

Systemprogrammierung

Grundlagen von Betriebssystemen

Teil C – X.3 Prozesssynchronisation: Semaphore und Sperren

Wolfgang Schröder-Preikschat

29. November 2019



Gliederung

Einführung

Semaphor

Definition

Anwendung

Implementierung

Ablaufunterbrechung

Mutex

Abgrenzung

Implementierung

Sperre

Grundsätzliches

Varianten

Zusammenfassung



Agenda

Einführung

Semaphor

Definition

Anwendung

Implementierung

Ablaufunterbrechung

Mutex

Abgrenzung

Implementierung

Sperre

Grundsätzliches

Varianten

Zusammenfassung



Lehrstoff

- das Konzept der **Maschinenprogrammebene** (s. [11]) kennenlernen, mit dem die Synchronisation gleichzeitiger Prozesse erreicht wird
 - binärer, allgemeiner bzw. ausschließender, zählender, privater Semaphor
 - zwei Varianten für zwei **Synchronisationsmuster**: ein- vs. mehrseitig
- die Implementierung eines Semaphors durchleuchten und sich damit auseinandersetzen, **wettlaufkritische Aktionen** zu bewältigen
 - ablaufinvariante bzw. unteilbare/atomare Semaphorprimitiven
 - beispielhaft diese als kritischen Abschnitt begreifen: Standardsicht
 - Ereignisvariable zur Bedingungssynchronisation in diesem Abschnitt
- den **Mutex** erklären als minimale (funktionale) Erweiterung eines binären Semaphors zum *autorisierten* wechselseitigen Ausschluss
 - genauer: ausschließender Semaphor mit Kontrolle der Eigentümerschaft
 - Eigentumslosigkeit für Semaphore als Merkmal nicht als Makel verstehen
- schließlich **Sperren** behandeln, um Atomanität der Primitiven eines Semaphors physisch gewährleisten zu können
 - Unterbrechungs-, Fortsetzungs- und Verdrängungssperre
 - d.h., Lösungen für (einkernige) Uniprozessorsysteme: **Pseudoparallelität**



Gliederung

Einführung

Semaphor

- Definition
- Anwendung
- Implementierung
- Ablaufunterbrechung

Mutex

- Abgrenzung
- Implementierung

Sperre

- Grundsätzliches
- Varianten

Zusammenfassung

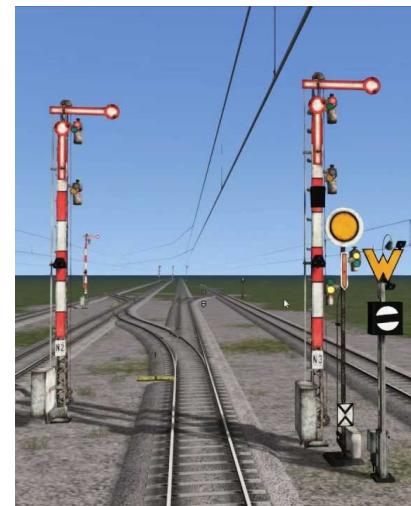
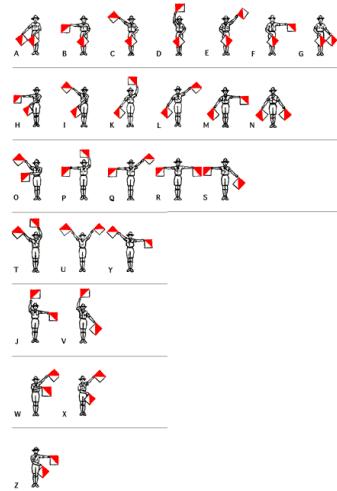
© wosch

SP (WS 2019/20, C – X.3)

2. Semaphor

X.3/5

Instrument zur Kommunikation und Koordination



© wosch

SP (WS 2019/20, C – X.3)

2.1 Semaphor – Definition

X.3/7

Semaphor

vgl. [8, S. 30]

- spezielle **ganzzahlige Variable** [4, p. 345] mit zwei Operationen [2]:

P Abk. für (Hol.) **prolaag**; alias *down*, *wait* oder *acquire*

- verringert¹ den Wert des Semaphors s um 1:

- i genau dann, wenn der resultierende Wert nichtnegativ wäre [3, p. 29]
- ii logisch uneingeschränkt [4, p. 345]

- ist oder war der Wert vor dieser Aktion 0, blockiert der Prozess
 - er kommt auf eine mit dem Semaphor assoziierte Warteliste

V Abk. für (Hol.) **verhoog**; alias *up*, *signal* oder *release*

- erhöht¹ den Wert des Semaphors s um 1

- ein ggf. am Semaphor blockierter Prozess wird wieder bereitgestellt
 - welcher Prozess von der Warteliste genommen wird, ist nicht spezifiziert

- beide Primitiven sind logisch oder physisch **unteilbare Operationen**

- ursprünglich definiert als **binärer Semaphor** ($s = [0, 1]$), generalisiert als **allgemeiner Semaphor** ($s = [n, m]$, $m > 0$ und $n \leq m$)

¹Nicht zwingend durch Subtraktion oder Addition im arithmetischen Sinn.



© wosch

SP (WS 2019/20, C – X.3)

2.1 Semaphor – Definition

X.3/6

Signalisierender Semaphor

allgemeiner, zählender, privater

- einseitige Synchronisation** (Beispielvorlage, vgl. auch [7, S. 33]):

```
1 typedef struct buffer {  
2     char ring[64]; /* buffer memory: ring buffer */  
3     int_t in, out; /* initial: {0,1}, {0,1} */  
4     semaphore_t free, data; /* initial: 64, 0 */  
5 } buffer_t;  
6  
7 void put(buffer_t *pool, char item) {  
8     P(&pool->free); /* block iff buffer is full: free = 0 */  
9     pool->ring[FAA(&pool->in, 1) % 64] = item;  
10    V(&pool->data); /* signal data availability */  
11}  
12  
13 char get(buffer_t *pool) {  
14     P(&pool->data); /* block iff buffer is empty: data = 0 */  
15     char item = pool->ring[FAA(&pool->out, 1) % 64];  
16     V(&pool->free); /* signal buffer-place availability */  
17     return item;  
18 }
```

- ist der Puffer voll, wartet der Produzent in Z. 8 auf den Konsumenten — der in Z. 14 auf den Produzenten wartet, wenn der Puffer leer ist
- diesbezügliche Signalisierungen (Z. 10 und 16) setzen die Prozesse fort



© wosch

SP (WS 2019/20, C – X.3)

2.2 Semaphor – Anwendung

X.3/8

- **mehrseitige Synchronisation** (Beispielvorlage, vgl. auch [7, S. 34]):

```

1 int FAA(int_t *ref, int val) {
2     P(&ref->mutex);
3     int aux = ref->value;      9  typedef struct {
4     ref->value += val;        10  int value;           /* data */
5     V(&ref->mutex);          11  semaphore_t mutex; /* lock */
6     12  } int_t;
7
8     return aux;
9 }
```

- im Anwendungszenario (S. 8) können Produzenten und Konsumenten gleichzeitig auf die Indexvariablen (in, out) zugreifen
 - und zwar wann immer der Puffer nicht voll bzw. nicht leer ist
- die gleichzeitigen Zugriffe müssen koordiniert erfolgen, um die konsistente Werteveränderung der Indexvariablen zuzusichern
 - dazu kommt **wechselseitiger Ausschluss** (*mutual exclusion*) zum Einsatz
- da die Zugriffszeitpunkte der beteiligten Prozesse unbekannt ist, ist jeder dieser Prozesse in Z. 2 (d.h., im *P*) ggf. zum Warten verurteilt



Optionen für die Absicherung von *P* und *V*

- **pessimistischer Ansatz**, der annimmt, dass gleichzeitige Aktionen mit demselben Semaphor wahrscheinlich sind
 - i **wechselseitiger Ausschluss** wird die Aktionen nicht überlappen lassen, weder sich selbst noch gegenseitig
 - *P* und *V* sind durch eine gemeinsame „Sperre“ pro Semaphor zu schützen
 - ii Schlafenlegen eines Prozesses in *P* muss implizit die **Entsperrung** des kritischen Abschnitts zur Folge haben
 - sonst wird kein *V* die Ausführung vollenden können
 - als Folge werden in *P* schlafende Prozesse niemals aufgeweckt
 - iii Aufwecken von Prozessen in *V* sollte bedingt erfolgen, und zwar falls wenigstens ein Prozess in *P* schlafengelegt wurde
 - legt die Implementierung als **Monitor** nahe — dann scheidet mehrseitige Synchronisation des Monitors durch Semaphore [6, S. 7] aber aus ☺
 - gegebenenfalls sind andere Sperrtechniken erforderlich ↗ S. 24
- **optimistischer Ansatz**, der obige Annahme eher nicht trifft und sich **nichtblockierende Synchronisation** zu eigen macht
 - knifflig, ein Thema für das fortgeschrittene Studium [12]...



- Programme für *P* und *V* bilden **kritische Abschnitte**

```

1 void ewd_prolaag(int *sema) { 8  void ewd_verhoog(int *sema) {
2     atomic { 9      atomic {
3         *sema -= 1; 10      *sema += 1;
4         if (*sema < 0) 11      if (*sema <= 0)
5             await(sema); 12      cause(sema);
6     } 13  }
7 } 14 }
```

- i gleichzeitiges Ausführen von *P* kann mehr Prozesse passieren lassen, als es der Semaphorwert (*sema*) erlaubt
- ii gleichzeitiges Zählen kann Werte hinterlassen, die nicht der wirklichen Anzahl der ausgeführten Operationen (*P*, *V*) entsprechen
- iii gleichzeitiges Auswerten der Wartebedingung (*P*) und Hochzählen (*V*) kann das Schlafenlegen (*await*) von Prozessen bewirken, obwohl die Wartebedingung für sie schon nicht mehr gilt („*lost wake-up*“)



Monitor als Programmierkonvention³

```

1 void mps_prolaag(semaphore_t *sema) {
2     enter(&sema->lock.bolt); /* lock critical section */
3     sema->load -= 1;          /* decrease semaphore value */
4     if (sema->load < 0)      /* resource(s) exhausted? */
5         await(&sema->lock); /* fulfilled, wait outside */
6     leave(&sema->lock.bolt); /* unlock critical section */
7 }

8
9 void mps_verhoog(semaphore_t *sema) {
10    enter(&sema->lock.bolt); /* lock critical section */
11    sema->load += 1;          /* increase semaphore value */
12    if (sema->load <= 0)     /* any waiting process? */
13        cause(&sema->lock); /* notify exactly one process */
14    leave(&sema->lock.bolt); /* unlock critical section */
15 }
```

- wechselseitiger Ausschluss der Ausführung von „Monitorprozeduren“ sicherzustellen, ist eine **manuelle Maßnahme** geworden
 - Synchronisationsklammern (Z. 2–6 und 10–14) explizit machen
 - für die erforderliche mehr- und einseitige Synchronisation ist eine dem Semaphor eigene Datenstruktur hinzuzufügen ↗ lock-Attribut



Semaphordatentyp als Verbund

- ```
1 typedef volatile struct semaphore {
2 int load; /* # of allowed/waiting processes */
3 guard_t lock; /* synchronisation variables */
4 } semaphore_t;
```
- der als ganze Zahl ( $\mathbb{Z}$ , int) repräsentierte Semaphorwert ( $s$ , load) gibt verschiedene Interpretationen:
    - $\mathbb{N}^*$  ■ Anzahl der Prozesse, für die  $P$  keine Blockierung bewirkt
    - 0 ■ der nächste  $P$  aufrufende Prozess wird blockieren
    - $\mathbb{Z}_+$  ■ als Betrag  $|s|$  genommen die Anzahl der blockierten Prozesse
  - zum Schutz (guard) von  $P/V$  sowie um über Prozesse zu wachen, die auf das Ereignis der Ausführung von  $V$  warten, dient:

```
5 typedef volatile struct guard {
6 detent_t bolt; /* device to arrest concurrent processes */
7 event_t wait; /* per-event waitlist of processes */
8 } guard_t;
```

    - für die **Sperre** (detent) gibt es sehr unterschiedliche Implementierungen:
      - Unterdrückungstechniken (S. 24) oder Schlösser bzw. Schlossvariablen [9]

© wosch

SP (WS 2019/20, C – X.3 )

2.3 Semaphor – Implementierung

X.3/13

## Plausibilitätsprüfung

- seien  $P_p, P_v$  **gleichzeitige Prozesse**, die  $P$  bzw.  $V$  ausführen, mit:

```
1 #define P(s) mps_prolaag(s) /* acquire resource */
2 #define V(s) mps_verhoog(s) /* release resource */
```
- sei  $s$  Zeiger auf Exemplar  $x$  von `semaphore_t` mit der **Vorbelegung**:

```
1 semaphore_t x = { 1 }; /* one resource, unlocked */
```
- dann ist festzustellen:
  - $P_p$  kann sich weder mit sich selbst noch mit  $P_v$  überlappen
  - $P_v$  kann sich weder mit sich selbst noch mit  $P_p$  überlappen
- darüber hinaus gilt als **Randbedingung**:
  - $P_p$  legt sich außerhalb des kritischen Abschnitts schlafen, wenn die durch  $s$  kontrollierte Anzahl von Ressourcen erschöpft ist; es gilt  $s \in \mathbb{Z}^*$ 
    - andere Exemplare von  $P_p$  oder  $P_v$  können den kritischen Abschnitt betreten
    - jeder weitere  $P_p$  erniedrigt  $s$ , auch jeder dieser  $P_p$  legt sich schlafen
  - $P_v$  weckt höchstens ein Exemplar von  $P_p$  auf, der mit anderen Prozessen um Eintritt in den kritischen Abschnitt konkurriert; es gilt  $s \in \mathbb{Z}_-$ 
    - aber jeder **Ersteintritt** von  $P_p$  erniedrigt  $s$  und blockiert  $P_p$ ; es gilt  $s \in \mathbb{Z}^*$
    - nur der  $P_p$ , der als einziger aufgeweckt wurde, begeht den **Wiedereintritt**
      - nur für diesen  $P_p$  ist die **Wartebedingung aufgehoben**, er verlässt  $P(s)$ ...

© wosch

SP (WS 2019/20, C – X.3 )

2.3 Semaphor – Implementierung

X.3/14

## Prozessblockade im kritischen Abschnitt

- ein Prozess, für den eine Wartebedingung erfüllt ist, während er einen kritischen Abschnitt belegt, muss sich wie folgt verhalten:
  - den kritischen Abschnitt freigeben, ihn faktisch verlassen
  - blockieren, bis ein anderer Prozess die Wartebedingung aufheben konnte
  - sich um den Wiedereintritt in den kritischen Abschnitt bewerben
- auf den ersten Blick ist die damit verbundene **Ablaufunterbrechung und -fortsetzung** eines Prozesses einfach zu bewerkstelligen

```
1 void await(guard_t *lock) {
2 leave(&lock->bolt); /* unlock critical section */
3 sleep(&lock->wait); /* delay process, reschedule CPU */
4 enter(&lock->bolt); /* lock critical section */
5 }
6
7 void cause(guard_t *lock) {
8 process_t *next = elect(&lock->wait);
9 if (next) /* one process unblocked */
10 ready(next); /* schedule process */
11 }
```

- auf den zweiten Blick zeigt sich eine **wettlaufkritische Aktionsfolge**
- das **Aufwecksignal** für den Prozess kann verlorengehen (*lost wake-up*)

© wosch

SP (WS 2019/20, C – X.3 )

2.4 Semaphor – Ablaufunterbrechung

X.3/15

## Wettlaufkritische Aktionsfolge beim Warten *lost wake-up*

- Ausgangssituation:
  - $P$  hat die Wartebedingung für den Prozess festgestellt
  - $V$  wird die Aufhebung eben dieser Bedingung signalisieren
- seien  $P_p$  und  $P_v$  **gleichzeitige Prozesse**, die  $P$  bzw.  $V$  ausführen:
  - $P_p$  hat den kritischen Abschnitt freigegeben, ist noch nicht blockiert
    - $P_v$  betritt den kritischen Abschnitt, ruft cause auf (S. 12, Z. 13)
    - elect findet  $P_p$  nicht auf der Warteliste (S. 15, Z. 8–9)
    - das Signal zur Aufhebung der Wartebindung erreicht  $P_p$  nicht
  - $P_p$  legt sich schlafen, wird der Warteliste hinzugefügt und blockiert
    - betritt niemand mehr den kritischen Abschnitt, blockiert  $P_p$  ewig
- die Lösung des Problems findet sich in den Aktionen, um  $P_p$  vom Zustand „laufend“ in den Zustand „blockiert“ zu überführen
  - für diese Überführung inkl. Wartelistenvermerk sorgt sleep...

© wosch

SP (WS 2019/20, C – X.3 )

2.4 Semaphor – Ablaufunterbrechung

X.3/16

## Wartelistenvermerk und Zustandsübergang

- **Schlafenlegen** eines Prozesses umfasst zwei wichtige Hauptschritte:
  - ii den aktuellen Prozess als „blockiert“ und für die Warteliste vermerken
  - ii einen anderen Prozess auswählen und diesem den Prozessor zuteilen

```
1 void sleep(event_t *wait) {
2 allot(wait); /* register that process will block */
3 block(); /* delay process, reschedule CPU */
4 }
```

- sleep aufbrechen und den mit allot gemeinten Wartelistenvermerk in den kritischen Abschnitt „hochziehen“:

```
1 void await(guard_t *lock) {
2 allot(&lock->wait); /* register that process will block */
3 leave(&lock->bolt); /* unlock critical section */
4 block(); /* delay process, reschedule CPU */
5 enter(&lock->bolt); /* lock critical section */
6 }
```

- damit kann  $P_p$  auf der Warteliste finden (S. 15, Z. 8–9)
- $P_p$  wird in den Zustand „bereit“ überführt (S. 15, Z. 10)
- woraufhin block erkennt, dass  $P_p$  nicht (mehr) zu blockieren ist

© wosch

SP (WS 2019/20, C – X.3 )

2.4 Semaphor – Ablaufunterbrechung

X.3/17

## Semaphor v. Mutex I

Konzeptebene

### Hinweis (Informatikfolklore)

Ein Semaphor kann von jedem Prozess freigegeben werden.

- diese Feststellung wird oft als Nachteil vorgebracht, jedoch sind dabei die Semaphorarten (allgemein, binär) zu unterscheiden
  - strenggenommen ist sie eine **Anforderung** für den allgemeinen Semaphor und lediglich eine **Option** für den binären Semaphor
  - ein binärer Semaphor schützt einen kritischen Abschnitt, wobei eben zu differenzieren ist, ob darin ein **Prozesswechsel** geschieht oder nicht
    - ⇒ ohne } muss { derselbe } Prozess den Semaphor freigeben
    - mit } muss { ein anderer }

### Hinweis (Informatikfolklore)

Ein Mutex kann nur von dem Besitzerprozess freigegeben werden.

- diese Feststellung wird oft als Vorteil vorgebracht, ist jedoch nur auf den binären Semaphor ausgerichtet
  - nämlich zum Schutz eines kritischen Abschnitts ohne Prozesswechsel!

© wosch

SP (WS 2019/20, C – X.3 )

3.1 Mutex – Abgrenzung

X.3/19

## Gliederung

Einführung

Semaphor

Definition

Anwendung

Implementierung

Ablaufunterbrechung

Mutex

Abgrenzung

Implementierung

Sperre

Grundsätzliches

Varianten

Zusammenfassung

© wosch

SP (WS 2019/20, C – X.3 )

3. Mutex

X.3/18

## Semaphor v. Mutex II

Technikebene

### Hinweis

Prüfung der **Berechtigung** zur Freigabe eines kritischen Abschnitts (KA) ist ungeeignet für einen allgemeinen Semaphor, optional für einen binären Semaphor und notwendig für einen Mutex.

- **notwendig** ■ ein **Mutex** sichert zu, dass die Freigabe von KA nur für den Prozess gelingen kann, der KA zuvor erworben hatte
  - durch Verwendung eines binären Semaphors, Erfassung und Überprüfung des Besitzrechts für KA (vgl. S. 21)

- **ungeeignet** ■  $P$  und  $V$  mit demselben **allgemeinen Semaphor** muss für verschiedene Prozesse möglich sein

- einseitige Synchronisation: Konsumenten und Produzenten

- optional ■ grundsätzlich lässt sich ein **binärer Semaphor** durch einen allgemeinen Semaphor  $S$  repräsentieren, wenn  $S \leq 1$ 
  - ungeeignet zum Schutz eines KA mit Prozesswechsel

- bei **unberechtigter Freigabe** sollte der Prozess abgebrochen werden
  - im privilegierten Modus ist das Rechensystem anzuhalten...

© wosch

SP (WS 2019/20, C – X.3 )

3.1 Mutex – Abgrenzung

X.3/20

## Spezialisierung eines binären Semaphors

- ein Mutex benutzt einen binären Semaphore, ersetzt ihn jedoch nicht
  - die Mutex-Datenstruktur setzt sich aus zwei Komponenten zusammen:
    - i einem binären Semaphore zum Schutz eines kritischen Abschnitts *und*
    - ii einer Handhabe zur eindeutigen Identifizierung eines Prozesses<sup>4</sup>
  - ausgehend davon seien die beiden folgenden Operationen definiert:
    - acquire - vollzieht *P* und registriert den aktuellen Prozess als Eigentümer
    - release - zeigt eine Ausnahme an, wenn der Prozess nicht Eigentümer ist
      - löscht ansonsten den Eigentümereintrag und vollzieht *V*
- ein dazu korrespondierender **Datentyp** kann wie folgt ausgelegt sein:

```
1 typedef volatile struct mutex {
2 semaphore_t sema; /* binary semaphore */
3 process_t *link; /* owning process or 0 */
4 } mutex_t;
```

<sup>4</sup>Auf Kernebene ist diese Handhabe der Zeiger zu einem Prozesskontrollblock, auf Benutzerebene ist sie die Prozessidentifikation.

## Gliederung

Einführung

Semaphor

Definition

Anwendung

Implementierung

Ablaufunterbrechung

Mutex

Abgrenzung

Implementierung

Sperre

Grundsätzliches

Varianten

Zusammenfassung



## Erwerben und freigeben eines Mutex

```
1 extern void panic(char*) __attribute__((noreturn));
2
3 void acquire(mutex_t *mutex) {
4 P(&mutex->sema); /* lockout */
5 mutex->link = being(ONESELF); /* register owner */
6 }
7
8 void release(mutex_t *mutex) {
9 if (mutex->link != being(ONESELF)) /* it's not me! */
10 panic("unauthorised release of mutex");
11
12 mutex->link = 0; /* deregister owner */
13 V(&mutex->sema); /* unblock */
14 }
```

- die **unberechtigte Freigabe** eines Mutex ist eine sehr **ernste Sache**
  - das nichtsequentielle Programm enthält einen **Softwarefehler** (*bug*)
  - Fehlerkode liefern ist keine Option, da seine Behandlung nicht sicher ist
  - anderes als eine **nichtmaskierbare Ausnahme** anzuzeigen, ist fraglich...



## Prozessauslösung verhindern

Holzhammermethode...

- gleichzeitigen Prozessen vorbeugen dadurch, dass der **Mechanismus** für ihre Auslösung zeitweilig außer Kraft gesetzt ist
- **Unterbrechung**
  - Ursprung unvorhersehbarer gleichzeitiger Abläufe
  - asynchron zum aktuellen Prozess *und* Betriebssystem
  - *first-level interrupt handler* (FLIH)
- **Fortsetzung**
  - anschließender Teil des FLIH, synchron zum Systemkern
  - *second-level interrupt handler* (SLIH)
- **Verdrängung**
  - anschließender Teil eines SLIH, synchron zum Planer
  - Aktionsfolge für die präemptive Prozessumschaltung
- damit werden immer auch Prozesse ausgesperrt, die überhaupt nicht in Konflikt mit dem aktuellen Prozess geraten werden ☺
  - kausal unabhängige gleichzeitige Abläufe werden unnötig unterbunden
  - unkritische Parallelität wird eingeschränkt, Leistungsfähigkeit beschnitten
- darüber hinaus: diese Techniken greifen nur prozessor(kern)lokal, sind **ungeeignet für** ein-, mehr- oder vielkernige **Multiprozessorsysteme**
  - für letztere ist auf Schlösser bzw. Schlossvariablen zurückzugreifen [9]



```

1 inline void enter(detent_t *nest) {
2 if (nest) { /* save contents of PSW */
3 ...
4 /* vgl. S. 36 */
5 }
6 asm volatile ("cli" : : : "cc"); /* disable interrupts */
7
8 inline void leave(detent_t *nest) {
9 if (!nest) /* enable interrupts */
10 asm volatile ("sti" : : : "cc");
11 else { /* restore contents of PSW */
12 ...
13 /* vgl. S. 36 */
14 }
15
16 typedef volatile struct detent {
17 psw_t flags; /* process status word */
18 } detent_t;
19
20 ■ dazu muss der Inhalt des Prozessorstatuswortes invariant gehalten werden

```



- **Verschachtelungen** erfordern bei Entsperrung die Wiederherstellung des Sperrzustands, der im Moment der Sperrung galt

```

15 typedef volatile struct detent {
16 psw_t flags; /* process status word */
17 } detent_t;
18
19 ■ dazu muss der Inhalt des Prozessorstatuswortes invariant gehalten werden

```

```

1 typedef volatile struct detent {
2 int flag; /* saved SLIH lock status */
3 } detent_t;
4
5 ■ der Sperre zur Unterbrechungsanforderung (interrupt request) sehr
6 ähnlich, nur wird ein Software- und nicht Hardwaresignal geblockt
7
8 void enter(detent_t *gate) {
9 gate->flag = avert(&slih); /* disable SLIH */
10
11 void leave(detent_t *gate) {
12 if (gate->flag == 0) { /* nested? */
13 grant(&slih); /* no, enable SLIH */
14 if (order(&slih)) /* SLIH pending? */
15 flush(&slih); /* yes, catch up... */
16 }
17
18 ■ während die Ausführung eines SLIH unterbunden ist, kann sein FLIH
19 allerdings zur Ausführung kommen
20 ■ der vom FLIH ausgelöste SLIH kommt ggf. in eine Warteschlange
21 ■ beim Verlassen des gesperrten Abschnitts wird diese abgebaut (Z. 11–12)

```



- im Grunde genommen die Spezialisierung des eben (S. 26) skizzierten Ansatzes, den SLIH zeitweilig zu maskieren
  - nicht jeder SLIH wird verzögert, sondern nur die zum Planer führenden
  - also jeder SLIH, der die **Umlaufplanung** (*rescheduling*) von Prozessen auslöst
- bei aktiver Sperre wird der aktuelle Prozess zwar unterbrochen, ihm wird jedoch nicht der Prozessor entzogen
  - der gesperrte Abschnitt wird auch als **nichtunterbrechender kritischer Abschnitt** (*non-preemptive critical section*, NPCS) bezeichnet
  - Entzug des Prozessors ist erst nach Verlassen dieses Abschnitts möglich
  - dabei muss es nicht wirklich zur Verzögerung der Prozesseinplanung kommen, wohl aber zu der der **Prozesseinlastung**
    - der Planer reiht den bereitgestellten Prozess strategiegemäß ein, wird den **Abfertiger** (*dispatcher*) ggf. zum Prozesswechsel auffordern
      - wenn die Umlaufplanung feststellt, den aktuellen Prozess wegschalten zu müssen
    - ist die Sperre im Moment der Aufforderung aktiv, wird der Aufruf an den Abfertiger jedoch zurückgestellt (*deferred procedure call*, DPC [1])
    - zurückgestellte Aufrufe werden beim Verlassen des gesperrten Abschnitts von dem dann aktuellen Prozess wieder aufgenommen und durchgeführt



Einführung  
Semaphore  
    Definition  
    Anwendung  
    Implementierung  
    Ablaufunterbrechung

Mutex  
    Abgrenzung  
    Implementierung

Sperre  
    Grundsätzliches  
    Varianten

Zusammenfassung



- Synchronisation in der Maschinenprogrammebene kann auf Konzepte von Betriebssystemen zurückgreifen
  - die den Zeitpunkt von Einplanung oder Einlastung gezielt beeinflussen
  - die Prozesse kontrolliert schlafen legen und wieder aufwecken
- typische ELOP dieser Ebene ist der **Semaphor**, ein Verbundatentyp bestehend aus Zähl- und Ereignisvariable
  - unterschieden wird zwischen binärem und allgemeinem Semaphor
  - seine Primitiven (*P*, *V*) bilden logisch bzw. physisch atomare Aktionen
- **Atomarität** der Semaphorprimitiven ist durch Techniken zu erreichen, die hierarchisch tiefer (d.h., auf Befehlssatzebene) angesiedelt sind
  - **Sperren** (physisch) von Unterbrechungen, Fortsetzungen, Verdrängungen
  - **Schlösser** (physisch) oder **nichtblockierende Synchronisation** (logisch)
- nicht zu vergessen der **Mutex**: eine Semaphorspezialisierung, die die Eigentümerschaft bei Freigabe prüft und letztere bedingt zulässt
  - der Mutex benutzt einen binären Semaphor, ersetzt ihn jedoch nicht
  - denn uneingeschränkte Semaphorfreigabe ist ein Merkmal, kein Makel



## Literaturverzeichnis II

- [5] HOARE, C. A. R.:  
Communicating Sequential Processes.  
In: *Communications of the ACM* 21 (1978), Nr. 8, S. 666–677
- [6] KLEINÖDER, J. ; SCHRÖDER-PREIKSCHAT, W. :  
Monitor.  
In: [10], Kapitel 10.2
- [7] KLEINÖDER, J. ; SCHRÖDER-PREIKSCHAT, W. :  
Nichtsequentialität.  
In: [10], Kapitel 10.1
- [8] KLEINÖDER, J. ; SCHRÖDER-PREIKSCHAT, W. :  
Prozesse.  
In: [10], Kapitel 6.1
- [9] KLEINÖDER, J. ; SCHRÖDER-PREIKSCHAT, W. :  
Schlösser und Spezialbefehle.  
In: [10], Kapitel 10.4
- [10] KLEINÖDER, J. ; SCHRÖDER-PREIKSCHAT, W. ; LEHRSTUHL INFORMATIK 4 (Hrsg.):  
*Systemprogrammierung*.  
FAU Erlangen-Nürnberg, 2015 (Vorlesungsfolien)



## Literaturverzeichnis I

- [1] BAKER, A. ; LOZANO, J. :  
Deferred Procedure Calls.  
In: *Windows 2000 Device Driver Book: A Guide for Programmers*.  
Prentice Hall, 2000
- [2] DIJKSTRA, E. W.:  
Over seinpalen / Technische Universiteit Eindhoven.  
Eindhoven, The Netherlands, 1964 ca. (EWD-74). –  
Manuskript. –  
(dt.) Über Signalmasten
- [3] 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)
- [4] DIJKSTRA, E. W.:  
The Structure of the “THE”-Multiprogramming System.  
In: *Communications of the ACM* 11 (1968), Mai, Nr. 5, S. 341–346



## Literaturverzeichnis III

- [11] KLEINÖDER, J. ; SCHRÖDER-PREIKSCHAT, W. :  
Virtuelle Maschinen.  
In: [10], Kapitel 5.1
- [12] SCHRÖDER-PREIKSCHAT, W. ; LEHRSTUHL INFORMATIK 4 (Hrsg.):  
*Concurrent Systems — Nebenläufige Systeme*.  
FAU Erlangen-Nürnberg, 2014 (Vorlesungsfolien)



## Semaphormonitor

Fiktion: Concurrent C++

```
1 template<monitor = signal urgent wait>
2 class Semaphore {
3 int load; // # of allowed/waiting processes
4 condition free; // to block/unblock processes
5
6 atomic:
7 Semaphore(int seed = 0) { load = seed; }
8
9 void prolaag() {
10 load -= 1;
11 if (load < 0)
12 free.wait();
13 }
14
15 void verhoog() {
16 load += 1;
17 if (load <= 0)
18 free.signal();
19 }
20};
21
22 void P(Semaphore& sema) {
23 sema.prolaag();
24
25 void V(Semaphore& sema) {
26 sema.verhoog();
27 }
```

- nur der Hoare'sche Monitor (*signal and urgent wait*) lässt hier die Formulierung von *prolaag()* zu, wie vorher (S. 10) skizziert
- Annahme ist die sofortige Wiederaufnahme des signalisierten Prozesses

© wosch

SP (WS 2019/20, C – X.3 )

6.1 Anhang – Semaphor

X.3/33

## Trennung der Zuständigkeiten

separation of concerns

```
1 void mps_prolaag(semaphore_t *sema) {
2 enter(&sema->lock.bolt); /* lock critical section */
3 while (--sema->load < 0) /* resource(s) exhausted? */
4 await(&sema->lock); /* fulfilled, wait outside */
5 leave(&sema->lock.bolt); /* unlock critical section */
6 }
7
8 void mps_verhoog(semaphore_t *sema) {
9 enter(&sema->lock.bolt); /* lock critical section */
10 if (sema->load >= 0) /* any waiting process? */
11 sema->load += 1; /* no, increase sema. value */
12 else {
13 sema->load = 1; /* yes, enable at most one */
14 rouse(&sema->lock); /* but notify all processes */
15 }
16 leave(&sema->lock.bolt); /* unlock critical section */
17 }
```

- der Semaphor entscheidet nur noch, wann ein Prozess zu deblockieren ist, jedoch überlässt es dem **Planer**, welcher dies sein wird
  - *V* weckt alle wartende Prozesse auf, lässt aber nur einen davon aus *P*
  - *P* zwingt erwachte Prozess zur Neuauswertung der Wartebedingung

© wosch

SP (WS 2019/20, C – X.3 )

6.1 Anhang – Semaphor

X.3/35

## Es ist nicht alles Gold, was glänzt...

- so naheliegend die Implementierung eines Semaphors als Monitor ist, sie muss sich einigen Herausforderungen stellen:
  - keine der heute gebräuchlichen Systemprogrammiersprachen hat den Begriff „Monitor“ als Sprachkonstrukt integriert, auch Java nicht:
    - *synchronized* – wechselseitiger Ausschluss
    - Methoden- oder Basisblockausführung
    - *wait* – blockiert den Prozess, hebt umfassendes *synchronized* auf
    - bewirbt umfassendes *synchronized* bei Wiederaufnahme
    - *notify* – deblockiert genau einen wartenden Prozess
    - *notifyAll* – deblockiert alle wartende Prozesse
  - darüberhinaus ist (Standard-) Java keine Systemprogrammiersprache, die zur Implementierung von hardwarenahen Programmen geeignet ist
- der Hoare'sche Monitor hat einen recht hohen Laufzeitaufwand und, bis auf CSP [5], keine praktische Umsetzung erfahren
  - vergleichsweise hohe Anzahl von Prozess-/Kontextwechsel
  - Atomarität der Aktionsfolgen *signal* → *wait* und wieder zurück
- ein Monitor abstrahiert vom Semaphor, um Fehler beim Umgang mit Semaphore zu vermeiden → **Bruch im Abstraktionsprinzip...**

© wosch

SP (WS 2019/20, C – X.3 )

6.1 Anhang – Semaphor

X.3/34

## FLIH verschachtelungsfähig maskieren

x86

```
1 inline void enter(detent_t *nest) {
2 if (nest) { /* save contents of PSW */
3 asm volatile (
4 "pushf\n\t" /* read from flags register */
5 "popl %0" /* save to prototype */
6 : "=m" (nest->flags) :
7 : "memory", "cc");
8 }
9 asm volatile ("cli" : : : "cc"); /* disable interrupts */
10 }
11
12 inline void leave(detent_t *nest) {
13 if (!nest) /* enable interrupts */
14 asm volatile ("sti" : : : "cc");
15 else {
16 asm volatile (
17 "pushl %0\n\t" /* read from prototype */
18 "popf" /* write to flags register */
19 : : "m" (nest->flags)
20 : "memory", "cc");
21 }
22 }
```

© wosch

SP (WS 2019/20, C – X.3 )

6.2 Anhang – Unterbrechungssperre

X.3/36

## Zurückgestellte Prozeduraufrufe

```
1 typedef struct sentry {
2 int lock; /* activiy state */
3 queue_t wait; /* deferred procedure calls */
4 } sentry_t;
5
6 inline int avert(sentry_t *gate) {
7 return FAS(&gate->lock, 1); /* try to activate section */
8 }
9
10 inline void grant(sentry_t *gate) {
11 gate->lock = 0; /* deactivate section */
12 }
13
14 inline chain_t *order(sentry_t *gate) {
15 return gate->wait.head.link; /* next DPC to be processed */
16 }
17
18 extern void flush(sentry_t *); /* process all pending DPCs */
19 extern sentry_t slih; /* kernel-global guardian */
```



## Atomare Lese-/Schreibaktion

```
1 inline int FAS(int *ref, int val) {
2 int aux;
3 asm volatile(
4 "xchgl %0, %1" /* atomic read-write action */
5 : "=q" (aux), "=m" (*ref)
6 : "r" (val), "m" (*ref)
7 : "memory", "cc");
8 return aux;
9 }
```

■ darauf und auf die Implementierung eines DPC (S. 37) basierende Kompilierung<sup>5</sup> von enter (S. 26) liefert:

```
10 _enter:
11 movl 4(%esp), %eax # get pointer to detent flag
12 movl $1, %ecx # get target activity state
13 ## InlineAsm Start
14 xchgl %ecx, _slih # exchange with sentry lock
15 ## InlineAsm End
16 movl %ecx, (%eax) # save former activity state
17 retl
```

<sup>5</sup>gcc -O3 -m32 -static -fomit-frame-pointer -S

