

## Monitor (engl. *monitor*)

Ein **synchronisierter abstrakter Datentyp**, d.h., ein ADT mit impliziten Synchronisationseigenschaften [66, 67]:

**mehrseitige Synchronisation** an der Monitorschnittstelle (**Semaphor**)

- ▶ gegenseitiger Ausschluss der Ausführung exportierter Prozeduren

**einseitige Synchronisation** innerhalb des Monitors (**Bedingungsvariable**)

**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 ggf. (genau einen oder alle) darauf blockierte Prozesse

☞ sprachgestützter Ansatz: Concurrent Pascal, PL/I, Mesa, ..., Java.

## Monitor ≡ Modul Monitor vs. Semaphor

**Kapselung** (engl. *encapsulation*)

- ▶ von mehreren Prozessen gemeinsam bearbeitete Daten müssen in Monitoren organisiert vorliegen
- ▶ als Konsequenz muss die Programmstruktur kritische Abschnitte explizit sichtbar machen
  - ▶ inkl. die zulässigen (an zentraler Stelle definierten) Zugriffsfunktionen

**Datenabstraktion** (engl. *information hiding*)

- ▶ wie ein Modul, so kapselt auch ein Monitor für mehrere Funktionen Wissen über gemeinsame Daten
- ▶ Auswirkungen lokaler Programmänderungen bleiben (eng) begrenzt

☞ ein Monitor ist Konzept der Ebene 5, ein Semaphor das der Ebene 3

## Modulkonzept 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 Prozedurauftrag 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** (Semaphore, Schloss-/Bedingungsvariablen)

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

## Signalisierung einer Wartebedingung erwarten *wait*

**Monitorfreigabe** als notwendiger Seiteneffekt beim Warten:

- ▶ andere Prozesse wären sonst an den Monitoreintritt gehindert
- ▶ als Konsequenz könnte die zu erfüllende Bedingung nie erfüllt werden
- ▶ schlafenlegende Prozesse würden nie mehr erwachen → **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

☞ aktives Warten im Monitor ist logisch komplex und leistungsmindernd

## Signalisierung einer Wartebedingung signal

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
- ▶ große Unterschiede liegen auch in **Besitzwechsel** bzw. **Besitzwahrung**
  - ▶ „falsche Signalisierungen“ werden toleriert oder nicht

## Signalisierungssemantiken Besitzwahrung

genau einen wartenden Prozess befreien ... nur welchen?

- ▶ bei mehr als einem wartenden Prozess ist eine Auswahl zu treffen
- ▶ die Auswahlentscheidung muss zur Fadeneinplanung korrespondieren
- ▶ ggf. ist bereits bei Blockierung möglichen Konflikten vorzubeugen

alle wartenden Prozesse befreien → Hansen [64]

- ▶ die Auswahlentscheidung unterliegt allein dem Planer
- ▶ Fadeneinplanung entgegenwirkende Konflikte werden ausgeschlossen
- ▶ verschiedene Belange sind sauber voneinander getrennt

- ▶ in beiden Fällen erfolgt die **Neuauswertung der Wartebedingung**
  - ▶ dadurch werden jedoch auch falsche Signalisierungen toleriert
- ▶ signalisierte Prozesse bewerben sich erneut um den Monitorzutritt

## Signalisierungssemantiken (Forts.) Besitzwechsel

Wechsel vom signalisierenden zum signalisierten Prozess → Hoare [67]

- ▶ nur einer von ggf. mehreren wartenden Prozessen wird signalisiert
  - ▶ der signalisierte Prozess setzte seine Berechnung im Monitor fort
  - ▶ als Folge muss der signalisierende Prozess den Monitor abgeben
- ▶ **Fortführungsbedingung** des signalisierten Prozesses ist garantiert
  - ▶ seit Signalisierung konnte kein anderer Prozess den Monitor betreten
  - ▶ kein anderer Prozess konnte die Fortführungsbedingung entkräften

- ▶ es erfolgt **keine Neuauswertung der Wartebedingung**
  - ▶ als Konsequenz werden falsche Signalisierungen nicht toleriert
- ▶ eine erhöhte Anzahl von Fadenwechseln ist in Kauf zu nehmen
- ▶ der signalisierende Prozess bewirbt sich erneut um den Monitorzutritt

## Datenmonitor mit Pufferbegrenzung

*Bounded buffer revisited...*

„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);
};
```

## Datenmonitor mit Pufferbegrenzung (Forts.)

Koordiniertes Leeren

```
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  
 Instanzenvariable:  
**free** aktueller „Pegelstand“

**Hansen'scher Monitor** Wartebedingung ist wiederholt zu überprüfen

- ▶ bewirbt signalisierte Prozesse erneut um den Monitorzutritt
  - die Phase ab der Signalisierung von **full** bis zum Wiedereintritt in den Monitor des signalisierten (leerenden) Prozesses ist teilbar
  - der Puffer könnte zwischenzeitig geleert worden sein → blockieren
- ▶ toleriert (fehlerbedingte) **falsche Signalisierungen** von **full**

## Datenmonitor mit Pufferbegrenzung (Forts.)

Koordiniertes Füllen

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

Bedingungsvariablen:  
**null** erwartet freien Platz  
**full** signalisiert einen Eintrag

Instanzenvariable: **free** führt Buch über den aktuellen „Pegelstand“

**Hansen'scher Monitor** Wartebedingung ist wiederholt zu überprüfen

- ▶ bewirbt signalisierte Prozesse erneut um den Monitorzutritt
  - die Phase ab der Signalisierung von **null** bis zum Wiedereintritt in den Monitor des signalisierten (füllenden) Prozesses ist teilbar
  - der Puffer könnte zwischenzeitig gefüllt worden sein → blockieren
- ▶ toleriert (fehlerbedingte) **falsche Signalisierungen** von **null**

## Monitorkonzepte im Vergleich

Hansen vs. Hoare

### Hansen'scher Monitor

```
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'scher Monitor

```
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

## Blockierende Synchronisation „considered harmful“

Probleme von Schlossvariablen, Semaphore und Monitore

**Leistung** (engl. *performance*) insbesondere in SMP-Systemen [63]

- „*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** [68]
  - ▶ Mars Pathfinder [69]

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