

# Systemprogrammierung

*Grundlage von Betriebssystemen*

## Teil C – X.2 Prozesssynchronisation: Monitore

Jürgen Kleinöder

16. November 2017



# Agenda

---

Einführung

Monitor

Eigenschaften

Architektur

Bedingungsvariable

Definition

Operationen

Signalisierung

Beispiel

Datenpuffer

Zusammenfassung



# Gliederung

---

## Einführung

### Monitor

Eigenschaften

Architektur

### Bedingungsvariable

Definition

Operationen

Signalisierung

### Beispiel

Datenpuffer

### Zusammenfassung



- Auseinandersetzung mit Begrifflichkeiten bezüglich “a shared variable and the set of meaningful operations on it” [5, p. 121]:
  - monitor ■ ursprünglich auch **kritischer Bereich** (*critical region*, [4, 5])
  - assoziiert Prozeduren mit einer gemeinsamen Variablen
    - versetzt einen Kompilierer in die Lage:
      - (a) die für die Variable definierten Operationen zu prüfen
      - (b) den wechselseitigen Ausschluss der Operationen zu erzwingen
  - condition ■ eine **Variable** für die gilt: “it does not have any stored value accessible to the program” [8, p. 550]
  - dient der Anzeige und Steuerung eines Wartezustands
    - für den jeweiligen Prozess innerhalb des Monitors
- die Funktionsweise des Monitors als ein **Mittel zur Synchronisation** verstehen, unabhängig linguistischer Merkmale
  - Erklärung verschiedener Stile: Hansen, Hoare, Concurrent Pascal, Mesa
  - diesbezügliche schematische Darstellung von Implementierungsvarianten
- jedoch schon die **problemorientierte Programmiersprachenebene** als Verortung dieser Konzepte im Rechengesystem identifizieren (s. [11])



# Instrument zur Überwachung

## Hinweis (Monitor [5, S. 121])

*The purpose of a monitor is to control the scheduling of resources among individual processes according to a certain policy.*

- ein Konzept kennenlernen, das als **programmiersprachlicher Ansatz** einzustufen ist, aber gleichsam darüber hinaus geht
  - ein klassenähnlicher synchronisierter Datentyp [5, 8, 12]
  - inspiriert durch SIMULA 67 [3, 2]
  - zuerst implementiert in Concurrent Pascal [6]
  - danach realisiert in unterschiedlichen Ausführungen [1, 7]
- die Technik ist grundlegend für die Systemprogrammierung und den systemnahen Betrieb von gekoppelten Prozessen
  - mit dem Monitorkonzept ist auch eine **Programmierkonvention** gemeint und nicht immer bloß ein **Programmiersprachenkonstrukt**
  - diese Konvention ist in jeder Programmiersprache nutzbar, jedoch nicht in jeder integriert und nicht von jedem Kompilierer umgesetzt



# Gliederung

---

Einführung

Monitor

Eigenschaften

Architektur

Bedingungsvariable

Definition

Operationen

Signalisierung

Beispiel

Datenpuffer

Zusammenfassung



# Synchronisierter abstrakter Datentyp: Monitor

- in den grundlegenden Eigenschaften ein **abstrakter Datentyp** [13], dessen Zugriffsoperationen implizit synchronisiert sind [5, 8]

**mehrseitige Synchronisation** an der Monitorschnittstelle

- wechselseitiger Ausschluss der Ausführung exportierter Prozeduren
  - realisiert mittels **Schlossvariablen** oder vorzugsweise **Semaphore**

**einseitige Synchronisation** innerhalb des Monitors

- logische Synchronisation mittels **Bedingungsvariable**:
  - wait** blockiert einen Prozess auf das Eintreten eines Ereignisses und gibt den Monitor implizit wieder frei
  - signal** zeigt das Eintreten eines Ereignisses an und deblockiert, je nach Typ des Monitors, einen oder alle darauf blockierte Prozesse
- bei **Ereigniseintritt** löst der betreffende Prozess ein Signal aus und zeigt damit die **Aufhebung einer Wartebedingung** an

- ein sprachgestützter Ansatz, bei dem der Übersetzer automatisch die Synchronisationsbefehle generiert

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



# Monitor $\equiv$ (eine auf ein Modul bezogene) Klasse

---

- ein Monitor ist einer **Klasse** [2] ähnlich und er besitzt alle Merkmale, die auch ein **Modul** [14] mitbringt

## Kapselung (*encapsulation*)

- von mehreren Prozessen gemeinsam bearbeitete Daten müssen, analog zu Modulen, in Monitoren organisiert vorliegen
- die Programmstruktur macht kritische Abschnitte explizit sichtbar
  - inkl. zulässige (an zentraler Stelle definierte) Zugriffsfunktionen

## Datenabstraktion (*information hiding*)

- wie ein Modul, so kapselt auch ein Monitor für mehrere Prozeduren Wissen über gemeinsame Daten
- Auswirkungen lokaler Programmänderungen bleiben begrenzt

## Bauplan (*blueprint*)

- wie eine Klasse, so beschreibt ein Monitor für mehrere Exemplare seines Typs den **Zustand** und das **Verhalten**
- er ist eine **gemeinsam benutzte Klasse** (*shared class*, [5])





# Klassenkonzept mit Synchronisationssemantik

*Monitor  $\equiv$  implizit synchronisierte Klasse*

## ■ **Monitorprozeduren** (*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 erzeugt die dafür notwendigen Steueranweisungen

## ■ **Synchronisationsanweisungen** $\leadsto$ **Bedingungsvariable**

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



# Wiederverwendbare unteilbare Ressource

- ein Monitor ist Bauplan für ein **Softwarebetriebsmittel**, mit dem verschiedene Sorten von Warteschlangen verbunden sind
  - in der **Monitorwarteschlange** befinden sich Prozessexemplare, die den Eintritt in den Monitor erwarten
    - sie warten nur, wenn der Monitor im Moment des Eintrittsversuchs bereits von einem anderen Prozess belegt war
    - erst bei Monitorfreigabe wird einer dieser Prozesse zur Auswahl bereitgestellt
  - die **Ereigniswarteschlange** enthält Prozessexemplare, die im Monitor die Aufhebung einer Wartebedingung erwarten
    - sie warten nur, wenn ein mit einer Bedingungsvariable verknüpftes Ereignis noch nicht eingetreten ist
    - beachte: dieses Ereignis bildet ein **konsumierbares Betriebsmittel** [9, S. 14]
- ein Prozess wartet jedoch stets außerhalb des Monitors, das heißt, er belegt den Monitor während seiner Wartezeit nicht
  - ansonsten könnte kein anderer Prozess den Monitor betreten und somit die Wartebedingung für einen Prozess aufheben
  - mit Aufhebung der Wartebedingung eines Prozesses, wird diesem der **Wiedereintritt** in den Monitor ermöglicht



# Monitor mit blockierenden Bedingungsvariablen

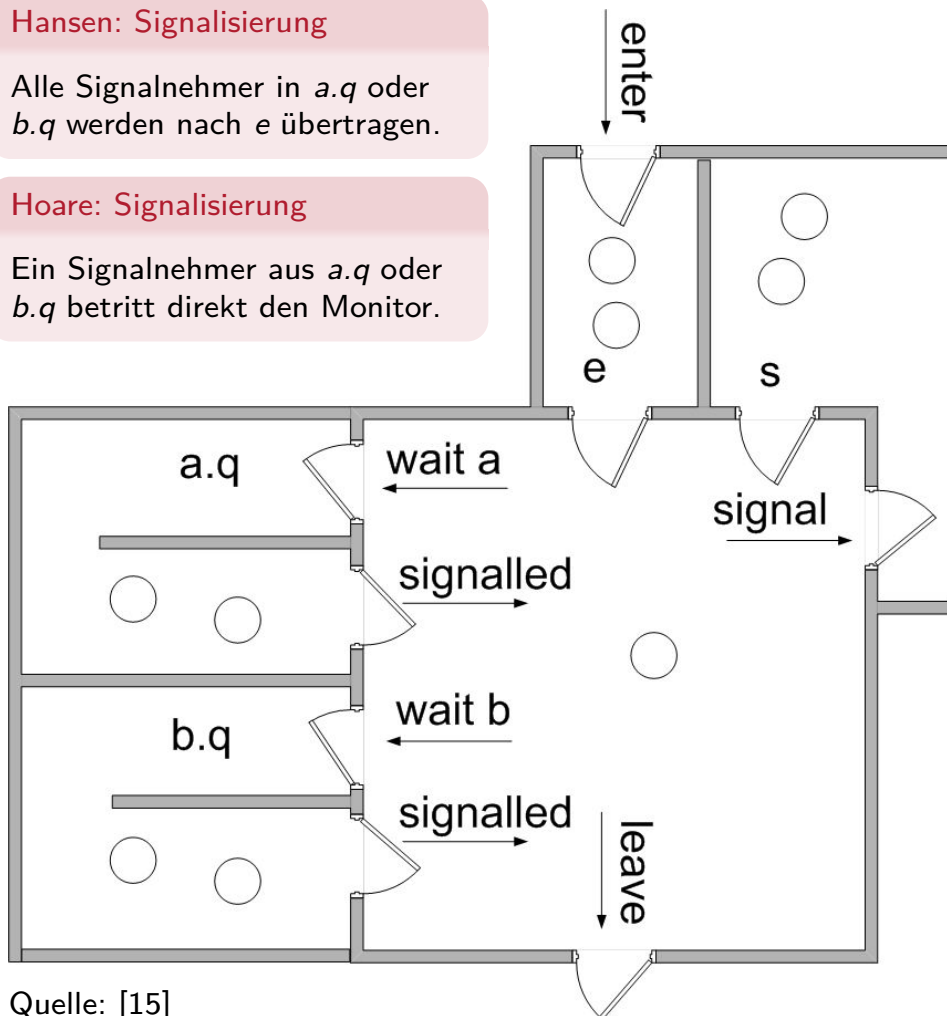
- nach Hansen [5] und Hoare [8], letzterer hier im Bild skizziert:

## Hansen: Signalisierung

Alle Signalnehmer in  $a.q$  oder  $b.q$  werden nach  $e$  übertragen.

## Hoare: Signalisierung

Ein Signalnehmer aus  $a.q$  oder  $b.q$  betritt direkt den Monitor.



Quelle: [15]

## Monitorwarteschlangen

$e$  der Zutrittsanforderer

$s$  der Signalgeber: **optional**

- Vorzugswarteliste oder vereint mit  $e$

## Ereigniswarteschlangen

$a.q$  für Bedingungsvariable  $a$

$b.q$  für Bedingungsvariable  $b$

## ■ Signalgeber blockieren

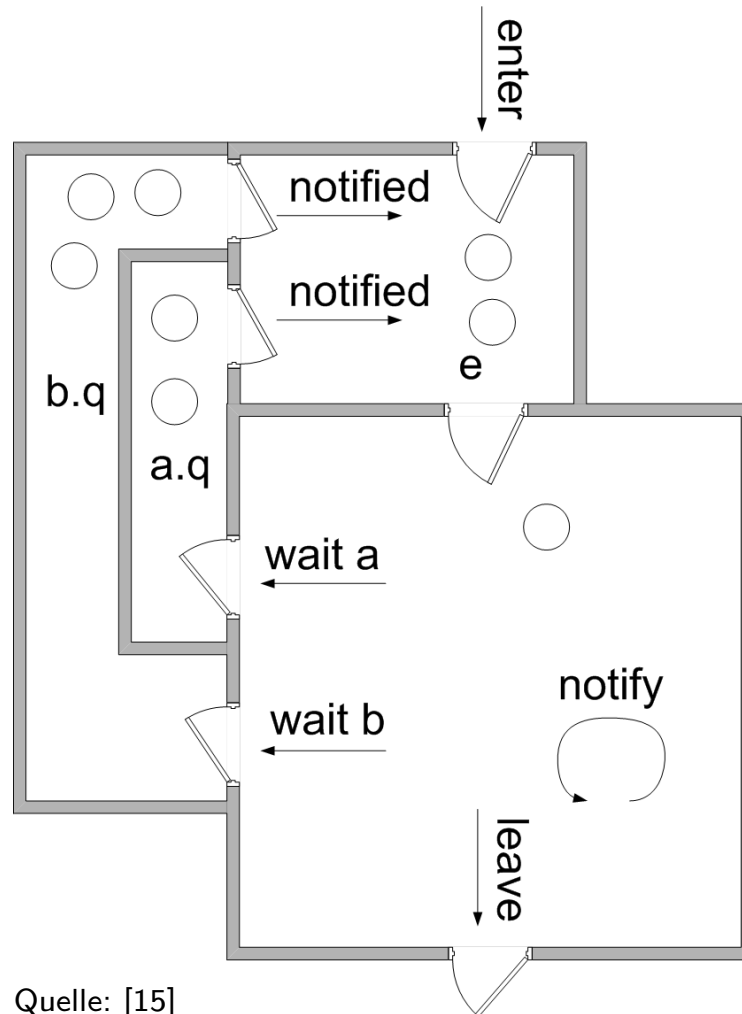
- warten außerhalb
- verlassen den Monitor

- **Wiedereintritt** falls *signal* nicht letzte Operation



# Monitor mit nichtblockierenden Bedingungsvariablen

- in Mesa [12]:



Quelle: [15]

## 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*)



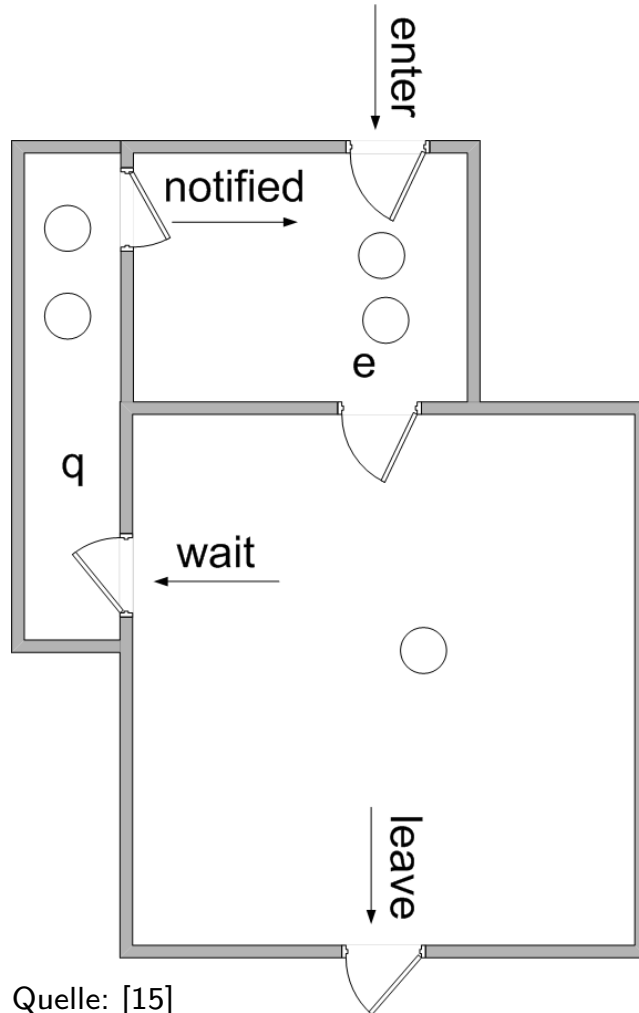
# Monitorvergleich: Hansen, Hoare, Mesa

- Ausgangspunkt für die Verschiedenheit der Monitorkonzepte ist die **Semantik der Bedingungsvariablen**:
  - blockierend ■ gibt dem Signalnehmer Vorrang
  - nichtblockierend ■ gibt dem Signalgeber Vorrang
- Folge davon ist eine unterschiedliche **Semantik der Signalisierung** für die betrachteten Monitorarten:
  - Hansen ■ verwendet blockierende Bedingungsvariablen
    - Signalisierung lässt den Signalgeber den Monitor verlassen, nachdem er alle Signalnehmer „bereit“ gesetzt hat
  - Hoare ■ verwendet blockierende Bedingungsvariablen
    - Signalisierung lässt den Signalgeber den Monitor verlassen und genau einen Signalnehmer fortfahren  $\leadsto$  *atomare Aktion*
  - Mesa ■ verwendet nichtblockierende Bedingungsvariablen
    - Signalisierung lässt den Signalgeber im Monitor fortfahren, nachdem er einen oder alle Signalnehmer „bereit“ gesetzt hat



# Monitor mit impliziten Bedingungsvariablen

- in Java, C#:



Quelle: [15]

## Monitorwarteschlange

**e** der Zutrittsanforderer und der signalisierten Prozesse

## Ereigniswarteschlange

**q** für den gesamten Monitor

- Objekte sind zwar keine Monitore, können aber als solche verwendet werden
- `synchronized`-Anweisung an Methoden oder Basisblöcken
- Signalisierung wie bei Mesa (S. 12)



# Gliederung

---

Einführung

Monitor

Eigenschaften

Architektur

Bedingungsvariable

Definition

Operationen

Signalisierung

Beispiel

Datenpuffer

Zusammenfassung



# Variable ohne Inhalt...

Hinweis (Bedingungsvariable (*condition variable* [8, S.550]))

*Note that a condition “variable” is neither true nor false; indeed, it does not have any stored value accessible to the program.*

- fundamentale Primitive [8] zur **Bedingungssynchronisation** wobei die Operationen folgende intrinsische Eigenschaften haben:
  - `signal` ■ zeigt ein Ereignis an
    - ist wirkungslos, sollte kein Prozess auf das Ereignis warten
    - nimmt genau einen wartenden Prozess sofort wieder auf
  - `wait` ■ setzt den Prozess bis zur Anzeige eines Ereignisses aus
    - gibt den Monitor bis zur Wiederaufnahme implizit frei
- auch **Ereignisvariable** (*event variable* [4]), mit den beiden zu oben korrespondierenden Operationen `cause` und `await`
  - alle dasselbe Ereignis erwartende Prozesse werden durch `cause` befreit<sup>1</sup>
  - wobei vorrangige Prozesse Vorrang beim Eintritt in den Monitor erhalten

<sup>1</sup>Wobei [4] Aussetzung/Fortsetzung des signalisierenden Prozesses offen lässt.





- `when (condition) wait(event)` mit *when* gleich:
  - `if` ■ Ereignisauslösung bis Prozesseinlastung ist **unteilbare Aktion**
    - dazwischen ist die Wartebedingung nicht erneut erfüllbar
    - d.h., kein Prozess kann zwischenzeitlich in den Monitor eintreten
  - `while` ■ sonst
- die Aktion, innerhalb eines Monitors zu warten, muss zwingend die **Monitorfreigabe** zur Folge haben:
  - andere Prozesse wären sonst am Monitoreintritt gehindert
  - als Folge könnte die Wartebedingung nie aufgehoben werden
  - schlafende Prozesse würden nie mehr erwachen  $\leadsto$  **Verklemmung**
- da in der Wartezeit ein anderer Prozess den Monitor betreten muss, ist explizit für **Konsistenz** der Monitordaten zu sorgen
  - ein oder mehrere andere Prozesse heben die Wartebedingung auf
  - als Folge werden sich Daten- bzw. Zustandsänderungen ergeben
  - vor Eintritt in die Wartephase muss der Datenzustand konsistent sein



- `cancel(condition) ... signal(event)` wobei *cancel* die Aktion zur Aufhebung der Wartebedingung (*condition*) repräsentiert
  - diese Bedingung ist ein **Prädikat** über den internen Monitorzustand
- Zweck der Signalisierung ist es, eine bestehende **Prozessblockade** in Bezug auf eine Wartebedingung zu beenden
  - warten Prozesse, muss die Operation für **Prozessfortschritt** sorgen
    - mindestens einer der das Ereignis erwartenden Prozesse wird deblockiert
    - höchstens ein Prozess rechnet nach der Operation im Monitor weiter
  - erwartet kein Prozess das Ereignis, ist die Operation wirkungslos
    - d.h., Signale dürfen in Bedingungsvariablen nicht gespeichert werden
- die Verfahren dazu sind teils von sehr unterschiedlicher Semantik und wirken sich auf die Programmierung auf
  - das betrifft etwa die Anzahl der deblockierten Prozesse
    - alle auf dasselbe Ereignis wartenden oder nur einer: **while** vs. **if** (S. 17)
    - falsche Signalisierungen werden toleriert (**while**) oder nicht (**if**)
  - bzw. ob **Besitzwechsel** oder **Besitzwahrung** des Monitors stattfindet



- **signal** befreit einen oder mehrere Prozesse und sorgt dafür, dass der aktuelle Prozess den Monitor abgibt
  - alle das Ereignis erwartenden Prozesse befreien  $\mapsto$  Hansen [4, S. 576]
    - alle Prozesse aus der Ereignis- in die Monitorwarteschlange bewegen
    - bei Freigabe alle  $n$  Prozesse (Monitorwarteschlange) „bereit“ setzen
    - $n - 1$  Prozesse reihen sich erneut in die Monitorwarteschlange ein
    - $\hookrightarrow$  Neuauswertung der Wartebedingung erforderlich (S. 17, **while**)
    - $\hookrightarrow$  falsche Signalisierungen (S. 29) werden toleriert
  - höchstens einen das Ereignis erwartenden Prozess befreien  $\mapsto$  Hoare [8]
    - einen einzigen Prozess der Ereigniswarteschlange entnehmen und fortsetzen
    - den signalisierenden Prozess in die Monitorwarteschlange eintragen
    - direkt vom signalisierenden zum signalisierten Prozess wechseln
    - $\hookrightarrow$  Neuauswertung der Wartebedingung entfällt (S. 17, **if**)
    - $\hookrightarrow$  falsche Signalisierungen (S. 29) werden nicht toleriert
- der signalisierende Prozess bewirbt sich erneut um den Monitor oder wird fortgesetzt, wenn der signalisierte Prozess den Monitor verlässt
  - letzteres (*urgent*, Hoare) greift auf eine **Vorzugswarteschlange** zurück



- **signal** befreit die auf das Ereignis wartenden Prozesse, setzt jedoch den aktuellen Prozess im Monitor fort
  - einen oder alle das Ereignis erwartenden Prozesse befreien  $\mapsto$  Mesa [12]
    - Prozess(e) aus der Ereignis- in die Monitorwarteschlange bewegen
    - bei Freigabe  $n \geq 1$  Prozesse (Monitorwarteschlange) „bereit“ setzen
    - Neuauswertung der Wartebedingung erforderlich (S.17, **while**)
    - falsche Signalisierungen (S.29) werden toleriert
- genau einen Prozess auszuwählen (Mesa, Hoare) birgt die Gefahr von **Prioritätsverletzung** [12]
  - betrifft die Entnahme eines Prozesses aus der Ereigniswarteschlange
  - Interferenz mit der Prozesseinplanung ist vorzubeugen/zu vermeiden
- mehrere oder gar alle Prozess auszuwählen (Mesa, Hansen) birgt das Risiko der erneuten Erfüllung der Wartebedingung
  - nach Fortsetzung des ersten befreiten Prozesses: erfüllt durch ihn selbst oder durch andere Prozesse, die zwischenzeitlich im Monitor waren
  - da ein Prozess nicht weiß, ob er als erster befreit wurde, muss jeder die Wartebedingung erneut auswerten



# Gliederung

---

Einführung

Monitor

Eigenschaften

Architektur

Bedingungsvariable

Definition

Operationen

Signalisierung

Beispiel

Datenpuffer

Zusammenfassung



```
1  template<typename T, unsigned N=64, monitor>
2  class Buffer {
3      T buffer[N];           // N should be power of two
4      unsigned in, out;
5      condition data, free;
6
7  atomic:                    // public, mutual exclusive methods
8      Buffer() { in = out = 0; }
9
10     void put(T item) {
11         when (((in + 1) % N) == out) free.wait();
12         buffer[in++ % N] = item;
13         data.signal();
14     }
15
16     T get() {
17         when (out == in) data.wait();
18         T item = buffer[out++ % N];
19         free.signal();
20         return item;
21     }
22 };
```



# Gliederung

---

Einführung

Monitor

Eigenschaften

Architektur

Bedingungsvariable

Definition

Operationen

Signalisierung

Beispiel

Datenpuffer

Zusammenfassung



- 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, Java) für den ein Ereignis signalisierenden Prozess
  - deblockiert einen (Hoare, Mesa, Java) oder alle (Hansen, Mesa, Java) auf ein Ereignis wartende Prozesse
  - die Wartebedingung für den jeweils signalisierten Prozess ist garantiert (Hoare) oder nicht garantiert (Hansen, Mesa, Java) aufgehoben
  - erfordert (Hansen, Mesa, Java) oder erfordert nicht (Hoare) die erneute Auswertung der Wartebedingung nach Wiederaufnahme
  - ist falschen Signalisierungen gegenüber tolerant (Hansen, Mesa, Java) oder intolerant (Hoare)
- Java-Objekte sind keine Monitore, wohl aber als solche verwendbar
  - so bietet Java keine Monitorobjekte, entgegen der Informatikfolklore...





# Literaturverzeichnis I

---

- [1] BUHR, P. A. ; FORTIER, M. :  
Monitor Classification.  
In: *ACM Computing Surveys* 27 (1995), März, Nr. 1, S. 63–107
- [2] DAHL, O.-J. ; MYHRHAUG, B. ; NYGAARD, K. :  
SIMULA Information: Common Base Language / Norwegian Computing Center.  
1970 (S-22). –  
Forschungsbericht
- [3] DAHL, O.-J. ; NYGAARD, K. :  
SIMULA—An ALGOL-Based Simulation Language.  
In: *Communications of the ACM* 9 (1966), Sept., Nr. 9, S. 671–678
- [4] HANSEN, P. B.:  
Structured Multiprogramming.  
In: *Communications of the ACM* 15 (1972), Jul., Nr. 7, S. 574–578
- [5] HANSEN, P. B.:  
*Operating System Principles*.  
Englewood Cliffs, N.J., USA : Prentice-Hall, Inc., 1973. –  
ISBN 0–13–637843–9



# Literaturverzeichnis II

---

- [6] HANSEN, P. B.:  
The Programming Language Concurrent Pascal.  
In: *IEEE Transactions on Software Engineering SE-I* (1975), Jun., Nr. 2, S. 199–207
- [7] HANSEN, P. B.:  
Monitors and Concurrent Pascal: A Personal History.  
In: BERGIN, JR., T. (Hrsg.) ; GIBSON, JR., R. G. (Hrsg.): *History of Programming Languages—II*.  
New York, NY, USA : ACM, 1996. –  
ISBN 0–201–89502–1, S. 121–172
- [8] HOARE, C. A. R.:  
Monitors: An Operating System Structuring Concept.  
In: *Communications of the ACM* 17 (1974), Okt., Nr. 10, S. 549–557
- [9] KLEINÖDER, J. ; SCHRÖDER-PREIKSCHAT, W. :  
Nichtsequentialität.  
In: [10], Kapitel 10.1
- [10] KLEINÖDER, J. ; SCHRÖDER-PREIKSCHAT, W. ; LEHRSTUHL INFORMATIK 4 (Hrsg.):  
*Systemprogrammierung*.  
FAU Erlangen-Nürnberg, 2015 (Vorlesungsfolien)



# Literaturverzeichnis III

---

- [11] KLEINÖDER, J. ; SCHRÖDER-PREIKSCHAT, W. :  
Virtuelle Maschinen.  
In: [10], Kapitel 5.1
- [12] 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
- [13] LISKOV, B. J. H. ; ZILLES, S. N.:  
Programming with Abstract Data Types.  
In: LEAVENWORTH, B. (Hrsg.): *Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages* Bd. 9.  
New York, NY, USA : ACM, Apr. 1974 (ACM SIGPLAN Notices 4), S. 50–59
- [14] PARNAS, D. L.:  
On the Criteria to be used in Decomposing Systems into Modules.  
In: *Communications of the ACM* 15 (1972), Dez., Nr. 12, S. 1053–1058
- [15] WIKIPEDIA:  
*Monitor (synchronization)*.  
[http://en.wikipedia.org/wiki/Monitor\\_\(synchronization\)](http://en.wikipedia.org/wiki/Monitor_(synchronization)), Dez. 2010



# Fiktion: *Concurrent C++*

- hier als **Klassenvorlage** (*class template*) ausgelegt, um in einfacher Weise eine generische Implementierung des Datenpuffers zu zeigen:
  - monitor** ■ Vorlagenparameter zur Deklaration der Klasse als Monitor
    - Verfeinerung durch *monitor=name*, wobei *name* sein kann:
      - i *signal* bzw. *signal continue* (vorgegeben) oder
      - ii *signal wait*, *signal urgent wait*, *signal return*
    - Exemplare dieser Klasse sind **Monitorobjekte**
  - condition** ■ Deklaration von einer oder mehreren Bedingungsvariablen
    - Voraussetzung für die Operatoren *wait* und *signal*
  - when** ■ Kontrollstruktur zum bedingten Warten (*wait*) im Monitor
    - je nach Verfeinerung des Monitors entspricht diese implizit:
      - *while* für *signal [continue]* und *signal return*
      - *if* sonst, d.h., für *signal wait* und *signal urgent wait*
    - jedoch kann jede Kontrollstruktur explizit gemacht werden
- aber auch ohne „Spracherweiterung“ lässt sich das Monitorkonzept integrieren, wenngleich auch mit Abstrichen  $\rightsquigarrow$  vgl. S. 30
  - Spezialisierung durch Vererbung `class Buffer : private Monitor`
  - Condition als Klasse mit `wait()/signal()`-Methoden, **when** umseitig



```
1 #ifdef __MONITOR_SIGNAL_AND_RESUME__
2 #define when if      /* waiting condition nullified */
3 #else
4 #define when while  /* re-evaluate waiting condition */
5 #endif
```

- die **Neuauswertung** der Wartebedingung kann entfallen, wenn:
  - i `signal` höchstens einen Prozess auswählt und
  - ii der ausgewählte Prozess unteilbar wieder aufgenommen wird
- in dem Fall, wird eine **falsche Signalisierung** jedoch nicht toleriert
  - dabei handelt es sich um einen **Programmierfehler**, d.h., `signal` kommt zur Ausführung, obwohl die Wartebedingung nicht aufgehoben ist
  - `signal` wurde entweder auf die richtige, dann aber zu unrecht, oder die falsche Bedingungsvariable appliziert
- Neuauswertung der Wartebedingung toleriert falsche Signalisierungen und ist notwendig, wenn mehrere Prozesse signalisiert werden
  - generelle Lösung für alle Signalisierungsvarianten...



```
1  template<typename T, unsigned N=64>
2  class Buffer : private Monitor {
3      T buffer[N];           // N should be power of two
4      unsigned in, out;
5      Condition data, free;
6  public:
7      Buffer() {
8          enter();           // critical section
9          in = out = 0;
10         leave();
11     }
12
13     void put(T item) {
14         enter();           // critical section
15         when (((in + 1) % N) == out) free.wait(*this);
16         buffer[in++ % N] = item;
17         data.signal();
18         leave();
19     }
20
21     T get() {
22         enter();           // critical section
23         when (out == in) data.wait(*this);
24         T item = buffer[out++ % N];
25         free.signal();
26         leave();
27
28         return item;
29     }
30 };
```



■ wait muss den Monitor kennen, der zu entsperren und anzufordern ist