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.

Vorlesung Parallele Programmierung in Java SS 2000 / Folie 16

Ziele:

Begriff der atomaren Aktion im Verzahnungsmodell

in der Vorlesung:

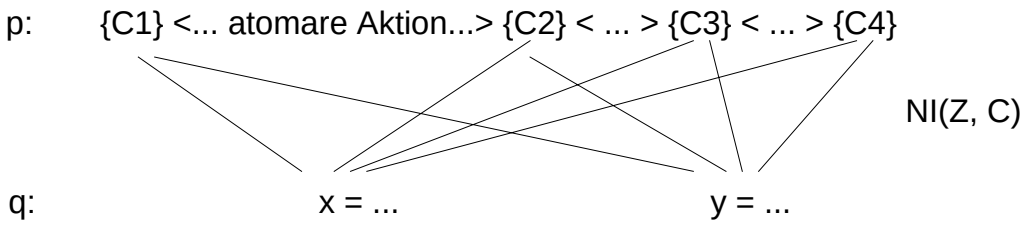
- Erläuterungen und Beispiele zu AMO.
- Das Beispiel von PPJ-15 variieren.

Verständnisfragen:

- Begründen Sie die AMO-Eigenschaft mit den Bregriffen "beobachtbare Zustände" und "Verzahnung"

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:



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.

Vorlesung Parallele Programmierung in Java SS 2000 / Folie 17

Ziele:

Zusammenhang: Verzahnung und Aussagen über Prozesse

in der Vorlesung:

dazu

- Erläuterung der NI
- Rolle der pre(Z) erläutern
- Verzahnungsmöglichkeiten erhöhen Zahl der NI-Nachweise
- Global gültige Aussagen vereinfachen die Nachweise
- Schwächere Aussagen vereinfachen die Nachweise

Verständnisfragen:

- Warum brauchen Aussagen über Zstände im inneren der atomaren Aktionen nicht berücksichtigt zu werden?

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.

Vorlesung Parallele Programmierung in Java SS 2000 / Folie 18

Ziele:

Grundbegriffe zur Synchronisation und Kommunikation

in der Vorlesung:

- Erläuterungen zur Kommunikation im gemeinsamen und verteilten Speicher
- Unterscheidung der beiden Synchronisationszwecke: gegenseitiger Ausschluss und Bedingungssynchronisation.
- Beispiele dazu
- Ausdrucksmittel dafür

Verständnisfragen:

- Geben Sie Beispiele, wo jeweils gegenseitiger Ausschluss oder Bedingungssynchronisation benötigt werden.

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.

Vorlesung Parallele Programmierung in Java SS 2000 / Folie 19

Ziele:

Grundkonzepte des Monitors verstehen

in der Vorlesung:

- Erläuterungen zu den 2 Synchronisationsmechanismen
- Bedingungsvariable sind nötig für Bedingssynchronisation im Monitor
- Beispiele dafür

Verständnisfragen:

- Sind Monitore ohne Bedingungsvariable brauchbar? Wofür?
- Warum muss die wait-Operation den Monitor freigeben?

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;
```


Vorlesung Parallele Programmierung in Java SS 2000 / Folie 20

Ziele:

Monitorkonzepte an einem einfachen Beispiel kennenlernen

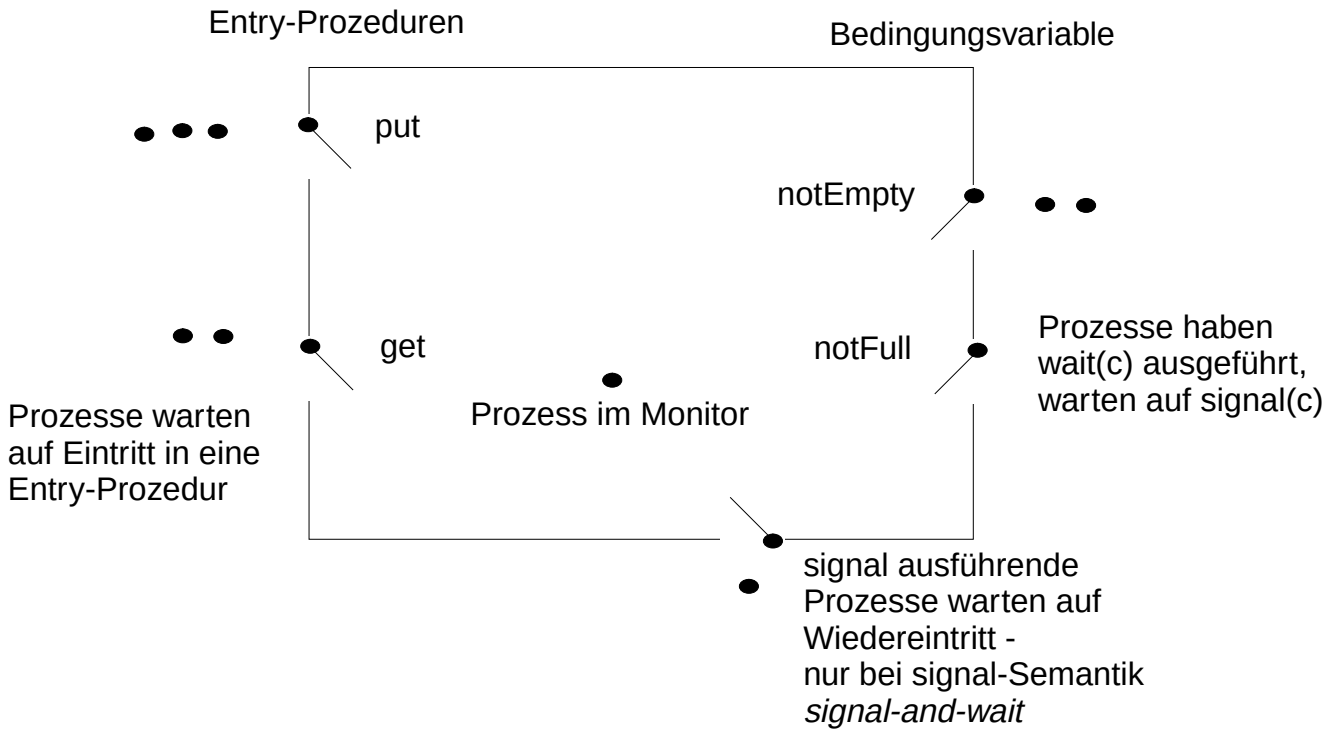
in der Vorlesung:

- 1 Monitor, n Produzenten-, m Konsumenten-Prozesse
- Erläuterungen der Monitor-Konstrukte
- Erläuterungen der Verwendung der Bedingungsvariablen
- Notation: Sprache SR, ähnlich zu der von Modula-2

Verständnisfragen:

- Beschreiben Sie die Rollen der beiden Bedingungsvariablen.
- Erklären Sie das Programm mit den Begriffen von PPJ-19.

Synchronisation im Monitor



Vorlesung Parallele Programmierung in Java SS 2000 / Folie 21

Ziele:

Veranschaulichung der Synchronisation im Monitor

in der Vorlesung:

- Erläuterungen der Wartebedingungen am Beispiel von PPJ-20
- Garantie: "höchstens 1 Prozess im Monitor"
- Begründung für das Warten bei signal

Verständnisfragen:

- Erläutern Sie die Synchronisationskonzepte von PPJ-19 an diesem Bild.
- Könnte man das Beispiel des bschränkten Puffers auch mit nur 1 statt 2 Bedingungsvariablen implementieren? Erläutern Sie.

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.

Vorlesung Parallele Programmierung in Java SS 2000 / Folie 22

Ziele:

Signal / Wait Semantik verstehen

in der Vorlesung:

Erläuterungen anhand des Bildes von PPJ-21

Verständnisfragen:

Warum muss man bei signal-and-continue Semantik in PPJ-20 die Bedingungen in Schleifen zu prüfen, obwohl die signal-Aufrufe am Ende der Prozeduren stehen?

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.

Vorlesung Parallele Programmierung in Java SS 2000 / Folie 23

Ziele:

Besonderheiten der Monitore in Java

in der Vorlesung:

- Objekte als Monitore
- gegenseitiger Ausschluß für jedes Objekt individuell
- Entry Prozeduren als synchronized Methoden
- gegenseitiger Ausschluß nur für Aufrufe von synchronized Methoden untereinander
- weitere Konstrukte: synchronized Klassenmethoden und Blöcke

Verständnisfragen:

Geben Sie Beispiele für Monitor-Methoden, die *nicht* unter gegenseitigem Ausschluß zu laufen brauchen.

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) {}
```


Vorlesung Parallele Programmierung in Java SS 2000 / Folie 24

Ziele:

Besonderheiten der Bedingungssynchronisation in Java verstehen

in der Vorlesung:

- Erläuterung der Operationen
- Verwendung bei mehreren verschiedenen Bedingungen.
- Verwendung von notify und notifyAll
- Konsequenz der signal-and-continue Semantik

Verständnisfragen:

- Zeigen Sie an verzahnten Ausführungen von Prozessen eine Situation, in der eine Bedingung C vor einem Aufruf von notify gilt, aber nach dem wait-Aufruf des dadurch gewckten Prozesses nicht mehr.

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;
   }
}
```


Vorlesung Parallele Programmierung in Java SS 2000 / Folie 25

Ziele:

Beispiel für eine Monitor-Klasse in Java

in der Vorlesung:

Erläuterungen dazu

- Veränderung der Wartebedingungen
- `notifyAll` begründen
- Zustandsübergänge bei `notifyAll` in der `get`-Operation am Beispiel erläutern

Verständnisfragen:

- Welche Zustände bezüglich der Erfüllung der Wartebedingungen sind möglich?
- Was bedeutet das für mehrere wartende Prozesse?
- Geben Sie detailliert an, wie mehrere wartende Prozesse auf den Aufruf `notifyAll()` reagieren.