Abgabe von Bruno Stendal, Martin Baer, Lukas Gewinner und Christian Schäfer
5. Aufgabenblatt zum Kurs

TI2: Rechnerarchitektur

von Bernadette Keßler

bis Freitag, den .2022, 10:15 Uhr.

Stack

Beantworten Sie die nachfolgenden Fragen und geben ggf. Ihre Quellen an.

1. Ein Stack ist ein Teil des Hauptspeichers, welcher stapelartig aufgebaut ist. Der „Stapel“ kann Objekte aufnehmen oder Elemente wieder abgeben (speichert zum Beispiel Program Status Word oder Rücksprungadressen beim Aufruf von Subroutinen) nach dem Last-In-First-Out-Prinzip (LIFO). Dabei werden mit den Maschinenbefehlen Push und Pop (Pull) Elemente abgelegt oder weggenommen. Push legt ein Element oben ab, wodurch der Stack nach unten Richtung „Keller“ wächst. Pop nimmt ein Element oben weg, wodurch der Stack kleiner wird (freier Speicher/Keller wird größer). Das ermöglicht, dass das oberste Element einfach mit dem Offset null adressiert werden kann. Hardware unterstützte Stacks sind direkter Teil des Hauptspeichers. Dafür gibt es spezielle Register mit dem „Stack Pointer“. Dieser enthält eine Speicheradresse und dient damit der Ansteuerung eines Stacks, indem auf die Spitze des Stacks gezeigt wird.

2. Jmp springt zu einer Speicheradresse, wodurch sich der Zeiger dauerhaft ändert. Call hingegen speichert die aktuelle Adresse, weil danach wieder auf die ursprüngliche Adresse zurückgegriffen wird mittels einer return Anweisung. Deshalb kann dies nach dem LIFO-Prinzip mit einem Stack umgesetzt werden. Jmp ist Teil eines Programms und springt in der Reihenfolge. Call initialisiert eine Subroutine, die dann am Ende an die Anfangsposition zurückkehrt, wodurch die Reihenfolge erhalten bleibt.

3. Statische Speicherallokation von globalen Variablen führen dazu, dass Variablen ihren Wert beibehalten, auch wenn sie innerhalb von Funktionen benutzt werden. Automatische Speicherallokation: Variablen, die lokal für Funktionen deklariert werden, werden auf dem Stack gespeichert und nach dem die Funktion zu Ende ist wieder entfernt, wodurch wieder Speicher freigegeben wird. So entsteht ein dynamischer Speicherbereich.

4. Ein Stackframe ist eine Gruppe bestehend aus Parametern, lokale Variablen und Rücksprungadressen. Es ist ein Art Rahmen, welcher um einen Teil des Stacks gezogen wird und gruppiert diesen somit in eine gewisse Anzahl an Stackframes. Ebenfalls im Frame ist ein Base Pointer. Dieser ist ein Register, welches auf eine fixierte Adresse zeigt. Bei Funktionen innerhalb eines Stacksframes wird somit ein konstanter Bezugspunkt definiert, wodurch dem Programm die Ermittlung der „Entfernung“ oder Position einer Variable einfacher gemacht wird. Der Leave-Befehl setzt den neuen Stack Pointer nach einem Funktionsaufruf in einem Stackframe, indem der Stack Pointer auf den Base Pointer gesetzt wird und dieser dann mit Pop entfernt wird. Anders gesagt „zerstört“ der Befehl den aktuellen Stackframe, wohingegen Enter einen neuen Frame anlegt, indem mit Push ein Base Pointer angelegt und mit dem Stack Pointer adressiert wird.

Fibonacci Zahlen

Auf dem letzten Zettel haben Sie die Wiederholung von Instruktionen durch Sprünge kennengelernt. Diese Art der Wiederholung wird Iteration genannt. Die andere Art der Wiederholung ist Rekursion. Machen Sie sich mit Rekursion auf Assemblerebene und dafür mit dem Callstack vertraut (Befehle: PUSH, POP, CALL, RET). Schreiben Sie in Assembler eine iterative und eine rekursive Fibonacci-Funktion.

```
1  Global asm_fib_it, asm_fib_rek
2
3  SECTION .text
4
5  asm_fib_it:
6                          ;rdi = n
7      XOR rdx, rdx        ;rdx = x = 0
8      MOV rcx, 1          ;rcx = y = 1
9      XOR r8, r8          ;r8 = k = 0
10     .schleife:
11         CMP rdi, 0      ;rdi wird mit Null verglichen.
12         JG .calc        ;wenn edi gr   er ist als Null, dann
       springt er zu .calc.
13         MOV rax, r8     ;der Wert in r8(k) wird auf rax
       verschoben, wobei r8 den Wert beh lt.
14         RET             ;gibt rax aus.
15
16
17
18     .calc:
19         MOV rdx, rcx    ;der Wert in rcx(y) wird nach rdx(x)
       verschoben, wobei rcx den Wert beh lt.
20         MOV rcx, r8     ;der Wert in r8(k) wird nach rcx(y)
       verschoben, wobei r8 den Wert beh lt.
21         MOV r10, rdx    ;der Wert in rdx(x) wird nach r10
       verschoben, wobei rdx den Wert beh lt.
22         ADD r10, rcx    ;r10 = r10(x) + rcx(y)
23         MOV r8, r10     ;das Ergebniss aus der Addition wird
       jetzt nach r8(k) verschoben, aus r10.
24         DEC rdi         ;rdi(n) wird im eins kleiner, also rdi(n
       ) = rdi(n) - 1
25         JMP .schleife   ;sprint zurueck zur Funktion .schleife.
26
27
28
29 asm_fib_rek:
30         mov rax, 0      ;ergebnis ausnullen zur sicherheit
31         mov rcx, 0      ;es muessen diese Register vor der
```

```
   Rekursion auf Null gesetzt werden
32
33 .check:                  ;check ob rdi unter 3
34         cmp rdi, 3       ;pseudocode zeile 2
35         jb .end          ;pseudocode zeile 3
36         jmp .fib_rek  ;pseudocode else statement
37
38 .end:
39         mov rax, 1       ;falls rekursions anker erreicht wird,
     rueckgabe 1
40         ret              ;rueckgabe -> rekursions schritt hoch
41
42 .fib_rek:
43         push rcx         ;rcx zweiter counter pushen
44
45         sub rdi, 1       ;counter herrab setzten
46         push rax         ;ergebnis pushen
47         push rdi         ;counter pushen -> fuer rueck operation
48
49         call .check      ;sprung fuer n-1
50
51         mov rcx, rax     ;ergebis sichern
52
53         pop rdi          ;counter pushen -> zeile 42
54         pop rax          ;ergebnis pushen
55         sub rdi, 1       ;fall n-2 (2mal -1, siehe zeile 40)
56
57         call .check      ;n-2 sprung
58
59         add rax, rcx     ;ergebnisse addieren
60         pop rcx          ;counter popen, wiederherstellen
61         ret              ;rueckgabe -> rekursions schritt hoch
```

Erklären Sie die Zeitunterschiede sowohl zwischen den rekursiven und iterativen Funktionen als auch zwischen den C und Assembler Funktionen. Warum wird Rekursion überhaupt benötigt?
Der Zeitunterschied zwischen der Iterativen und der Rekursiven Funktion entsteht aufgrund des Overheads.
In unserem Beispiel besitzt die Rekursionslösung zwei Nachteile.

- Laufzeit, aufgrund des erzeugten Overhead durch Funktions-calls dauert die Funktion deutlich länger
- Speicher, durch die mitunter große Rekursionstiefe werden die Stacks sehr groß und damit die Anwendung Speicherintensiv

Die Vorteile von Rekursion sind:

- Einfacherer Code bei komplexen Problemen, viele Mathematische Probleme lassen sich einfacher in einer Rekursiven Form anstatt einer Iterativen Form abbilden
- Rekursive Funktionen, können der Rekursion Informationen in Form von Parametern übergeben, womit man komplexere Abhängigkeiten modellieren kann als bei der Iterationen Funktion.
- Bestimmte Operationen auf Daten sind Rekursiv einfacher z.B. Rotation von AVL-Bäumen können einfacher modelliert werden, da die Rotation des Teilbaumes einfach in einer Rekursion stattfinden kann, wo dieser der die Wurzel darstellt.

Der C Code ist in unserem Fall schneller, da dieser durch den Compiler, da die -O2 Flag im Makefile gesetzt ist, optimiert wird. Durch die -O2 Flag, welche identisch zur -xO2 Flag ist, werden unter anderem Algebraische-, Kopier-, Rekrusions- und Schleifenoptimierung durchgeführt.

Die Zahl innerhalb der Flag steht dabei für das Level der Optimierung wobei 1, das geringste und 5 das höchste mit den größten Auswirkungen ist.

Somit ist der NASM Code langsamer da er unoptimiert direkt übersetzt und ausgeführt wird.

Quelle für die Spezifizierung der -x02 Quelle: Oracle CC Users Guide