

# Systemprogrammierung

## Prozesssynchronisation: Befehlssatzebene

Wolfgang Schröder-Preikschat

Lehrstuhl Informatik 4

7. Juni 2011

## Prozesssynchronisation auf der Befehlssatzebene

**Alleinstellungsmerkmal** dieser Abstraktionsebene sind allgemein die in der **CPU** manifestierten Fähigkeiten eines Rechensystems, hier:

- (a) in Bezug auf die Bereitstellung von Spezialbefehlen und
- (b) hinsichtlich der Semantik dieser Befehle zur Prozessverarbeitung

Techniken zur Synchronisation gleichzeitiger Prozesse können demzufolge nur auf sehr einfache, elementare Konzepte zurückgreifen

zu (a) die Möglichkeit, externe/interne Prozesse aussperren zu können

- Unterbrechungssperre X
- Schlossvariable, Umlaufsperre X

zu (b) die Möglichkeit, kritische Abschnitt so auszuführen, dass gleichzeitige Prozesse nicht ausgesperrt werden

- nichtblockierende Synchronisation X

## Gliederung

### 1 Unterbrechungssteuerung

- Prinzip
- Implementierung

### 2 Schlossvariable

- Definition
- Implementierung
- Diskussion

### 3 Nichtblockierende Synchronisation

- Prinzip
- Elementaroperation
- Anwendung
- Diskussion

### 4 Zusammenfassung

### 5 Anhang

## Kontrolle asynchroner Programmunterbrechungen

Ansatz: asynchrone Programmunterbrechungen entweder verhindern oder tolerieren, und zwar durch **Verzögerung** der...

überlappenden Aktivität  $\mapsto$  pessimistisches Verfahren

- Spezialbefehle der  $\begin{cases} \text{Ebene 2: cli, sti (x86)} \\ \text{Ebene 3: sigprocmask (POSIX)} \end{cases}$  „hart“

überlappten Aktivität  $\mapsto$  optimistisches Verfahren

- durch Spezialbefehle und Programme der Ebene 2:  
**CISC**  $\hookrightarrow$  CAS (IBM 370, m68020+), CMPXCHG (i486+)  
**RISC**  $\hookrightarrow$  LL/SC (DEC Alpha, MIPS, PowerPC)
- ohne Spezialbefehle  $\sim$  BS/BST „weich“

**Unterbrechungen sperren: einfach — aber nicht immer zweckmäßig**

- Faustregel: harte Synchronisation ist möglichst zu vermeiden

## Wiedersehen mit einem alten Problem: Überlapptes Zählen

```
int wheel = 0;
```

Plötzlich...

```
void __attribute__((interrupt))  
niam () {  
    wheel++;  
}
```

Schlecht...

```
main () {  
    for (;;)  
        printf("%u\n", wheel++);  
}
```

### Wettlaufsituation

- kritischer Abschnitt: `++`
- Laufgefahr vorbeugen
- ELOP `int fai(int*)`
- unteilbares Zählen konstruktiv und problemadäquat sicherstellen

Besser...

```
main () {  
    for (;;)  
        printf("%u\n", fai(&wheel));  
}
```

## Unterbrechungssperre: x86

```
typedef unsigned short irq_t; /* flags register placeholder */  
  
void irq_block(); /* disable interrupts at CPU */  
void irq_admit(); /* enable interrupts at CPU */  
  
irq_t irq_avert(); /* save CPU interrupt level then block */  
void irq_treva(irq_t); /* restore saved CPU interrupt level */  
  
void irq_block() {  
    asm volatile("cli");  
}  
  
irq_t irq_avert() {  
    irq_t flags;  
    asm volatile(  
        "pushf; pop %0; cli"  
        : "=g" (flags));  
    return flags;  
}  
  
void irq_admit() {  
    asm volatile("sti");  
}  
  
void irq_treva(irq_t flags) {  
    asm volatile(  
        "push %0; popf"  
        : : "g" (flags));  
}
```

## Verhinderung vs. Tolerierung von *Interrupts*

### Verhinderung

```
int fai (int *ref) {  
    int aux;  
  
    asm volatile ("cli");  
    aux = (*ref)++;  
    asm volatile ("sti");  
  
    return aux;  
}
```

- Komplexbefehl der Ebene 3
- privilegierte Befehle cli/sti
- Elementaroperation des BS

### Tolerierung

```
int fai (int *ref) {  
    int aux = 1;  
  
    asm volatile ("xaddl %0,%1"  
        : "=g" (aux), "=g" (*ref)  
        : "0" (aux), "1" (*ref));  
  
    return aux;  
}
```

- Komplexbefehl der Ebene 2
- unprivilegierter Befehl xadd
- Elementaroperation der CPU

### Beachte: Multiprozessorbetrieb

- die Befehle haben nur lokale Signifikanz für „ihren“ Prozessor(kern)

## Schutz kritischer Abschnitte durch Unterbrechungssperre

### solo-Optionen: Unterbrechungssperre

```
typedef irq_t solo_t;  
  
#define CS_ENTER(solo) irq_block()  
#define CS_LEAVE(solo) irq_admit()  
  
#define CS_ENTER(solo) *solo = irq_avert()  
#define CS_LEAVE(solo) irq_treva(*solo)
```

- für unverschachtelte KA
- ohne Zustandssicherung
- für verschachtelte KA
- mit Zustandssicherung

### Beachte: Unabhängige gleichzeitige Prozesse

- werden unnötig zurückgehalten, obwohl sie den KA nicht durchlaufen

### Beachte: Unterbrechungsverzögerung

- die größte WCET<sup>a</sup> aller durch Unterbrechungssperre gesicherten KA
- erhöhtes Risiko des Verlusts von Unterbrechungsanforderungen

<sup>a</sup>Abk. für (engl.) worst case execution time.

## Gliederung

### 1 Unterbrechungssteuerung

- Prinzip
- Implementierung

### 2 Schlossvariable

- Definition
- Implementierung
- Diskussion

### 3 Nichtblockierende Synchronisation

- Prinzip
- Elementaroperation
- Anwendung
- Diskussion

### 4 Zusammenfassung

### 5 Anhang

## Schlossalgorithmus: Prinzip — mit Problem(en)

```
typedef struct lock {
    volatile bool busy; /* status of critical section */
    ...
} lock_t;
```

### Laufgefahr

```
void lv_acquire (lock_t *lock) {
    while (lock->busy);
    lock->busy = true;
}
```

```
void lv_release (lock_t *lock) {
    lock->busy = false;
}
```

### Abprüfen und setzen (engl. *test and set*, TAS) der Schloßvariablen

- muss als **Elementaroperation** und wirklich atomar ausgelegt sein

## Schlossvariable (engl. *lock variable*)

**Datentyp**, der zwei grundlegende Operationen definiert:

acquire (auch: *lock*)  $\models$  Eintrittsprotokoll

- verzögert einen Prozess, bis das zugehörige Schloss offen ist
  - bei geöffnetem Schloss fährt der Prozess unverzögert fort
  - verschließt das Schloss („von innen“), wenn es offen ist

release (auch: *unlock*)  $\models$  Austrittsprotokoll

- öffnet ein Schloss, ohne den öffnenden Prozess zu verzögern

Implementierungen dieses Konzepts werden auch als **Schlossalgorithmen** (engl. *lock algorithms*) bezeichnet

## Schlossvariable atomar abprüfen und setzen

### Kritischer Abschnitt: solo\_t

```
int tas(lock_t *lock) {
    bool busy;

    CS_ENTER(&lock->gate);
    busy = lock->busy;
    lock->busy = true;
    CS_LEAVE(&lock->gate);

    return busy;
}
```

**lock\_t** mit solo-Option:

- Verdrängungssperre oder
- Unterbrechungssperre
- untauglich für Multiprozessoren

### Elementaroperation: x86

```
int tas(volatile bool *lock) {
    bool busy;

    busy = true;
    asm volatile("lock xchgb %0,%1"
                : "=q" (busy), "=m" (*lock)
                : "0" (busy));

    return busy;
}
```

- „read-modify-write“
- atomarer Buszyklus
- Operandenaustausch
- tauglich für Multiprozessoren

## Umlaufsperre (engl. *spin lock*)

### „Drehschloss“

```
void lv_acquire (lock_t *lock) {
    while (TAS(lock));
}
```

TAS kommt in zwei Varianten (S. 12):  
 ↪ int tas(lock\_t \*)  
 ↪ int tas(volatile bool \*)

### Gefahr von Leistungsabfall

- pausenloses Schleifen allein nur mit TAS:
  - (a) erhöht das Risiko, Anforderungen von Programmunterbrechungen oder Prozessverdrängungen zu verpassen
    - die Unterbrechungs- bzw. Verdrängungssperre ist fast nur noch gesetzt
  - (b) hindert andere Prozessoren am Buszugang bzw. sorgt für eine überaus hohe Last im Kohärenzprotokoll des Zwischenspeichers
    - der Prozessor führt fast nur noch „read-modify-write“-Zyklen durch
- das Problem verschärft sich massiv, wenn (viele) gleichzeitige Prozesse den **Wettstreit** (engl. *contention*) um das „Drehschloss“ aufnehmen

## Aktives Warten (engl. *busy waiting*)

**Unzulänglichkeit der Schlossalgorithmen:** der aktiv wartende Prozess...

- kann keine Änderung der Bedingung herbeiführen, auf die er wartet
- behindert andere Prozesse, die sinnvolle Arbeit leisten könnten
- schadet damit letztlich auch sich selbst

*Je länger der Prozess den Prozessor für sich behält, umso länger muss er darauf warten, dass andere Prozesse die Bedingung erfüllen, auf die er selbst wartet.*

- in den meisten Fällen sind Effizienzeinbußen in Kauf zu nehmen
- es sei denn, jeder Prozess hat seinen eigenen realen Prozessor(kern)

**Allgemein ein nur bedingt effektives Verfahren**

- notwendige Bedingung**     • kurze Laufzeit des zu schützenden KA  
**hinreichende Bedingung**     • CPU-Abgabe nach gescheitertem Versuch

## Sensitive Umlaufsperre

### Nichtinvasives Warten

```
void lv_acquire (lock_t *lock) {
    do {
        while (lock->busy);
    } while (TAS(lock));
}
```

### Zurücktretendes Warten

```
void lv_acquire(lock_t *lock) {
    while (true) {
        while (lock->busy);
        if (!TAS(lock)) break;
        lv_backoff(lock);
    }
}
```

Eigenschaften wie zuvor, zusätzlich:

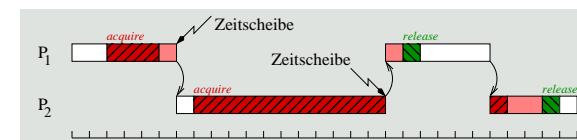
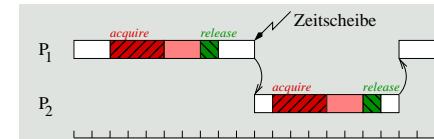
- Wettstreit (engl. *contention*) aus dem Wege gehend
- zusätzliche Wartezeit („*back-off*“) nach gescheitertem TAS
- prozessspezifisch, ansteigend

### Exponentielles Zurücktreten (engl. *exponential back-off*)

- beschränkte Wartezeitverdopplung mit jedem gescheitertem Versuch

## Aktives Warten ohne Prozessorabgabe

*„Spin locking considered harmful“*



	$T_s$	$T_q$	$T_q/T_s$
$P_1$	12	12	1.0
$P_2$	8	8	1.0

	$T_s$	$T_q$	$T_q/T_s$
$P_1$	12	24	2.0
$P_2$	17	23	1.35

### Verbesserung: Prozessorabgabe in der Warteschleife (vgl. S. 34)

- laufend ↪ bereit     in Laufbereitschaft bleiben     *ps\_forgo*  
 laufend ↪ blockiert     schlafend die Schlossfreigabe erwarten     *ps\_sleep*

## Gliederung

### 1 Unterbrechungssteuerung

- Prinzip
- Implementierung

### 2 Schlossvariable

- Definition
- Implementierung
- Diskussion

### 3 Nichtblockierende Synchronisation

- Prinzip
- Elementaroperation
- Anwendung
- Diskussion

### 4 Zusammenfassung

### 5 Anhang

## Optimistisches Verfahren der Synchronisation

Nebenläufigkeit unterstützen, nicht einschränken wie beim wechselseitigen Ausschluss

Koordinierung sich einander ggf. überlappender Aktivitäten, ohne dabei **gleichzeitige Prozesse** auszuschließen

- toleriert (pseudo-) parallele Programmausführungen
  - parallel • Multiprozessor, wirkliche Parallelität
  - pseudoparallel • Uniprozessor, Parallelität durch Unterbrechungen
- die Verfahren greifen auf nichtprivilegierte Befehle der ISA zurück
  - CISC • TAS, FAA, CAS bzw. CMPXCHG
  - RISC • LL/SC
- d.h., sie funktionieren im Benutzer- wie auch im Systemmodus

### Beachte

- kein wechselseitiger Ausschluss  $\Rightarrow$  **Verklemmungsvorbeugung**
- die **benutzten** Befehle müssen „echte“ **Elementaroperationen**

## Blockierende Synchronisation „considered harmful“

Probleme von Schlossvariablen, Semaphore und Monitore

**Leistung** (engl. *performance*) insbesondere in SMP-Systemen [1]

- „*spin locking*“ reduziert ggf. massiv Busbandbreite

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

- ein im kritischen Abschnitt scheiternder Prozess kann schlimmstenfalls das ganze System lahm legen

**Einplanung** (engl. *scheduling*) wird behindert bzw. nicht durchgesetzt

- un- bzw. weniger wichtige Prozesse können wichtige Prozesse „ausbremsen“ bzw. scheitern lassen
- **Prioritätsverletzung, Prioritätsumkehr** [3]
  - Mars Pathfinder [5, 2]

**Verklemmung** (engl. *deadlock*) einiger oder sogar aller Prozesse

## Muster nichtblockierender Synchronisation (NBS)

erledige NBS mit CAS;

wiederhole

ziehe *lokale Kopie* des Inhalts der *Adresse* einer globalen Variablen; verwenden die Kopie, um einen neuen *lokalen Wert* zu berechnen; versuche CAS: sichere den *lokalen Wert* an die *Adresse*, wenn ihr Inhalt immer noch mit dem Wert der *lokalen Kopie* identisch ist; solange CAS scheitert; **basta**.

- |      |  |
|------|--|
| pros | <ul style="list-style-type: none"> <li>• Tolerierung beliebiger Überlappungsmuster</li> <li>• transparent für die Einplanung: keine Prioritätsumkehr</li> <li>• <b>Verklemmungsvorbeugung</b>: kein wechselseitiger Ausschluss</li> <li>• Robustheit: keine hängenden Sperren bei Programmabbrüchen</li> </ul> |
| cons | <ul style="list-style-type: none"> <li>• Wiederverwendung sequentieller Altsoftware unmöglich</li> <li>• Gefahr von <b>Verhungern</b> (engl. <i>starvation</i>) in simplen Lösungen</li> <li>• Entwicklung nebenläufiger Varianten im Regelfall nicht trivial</li> </ul>                                       |

## Bedingte Wertzuweisung: *Compare and Swap*, CAS

```
int cas(word_t *ref, word_t exp, word_t val) {
    solo_t gate;      /* assume bus resp. interrupt lock */
    bool done;

    CS_ENTER(&gate);
    if (done = (*ref == exp)) *ref = val;
    CS_LEAVE(&gate);

    return done;
}
```

- ref** • die Adresse des atomar, bedingt zu ändernden Speicherworts  
**exp** • der unter der Adresse **ref** erwartete alte Wert  
**val** • der unter der Adresse **ref** zu speichernde neue Wert  
 • die Speicherung erfolgt nur, wenn der unter **ref** gespeicherte Wert dem erwarteten Wert **exp** gleicht  
 • lag Gleichheit vor, liefert die Funktion *true*, anderenfalls *false*

## Sperrfreie (engl. *lock-free*) Wartestapelmanipulation

LCFS (Abk. für engl. *last come, first served*)

### Aufnahme in die Liste...

```
void lf_push(chain_t *head, chain_t *item) {
    do item->link = head->link;           /* is elected head */
    while (!CAS(&head->link, item->link, item)); /* try push item */
}
```

### Entnahme aus der Liste...

```
chain_t *lf_pull(chain_t *head) {
    chain_t *item;

    do if ((item = head->link) == 0) break; /* access head */
    while (!CAS(&head->link, item, item->link)); /* try pull item */

    return item;
}
```

```
#define CAS(r,e,v) cas((word_t*)r, (word_t)e, (word_t)v)
```

## Bedingte Wertzuweisung: CAS als Spezialbefehl (x86)

```
ZF = (eax == *ref) ? (*ref = val, true) : (eax = *ref, false)
int cas(word_t *ref, word_t exp, word_t val) {
    unsigned char done;      /* "sete" writes to low byte of operand */

    asm volatile(
        "lock\n\t"           /* next comes atomic read-modify-write */
        "cmpxchgl %2,%1\n\t" /* conditional exchange of operands */
        "sete %0"           /* transfer value of zero flag (ZF bit) */
        : "=q" (done), "=m" (*ref)
        : "r" (val), "m" (*ref), "a" (exp)
        : "memory");
}

return done;
}
```

- lock** • setzt die Bussperre für den nachfolgenden Befehl  
 • zwingend für Multi(kern)prozessorsysteme, optional sonst  
**cmpxchgl** • *compare & exchange*:  $ZF = true \rightarrow$  Speicherung erfolgt  
**sete** • definiert Variable **done** mit dem Wert der ZF-Flagge

## Problem ABA

Transaktion gelingt trotz zwischenzeitlicher Änderungen

Phänomen der nichtblockierenden Synchronisation auf Basis eines CAS, d.h., einer ELOP, die inhaltsbasiert arbeitet<sup>1</sup>

- angenommen zwei Fäden,  $F_1$  und  $F_2$ , stehen im Wettstreit um eine gemeinsame Variable  $V$ 
  - $F_1$  • liest den Wert  $A$  von  $V$ , speichert diesen als Kopie, wird dann allerdings vor dem  $CAS_V$  für unbestimmte Zeit verzögert
  - $F_2$  • durchläuft dieselbe Sequenz, schafft jedoch mittels  $CAS_V$  den Wert  $B$  an  $V$  zuzuweisen
  - anschließend wird (in einem weiteren Durchlauf dieser Sequenz) wieder der ursprüngliche Wert  $A$  an  $V$  zugewiesen
  - $F_1$  • setzt seine Ausführung mit  $CAS_V$  fort, erkennt, dass  $V$  den Wert  $A$  seiner Kopie speichert und überschreibt  $V$
- im Ergebnis kann dieses Überlappungsmuster dazu führen, dass  $F_1$  mittels  $CAS_V$  einen falschen Wert nach  $V$  transferiert

<sup>1</sup>Bei Adressreservierung wie z.B. mit LL/SC besteht dieses Problem nicht.

## Problem ABA: Wartestapel mit Wettlaufsituation

Ausgangszustand der (LCFS) Liste:  $head \diamond A \diamond B \diamond C$ ,  $head$  ist  $\text{ref}_{CAS}$ :

CAS-Parameter						
$\mathfrak{m}_i$	Op.	*ref	exp	val	Liste	
1.	$F_1$	<i>pull</i>	A	A	B	unverändert
2.	$F_2$	<i>pull</i>	A	A	B	$\text{ref} \diamond B \diamond C$
3.	$F_2$	<i>pull</i>	B	B	C	$\text{ref} \diamond C$
4.	$F_2$	<i>push</i>	C	C	A	$\text{ref} \diamond A \diamond C$
5.	$F_1$	<i>pull</i>	A	A	<b>B</b>	$\text{ref} \diamond B \diamond \odot$ A $\diamond$ C verloren

1.  $F_1$  wird im *pull* vor CAS unterbrochen, behält lokalen Zustand bei
- 2.–4.  $F_2$  führt die Operationen komplett aus, aktualisiert die Liste
5.  $F_1$  beendet *pull* mit dem zum Zeitpunkt 1. gültigen lokalen Zustand

## Generationszähler „considered harmful“?

**Abhilfe (engl. workaround)** zum ABA-Problem  $\rightsquigarrow$  Bedingte Lösung

- die Effektivität des Lösungsansatzes steht und fällt mit dem für den Generationszähler definierten **endlichen Wertebereich**
  - dessen Auslegung letztlich vom jeweiligen Anwendungsfall abhängt
- **Überlappungsmuster** gleichzeitiger Prozesse haben Einfluss auf den für den Generationszähler zur Verfügung stehenden Wertebereich
  - bestimmt durch Zusammenspiel *und* Anzahl der wettstreitigen Fäden
  - ein Bit kann reichen, ebenso, wie ein *unsigned int* zu klein sein kann
- diese, dem jeweiligen Anwendungsfall zu entnehmenden Muster zu entdecken, ist zumeist schwer und nicht selten unmöglich

**Vorbeugung (engl. prevention)** muss zuerst kommen — sofern machbar:

- beliebige Überlappungsmuster konstruktiv (Entwurf) ausschließen
- oder auf **Adressreservierungsverfahren** der Hardware zurückgreifen
  - unterstützt nicht jede Hardware, ist nur typisch für RISC
  - z.B. ELOP-Paar *load linked, store conditional* (LL/SC) verwenden

## Kritische Variable mittels „Zeitstempel“ absichern

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

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

$\text{sizeof}(\text{chain\_t}) \rightsquigarrow 4 = 2^2 \Rightarrow n = 2$   
 $\Rightarrow \text{chain\_t} *$  ist Vielfaches von 4  
 $\Rightarrow \text{chain\_t} *_{\text{Bits}[0:1]}$  immer 0

- DCAS**
- Platzhalter für  $n$ -Bit Marke/Zähler in jedem Zeiger
  - 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 sodann den Zeigerwert um eine Generation

## Wiederholungsversuche — Aktives Warten?

Scheitern der etwa durch CAS<sup>2</sup> abzuschließenden **Transaktion** zieht die Wiederholung des kompletten Vorbereitungsvorgangs nach sich

- je höher der Grad an Wettstreitigkeit unter gleichzeitigen Prozessen, umso höher die Wahrscheinlichkeit, dass CAS scheitert
- ein zur Umlauf sperre sehr ähnliches Problem ergibt sich, das jedoch durch **sensitive Techniken** gleicher Art lösbar ist (vgl. S. 13–14)
  - Häufigkeit von „*read-modify-write*“-Zyklen pro Durchlauf minimieren
  - prozessspezifisches Zurücktreten vom erneuten Transaktionsversuch
  - variable Wartezeiten, um Konflikte bei Wiederholungen zu vermeiden

### Unterschied zur Umlauf sperre

- gleichzeitige Prozesse müssen nicht untätig darauf warten, dass ein kritischer Abschnitt frei ist, d.h., das TAS gelingt
- sie verlassen die Schleife jedoch erst, wenn das CAS gelingt, kommen in der Schleife allerdings mit ihren (lokalen) Berechnungen voran

<sup>2</sup>Für LL/SC-artige Elementaroperationen gilt dies ebenfalls.

## Gliederung

### 1 Unterbrechungssteuerung

- Prinzip
- Implementierung

### 2 Schlossvariable

- Definition
- Implementierung
- Diskussion

### 3 Nichtblockierende Synchronisation

- Prinzip
- Elementaroperation
- Anwendung
- Diskussion

### 4 Zusammenfassung

### 5 Anhang

## Literaturverzeichnis

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

[2] JONES, M. B.: *What really happened on Mars?* [http://www.research.microsoft.com/~mbj/Mars\\_Pathfinder/Mars\\_Pathfinder.html](http://www.research.microsoft.com/~mbj/Mars_Pathfinder/Mars_Pathfinder.html), 1997

[3] 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

[4] PAPAMARCOS, M. S. ; PATEL, J. H.: A Low-Overhead Coherence Solution for Multiprocessors with Private Cache Memories. In: *Proceedings of the 11th Annual International Symposium on Computer Architecture (ISCA '84)*, June 5–7, 1984, Ann Arbor, Michigan, USA, ACM Press, 1984, S. 348–354

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

## Resümee

### • Unterbrechungssteuerung

- verlangt von den Prozessen besondere Rechte: **privilegierte Befehle**
- unbeteiligte gleichzeitige Prozesse werden unnötig ausgesperrt
- externe Ereignisse (*Interrupts*) können verloren gehen
- ungeeignet für Multi(kern)prozessorsysteme: lokale Signifikanz

### • die Schlossvariable kommt meist mit **Umlaufsperrre** zum Einsatz

- pessimistischer Ansatz zum Schutz kritischer Abschnitte: *leicht*
  - gleichzeitige Prozesse werden als sehr wahrscheinlich angenommen
  - bezogen auf den jeweils zu schützenden kritischen Abschnitt
- im Regelfall bedingen Leistungsanforderungen **sensitive Verfahren**
  - Häufigkeit von „*read-modify-write*“-Zyklen pro Durchlauf minimieren
  - prozessspezifisches Zurücktreten vom erneuten Schließversuch mit TAS
  - variable Wartezeiten, um Konflikte bei Wiederholungen zu vermeiden
- als blockierende Synchronisation besteht hohe **Verklemmungsgefahr**

### • nichtblockierende Synchronisation

- optimistischer Ansatz zum Schutz kritischer Abschnitte: *schwer*
- die Verfahren müssen ebenfalls sensitiv für Plattformeigenschaften sein

## Gliederung

### 1 Unterbrechungssteuerung

- Prinzip
- Implementierung

### 2 Schlossvariable

- Definition
- Implementierung
- Diskussion

### 3 Nichtblockierende Synchronisation

- Prinzip
- Elementaroperation
- Anwendung
- Diskussion

### 4 Zusammenfassung

### 5 Anhang

## Zurücktreten vom aktiven Warten mit Prozessorabgabe

```
void lv_backoff(lock_t *lock) {
    time_t t0, t1, t2;

    t0 = ps_pitch(pd_being());

    t1 = t_stamp();
    lv_suspend(lock);
    t2 = t_stamp();

    if (t_pitch(t1, t2) < t0)
        ps_delay(t0 - t_pitch(t1, t2));
}
```

### Prozessverwaltung

*ps\_pitch* definiert den Abstand zum nächsten Versuch  
*ps\_delay* verzögert den Prozess: Prozessorabgabe

### Zeitverwaltung

*t\_stamp* liefert einen Zeitwert  
*t\_pitch* berechnet Zeitabstand

### Beachte: Benutzte Abstraktionsebene

- Prozessorabgabe setzt ein Prozesskonzept voraus, das typischerweise von einem Betriebssystem bereitgestellt wird
- in dieser Implementierungsvariante wird die Umlaufsperre zu einem Konzept der Maschinenprogrammebene

## Prozessorabgabe: passives Warten auf Freigabe der Sperre

### Laufbereit bleibend

```
void lv_suspend(lock_t *lock) {
    ps_forgo();
}
```

Effektivität hängt stark ab von der Umplanungsstrategie:

- RR** der Prozess kommt ans Ende der Bereitliste ☺  
**sonst** seine (stat./dyn.) Priorität bestimmt die Listenposition
- ggf. landet er ganz vorne
  - so dass er weiterläuft ☺

### Wach bleiben oder schlafen legen...

- eine Frage des jew. Anwendungsprofils und der Einplanungsstrategie

### Sperrfreigabe erwartend

```
void lv_suspend(lock_t *lock) {
    ps_sleep(&lock->bell);
}
```

### schlafen legende Schlossvariable

(engl. *sleeping lock*)

- Bedingungssynchronisation

### Angepasste Sperrfreigabe

```
void lv_release (lock_t *lock) {
    lock->bolt = 0;
    ps_rouse(lock);
}
```