Atomare Aktionen

Atomare Aktion: Folge von (einer oder mehreren) Operationen, deren interne Zustände nicht beobachtet werden können, weil

- es eine **unteilbare Maschineninstruktion** ist,
- weil globale Variablen nur einmal zugegriffen werden (**AMO**),
- weil durch **Synchronisation** die Verzahnung mit anderen Prozessen eingeschränkt wird.

At-most-once Eigenschaft (AMO):

- **Ausdruck A:**
  A hat höchstens eine Variable v, die von einem anderen Prozess geschrieben wird, und v kommt nur einmal in A vor.
- **Zweisung x := A** :
  A ist AMOund x wird nicht von einem anderen Prozess gelesen oder
  x wird von einem anderen Prozess gelesen aber A hat nur Prozess-lokale Variable.
- **Anweisungsfolge:**
  eine Anweisung ist AMO und alle anderen enthalten nur Prozess-lokale Variablen.

Interferenz zwischen Prozessen

Für jede **Aussage C über beobachtbare Zustände** eines Prozesses p muss Nicht-Interferenz NI (Z, C) mit jeder **Zuweisung Z** jedes anderen Prozesses q gelten:

p: {C1} <... atomare Aktion...> {C2} < ... > {C3} < ... > {C4}

NI(Z, C)

q: x = ... y = ...

Nicht-Interferenz NI (Z, C) zwischen Zuweisung Z: x = e in q und Aussage C über p gilt, wenn man schließen kann

$$\{ C \wedge pre(Z) \} \quad Z \quad \{ C \}$$

d. H. die Ausführung von **Z lässt C invariant**, falls die schwächste Vorbedingung von Z pre(Z) überhaupt zulässt, dass Z ausgeführt wird, wenn C gilt.

Kommunikation und Synchronisation paralleler Prozesse

Kommunikation zwischen parallelen Prozessen: Datenaustausch durch

- Benutzung gemeinsamer, globaler **Variable**
  nur im Programmiermodell **gemeinsamer Speicher**

- **Botschaften** in Programmiermodellen **verteilter** oder gemeinsamer **Speicher**
  synchrone Botschaften: Sender wartet auf Empfänger (Sprachen: CSP, Occam, Ada, SR)
  asynchrone Botschaften: Sender wartet nicht auf Empfänger (Sprachen: SR)

Synchronisation paralleler Prozesse:

- **gegenseitiger Ausschluss (mutual exclusion):**
  bestimmte Anweisungsfolgen (kritische Abschnitte) dürfen nicht gleichzeitig von mehreren Prozessen ausgeführt werden.

- **Bedingungssynchronisation (condition synchronization):**
  ein Prozess wartet, bis eine bestimmte Bedingung durch einen anderen Prozess erfüllt wird.

Sprachkonstrukte zur Synchronisation:
Semaphore, Monitore, Bedingungsvariable (Programmiermodell gemeinsamer Speicher),
Botschaften (siehe oben)

Deadlock (Verklemmung):
Einige Prozesse warten wechselweise aufeinander, sind dadurch endgültig blockiert.

Monitor - allgemeines Prinzip

Monitor [T. Hoare 1974, P. Brinch Hansen 1975]

- Programmmodul im Programmiermodell „gemeinsamer Spreicher“, kapselt Daten mit Operationen darauf
- Prozesse rufen Operationen (**Entry Prozeduren**) auf; der Monitor ist **passiv**
- Monitor garantiert **gegenseitigen Ausschluss für die Aufrufe der Entry Prozeduren:** zu jedem Zeitpunkt ist höchstens ein Prozess im Monitor.
- **Bedingungssynchronisation** durch **Bedingungsvariable**; werden lokal im Monitor benutzt.

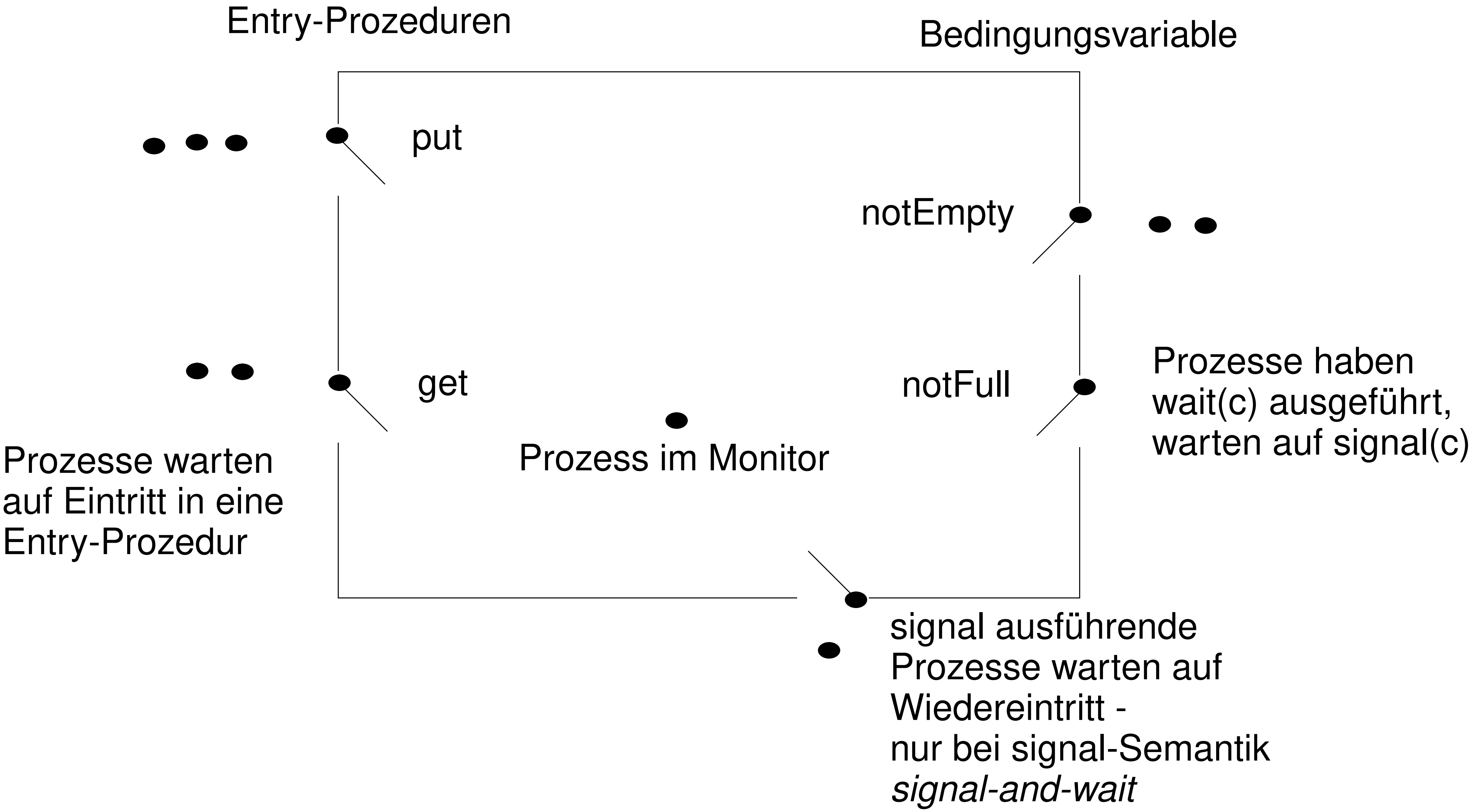
Bedingungsvariable c mit 2 Operationen darauf:

- **wait (c)** ausführender Prozess gibt den Monitor frei,
  wartet in einer Warteschlange zu c auf ein nachfolgendes signal(c),
  setzt dann im Monitor fort.
- **signal(c):** ausführender Prozess gibt **einen** beliebigen der auf c wartenden Prozesse frei;
  setzt dann im Monitor fort (signal-and-continue);
  hat keinen Effekt, wenn keiner auf c wartet.

Beispiel: beschränkter Puffer

```
monitor Buffer
   buff: Queue (k);
   notFull, notEmpty: Condition;   2 Bedingungsvariable: Zustand des Puffers

   entry put (d: Data)
      do length(buf) = k -> wait (notFull); od;
      enqueue (buf, d);
      signal (notEmpty);
   end;

   entry get (var d: Data) x: Data;
      do length (buf) = 0 -> wait (notEmpty); od;
      x := front (buf); dequeue (buf);
      signal (notFull);
   end;
end;

process Producer (i: 1..n) d: Data;
   loop d := produce(); Buffer.put(d); end;
end;

process Consumer (i: 1..m) d: Data;
   loop Buffer.get(d); consume(d); end;
end;
```

Synchronisation im Monitor



Signal-Wait Varianten

Prozesse in Konkurrenz um den Monitor:

- durch wait(c) blockierter Prozess,
- signal(c) ausführender Prozess, der ihn freigibt,
- auf Entry-Prozeduren wartende Prozesse.

signal-and-continue Semantik:

Der signal ausführende Prozess setzt sofort im Monitor fort.
Der freigegebene Prozess muss warten bis der Monitor frei ist.

Er muss die Wartebedingung erneut prüfen - auch wenn sie bei signal galt:

```
do length(buf) = k -> wait(notFull); od;
```

signal-and-wait Semantik:

Der signal ausführende Prozess muss warten bis der Monitor frei ist.
Variante **signal-and-urgent-wait**:
signal-Prozess hat höhere Priorität als Eintritt in Entry-Prozeduren

signal-and-exit Semantik:

Der signal ausführende Prozess verlässt den Monitor;
der freigegebene Prozess kann sofort im Monitor fortsetzen.

Monitore in Java: gegenseitiger Ausschluss

Objekte jeder Klasse können als **Monitor** verwendet werden.

Entry Prozeduren:

Methoden einer Klasse, die kritische Abschnitte auf Objektvariablen implementieren, können als `synchronized` gekennzeichnet werden:

```
class Buffer
{  synchronized public void put (Data d) {...}
   synchronized public Data get () {...}
   ...
   private Queue buf;
}
```

Aufrufe von `synchronized` Methoden von mehreren Prozessen für dasselbe Objekt werden unter **gegenseitigem Ausschluß** ausgeführt.
Synchronisation durch eine interne Synchronisationsvariable des Objektes (lock).

Nicht-`synchronized` Methoden können zu beliebigen Zeitpunkten aufgerufen werden.

Es gibt auch `synchronized` **Klassenmethoden**: sie werden in Bezug auf die Klasse unter gegenseitigem Ausschluss aufgerufen.

Mit `synchronized` **Blöcken** kann man einen kritischen Abschnitt bezüglich eines beliebigen Objektes unter gegenseitigen Ausschluss stellen.

Monitore inJava: Bedingungssynchronisation

Es gibt nur eine einzige Menge mit **`wait`** blockierter Prozesse,
keine deklarierbaren Bedingungsvariable.

Operationen zur Bedingungssynchronisation
müssen aus **`synchronized`** Methoden aufgerufen werden:

- **`wait()`** **blockiert** den aufrufenden Prozess,
gibt das Monitorobjekt frei und
wartet in einer unspezifischen Prozessmenge zu dem Objekt.

- **`notifyAll()`** gibt **alle** Prozesse frei, die mit **`wait`** auf das Objekt warten;
sie konkurrieren dann um den Monitor;
der aufrufende Prozess setzt im Monitor fort (signal-and-continue Semantik).

- **`notify()`** gibt **einen beliebigen** der Prozesse frei, die mit **`wait`** auf das Objekt warten;
der aufrufende Prozess setzt im Monitor fort (signal-and-continue Semantik).
Ist nur verwendbar, wenn alle Prozesse auf dieselbe Bedingung warten.

In Schleife warten, weil bei **signal-and-continue**-Semantik
nach **`notify, notifyAll`** die **Wartebedingung verändert** werden könnte:

```
while (!notFull) try { wait(); } catch (InterruptedException e) {}
```

Monitor-Klasse für Beschränkte Puffer

```
class Buffer
{  private Queue buf;          // Schlange der Länge n zur Aufnahme der Elemente
   public Buffer (int n) {buf = new Queue(n); }

   synchronized public void put (Object elem)
   {                          // ein Produzenten-Prozeß versucht, ein Element zu liefern
      while (buf.isFull()) // warten bis der Puffer nicht voll ist
         try {wait();} catch (InterruptedException e) {}
      buf.enqueue (elem);  // Wartebedingung der get-Methode hat sich verändert
      notifyAll();              // jeder blockierte Prozeß prüft seine Wartebedingung
   }
   synchronized public Object get ()
   {                       // ein Konsumenten-Prozeß versucht, ein Element zu nehmen
      while (buf.isEmpty())                    // warten bis der Puffer nicht leer ist
         try {wait();} catch (InterruptedException e) {}
      Object elem = buf.first();
      buf.dequeue();          // Wartebedingung der put-Methode hat sich verändert
      notifyAll();              // jeder blockierte Prozeß prüft seine Wartebedingung
      return elem;
   }
}
```