Systemprogrammierung

Grundlagen von Betriebssystemen

Teil C – X.4 Prozesssynchronisation: Kreiseln und Spezialbefehle

28. November 2024

Rüdiger Kapitza

(© Wolfgang Schröder-Preikschat, Rüdiger Kapitza)

Lehrstuhl für Informatik 4
Systemsoftware

Friedrich-Alexander-Universität
Technische Fakultät

Agenda

- Einführung
- Umlaufsperre
  - Definition
  - Funktionsweise
  - Schlossalgorithmen
  - Diskussion
- Transaktion
  - Motivation
  - Prinzip
  - Beispiele
  - Diskussion
- Zusammenfassung

Gliederung

Einführung

Umlaufsperre

- Definition
- Funktionsweise
- Schlossalgorithmen
- Diskussion

Transaktion

- Motivation
- Prinzip
- Beispiele
- Diskussion

Zusammenfassung

Lehrstoff

- Konzepte der **Befehlssatzebene** (s. [10]) kennenlernen, womit die Synchronisation gleichzeitiger Prozesse erreicht wird
  - **Umlaufsperren**, d.h., Sperren für mehr-/vielkernige Multiprozessoren
  - sperrfreie Synchronisation mittels (Mikro-)**Transaktionen**
- für Umlaufsperren typische **Schlossalgorithmen** behandeln und ihre Auswirkungen auf andere Prozesse 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

Umlaufsperre

- Definition
- Funktionsweise
- Schlossalgorithmen
- Diskussion

Transaktion

- Motivation
- Prinzip
- Beispiele
- Diskussion

Zusammenfassung

Umlaufsperre

Definition

Sperren durch schnelle Drehung

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

- diesem aber nicht ganz unähnlich:
  - Passierende ≡ Prozesse
  - Blickwendung ≡ Zustandsabfrage
  - Umlauf ≡ 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 in sogenannten **Schlossalgorithmen** (*lock algorithms*)

[1] vgl. auch `https://de.wikipedia.org/wiki/Umlaufgitter` (28/10/15).

Grundsatz

*Die **Umlaufsperre** 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 Sperre** zur Vorbeugung unvorhersehbarer Latenz,[2] beispielsweise eine Unterbrechungssperre — die allerdings den privilegierten Modus voraussetzt.*

[2] Wegen der sonst für gewöhnlich möglichen Unterbrechung und Verzögerung des die Sperre haltenden Prozesses: *lock-holder preemption*.

Umlaufsperre

Funktionsweise

Schlossalgorithmus: naive Fassung

busy waiting

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

```
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 Umlaufsperre 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  }
```

  - **wettlaufkritische 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. 24–27] scheiden aus: sie wirken nur lokal auf dem Prozessor, wofür Umlaufsperren eben nicht vorgesehen sind (S. 8)
      - 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 der Werte der Variable verändert sich 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**
    - unilaterale Sperre asynchroner Programmunterbrechungen[3] und
    - multilaterale Sperre gleichzeitiger Buszugriffe „von außen kommend“

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

Implementierung

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

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

(Heute eher *atomic_exchange* vgl. C++11 atomics)

- Anwendung dieser Funktion für den Schlossalgorithmus:

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

- Kompilierung in Assembliersprache (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 now "true"
```

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

Umlaufsperre

Schlossalgorithmen

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:
    - $n - 1$ Kopien invalidieren und 1 Original zum auslösenden Prozessor bewegen
      oder
    - 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 den Bus für andere Prozessoren
    - dazwischen liegen nur wenige (z.B. drei, vgl. S. 13) 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 bedingtem Schreiben, mit Ablesen oder 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:

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

  - 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. 46)
- in nichtfunktionaler Hinsicht skaliert die Lösung schlecht
  - ***bus-lock burst*** $\rightsquigarrow$ Kreiseln mit Ablesen oder mit Zurückhaltung…

Kreiseln mit Ablesen

spin on read

```
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 ist[4]
    - 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

↪ dabei wird eine Umlaufsperre aber gerade oft für diesen Fall favorisiert ☹

- bei großem Wettstreit stauen sich nach wie vor viele Prozesse (Z. 4)
  - sich wiederholende **Häufung der Bussperre** (*bus-lock burst*)
  - Prozessen anderer Prozessoren wird stoßartig Buszugriffe verwehrt
- in nichtfunk. Hinsicht skaliert die Lösung mehr oder weniger
  - Kreiseln mit Zurückhaltung: **Stauauflösung**, Lücken schaffen…

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

Kreiseln mit Zurückhaltung

spin with backoff

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

Verhungern

starvation

- alle bisher diskutierten Verfahren können Prozessen eine **nach oben unbegrenzte Wartezeit** bescheren
  - jedoch wird einem System gekoppelter Prozesse Fortschritt zugesichert — wenn Verklemmungen einmal außer Acht gelassen werden
  - jedoch können einzelne Prozesse ewig im Eintrittsprotokoll kreiseln
- grenzenlose Verzögerung einzelner Prozesse ist vorzubeugen
- die Wartezeit muss für jeden Prozess limitiert 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)

Kreiseln mit proportionaler Zurückhaltung

s. auch S. 47

- das **Maß der angestauten Prozesse** bestimmt den Wettstreitgrad, der im Moment der Sperrverhängung gilt ⇝ 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;
```

```
#define FAA __sync_fetch_and_add                /* atomic */
```

```
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 */
```

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 ☺

Umlaufsperre

Diskussion

Zwischenzusammenfassung

- **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
- Defizite, die blockierende Synchronisation grundsätzlich betreffen, jedoch mit Umlaufsperren besonders zum Vorschein kommen
  - Alternative ist die **nichtblockierende Synchronisation**: **Transaktion**

Gliederung

- Einführung
- Umlaufsperre
  - Definition
  - Funktionsweise
  - Schlossalgorithmen
  - Diskussion
- Transaktion
  - Motivation
  - Prinzip
  - Beispiele
  - Diskussion
- Zusammenfassung

Transaktion

Motivation

Nachteile blockierender Synchronisation

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

  Leistung (*performance*) paralleler Systeme 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 Verhungern (*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.

Transaktion

Prinzip

Kreiseln mit Bedingung

compare and swap, CAS

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

Eine **Konsistenzeinheit**, 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**

Bedingte Wertzuweisung

vgl. auch S. 16

- **atomare Bestätigungsaktion** einer Transaktion (S. 28, 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 Assembliersprache, siehe S. 50

- 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 **Schleifenrunpf** können durch gleichzeitige Prozesse geschehen, sie unterliegen nicht dem wechselseitigen Ausschluss
  - nur die **Fußsteuerung** der Schleife mit dem CAS läuft synchronisiert ab

Transaktion

Beispiele

Atomare multiplikative Variablenänderung

- Fassung als klassischer **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 wechelseitigem Ausschluss

Atomare Listenmanipulation I

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

```
1  typedef struct chain {
2      struct chain *link;
3  } chain_t;
```

```
4  typedef struct chainlock {
5      chain_t item;
6      detent_t bolt;
7  } chainlock_t;
```

- Einfügeoperation als klassischer **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 wechelseitigen Ausschluss

Atomare Listenmanipulation II

- 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 wechelseitigen Ausschluss

Transaktion

Diskussion

Umlaufsperre vs. Wiederholung

…*rollback* einer Transaktion

- in beiden Fällen können gekoppelte Prozesse ins Kreiseln geraten

Umlaufsperre

- gleichzeitige Prozesse kreiseln ohne Nutzen für sich selbst
- sie kommen in der Schleife nicht mit Berechnungen voran
- Schleifendauer bedeutet **Wartezeit**

Transaktion

- **Nutzarbeit** startet mit einer atomaren Aktion (z.B. CAS)
- 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)

- Unkosten des kritischen Abschnitts und der Transaktion abwägen
  - seien t_{ka} die Zeitdauer und t_{lock} die Unkosten des kritischen Abschnitts
  - ferner seien t_{na} die Zeitdauer und $o_{na} = t_{na} - t_{ka}$ die Unkosten des nebenläufigen Abschnitts, $t_{na} \geq t_{ka}$ angenommen
  - sei N die Zahl gekoppelter Prozesse

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

↪ Umlaufsperren „rechnen“ sich, falls:

↪ Entwicklungsaufwand und Blockierungsnachteile unberücksichtigt…

Gliederung

- Einführung
- Umlaufsperre
  - Definition
  - Funktionsweise
  - Schlossalgorithmen
  - Diskussion
- Transaktion
  - Motivation
  - Prinzip
  - Beispiele
  - Diskussion
- Zusammenfassung

Resümee

- mit dem Konzept der **Umlaufsperre** 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

Zusammenfassung

Bibliographie

Literaturverzeichnis (1)

[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. ; Rosenburg, B. S.:
Experience Developing the RP3 Operating System.
In: *Computing Systems* 4 (1991), Nr. 3, S. 183–216

Literaturverzeichnis (2)

[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

[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)

Literaturverzeichnis (3)

[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 (4)

[9] Jones, M. B.:
What really happened on Mars?
`http://www.cs.cornell.edu/courses/cs614/1999sp/papers/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

Literaturverzeichnis (5)

[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

[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

Literaturverzeichnis (6)

[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

Anhang

Schlossalgorithmen

Kreiseln mit CAS

- 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 now "true"
```

5–7
- die eigentliche Umsetzung von CAS, abgebildet auf `cmpxchg` (x86)
- wobei `lock` lediglich die Atomarität dieses Befehls erzwingt

- erkennbar sind auch die recht wenigen zusätzlichen Operationen der Umlaufsperre in der Wartephase (Z. 5–9)

Skalierungsproblem (vgl. S. 16: *bus-lock burst*)

Je mehr Prozesse gleichzeitig in die Schleife eintreten, desto länger die nahtlose Sequenz der busatomaren Befehle `cmpxchg`.

Atomar „abrufen und Φ tun“

fetch and Φ

- sei $\Phi =$ „addieren“ $\rightsquigarrow$ FAA (*fetch and add*, vgl. S. 20):

```
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 Assembliersprache (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 $\Phi =$ „einspeichern“ $\rightsquigarrow$ 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. 13)

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 **Datentyperweiterung** (vgl. S. 10) 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 **Schlafsperre** (*sleeping lock*) gäbe hier den Prozessor frei
  - die Unkosten dafür sollten aber in die **effektive Verweilzeit** einfließen ☹

Anhang

Transaktionsprinzip

Bedingte Wertezuweisung

- **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      # set up 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       # extend 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 Kompilieres sind beträchtlich
  - drei Befehle (Z. 6–8 bzw. S. 46, Z. 5–7) gegenüber 12 Befehlen
  - Parameterbe- und -entsorgung sowie Prozeduraufruf kämen hinzu…

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`

Stapel mit Wettlaufsituation

- Ausgangszustand der Liste: *head* ⇨ *A* ⇨ *B* ⇨ *C*, *head* ist $\texttt{ref}_{CAS}$:

	ℳ	Op.	`*ref`	`old`	`new`	Liste	
1.	P_1	`pull`	*A*	*A*	*B*	unverändert	
2.	P_2	`pull`	*A*	*A*	*B*	`ref` ⇨ *B* ⇨ *C*	
3.	P_2	`pull`	*B*	*B*	*C*	`ref` ⇨ *C*	
4.	P_2	`push`	*C*	*C*	*A*	`ref` ⇨ *A* ⇨ *C*	
5.	P_1	`pull`	*A*	*A*	*B*	`ref` ⇨ *B* ⇨ ☹	*A* ⇨ *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 Grundproblem 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.:

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

- 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