

DS-Systeme

Kapitel 11

**Funktionen
Teil 2**



Inhaltsverzeichnis

Thema	Seite
Lokale Daten auf dem Stack speichern	3
- call by reference	4
- Aufgabe	7
- Aufgabe	8
Non-Leaf Funktionen	10
- Aufgabe	14
Funktionen unter Verwendung von Framepointern	16
Rekursive Funktionen	22

Lokale Daten auf dem Stack speichern

Speicher auf dem Stack wird benötigt, wenn

- Funktionen aufgerufen werden:
 - aktuelle Parameter (ab dem 7. Parameter)
 - Rücksprungadresse
 - gesicherte Register
 - lokale Variablen
- es nicht genügend Register für lokale Daten in der aktuellen Funktion vorhanden sind
- der C/C++-Adressoperator (&) auf eine lokale Variable angewendet wird
- lokale Variablen Arrays oder Strukturen sind

Speicher auf dem Stack wird allokiert, indem sie der Stackpointer **rsp** um die entsprechende Anzahl Bytes vermindert wird.

Anschließend kann man entweder vom Stackpointer **rsp** oder vom Framepointer **rbp** aus auf die allokierten Speicherbereiche zugreifen.

Lokale Daten auf dem Stack speichern - call by reference

Beispiel:

C/C++ - Code:

```
void swap(long *xp, long *yp);

void caller()
{
    long arg1 = 534;
    long arg2 = 1057;
    swap(&arg1, &arg2);
}

void swap(long *xp, long *yp){
    long x = *xp;
    long y = *yp;
    *xp = y;
    *yp = x;
}
```

Vorgehensweise bei call by reference:

1. Caller legt die Variablen, deren Adressen benötigt werden, auf dem Stack ab.
2. Die Adressen der Variablen werden als Funktionsparameter in die üblichen Register kopiert (**rdi**, **rsi**, u.s.w.).
3. Funktion wird aufgerufen.
4. Caller gibt den Stack wieder frei.

Lokale Daten auf dem Stack speichern - call by reference

Beispiel:

```
swap:
    movq (%rdi), %rdx
    movq (%rsi), %rax
    movq %rax, (%rdi)
    movq %rdx, (%rsi)
    ret
```

```
# void caller()
caller:
    subq $16, %rsp
    movq $534, 8(%rsp)
    movq $1057, (%rsp)

    movq %rsp, %rsi
    leaq 8(%rsp), %rdi

    call swap

    addq $16, %rsp
    ret
```

arg1
arg2

1

2

3

4

Stack von **caller()** vor dem Aufruf von **swap()**

+24	<values passed to caller here no>	+31
+16	Return Address	+23
+8	arg1	+15
stack pointer -> +0	arg2	+7

Beim Anlegen von lokalen Variablen werden diese in der Reihenfolge ihres Auftretens im C-Code von den höheren nach den niedrigeren Adressen im Stack abgelegt.

Diese Vorgehensweise ist dann nötig, wenn die Variablen nicht in den Abschnitten **.data**, **.rodata** oder **.bss** abgelegt wurden und damit über keine Adressen verfügen, man in der Funktion aber auf ihre Adressen zugreifen möchte/muss.

Lokale Daten auf dem Stack speichern - call by referencekommentierter Assemblercode:

```
swap:
    movq (%rdi), %rdx
    movq (%rsi), %rax
    movq %rax, (%rdi)
    movq %rdx, (%rsi)
    ret

# void caller()
caller:
    subq $16, %rsp          # Allocate 16 bytes for stack frame

    movq $534, 8(%rsp)      # Store 534 in arg1
    movq $1057, (%rsp)      # Store 1057 in arg2

    movq %rsp, %rsi         # Set &arg2 as second argument
    leaq 8(%rsp), %rdi       # Compute &arg1 as first argument

    call swap               # Call swap(&arg1, &arg2)

    addq $16, %rsp          # Deallocate stack frame
    ret                     # Return
```

Lokale Daten auf dem Stack speichern

C/C++ - Code:

```
int myfn(int a, int b, int c,  
        int d, int e, int f,  
        int g){  
    return a + b + c + d + e + f + g;  
}  
  
int main() {  
    int a = 1;  
    int b = 2;  
    int c = 3;  
    int d = 4;  
    int e = 5;  
    int f = 6;  
    int g = 7;  
    int res = 0;  
    res = myfn(a, b, c, d, e, f, g);  
    printf("sum = %d\n", res);  
    return 0;  
}
```

Aufgabe:

Schreiben Sie zu dem Programm den entsprechenden Assembler-Code und halten sich dabei an die **Calling Conventions**.

Achten Sie dabei auf die korrekten Datentypen.

Lokales Speichern auf dem Stack - Aufgabe

Assembler - Code:

```
##function call_proc()
call_proc:
    subq $32, %rsp      # Allocate 32-byte
    movq $1, 24(%rsp)   # Store 1 in &x1
    movl $2, 20(%rsp)   # Store 2 in &x2
    movw $3, 18(%rsp)   # Store 3 in &x3
    movb $4, 17(%rsp)   # Store 4 in &x4
    leaq 17(%rsp), %rax  # Create &x4
    movq %rax, 8(%rsp)   # Store &x4 as arg8
    movb $4, (%rsp)     # Store 4 as arg7
    leaq 18(%rsp), %r9   # Pass &x3 as arg6
    movw $3, %r8d        # Pass 3 as arg5
    leaq 20(%rsp), %rcx  # Pass &x2 as arg4
    movl $2, %edx        # Pass 2 as arg3
    leaq 24(%rsp), %rsi  # Pass &x1 as arg2
    movq $1, %rdi        # Pass 1 as arg1
```

```
# function call_proc()
# continued
#...
# Deallocate stack frame
addq $32, %rsp
## Return / End of function
## call_proc()
ret
```

Die einzigen **Funktionsargu-
mente**, die auf dem Stack
abgelegt werden.

+24		+31
+16		+23
+8		+15
stack pointer -> +0		+7

Aufgabe:

Füllen Sie das Stacklayout
vor Ausführung der roten
Markierung aus.

Non-Leaf-Funktionen

- Leaf-Funktionen rufen keine andere Funktion auf.
- Non-Leaf-Funktionen rufen andere Funktionen auf.
- Non-Leaf-Funktionen sind etwas aufwendiger zu programmieren, da Sie die **Caller** Save-Regel (Sichern und Wiederherstellen von **Caller** Save-Registern) und die **Callee** Save-Regel (Sichern und Wiederherstellen von **Callee** Save-Registern) umsetzen müssen.

Non-Leaf-Funktionen

Beispiel:

C/C++ - Code:

```
long Q(long a);
```

Leaf-Funktion

```
long P(long x, long y)
```

Non-Leaf-Funktion

```
{
    long u = Q(y);
    long v = Q(x);
    return u + v;
}
```

callee save-Register werden auf dem Stack gesichert und vor Verlassen der Funktion wieder zurückgespeichert.

Assembler - Code:

```
# long P(long x, long y)
# x in rdi, y in rsi
P:
```

```
pushq %rbp
pushq %rbx
```

```
subq $8, %rsp
```

```
movq %rdi, %rbp
```

```
movq %rsi, %rdi
```

```
call Q
```

```
movq %rax, %rbx
```

```
movq %rbp, %rdi
```

```
call Q
```

```
addq %rbx, %rax
```

```
addq $8, %rsp
```

```
popq %rbx
```

```
popq %rbp
```

```
ret
```

Verwendung **der callee save**-Register.

Non-Leaf-FunktionenBeispiel:

C/C++ - Code:

```

long Q(long a);

long P(long x, long y)
{
    long u = Q(y);
    long v = Q(x);
    return u + v;
}

```

* Das Ausrichten (aligning) des Stackframes ist nur dann nötig, wenn die aufzurufende Funktion möglicherweise selbst Werte auf dem Stack ablegt oder den Stackpointer verschiebt.

Assembler - Code kommentiert:

```

# long P(long x, long y)
# x in rdi, y in rsi
P:
    pushq %rbp          # Save %rbp
    pushq %rbx          # Save %rbx
    subq $8, %rsp       # Align stack frame*
    movq %rdi, %rbp     # Save x
    movq %rsi, %rdi     # Move y to first argument
    call Q              # Call Q(y)
    movq %rax, %rbx     # Save result
    movq %rbp, %rdi     # Move x to first argument
    call Q              # Call Q(x)
    addq %rbx, %rax     # Add saved Q(y) to Q(x)
    addq $8, %rsp       # Deallocate last part of stack
    popq %rbx           # Restore %rbx
    popq %rbp           # Restore %rbp
    ret

```

Non-Leaf-Funktionen

Assembler - Code:

```

# long P(long x, long y)
# x in rdi, y in rsi
P:
    pushq %rbp
    pushq %rbx
    subq $8, %rsp
    movq %rdi, %rbp
    movq %rsi, %rdi
    call Q
    movq %rax, %rbx
    movq %rbp, %rdi
    call Q
    addq %rbx, %rax
    addq $8, %rsp
    popq %rbx
    popq %rbp
    ret

```

Before
first call
of Q
(Q(y))

stack pointer -> +0

+32		+39
+24	<ret. addr. caller of P(>	+31
+16	%rbp	+23
+8	%rbx	+15
+0		+7

After
pop

stack pointer -> +0

+32		+39
+24		+31
+16		+23
+8		+15
+0	<ret. addr. caller of P(>	+7

Non-Leaf-Funktionen

```

# function call_proc()
call_proc:
    subq $32, %rsp          # Allocate 32-byte
    movq $1, 24(%rsp)       # Store 1 in &x1
    movl $2, 20(%rsp)       # Store 2 in &x2
    movw $3, 18(%rsp)       # Store 3 in &x3
    movb $4, 17(%rsp)       # Store 4 in &x4
    leaq 17(%rsp), %rax      # Create &x4
    movq %rax, 8(%rsp)      # Store &x4 as arg8
    movl $4, (%rsp)         # Store 4 as arg7
    leaq 18(%rsp), %r9      # Pass &x3 as arg6
    movl $3, %r8d           # Pass 3 as arg5
    leaq 20(%rsp), %rcx     # Pass &x2 as arg4
    movl $2, %edx           # Pass 2 as arg3
    leaq 24(%rsp), %rsi     # Pass &x1 as arg2
    movl $1, %edi           # Pass 1 as arg1
    # Call another function: proc()
    call proc

    # Deallocate stack frame
    addq $32, %rsp
    # end of function call_proc()
    ret

```

Aufgabe:

Übersetzen Sie diesen Teil von **call_proc()** nach C.

```

movslq 20(%rsp), %rdx
addq 24(%rsp), %rdx
movswl 18(%rsp), %eax
movsbl 17(%rsp), %ecx
subl %ecx, %eax
cltq
imulq %rdx, %rax

```

Funktionen unter Verwendung von Framepointern

Framepointer (auch Base- bzw. Bottompointer **%ebp**) zeigten bei 32-Bit-Architekturen auf das untere Ende eines Rahmens (Frame), während der Stackpointer **%esp** auf das obere Ende zeigte.

- Früher wurden auf der x86-Architektur **immer** Framepointer verwendet.

- Vorteil:

Einfacherere Adressierung von Funktionsargumenten und lokalen Variablen.

- Nachteil:

Overhead:

Der Framepointer musste bei jedem **call** in einer Funktion gesichert und vor **ret** wieder hergestellt werden.

Funktionen unter Verwendung von Framepointern

Beispiel:

C/C++-Code:

```
void swap(long *xp, long *yp)
{
    long x = *xp;
    long y = *yp;
    *xp = y;
    *yp = x;
}

void caller()
{
    long arg1 = 534;
    long arg2 = 1057;
    swap(&arg1, &arg2);
}
```

Funktionen unter Verwendung von FramepointernAssembler-Code kommentiert:

```

swap:
    pushq %rbp                # Save old frame pointer
    movq %rsp, %rbp          # Set frame ptr to current stack ptr
    movq (%rdi), %rdx
    movq (%rsi), %rax
    movq %rax, (%rdi)
    movq %rdx, (%rsi)
    popq %rbp                # Restore old frame pointer
    ret

caller:
    pushq %rbp                # Save old frame pointer
    movq %rsp, %rbp          # Set frame ptr to current stack ptr
    subq $16, %rsp           # Allocate 16 Bytes on stack
    movq $534, -8(%rbp)
    movq $1057, -16(%rbp)
    leaq -16(%rbp), %rsi      # Compute &arg2 as second argument
    leaq -8(%rbp), %rdi       # Compute &arg1 as first argument
    call swap                 # Call swap(&arg1, &arg2)
    leave                     # Restores frame ptr and deallocs stack
    ret                       # Return

```

arg1
arg2

Funktionen unter Verwendung von Framepointern

Assembler-Code kommentiert:

```

swap:
    pushq %rbp
    movq %rsp, %rbp
    movq (%rdi), %rdx
    movq (%rsi), %rax
    movq %rax, (%rdi)
    movq %rdx, (%rsi)
    popq %rbp
    ret

caller:
    pushq %rbp
    movq %rsp, %rbp
    subq $16, %rsp
    movq $534, -8(%rbp)
    movq $1057, -16(%rbp)
    leaq -16(%rbp), %rsi
    leaq -8(%rbp), %rdi
    call swap
    leave
    ret
    
```

arg1
arg2

```

# Save old frame pointer
# Set frame ptr to current stack ptr
# Allocate 16 Bytes on stack

# Compute &arg2 as second argument
# Compute &arg1 as first argument
# Call swap(&arg1, &arg2)
# Restores frame ptr and deallocs stack
# Return
    
```

Stack von **caller()** vor dem Aufruf von **swap()** mit Framepointer als Referenz.

		<values passed to caller here no>	
	+8	Return Address	+15
frame pointer ->	+0	Saved %rpb	
	-8	arg1	-15
stack pointer ->	-16	arg2	-23

Funktionen unter Verwendung von Framepointern

ohne Framepointer

```

swap:
    movq (%rdi), %rdx
    movq (%rsi), %rax
    movq %rax, (%rdi)
    movq %rdx, (%rsi)
    ret

# void caller()
caller:
    subq $16, %rsp
    movq $534, 8(%rsp)
    movq $1057, (%rsp)

    movq %rsp, %rsi
    leaq 8(%rsp), %rdi

    call swap

    addq $16, %rsp
    ret

```

Anmerkung:

Im Vergleich zur reinen Verwendung des Stackpointers:

- Offsets zu Übergabeparametern bleiben positiv, sind aber um 8 größer (da sich der alte Framepointer auch auf dem Stack befindet). (Dies ist interessant, falls der Callee mehr als sechs Argumente erhält.)
- Offsets zu lokalen Variablen sind nun negativ.

mit Framepointer

```

swap:
    pushq %rbp
    movq %rsp, %rbp
    movq (%rdi), %rdx
    movq (%rsi), %rax
    movq %rax, (%rdi)
    movq %rdx, (%rsi)
    popq %rbp
    ret

caller:
    pushq %rbp
    movq %rsp, %rbp
    subq $16, %rsp
    movq $534, -8(%rbp)
    movq $1057, -16(%rbp)
    leaq -16(%rbp), %rsi
    leaq -8(%rbp), %rdi
    call swap
    leave
    ret

```

Funktionen unter Verwendung von Framepointern

Die Befehle **enter** und **leave** werden verwendet, um das Setzen und Wiederherstellen des Framepointers zu vereinfachen.

- **function prologue** (function intro):

```
pushq %rbp
movq %rsp, %rbp
```

or

```
enter $0, $0
# op1 mostly zero (nesting level)
# op2 how many bytes to allocate on stack;
# -> an op2!= 0 would cause another code line:
# subq $<op2>, %rsp
# however enter is rarely used
```

- **function epilogue** (function exit):

```
movq %rbp, %rsp
popq %rbp
```

or

```
leave
```

Rekursive FunktionenBeispiel:

C/C++ - Code:

```

long rfact(long n)
{
    if (n <= 1)
        return 1;
    return n * rfact(n - 1);
}

```

Beachten Sie, dass jeder Aufruf einer rekursiven Funktion einen eigenen Stackframe erhält.

Was **nicht** mehrfach zur Verfügung steht, sind die Register.

Assembler - Code:

```

# long rfact(long n)
# n in rdi

rfact:
    pushq %rbx
    movq %rdi, %rbx
    movq $1, %rax
    cmpq $1, %rdi
    jle .L35
    leaq -1(%rdi), %rdi
    call rfact
    imulq %rbx, %rax

.L35:
    popq %rbx
    ret

```

Nach dem Aufruf der rekursiven Funktion ist der Stack des Callers wieder im ursprünglichen Zustand. Deshalb kann man ohne irgendeine Rechnung wieder auf die zuvor gespeicherten Elemente (hier **rbx**) zugreifen.

Rekursive Funktionen

C/C++ - Code:

```

long rfact(long n)
{
    if (n <= 1)
        return 1;
    return n * rfact(n - 1);
}

```

Assembler - Code kommentiert:

```

.globl rfact
.type rfact, @function
# long rfact(long n)
# n in rdi
rfact:
    pushq %rbx                # Save rbx
    movq %rdi, %rbx          # Store n in callee-saved register
    movq $1, %rax             # Set return value = 1
    cmpq $1, %rdi             # Compare n:1
    jle .L35                  # If <=, goto done
    leaq -1(%rdi), %rdi       # Compute n-1
    call rfact                 # Call rfact(n-1)
    imulq %rbx, %rax           # Multiply result by n
.L35:                         # done:
    popq %rbx                 # Restore rbx
    ret                       # Return

```

Rekursive Funktionen

Assembler - Code:

```
# long rfact(long n)
# n in rdi

rfact:
    pushq %rbx
    movq %rdi, %rbx
    movq $1, %rax
    cmpq $1, %rdi
    jle .L35
    leaq -1(%rdi), %rdi

    call rfact

    imulq %rbx, %rax
.L35:
    popq %rbx
    ret
```

