

Rechnerorganisation: Realisierungsmöglichkeiten Prozeduraufruf

Robert Tolksdorf
Freie Universität Berlin

[1] © Robert Tolksdorf, Berlin

Prozeduraufrufe

- Maschinenbefehle wie JSR / RTS oder CALL / RET leiten nur den Kontrollfluss geeignet um
 - Sichern der Adresse des nächsten Befehls auf Stack
 - Sprung
 - Laden der Rücksprungadresse vom Stack
 - Rücksprung
- Für
 - Parameterübergabe
 - Ergebnisübergabeist eine *Aufrufkonvention* nötig
- Kann sein
 - sprachspezifisch
 - compilerspezifisch
 - prozessorspezifisch

[2] © Robert Tolksdorf, Berlin

Kontrollflusssteuerung

- Statisch:
 - 08: ...
 - 09: jmp 20
 - 10: ...
 - 20: ... /* Start Unterprogramm */
 - ...
 - 35: jmp 10 /* Ende Unterprogramm */
- Kann nicht an anderer Stelle ausgeführt werden, Ziel- und Rücksprungadresse fest

[3] © Robert Tolksdorf, Berlin

Kontrollflusssteuerung

- Rücksprungadresse in Register halten:
 - 08: ...
 - 09: movl \$11, %eax
 - 10: jmp 20
 - 11: ...
 - 12: movl \$14, %eax
 - 13: jmp 20
 - 14: ...
 - ...
 - 20: ... /* Start Unterprogramm */
 - ...
 - 35: jmp %eax /* Ende Unterprogramm */
- Keine Rekursion möglich

[4] © Robert Tolksdorf, Berlin

Kontrollflusssteuerung

- Rücksprungadresse auf Stack legen:
08: ...
09: `movl $11, (%esp)+`
10: `jmp 20`
11: ...
20: ... /* Start Unterprogramm "func" */
...
35: `jmp -(%esp)`
- Verträgt Rekursion
- Stack kann
 - vom hohen zum niedrigen Speicherbereich wachsen (80x86)
 - vom niedrigen zum hohen Speicherbereich wachsen
- Stackpointer kann auf
 - letzten abgelegten Wert zeigen (80x86)
 - auf nächsten freien Platz zeigen

[5] © Robert Tolksdorf, Berlin

Kontrollflusssteuerung

- Stack kann
 - vom hohen zum niedrigen Speicherbereich wachsen (80x86)
 - vom niedrigen zum hohen Speicherbereich wachsen
- Stackpointer kann auf
 - letzten abgelegten Wert zeigen (80x86)
 - auf nächsten freien Platz zeigen
- Synonyme 80x86:
`movl $11, -(%esp); jmp 20` `call 20`
`jmp (%esp)+` `ret`

[6] © Robert Tolksdorf, Berlin

Werteübergabe

- Aufrufer und Aufgerufener kommunizieren
 - Parameter
 - Ergebnisse
- Mögliche „Übergabeorte“
 - Register
 - Anzahl beschränkt
 - nicht geeignet für rekursive Unterprogramme
 - festgelegte Speicherzellen
 - langsam
 - nicht geeignet für rekursive Unterprogramme
 - Stack
 - langsam
 - geeignet für rekursive Unterprogramme

[7] © Robert Tolksdorf, Berlin

Übergabe über Register

- Hauptprogramm:
...
`movl y, %eax` `/* x = subtract(y, z); */`
`movl z, %ebx`
`call subtract`
`movl %eax, x`
...- Unterprogramm `subtract(a, b)`:
`subtract:`
`subl %ebx, %eax` `/* return a - b; */`
`ret`

[8] © Robert Tolksdorf, Berlin

Parameter auf Stack, Ergebnis in Register

- Hauptprogramm:

```
movl z, -(%esp)          /* x = subtract(y, z); */
movl y, -(%esp)
call subtract
addl $8, %esp
movl %eax, x
...
```
- Unterprogramm `subtract(a, b)`:

```
subtract:
movl 4(%esp), %eax       /* return a - b; */
subl 8(%esp), %eaxret
```

[9] © Robert Tolksdorf, Berlin

Lokale Variablen

- Unterprogramm benutzt lokale Variablen, ihre Lebensdauer ist gleich der Lebensdauer der Prozedur
- Mögliche Speicherorte:
 - Register-Variablen
 - für die am häufigsten verwendeten Variablen (z.B. Schleifen-Zähler und temporäre Variablen zur Ausdrucksberechnung)
 - Variablen im Speicher an festgelegten Adressen
 - für Werte, die beim wiederholten Aufruf des Unterprogramms intakt sein müssen (static-Werte)
 - Variablen auf dem Stack
 - für seltener verwendete Variablen oder große Variablen (z.B. Records oder Arrays)
 - Zwischenspeicher für Register-Variablen (z.B. für Rekursion)

[10] © Robert Tolksdorf, Berlin

Beispiel

```
int proc(void)
{
    static int init = 0; /* Speicher an fester Adresse */
    register int i;      /* Register-Variable */
    int b;               /* Variable auf Stack */
    if (!init) {
        initialize();
        init = 1;
    }
    for (i = 0; i < 1000; i++) {
        if (test(i)) printf("%d\n", i);
    }
    b = i + 1;
    return b;
}
```

[11] © Robert Tolksdorf, Berlin

Stackvariable

- Auf Stack Platz reservieren, dann relativ mit Offset zum SP adressieren, hinterher aufräumen

```
void proc(void) {
    long int a;
    short int b;
    a = ...;
    b = ...;
    ...
}
```
- ergibt compiliert:

```
proc:    subl $6, %esp
        movl ..., 0(%esp)
        movw ..., 4(%esp)
        ...
        addl $6, %esp
        ret
```
- geeignet für rekursive Unterprogramme

[12] © Robert Tolksdorf, Berlin

Stackframe

- Ein Unterprogramm braucht Speicherbereiche für
 - übergebene Parameter
 - Rücksprungadresse
 - Lokale Variablen
 - Parameter für eigene Aufrufe

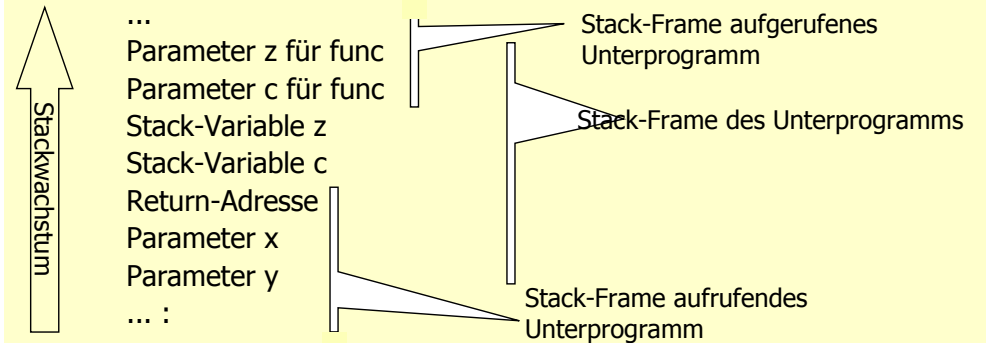
- Für aufgerufene Prozedur heißt dieser Bereich „Stackframe“

Stackframe

- Prozedur:

```
void proc(long int x, short int y) {  
    long int z;  
    char c;  
    ...  
    func(z, c);  
}
```

- Stackframes:



Veränderter Offset

- Aber:
Der Offset vom Stackpointer zu lokalen Variablen ändert sich:
 - z ist bei 0(%esp):
Stack-Variable z
Stack-Variable c
...
 - z ist bei 4(%esp):
Parameter c für func
Stack-Variable z
Stack-Variable c
...
 - z ist bei 8(%esp):
Parameter z für func
Parameter c für func
Stack-Variable z
Stack-Variable c
...

Veränderter Offset

- Prozedur

```
long int x;  
...  
proc(x, x);  
...  
ergibt compiliert:  
subl $4, %esp  
...  
pushl 0(%esp)  
pushl 4(%esp) /* rel. Adresse von x durch pushl geändert! */  
call proc  
addl $8, %esp  
...
```

Framepointer

- Spezielles Extra-Register (Frame-Pointer; i80x86: %ebp) enthält Adresse für Stack-Variablen- und Parameter-Zugriff:
- Beispiel ergibt kompiliert:

```
pushl %ebp          /* Platz für x reservieren */
movl %esp, %ebp
subl $4, %esp
...
pushl -4(%ebp)     /* proc(x,x) */
pushl -4(%ebp)     /* %ebp ändert sich durch pushl nicht */
call proc
addl $8, %esp
...
movl %ebp, %esp    /* Platz für x freigeben */
popl %ebp
ret
```

[Quellenangabe: Volkmar Sieh: Organisation und Technologie von Rechenystemen IV]