2.7 Funktionen/Rekursive Aufrufe

2.7.1 Funktionen, die sich selbst aufrufen – geht das?

Im folgenden Programm gibt es eine Funktion, die sich selbst aufruft:

```
1  #include <stdio.h>
2
3  void funk(int x)
4  {
5      printf("Davor....x=%d\n",x);
6      if(x>0)
7          funk(x-1);
8      printf("Danach...x=%d\n",x);
9  }
10
11 int main(void)
12 {
13     funk(6);
14     return 0;
15 }
```

Wenn man das Programm compiliert und ausführt, sieht es so aus:

Terminal

```
Davor....x=6
Davor....x=5
Davor....x=4
Davor....x=3
Davor....x=2
Davor....x=1
Davor....x=0
Danach...x=0
Danach...x=1
Danach...x=2
Danach...x=3
Danach...x=4
Danach...x=5
Danach...x=6
```

Offenbar wird die Funktion `funk()` sieben mal aufgerufen, einmal durch `main()` mit x=6 und die anderen Male durch `funk()` selbst. Im RAM sieht es dann aus wie in Abb. 1 angedeutet.

2.7.2 Rekursion

In der Informatik nennt man dieses Ereignis, dass sich eine Funktion selbst aufruft, *Rekursion*, genauer gesagt, direkte Rekursion[1].

Damit die Rekursion funktioniert, müssen zwei Dinge sichergestellt sein:

a) Die Rücksprungadresse muss jedes Mal, also für jeden Aufruf der Funktion, an einer neuen Stelle gespeichert werden.

   Das ist der Fall, weil alle Systeme (von ein paar 4- und 8-Bit-Systemen, die mehr als 40 Jahre alt sind, abgesehen) über einen so genannten *Stack* verfügen (auf deutsch Stapelspeicher), den man sich vorstellen kann wie einen Tellerstapel im Küchenschrank: Man legt den als

[1] Indirekte Rekursion liegt dann vor, wenn sich zwei (oder mehr) Funktionen gegenseitig aufrufen, also A ruft B *und* B ruft A.

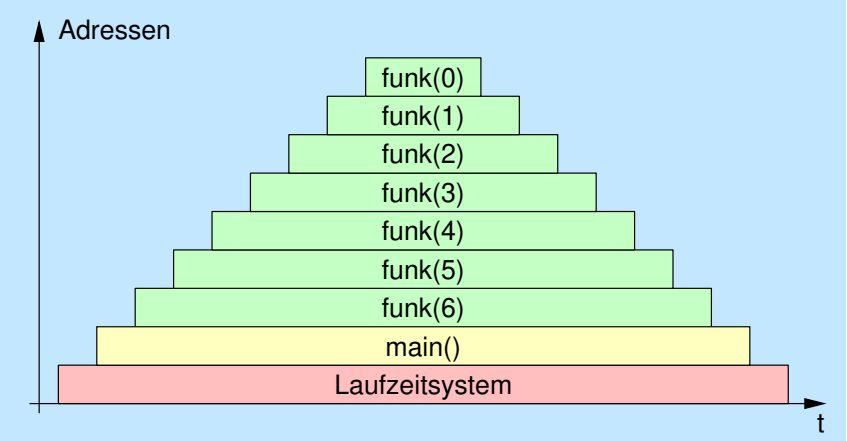

Abbildung 1: RAM-Aufteilung, wenn sich eine Funktion selbst aufruft (Rekursion)

letztes gewaschenen Teller oben auf den Stapel. Wenn man einen Teller braucht, nimmt man wiederum von oben einen Teller vom Stapel herunter.

Und genau dieser *Stack* ist in Abbildung 1 dargestellt, allerdings nicht in Einzelheiten.

b) Nun wird die Funktion einmal mit x=6 aufgerufen, einmal mit x=5 usw. Diese Zahl kann nicht immer an derselben Stelle abgelegt sein, sonst hätten wir nicht die oben gezeigte Ausgabe des Programms.

Offenbar braucht jede Version der Funktion – man sagt: jede *Instanz* – auch einen eigenen Bereich für Daten, also für:

- Parameter
- Rückgabewert[2]
- so genannte lokale Variablen in den Funktionen (!)

Nur dann kann jede Instanz auch eigenständig arbeiten.

Das Problem ist also ähnlich wie bei der Rücksprungadresse – und deshalb nimmt man auch dieselbe Lösung: Auch diese Daten werden auf dem Stack abgelegt[3].

[2] nicht ganz zwingend, da zu jeder Zeit immer nur eine Funktion beendet wird

[3] Es sei denn, man hält z. B. die Parameter in bestimmten CPU-Registern, dann muss man im Unterprogramm eben die Register auf dem Stack sichern

2.7.3 Aufbau des Stacks

Am folgenden Programmbeispiel (`main()` ruft eine Funktion) kann man dann zeigen, wie der Stack aufgebaut wird:

```
1  #include <stdio.h>
2  double funktion(int a, long b)
3  {
4      char c;
5      double d;
6      printf("a:%p, b:%p, c:%p, d:%p\n", &a,&b,&c,&d);
7      d=(double)a*b;
8      return d;
9  }
10 int main(void)
11 {
12     double x;
13     printf("x:%p, main:%p\n", &x);
14     x=funktion(3, 5);
15     return 0;
16 }
```

Terminal

```
schueler@debian964:~$ gcc src/stackbsp.c
schueler@debian964:~$ src/a.out
x:0xbfe4c358
a:0xbfe4c340
b:0xbfe4c344
c:0xbfe4c32f
d:0xbfe4c320
```

Durch die `printf()`-Aufrufe sieht man, an welchen Adressen die einzelnen Parameter und Variablen liegen. Die Lage der Rücksprungadressen und der Rückgabewerte kann man nicht *so* einfach ermitteln, sie sind aber auch kein Geheimnis. Der Stack enthält damit für jedes Unterprogramm

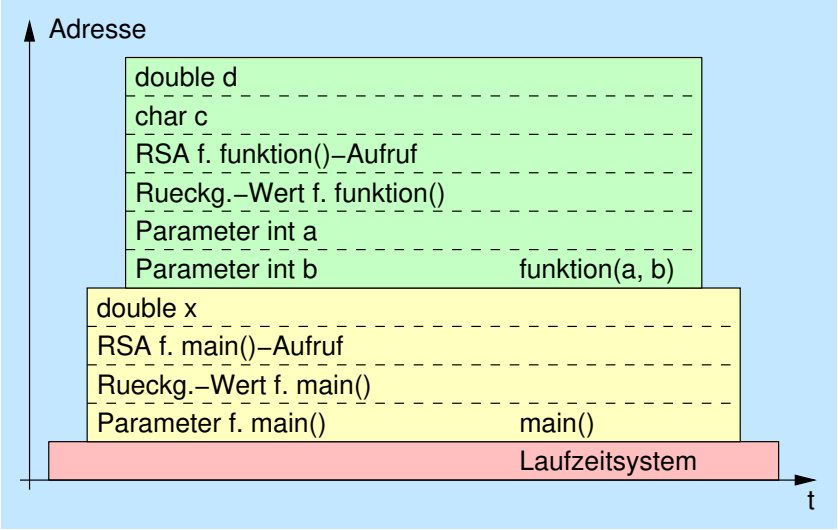

Abbildung 2: Ungefähres Aussehen des Stacks beim Beispielprogramm

ungefähr[4] ein Aussehen wie in Abbildung 2 – nur mit dem Unterschied, dass bei uns die Stackadressen mit zunehmender Nutzung fallen – der Stack hängt bei PCs quasi „von der Decke“.

2.7.4 Wozu braucht man Rekursion?

Manche Probleme lassen sich mit Rekursion einfacher lösen als ohne, z. B. eine Funktion, die die Fakultät einer Zahl berechnet. Diese Funktion braucht man oft in der Statistik. Die Fakultät von 4 berechnet sich zu $4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24$. Die Fakultät von x ist also x mal so groß wie die von x-1:

```
1  #include <stdio.h>
2  unsigned int fak(unsigned int x)
3  {
4      if(x==1 || x==0)
5          return 1;
6      return x*fak(x-1);
7  }
8  int main(void)
9  {
10     printf("%u\n", fak(4));
11     return 0;
12 }
```

Ohne Rekursion geht es in diesem Fall aber auch:

```
1  #include <stdio.h>
2  unsigned int fak(unsigned int x)
3  {
4      unsigned int lauf;
5      unsigned int erg=1;
6      if(x==0) return 1;
7      for(lauf=2; lauf<=x; ++lauf)
8          erg=erg*lauf;
9      return erg;
10 }
11 int main(void)
12 {
13     printf("%u\n", fak(4));
14     return 0;
15 }
```

Die Version mit Rekursion ist einfacher zu programmieren, im Programmlauf aber wahrscheinlich etwas langsamer, weil so viele Funktionsaufrufe mit Stack-Aufbau passieren müssen.

In den Programmen `printint.c` und `nullstellen.c` findet man weitere Beispiele für Rekursion, die andernfalls viel komplizierter zu programmieren wären.

[4] Nicht ganz genau, z. B. fehlt das gesicherte *base pointer*-Register