögliche

Implementierung des CAS-Befehls wird in Listing 1 gezeigt.

```
1   bool CAS(int *speicherstelle, int alt,
2            int neu)
3   {
4           if(*speicherstelle != alt)
5           {
6                   return false;
7           }
8           *speicherstelle = neu;
9           return true;
10  }
```

Listing1: Pseudocode für einen CAS-Befehl

Auf Einprozessormaschinen könnten fehlende Maschinenbefehle durch Ausschalten der Interrupts und Implementierung des Befehl in Software theoretisch nachgebaut werden. Dies ist jedoch auf Mehrprozessorsystemen nicht der Fall, da hier echte Parallelität herrscht und Nebenläufigkeit auch zu Fäden auf anderen Prozessoren besteht (vgl. [5]).

3.2 Echtzeitfähigkeit

Wie bereits aufgezeigt, ist es wichtig, dass eine gewisse Aufgabe sicher bis zu einem gewissen Zeitpunkt fertiggestellt ist. Das kann eine Motorsteuerung oder auch eine Bremskraftverstärkung in einem Automobil sein. In beiden Fällen muss jedoch die Berechnung garantiert nach einer gewissen Zeit fertig sein, sonst wird das Ergebnis wertlos, weil beispielsweise ein Bremsen nicht mehr möglich ist. Diese Eigenschaft kann jedoch nicht mit erhöhen von Geschwindigkeit erreicht werden, sondern es muss analytisch die Einhaltung der Termine garantiert werden.
Dafür gibt es die Möglichkeit einen festen Ablaufplan zu erstellen, bei dem verschiedene Aufgaben nacheinander ausgeführt werden und die Laufzeiten der einzelnen Jobs zuvor bestimmt werden. Dieses Verfahren wird als statisches Scheduling bezeichnet und ist sehr sicher gegen Terminverletzungen, jedoch aber auch recht unflexibel.
Weitaus variabler ist eine prioritätsgesteuerte Ablaufplanung. Bei dieser wird jedem Faden eine Priorität zugeordnet. Ein echtzeitfähiger Ablaufplaner (***Scheduler***) garantiert daraufhin, dass ein Faden mit höherer Priorität vor einem mit niedriger eingeplant wird. Problematische sind dabei jedoch gemeinsame Datenstrukturen. Hier kann es durch gegenseitigen Ausschluss zu unbestimmter Prioritätenumkehr oder Verklemmungen kommen.

3.3 Unbestimmte Prioritätenumkehr

Die meisten der in der Praxis eingesetzten Echtzeitbetriebssysteme sind prioritätsgesteuert. Das bedeutet, dass jeder Faden eine Priorität besitzt und der Faden mit der höheren immer bevorzugt wird. Bei präemptiven Systemen, die Verdrängung eines Threads durch einen anderen zulassen, wird somit ein niederpriorer Faden damit durch einen höherprioren verzögert.
Damit ist aber auch das Szenario vorstellbar, das in Abbildung 1 gezeigt wird:
Ein niederpriorer Faden A belegt einen kritischen Abschnitt, wie in de Zeichnung zu Zeitpunkt α geschieht. Während er diesen abarbeitet wird jedoch ein höherpriorer Faden B bereit (β) und wird eingelastet. Dieser versucht ebenfalls den selben kritischen Abschnitt zu betreten (γ), muss jedoch warten, da dieser bereits von Faden A belegt ist. Während Faden A nun weiterarbeitet, wird ein mittelpriorer Faden C bereit und wird eingelastet (δ). Dieser wird eine unbestimmte Zeit ausgeführt, bis wieder Faden A an der Reihe ist (ε) und den kritischen Abschnitt zu Ende bringt. Erst dann kann Faden B den kritischen Abschnitt betreten (ζ).

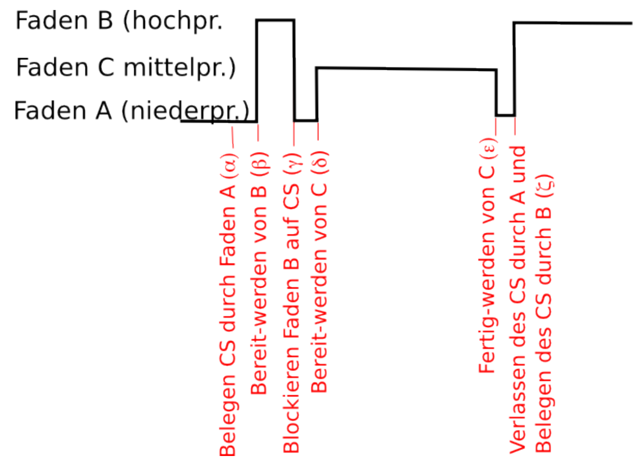

Abbildung 1: Unbestimmte Prioritätenumkehr. Ein mittelpriorer Faden C verzögert einen hochprioren Faden B, der vom niederprioren Faden A abhängt.

Das Problematische an diesem Beispiel ist, dass der mittelpriore Faden C den hochprioren Faden B ausbremst, obwohl keine Kopplung zwischen beiden bestehen muss. Dadurch kann keine Aussage über die Wartezeit von Faden B gemacht werden.
Anders ausgedrückt, sind dadurch die Prioritäten der Fäden B und C umgekehrt, da der höherpriore auf den Niederprioren wartet.
In der Praxis bedeutet das also, dass keine Aussage über die WCET eines Programmstücks ermittelt werden kann, da diese Verzögerung nicht beschränkt ist. Es kann eine beliebige Zahl von Fäden bereit werden, während Faden A den kritischen Abschnitt abarbeitet. Damit geht also die Echtzeitfähigkeit verloren. Deshalb muss unbestimmte Prioritätenumkehr in Echtzeitsystemen verhindert werden.

3. Algorithmen

Es einige Algorithmen, die nicht-blockierende Synchronisation erlauben. Drei dieser Verfahren sollen hier vorgestellt werden.

3.1 Nicht-blockierende Synchronisation mit CAS

Eine Variante, eine solche Synchronisation zu realisieren, funktioniert mit Hilfe des CAS-Befehls. Dabei existiert ein

Zeiger zur Datenstruktur, auf den alle Fäden (***Thread***) zugreifen können. Will ein Faden auf die Daten schreibend zugreifen, so kopiert er sich die komplette Struktur sowie den Zeiger zu den Daten. Daraufhin arbeitet er auf der Kopie der Daten. Wenn dieser fertig ist, versucht er mit einem CAS-Befehl den Zeiger zu den Daten umzusetzen, sodass dieser auf seine Arbeitskopie zeigt. In der CAS-Instruktion wird geprüft, ob der beim Kopieren gespeicherte Zeigerwert mit dem Aktuellen übereinstimmt. Wenn der Zeiger sich verändert hat, so hat ein anderer Faden die Daten geändert, seitdem die Kopie der Daten erstellt wurde und der gesamte Vorgang muss wiederholt werden.
Das bedeutet also, dass jeder Faden unabhängig auf seinen eigenen Daten arbeitet und lediglich ein Zeiger beim schreibenden Zugriff umgesetzt wird. Beim Umlenken des Zeigers muss allerdings beachtet werden, dass die Berechnung auf aktuellen Daten stattgefunden hat. Dafür wird die CAS-Instruktion verwendet. Hat nämlich ein anderer Faden in der Zeit zwischen Kopieren der Daten und Umsetzen des Zeigers die Daten verändert, so ist die Kopie potentiell veraltet. Daher muss daraufhin der Vorgang des Kopierens, Datenverarbeitens und Umsetzen des Zeigers wiederholt werden. Das kann eintreten, wenn zwei Fäden gleichzeitig versuchen, auf die Daten zuzugreifen.
Das Fehlschlagen des CAS-Befehls kann natürlich mehrfach hintereinander vorkommen, da beim zweiten Durchlauf ebenso ein weiterer Faden die Daten verändern kann.
Wenn nun mehr als zwei Fäden gleichzeitig versuchen, die Daten zu verändern, werden alle bis auf einen scheitern. Damit diese nicht sofort wieder kollidieren, kann eine zufällige Zeit gewartet werden. Dabei eignet sich ein Backoff, das heißt eine Zeitspanne, die dem Zufall unterliegt und mit der Anzahl der Fehlversuche potentiell höher werden kann. Das bedeutet, wenn viele Fäden gleichzeitig versuchen die Daten zu schreiben, wird beim mehrfachen Wiederholen die Wartezeit über eine längere Zeitspanne verteilt sein. Dadurch wird die Anzahl der Kollisionen vermindert. Dieses Verfahren ist unter anderem auch von dem Medienzugriffsverfahren CSMA/CD her bekannt.
Das bedeutet insgesamt, dass jeder Faden sich eine Kopie der Daten holt, auf diesen arbeitet und sie dann versucht atomar zu schreiben, indem er einen globalen Zeiger umsetzt.
Damit besteht bei diesem Algorithmus die Gefahr, dass ein Faden eine unbekannte Zeit warten muss oder sogar verhungert. Dieses tritt dann ein, wenn der CAS-Befehl bei einem Faden mehrfach scheitert.
Verklemmungen können, wie bei nicht-blockierender Synchronisation üblich, nicht auftreten. Eine Verklemmung ist ausgeschlossen, da der Thread an keiner Stelle blockiert.

3.2 Helping mit Wartestapel

Eine weitere Variante ist das sogenannte Helping mit Wartestapel. Bei dieser Variante wird jedes zu synchronisierende Objekt mit einem Wartestapel (***Wait stack***) ausgestattet, der das Helping (Helfen) realisiert.

Der Algorithmus lässt sich am einfachsten an einem Beispiel verdeutlichen, das auch in Abbildung 2 veranschaulicht wird:

Wenn ein Faden A auf einen kritischen Abschnitt (CS) zugreifen will, der von einem Faden B gehalten wird, so wird der Faden A auf dem Helping Stapel des synchronisierten Objekts gespeichert. Daraufhin übergibt der Faden A seine restliche CPU-Zeit an Thread B, um diesen zu beschleunigen und selbst wieder möglichst schnell fortfahren zu können. Wenn Faden A daraufhin wieder eingelastet wird, prüft er, ob fortgefahren werden kann. Ist das nicht der Fall, so hilft er wieder dem anderen Faden B fertig zu werden, das heißt, Faden A überlässt Faden B seine CPU-Zeit. Außerdem übergibt Faden A seine Priorität an Faden B, falls er eine höhere Priorität besitzt. Somit kann verhindert werden, dass es zu einer unbestimmten Prioritätsumkehr kommt (vgl. [2]). Dieses Verfahren implementiert somit eine Variante von Prioritätsvererbung (***priority inheritance***).
Wenn Faden B die Resource wieder frei gibt, dann übergibt er diese an den Faden, der oben auf dem Stack liegt, was in diesem Fall Thread A wäre. Wenn Faden A das nächste mal eingelastet wird, wird er feststellen, dass der kritische Abschnitt „ihm gehört“ und wird mit der Ausführung des kritischen Abschnitts beginnen.
Der Vorteil des Stacks liegt darin, dass damit sichergestellt werden kann, dass für ein System mit festen Prioritäten der höchstpriore Faden oben auf dem Stack liegt. Das resultiert aus der Tatsache, dass jeder Faden seine Priorität an den momentanen Halter des kritischen Abschnitts übergibt, wenn er von höherer Priorität ist und oben auf den Stack gelegt wird. Damit wird ausgeschlossen, dass ein Faden von

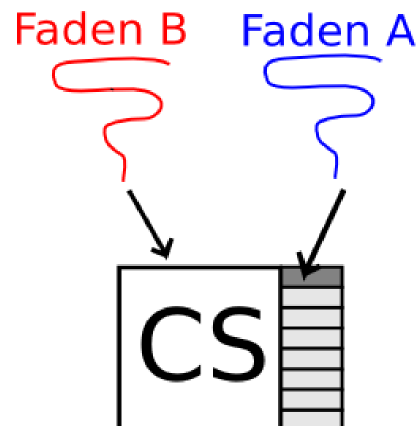

Abbildung 2: Der kritische Abschnitt CS ist von Faden A belegt. Wenn Faden B darauf zugreifen will, so wird dieser auf dem Stapel des kritischen Abschnitts gespeichert.

niedrigerer Priorität als der es obersten Stackobjekts noch lauffähig werden kann. Daher kann auch kein Thread von niedrigerer Priorität noch versuchen, den kritischen Abschnitt zu betreten. Das verhindert somit Prioritätsverletzungen.

Für ein Multiprozessorsystem ergeben sich jedoch weitere Herausforderungen. So muss festgelegt werden, welche CPU beim Helping übergeben werden soll, wenn die beiden Threads auf unterschiedlichen CPUs arbeiten.

Im vorangegangen Beispiel könnte Faden A auf die CPU von Faden B migrieren und gegebenenfalls die Priorität vererben. Dieses Vorgehen wird remote Helping genannt (vgl. [1]).
Es könnte aber auch Faden B zeitweilig auf der CPU von Faden A aufgeführt werden. Dieses Verfahren wird als local Helping bezeichnet (vgl. [1]).
Ein wichtiger Unterschied zwischen den beiden Verfahren wird bei folgendem Szenario deutlich: Faden A auf CPU 1 versucht wiederum auf eine Ressource zuzugreifen, die von Faden B auf CPU 2 gehalten wird. Bei local Helping ist es möglich, dass Faden A Faden B hilft, obwohl Faden B auf seiner CPU von einem anderen Faden C, der eine noch höhere Priorität besitzt, unterbrochen wurde. Bei remote Helping würde es zu keinem Beschleunigen von Faden B kommen. Das wird in Abbildung 3 verdeutlicht.

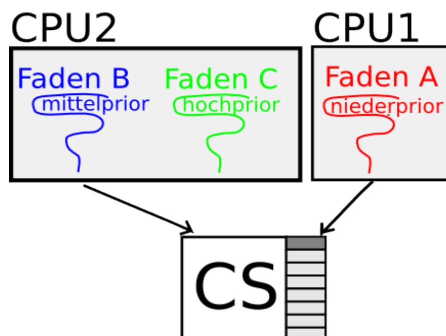

Abbildung 3: Bei local helping kann der niederpriore Faden A dem mittelprioren Faden B auf einer anderen CPU helfen, obwohl dort ein hochpriorer Faden bereit ist

Ein weiterer Vorteil von local Helping liegt darin, dass während des Helpings keine Interprozessorsynchronisation nötig ist, die besonders teuer ist. Außerdem wird lediglich eine normale CPU-lokale Bereitliste benötigt.

Eine andere Fragestellung auf Multiprozessorsystemen ist, die Frage nach dem Verhalten, wenn ein Faden A einem Faden B auf einer anderen CPU hilft, wobei dieser warten muss.
Eine Möglichkeit dabei ist, dass der Faden A einen Interprozessor-Interrupt für eine Callback-Funktion bei der CPU von Faden B registriert. Diese wird nach Beendigung des kritischen Abschnitts oder Beendigung des Faden aufgerufen und weckt Faden A auf. Diese Variante wird als „Sleep and Callback“ bezeichnet (vgl. [1]).
Eine weitere Möglichkeit ist, dass Faden A den Zustand von Faden B periodisch abfragt und auf ein Beenden des kritischen Abschnitts oder des gesamten Fadens wartet. Diese Variante wird als Polling bezeichnet.
Sowohl die Latenz als auch die Laufzeit sind bei der „Sleep and Callback“-Variante höher, da hier z.B. der Interprozessor-Interrupt viel Zeit verbraucht. Jedoch kann auch eine höhere CPU-Auslastung erreicht werden, da nicht immer eine ganze Zeitscheibe bis zum nächsten Nachfragen des Fadenzustands vergeht. Jedoch sind die meisten kritischen Abschnitte in ihrer Ausführung kürzer als eine Zeitscheibe.

Außerdem stellt sich noch die Frage, wie nach einem Helping-Vorgang eine Einplanung (***Scheduling***) der Fäden abläuft. Auch diese Problematik kann an einem Beispiel gut verdeutlicht werden: Wiederrum will der hochpriore Faden A, der auf CPU 1 läuft, auf eine Ressource zugreifen, die der niederpriore Faden B (auf CPU 2) hält. Somit kommt es zum sogenannten Helping. In diesem Fall soll local Helping betrachtet werden. Damit wird also A für die Dauer seines kritischen Abschnitts auf CPU 2 laufen. Im Anschluss muss der Einplaner (***Scheduler***) bestimmen, wie fortgefahren werden soll. Ein simples Umschalten zum neuen Halter der Ressource ist nicht möglich, da es möglich ist, dass dieser bereits auf einer anderen CPU läuft. Auch wird Faden A auf seine ursprüngliche CPU (***Home CPU***) zurückkehren, da auch aus anderen Gründen wie der Verwendung von schnellerem CPU-lokalen Speicher eine statische CPU-Bindung sinnvoll ist. Daher muss also eine neue Ablaufplanung erfolgen und es kann nicht einfach zum nächsten Faden umgeschalten werden.

3.3 Single-Server

Ein weiterer Ansatz ist ein Derivat des sogenannten Single-Server nach [3]. Dabei kann jeder Faden Besitzer verschiedener Objekte sein. Diese Objekte können Daten oder auch Codeabschnitte sein. Code kann generell nur von seinem Besitzer ausgeführt werden.
Den unkritischen Code, also denjenigen, den immer nur ein Faden ausführt, besitzt der Faden immer. Kritische Abschnitte hingegen muss der Faden erst in seinen Besitz holen, bevor er sie ausführen kann.
Ein Objekt hingegen hat immer genau einen Besitzer. Dieser Besitzer kann dann auf dem Objekt arbeiten, bis es ihm von einem anderen Faden „abgenommen“ wird.
Um kritische Abschnitte zu schützen gibt es die Möglichkeit Fäden abzuschließen. Damit kann kein anderer Faden mehr Objekte, die dem verschlossenen Faden gehören, in Besitz nehmen.
Wenn also ein Faden einen kritischen Abschnitt betreten will, so versucht diesen zu besetzen, indem der Faden den Besitzer des Objektes zu ändern versucht. Anschließend verschließt er sich selbst. Damit kann für keines der Objekte, die der Faden aktuell hält, mehr der Besitzer geändert werden. Daher kann auch kein anderer Faden mehr Besitzer des kritischen Abschnitts werden, sodass diesen auch kein anderer Faden mehr betreten kann. Wenn der Besitzer fertig ist, entsperrt er sich selbst wieder und ein andere Fäden können die Besitzverhältnisse wieder ändern.
Wenn das Ändern des Besitzers scheitert, ist der kritische Abschnitt gerade von einem andern Faden belegt. Dann überlässt der anfragende Faden dem aktuellen Besitzer seine restliche CPU-Zeit, um diesen zu beschleunigen. Auch vererbt er ihm seine Priorität, wenn diese höher ist. Es findet also ein Helping statt. Dabei können wiederum die gleichen Überlegungen angestellt werden, wie beim Helping mit Wartestapel.

Der Nachteil dieses Verfahrens liegt darin, dass bei jedem Sperren eines Abschnitts ein Kontextwechsel stattfinden muss. Damit sind unter anderem alle Caches geleert, was sich merklich auf die Performanz des Systems auswirkt.
Nach [2] ist ein Zähler, der auf den Algorithmus Helping

mit Wartestapel aufbaut, circa 2,5 mal so schnell wie einer, der mit dem Single-Server Derivat arbeitet.

4. Echtzeiteigenschaften der Verfahren

In Bezug auf Echtzeitsysteme ergeben sich besondere Eigenschaften, die erfüllt sein müssen. So darf es zu keinen Prioritätsverletzungen kommen. Dies wäre zum Beispiel dann der Fall, wenn ein niederpriorer Faden vor einem höherprioren Faden eingeplant wird, obwohl es keinen zwingenden Grund wie Datenabhängigkeiten oder ähnliches dafür gibt.

In einem Mehrprozessorsystem müssen auch Prioritätsverletzungen über CPU-Grenzen hinweg vermieden werden. Sonst könnte ein mittelpriorer Faden auf einer CPU ausgeführt werden und einen hochprioren Faden auf einer anderen CPU ausbremsen. Dies geschieht, wenn ein hochpriorer Faden A auf CPU 1 auf eine Ressource zugreifen will, die momentan von einem niederprioren Faden B auf CPU 2 gehalten wird. Dann könnte ein mittelpriorer Faden C auf CPU 2 eingelastet werden, der Faden B verdrängt und folglich auch Faden A ausbremst. Somit ist Prioritätsvererbung auch über CPU-Grenzen hinweg für die Einhaltung von Echtzeiteigenschaften nötig. Dies wird in Abbildung 4 verdeutlicht.

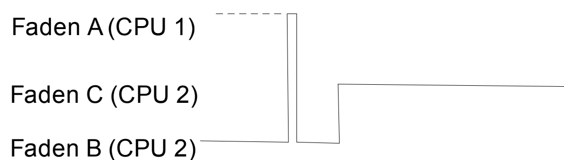

Abbildung 4: Der mittelpriore Faden C auf CPU 2 verdrängt den niederprioren Faden B auf der selben CPU und verzögert damit den hochprioren Faden A auf einer anderen CPU, der vom niederprioren Faden abhängt. Somit kommt es zu unbestimmter Prioritätenumkehr auch über CPU-Grenzen hinweg

Eine besondere Herausforderung stellen dabei Synchronisationspunkte dar. Hier kann es zum Verhungern eines Fadens oder zu Prioritätsumkehr kommen. Um konkret die Echtzeitfähigkeit der drei vorgestellten Synchronisationsverfahren zu untersuchen, werden diese drei Techniken miteinander verglichen.

Eine zentrale Eigenschaft von Echtzeitsystemen muss die Analysierbarkeit sein. Dazu gehört, dass das Verhungern eines Fadens ausgeschlossen werden muss. Denn könnte ein Faden verhungern, so würde er alle seine Terminschranken (***Deadlines***) verpassen. Diese Eigenschaft kann für die Synchronisation mittels CAS nicht garantiert werden. Es ist nämlich möglich, dass bei einem Faden die CAS-Operation jedes mal fehlschlägt, weil immer ein anderer Faden bereits ein Update auf der Datenstruktur durchgeführt hat. In diesem Fall würde der betroffene Prozess verhungern. Zwar kann mit Hilfe eines Backoffs in der Wartezeit nach der CAS-Instruktion praktisch ein Verhungern vermieden werden, jedoch kann diese Eigenschaft weder nachgewiesen werden noch analysiert werden. Auch wird die Wartezeit des Fadens damit nicht-deterministisch.
Ein Verhungern kann beim Helping nach [1] bzw. [2] ebenso nicht ausgeschlossen werden. Es kann zwar für ein Einprozessorsystem gezeigt werden (vgl. [6]), jedoch gab es bisher keinen Beweis für ein Multiprozessorsystem. Ebenso kann auch kein Beweis für ein Single-Server-System aufgestellt werden.
In der Realität jedoch können Aussagen für den Single-Server und das Helping mit Wartestapel getroffen werden. Da alle Fäden, die potentiell auf einen kritischen Abschnitt zugreifen wollen, bekannt sind, kann mit Hilfe ihrer Periode und der maximalen Ausführungszeit des kritischen Abschnitts Aussagen getroffen werden.

Des weiteren darf es in einem Echtzeitsystem keine unbestimmte Prioritätsumkehr geben. Dies würde bedeuten, dass ein niederpriorer Faden vor einem höherprioren Faden eingeplant wird, obwohl es keine inhaltliche Kopplung wie Datenabhängigkeiten zwischen beiden gibt. Bei der Variante mit CAS-Befehl kann es zu einer solchen unbestimmten Prioritätsumkehr kommen. Dies wäre beispielsweise der Fall, wenn ein niederpriorer Faden A auf einer CPU 1 mit dem Update der Daten vor einem anderen, hochprioren Faden B auf einer anderen CPU 2 fertig wird. Auf dieser läuft noch ein mittelpriorer Faden C. Damit bremst Faden A Faden C aus.
Eine Prioritätsumkehr kann für Helping mit Stack ausgeschlossen werden, da immer der höchstpriore Faden „oben“ auf dem Stack liegt und der wartende Faden auch seine Priorität gegebenenfalls abgibt.
Auch beim Single-Server Verfahren kann diese unbestimmte Prioritätenumkehr ausgeschlossen werden, da ein Helping wie beim Algorithmus mit Wartestapel stattfindet.

Insgesamt lässt sich damit also feststellen, dass sowohl der Single-Server Algorithmus als auch das Helping mittels Stack für Echtzeitsysteme geeignet sind, solange sie auf Einprozessormaschinen laufen. In Mehrprozessorsystemen ist der Nachweis der Echtzeitfähigkeit bisher nicht möglich. Die Variante mit Hilfe der CAS-Instruktion ist jedoch nicht geeignet, da sie nicht deterministisch in der Wartezeit arbeitet und auch keine unbestimmte Prioritätenumkehr ausschließt.

5. Multiprozessor-Kompatibilität

Durch ein Mehrprozessorsystem ergeben sich einige neue Herausforderungen und Fragestellungen. So kann festgestellt werden, das vor allem Interprozessorkommunikation sehr zeitintensiv ist. Das folgt aus der Tatsache, dass dafür ein Interprozessorinterrupt benötigt wird und dadurch unter anderem auch Caches invalide werden können.
Daher sind auch CPU-lokale Daten besonders schnell. Dies wird durch die Tatsache verstärkt, dass einige Architekturen eine Speicherzuordnung kennen. Dies bedeutet also, dass jedem Prozessor ein Teil des Speichers favorisiert zugeordnet ist und er damit schneller auf diesen zugreifen kann. Datenstrukturen, die häufig zugegriffen werden, wie zum Beispiel Bereit-Listen, sind dafür besonders interessant.

Daneben sorgt laut [1] eine statische CPU-Bindung für höhere Performanz. Auch hier würden wiederum Caches invalide werden. Außerdem müssen dann auch wieder Interprozessor-Interrupts eingesetzt werden.

Eine weitere Herausforderung liegt darin, dass für Einprozessorsysteme auch Aussagen herangezogen werden können, die das Scheduling berücksichtigt. So kann zum Beispiel eine Oberschranke für die Anzahl der Wiederholungen beim Zugriff auf eine Datenstruktur ermittelt werden, wenn es ein zeitgetriebenes Einplanungsverfahren ist, da hierbei die Anzahl der höherprioren Fäden und deren Periode bekannt ist. Solche Beweise sind jedoch für Mehrprozessorsysteme nicht ohne weiteres möglich, da hier echt Parallelität herrscht (vgl. [6]).

Die Variante des Helping mit Wartestapel ist multiprozessorfähig (vgl. [1]), jedoch geht dabei zumindest die harte Echtzeitfähigkeit verloren. Gerade mit Hilfe des Local Helpings entsteht eine echter Vorteil gegenüber Einprozessorsystemen: Ein wartender Faden kann quasi seine CPU übergeben und somit einem anderen Faden auf der anderen CPU beschleunigen (vergleiche Kapitel 4.2). Das ermöglicht die Beschleunigung von hochprioren Fäden und stellt auch die Prioritätsreihenfolge zwischen den Prozessoren sicher.
Die andere echtzeitfähige Variante mit Single-Server scheint auch für Mehrkernsysteme geeignet, wenngleich in [3] diese Herausforderung nicht ausführlich diskutiert wird. Auch hier geht zumindest die harte Echtzeitfähigkeit verloren. Für das Helping gelten die selben Regeln wie für die Variante mit Helping Stack. Dabei aber auch wiederum beachtet werden, dass das Sperren auch über CPU-Grenzen hinweg atomar geschieht.
Die Alternative mit CAS-Instruktion ist auch für Mehrkernsysteme geeignet. Aber auch hier muss die Hardware eine atomare Ausführung der CAS-Instruktion über CPU-Grenzen hinweg garantieren.

6. Fazit

Insgesamt lässt sich resümieren, dass nicht blockierende Synchronisation für Echtzeitsysteme eine besondere Herausforderung darstellt. Hierbei kann der aus der PC-Welt bekannte Ansatz mittels eines CAS-Befehls eine atomare Änderung der Daten zu erzielen, nicht verwendet werden, da hier Unvorhersagbarkeiten durch Wiederholen der Aktion eintreten können, wenn kein zusätzliches Wissen vorhanden ist.
Daher müssen dafür andere Varianten verwendet werden. Dabei gibt es die Möglichkeit, das mit Hilfe eines sogenannten „Single-Servers“ zu realisieren. Dabei werden kritische Abschnitte einem Faden zugeordnet und für jeden solchen Abschnitt darf es immer nur einen Besitzer geben, wodurch Sequentialisierung erreicht wird. Beim Eintritt in den kritischen Bereich, wird dann der Besitzerfaden gelockt, wodurch kein anderer Faden in diese Sektion eintreten kann. Um Echtzeitfähigkeit zu erreichen, erfolgt ein Helfen eines Faden (***Helping***), das zur Vererbung der Fadenpriorität und Übergabe der CPU-Zeit von wartenden an gelockte, ausführende Fäden führt. Somit kann eine Prioritätsverletzung wie beispielsweise unbestimmte Prioritätsumkehr verhindert werden.
Eine andere Variante ist das Helping mit Wartestapel. Dabei wird beim Eintritt in einen kritischen Abschnitt dieser gelockt. Wenn ein anderer Faden in einen solchen eintreten will, obwohl dieser bereits gelockt ist, wird er verzögert und auf einem Stack gespeichert. Außerdem findet ein Helping statt. Auch dadurch kann die Prioritätsreihenfolge sichergestellt werden.
Jedoch ist weder die Variante des Single-Servers noch die des Helping mit Wartestapel ein echtes Nicht-blockierende Synchronisationsverfahren. Diese sind nicht-blockierend implementierte Verfahren, die jedoch intern blockierend arbeiten.

Weiter sind die beiden Verfahren für den Einprozessorfall echtzeitfähig. Ebenso sind beide für Multiprozessorsysteme geeignet, jedoch kann dabei die Echtzeitfähigkeit nicht mehr eingehalten werden. Der Vorteil des Helping stack gegenüber dem Single-Server liegt in dessen besserer Performanz.

In Zukunft wird jedoch noch einige Forschungsarbeit auf diesem Gebiet nötig sein, um Verfahren zu finden und weiter zu analysieren, die Echtzeitfähigkeit in Multiprozessorsystemen auf der Basis nicht-blockierender Synchronisation zulassen. Die Parallelität schafft einige neue Herausforderungen, die gerade mit steigender Prozessoranzahl verstärkt werden.

LITERATUR

[1] M. Hohmuth, M. Peter, „Helping in a multiprocessor environment“, Proceedings of the Second Workshop on Microkernel-based Systems, 2001

[2] M. Hohmuth, H. Härtig. Pragmatic nonblocking synchronization for real-time systems, in USENIX Annual Technical Conference, Boston, MA, June 2001

[3] Henry Massalin and Calton Pu, „A lock-free multiprocessor OS kernel“, Technical Report CUCS-005-91, Columbia University, 1991.

[4] H. Baumgartl, „Nichtblockierende Synchronisation“, Chemnitz 2008

[5] Florian Schricker, „Nicht-blockierende Synchronisation für Echtzeitsysteme“, Koblenz 2005

[6] James H. Anderson, Srikanth Ramamurthy und Rohit Jain, Real-Time Computing with Lock-Free Shared Objects, ACM Transactions of Computer Systems, 1997