Praxis des Programmierens

Vorlesung 25.05.1999

Uwe Waldmann

3 Java (Fortsetzung)

3.($n-1$) Synchronisation

Im allgemeinen können mehrere Threads gleichzeitig Methoden ausführen, die die gleichen Daten manipulieren. Offensichtlich kann dies zu inkonsistenten oder zumindest fehlerhaften Daten führen.

Ausweg: Solange ein Thread ein Programmstück

```
synchronized (obj) {
  ...
}
```

oder eine als `synchronized` deklarierte Methode des Objektes `obj` ausführt, wird jeder *andere* Thread, der versucht, ein derartiges für `obj` synchronisiertes Programmstück auszuführen, angehalten. Erst wenn der erste Thread `obj` wieder freigegeben hat, wird *einer* der angehaltenen Threads wieder aktiviert.

Es ist üblich, aber nicht notwendig, daß das Objekt, für das exklusiver Zugriff erreicht werden soll, selbst als *Lock*-Objekt verwendet wird.

Wichtig: `synchronized` verhindert nur, daß ein anderer Thread ebenfalls ein `synchronized`-Programmstück ausführt, nicht aber, daß ein anderer Thread außerhalb eines `synchronized`-Programmstückes auf `obj` zugreift.

3.n Warten

Die Methode `wait()` erlaubt es einem Thread, darauf zu warten, daß ein anderer Thread irgendeine Bedingung erfüllt. (Z. B. kann ein Thread, der aus einer Warteschlange etwas lesen will, falls die Warteschlange leer ist, darauf warten, daß ein anderer Thread ein Element in die Warteschlange einfügt.)

Die Methode `wait()` ist in der Klasse `Object` definiert. Sie kann nur für Objekte `obj` aufgerufen werden, für die der jeweilige Thread gerade synchronisiert ist. Der Aufruf von `wait()` hat mehrere Konsequenzen: Erstens gibt der aufrufende Thread den Lock für `obj` auf (so daß andere Threads für `obj` synchronisierte Programmstücke ausführen können). Zweitens wird der aufrufende Thread selbst angehalten und landet in der `obj` zugeordneten Wartemenge. Drittens bekommt er dort eine spezielle Markierung, und wird, solange er diese Markierung besitzt, nicht wieder aktiviert, selbst wenn kein anderer Thread ein für `obj` synchronisiertes Programmstück ausführt.

Diese Markierung wird erst dann wieder gelöscht, wenn ein anderer Thread die Methode `obj.notifyAll()` aufruft und so signalisiert, daß Bedingungen, auf die andere Thread gewartet haben, nun *möglicherweise* erfüllt sind. Auch `obj.notifyAll()` ist nur innerhalb eines für `obj` synchronisierten Programmstücks zulässig.

Wichtig: Wenn ein Thread nach einem `wait()` wieder aktiviert wird, dann folgt daraus *nicht*, daß die Bedingung, auf die er gewartet hat, nun erfüllt ist – nur, daß sie es sein könnte. Aus diesem Grund sollte `wait()` immer in einer `while`-Schleife verwendet werden:

```
synchronized void doWhenCondition() {
  while (!condition) {
    wait();
  }
  ...
}
```

Dieses `while` kann nicht durch ein `if` ersetzt werden.

Beispiel: Warteschlange (siehe Arnold und Gosling, The Java Programming Language, Kap. 9.3.)

4 Projekt: Aussagenlogische Erfüllbarkeit

4.1 Komplexitätsabschätzungen

Landau-Symbol: Seien f und g Funktionen von $\mathbb{N}$ nach $\mathbb{R}$. Man sagt, daß $f(n)$ in der Klasse $O(g(n))$ liegt, falls natürliche Zahlen k und n_0 existieren, so daß für alle $n \geq n_0$ gilt $|f(n)| \leq k\,|g(n)|$.

Es gilt $O(1) \subset O(\log n) \subset O(n) \subset O(n \log n) \subset O(n^2) \subset O(n^3) \subset \ldots \subset O(2^n) \subset O(3^n) \subset \ldots \subset O(2^{2^n}) \subset \ldots$

Man sagt, daß eine Funktion polynomiell beschränkt ist, wenn sie in $O(n^k)$ für irgenein $k \in \mathbb{N}$ liegt.

Jede beliebige Exponentialfunktion c^n mit $c > 1$ wächst stärker als *jedes beliebige* Polynom.

Bei Programmen interessant: Laufzeit und Speicherplatzbedarf in Abhängigkeit von der Größe der Eingabe.

Laufzeit $\widehat{=}$ Anzahl der Rechenschritte eines Prozessors $\widehat{=}$ Anzahl der ausgeführten Elementaroperationen: Zuweisungen, Vergleiche, arithmetische Operationen, ... (sofern Anzahl der Bits der Operanden konstant).

Vorsicht: Bibliotheksfunktionen müssen, sofern sie nicht mit konstanter Zeit auskommen, mitanalysiert werden.

4.2 Aussagenlogische Erfüllbarkeit

Eine Belegung φ bildet eine Menge von logischen Variablen (booleschen Variablen, Atomen) auf die Wahrheitswerte wahr und falsch ab. (In Zukunft immer: $\mathsf{wahr} = 1$, $\mathsf{falsch} = 0$.)

Wenn man eine Belegung für eine Menge von logischen Variablen hat, dann weist man damit auch jeder aussagenlogischen Formel über diesen Variablen einen Wahrheitswert zu.

Beispiel: Bildet φ A auf 1, B auf 1 und C auf 0 ab, dann besitzt die Formel $(A \vee B) \wedge (\neg A \vee C)$ unter φ den Wert $(1 \vee 1) \wedge (\neg 1 \vee 0) = 0$.

Problem: Gegeben eine aussagenlogische Formel F, existiert eine Belegung φ, so daß $\varphi(F) = 1$? Triviale Lösung: Wertetabelle aufstellen (aber: bei n Variablen müssen 2^n Kombinationen untersucht werden).

Wir betrachten spezielle Klasse von Formeln: Klauselmengen (Formeln in konjunktiver Normalform).

Literal: A, $\neg A$, wobei $A \in V$ logische Variable (Atom) ist.

Klausel: Disjunktion von n Literalen ($n \geq 0$): 0, $[\neg]A_1$, $[\neg]A_1 \vee [\neg]A_2 \vee \ldots \vee [\neg]A_n$.

Klauselmenge: Menge (Konjunktion) von m Klauseln ($m \geq 0$).

Belegung: Funktion $\varphi : V \to \{0, 1\}$.

$A \in V$ ist wahr [falsch] unter einer Belegung φ, falls $\varphi(A) = 1$ [$\varphi(A) = 0$].

$\neg A$ ist wahr [falsch] unter einer Belegung φ, falls A unter φ falsch [wahr] ist.

Eine Klausel Γ ist wahr unter einer Belegung φ, falls *mindestens ein* Literal von Γ unter φ wahr ist. Anderenfalls ist Γ falsch unter φ.

Insbesondere folgt daraus: Eine *leere Klausel* ist unter jeder Belegung *falsch.*

Eine Klauselmenge M ist wahr unter einer Belegung φ, falls *jede* Klausel in M unter φ wahr ist. Anderenfalls ist M falsch unter φ.

Insbesondere folgt daraus: Eine *leere Klauselmenge* ist unter jeder Belegung *wahr.*

Eine Klausel oder Klauselmenge heißt *erfüllbar*, falls eine Belegung existiert, unter der sie wahr ist. (Anderenfalls heißt sie unerfüllbar.)

Eine Klausel oder Klauselmenge heißt *allgemeingültig*, falls sie unter jeder Belegung wahr ist.