7 Sprachkonzepte zur nebenläufigen und verteilten Programmierung

Bisher haben wir nur Programmiersprachen mit sequentiellen Berechnungsprinzipien betrachtet. Durch die starke Vernetzung von Rechnern und auch mehreren CPUs in einzelnen Rechnern ist auch das Rechnen in Prozessen und Netzwerken wichtig. In diesem Kapitel wollen wir hierzu geeignete Programmiersprachen und Sprachkonzepte betrachten.

7.1 Grundbegriffe und Probleme

Unter einem **Prozess** verstehen wir den Ablauf eines sequentiellen Programms, d. h. ein Prozess enthält alle dafür notwendigen Informationen, wie Programmzähler, Speicher, u.ä. Somit müssen wir folgende Begriffe auseinanderhalten:

- Programm: statische Beschreibung möglicher Abläufe
- Prozess: ein dynamischer Ablauf eines Programms

Der Ablauf eines Programms/Prozesses kann neue Prozesse erzeugen. Falls ein Programm mehrere Prozesse beschreibt, sprechen wir auch von einem **Thread** („Faden“, leichtgewichtiger Prozess): Darunter verstehen wir einen Ablauf in einem Programm, z. B. die Veränderung eines Programmzählers. Eine **multi-threaded language/system** enthält in der Regel mehrere Threads. Alternativ wird manchmal auch der Begriff **multi-processing** verwendet, aber dieser wird eher für Systeme mit mehreren Betriebssystemprozessen benutzt.

Die Motivation für die Mehrprozessprogrammierung besteht unter anderem in der Verfolgung der folgenden Ziele:

- Erhöhung der Effizienz (Nutzung mehrerer CPUs)
- natürliche Problemabstraktion:
  - physikalisch verteiltes System (z. B. Internet)
  - mehrere Benutzer, Simulationen, ...

Zum besseren Verständnis geben wir die folgende Begriffsabgrenzung an (dies ist leider nicht ganz einheitlich):

Nebenläufiges System: Enthält mehrere Threads/Prozesse

Verteiltes System: Prozesse laufen auf unterschiedlichen Prozessoren, die oft weit auseinderliegen, aber mit einem Netzwerk verbunden sind.

Paralleles System: Im Prinzip ist dies verteiltes System, aber die Prozessoren sind enger gekoppelt (z.B. gemeinsamer Speicher, schneller Bus). Daher spricht man auch oft von einem **Parallelrechner**.

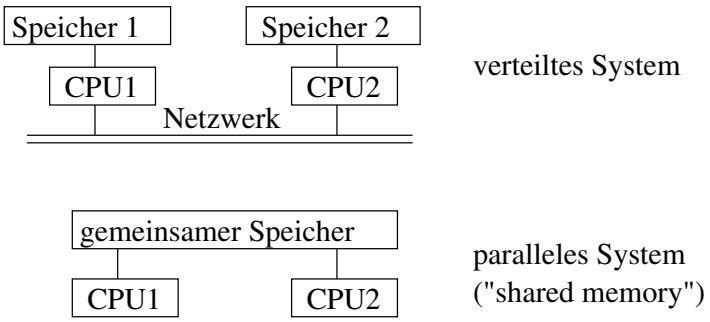

Abbildung 7.1: Vergleich verteiltes/paralleles System

Nebenläufigkeit/Verteiltheit: Dies ist eine logische Eigenschaft, die durch die Programmiersprache explizit unterstützt werden muss.

Parallelität: Dies ist eine Eigenschaft der Implementierung, die durch eine Programmiersprache unterstützt werden kann, aber auch implizit, z. B. durch einen parallelisierenden Compiler, realisiert wird. Allerdings sind die Probleme bei der Programmierung in beiden Fällen ähnlich.

In nebenläufigen Systemen können u.a. die folgenden Probleme auftreten:

1. Prozesse laufen nicht unabhängig ab.
2. Prozesse kopieren und tauschen Daten aus.
3. Prozesse greifen auf eine gemeinsame Datenbasis zu und verändern diese.
4. Prozesse haben Abhängigkeiten (das Ergebnis eines Prozesses wird von anderen Prozessen genutzt).
5. Prozesse müssen auf andere Prozesse warten.
6. Mehrere Prozesse müssen eventuell auf einem Prozessor ablaufen (interleaving).

Die Fälle 2 und 3 sind Probleme der *Kommunikation*, 4 und 5 Probleme der *Synchronisation*, und 6 ein Problem des *Scheduling* (dieses wird hier nicht weiter betrachtet, da dies eine typische Aufgabe des Betriebssystems ist). Kommunikation und Synchronisation hängen häufig eng zusammen, da oft durch Kommunikation auch synchronisiert wird. Ein wichtiger Mechanismus zur Lösung dieser Probleme ist der **gegenseitige Ausschluss**.

Beispiel 7.1 (Gegenseitiger Ausschluss)**.** Zwei Prozesse verändern gemeinsame Daten:

```
P1: ⋮              P2: ⋮
    x=x*2;             x=x+1;
    ⋮                  ⋮
```

Falls initial `x=0` gilt, sollte nach Ablauf von P1 und P2 `x=1` oder `x=2` gelten, je nachdem, welche Zuweisungsinstruktion eher ausgeführt wird. Allerdings ist eine Zuweisung wie `x=x*2` oder `x=x+1` keine Elementaroperation, sondern `x=x+1` besteht z.B. aus den Maschinenanweisungen

```
load x in a;
incr a;
store a in x
```

Somit ist auch die folgende Ausführung möglich (hier handelt es sich um ein Zeitdiagramm, wobei die Zeit von oben nach unten abläuft):

```
P1            x=0    P2
⋮                    ⋮
load:  a=0           ...
                     load: a=0
mult2: a=0
                     incr: a=1
               x=1   store
store          x=0
```

Eine Lösung dieses Problems wäre der gegenseitige Ausschluss der Modifikation von `x` in den Prozessen P1 und P2. Man sagt auch, dass „`x=x+1`“ ein **kritischer Bereich** ist.

Ein einfacher Mechanismus zur Realisierung des gegenseitigen Ausschlusses sind **Sperren** (**locks**) auf gemeinsamen Ressourcen.

Beispiel 7.2 (Locks)**.**

```
p1: ...
    lock(x);
    x:=x*2;
    unlock(x);
    ...
```

`lock(x)`: Prüfe, ob Objekt `x` gesperrt:

- Falls ja: warte, bis `x` nicht gesperrt.
- Falls nein: sperre `x`.

Diese Operation ist atomar (nicht unterbrechbar wie z.B. eine Zuweisung).

`unlock(x)`: Entsperre `x` (und aktiviere einen evtl. wartenden Prozess).

Hierdurch wird das obige Problem vermieden, aber es ergeben sich auch neue Probleme, die wir am klassischen Beispiel der dinierenden Philosophen erläutern wollen.

Beispiel 7.3 (Dinierende Philosophen (Dining Philosophers))**.**

- 5 Philosophen (Prozesse, $P_1, \ldots, P_5$) essen und denken abwechselnd
- Es befindet sich jeweils ein Stäbchen S_i zwischen den Philosophen P_i und P_{i+1} ($\approx$ gemeinsame Ressource).
- P_i benötigt zum Essen seine beiden Nachbarstäbchen S_i und S_{i+1}. Da die Philosophen im Kreis am Tisch sitzen, soll gelten: $S_6 \equiv S_1$.

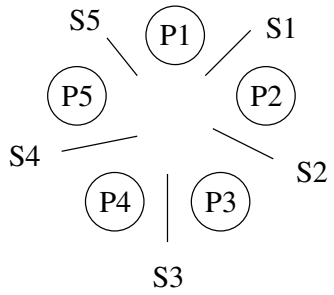

Abbildung 7.2: Struktur der essenden Philosophen

Je nach Programmierung können nun folgende Probleme auftreten:

Verklemmung (Deadlock) Das Programm für den Philosophen P_i lautet wie folgt ($i = 1, \ldots, n$):

```
loop
   lock(S_i);
   lock(S_{i+1});
   esse;
   unlock(S_i);
   unlock(S_{i+1});
   denke;
end
```

Falls alle P_i gleichzeitig `lock(`S_i`)` ausführen, sind alle Philosophen blockiert.

Blockade (Livelock) Es entsteht kein Deadlock, aber kein Prozess macht Fortschritte. Hierbei ist der Code wie oben, aber nach `lock(`S_i`)` wird nun `unlock(`S_i`)` ausgeführt, falls S_{i+1} gesperrt ist. Hierdurch können eventuell alle Philosophen P_i in die folgende Schleife geraten:

```
lock(Si);
unlock(Si);
lock(Si);
unlock(Si);
⋮
```

Fairness Jeder Philosoph P_i, der essen möchte, soll dies irgendwann tun können. Eine unfaire Lösung wäre: Lasse nur P_1 und P_3 essen. Ebenfalls unfair(!) ist: Lasse reihum P_1, P_2, P_3, P_4, P_5 essen. Dies ist unfair, denn die Philosophen haben verschieden viel Hunger und brauchen unterschiedlich lange zum Essen.

Busy Waiting Falls `lock(`S_i`)` ausgeführt werden soll, prüfe immer wieder, ob die Sperre auf S_i noch vorhanden ist. Diese Prüfschleife belastet den Prozessor und sollte vermieden werden. Wie man dies machen kann, wird im nächsten Abschnitt erläutert.