

Systemprogrammierung

Grundlage von Betriebssystemen

Teil C – X.4 Prozesssynchronisation: Kreiseln und Spezialbefehle

Wolfgang Schröder-Preikschat

26. November 2015



Agenda

Einführung

Umlaufsperr

Definition

Funktionsweise

Schlossalgorithmen

Diskussion

Transaktion

Motivation

Prinzip

Beispiele

Diskussion

Zusammenfassung



Gliederung

Einführung

Umlaufsperr

Definition

Funktionsweise

Schlossalgorithmen

Diskussion

Transaktion

Motivation

Prinzip

Beispiele

Diskussion

Zusammenfassung



- Konzepte der **Befehlssatzebene** (s. [10]) kennenlernen, womit die Synchronisation gleichzeitiger Prozesse erreicht wird
 - **Umlaufsperr**en, d.h., Sperren für mehr-/vielkernige Multiprozessoren
 - sperrfreie Synchronisation mittels (Mikro-) **Transaktionen**
- für Umlaufsperr
- en typische **Schlossalgorithmen** behandeln und ihre Auswirkungen auf andere Prozessen untersuchen
 - grundsätzliche wie auch spezielle Schwachstellen thematisieren
 - schrittweise Techniken für Verbesserungen entwickeln und erklären
- als Antwort zu Unzulänglichkeiten von Schlossalgorithmen, im Ansatz die **nichtblockierende Synchronisation** vorstellen
 - dazu einfache, musterhaftige **transaktionale Programme** diskutieren
 - sie als **nebenläufige Abschnitte** mit kritischen Abschnitten vergleichen
- die Bedeutung der **Spezialbefehle** und der diesbezüglichen Rolle von Schleifenkonstruktionen für beide Konzepte erfassen
 - sehen, wie TAS, CAS und FAA genutzt wird und implementiert ist
 - Gemeinsamkeiten und Unterschiede bei Anwendung der Befehle erkennen



Gliederung

Einführung

Umlaufsperr

Definition

Funktionsweise

Schlossalgorithmen

Diskussion

Transaktion

Motivation

Prinzip

Beispiele

Diskussion

Zusammenfassung



Kein „Drängelgitter“, das Passierende, durch Blickwendung im Umlauf weiter induziert, zur Vorsicht aufruft.



- diesem aber nicht ganz unähnlich:
 - Passierende $\equiv$ Prozesse
 - Blickwendung $\equiv$ Zustandsabfrage
 - Umlauf $\equiv$ Schleife
- durch **Kreiseln** (*spin*) das **Sperren** (*lock*) von Prozessen steuern:
 - acquire** ■ verzögert den aktuellen Prozess, solange die Sperre gesetzt ist
 - **aktives Warten** (*busy waiting*) lässt den Prozess kreiseln
 - verhängt die Sperre, sobald sie aufgehoben wurde
 - release** ■ hebt die Sperre auf, ohne den aktuellen Prozess zu verzögern
 - alias *lock* (sperren) und *unlock* (entsperren)
- die Prozesssteuerung manifestiert sich in Verfahren, die in Form von sogenannten **Schlossalgorithmen** (*lock algorithms*) umgesetzt sind

¹vgl. auch <https://de.wikipedia.org/wiki/Umlaufgitter> (28/10/15).



Die **Umlaufsperr**e dient der Synchronisation gleichzeitiger (gekoppelter) Prozesse **verschiedener Prozessoren** oder Prozessorkerne eines Rechensystems mit gemeinsamem Speicher.

Synchronisation solcher Prozesse **desselben Prozessors** erfordert gegebenenfalls zusätzlich eine **unilaterale Sperr**e zur Vorbeugung unvohersehbarer Latenz,² vorzugsweise eine Unterbrechungssperre.

²Wegen Unterbrechung des die Sperr



- in einfachster Form bildet eine **binäre Schlossvariable** die Grundlage z.B. entsprechend folgenden Datentyps:

```
1  #include <stdbool.h>
2
3  typedef volatile struct lock {
4      bool busy;          /* initial: false */
5  } lock_t;
```

- auf einem Exemplar dieses Datentyps operieren die Primitiven einer Umlaufsperr dem **Prinzip** nach wie folgt:

```
6  void acquire(lock_t *lock){
7      while (lock->busy); /* spin until lock release */
8      lock->busy = true;  /* claim lock-out */
9  }
10
11 void release(lock_t *lock){
12     lock->busy = false; /* abandon lock-out */
13 }
```

- das wäre zu schön, um wahr zu sein: **wettlaufkritische Aktionsfolge** ☹



Problemanalyse

■ Ausgangssituation:

- **gleichzeitige Prozesse**
- die in `acquire` geschehen

```
1 void acquire(lock_t *lock){  
2     while (lock->busy);  
3     lock->busy = true;  
4 }
```

■ wettkritische Aktionsfolge:

- 1 ■ $n > 1$ Prozesse rufen gleichzeitig `acquire` auf, betreten den Rumpf
- 2 ■ sie werten gleichzeitig den Zustand der Schlossvariablen (`busy`) aus
 - alle stellen gleichzeitig die Aufhebung der Wartebedingung/Sperre fest
 - als Folge verlassen alle gleichzeitig die kopfgesteuerte Schleife
- 3 ■ alle Prozesse verhängen gleichzeitig die (soeben aufgehobene) Sperre
- 4 ■ sie verlassen gleichzeitig `acquire`, betreten den kritischen Abschnitt
→ wechselseitiger Ausschluss scheitert: $n > 1$ Prozesse fahren fort ☹

■ Lösungsansatz:

- die Aktionsfolge „Sperre prüfen und ggf. verhängen“ atomar auslegen
- Atomarität ist in dem Fall nur mit Mitteln der Befehlssatzebene erreichbar
 - unilaterale Sperren [15, S. 26–29] scheiden aus: sie wirken nur lokal auf dem Prozessor, wofür Umlaufsperren eben nicht vorgesehen sind (S. 7)
 - stattdessen sind in Hardware implementierte **Spezialbefehle** erforderlich



Sperre prüfen und verhängen — atomar

- testen, das Ergebnis vermerken, und setzen: **test and set** [7, p.144]

```
1 bool TAS(bool *ref) {  
2     atomic { bool aux = *ref; *ref = true; }  
3     return aux;  
4 }
```

- diese Operation wirkt immer schreibend auf den Arbeitsspeicher aber die Werteveränderung der adressierten Variablen erfolgt nur bedingt
 - nämlich nur, wenn die Variable den Wert 0 bzw. false enthielt
 - jedoch wird unbedingt der Wert 1 bzw. true in die Variable geschrieben
 - das Operationsergebnis ist der Variablenwert vor dem Überschreiben
- bei Operationsausführung durch die Hardware erfolgt **wechselseitiger Ausschluss** gleichzeitiger Speicherbuszugriffe der Prozessoren
 - bewirkt wird ein **atomarer Lese-/Schreibzyklus** in zwei Dimensionen
 - i unilaterale Sperre asynchroner Programmunterbrechungen³
 - ii multilaterale Sperre gleichzeitiger Buszugriffe „von außen kommend“

³Normal: heute lässt ein Prozessor Programmunterbrechungen bei sich erst am Ende des im Moment der Unterbrechungsanforderung interpretierten Befehls zu.



- Reduktion auf eine atomare, in GCC eingebaute intrinsische Funktion:

```
1 #define TAS(ref) __sync_lock_test_and_set(ref, 1)
```

- Anwendung dieser Funktion für den Schlossalgorithmus:

```
2 void acquire(lock_t *lock) {  
3     while (TAS(&lock->busy));    /* spin until claimed */  
4 }
```

- Kompilierung in Assemblersprache (ASM86, AT&T Syntax):

```
5 _acquire:  
6     movl    4(%esp), %eax # get pointer to lock variable  
7 LBB0_1:    # come here to retry (while)...  
8     movb    $1, %cl      # want to change lock to "true"  
9     xchgb   %cl, (%eax)   # atomically swap operand values  
10    testb   $1, %cl      # check former lock value  
11    je      LBB0_1       # if it equals "true", retry  
12    ret                     # was "false" and is "true", now
```

- die relevante atomare Operation findet sich in Zeile 9: xchgb [8]



Kreiseln mit TAS

```
1 void acquire(lock_t *lock) {  
2     while (TAS(&lock->busy));    /* spin until claimed */  
3 }
```

- naive Lösung mit schädlicher Wirkung auf **Pufferspeicher** (*cache*)
 - unbedingtes Schreiben bei **Wettstreit** löst massiven Datentransfer aus:
 - i $n - 1$ Kopien invalidieren und 1 Original zum auslösenden Prozessor bewegen
oder
 - ii 1 Original schreiben und $n - 1$ Kopien bei anderen Prozessoren aktualisieren
 - die Pufferspeicherzeile (*cache line*) mit der Schlossvariablen „flattert“
- hinzu kommt schädliche Wirkung auf kausal unabhängige Prozesse
 - nahezu anhaltender wechselseitiger Ausschluss von Speicherbuszugriffen
 - jede Ausführung von TAS sperrt der Bus für andere Prozessoren
 - dazwischen liegen nur wenige (z.B. drei, vgl. S. 11) normale Befehle
 - blockt Prozessoren, wenn Prozessdaten nicht im Pufferspeicher vorliegen
 - erzeugt **Störung** (*interference*) in Prozessen anderer Prozessoren
- in nichtfunktionaler Hinsicht skaliert die Lösung ziemlich schlecht
 - kreiseln mit bedingtes Schreiben, mit ablesen oder mit Zurückhaltung...



Kreiseln mit CAS

```
1 #define CAS __sync_bool_compare_and_swap
2 void acquire(lock_t *lock) {
3     while (!CAS(&lock->busy, false, true));
4 }
```

- wobei Funktionssignatur **CAS(Variable, Prüfwert, Neuwert)** einen atomaren Spezialbefehl wie folgt definiert beschreibt:

$$\text{CAS} = \begin{cases} \text{true} \rightarrow \text{Neuwert zugewiesen,} & \text{falls Variable} = \text{Prüfwert} \\ \text{false,} & \text{sonst} \end{cases}$$

- der Befehl schreibt nur, wenn die **Gleichheitsbedingung** erfüllt ist
- die schädliche Wirkung auf den Pufferspeicher bleibt aus, nicht aber auf kausal unabhängige Prozesse
 - nahezu anhaltender wechselseitiger Ausschluss von Speicherbuszugriffen
 - ungünstiges Verhältnis zur Anzahl normaler Befehle (1:3, vgl. S. 38)
- in nichtfunktionaler Hinsicht skaliert die Lösung schlecht
 - kreiseln mit ablesen oder mit Zurückhaltung...



```
1 void acquire(lock_t *lock) {  
2     do {  
3         while (lock->busy);  
4     } while (!CAS(&lock->busy, false, true));  
5 }
```

- schwächt Wettstreit beim Buszugriff und damit Interferenz ab
 - 3 ■ die eigentliche Warteschleife, fragt nur den Pufferspeicher ab
 - keine Datenbuszugriffe, kausal unabhängige Prozesse bleiben ungestört
 - 4 ■ die Sperre wird verhängt, wenn sie immer noch aufgehoben ist⁴
 - betrifft gekoppelte (gleichzeitige) Prozesse stark, andere jedoch kaum
- steht und fällt allerdings mit der Länge des kritischen Abschnitts
 - ist er zu kurz, degeneriert die Lösung zum Kreiseln mit CAS
- bei großem Wettstreit stauen sich nach wie vor viele Prozesse (Z. 4)
 - sich wiederholende **Häufung der Bussperre** (*bus lock burst*)
 - Prozessen anderer Prozessoren werden stoßartig Buszugriffe verwehrt
- in nichtfunktionaler Hinsicht skaliert die Lösung mehr oder weniger
 - kreiseln mit Zurückhaltung: **Stauauflösung**, Abstand/Lücken schaffen...

⁴Beachte, dass der kreiselnde Prozess überholt worden sein kann.



Definition (*backoff*)

Statische oder dynamische **Verweilzeit**, prozessorweise abgestuft, bis zur Wiederaufnahme der vormals wettstreitigen Aktion.

```
1 void acquire(lock_t *lock) {
2     do {
3         while (lock->busy); /* spin on read */
4         if (CAS(&lock->busy, false, true))
5             return;          /* lock acquired, done */
6         backoff(lock->time, earmark());
7     } while (true);          /* contention faced, retry */
8 }
```

- angenommen sei eine *sperrenspezifische Verweilzeit* (time)

earmark ■ liefert die Nummer des ausführenden Prozessor(kern)s

- der Telekommunikation entlehnter Ansatz zur **Blockierungskontrolle** (*congestion control*) bei **Kanalüberzeichnung**:

- statische (ALOHA [1]) oder dynamische (Ethernet [14]) Verzögerungen

- **Stauauflösung** (*contention resolution*), ausgeübt zum Sendezeitpunkt



Verweilzeit bestimmen und absitzen

- ein Parameter, der von der **Restlaufzeit** des im kritischen Abschnitt operierenden Prozesses und vom **Wettstreitgrad** abhängt
 - bestenfalls kann dies nur ein gut abgeschätzter **Näherungswert** sein

- minimale **Datentypenerweiterung** (vgl. S. 8) für Sperrexemplare:

```
1 typedef volatile struct lock {
2     bool busy;    /* initial: false */
3     long time;    /* duration of critical section */
4 } lock_t;
```

- time ist Zeitwert des günstigsten, mittleren oder schlechtesten Falls
 - *best-, mean- oder worst-case execution time* (BCET, MCET bzw. WCET)
- durch **statische Programmanalyse** des betreffenden kritischen Bereichs

- im Vergleich zur Zeitanalyse ist der Zeitverbrauch nahezu einfach:

```
1 void backoff(long time, int rate) {
2     volatile long term = time * rate;
3     while (term--);    /* just spend processor cycles */
4 }
```

- eine **Schlafsperr**e (*sleeping lock*) gäbe hier den Prozessor frei
- die Unkosten dafür sollten aber in die **effektive Verweilzeit** einfließen ☹



- alle bisher diskutierten Verfahren können Prozessen eine **nach oben unbegrenzte Wartezeit** bescheren
 - die Verfahren sichern einem System gekoppelter Prozesse Fortschritt zu — wenn Verklemmungen einmal außer Acht gelassen werden
 - jedoch können einzelne Prozesse ewig im Eintrittsprotokoll kreiseln
- Verhungering einzelner Prozesse ist unbedingt vorzubeugen
- die Wartezeit muss für jeden Prozess begrenzt sein
 - eine obere Schranke ist notwendig
- jedoch darf für jeden Prozess die **effektive Wartezeit** bis zur oberen Schranke variabel sein
- das meint Verfahren, die (a) fair für die Prozesse und (b) auch noch frei von Interferenz mit dem Planer sind
 - (a) ist durchaus einfach (s. umseitig), verträgt sich aber selten mit (b)



- das **Maß der angestauten Prozesse** bestimmt den Wettstreitgrad, der im Moment der Sperrverhängung gilt $\leadsto$ Wettstreiter zählen
- Ideengeber ist der sog. Bäckereialgorithmus [12]: ***ticket spin lock***

```
1 typedef volatile struct lock {
2     long next;    /* number being served next */
3     long this;    /* number being currently served */
4     long time;    /* duration of critical section */
5 } lock_t;

6 void acquire(lock_t *lock) {
7     long self = FAA(&lock->next, 1);    /* my number served */
8     if (self != lock->this) {            /* wait one's turn */
9         backoff(lock->time, self - lock->this);
10        while (self < lock->this);
11    }
12 }
13
14 void release(lock_t *lock) { lock->this += 1; } /* next one */
```

#define FAA __sync_fetch_and_add /* atomic */

Der Wert *self* – *this* gibt die Anzahl der Prozesse, die den kritischen Abschnitt zuerst durchlaufen werden.

- ein der mittels **Wartemarkenspender** sowie **Personenaufrufanlage** realisierten Kundenverkehrssteuerung entlehnter Ansatz ☺



- **Sperren** wirken einseitig (unilateral: Unterbrechungs-, Fortsetzungs-, Verdrängungssperre) oder **mehrseitig** (multilateral: Umlaufsperre)
 - nur die Umlaufsperre wirkt auf die tatsächlich gekoppelten Prozesse
- Umlaufsperren und **verdrängende Prozesseinplanung** integriert im selben Bezugssystem vertragen sich nicht ohne weiteres
 - Prozessorentzug des Schlosshalters (*lock holder preemption*) ist möglich
 - führt auch bei kleinsten kritischen Abschnitten zu hohem Leistungsverlust
 - Stau gekoppelter Prozesse verlängert sich, unbestimmte Verweilzeit
 - jeder Art von Verzögerung des Schlosshalters muss vorgebeugt werden
 - Konsequenz ist, zusätzlich eine **Unterbrechungssperre** zu verhängen
- **wechselseitiger Ausschluss** ist kein Allheilmittel, um Aktionsfolgen mit wettlaufkritischen Eigenschaften abzusichern
 - **arbeitsloses Kreiseln** für den wartenden Prozess und gleichzeitig damit **Störung** anderer Prozesse, die mit ihm denselben Prozessor teilen
 - **Verklemmungsgefahr** (*deadly embrace* [5, S.73]) gekoppelter Prozesse
- ein Manko, dass blockierende Synchronisation grundsätzlich betrifft, jedoch mit Umlaufsperren besonders zum Vorschein kommt
 - Alternative ist die **nichtblockierende Synchronisation: Transaktion**



Gliederung

Einführung

Umlaufsperr

Definition

Funktionsweise

Schlossalgorithmen

Diskussion

Transaktion

Motivation

Prinzip

Beispiele

Diskussion

Zusammenfassung



Nachteile blockierender Synchronisation

- Probleme des in **Software** erzwungenen wechselseitigen Ausschlusses gekoppelter Prozesse durch Monitore, Semaphore oder Sperren

Leistung (*performance*) paralleler Systeme bricht ein/nimmt ab

- Kreiseln vor Sperren reduziert Busbandbreite [3]
- höherer Anteil sequentieller Programmbereiche [2]

Robustheit (*robustness*) „single point of failure“

- im kritischen Abschnitt „abstürzen“ lässt diesen gesperrt
- schlimmstenfalls wird das ganze System lahmgelegt

Interferenz (*interference*) mit dem Planer

- Planungsentscheidungen werden nicht durchgesetzt
- **Prioritätsverletzung**, **Prioritätsumkehr** [13]
 - Mars Pathfinder [16, 9]

Lebendigkeit (*liveness*) einiger oder sogar aller Prozesse

- Gefahr von **Verhungerung** (*starvation*)
- inherent anfällig für **Verklemmung** (*deadlock*)

- etwas anderes ist wechselseitiger Ausschluss in der **Hardware**, insb. bei der Ausführung von Spezialbefehlen (TAS, CAS, FAA)



Pessimistischer vs. optimistischer Ansatz

- **softwaregesteuerter wechselseitiger Ausschluss** trifft eine negative Erwartung in Bezug auf gleichzeitige Prozesse
 - es wird die wettlaufkritische Aktionsfolge geben: **pessimistischer Ansatz**
 - um den Konflikten vorzubeugen, wird frühzeitig die Reißleine gezogen
 - da die Prozesse wahrscheinlich nur für kurze Zeit blockieren werden ☹️
- demgegenüber stehen Paradigmen, die eine positive Erwartung treffen und gleichzeitige Prozesse in Software nicht ausschließen
 - die wettlaufkritische Aktionsfolge gibt es nicht: **optimistischer Ansatz**
 - falls doch, sind Konflikte nachträglich erkenn- und behandelbar 😊
 - dazu wird **hardwaregesteuerter wechselseitiger Ausschluss** benutzt
- hierzu greift letzteres auf Konzepte zurück, die ihren Ursprung in der Programmierung von Datenbanksystemen finden

Definition (Optimistic Concurrency Control [11])

Method of coordination for the purpose of updating shared data by mainly relying on **transaction backup** as control mechanisms.



Definition (Transaktion, nach [6, S. 624])

Eine **Konsistenzinheit**, die Aktionsfolgen eines Prozesses gruppiert.

- die Aktionsfolge ist nicht atomar, jedoch wird das berechnete Datum einer gemeinsamen Variablen nur bei **Isolation** übernommen
 - d.h., wenn diese Folge **zeitlich** isoliert von sich selbst stattfindet
 - wozu sie **ablaufinvariant** für gekoppelte Prozesse formuliert sein muss
- eine solche Aktionsfolge wird i.A. durch eine **fußgesteuerte Schleife** umfasst, in der gleichzeitige Prozesse stattfinden können

```
1  ERLEDIGE Transaktion:
2      WIEDERHOLE
3          erstelle die lokale Kopie des Datums an einer globalen Adresse;
4          verwende diese Kopie, um ein neues Datum zu berechnen;
5          versuche, das neue Datum an der globalen Adresse zu bestätigen;
6      SOLANGE die Bestätigung gescheitert ist;
7  BASTA.
```

- zur Bestätigung (Z. 5) kommt ein **Spezialbefehl** der CPU zum Einsatz
- nur für den gilt **hardwaregesteuerter wechselseitiger Ausschluss**



- **atomare Bestätigungsaktion** einer Transaktion (S. 23, Z. 5):

```
1 atomic bool CAS(type *ref, type old, type new) {  
2     return (*ref == old) ? (*ref = new, true) : false;  
3 }
```

true ■ Bestätigung gelang, neues Datum geschrieben

false ■ Bestätigung scheiterte, referenzierte Variable ist unverändert

- Reduktion auf eine atomare, in GCC eingebaute intrinsische Funktion:

```
1 #define CAS __sync_bool_compare_and_swap
```

■ zur Kompilierung o.g. Funktion nach Assemblersprache, siehe S. 40

- typisches Muster (*pattern*) einer fußgesteuerten (CAS) Transaktion:

```
1 do /* transaction */ {  
2     any_t old = *ref;          /* make local copy */  
3     any_t new = handle(old);   /* compute some value */  
4 } while (!CAS(ref, old, new)); /* try to commit */
```

■ alle Aktionen im **Schleifenrumpf** können durch gleichzeitige Prozesse geschehen, sie unterliegen nicht dem wechselseitigen Ausschluss

■ nur die **Fußsteuerung** der Schleife mit dem CAS läuft synchronisiert ab



Atomare multiplikative Variablenänderung

■ Fassung als **kritischer Abschnitt** zum Vergleich:

```
1 long mult(long_t *ref, long val) {
2     long new;
3
4     enter(&ref->bolt);          /* lock critical section */
5     new = ref->data * val;      /* perform computation */
6     leave(&ref->bolt);         /* unlock critical section */
7
8     return new;
9 }
```

```
10 typedef struct longlock {
11     long data;
12     detent_t bolt;
13 } long_t;
```

■ semantisch äquivalente Fassung als **nebenläufiger Abschnitt**:

```
14 long mult(long *ref, long val) {
15     long new, old;
16
17     do old = *ref;              /* make copy, compute & commit */
18     while (!CAS(ref, old, new = old * val));
19
20     return new;
21 }
```

- funktional ist die Multiplikation zu leisten, die ungesperrt stattfindet
- nur die Bestätigung des Ergebnisses unterliegt wechselseitigem Ausschluss



- einfach verkettete Liste, Verarbeitung nach LIFO (*last in, first out*):

```
1 typedef struct chain {          4 typedef struct chainlock {
2     struct chain *link;         5     chain_t *item;
3 } chain_t;                     6     detent_t bolt;
                                7 } chainlock_t;
```

- Einfügeoperation als **kritischer Abschnitt** zum Vergleich:

```
8 void push(chainlock_t *head, chain_t *item) {
9     enter(&head->bolt);          /* lock critical section */
10    item->link = head->item.link; /* prepend item */
11    head->item.link = item;       /* adjust head pointer */
12    leave(&head->bolt);           /* unlock critical section */
13 }
```

- semantisch äquivalente Fassung als **nebenläufiger Abschnitt**:

```
14 void push(chain_t *head, chain_t *item) {
15     do item->link = head->link;   /* prepend item & commit */
16     while (!CAS(&head->link, item->link, item));
17 }
```

- funktional ist das Voranstellen und die Kopfzeigeraktualisierung zu leisten
- nur letztere Aktion unterliegt dem wechselseitigen Ausschluss



- Entnahmeoperation als **kritischer Abschnitt** zum Vergleich:

```
1 chain_t *pull(chainlock_t *head) {
2     chain_t *item;
3
4     enter(&head->bolt);      /* lock critical section */
5     if ((item = head->item.link) != 0)
6         head->item.link = item->link;
7     leave(&head->bolt);      /* unlock critical section */
8
9     return item;
10 }
```

- semantisch äquivalente Fassung als **nebenläufiger Abschnitt**:

```
11 chain_t *pull(chain_t *head) {
12     chain_t *item;
13
14     do if ((item = head->link) == 0) break;
15     while (!CAS(&head->link, item, item->link));
16
17     return item;
18 }
```

- funktional ist das Entfernen und die Kopfzeigeraktualisierung zu leisten
- nur letztere Aktion unterliegt dem wechselseitigen Ausschluss



Definition (ABA, auch A-B-A)

*The ABA problem is a **false positive** execution of a CAS-based speculation on a shared location L_i . [4, p. 186]*

- CAS wurde erfolgreich ausgeführt, die Transaktion scheint gelungen:
 - i die beiden verglichenen Operanden waren identisch, womit die Gültigkeit einer bestimmten Bedingung behauptet wird (*positive*),
 - ii aber diese Behauptung ist faktisch nicht korrekt (*false*)
- angenommen Prozesse P_1 and P_2 verwenden gleichzeitig Adresse L_i
 - Wert A gelesen von P_1 aus L_i meint einen bestimmten Zustand S_1 , aber P_1 wird verzögert, bevor der neue Wert an L_i bestätigt werden kann
 - zwischenzeitlich ändert P_2 den Wert an L_i in B und dann zurück in A , meint damit aber einen neuen globalen Zustand $S_2 \neq S_1$
 - P_1 fährt fort, erkennt, dass in L_i der Wert A steht und agiert aber unter der Annahme, dass der globale Zustand S_1 gilt — was jetzt falsch ist
- die **Kritikalität** solcher falsch positiven Ergebnisse steht und fällt mit dem Problem: `mult` ist unkritisch, nicht aber `push` und `pull`



- Ausgangszustand der Liste: $head \rightarrow A \rightarrow B \rightarrow C$, $head$ ist ref_{CAS} :

	$\mathfrak{M}$	Op.	*ref	old	new	Liste
1.	P_1	pull	A	A	B	unverändert
2.	P_2	pull	A	A	B	$ref \rightarrow B \rightarrow C$
3.	P_2	pull	B	B	C	$ref \rightarrow C$
4.	P_2	push	C	C	A	$ref \rightarrow A \rightarrow C$
5.	P_1	pull	A	A	B	$ref \rightarrow B \rightarrow \text{☹}$ A $\rightarrow$ C verloren

1. P_1 wird im pull vor CAS verzögert, behält lokalen Zustand bei

2.–4. P_2 führt die drei Transaktionen durch, aktualisiert die Liste

5. P_1 beendet pull mit dem zu 1. gültigen lokalen Zustand

- beachte: das eigentliche Problem ist die Wiederverwendung derselben Adresse, wofür ein **beschränkter Adressvorrat** die Ursache ist

- in 64-Bit-Systemen ist der Adressvorrat logisch nahezu unerschöpflich...



Kritische Variable mittels „Zeitstempel“ absichern

- **Abhilfe** besteht darin, den umstrittenen Zeiger (nämlich `item`) um einen problemspezifischen **Generationszähler** zu erweitern

Etikettieren

- Zeiger mit einem Anhänger (*tag*) versehen
- Ausrichtung (*alignment*) ausnutzen, z.B.:

$$\begin{aligned} \text{sizeof}(\text{chain_t}) &\leadsto 4 = 2^2 \Rightarrow n = 2 \\ &\Rightarrow \text{chain_t} * \text{ ist Vielfaches von } 4 \\ &\Rightarrow \text{chain_t} *_{\text{Bits}[0:1]} \text{ immer } 0 \end{aligned}$$

- Platzhalter für n -Bit Marke/Zähler in jedem Zeiger

DCAS

- Abk. für (engl.) *double compare and swap*
- Marke/Zähler als elementaren Datentyp auslegen
 - *unsigned int* hat Wertebereich von z.B. $[0, 2^{32} - 1]$
- zwei Maschinenworte (Zeiger, Marke/Zähler) ändern

- `push` bzw. `pull` verändern dann den Anhänger bzw. die Marke des Zählers (`item`) mit jedem Durchlauf um eine Generation



- in beiden Fällen können gekoppelte Prozesse ins Kreiseln geraten
 - Umlaufsperr**
 - gleichzeitige Prozesse kreiseln ohne Nutzen für sich selbst
 - sie kommen in der Schleife nicht mit Berechnungen voran
 - Schleifendauer bedeutet **Wartezeit**
 - **Nutzarbeit** startet mit einer atomaren Aktion (z.B. CAS)
 - Transaktion**
 - gleichzeitige Prozesse kreiseln mit Nutzen für sich selbst
 - sie kommen in der Schleife mit Berechnungen voran
 - Schleifendauer bedeutet **Nutzarbeitszeit**
 - **Nutzarbeit** endet mit einer atomaren Aktion (z.B. CAS)
- der dabei anfallende Aufwand lässt sich grob wie folgt überschlagen:
 - seien t_{ka} die Zeitdauer und t_{lock} die Unkosten (Ein-/Austrittsprotokoll) des kritischen Abschnitts
 - seien t_{na} die Zeitdauer und $o_{na} = t_{na} - t_{ka}$ die Unkosten (transaktionales Programm) des nebenläufigen Abschnitts, $t_{na} \geq t_{ka}$ angenommen
 - sei N die Zahl gekoppelter Prozesse
 - dann „rechnen“ sich Umlaufsperr
bzw. Transaktionen, falls:

$$\sum_{n=1}^N t_{lock}^n < \sum_{n=1}^N o_{na}^n$$

↪ Entwicklungsaufwand und Blockierungsnachteile unberücksichtigt...



Gliederung

Einführung

Umlaufsperr

Definition

Funktionsweise

Schlossalgorithmen

Diskussion

Transaktion

Motivation

Prinzip

Beispiele

Diskussion

Zusammenfassung



- mit dem Konzept der **Umlaufsperr**e wird wechselseitiger Ausschluss softwaregesteuert umgesetzt
 - pessimistischer Ansatz zum Schutz kritischer Abschnitte: *leicht*
 - negative Erwartung, dass sich Prozesse gleichzeitig an einer Stelle treffen
 - gekoppelte Prozesse blockieren wahrscheinlich nur für kurze Zeit
 - kritischer Aspekt ist die starke **Störanfälligkeit** bei hohem Wettstreit
 - Häufigkeit von „*read-modify-write*“-Zyklen pro Durchlauf minimieren
 - prozessspezifische Zurückhaltung vom wiederholten Sperrversuch
 - variable Verweilzeiten, um Konflikte bei Wiederholungen zu vermeiden
 - als blockierende Synchronisation besteht hohe **Verklemmungsgefahr**
- im Gegensatz dazu die **nichtblockierende Synchronisation**, bei der wechselseitiger Ausschluss ein Merkmal der Hardware ist
 - optimistischer Ansatz zum Schutz kritischer Abschnitte: *schwer*
 - positive Erwartung, dass Prozesse nicht gleichzeitig zusammentreffen
 - verklemmungsfrei, robust, nichtsequentiell, störunanfällig
- obwohl grundweg verschieden, sind **Spezialbefehle** der Hardware die beiden Konzepten gemeinsame Grundlage: TAS, CAS, FAA



Literaturverzeichnis I

- [1] ABRAMSON, N. :
The ALOHA System: Another Alternative for Computer Communication.
In: *Proceedings of the Fall Joint Computer Conference (AFIPS '70)*.
New York, NY, USA : ACM, 1970, S. 281–285

- [2] AMDAHL, G. M.:
Validity of the Single-Processor Approach to Achieving Large Scale Computing Capabilities.
In: *Proceedings of the AFIPS Spring Joint Computer Conference (AFIPS 1967)*,
AFIPS Press, 1967, S. 483–485

- [3] BRYANT, R. ; CHANG, H.-Y. ; ROSENBERG, B. S.:
Experience Developing the RP3 Operating System.
In: *Computing Systems* 4 (1991), Nr. 3, S. 183–216

- [4] DECHEV, D. ; PIRKELBAUER, P. ; STROUSTRUP, B. :
Understanding and Effectively Preventing the ABA Problem in Descriptor-based Lock-free Designs.
In: *Proceedings of the 13th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2010)*, IEEE Computer Society, 2010. – ISBN 978-1-4244-7083-9, S. 185–192



Literaturverzeichnis II

- [5] DIJKSTRA, E. W.:
Cooperating Sequential Processes / Technische Universiteit Eindhoven.
Eindhoven, The Netherlands, 1965 (EWD-123). –
Forschungsbericht. –
(Reprinted in *Great Papers in Computer Science*, P. Laplante, ed., IEEE Press, New
York, NY, 1996)

- [6] ESWARAN, K. P. ; GRAY, J. N. ; LORIE, R. A. ; TRAIGER, I. L.:
The notions of consistency and predicate locks in a database system.
In: *Communications of the ACM* 19 (1976), Nr. 11, S. 624–633

- [7] IBM CORPORATION (Hrsg.):
IBM System/370 Principles of Operation.
Fourth.
Poughkeepsie, New York, USA: IBM Corporation, Sept. 1 1974.
(GA22-7000-4, File No. S/370-01)

- [8] INTEL CORPORATION:
XCHG.
In: *x86 Instruction Set Reference*.
Rene Jeschke, Nov. 2015. –
http://x86.renejeschke.de/html/file_module_x86_id_328.html



Literaturverzeichnis III

- [9] JONES, M. B.:
What really happened on Mars?
http://www.research.microsoft.com/~mbj/Mars_Pathfinder/Mars_Pathfinder.html, 1997
- [10] KLEINÖDER, J. ; SCHRÖDER-PREIKSCHAT, W. :
Virtuelle Maschinen.
In: LEHRSTUHL INFORMATIK 4 (Hrsg.): *Systemprogrammierung*.
FAU Erlangen-Nürnberg, 2015 (Vorlesungsfolien), Kapitel 5.1
- [11] KUNG, H.-T. ; ROBINSON, J. T.:
On Optimistic Methods for Concurrency Control.
In: *ACM Transactions on Database Systems* 6 (1981), Jun., Nr. 2, S. 213–226
- [12] LAMPORT, L. :
A New Solution of Dijkstra's Concurrent Programming Problem.
In: *Communications of the ACM* 17 (1974), Aug., Nr. 8, S. 453–455
- [13] LAMPSON, B. W. ; REDELL, D. D.:
Experiences with Processes and Monitors in Mesa.
In: *Communications of the ACM* 23 (1980), Febr., Nr. 2, S. 105–117



Literaturverzeichnis IV

- [14] METCALFE, R. M. ; BOOGS, D. R.:
Ethernet: Distributed Packet Switching for Local Computer Networks.
In: *Communications of the ACM* 19 (1976), Jul., Nr. 5, S. 395–404

- [15] SCHRÖDER-PREIKSCHAT, W. :
Semaphore.
In: LEHRSTUHL INFORMATIK 4 (Hrsg.): *Concurrent Systems — Nebenläufige Systeme*.
FAU Erlangen-Nürnberg, 2014 (Vorlesungsfolien), Kapitel 7

- [16] WILNER, D. :
Vx-Files: What really happened on Mars?
Keynote at the 18th IEEE Real-Time Systems Symposium (RTSS '97), Dez. 1997



■ CAS als **intrinsische Funktion** des Kompilierers:

```
1  _acquire:
2      movl    4(%esp), %ecx    # get pointer to lock variable
3      movb    $1, %dl         # want to set lock "true"
4  LBB0_1:                    # come here for retry (while)
5      xorl    %eax, %eax      # test value is "false"
6      lock                                # next instruction is atomic
7      cmpxchgb %dl, (%ecx)    # compare and swap values
8      testb   %al, %al        # check if "true" was read
9      jne     LBB0_1          # if so, retry
10     ret                      # was "false" and is "true", now
```

5–7 ■ die eigentliche Umsetzung von CAS

- hier abgebildet auf ein `cmpxchg` des x86
- wobei `lock` lediglich die Atomarität dieses Befehls erzwingt

■ erkennbar sind auch die recht wenigen zusätzlichen Operationen der Umlaufsperr in der Wartephase (Z. 8–9)

- wobei die Variante schlecht skaliert (S. 13) und Verbesserung bedarf...



- sei $\Phi = \text{„addieren“} \rightsquigarrow \text{FAA (fetch and add):}$

```
1 type FAA(type *ref, type val) {  
2     atomic { type aux = *ref; *ref = aux + val; }  
3     return aux;  
4 }
```

- Reduktion auf eine atomare, in GCC eingebaute intrinsische Funktion:

```
5 #define FAA __sync_fetch_and_add
```

- Kompilierung in Assemblersprache (ASM86, AT&T Syntax):

```
6 _acquire: ...  
7     movl    16(%esp), %eax    # get pointer to global variable  
8     movl    $1, %ecx         # constant term to be added  
9     lock                                # make next instruction atomic  
10    xaddl    %ecx, (%eax)     # exchange and add operands  
11    ...                      # ecx now holds the value fetched
```

- sei $\Phi = \text{„einspeichern“} \rightsquigarrow \text{FAS (fetch and store)}$

- ein weiterer, überaus universell verwendbarer Spezialbefehl

- Reduktion auf eine atomare, in GCC eingebaute intrinsische Funktion:

```
12 #define FAS __sync_lock_test_and_set
```

- letztlich die Basisoperation, um TAS verfügbar zu machen (vgl. S. 11)



■ prozedurale Abstraktion von CAS:

```
1  _CAS :  
2      pushl    %esi          # save non-volatile data  
3      movl     12(%esp), %ecx # get test value  
4      movl     16(%esp), %edx # get target value  
5      movl     8(%esp), %esi  # get pointer to shared variable  
6      movl     %ecx, %eax     # setup test value for CAS  
7      lock                    # next instruction is atomic  
8      cmpxchgl %edx, (%esi)   # compare and swap (CAS) values  
9      cmpl     %ecx, %eax     # check if CAS succeeded (ZF=1)  
10     sete     %al            # expand ZF bit into operand  
11     movzbl   %al, %eax      # extent operand value to word size  
12     popl     %esi          # restore non-volatile data  
13     ret                    # "true" if succeeded, "false" else
```

■ die Unkosten (*overhead*) im Vergleich zur intrinsischen Funktion des Kompiliertes sind beträchtlich

- drei Befehle (Z. 6–8 bzw. S. 38, Z. 5–7) gegenüber 12 Befehle
- Parameterbe- und -entsorgung sowie Prozeduraufruf kämen hinzu...

