Informatik Rechnerinterne Vorgänge: Der Stack

Gierhardt

Der Code und die einfachen Daten eines Programmes beanspruchen einen festen Bereich im Rechnerspeicher. In einem Java-Programm sind die globalen Variablen zu deklarieren, damit der Compiler den nötigen Speicherplatz berechnen kann. Während des Programmlaufes können sich natürlich noch die Werte der Variablen ändern, nicht aber der dazu erforderliche Speicherplatz.
In vielen Situationen ist es aber notwendig und sinnvoll, einen dynamisch mit den Anforderungen wachsenden Datenbereich zu haben. Beispiele:

1. Beim Aufruf eines Unterprogrammes (Methode) muss irgendwo die **Rücksprungadresse** gespeichert werden. Bei Rekursion ist die Anzahl der nötigen Speicheradressen vorher nicht bekannt.

2. Unterprogramme bzw. Methoden können mit **lokalen Variablen** arbeiten, die nur während des Ablaufs des Unterprogrammes Speicherplatz beanspruchen. Nach Abarbeitung des Unterprogrammes wird der nötige Speicher wieder freigegeben.

3. Unterprogrammen können **Parameter** übergeben werden, die nur während des Ablaufs des Unterprogrammes benötigt werden. Bei Rekursion müssen die Parameter aller noch laufenden Unterprogramm-Instanzen gespeichert bleiben.

4. Funktionen müssen über irgendeine Speicheradresse ihre **Rückgabewerte** an das aufrufende Programm übergeben können.

Für all dies ist der sogenannte *Stack* (Stapel) zuständig.
Für weitere dynamische Datenstrukturen (z.B. Baumstrukturen, Listen u.a.) werden wir später den *Heap* (Haufen) kennen lernen.
Man kann sich den Stack als einen Stapel bestehend aus Zetteln vorstellen, wobei man immer nur den obersten Zettel lesen, einen neuen Zettel drauflegen oder einen Zettel wegnehmen kann.
Beim DC muss man sich diesen Stapel auf dem Kopf stehend vorstellen, denn der Stack wächst hierbei von der Speicherstelle 127 an abwärts. In dem SP (=StackPointer) genannten Register steht immer die Adresse der nächsten freien Speicherstelle auf dem Stack, zu Beginn eines Programmes steht darin also immer die Zahl 127.

1. **Rücksprungadresse:** Beim Aufruf eines Unterprogrammes wird durch den Befehl `JSR` der Wert der im Programm folgenden Speicheradresse auf dem Stack abgelegt und der StackPointer um 1 erniedrigt. Beim Auftreten von `RTN` wird der StackPointer zuerst um 1 erhöht und dann der Wert an dieser Stelle in den PC (Program Counter) übertragen. Damit kann das Programm an der Stelle weitermachen, die dem Aufruf des Unterprogrammes folgt.

2. **Lokale Variablen:** Nach dem Sprung zum Programmcode des Unterprogrammes kann durch `PSH`-Befehle auf dem Stack Platz für lokale Variable geschaffen werden. Dadurch ändert sich natürlich der SP. Durch spezielle Befehle, die Adressen relativ zum SP ansprechen können, kann mit diesen lokalen Variablen gearbeitet werden. Vor dem Befehl `RTN` müssen die zu Beginn des Unterprogrammes mit `PSH` angeforderten Speicherplätze durch die gleiche Anzahl an `POP`-Befehlen wieder freigegeben werden,

damit die Rücksprungadresse bereit steht. Achtung: Im Akkumulator steht nach den `POP`-Befehlen ein Wert einer lokalen Variablen.

3. **Parameter** für ein Unterprogramm müssen vor dem `JSR`-Befehl mit `PSH` auf den Stack gebracht werden. Sie stehen dann auf dem Stack oberhalb der Rücksprungadresse. Mit den speziellen Stack-Befehlen kann man auch auf diese Variablen zugreifen. Nach dem Rücksprung müss der benötigte Speicherplatz für die Parameter mit der entsprechenden Anzahl an `POP`-Befehlen wieder freigegeben werden. Achtung: Im Akkumulator steht nach den `POP`-Befehlen ein Wert einer lokalen Variablen, der aber nicht mehr benötigt wird.

4. **Rückgabewerte** kann man wie Parameter behandeln. Mit `PSH` macht man Platz für einen solchen Wert, wobei dabei der übergebene Wert unwichtig ist. Mit den speziellen Stackbefehlen kann die entsprechende Speicherstelle beschrieben werden. Der Wert, den man nach dem `RTN` durch `POP` im Akkumulator erhält, ist der Funktionswert des Unterprogrammes.

5. Bei rekursiven Aufrufen von Unterprogrammen mit Parametern ändert sich während des Laufes des Unterprogrammes der SP durch die notwendigen `PSH`-Befehle. Um trotzdem auf die lokalen Variablen und die Parameter mit fester Adresse zugreifen zu können, gibt es neben dem SP das Hilfsregister BP (=BasePointer), in das der SP zu Beginn eines Unterprogrammes kopiert werden kann. Es gibt außerdem spezielle Befehle, die die Adressierung des Stacks über den BP ermöglichen. Am Ende des Unterprogrammes kopiert man dann den BP zurück in den SP.

Beispiel: Das folgende Programm berechnet die Summe der ganzen Zahlen von 1 bis einer einzulesenden Zahl k. Die Berechnung erfolgt mit einer `while`Schleife in einer Methode `int addiere(int zahl)` mit Parameterübergabe und Rückgabewert.

```
1  ; PROGRAM SumJSRSt.DCL
2  ; Die ganzen Zahlen von 1 bis k werden addiert
3  ; mit einer WHILE-Schleife in einer Methode Addiere
4  ; inklusive Parameteruebergabe und Rueckgabewert
5
6          JMP Anfang
7
8  null    DEF 000
9  summe   DEF 000
10 k       DEF 5
11
12 Anfang                      ; public void main(...)
13         INM k               ; {  In.read(k);
14         LDA k
15         PSH                 ;    // Platz fuer Funktionserg. (returnwert), Wert unwichtig
16         PSH                 ;    // k auf Stack, in zahl kopieren
17         JSR addiere         ;    // Methodenaufruf
18         POP                 ;    // lokale Variable bzw. Parameter zahl vernichten
19         POP                 ;    // Funktionsergebnis vom Stack nehmen in Akku
20         STA summe           ;    summe = addiere(k);
21         OUT summe           ;    Out.print(summe);
22         END                 ; }
23
24 addiere                      ; int addiere(int zahl)
25         PSH                 ; {  int ergebnis (Platz fuer lokale Variable)
26         returnwert EQUAL 4  ;    //  SP + 4
27         zahl       EQUAL 3  ;    //  SP + 3
28         RSprAdr    EQUAL 2  ;    //  SP + 2
29         ergebnis   EQUAL 1  ;    //  SP + 1
30         LDA null            ;
31         STAS ergebnis       ;      ergebnis = 0;
32 WhileAnfang
33         LDAS zahl           ;      while (zahl > 0)
34         JNP WhileEnde       ;      {
35         LDAS ergebnis       ;
36         ADDS zahl           ;
37         STAS ergebnis       ;        ergebnis = ergebnis + zahl;
38         LDAS zahl
39         DEC
40         STAS zahl           ;        zahl = zahl - 1;
41         JMP WhileAnfang     ;      } // end von while
42 WhileEnde                   ;
43         LDAS ergebnis
44         STAS returnwert     ;      return ergebnis;
45         POP                 ;      lok. Variable ergebnis vernichten
46         RTN                 ;    } end von addiere
```

Aufgaben:

1. Das oben dargestellte Programm ist zu analysieren. Es wird z.B. die Zahl 3 eingelesen. Zeichne den Stack mit vollständigem Inhalt,

   (a) nachdem `JSR addiere` ausgeführt wurde.

   (b) wenn zum ersten Mal im Programm `WhileAnfang` erreicht wird.

   (c) bevor `RTN` ausgeführt wird.

   (d) bevor `OUT summe` ausgeführt wird.

2. Schreibe ein Programm, das das Quadrat einer beliebigen Zahl berechnet:

   (a) mit einem Unterprogramm `quadrat` ohne Parameter und Rückgabewert.

   (b) mit einem Unterprogramm `quadrat`, das die zu quadrierende Zahl als Parameter über den Stack geliefert bekommt.

   (c) mit einem Unterprogramm `quadrat`, das die zu quadrierende Zahl als Parameter über den Stack geliefert bekommt und das Ergebnis als Rückgabewert bzw. Funktionswert zurück liefert.

3. Was bewirkt das folgende Programm?

```
1         JSR WasIstDas
2         END
3  WasIstDas                    ; void WasIstDas()
4                               ; {   Ruecksprungadr.  SP+2
5                               ;   int zahl;           SP+1
6         zahl EQUAL 1          ;                      SP
7         PSH                   ;   // Platz fuer zahl schaffen
8         INS zahl              ;   zahl = In.readInt();
9         OUS zahl              ;   Out.print(Zahl);
10        LDS zahl              ;
11        JZE EndIf             ;   IF (    )
12                              ;       {
13        JSR WasIstDas         ;        WasistDas();
14        OUS zahl              ;        Out.print(zahl);
15                              ;       }
16 EndIf
17        POP                   ;   // Platz fuer zahl freigeben
18        RTN                   ; } // Ende von WasIstDas
```

4. Das Programm mit der Methode `WasistDas` ist so umzuformulieren, dass statt des SP der BP für die Adressierung verwendet wird.

5. Schwierig! Die Fibonacci-Zahlen sollen rekursiv ausgegeben werden.

6. Schwierig! Das Problem der „Türme von Hanoi“ soll mit DC gelöst werden.