Systemprogrammierung

Prozesssynchronisation: Hochsprachenebene

Wolfgang Schröder-Preikschat

Lehrstuhl Informatik 4

16. November 2011

Gliederung

Synchronisierter abstrakter Datentyp: Monitor

Datentyp mit impliziten Synchronisationseigenschaften [2, 3]:

mehrseitige Synchronisation an der Monitorschnittstelle
- wechselseitiger Ausschluss der Ausführung exportierter Prozeduren
- realisiert mittels **Schlossvariablen** oder vorzugsweise **Semaphore**

einseitige Synchronisation innerhalb des Monitors
- bei Bedarf, Bedingungssynchronisation abhängiger Prozesse
- vermöge **Bedingungsvariable** und zwei Elementaroperationen:

  wait blockiert einen Prozess auf das Eintreten eines Signals/einer Bedingung und gibt den Monitor implizit wieder frei

  signal zeigt das Eintreten eines Signals/einer Bedingung an und deblockiert (genau einen oder alle) darauf blockierte Prozesse

Sprachgestützter Ansatz

- Concurrent Pascal, PL/I, Mesa, . . . , Java

Monitor ≡ (eine auf ein Modul bezogene) Klasse

Kapselung (engl. *encapsulation*)
- von mehreren Prozessen gemeinsam bearbeitete Daten müssen, modulgleich, in Monitoren organisiert vorliegen
- als Konsequenz macht die Programmstruktur kritische Abschnitte explizit sichtbar

Datenabstraktion (engl. *information hiding*)
- wie ein Modul, so kapselt auch ein Monitor für mehrere Funktionen Wissen über gemeinsame Daten
- Auswirkungen lokaler Programmänderungen bleiben begrenzt

Bauplan (engl. *blueprint*)
- wie eine Klasse, so beschreibt ein Monitor für mehrere Exemplare seines Typs den **Zustand** und das **Verhalten**
- er ist eine gemeinsam benutzte Klasse (engl. *shared class*, [2])

Klassenkonzept erweitert um Synchronisationssemantik

Monitor ≡ implizit synchronisierte Klasse

Monitorprozeduren (engl. *monitor procedures*)

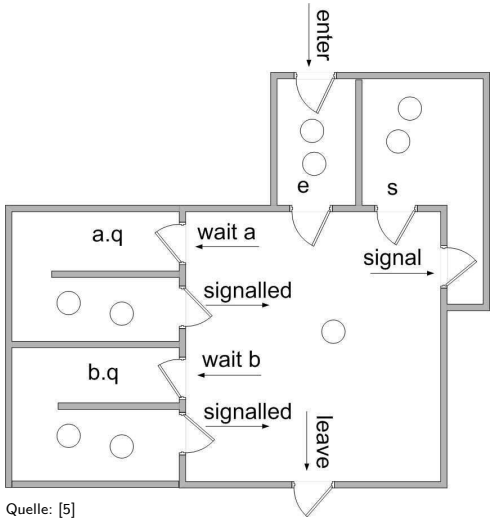
- schließen sich bei konkurrierenden Zugriffen durch mehrere Prozesse in ihrer Ausführung gegenseitig aus
  - der erfolgreiche Prozeduraufruf sperrt den Monitor
  - bei Prozedurrückkehr wird der Monitor wieder entsperrt
- repräsentieren per Definition kritische Abschnitte, deren Integrität vom Kompilierer garantiert wird
  - die „Klammerung“ kritischer Abschnitte erfolgt automatisch
  - der Kompilierer setzt die dafür notwendigen Steueranweisungen ab

Synchronisationsanweisungen

- sind Querschnittsbelang eines Monitors und nicht des gesamten nichtsequentiellen Programms
- sie liegen nicht quer über die ganze Software verstreut vor

Monitor mit beidseitig blockierenden Bedingungsvariablen

Hansen [2] und Hoare [3]



Quelle: [5]

Monitorwarteschlangen

- e der Zutrittsanforderer
- s der Signalgeber: optional
  - ggf. vereint mit *e*

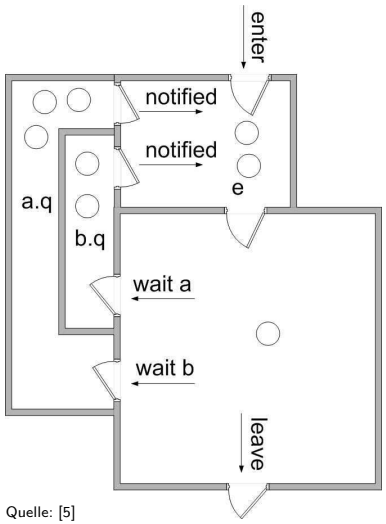
Ereigniswarteschlangen

- a.q für Bedingungsvariable *a*
- b.q für Bedingungsvariable *b*

- Signalgeber blockieren
  - warten außerhalb
  - verlassen den Monitor
- zieht Wiedereintritt nach

Monitor mit einseitig blockierenden Bedingungsvariablen

Mesa [4]



Quelle: [5]

Monitorwarteschlange

- e der Zutrittsanforderer *und* der signalisierten Prozesse

Ereigniswarteschlangen

- a.q für Bedingungsvariable *a*
- b.q für Bedingungsvariable *b*

- Signalgeber fahren fort
  - „Sammelaufruf“ möglich
  - $n > 1$ Ereignisse signalisierbar
- Signalnehmer starten erst nach Monitorfreigabe (*leave*)

Gliederung

Signalisierung einer Fortführungsbedingung erwarten: *wait*

Wartebedingung festlegen

Monitorfreigabe als notwendiger Seiteneffekt beim Warten [1]:

- andere Prozesse wären sonst an den Monitoreintritt gehindert
- als Folge könnte die zu erfüllende Bedingung nie erfüllt werden
- schlafende Prozesse würden nie mehr erwachen $\rightsquigarrow$ Verklemmung

Monitordaten sind in einem konsistenten Zustand zu hinterlassen

- andere Prozesse aktivieren den Monitor während der Blockadephase
- als Folge sind (je nach Funktion) Zustandsänderungen zu erwarten
- vor Eintritt in die Wartephase muss der Datenzustand konsistent sein

[1] Aktives Warten (engl. *busy waiting*) eines Prozesses, also ohne Prozessorabgabe, ist innerhalb eines Monitor logisch komplex und nicht nur dort leistungsmindernd.

Signalisierung einer Fortführungsbedingung: *signal*

Wartebedingung aufheben

Prozessblockaden in Bezug auf eine Wartebedingung werden aufgehoben

- im Falle wartender Prozesse sind als Anforderungen zwingend:
  - wenigstens ein Prozess deblockiert an der Bedingungsvariablen
  - höchstens ein Prozess rechnet nach der Operation im Monitor weiter
- erwartet kein Prozess ein Signal, ist die Operation wirkungslos
  - d.h., Signale dürfen in Bedingungsvariablen nicht gespeichert werden

Lösungsansätze hierzu sind z.T. von sehr unterschiedlicher Semantik

- das betrifft etwa die Anzahl der befreiten Prozesse:
  - alle auf die Bedingung wartenden oder genau nur einer
- gr. Unterschiede liegen auch in **Besitzwechsel** bzw. **Besitzwahrung**
  - „falsche Signalisierungen“ werden toleriert oder nicht

Besitzwechsel: *signal and (urgent) wait*

Signalisierender Prozess gibt die Kontrolle über den Monitor ab, wird inaktiv

alle das Ereignis erwartenden Prozesse befreien $\mapsto$ Hansen [1]

- alle Prozesse aus der Ereignis- in die Monitorwarteschlange bewegen
- bei Freigabe alle n Prozesse der Monitorwarteschlange freistellen
- $n-1$ Prozesse reihen sich erneut in die Monitorwarteschlange ein

höchstens einen das Ereignis erwartenden Prozess befreien $\mapsto$ Hoare [3]

- nur einen Prozess der Ereigniswarteschlange entnehmen (vgl. S. 12)
- den signalisierenden Prozess der Monitorwarteschlange zuführen
- direkt vom signalisierenden zum signalisierten Prozess wechseln

Hoare: Neuauswertung der Wartebedingung entfällt

- Fortführungsbedingung des signalisierten Prozesses ist garantiert
  - seit Signalisierung war kein anderer Prozess im Monitor drin
  - kein anderer Prozess konnte die Fortführungsbedingung entkräften
- der signalisierende Prozess bewirbt sich erneut um Monitorzutritt
  - „falsche Signalisierungen“ (vgl. S. 12) werden nicht toleriert

Besitzwahrung: *signal and continue*

Signalisierender Prozess behält die Kontrolle über den Monitor, bleibt aktiv

einen oder alle das Ereignis erwartenden Prozesse befreien $\mapsto$ Mesa [4]

- Prozess(e) aus der Ereignis- in die Monitorwarteschlange bewegen
- bei Freigabe nur einen Prozess der Monitorwarteschlange freistellen

Mesa/Hoare: Gefahr von Prioritätsverletzung [4]

- bedingt durch die Auswahlentscheidung, die festlegt, welcher Prozess freigestellt bzw. der Ereigniswarteschlange entnommen werden soll
- Interferenz mit der Prozesseinplanung ist vorzubeugen/zu vermeiden

Mesa/Hansen: Neuauswertung der Wartebedingung erforderlich

- Fortführungsbedingung des signalisierten Prozesses nicht garantiert
  - ein anderer Prozess kann den Monitor zwischenzeitlich betreten haben
- signalisierte Prozesse bewerben sich erneut um den Monitorzutritt
  - „falsche Signalisierungen“ (an den falschen Prozess) werden toleriert

Gliederung

Zwischenspeicher mit Pufferbegrenzung

Ein *bounded buffer* in „Concurrent C++“

```
class Ringbuffer {
    char     data[NDATA];
    unsigned nput, nget;
public:
    Ringbuffer ()     { nput = nget = 0; }
    char fetch ()     { return data[nget++ % NDATA]; }
    void store (char) { data[nput++ % NDATA] = item; }
};
```

```
monitor Buffer : private Ringbuffer {
    unsigned  free;
    condition null, full;
public:
    Buffer ()         { free = NDATA; }
    char fetch ();
    void store (char);
};
```

`monitor` wechselseitiger Ausschluss
- `Buffer::fetch()`
- `Buffer::store()`

`condition` Bedingungsvariable
- `null`: kein Platz
- `full`: $n \geq 1$ Daten

- `free` verwaltet den „Pegelstand“

Koordiniertes Leeren

Monitor im Stil von Hansen oder Mesa

```
char Buffer::fetch () {
    char item;
    while (free == NDATA) full.await();
    item = Ringbuffer::fetch();
    free++;
    null.signal();
    return item;
}
```

Bedingungsvariablen:

`full` erwartet einen Eintrag

`null` signalisiert freien Platz

Instanzvariable:

`free` verbucht freien Platz

Wartebedingung ist wiederholt zu überprüfen: `while`

- bewirbt signalisierte **Konsumenten** erneut um den Monitorzutritt
  - die Phase ab der Signalisierung von `full` durch den Produzenten bis zum Wiedereintritt des Konsumenten in den Monitor ist nebenläufig
  - der Puffer könnte zwischenzeitig geleert worden sein $\rightsquigarrow$ erneut warten
- toleriert (fehlerbedingte) falsche Signalisierungen von `full`

Koordiniertes Füllen

Monitor im Stil von Hansen oder Mesa

```
void Buffer::store (char item) {
    while (!free) null.await();
    Ringbuffer::store(item);
    free--;
    full.signal();
}
```

Bedingungsvariablen:

`null` erwartet freien Platz

`full` signalisiert einen Eintrag

Instanzvariable: `free` verbucht einen weiteren Puffereintrag

Wartebedingung ist wiederholt zu überprüfen: `while`

- bewirbt signalisierte **Produzenten** erneut um den Monitorzutritt
  - die Phase ab der Signalisierung von `null` durch den Konsumenten bis zum Wiedereintritt des Produzenten in den Monitor ist nebenläufig
  - der Puffer könnte zwischenzeitig gefüllt worden sein $\rightsquigarrow$ erneut warten
- toleriert (fehlerbedingte) falsche Signalisierungen von `null`

Gliederung

Monitorkonzepte im Vergleich

Hansen/Mesa vs. Hoare

Hansen und Mesa

```
while (free == NDATA) full.await();
while (!free) null.await();
```

Prozessen wird nicht garantiert, dass nach ihrer Signalisierung die Fortführungsbedingung gilt

- andere Prozesse können den Monitor betreten haben
- Wartebedingung erneut prüfen
- evtl. falsche Signalisierungen werden toleriert

Hoare

```
if (free == NDATA) full.await();
if (!free) null.await();
```

Prozessen wird garantiert, dass nach ihrer Signalisierung die Fortführungsbedingung gilt

- kein anderer Prozess konnte den Monitor betreten haben
- Wartebedingung einmal prüfen
- evtl. falsche Signalisierungen werden nicht toleriert

Resümee

- ein Monitor ist ein **ADT** mit impliziten Synchronisationseigenschaften
  - mehrseitige Synchronisation von Monitorprozeduren
  - einseitige Synchronisation durch Bedingungsvariablen
- die **Architektur** lässt verschiedene Ausführungsarten zu
  - Monitor mit beid- oder einseitig blockierenden Bedingungsvariablen
- Unterschiede liegen vor allem in der **Semantik der Signalisierung**:
  - wirkt blockierend (Hansen, Hoare) oder nichtblockierend (Mesa) für den ein Ereignis signalisierenden Prozess
  - stellt einen (Hoare, Mesa) oder alle (Hansen, Mesa) auf ein Ereignis wartende Prozesse frei
  - die Fortführungsbedingung für den jeweils signalisierten Prozess wird garantiert (Hoare) oder nicht garantiert (Hansen, Mesa)
  - erfordert (Hansen, Mesa) oder erfordert nicht (Hoare) die erneute Auswertung der Wartebedingung bei Fortführung
  - ist falschen Signalisierungen gegenüber tolerant (Hansen, Mesa) oder intolerant (Hoare)

Literaturverzeichnis

[1] Hansen, P. B.:
Structured Multiprogramming.
In: *Communications of the ACM* 15 (1972), Jul., Nr. 7, S. 574–578

[2] Hansen, P. B.:
Operating System Principles.
Prentice Hall International, 1973

[3] Hoare, C. A. R.:
Monitors: An Operating System Structuring Concept.
In: *Communications of the ACM* 17 (1974), Okt., Nr. 10, S. 549–557

[4] 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

[5] Wikipedia:
Monitor (synchronization).
http://en.wikipedia.org/wiki/Monitor_(synchronization), Dez. 2010