Wartefreie Synchronisation

Martha Willhaug
Friedrich-Alexander-Universität Erlangen-Nürnberg

Martha.Willhaug@informatik.uni-erlangen.de

ABSTRACT
Heutzutage sind Mehrkernprozessorkerne immer verbreiteter und der Trend geht dahin, dass sich nicht mehr der Takt der CPUs verdoppeln wird, sondern dass die Anzahl der Prozessorkerne zunimmt. Das Problem der Synchronisation gewinnt dadurch immer mehr an Bedeutung, denn symmetrische Mehrkernprozessoren arbeiten meistens auf gemeinsamen Speicher. Dieses Problem ist bereits von Uniprozessorsystemen bekannt, wenn mehrere Prozesse auf gemeinsamen Daten arbeiten.

Die Motivation von wartefreier Synchronisation ist die Gewährleistung, dass in jedem Schritt die Prozesse einen Fortschritt machen. Dadurch werden typische Probleme anderer Synchronisationsparadigmen vermieden.

1. Einleitung
Dieses Papier beschreibt wie Probleme, in Echtzeitsystemen, konventioneller Synchronisationsparadigmen mit Hilfe von wartefreier Sycnhronisation umgangen werden können. Weiterhin beschreibt es welche Objekte sich, unter bestimmten Voraussetzungen, für wartefreie Synchronisation eignen und stellt zum Schluss einen Algorithmus vor, der jedes Objekt in ein wartefreies Objekt transformiert.
Bei der Synchronisation konkurrenter Prozesse können verschiedene Probleme auftreten, wie zum Beispiel die Verhungerung (*starvation*), Verklemmungen (*deadlocks/ livelocks*) und Prioritätsumkehr. Die wartefreie Synchronisation ist eine nicht-blockierende Synchronisation, die zusätzlichen Kriterien genügen muss. Gerade in Echtzeitsystemen, muss gewährleistet werden, dass Zeitvorgaben (*deadlines*) eingehalten werden. Dies könnte mittels wartefreier Synchronisation realisiert werden, da hier garantiert wird, dass jeder Prozess in einer endlichen Anzahl an Schritten fertig wird und somit die Überschreitung der Zeitvorgabe durch blockierende Prozesse verhindert werden kann.

1.1 Blockierende Verfahren
Wenn verschiedene Prozesse auf gemeinsamen Daten arbeiten, ist es unumgänglich diese zu synchronisieren. Der naive Ansatz ist dabei blockierende Verfahren einzusetzen. Dies kann beispielsweise durch Sequentialisierung von kritischen Abschnitten erfolgen. Jedoch können dabei Probleme auftreten. So möchte beispielsweise ein Prozess eine Operation auf einer gemeinsamen Datenstruktur ausführen. Also betritt er den kritischen Abschnitt, der durch ein bestimmtes Eintrittsprotokoll geregelt wird (Semaphoren, etc.) [2][3]. Der Prozess prüft erst, ob der Abschnitt frei ist, falls ja, sperrt er diesen und versucht seine Operationen auszuführen. Dabei kann im ersten Fall durch Verletzung von Reihenfolgenrestriktionen beim Akquirieren und Freigeben von Schlossvariablen zu Verklemmungen kommen. Dabei wartet jeder andere Prozess auf die Freigabe, die nicht eintritt und somit kann keiner seine Operation ausführen. Das bekannte Philosophenproblem [2] führt zur Verhungerung von Prozessen. Dabei brauchen zum Beispiel zwei Prozesse zwei Betriebsmittel um ihre Operation auszuführen. Der erste Prozess reserviert sich das erste Betriebsmittel und der zweite Prozess reserviert sich das zweite Betriebsmittel. Falls das Verfahren blockierend ist, warten also beide Prozesse auf die Freigabe des benötigten Betriebsmittels. Das Resultat ist, dass beide Prozesse als Folge der Verklemmung verhungern. Bei diesen blockierenden Verfahren ist es nicht möglich Zeitvorgaben einzuhalten.

1.2 Nicht-blockierende Verfahren
Nicht-blockierende Verfahren sind ein nächster Ansatzpunkt, die vor allem Verklemmungen vermeiden. Bei nicht-blockierender Synchronisation gibt es solche die behinderungsfrei, lockfrei oder wartefrei sind [4]. Behinderungsfreiheit (*obstruction-freedom*) garantiert, dass letztendlich ein einzelner Prozess seine Operation abschließen kann, falls er „in Isolation“ ausgeführt wird [10]. Damit ist das Problem der Verklemmung gelöst, aber es kann zur Verhungerung kommen, wenn sich zwei Prozesse jeweils gegenseitig in ihrem Fortschritt beeinflussen. Auch dieses Verfahren ist nicht für Echtzeitsysteme geeignet, da hier nicht garantiert werden kann, dass die Zeitvorgaben eingehalten werden können. Sperr-/Lock-Freiheit (*lock-freedom*) erlaubt, dass bestimmte Prozesse verhungern, garantiert aber den globalen Fortschritt des Systems. Falls zum Beispiel der Prozess A den Prozess B verdrängt, so kann der Prozess B den Prozess A nicht mehr verdrängen. Prozess B könnte in diesem Fall verhungern, denn er kann Prozess A nicht unterbrechen, obwohl er für seine Operation beispielsweise das Betriebsmittel von A benötigen würde.

1.3 Ausblick
Der nächste Abschnitt gibt eine Definition für wartefreie Synchronisation an. In Abschnitt 2.1 werden Unmöglichkeits-resultate vorgestellt, die vor allem nötig sind um zu zeigen, ob eine wartefreie Implementierung möglich ist und am Schluss, in Abschnitt 2.2, ist ein Algorithmus aufgezeigt, mit Hilfe dessen jedes Objekt als wartefreies Objekt implementiert werden kann.

2. Definition

Definition 1: Bei wartefreier Synchronisation wird gewährleistet, dass jeder Prozess jede Operation in endlich vielen Schritten beendet, unabhängig von der Ausführungsgeschwindigkeit anderer Prozesse [1].

Definition 2: Eine wartefreie Implementierung eines Objekts ist k-bedingt wartefrei, falls ein k > 0 existiert und jede Operation in k Schritten ihre Operation beenden kann [8].

Aus Definition 1 folgen verschiedene Eigenschaften. Kein Prozess kann von anderen Prozessen blockiert werden, die ein bestimmtes gemeinsames Betriebsmittel nutzen oder deren unterschiedlichen Ausführungsgeschwindigkeiten, davon abgehalten werden, die Operation zu beenden. Definition 2 gibt an wie viele Schritte genau, nämlich k Schritte, bis zur Beendigung der Operation benötigt werden. Damit sind die Echtzeitgarantien bei wartefreier Synchronisation gegeben und Szenarien wie in Abschnitt 2.1 sind ausgeschlossen.

2.1 Unmöglichkeitsresultate

2.1.1Konsensusnummer

Definition 3: Die Konsensusnummer für ein Objekt X ist die größte Zahl n, für die X den n-Prozess-Konsensus (Übereinstimmung) löst. Falls kein größtes n existiert, ist die Konsensusnummer unendlich [1].

Dabei ist die Idee, dass jedes Objekt X eine Konsensusnummer (*consensus number*) bekommt, nach der dann bestimmt wird, wie viele Prozesse jeweils damit wartefrei implementiert werden können. Herlihy sieht in [1] sowohl die Queue und den Stack als auch den CAS-Befehl als ein Objekt an, mit dem ein Konsensus, mit der entsprechenden Anzahl an Prozessen, gebildet werden kann.
In Tabelle 1 ist eine Tabelle dargestellt, in der jeweils die Konsensusnummer mit dem zugehörigen Objekt dargestellt ist.

Consensus Number	Object
1	read/write registers
2	test&set, swap, fetch&add, queue, stack
⋮	⋮
2n-2	n-register assignment
⋮	⋮
∞	memory-to-memory, move and swap, augmented queue, compare&swap, sticky byte

Tabelle 1 : Objekte mit zugehöriger Konsensusnummer [1]

Die Aussage „das Objekt X löst den n-Prozess-Konsensus“ bedeutet, dass X *n* Prozesse verwalten kann, die auf einer Menge von gemeinsamen Daten arbeiten und untereinander kommunizieren. Näheres dazu in Abschnitt 2.1.2.

Die Definition hat zur Folge, dass wenn ein Objekt Y ein Objekt X implementiert und X den n-Prozess Konsensus löst, dann löst auch Y den n-Prozess-Konsensus. Mit anderen Worten: Ein Objekt mit höherer Konsensusnummer kann nicht aus einem Objekt mit niedrigerer Konsensusnummer nachgebildet werden. Andersrum ist es jedoch möglich, ein Objekt mit einer niedrigeren Konsensusnummer aus einem Objekt mit einer höheren Konsensusnummer darzustellen.

Es existiert nur dann eine wartefreie Implementierung von n Prozessen, wenn das Objekt die Konsensusnummer größer oder gleich n besitzt. Falls ein Objekt die Konsensusnummer n hat, so handelt es sich in diesem Fall um ein wartefreies Objekt für n Prozesse. Somit wird durch ein wartefreies Objekt garantiert, dass jeder Prozess seine Operation in einer endlichen Anzahl an Schritten beenden kann.

2.1.2 Konsensus-Protokoll

Informell ist ein Konsensus-Protokoll ein System aus n Prozessen, die auf einer Menge von gemeinsamen Daten arbeiten. Dabei startet jeder Prozess mit einem Eingabewert von einer Domäne D. Die Prozesse kommunizieren miteinander, indem sie Operation auf den gemeinsam Daten ausführen. Anschließend einigen sie sich auf einen gemeinsamen Eingabewert und halten. Ein solches Protokoll muss folgende Eigenschaften erfüllen [1]:

(1) konsistent: Alle Prozesse, die an der Konsensusbildung beteiligt sind, einigen sich auf einen gleichen Eingabewert
(2) wartefrei: Jeder Prozess entscheidet den Konsensus nach einer endlichen Anzahl an Schritten
(3) valid: Der Wert der gemeinsamen Entscheidung ist der Eingabewert für einen anderen Prozess

Dabei kann der Zustand des Protokolls mehrere Werte annehmen. Der Zustand ist zweiwertig (*bivalent*), falls die Entscheidung über einen gemeinsamen Wert noch nicht gefallen ist, ansonsten ist sie einwertig (*univalent*). Der Zustand ist x-wertig (*x-valent*), falls die Entscheidung den Wert x hat. Der Endzustand wird dadurch bestimmt, dass der Wert von dem Prozess zurückgeliefert wird, der als erstes seine Operation ausführen konnte.

Dieses Protokoll ist ein theoretisches Hilfsmittel und ist nötig, um die verschiedenen Konsensusnummern der Objekte zu beweisen. Weiterhin ist es wichtig, dass alle Prozesse nachvollziehen können, welche Operationen in welcher Reihenfolge auf dem Objekt ausgeführt wurden.
Mit dem Konsensus-Protokoll kann nun beweisen werden, warum atomare Lese-/Schreibregister die Konsensusnummer 1 haben.

2.1.3Atomare Lese-/Schreibregister

Wie bereits erwähnt und wie auch aus Tabelle 1 ersichtlich ist, hat das atomare Lese-/Schreibregister die Konsensusnummer 1. Daraus folgt, dass zwei Prozesse keinen Konsensus über das atomare Lese-/Schreibregister bilden können. Im Konsensus-Protokoll gibt es keinen zweiwertigen Zustand. Dies folgt aus der Konsensusnummer, denn die Entscheidung über den Wert trifft nur ein Prozess. Der weitere Abschnitt befasst sich mit der Konsensusbildung über ein atomares Lese-/Schreibregister.

Angenommen die beiden Prozesse P und Q arbeiten auf gemeinsamen Daten und beide führen nach Beendigung ihrer Operationen jeweils in einen einwertigen Zustand x bzw. y, so kann folgender Widerspruchsbeweis hergeleitet werden.

Nun liest P aus dem Register und Q schreibt in das Register. Falls nun P zuerst liest, so gibt er den Zustand x zurück, daraufhin schreibt Q in das gleiche Register und gibt auch den Zustand x zurück, da P zuerst seine Operation ausgeführt hat und diese als Eingabewert für Q dient. Falls jedoch P nicht aus dem Register liest und Q trotzdem in das Register schreibt, gibt es dann den Zustand y zurück. Dies ist ein Widerspruch, da in bei den Fällen Q seine Operationen komplett ausführen kann, jedoch verschiedene Zustände zurückgibt und damit eine eindeutige Konsensusbildung nicht möglich war. Dies ist in Abbildung 1 verdeutlicht.

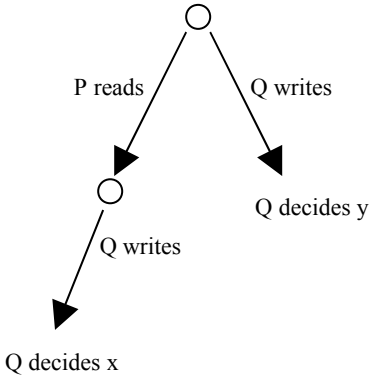

Abbildung 1: P liest und Q schreibt in das Register (vgl. [1])

Angenommen P und Q schreiben in verschiedene Register. Falls nun zuerst P in das eine Register 1 schreibt (also x-wertig) und danach Q in das Register 2 schreibt (also y-wertig), so ist das das gleiche Szenario, als wenn zuerst Q in das Register 2 und P danach in das Register 1 schreibt. In beiden Fällen gibt P den Zustand x und Q den Zustand y zurück, allerdings ist dies wieder ein Widerspruch, da in beiden Fällen das Endresultat zwar das gleiche ist, allerdings geben P und Q jeweils verschiedene Zustände zurück. Das Ganze kann auch so betrachtet werden, dass es für jeden Prozess ein eigenes Register gibt, also löst hier jedes einzelne Register den 1-Prozess-Konsensus, denn P und Q entscheiden nicht über die Daten, da diese disjunkt sind vgl. Abbildung 2.

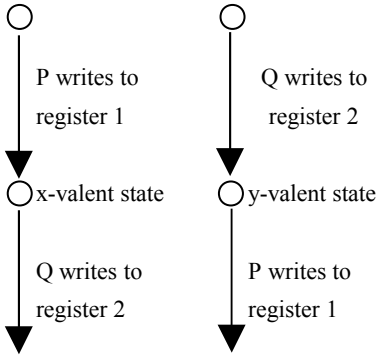

Abbildung 2: P und Q schreiben in verschieden Register (vgl. [1])

Es sei nun angenommen, dass P und Q in das gleiche Register schreiben. Falls zuerst P in das Register schreibt, gibt es danach den Zustand x zurück. Falls nun aber zuerst Q in das Register schreibt und danach schreibt P in das Register so gibt P den Zustand y zurück. Dies ist ein Widerspruch, da P jedes Mal seine Operation vollständig ausführen kann, jedoch jeweils verschiedene Zustände zurückliefert (vgl. Abbildung 3).

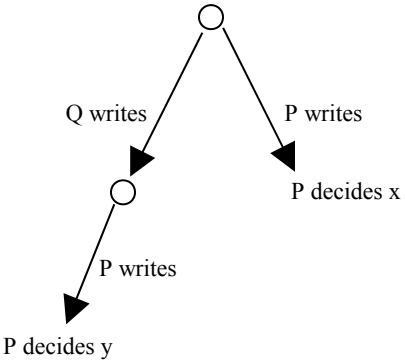

Abbildung 3: P und Q schreiben in das gleiche Register (vgl. [1])

Aufgrund dieser Erläuterungen wird nun klar, warum mit atomaren Lese-/Schreibregistern keine „mächtigeren“ Objekte nachgebildet werden können. Wie in Abbildung 2, sind hier zwei verschiedene Register benutzt worden, mit zwei Prozessen, die jeweils disjunkt in diese geschrieben haben. Angenommen, dass diese zwei Prozesse beliebig in diese zwei Register schreiben, so werden diese nicht nur jeweils einen anderen Zustand zurückgeben, sondern werden auch noch ständig ihre Ergebnisse in den Registern überschreiben.
Ähnliches kann bei Datenbanken beobachtet werden. Falls die einzelnen Lese- und Schreibzugriffe auf disjunkten Tabellen arbeiten, geht alles gut, falls jedoch diese Lese- und Schreibzugriffe auf gleichen Tabellen arbeiten, muss ein Mechanismus gefunden werden. Dieser Mechanismus muss sicherstellen, dass zum Beispiel nicht gleichzeitig gelesen wird, obwohl ein anderer Zugriff in diesem Moment den Inhalt verändert. Er muss aber auch verhindern, dass nicht von zwei Transaktionen gleichzeitig in die gleiche Tabelle geschrieben wird, denn nach so einem Zugriff ist nicht klar, was in der Tabelle steht.

Wird nun noch zusätzlich erlaubt, dass die Register zum Lesen und Schreiben auch noch Modifizieren können, wie dies zum Beispiel bei test&set (TAS), compare&swap (CAS), fetch&add (FAA) und weiteren der Fall ist, so ist das Resultat, dass diese Operationen mindestens Konsensusnummer 2 haben[1].

CAS-Register haben Konsensusnummer unendlich. Dies hat zwar zur Folge, dass ein CAS-Objekt den „unendlich-Prozess“-Konsensus löst, jedoch sei angemerkt, dass es trotzdem noch keine Garantie für die Konsistenz bzw. den problemlosen Einsatz in der Praxis garantiert. Das bekannte Problem ist das ABA-Problem [6].

2.1.4 FIFO-Queue

Im vorhergehenden Abschnitt wurde gezeigt, dass atomare Lese-/Schreibregister Konsensusnummer 1 haben, d.h. also dass sie in der Hierarchie ganz unten anzusiedeln sind, wie es aber mit den Queues aussieht, werde ich nun im Folgenden erläutern. Hierbei geht es um die Spezifikation einer FIFO-Queue, die nur 2

Prozesse definiert. Dabei wird die FIFO-Queue als Objekt betrachtet, mit der ein Konsensus gebildet werden kann.

Im Folgenden geht es um die Konsensusbildung über eine FIFO-Queue mit 3 Prozessen. Dies wird zum Widerspruch geführt, indem der dritte Prozess nicht nachvollziehen kann, was auf der Queue passiert ist, dies ist aber wichtig für eine konsistente Konsensusbildung.

Angenommen es gibt 3 Prozesse P, Q und R. P und Q machen dabei einen Entscheidungsschritt, wobei P das Protokoll zu einem x-wertigen Zustand führt und Q zu einem y-wertigen.

Im ersten Fall führen nun P und Q eine dequeue-Operation aus. Dabei arbeiten alle 3 Prozesse auf der gleichen Queue. Falls nun P zuerst seine dequeue-Operation ausführt und danach Q, so einigen sich beide auf den Entscheidungswert x (im Protokoll wird also der Zustand x angenommen und dieser an R weitergegeben). Falls nun aber zuerst Q seine dequeue-Operation ausführen kann und danach P, so führt dies zwar zum gleichen Ergebnis, allerdings ist das Protokoll im Zustand y und dieser dient dann als Eingabewert für R. Der Widerspruch ist, dass in beiden Fällen das Endresultat auf der Queue das gleiche ist, allerdings werden verschiedene Zustände im Protokoll erreicht. Mit anderen Worten kann das Szenario auch folgendermaßen erklärt werden. Die zwei Prozesse P und Q kommunizieren untereinander und einigen sich auf einen gemeinsamen Entscheidungswert. In diesem Fall wissen sie also in welcher Reihenfolge die zwei dequeue-Operationen stattgefunden haben. Für R ist dies nicht ersichtlich, denn er bekommt jedes Mal einen anderen Eingabewert. Ausgehend von diesem Werten kann R nicht rekonstruieren was auf der Queue passiert ist. Auf Grund des Eingabewertes x könnte es nämlich auch sein, dass P nur eine enqueue-Operation ausgeführt hat. Dieses Szenario wird in Abbildung 4 verdeutlicht.

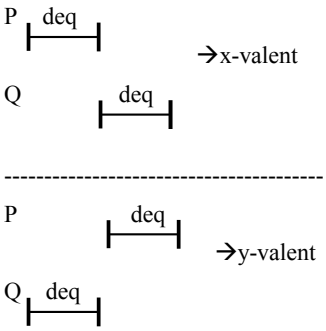

Abbildung 4: P und Q führen dequeue-Operationen aus

Im zweiten Fall führt P eine enqueue-Operation und Q eine dequeue-Operation aus. Falls die Liste nicht leer ist, so macht es keinen Unterschied, ob zuerst P seine enqueue-Operation ausführt und danach Q seine dequeue-Operation oder umgekehrt, da beide Prozesse untereinander kommunizieren. Allerdings kann in diesem Fall R nicht unterscheiden, in welcher Reihenfolge die Elemente eingefügt bzw. entfernt wurden bzw. was überhaupt auf der Queue passiert ist, denn wie bereits erwähnt geben x und y als Eingabewert für R keinen Aufschluss, welche Operationen ausgeführt wurden. Falls die Liste nun leer ist und Q versucht, ein Element zu entfernen, so führt dies zwar im ersten Moment zu einem Fehler, aber P kann danach trotzdem ein Element einfügen. Danach einigen sich die beiden Prozesse auf einen y-wertigen Zustand. Falls jedoch P nur ein Element einfügt, ohne dass vorher Q auf der Queue gearbeitet hat, so ist das Protokoll im x-wertigen Zustand. Da beide Prozesse miteinander kommunizieren, kann Q ableiten, wenn P x zurückliefert, dass P eine enqueue-Operation ausgeführt hat. Beide Fälle führen zwar auf der Queue zum gleichen Ergebnis, nämlich dass die Queue nur das Element enthält, welches P eingefügt hat. Allerdings gibt es für R zwei verschiedene Eingabewerte (vgl. Abbildung 5).

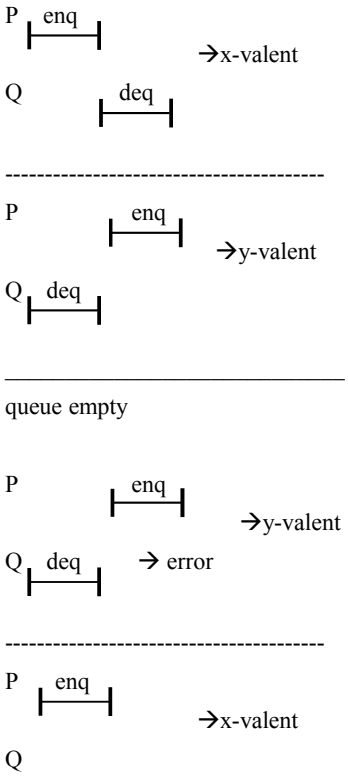

Abbildung 5: P führt enqueue-Operation und Q führt dequeue-Operation aus

Zuletzt fügen sowohl P als auch Q ein Element in die Queue ein. Zuerst fügt P das Element p in die Queue ein und danach Q das Element q. P läuft nun solange, bis es p wieder von der Queue entfernt. P kann keinen Entscheidungswert annehmen, bevor es nicht entweder p oder q in der Queue entdeckt. Somit laufen also die beiden Prozesse bis P das Element p von der Liste entfernt und danach Q das Element q. Der Zustand, der danach angenommen wird, ist x-wertig, weil P zuerst seine enqueue-Operation ausführt und P und Q sich auf den gemeinsamen Entscheidungswert x einigen, nachdem auch Q seine enqueue-Operation ausgeführt hat. Umgekehrt nimmt auch wieder P zuerst p aus der Queue und danach Q. Analog ist folgendes Szenario: Zuerst fügt Q das Element q, dann P das Element p in die Queue ein. P läuft nun solange, bis es das von Q eingefügte Element q wieder entfernt und daraufhin entfernt Q das von P eingefügte Element p. In diesem Fall führt dies zu einem y-wertigen Zustand. Wieder einmal ist die Queue nach beiden Operation der Prozesse P und Q im gleichen Zustand, allerdings wird im ersten Fall x und

im zweiten Fall y als Eingabewert für R zurückgegeben. Hier findet also R wieder die gleiche Queue vor, allerdings kann er nicht entscheiden, was genau auf der Queue passiert ist und in welcher Reihenfolge dies erfolgte. Dies ist in Abbildung 6 veranschaulicht.

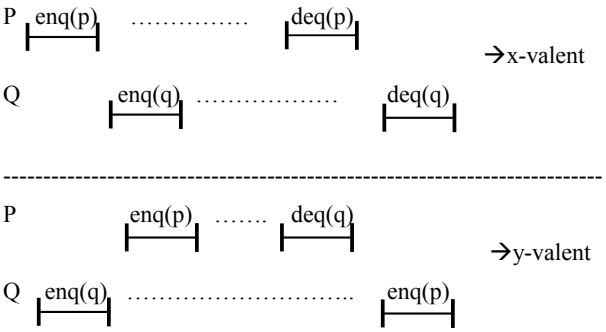

Abbildung 6: P und Q fügen Element in die Queue ein

Hiermit ist nun bewiesen, dass die FIFO-Queue ein wartefreies Objekt für zwei Prozesse darstellt. Es existiert also eine wartefreie Implementierung von zwei Prozessen, die als Objekt die FIFO-Queue verwenden.

2.2 Ein universelles wartefreies Konstrukt

Im Abschnitt 2.3 war das Hauptaugenmerk darauf gelegt, zu zeigen, warum ein Objekt eine bestimmte Konsensusnummer hat. Daraus folgte auch, dass ein Objekt mit Konsensusnummer n ein wartefreies Objekt für n Prozesse ist. Dies bedeutet, dass hieraus eine wartefreie Implementierung der n Prozesse folgt.

In diesem Abschnitt geht es darum, mit einem Algorithmus aus jedem sequentiellen Objekt ein wartefreies Objekt zu machen.

Mit Hilfe des Algorithmus wird auch das Problem der unbegrenzten Prioritätsumkehr vermieden [5]. Ein Prozess H mit einer hohen Priorität erkennt, dass das benötigte Betriebsmittel von einem niedrig-priorisiertem Prozess L benutz wird. Also verdrängt er L, fordert das Betriebsmittel an und blockiert. L setzt seine Ausführung fort. Dann kommt ein Prozess M mit einer mittleren Priorität an, verdrängt L und beendet danach seine Ausführung. Danach setzt L seine Ausführung fort, da H immer noch blockiert. Anschließend bekommt H das Betriebsmittel von L, verletzt seinen Termin, weil zwischendrin M gelaufen ist, wird dann fertig mit seiner Ausführung und gibt das Betriebsmittel wieder an L zurück, der dann noch zu Ende läuft. Obwohl H die höchste Priorität von den drei Prozessen hat, verletzt er seine Zeitvorgabe und der mittel-priorisierte Prozesse läuft durch, obwohl dessen Zeitvorgabe viel später gewesen wäre als die von H. Lösungsansätze für diese Problem sind beispielsweise Prioritätsvererbung (priority inheritence) [5]. Wartefreie Synchronisation umgeht das Problem. Ein Prozess X führt eine bestimmte Operation aus und erkennt dabei, dass er auf ein Betriebsmittel zugreifen muss, das gerade von einem niedriger-priorisiertem Prozess Y benutzt wird. Der Prozess X fügt sich hier dem Helfer-Stack des Betriebsmittels zu und gibt damit seine CPU-Zeit dem Prozess Y ab. Dadurch wird vorübergehen die Priorität von Y erhöht und X kann schnellstmöglich selber das gemeinsame Betriebsmittel nutzen.

Um den Pseudo-Code für den wartefreien Algorithmus zu verstehen, müssen nun dafür zwei Funktionen für die Prozesse definiert werden [1].

(1) `Announce` ist ein n-elementiges Feld, dessen i-tes Element ein Zeiger auf die Speicherzelle i ist. Im Initialzustand zeigen alle Elemente auf eine festgelegte Speicherzelle.
(2) `Head` ist ein n-elementiges Feld, dessen i-tes Element ein Zeiger auf die letzte Speicherzelle, die von i überwacht (angeschaut) wurde.

Ein Objekt wird dabei durch eine doppelt-verkettete Liste dargestellt. Desweiteren sind noch folgende Felder nötig.

(1) `Seq` ist die Speicherzellennummer in der Liste. Im Initialzustand ist diese Nummer 0, falls der Thread noch nicht in die Liste eingelastet wurde, ansonsten ist sie positiv. Sequenznummern einer erfolgreichen Speicherzelle werden um eins inkrementiert.
(2) `Inv` ist invocation (Operationsname und Wert des Arguments)
(3) `New` ist ein Konsensusobjekt, dessen Wert das Paar (new.state, new.result) ist. Dabei ist der erste Parameter der Zustand, inklusive Operation, des Objekts und der zweite ist der Ergebniswert.
(4) `Before` ist ein Zeiger zur vorhergehenden Speicherzelle in der Liste. Dieses Feld ist nur zur freien Speicherverwaltung.
(5) `After` ist ein Konsensusobjekt, dessen Wert ein Zeiger auf die nächste Speicherzelle in der Liste ist.

Dabei arbeitet das Protokoll folgendermaßen. Zuerst allokiert und initialisiert P eine Speicherzelle, um die Operation zu repräsentieren In Zeile 2 ist dabei die Sequenznummer 0, da der Prozess P noch nicht in die Liste eingelastet wurde. In `inv` (Zeile 3), kann in diesem Fall etwas stehen, da es nur die Operation repräsentiert. Zeile 4 und 5 erzeugen ein Konsensus-Objekt und `after` ist in diesem Fall noch null, da P im Moment der einzige Prozess ist. In Zeile 8 setzt P einen Zeiger auf die Speicherzelle, um zu gewährleisten, dass falls nicht seine Speicherzelle in die Liste eingelastet wird, so wenigstens die eines anderen. Nun wird für jeden weiteren Prozess Q durchiteriert und das Maximum aus `head[P]` und `head[Q]` ermittelt (Zeilen 9-12). P setzt damit seinen `head`-Zeiger auf die Speicherzelle in der Liste, die möglichst am Ende ist. Solange P noch nicht in die Liste eingelastet wurde, also die Sequenznummer 0 hat (Zeile 12), so wird die Variable c (vom Typ *cell) auf `head[P]` gesetzt. In Zeile 14 entscheidet sich P einem anderen Prozess zu helfen, dafür schaut er nach, ob der andere Prozess eine Speicherzelle hat, die noch nicht eingelastet wurde. Falls dies der Fall ist so versucht, den anderen Prozess einzulasten (Zeile 16), ansonsten sich selber (Zeile 17). P versucht `head[P].after` (=c) auf die Speicherzelle zu setzen, die er versucht einzulasten (Zeile 19). Dabei muss sichergestellt werden, dass das `after`-Feld eine Konsensus-Speicherzelle ist, welche garantiert, dass nicht mehrere Prozesse gleichzeitig darauf zugreifen. Somit kann nur ein Prozess dieses Feld setzen. Mit Zeile 20 initialisiert P die verbliebenen Felder der nächsten Speicherzelle. Dies muss ein Konsensus-Objekt sein, da sonst verschiedene Prozesse versuchen würden, das `new`-Feld zu setzen und das Ganze damit nicht-determiniert. Da die Variable d, die Speicherzelle enthält, die P versucht hat einzulasten, wird dessen `before`-Zeiger auf die von c gesetzt und danach dessen Sequenznummer addiert (Zeile 21, 22). P versucht also einem Prozess zu helfen in die Liste eingelastet zu werden, falls dies passiert, wird dessen Sequenznummer in Zeile 22 addiert. Andernfalls hat Prozess P

versucht, sich selber in die Liste einzulasten und damit wird seine Sequenznummer inkrementiert. P macht also solange weiter, und versucht sich selber in die Liste einzulasten, bis er das schafft und seine Sequenznummer erhöht wird. Aus der Definition geht hervor, dass die Sequenznummer ungleich 0 ist, falls der Prozess eingelastet wurde und dies wird in dem Algorithmus auch realisiert.

```
1  universal(what: INVOC) returns(RESULT)
2   mine: cell:=[seq: 0,
3      inv: what,
4      new: create(consensus_object),
6      before:create(consensus_object),
7      after: null]
8      ⟨announce[P]:= mine;
              start(P):= max(head)⟩
9   for each process Q do
10     head[P] := max(head[P], head[Q])
11  end for
12  while announce[P].seq = 0 do
13     c: *cell := head[P]
14     help:*cell:=announce[(c.seq mod n)+1]
15     if help.seq = 0
16     then prefer := help
17     else prefer := announce[P]
18     end if
19     d := decide(c.after, prefer)
20     decide(c.new,apply(d.inv,c.new.state))
21     d.before := c
22     d.seq := c.seq + 1
23  ⟨head[P]:=d;
       (∀Q)concur(Q):=concur(Q)∪{d}⟩
24  end while
25     ⟨head[P]:=announce[P];
              (∀Q)concur(Q):=concur(Q)∪{d}⟩
26     return (announce[P].new.result)
27 end universal
```

Abbildung 7: Pseudo-Code des wartefreien Algorithmus[1]

`Concur(P)` und `Start(P)` sind in den Pseudo-Code Hilfsvariablen [1].

(1) `concur(P)` ist die Menge an Speicherzellen, dessen Speicheradressen in dem `head`-Feld, seit dem letzten announce-Aufruf von P, gespeichert wurden.

(2) `start(P)` ist der Wert von `max(head)`.

Dabei sei zu beachten, dass $|concur(P)| + start(P) = max(head)$.

Damit lassen sich nun auch die Zeilen 23 und 25 erklären.

In Zeile 23 steht `head[P]` auf d, da d in Zeile 19 auf die Speicherzelle gesetzt wurde, die eingelastet wurde. Und für alle anderen Prozesse Q gilt, dass die Menge der Speicherzellen, die seit der letzten durchlaufenen Schleife um d erweitert wurde.

Nach der Schleife gilt, dass nun d in die Menge, der von P angeschauten Speicherzellen, hinzugefügt wurde.

An dieser Stelle sei noch angemerkt, dass das Protokoll aus Abbildung 14 korrekt und bedingt wartefrei ist.

In diesem Fall ist Linearisierbarkeit [7] gegeben, denn die Reihenfolge in der die Speicherzellen eingelastet werden mit der partiellen Ordnung der Operationen übereinstimmt.

Das Protokoll ist bedingt wartefrei, da P die while-Schleife maximal (n+1)-mal durchlaufen kann, denn nach jedem Durchlauf wird die Sequenznummer von P dekrementiert (denn P schaut sich maximal n anderen Prozesse an).

3. Fazit

Der Vorteil von wartefreier Synchronisation ist, dass sie für Echtzeitsysteme geeignet ist. Denn die anderen nicht-blockierenden Verfahren lassen keine Aussage über die Ausführungszeit zu. Bei wartefreier Synchronisation ist die WCET bekannt, da jede Operation in endlich vielen Schritten zur Ausführung kommt. Weiterhin vermeidet wartefreie Synchronisation typische Probleme anderer Synchronisations-paradigmen.

Der Nachteil von wartefreier Synchronisation ist, dass der Speicher gekonnt verwaltet werden muss, denn für jeden Prozess muss eine Speicherzelle angelegt werden und der Prozess P braucht mindestens $O(n^2)$ Leseoperationen[1]. Desweiteren ist auch nicht klar, ob ein Objekt aus der einen Ebene aus einem anderen Objekt aus der gleichen Ebene nachgebildet werden kann [9].

4. Referenzen

[1] M. Herlihy. Wait-Free Synchronization. ACM Transactions on Programming Language and Systems, 13(1), pp. 124–149, 1991

[2] Andrew S. Tanenbaum. Operating Systems: Desgin and Implementation. Prentice Hall, 1987

[3] J. Archer Harris. Moderne Betriebssysteme. mitp, 2007

[4] Dinesh P. Mehta, Sartaj Sahni. Handbook of data structures and applications. Chapman & Hall/CRC, 2004

[5] Bernhard Freisleben. Mechanismen zur Synchronisation paralleler Prozesse. Springer-Verlag, 1987

[6] Damian Dechev, Peter Pirkelbauer, Bjarne Stroustrup. Lock-free Dynamically Resizable Arrays. Texas A&M University. Springer-Verlag, 2006

[7] M. P. Herlihy and J.Wing. „Linearizability: A correctness condition for concurrent objects. ACM Transactions on Programming Language and Systems, 12(3), pp. 463–492, 1990

[8] James Aspnes and Maurice Herlihy. „Wait-free data structures in the asynchronous PRAM model". Second Annual ACM Symposium on Parallel Algorithms and Architectures, July 1990, pp. 340–349.

[9] Matei David. „A Single-Enqueuer Wait-Free Queue Implementation". Department of Computer Science, University of Toronto. DISC 2004, Springer-Verlag, pp.132–143, 2004

[10] Maurice Herlihy,Victor Luchangco, Mark Moir. Obstruction-Free Synchronization: Double-Ended Queues as an Example. CDCS 03, pp. 522–529

1. Wird die äußere Form der Arbeit dem Anspruch an wissenschaftliche Texte gerecht?

2. Ist das Dokument klar strukturiert und ein durchgängiger roter Faden zu erkennen?

3. Sind die einzelnen Gliederungspunkte klar und verständlich dargelegt?

4. Werden komplexe Zusammenhänge genügend klar veranschaulicht und auf möglichst einfache Art und Weise dargestellt?

5. Wie ist der Gesamteindruck? Hast du verstanden, was der Verfasser dir vermitteln will?