Funktionen: global und lokal

Montag, 6. Mai 2019 07:30

Funktionen in C

```
f(...) {...}
g(...) {...}
```

f ruft g auf
g ruft g auf
:

⟹ Rückwärts verkettete Liste(!)

Objekthierarchie

Referenzen auf Objekte
≙ 2 Pointern

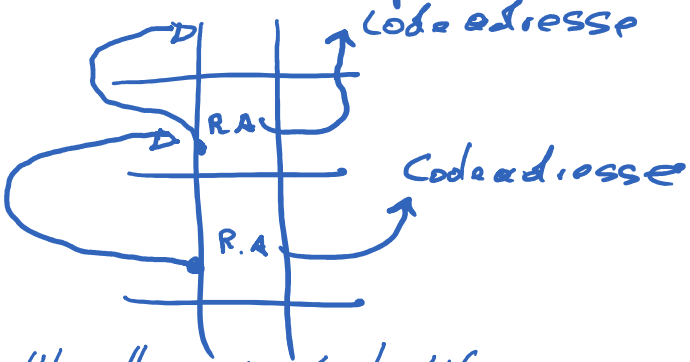
Klasse 1, S1 — 01
|
Klasse 2, S2 — 02
|
Klasse 3, S3 — 03

Klasse3 mo:
statisch: Liste
dynamisch: rückwärts verketteten Baum

Verwaltet:

Klassisch Stack

Verkettung mit Returnadresse und Framepointer



pro aktivem Funktionsaufruf:
- Parameter
- Returnadresse
- lokale Variablen und
- Verwaltungsdaten

Name: Frame / Rahmen

aktuelle Architekturen

Kontextwechsel, vgl. Listenverwaltung

Benötigt: ESP, EIP, EBP

EBP: Base Pointer, Frame Pointer

```
int f(int n) {(*)
    int l;

(**) return l;
}
```

```
int i = f(5);
        ↓
push 5
call f
add ESP, 4
```

callee ↔ Aligned Stackframe ↔ caller

callee saved Register

caller saved

Bsp.:

```
mov EAX, 5
push 2
call f
add ESP, 4
```

f(2);

push EAX

EAX = ? (hier im Bsp. 2)

pop EAX

Caller Saved

Caller - saved	Caller -
ESP	EAX
EBP	ECX
EBX	EDX
ESI	
EDI	

Warum die Aufteilung:

1. Es muss Caller-Saved Register geben

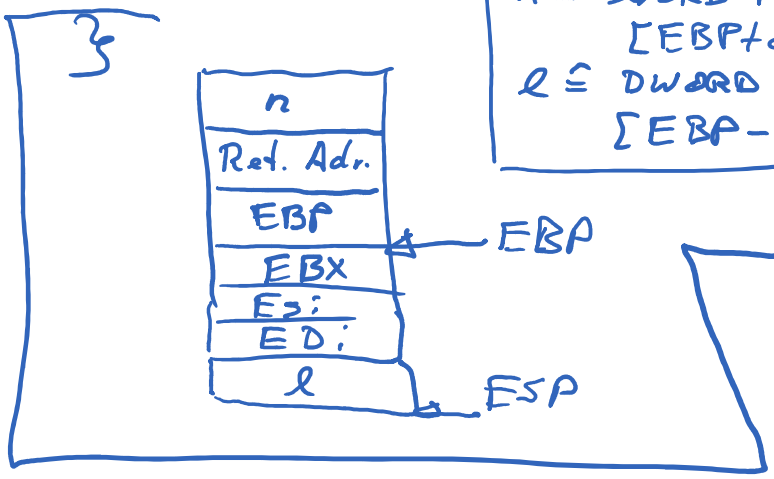
2. Arithmetische Operationen sind "einfacher" als Adressberechnung

Auf-/Abbau des Stackframes:

```
int f(int n){
  int l;
  return n;
}
```

Prolog:
```
push EBP
mov EBP, ESP
push EBX
push ESI
push EDI
sub ESP, 4
```

Anm.:
n ≙ DWORD PTR [EBP+8]
l ≙ DWORD PTR [EBP-16]



EPILOG:
```
mov EAX, DWORD PTR [EBP+8]
Add ESP, 4
pop EDI
pop ESI
pop EBX
pop EBP
Ret
```

Deklarationsspezifikation:

```
__declspec( naked )
```

Anmerkungen:

1. Return:

In C:
```
push Parameter
call fkt
add ESP, n
und in fkt:
  ⋮
ret
```

In Java
```
push Parameter
call fkt
  ⋮
und in fkt
  ⋮
ret n
```

2. Enter / Leave:

Enter Bytes, 0

≙ push EBP
Mov EBP, ESP
Sub ESP, Bytes

Leave:

Mov ESP, EBP
pop EBP

3. Adressieren von Werten im Stackframe:

über

EBP

+ feste Offsets
− ein Register weniger verfügbar
+ Allokieren dynamisch Speicher im Stack ist möglich
Subtraktion von ESP
≙ _alloca

ESP

nur malloc → Heap

4. push und pop Sequenzen

brauchen Codebytes + Zeit

1/2 Register, Caller, andere Hälfte Caller saved

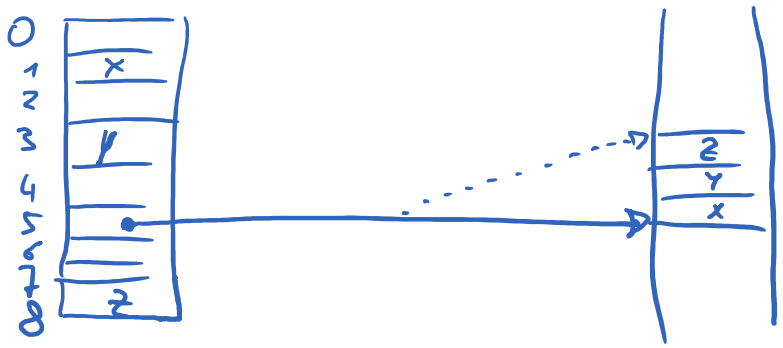
1/2 Context

Gesamt context wechsel:
pushall / popall

Aktuelle Architekturen, z.B. ARM

Multiple Load / Store Instruktionen

ldm {R1, R3, R8}, R5



ldm {....}, R5++ | ++R5
--R5
R5--

$\Rightarrow$ Push/Pop Sequenzen werden bei ARM auf je eine Instruktion reduziert

5. Registerallokation:

Optimiere Zuordnung von Werten zu Registern

Ziel: Anzahl der Speicherzugriffe reduzieren

Idee: