F.10 Koordinierungsmechanismen

1 UNIX — Semaphore

- Atomare Operationen auf einem Semaphorvektor
  Realisierung folgender theoretischer Konzepte:
  - ◆ Vektoradditionssystem (VAS)
    - ➥ mehrere Semaphorvariablen können in einer atomaren Operation individuell dekrementiert (P-Operation) bzw. inkrementiert (V-Operation) werden
  - ◆ up/down-System - ähnliche Funktionalität
    (Lösung des 2. Leser/Schreiber-Problems)
    - ➥ in einer atomaren Operation können mehrere Variablen auf 0 getestet werden und im Erfolgsfall werden Variablen des Semaphorvektors modifiziert
- Include-Dateien: `<sys/types.h>` `<sys/ipc.h>` `<sys/sem.h>`

1 UNIX — Semaphore (2)

- Systemschnittstelle:

semget(2)	Erzeugen eines neuen Semaphorverktors, bzw. Zugriff zu einem bestehenden Semaphor besorgen
semctl(2)	Statuskontrolle, Initialisierung und Löschen
semop(2)	Semaphor-Operationen

- Semaphor-Operation
  - ➥ **Aufruf:**

```
int semid;
int nsops = N;
struct sembuf sbuf[N];
...
semop(semid, sbuf, nsops);
```


1 UNIX — Semaphore (3)

- Eine Semaphor-Operation ist aus mehreren Einzeloperationen zusammengesetzt, jede Einzeloperation ist durch eine Struktur ***sembuf*** beschrieben:
  - ➥ Struktur:

```
struct sembuf {
    short sem_num;          /* Sem. Nummer
*/
    short sem_op;           /* Operation
*/
```

sem_num	Nummer der Semaphorkomponente auf die sich die Operation bezieht
sem_op < 0	P-Operation (VAS)
sem_op > 0	V-Operation (VAS)
sem_op = 0	Test auf 0 (2. Leser/Schreiber-Problem)

- Eine Semaphor-Operation blockiert immer dann als ganzes, wenn eine der Einzeloperationen blockiert!

1 UNIX — Semaphore (4)

★ **UNIX-Semaphore als Vektoradditionssystem**

- ◆ vgl. F. Hofmann: Betriebssysteme: Grundkonzepte und Modellvorstellungen; 2. Aufl., Kap. 2.4, S. 73

  Vektoraddition: $e \in \mathbb{Z}^{m}$, Semaphorvektor $S \in \mathbb{Z}^{m}$,
  $\Phi: {}'S + e \geq 0 \qquad \Theta: S = {}'S + e$

- **Realisierung**:
  Eine Operation e auf dem Semaphorvektor wird durch ein Feld von sembuf-Strukturen beschrieben:

  $\forall\, e_i \in e \wedge e_i \neq 0: \exists$ sembuf (sembuf.sum_num = i $\wedge$ sembuf.sem_op = e_i)
  $\forall\, e_i \in e \wedge e_i = 0: \nexists$ sembuf (sembuf.sum_num = i)

1 UNIX — Semaphore (5)

... UNIX-Semaphore als Vektoradditionssystem

sem_op < 0 P-Teiloperation
sem_op wird vom Wert der Semaphorkomponente abgezogen, wenn diese dadurch nicht kleiner 0 wird.
Gesamtoperation blockiert oder liefert Fehler (je nach ***flag***) wenn nicht möglich

sem_op > 0 V-Teiloperation
sem_op wird zum Wert der Semaphorkomponente addiert

kein sembuf für Semaphorkomponenten, die unverändert bleiben sollen!

1 UNIX — Semaphore (6)

★ **2. Leser-Schreiber-Problem**

- vgl. F. Hofmann: Betriebssysteme: Grundkonzepte und Modellvorstellungen; 2. Aufl., Kap. 2.4, S. 76

■ UNIX-Sempahore realisieren kein echtes *up/down*-System, entscheidend für das 2. Leser-Schreiber-Problem ist aber:

- ➥ atomarer Test mehrerer Variablen, verbunden mit Modifikation anderer Variablen
- ➥ beliebig viele P-Operationen ermöglichen und so lange warten können, bis die entsprechenden V-Operationen komplett durchgeführt wurden
  - ➤ alle anderen Koordinierungssysteme können nur von einem endlichen Wert bis 0 belegen!

1 UNIX — Semaphore (7)

... 2. Leser-Schreiber-Problem

■ Lösung mit UNIX-Semaphoren

- ➥ Semaphorvektor mit 3 Komponenten
  1. gegenseitiger Ausschluß zwischen Schreibern
  2. Zähler für Leser
  3. Anmeldung der Schreiber

  Vorbelegung: $\begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$

- ➥ **Leser**

  semop $\begin{pmatrix} - \\ 1 \\ 0 \end{pmatrix}$; *lesen*; semop $\begin{pmatrix} - \\ -1 \\ - \end{pmatrix}$;

- ➥ **Schreiber**

  semop $\begin{pmatrix} - \\ - \\ 1 \end{pmatrix}$; semop $\begin{pmatrix} -1 \\ 0 \\ - \end{pmatrix}$; *schreiben*; semop $\begin{pmatrix} 1 \\ - \\ -1 \end{pmatrix}$;

  – bedeutet, daß bei semop für die entsprechende Vektorkomponente keine sembuf-Struktur übergeben wird!

2 Pthreads-Koordinierung

■ UNIX-Semaphore für Koordinierung von leichtgewichtigten Prozessen zu teuer

- Implementierung durch den Systemkern
- komplexe Datenstrukturen

■ Bei Koordinierung von Threads reichen meist einfache ***mutex***-Semaphore

- gewartet wird durch Blockieren des Threads oder durch *busy wait (Spinlock)*

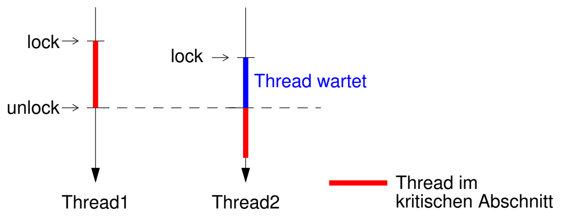
■ Komplexere Semaphore können alleine mit Mutexes nicht implementiert werden

- ➥ Problem:
  - Ein Mutex sperrt die Datenstruktur des komplexen Semaphors
  - Der Zustand der Datenstruktur erlaubt die Operation nicht
  - Blockieren an einem weiteren Mutex kann zu Verklemmungen führen
- ➥ Lösung: mutex in Verbindung mit sleep/wakeup-Mechanismus
- ➥ **Condition Variables**

2 Pthreads-Koordinierung (2)

★ **Mutexes**

■ Koordinierung von kritischen Abschnitten



2 Pthreads-Koordinierung (3)

... **Mutexes (2)**

■ Realisierung:

- ◆ Implementierung weitgehend im Anwendungsmodus
- ◆ Voraussetzung: atomarer Maschinenbefehl zum Testen und Belegen der Semaphorvariablen
  (z. B. Test-and-Set oder Compare-and-Swap)
- ◆ Ist Semaphor frei kann, der Thread den kritischen Abschnitt ohne Verzögerung betreten
- ◆ Ist Sempahor belegt kann alternativ
  - – der Thread den Prozessor aufgeben (thread_yield)
    (normale Pthreads-Vorgehensweise)
  - – der Thread aktiv auf die Freigabe warten (Spin)
    (Option z. B. bei der KSR Pthreads-Implementierung)
- ◆ Bei der unlock-Operation werden blockierte Threads aufgeweckt (thread_resume)

2 Pthreads-Koordinierung (4)

... **Mutexes (3)**

■ Schnittstelle

◆ Mutex erzeugen

```
pthread_mutex_t m1
s = pthread_mutex_init(&m1, pthread_mutexattr_default);
```

◆ Lock & unlock

```
s = pthread_mutex_lock(&m1);
... kritischer Abschnitt
s = pthread_mutex_unlock(&m1);
```

◆ weitere Funktionen

pthread_mutex_trylock: wie lock, blockiert aber nicht wenn krit. Abschnitt belegt ist

pthread_mutex_getowner: liefert pthread-ID des aktuellen lock-Besitzers

pthread_mutex_destroy: löscht Mutex

2 Pthreads-Koordinierung (5)

★ **Condition Variables**

■ Mechanismus zum Blockieren (mit gleichzeiter Freigabe des aktuellen kritischen Abschnitts) und Aufwecken (mit neuem Betreten des kritischen Abschnitts) von Threads