Der Stack

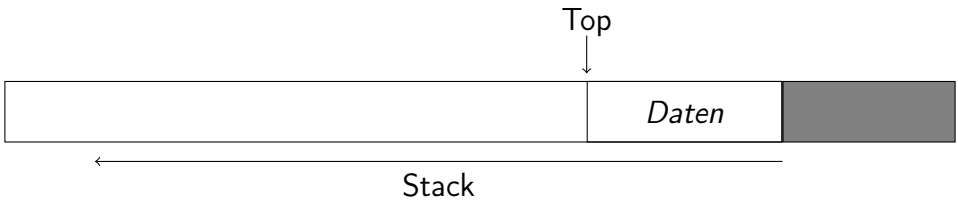
- Speicherbereich von Programmen
- Besonderer Verwendungszweck
- und besonderes Zugriffsmuster

Verwendung des Stacks

- Auch “Call-Stack” (dt. “Aufrufstapel”) genannt
- Informationen bezügl. Funktionen und Funktionsaufrufen
- Unter anderem Funktionsvariablen
- und Rücksprungadressen
- Teils aber auch Funktionsparameter
- und Rückgabewerte

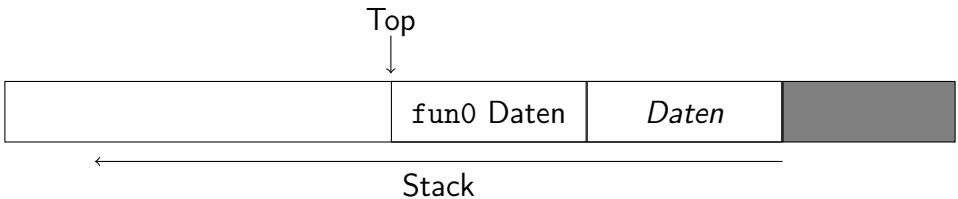
Zugriff auf den Stack

"Last In - First Out" Prinzip



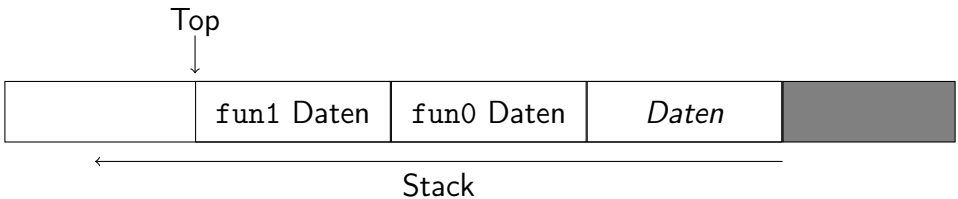
Zugriff auf den Stack

"Last In - First Out" Prinzip

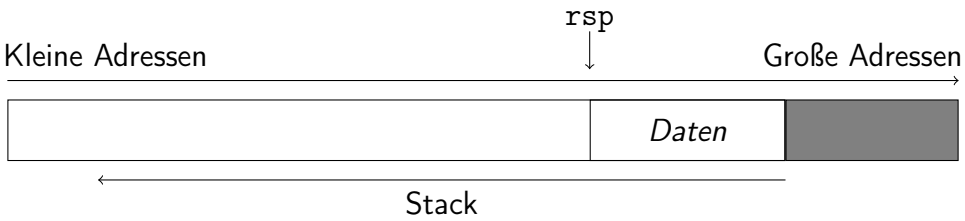


Zugriff auf den Stack

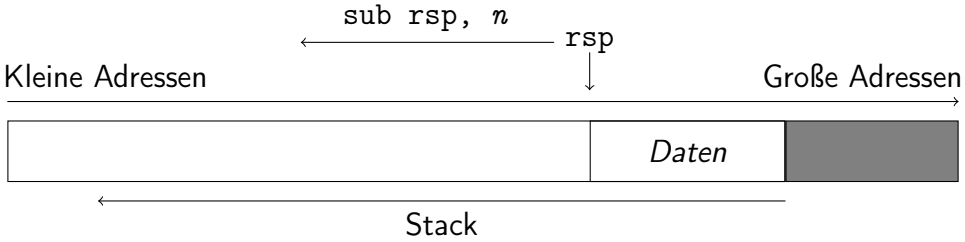
"Last In - First Out" Prinzip



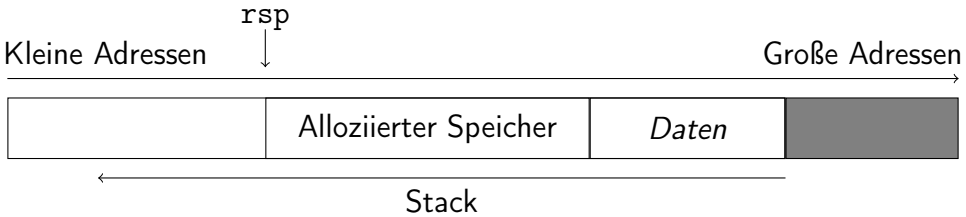
Manipulation des Stacks auf x86-64 Systemen



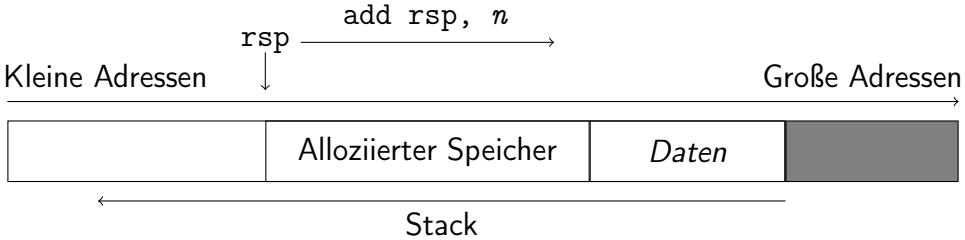
Manipulation des Stacks auf x86-64 Systemen



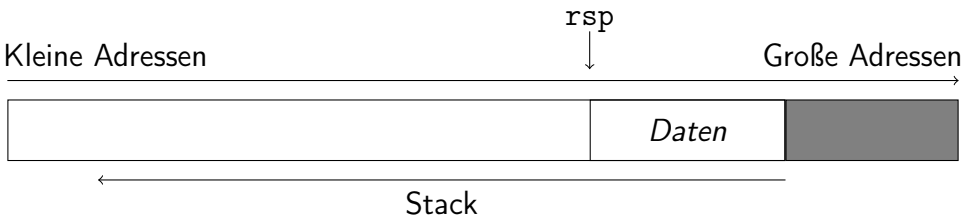
Manipulation des Stacks auf x86-64 Systemen



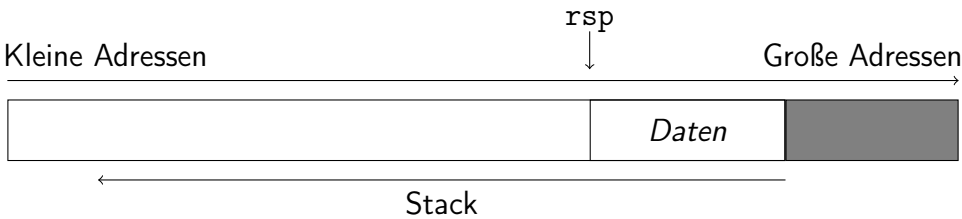
Manipulation des Stacks auf x86-64 Systemen



Manipulation des Stacks auf x86-64 Systemen



Manipulation des Stacks auf x86-64 Systemen



push/pop *reg64* / *mem64*

- `push` *op* $\longrightarrow$ `sub rsp, 8; [rsp]` $\leftarrow$ *op*
- `pop` *op* $\longrightarrow$ *op* $\leftarrow$ `[rsp]; add rsp, 8`

Quiz: Der Stack

Wie können nach der folgenden Instruktionssequenz die Werte auf dem Stack wieder in ihre ursprünglichen Register geladen werden?

```
push rax
push rcx
```

☐
```
mov rax, [rsp + 8]
mov rcx, [rsp]
```

☐
```
pop rcx
pop rax
```

☐
```
pop rax
pop rcx
```

☐
```
mov rax, [rsp + 16]
mov rcx, [rsp + 8]
```

Funktionsaufrufe und der Stack

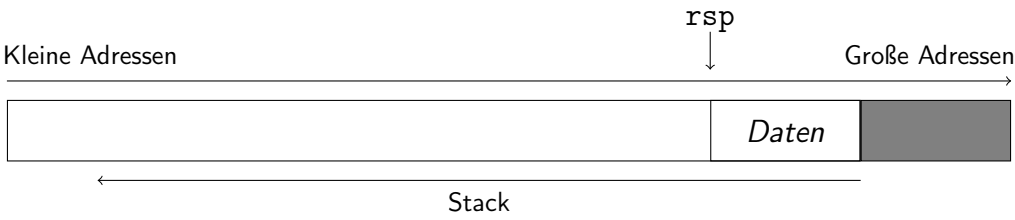
- Rücksprungadressen stehen auf dem Stack
- Impliziter Speicherzugriff durch Funktionsaufruf

```
call label/reg64
ret
```

- `call` *op* $\longrightarrow$ “`push rip`”; `jmp` *op*
- `ret` $\longrightarrow$ “`pop rip`”

Funktionsaufrufe und der Stack: Beispiel

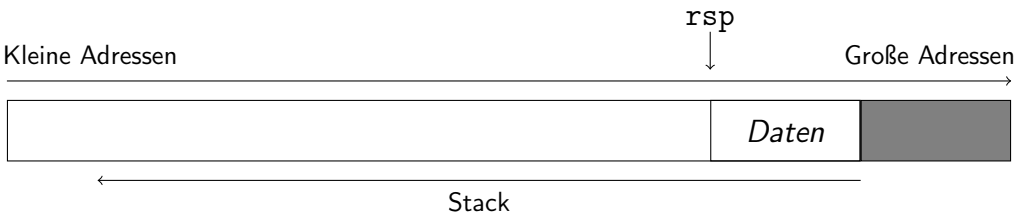
```
1    ...
2    call fun      ←
3    cmp rax, 42
4    ...
5
6  fun:
7    sub rsp, 24
8    ...
9    add rsp, 24
10   ret
```



Funktionsaufrufe und der Stack: Beispiel

```
1    ...
2    call fun
3    cmp rax, 42
4    ...
5
6  fun:
7    sub rsp, 24
8    ...
9    add rsp, 24
10   ret
```

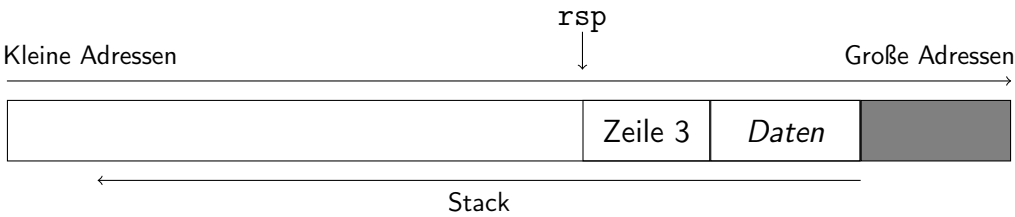
⟵ rip ≙ “Zeile 3”



Funktionsaufrufe und der Stack: Beispiel

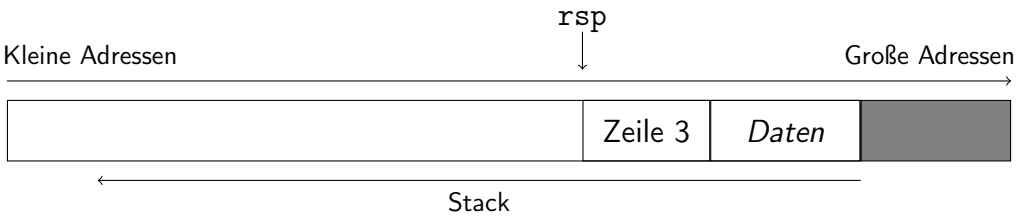
```
1    ...
2    call fun
3    cmp rax, 42
4    ...
5
6  fun:
7    sub rsp, 24
8    ...
9    add rsp, 24
10   ret
```

⟵ `rip` $\hat{=}$ “Zeile 3”



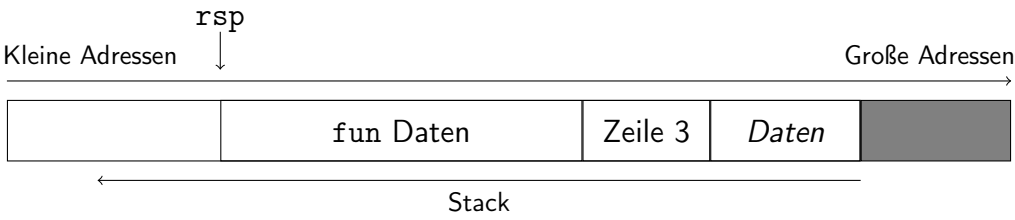
Funktionsaufrufe und der Stack: Beispiel

```
1    ...
2    call fun
3    cmp rax, 42
4    ...
5
6  fun:
7    sub rsp, 24   ←
8    ...
9    add rsp, 24
10   ret
```



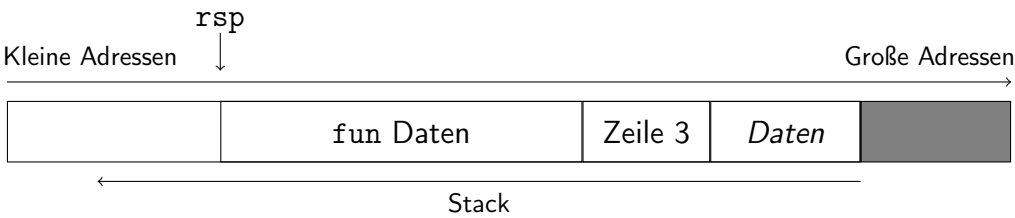
Funktionsaufrufe und der Stack: Beispiel

```
1    ...
2    call fun
3    cmp rax, 42
4    ...
5
6  fun:
7    sub rsp, 24   ←
8    ...
9    add rsp, 24
10   ret
```



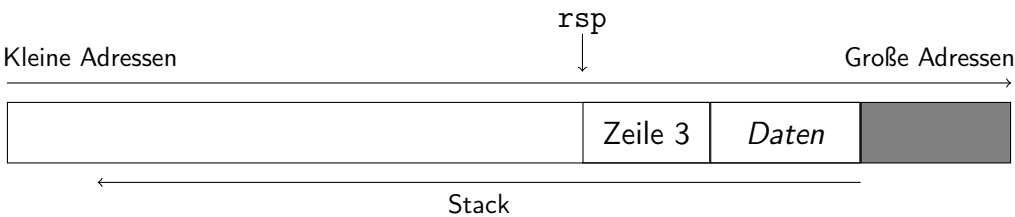
Funktionsaufrufe und der Stack: Beispiel

```
1    ...
2    call fun
3    cmp rax, 42
4    ...
5
6  fun:
7    sub rsp, 24
8    ...
9    add rsp, 24   ←
10   ret
```



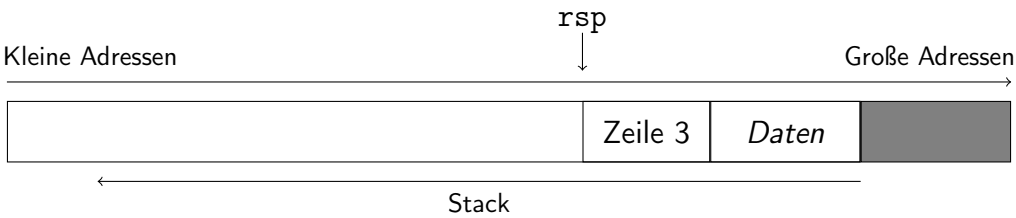
Funktionsaufrufe und der Stack: Beispiel

```
1    ...
2    call fun
3    cmp rax, 42
4    ...
5
6  fun:
7    sub rsp, 24
8    ...
9    add rsp, 24   ←
10   ret
```



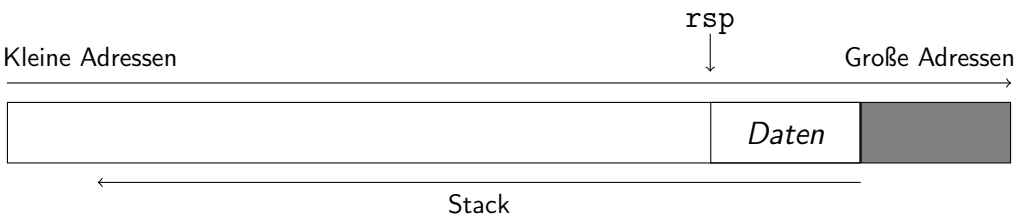
Funktionsaufrufe und der Stack: Beispiel

```
1    ...
2    call fun
3    cmp rax, 42
4    ...
5
6  fun:
7    sub rsp, 24
8    ...
9    add rsp, 24
10   ret          ←
```



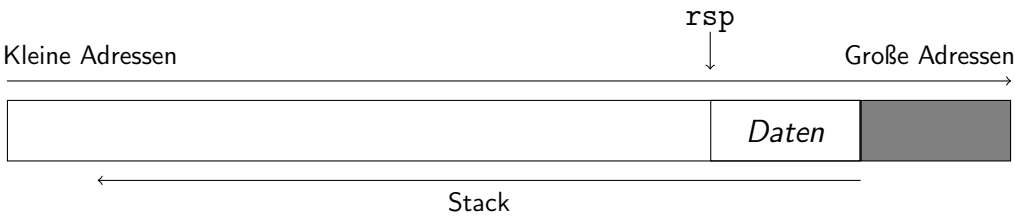
Funktionsaufrufe und der Stack: Beispiel

```
1    ...
2    call fun
3    cmp rax, 42
4    ...
5
6  fun:
7    sub rsp, 24
8    ...
9    add rsp, 24
10   ret          ←
```



Funktionsaufrufe und der Stack: Beispiel

```
1    ...
2    call fun
3    cmp rax, 42   ←
4    ...
5
6  fun:
7    sub rsp, 24
8    ...
9    add rsp, 24
10   ret
```



Quiz: Funktionsaufrufe und der Stack (1)

Welche Probleme können bei rekursiven Funktionsaufrufen via `call` bezüglich der Größe des Stacks auftreten?

- ☐ Ab einer gewissen Rekursionstiefe kann nicht mehr genügend Stack-Speicher alloziiert werden.
- ☐ Ab einer gewissen Rekursionstiefe wird der Stack-Speicher wieder von unten nach oben überschrieben.
- ☐ Es können keine Probleme auftreten. Der Stack wird immer dynamisch erweitert.

Quiz: Funktionsaufrufe und der Stack (2)

Wie können Stack Overflows, die durch rekursive Funktionsaufrufe via `call` entstehen, vermieden werden?

- ☐ Durch die Verwendung eines iterativen statt eines rekursiven Algorithmus.
- ☐ Indem man in Funktionen nur Speicher auf dem Heap alloziiert.
- ☐ In dem Fall dass die letzte Instruktion vor einem `ret` ein `call` ist, indem man `call` durch `jmp` ersetzt.
- ☐ Wir können diese Probleme nicht vermeiden.