Übung zu Betriebssysteme

Threadumschaltung

6. & 8. Dezember 2017

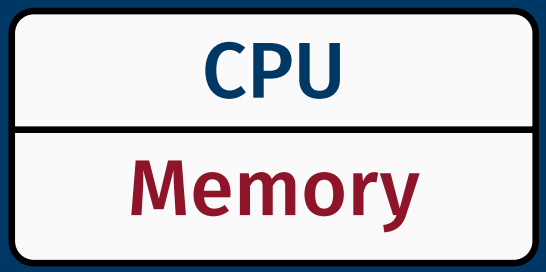
Andreas Ziegler
Bernhard Heinloth

Lehrstuhl für Informatik 4
Friedrich-Alexander-Universität Erlangen-Nürnberg

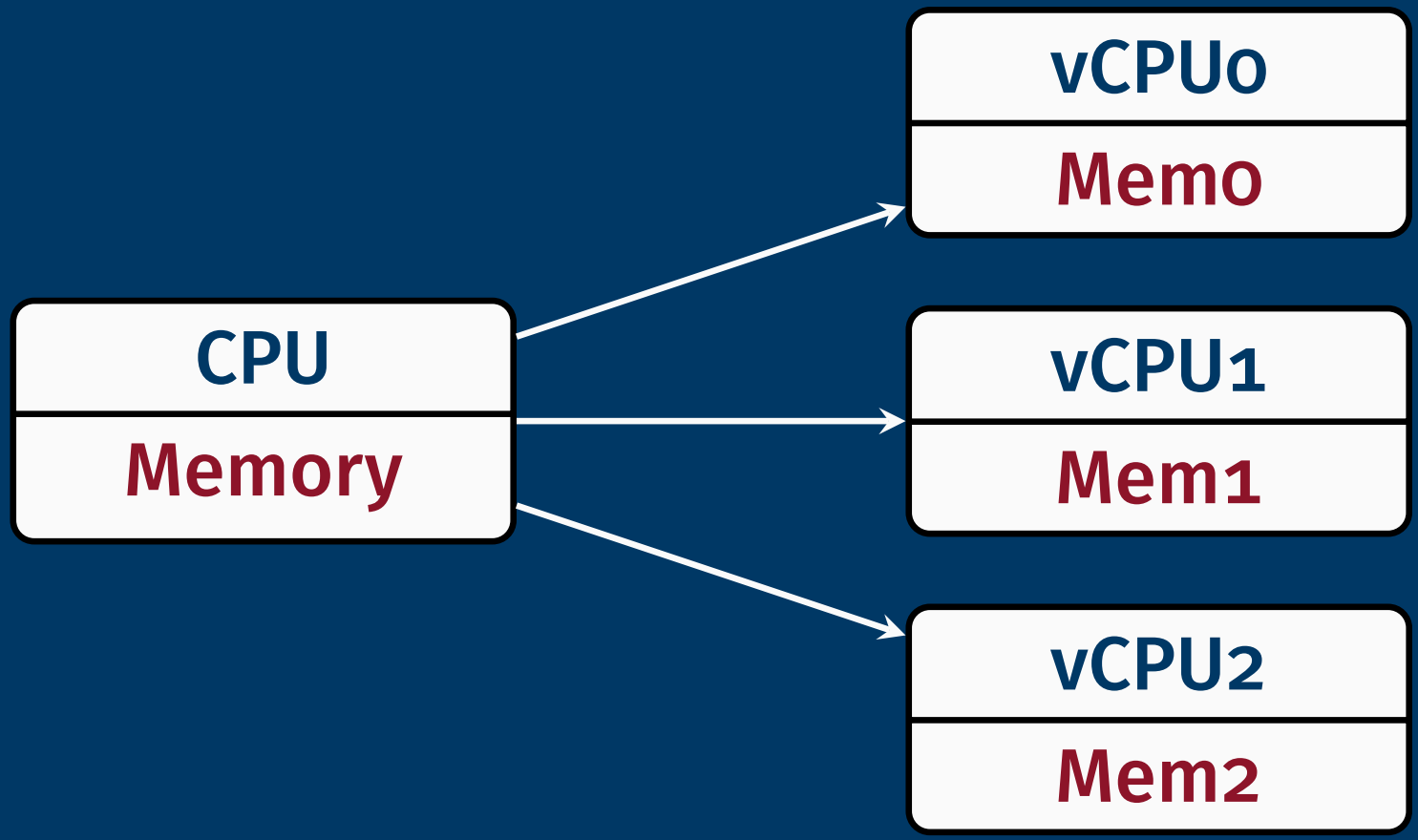
Lehrstuhl für Verteilte Systeme
und Betriebssysteme

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT



tatsächliche Hardware



tatsächliche Hardware → illusionierte Hardware

Speicher virtualisieren

CPU virtualiseren

Speicher virtualisieren ist einfach:

```
#define MEMSIZE 100
char Mem0[MEMSIZE];
char Mem1[MEMSIZE];
char Mem2[MEMSIZE];
```

CPU virtualiseren

Speicher virtualisieren ist einfach:

```
#define MEMSIZE 100
char Mem0[MEMSIZE];
char Mem1[MEMSIZE];
char Mem2[MEMSIZE];
```

CPU virtualiseren

- nicht teilbar im Ort

Speicher virtualisieren ist einfach:

```
#define MEMSIZE 100
char Mem0[MEMSIZE];
char Mem1[MEMSIZE];
char Mem2[MEMSIZE];
```

CPU virtualiseren

- nicht teilbar im Ort
- aber teilbar in der Zeit

Koroutinen

Motivation: mehr Aktivitätsträger als CPUs

```
void foo(){
  int f = 42;

  while (f--){
    kout << "foo"
         << f
         << endl;

  }

}
```

```
void bar(){
  int b = 23;

  while (b--){
    kout << "bar"
         << b
         << endl;

  }

}
```

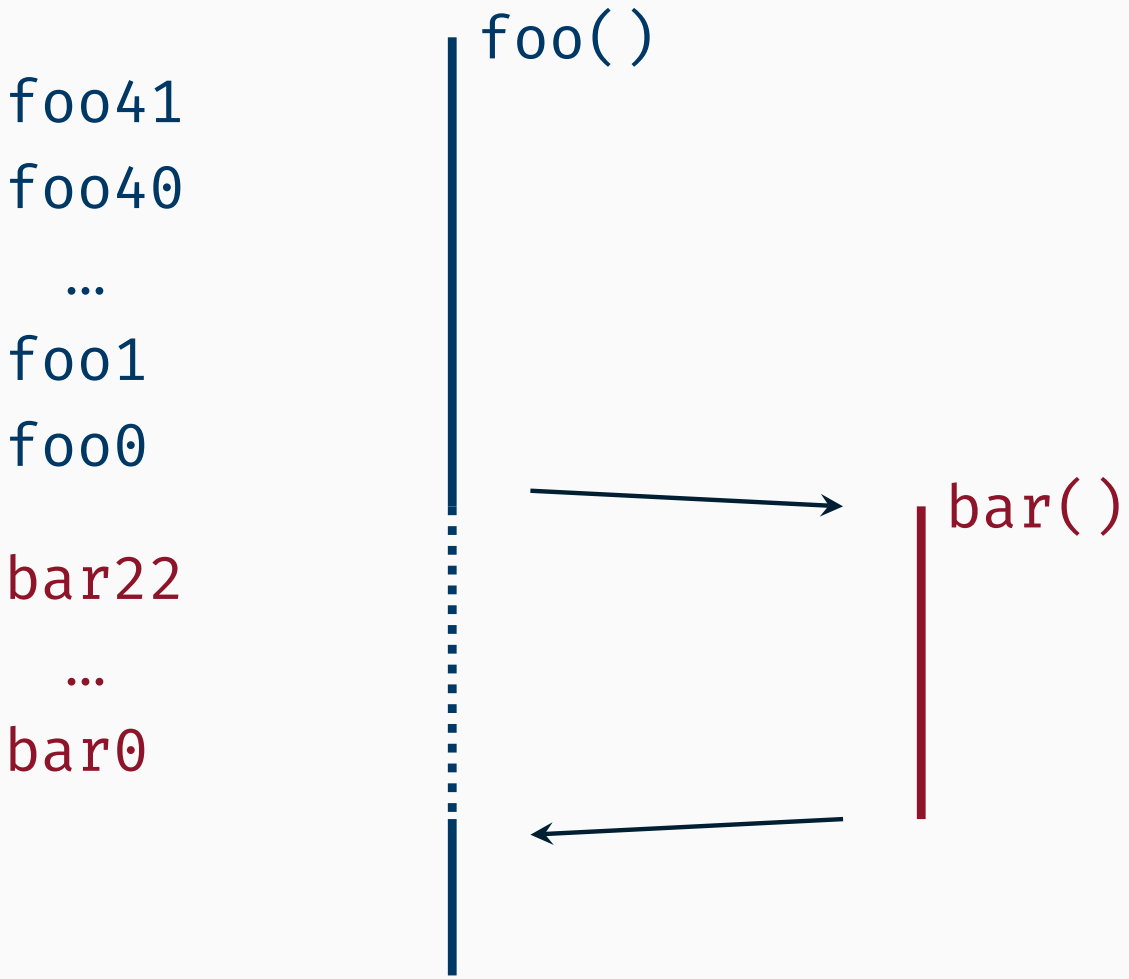
Einseitiger Aufruf

Einseitiger Aufruf

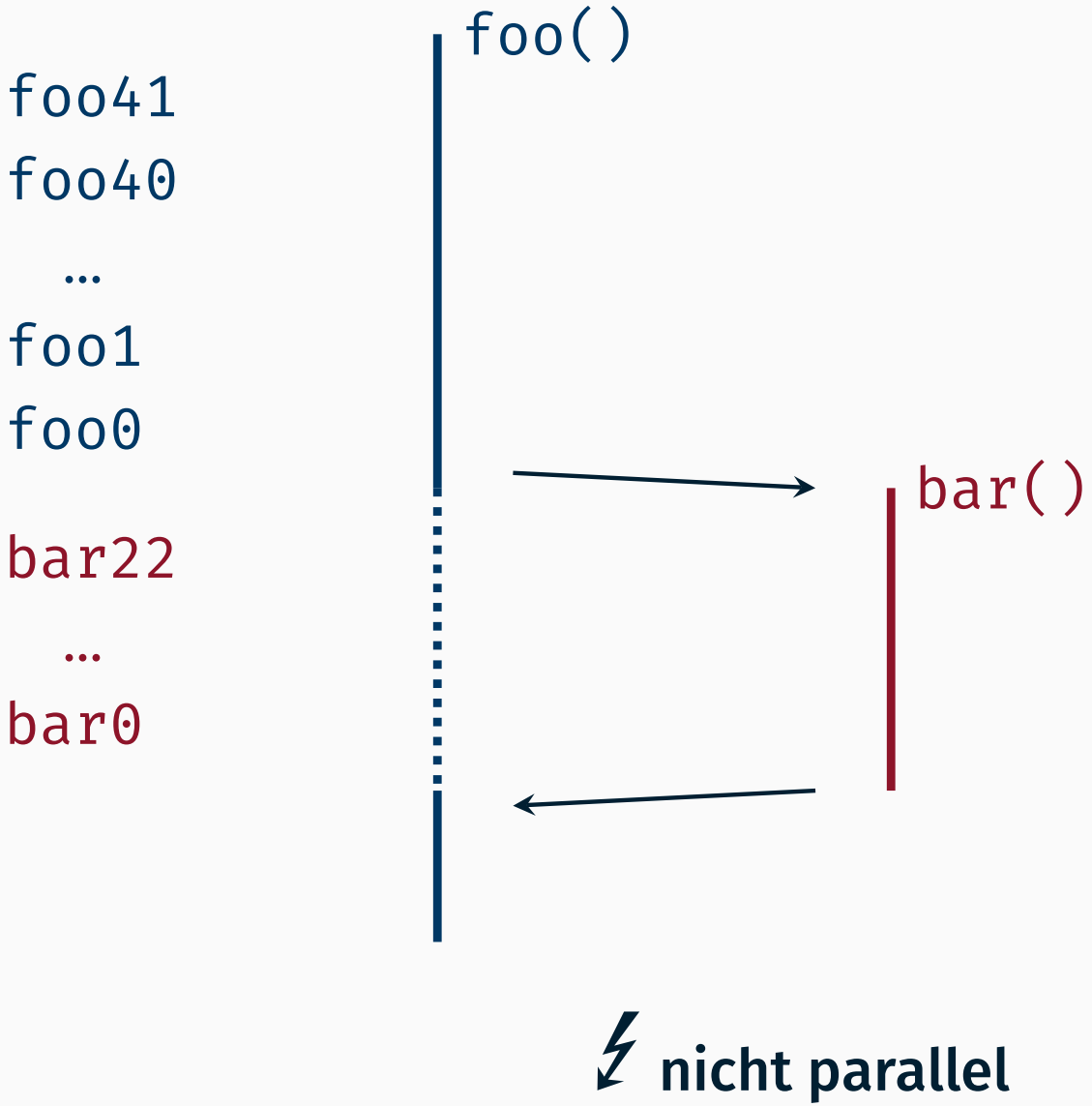
```
void foo(){
  int f = 42;

  while (f--){
    kout << "foo"
         << f
         << endl;

  }
  bar();
}
```

```
void bar(){
  int b = 23;

  while (b--){
    kout << "bar"
         << b
         << endl;

  }

}
```

Einseitiger Aufruf



Einseitiger Aufruf



nicht parallel

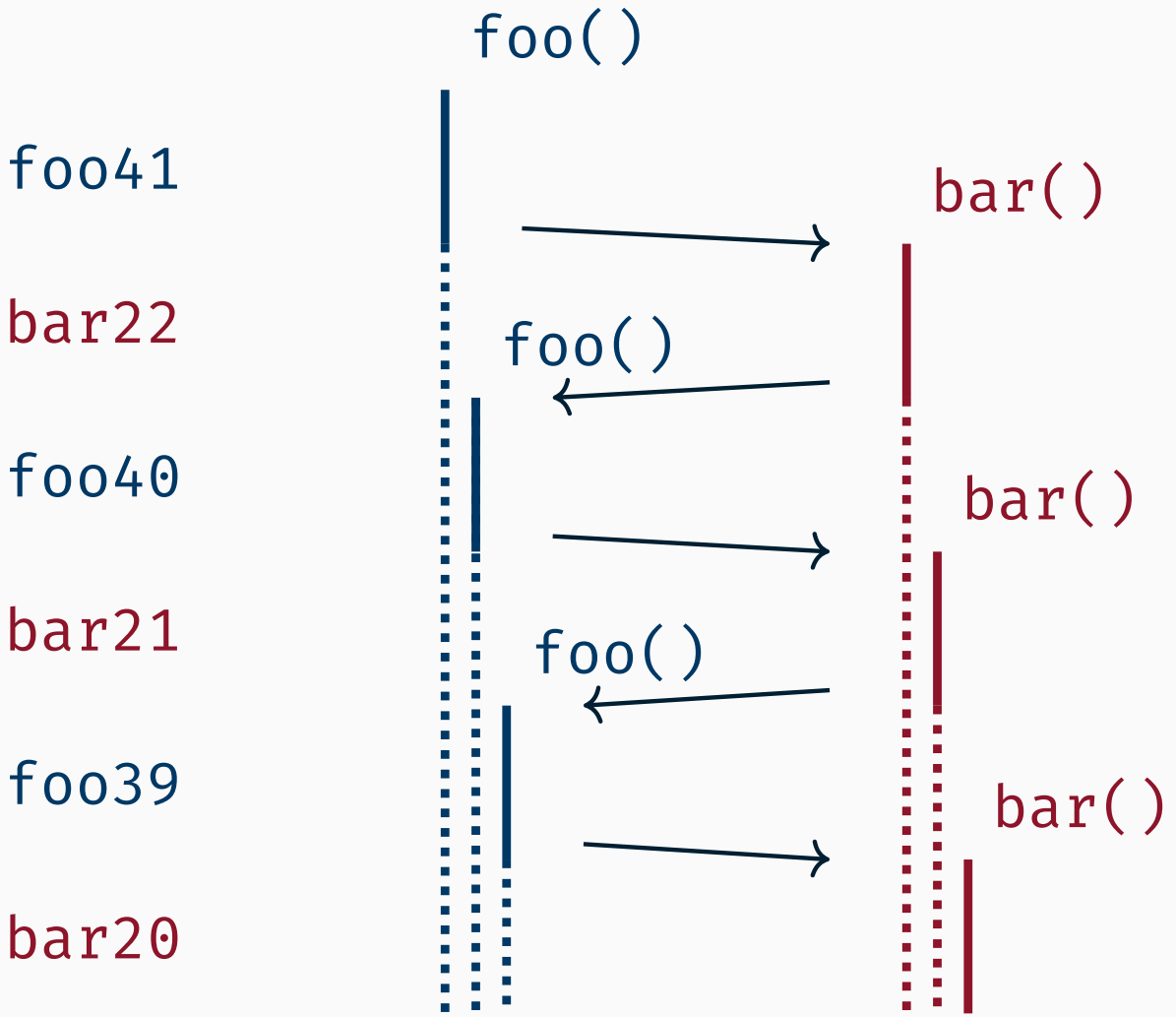
Gegenseitiger Aufruf

Gegenseitiger Aufruf

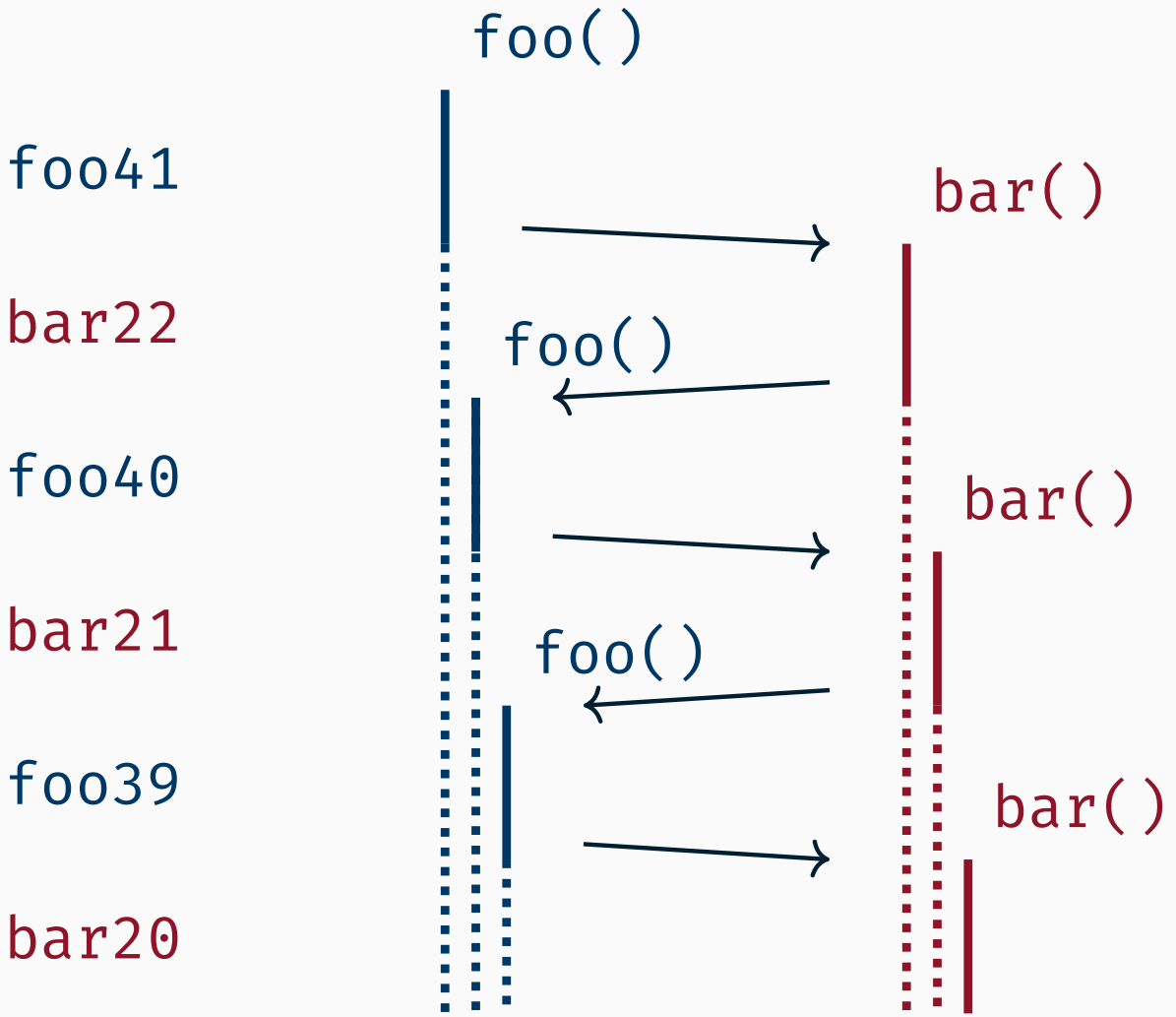
```
void foo(){
  static int f = 42;

  while (f--){
    kout << "foo"
         << f
         << endl;
    bar();
  }

}
```

```
void bar(){
  static int b = 23;

  while (b--){
    kout << "bar"
         << b
         << endl;
    foo();
  }

}
```

Gegenseitiger Aufruf

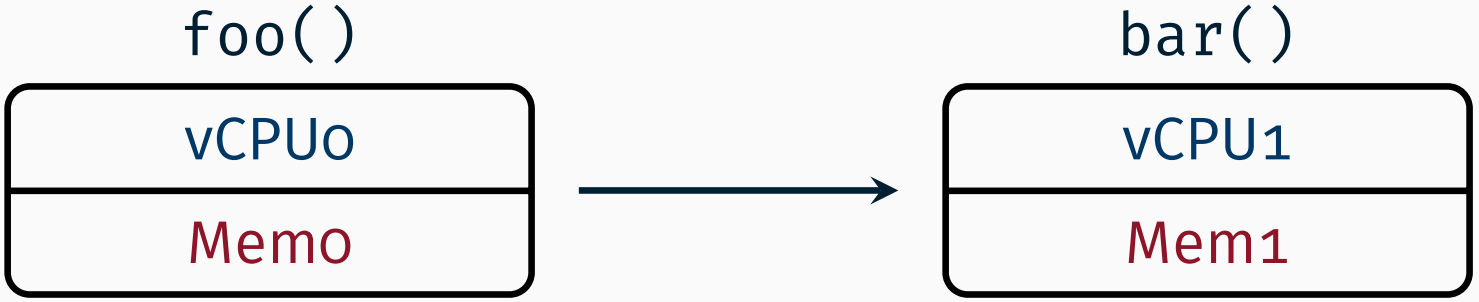


Gegenseitiger Aufruf

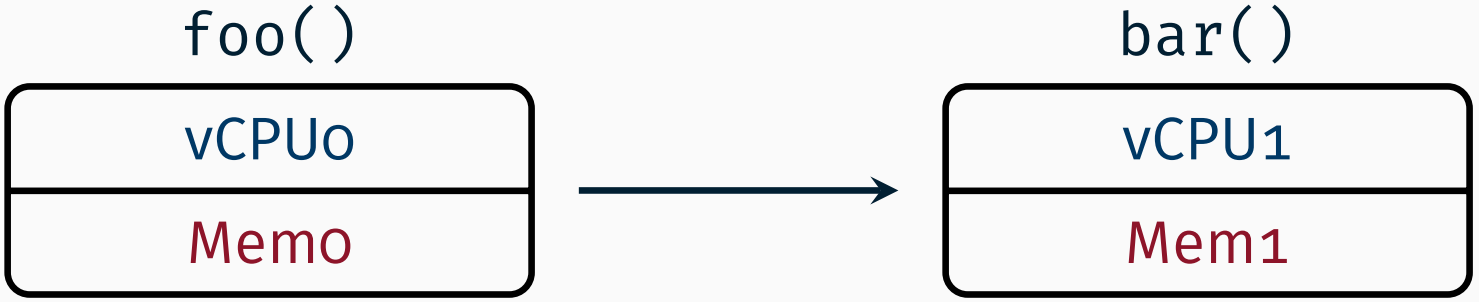


⚡ **hoher Speicherverbrauch & Endlosrekursion**

Umschaltung

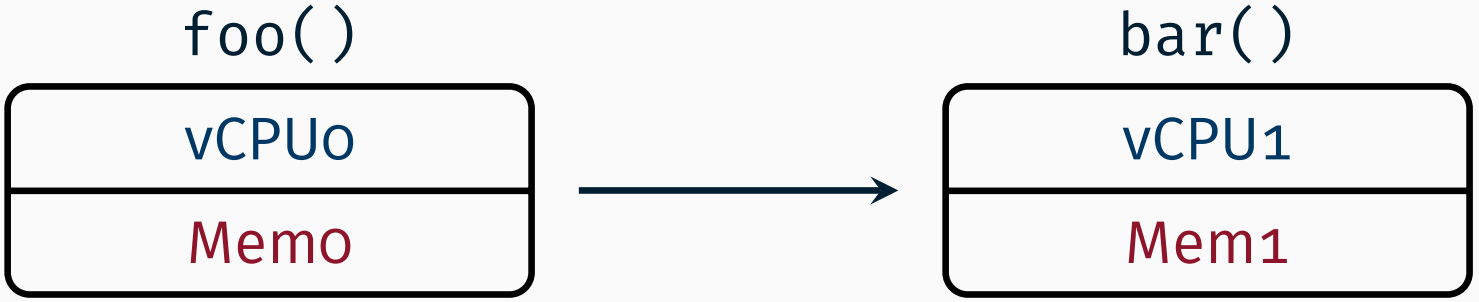


Umschaltung



Kontrollflusszustand sichern & laden

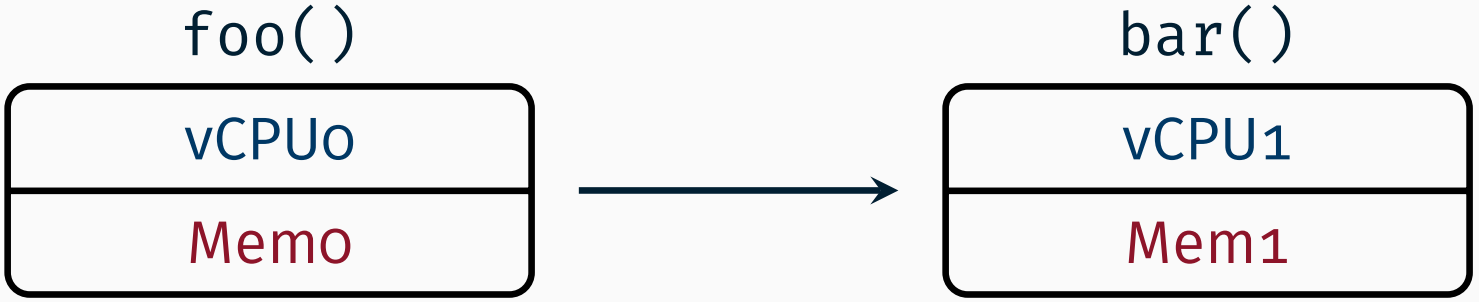
Umschaltung



Kontrollflusszustand sichern & laden

- Stack
- Register

Umschaltung

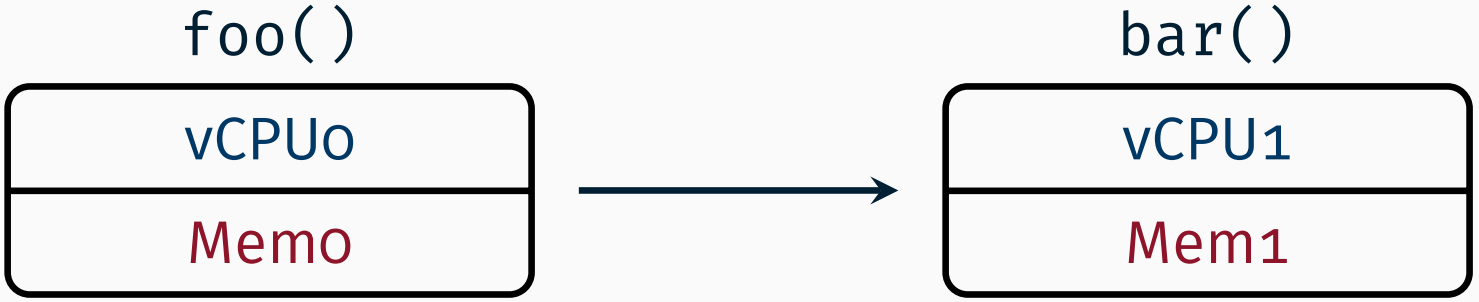


Kontrollflusszustand sichern & laden

- Stack
- Register

} `toc` (Thread of control)

Umschaltung



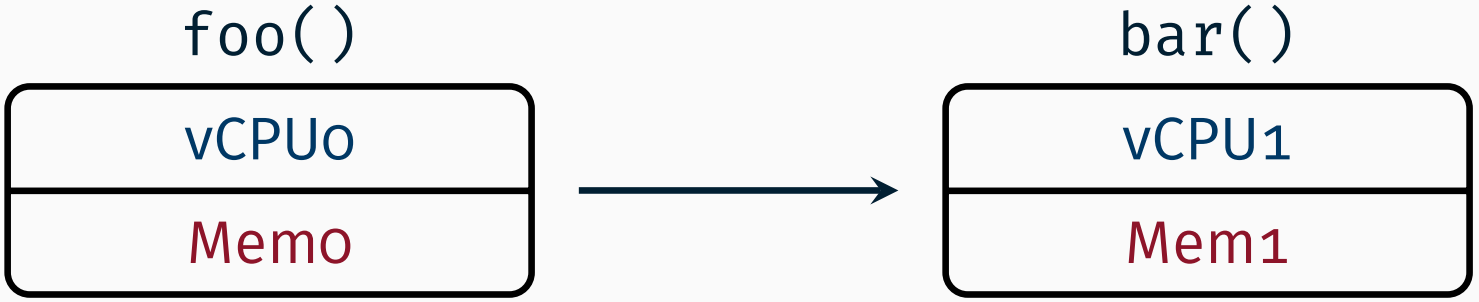
Kontrollflusszustand sichern & laden

- Stack
- Register

} `toc` (Thread of control)

- Umschaltefunktion

Umschaltung



Kontrollflusszustand sichern & laden

- Stack
- Register

} `toc` (Thread of control)

- Umschaltefunktion

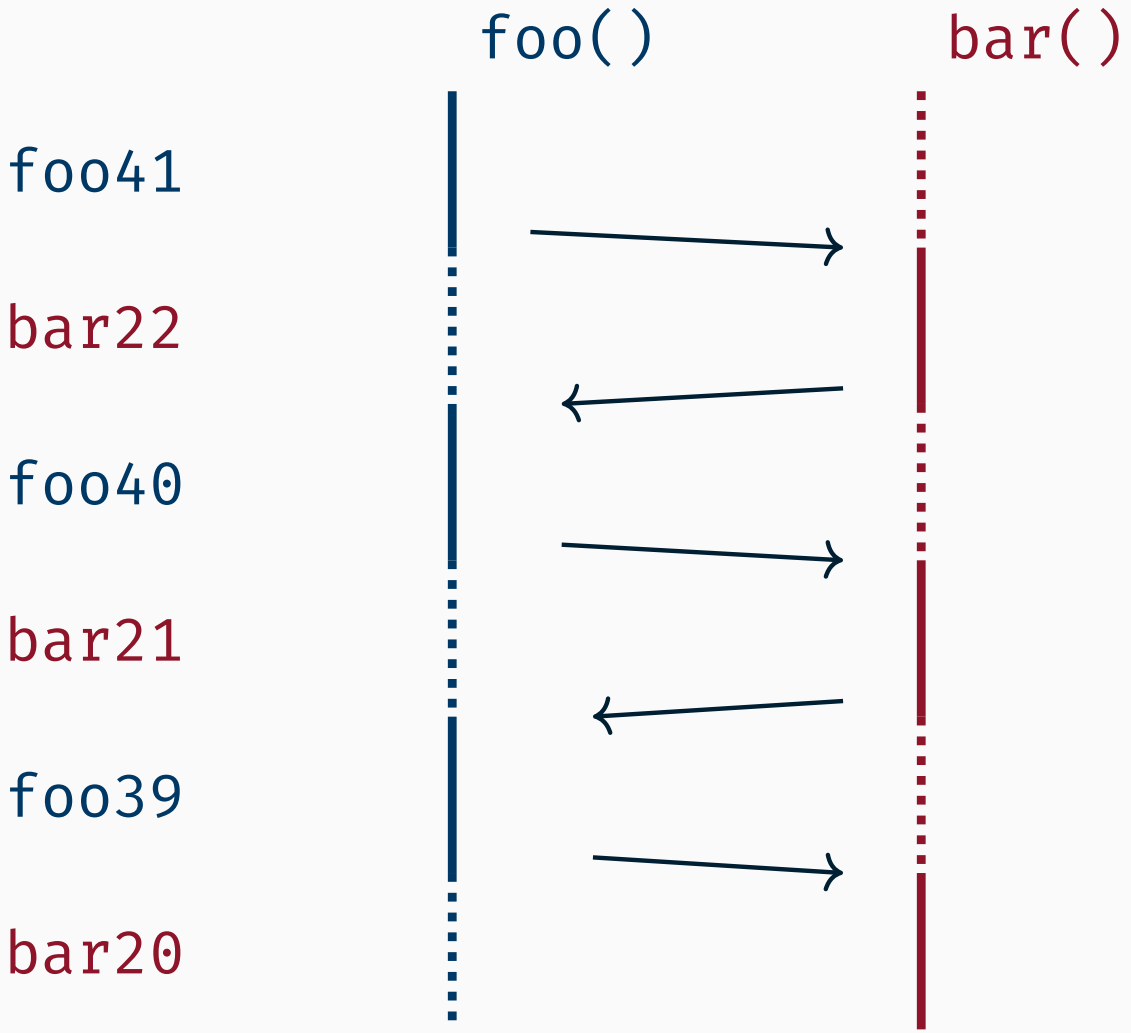
```
toc_switch(toc *now, toc *then);
```

Umschaltung

```
void foo(){
  int f = 42;

  while (f--){
    kout << "foo"
         << f
         << endl;
    toc_switch(
        &toc_foo,
        &toc_bar
      );
  }

}
```

```
void bar(){
  int b = 23;

  while (b--){
    kout << "bar"
         << b
         << endl;
    toc_switch(
        &toc_bar,
        &toc_foo
      );
  }

}
```

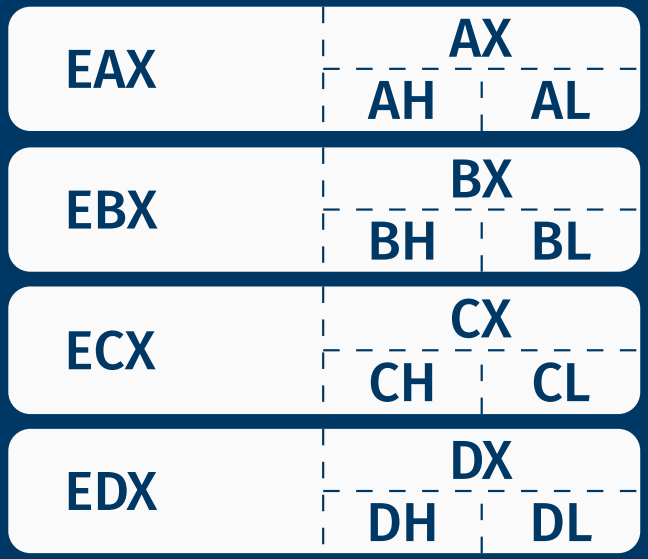
Umschaltung

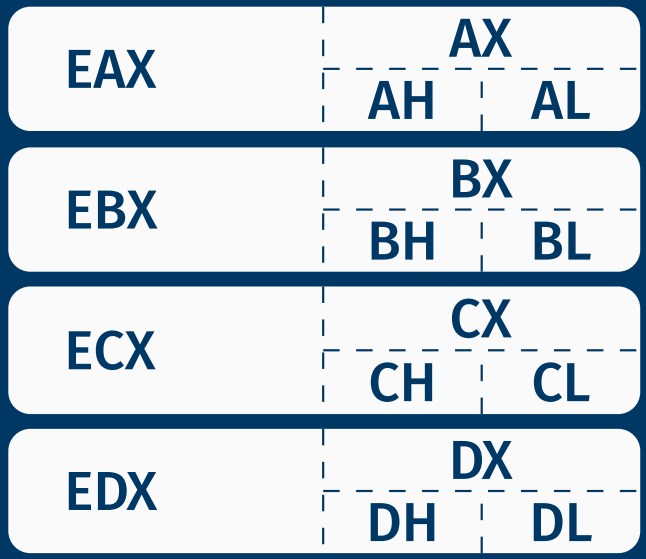


Kontextsicherung und -wechsel

```
void foo(){
  ...

  // flüchtige
  // Register
  // sichern

  bar();

  // flüchtige
  // Register
  // wiederherstellen

  ...
}
```

```
void bar(){

  // nicht-flüchtige
  // Register
  // sichern

  ...

  // nicht-flüchtige
  // Register
  // wiederherstellen

  return;

}
```


EAX
AX
AH
AL
EBX
BX
BH
BL
ECX
CX
CH
CL
EDX
DX
DH
DL



ursprünglich Akkumulatorregister

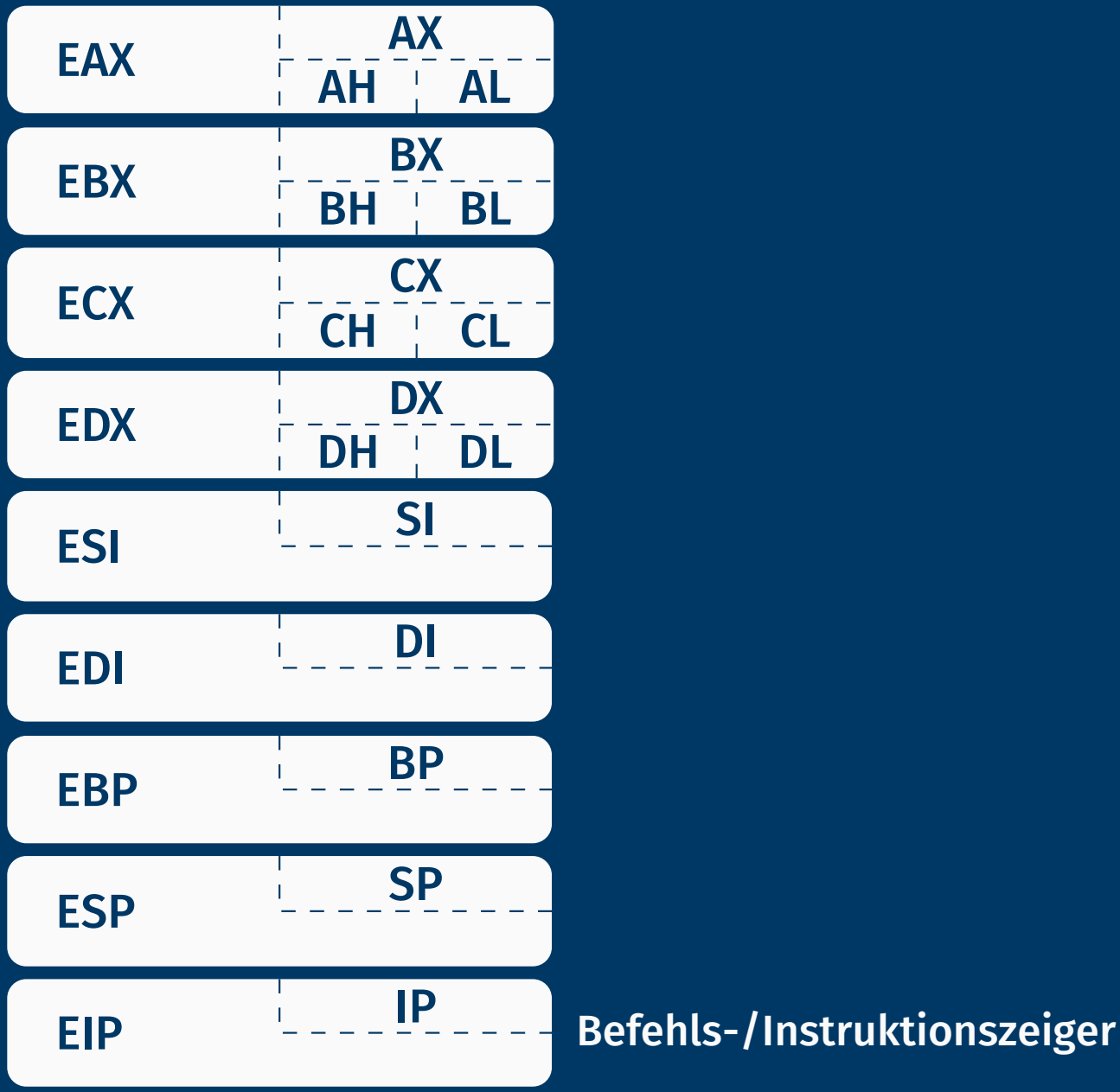
ursprünglich Basisregister

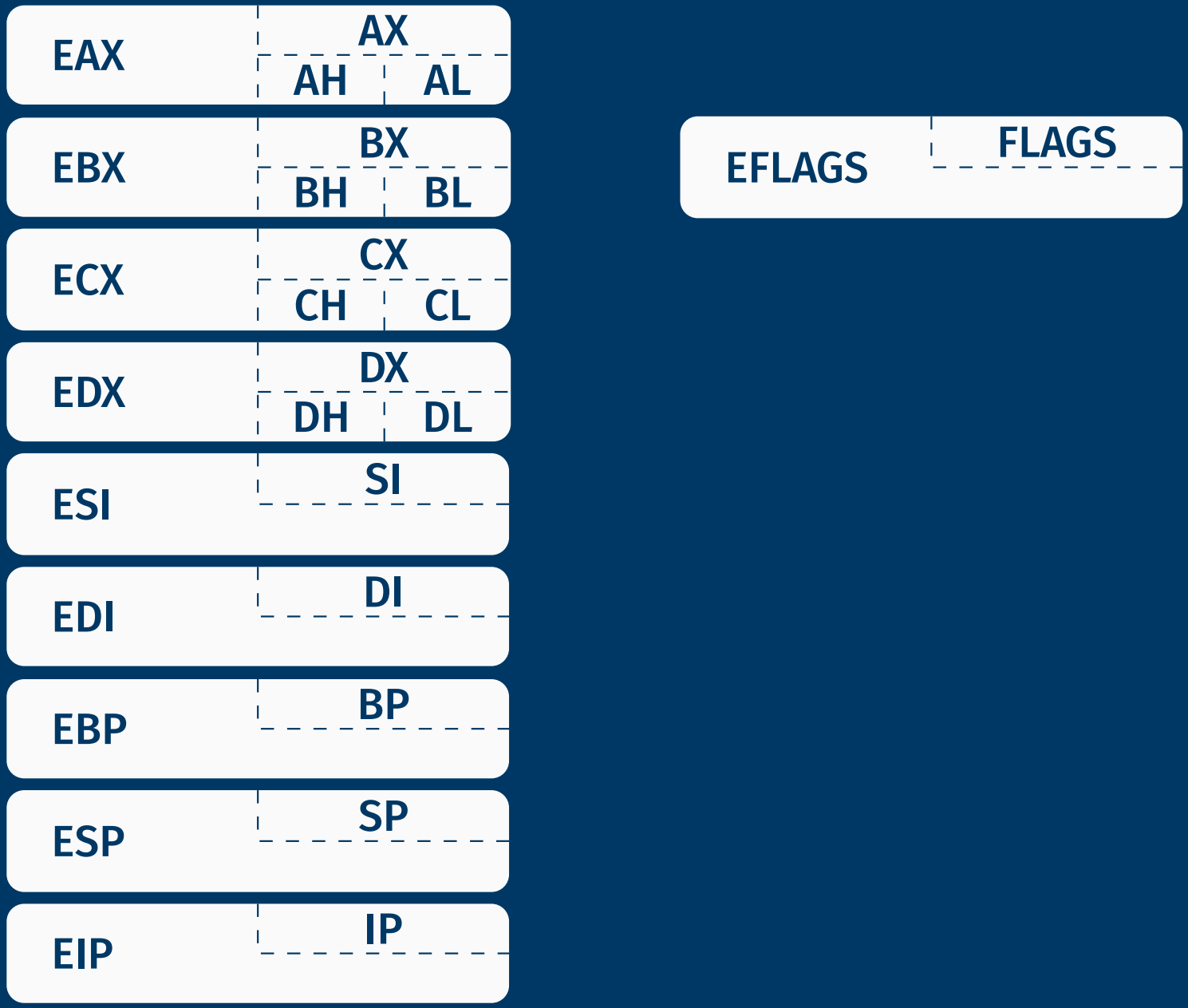
ursprünglich Zählerregister

Daten-/Allzweckregister

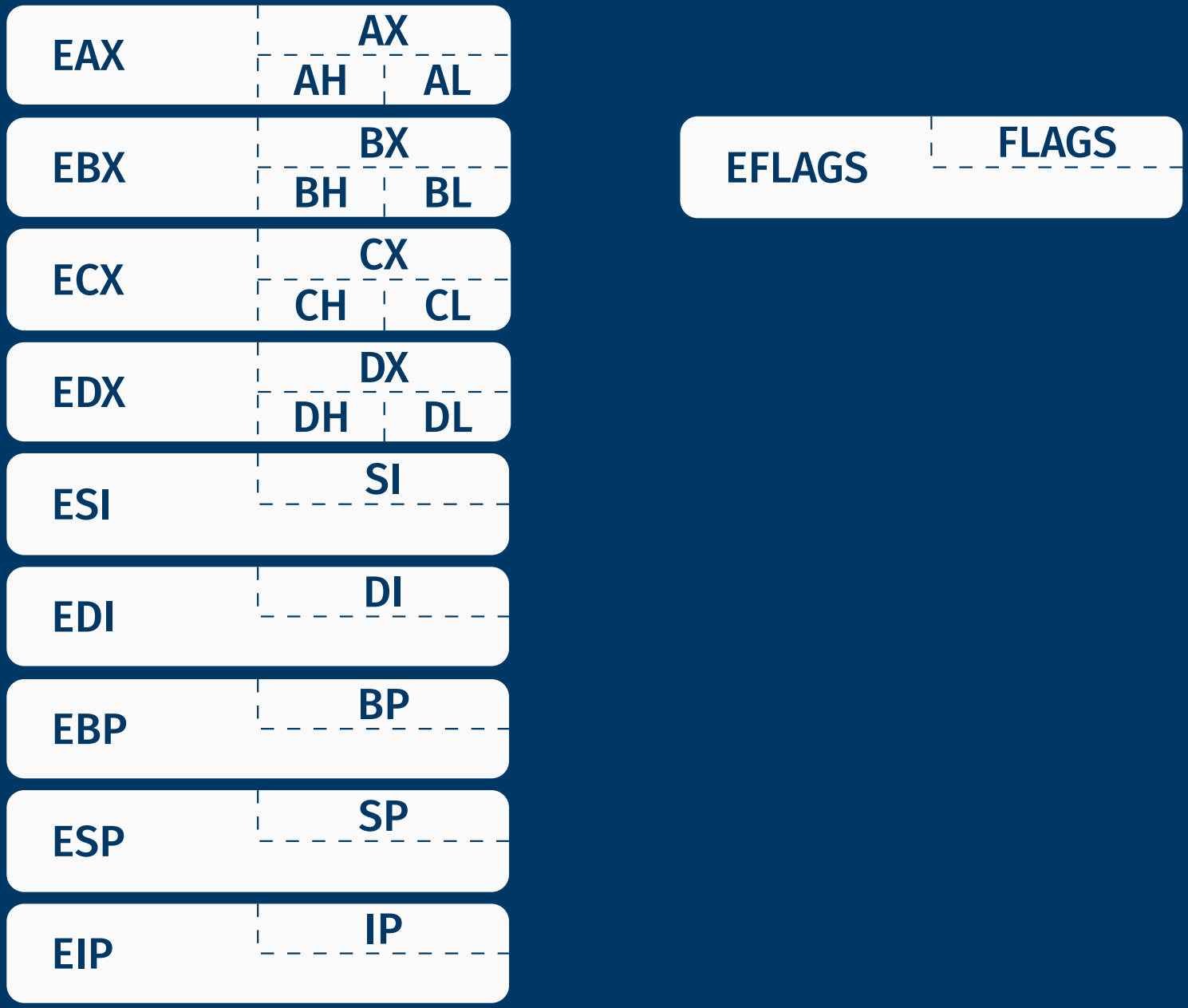

EAX
AX
AH
AL
EBX
BX
BH
BL
ECX
CX
CH
CL
EDX
DX
DH
DL
ESI
SI
ursprünglich Zeichenkettenquellindex
EDI
DI
ursprünglich Zeichenkettenzielindex


EAX
AX
AH
AL
EBX
BX
BH
BL
ECX
CX
CH
CL
EDX
DX
DH
DL
ESI
SI
EDI
DI
EBP
BP
Basiszeiger für Stapelrahmen
ESP
SP
Stapelzeiger

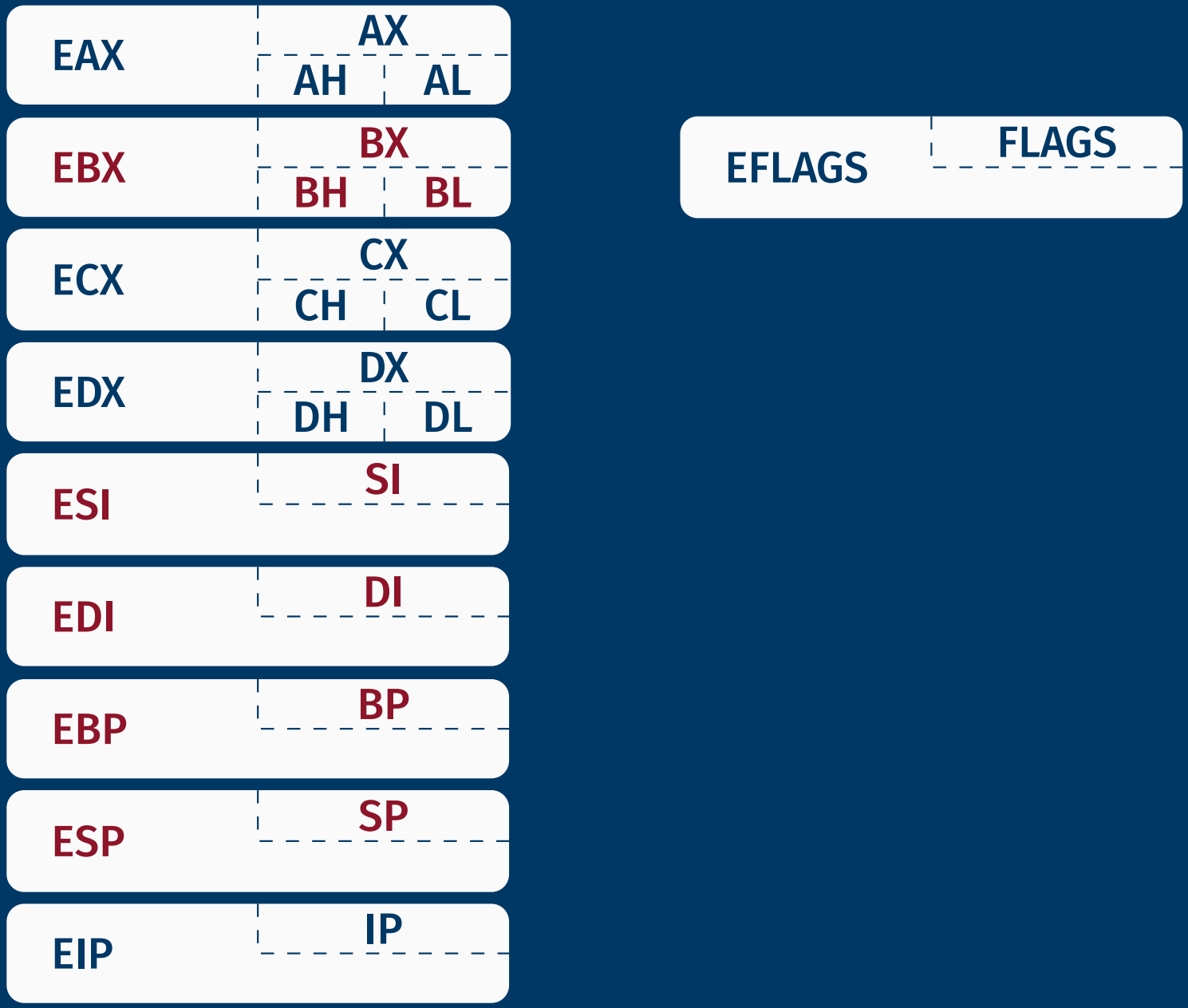

EAX
AX
AH
AL
EBX
BX
BH
BL
ECX
CX
CH
CL
EDX
DX
DH
DL
ESI
SI
EDI
DI
EBP
BP
ESP
SP
EIP
IP
Befehls-/Instruktionszeiger


EAX
AX
AH
AL
EBX
BX
BH
BL
ECX
CX
CH
CL
EDX
DX
DH
DL
ESI
SI
EDI
DI
EBP
BP
ESP
SP
EIP
IP
EFLAGS
FLAGS


EAX
AX
AH
AL
EBX
BX
BH
BL
ECX
CX
CH
CL
EDX
DX
DH
DL
ESI
SI
EDI
DI
EBP
BP
ESP
SP
EIP
IP
EFLAGS
FLAGS
CS
DS
SS
ES
FS
GS
+ FPU, MMX, SSE, ...



Flüchtige versus nicht-flüchtige Register



Flüchtige versus nicht-flüchtige Register

Struktur zur Kontextsicherung

```
struct toc {
  void *ebx;
  void *esi;
  void *edi;
  void *ebp;
  void *esp;
};
```

Struktur zur Kontextsicherung

```
struct toc {
  void *ebx;
  void *esi;
  void *edi;
  void *ebp;
  void *esp;
};
```

Kontext sichern nach

```
struct toc * toc_obj;
```

Struktur zur Kontextsicherung

```
struct toc {
  void *ebx;
  void *esi;
  void *edi;
  void *ebp;
  void *esp;
};
```

Kontext sichern nach

```
struct toc * toc_obj;

mov eax, toc_obj
mov [eax +  0], ebx
mov [eax +  4], esi
mov [eax +  8], edi
mov [eax + 12], ebp
mov [eax + 16], esp
```

Struktur zur Kontextsicherung

```
struct toc {
  void *ebx;
  void *esi;
  void *edi;
  void *ebp;
  void *esp;
};
```

Kontext sichern nach

```
struct toc * toc_obj;
```

```
mov eax, toc_obj
mov [eax + ebx_offset], ebx
mov [eax + esi_offset], esi
mov [eax + edi_offset], edi
mov [eax + ebp_offset], ebp
mov [eax + esp_offset], esp
```

Die Abstände sind auch in der vorgegebenen Datei `toc.inc` *definiert*

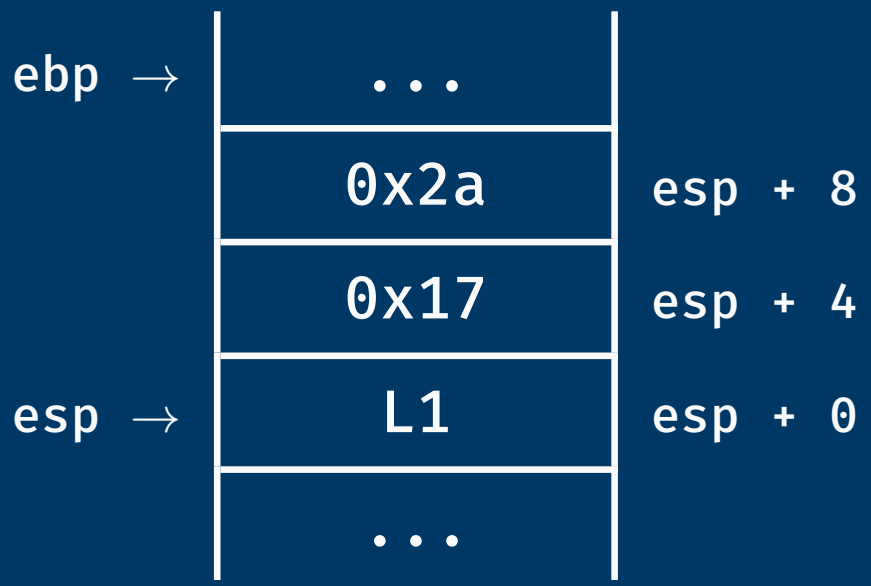
```
int i = add(23, 42);
```

int i = add(23, 42);

```
; 2. Parameter
    push 0x2a

; 1. Parameter
    push 0x17

; Funktionsaufruf
    call add

; Rücksprungadresse
L1:
```

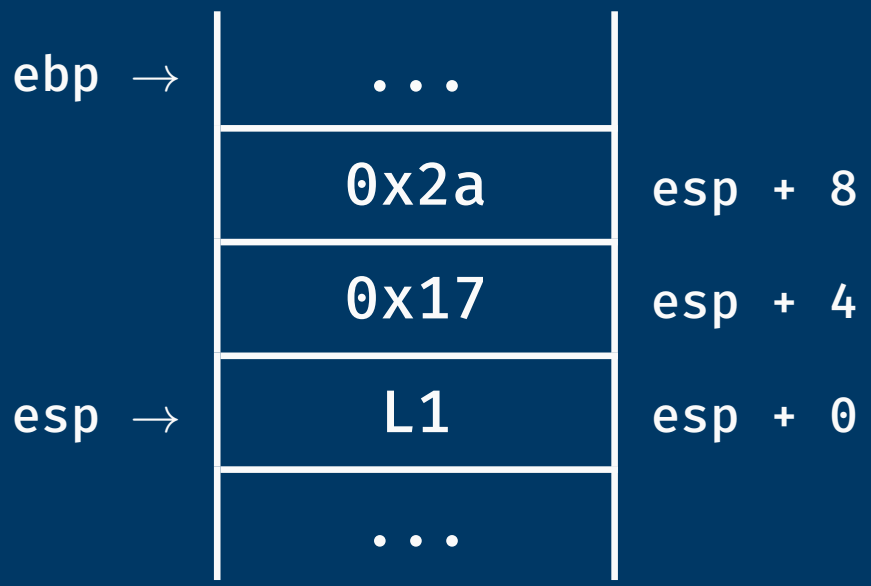
int i = add(23, 42);

```
; 2. Parameter
    push 0x2a

; 1. Parameter
    push 0x17

; Funktionsaufruf
    call add

; Rücksprungadresse
L1:
```



int i = add(23, 42);

```
; 2. Parameter
    push 0x2a

; 1. Parameter
    push 0x17

; Funktionsaufruf
    call add

; Rücksprungadresse
L1:

; SP wiederherstellen
    add esp, 8

; Rückgabe sichern
    mov ebx, eax
```

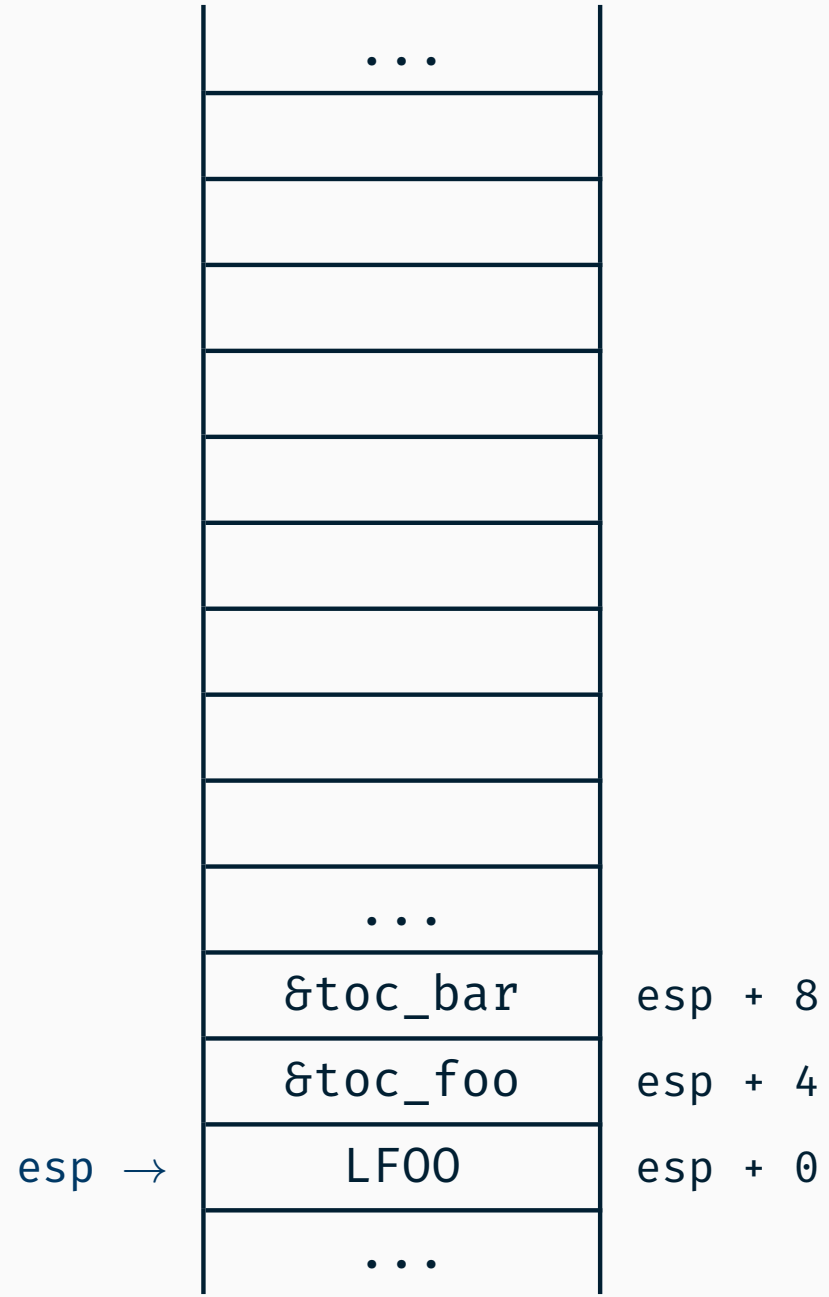


Kontextwechsel

```
void foo(){
  int f = 42;

  while (f--){
    kout << "foo"
         << f
         << endl;
    toc_switch(
        &toc_foo,
        &toc_bar
      );
LFOO:
  }
}
```

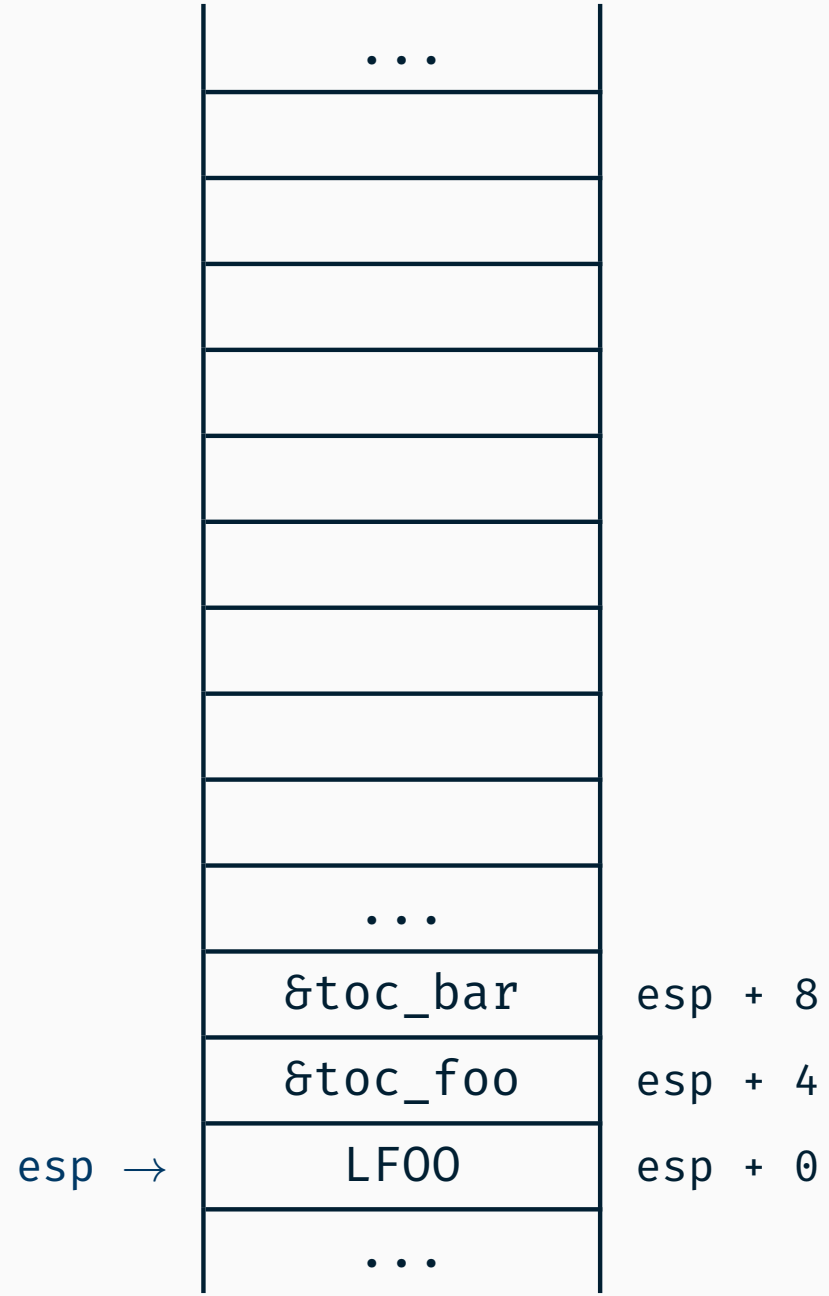
Kontextwechsel

```
void foo(){
  int f = 42;

  while (f--){
    kout << "foo"
         << f
         << endl;
    toc_switch(
        &toc_foo,
        &toc_bar
      );
LFOO:
  }
}
```



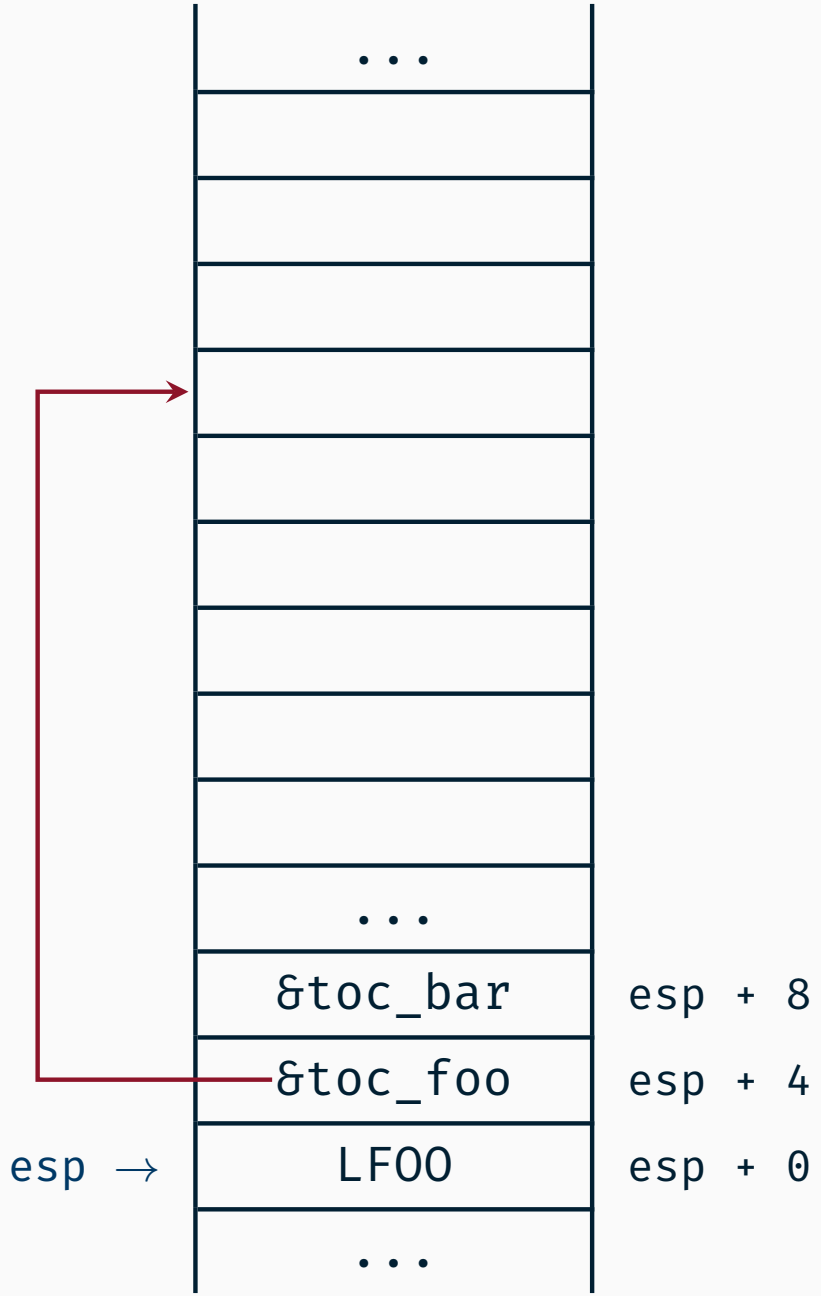
Kontextwechsel

```
; in Assembler
toc_switch:
```



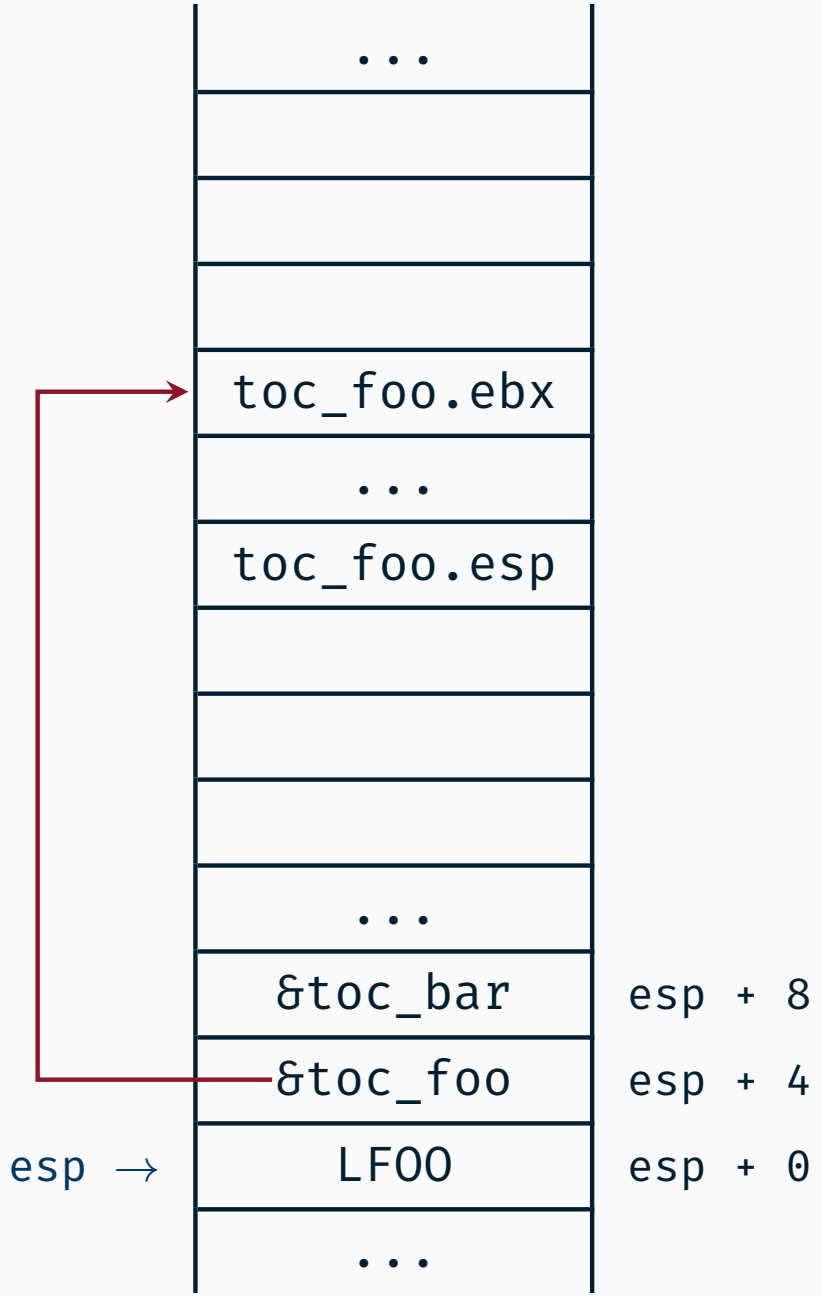
Kontextwechsel

```
; in Assembler
toc_switch:
```



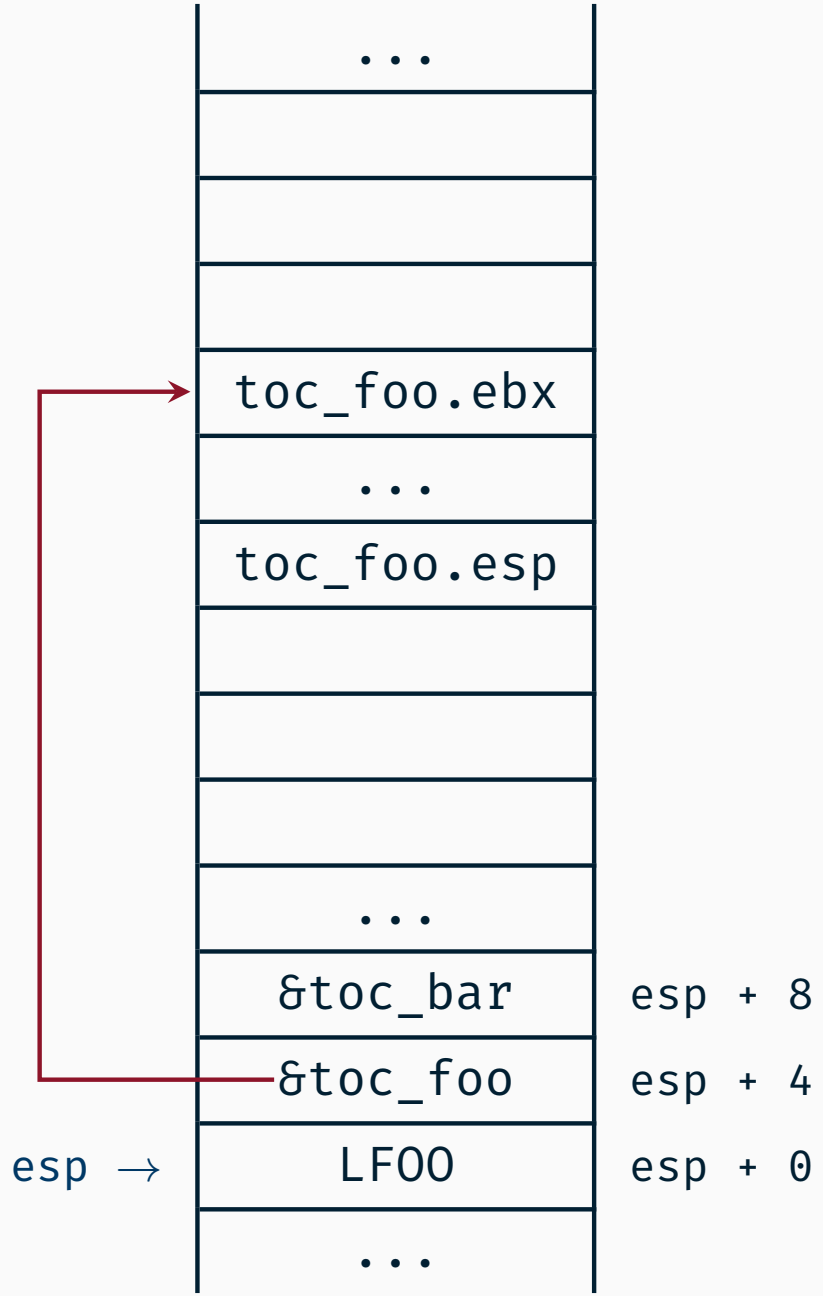
Kontextwechsel

```
; in Assembler
toc_switch:
```



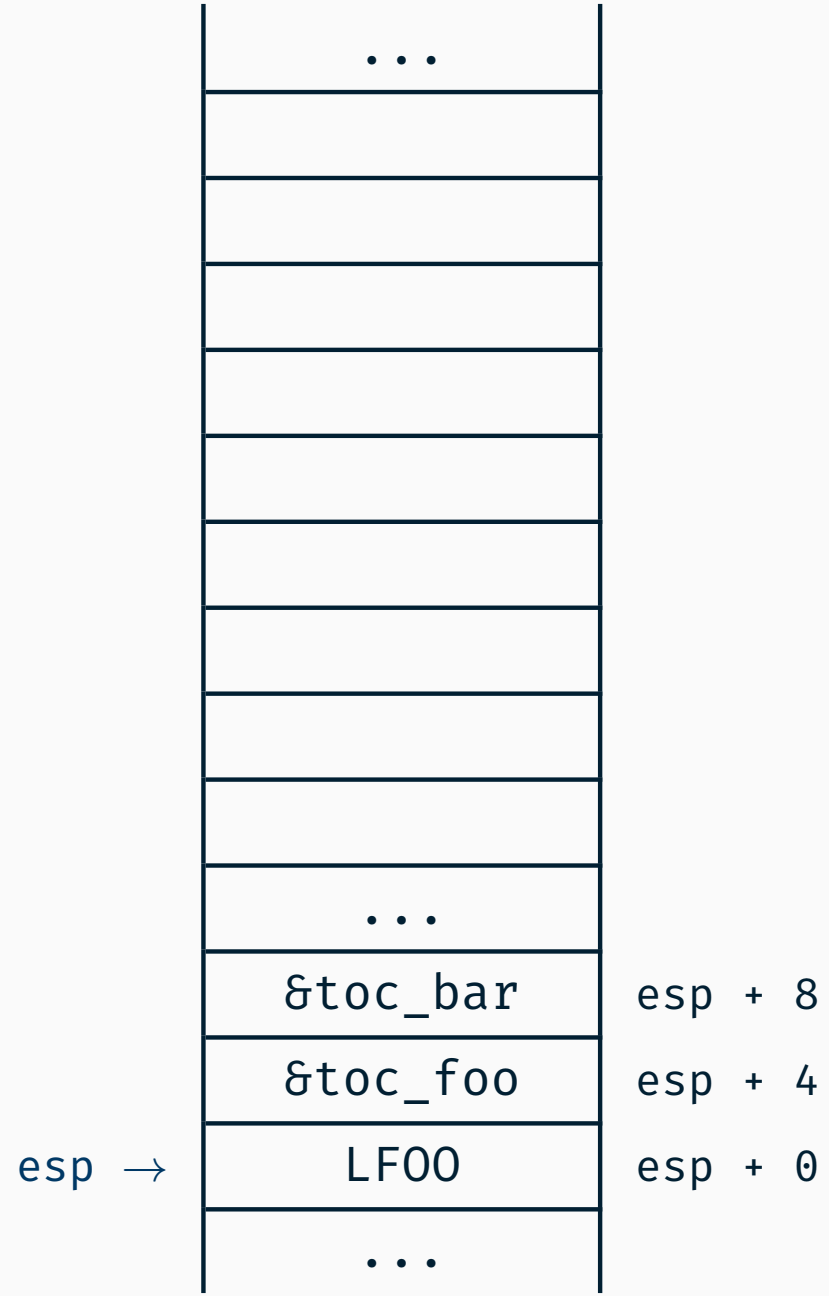
Kontextwechsel

```
; in Assembler
toc_switch:

; vorherige Register sichern
  mov eax, [esp + 4]
  mov [eax + ebx_offset], ebx
  ...
  mov [eax + esp_offset], esp
```



Kontextwechsel

```
; in Assembler
toc_switch:

; vorherige Register sichern
  mov eax, [esp + 4]
  mov [eax + ebx_offset], ebx
  ...
  mov [eax + esp_offset], esp
```



Kontextwechsel

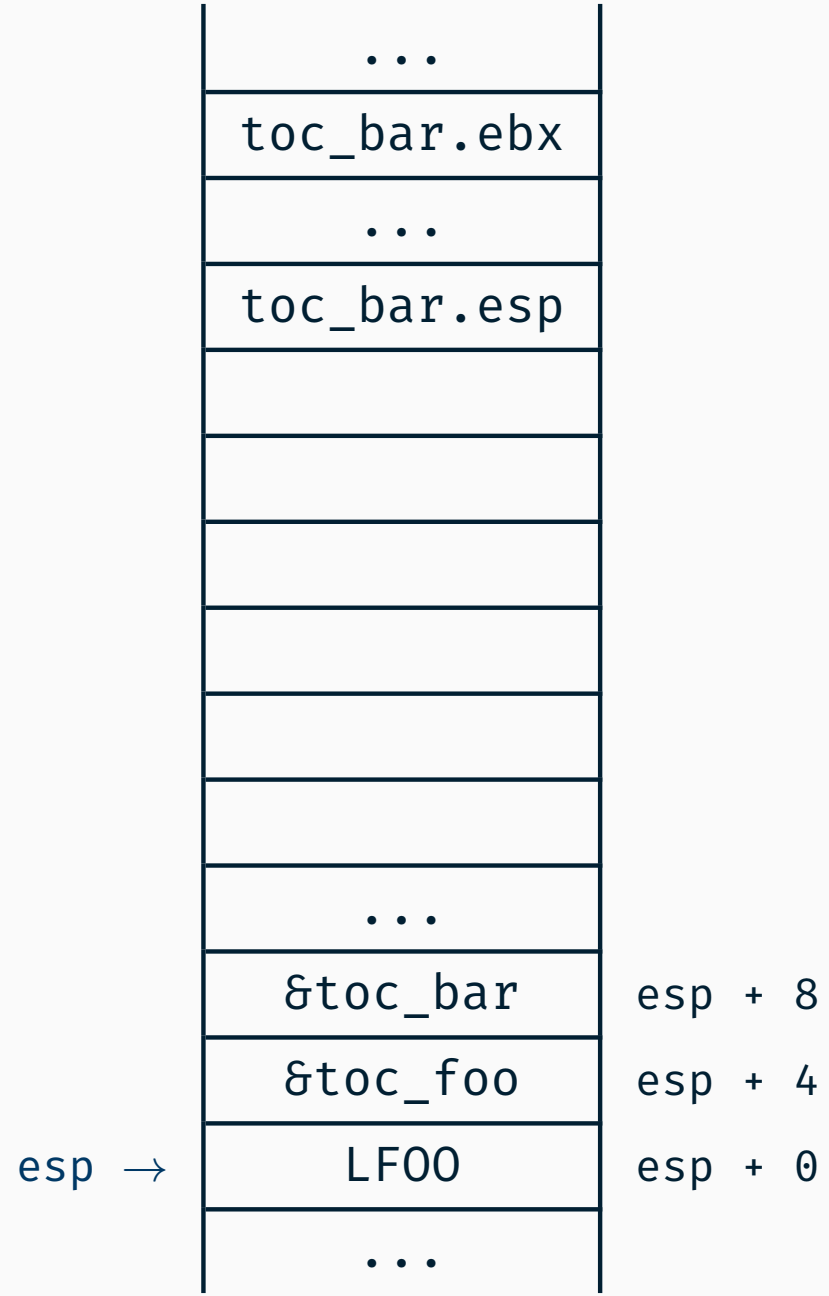
```
; in Assembler
toc_switch:

; vorherige Register sichern
  mov eax, [esp + 4]
  mov [eax + ebx_offset], ebx
  ...
  mov [eax + esp_offset], esp
```

Kontextwechsel

```
; in Assembler
toc_switch:

; vorherige Register sichern
  mov eax, [esp + 4]
  mov [eax + ebx_offset], ebx
  ...
  mov [eax + esp_offset], esp

; neue Register laden
  mov eax, [esp + 8]
  mov ebx, [eax + ebx_offset]
  ...
  mov esp, [eax + esp_offset]
```

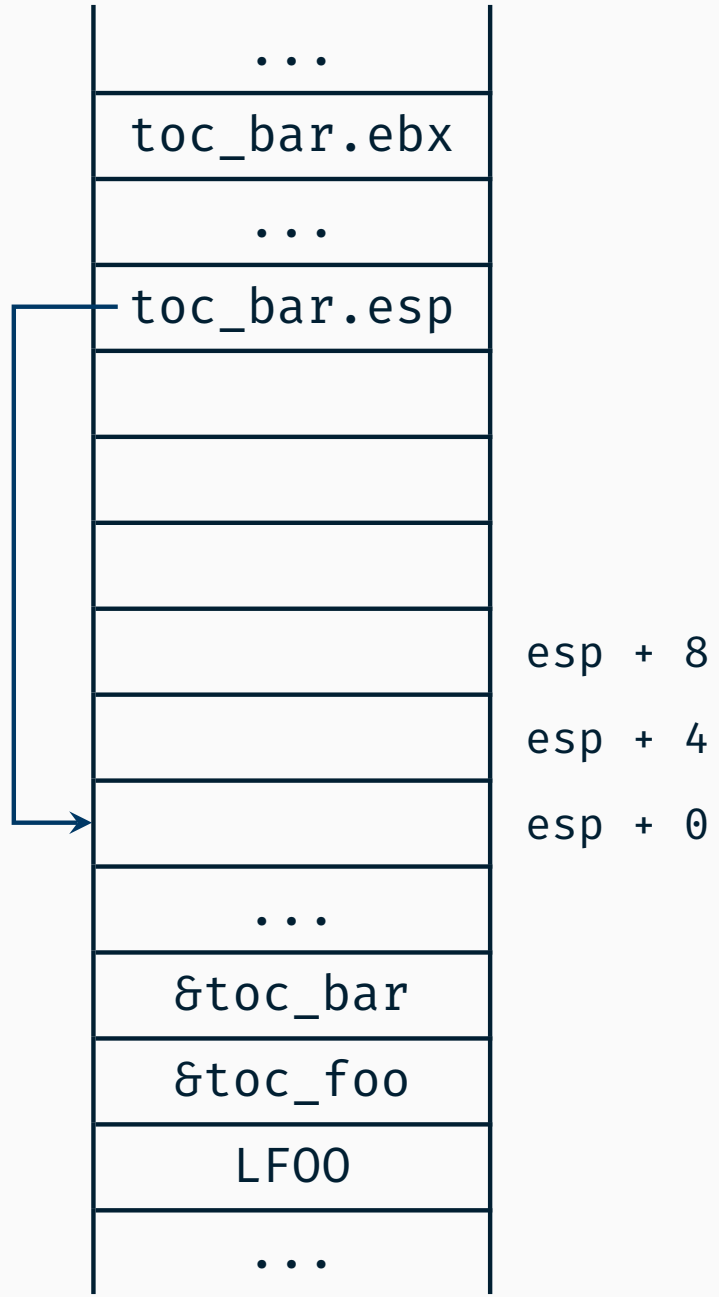
Kontextwechsel

```
; in Assembler
toc_switch:

; vorherige Register sichern
  mov eax, [esp + 4]
  mov [eax + ebx_offset], ebx
  ...
  mov [eax + esp_offset], esp

; neue Register laden
  mov eax, [esp + 8]
  mov ebx, [eax + ebx_offset]
  ...
  mov esp, [eax + esp_offset]
```



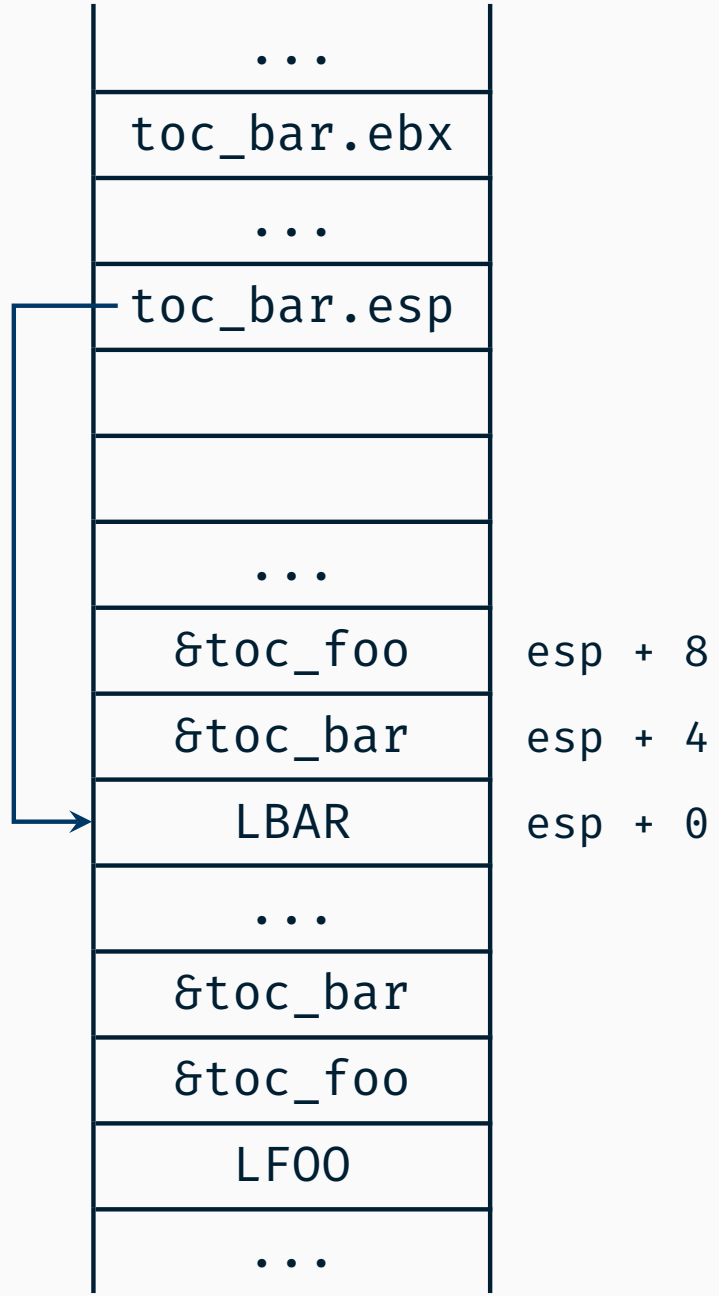
Kontextwechsel

```
; in Assembler
toc_switch:

; vorherige Register sichern
  mov eax, [esp + 4]
  mov [eax + ebx_offset], ebx
  ...
  mov [eax + esp_offset], esp

; neue Register laden
  mov eax, [esp + 8]
  mov ebx, [eax + ebx_offset]
  ...
  mov esp, [eax + esp_offset]
```



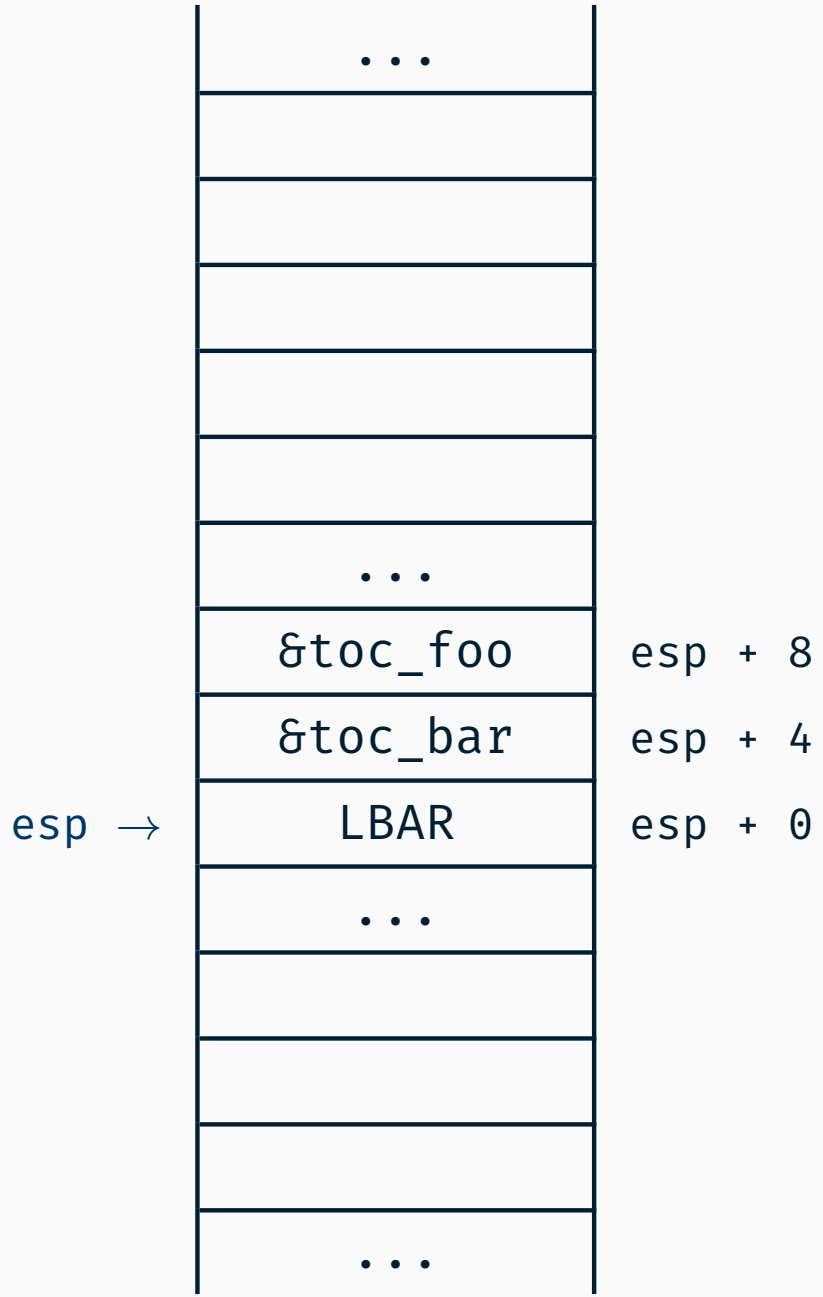
Kontextwechsel

```
; in Assembler
toc_switch:

; vorherige Register sichern
  mov eax, [esp + 4]
  mov [eax + ebx_offset], ebx
  ...
  mov [eax + esp_offset], esp

; neue Register laden
  mov eax, [esp + 8]
  mov ebx, [eax + ebx_offset]
  ...
  mov esp, [eax + esp_offset]
```



Kontextwechsel

```
; in Assembler
toc_switch:

; vorherige Register sichern
  mov eax, [esp + 4]
  mov [eax + ebx_offset], ebx
  ...
  mov [eax + esp_offset], esp

; neue Register laden
  mov eax, [esp + 8]
  mov ebx, [eax + ebx_offset]
  ...
  mov esp, [eax + esp_offset]

; "Zurückkehren"
  ret
```



Koroutine (erstmalig) starten

```
void coroutine(void *arg){ ... }
```

Koroutine (erstmalig) starten

```
void coroutine(void *arg){ ... }
```

- Wie rufe ich die aller erste Koroutine auf?

Koroutine (erstmalig) starten

```
void coroutine(void *arg){ ... }
```

- Wie rufe ich die aller erste Koroutine auf?
  - Kann ich mit leeren Registern starten?

Koroutine (erstmalig) starten

```
void coroutine(void *arg){ ... }
```

- Wie rufe ich die aller erste Koroutine auf?
  - Kann ich mit leeren Registern starten? Nicht ganz.
  - `toc_go` (halbes `toc_switch`, ohne sichern des aktuellen Kontexts)

Koroutine (erstmalig) starten

```
void coroutine(void *arg){ ... }
```

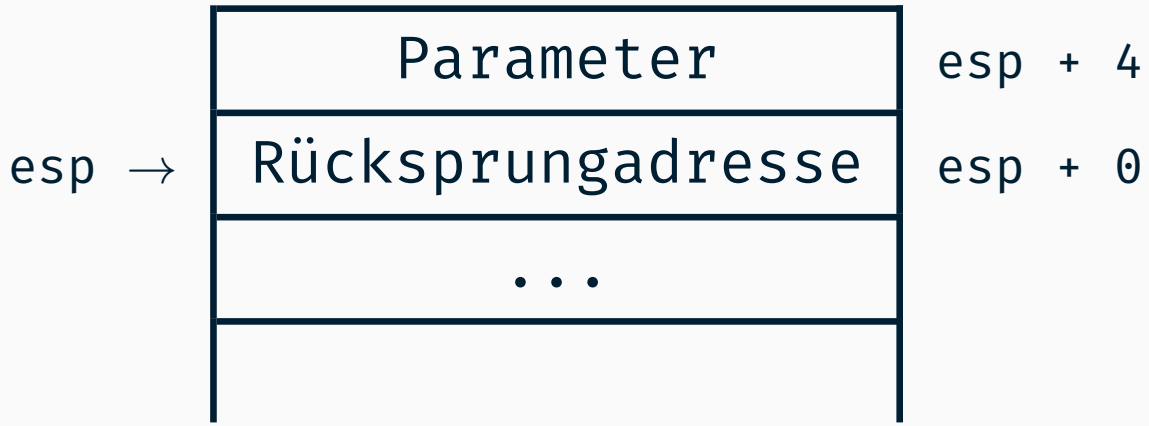
- Wie rufe ich die aller erste Koroutine auf?
  - Kann ich mit leeren Registern starten? Nicht ganz.
  - `toc_go` (halbes `toc_switch`, ohne sichern des aktuellen Kontexts)
- Was wird für jede Koroutine gebraucht?

Koroutine (erstmalig) starten

```
void coroutine(void *arg){ ... }
```

- Wie rufe ich die aller erste Koroutine auf?
  - Kann ich mit leeren Registern starten? Nicht ganz.
  - `toc_go` (halbes `toc_switch`, ohne sichern des aktuellen Kontexts)
- Was wird für jede Koroutine gebraucht?
  - Kontextstruktur `toc`
  - eigenen, präparierter Stack

Stack aufsetzen



Stack aufsetzen

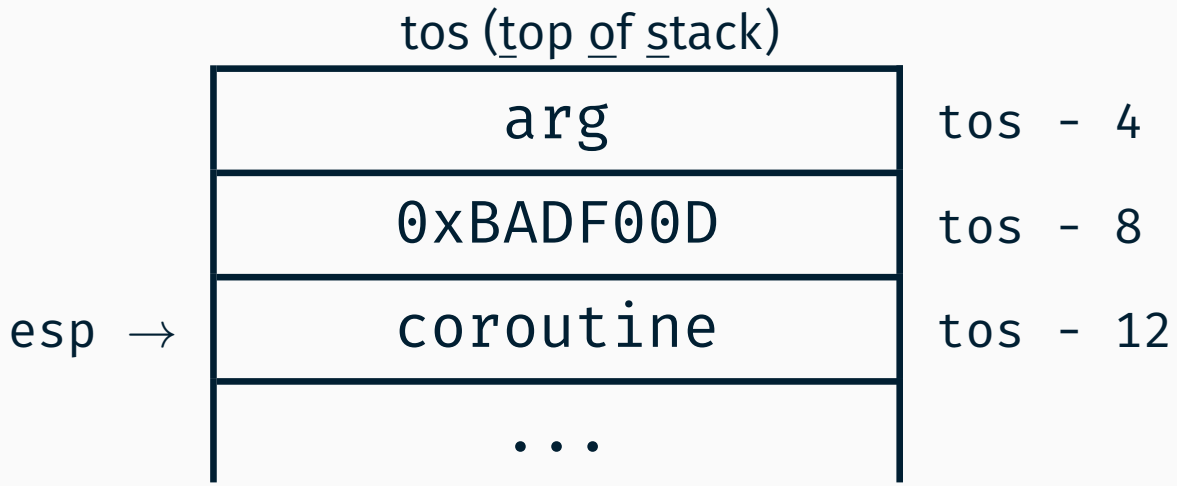
```
void coroutine(void *arg){ ... }
```

Stack aufsetzen

```
void coroutine(void *arg){ ... }
```

Stack aufsetzen

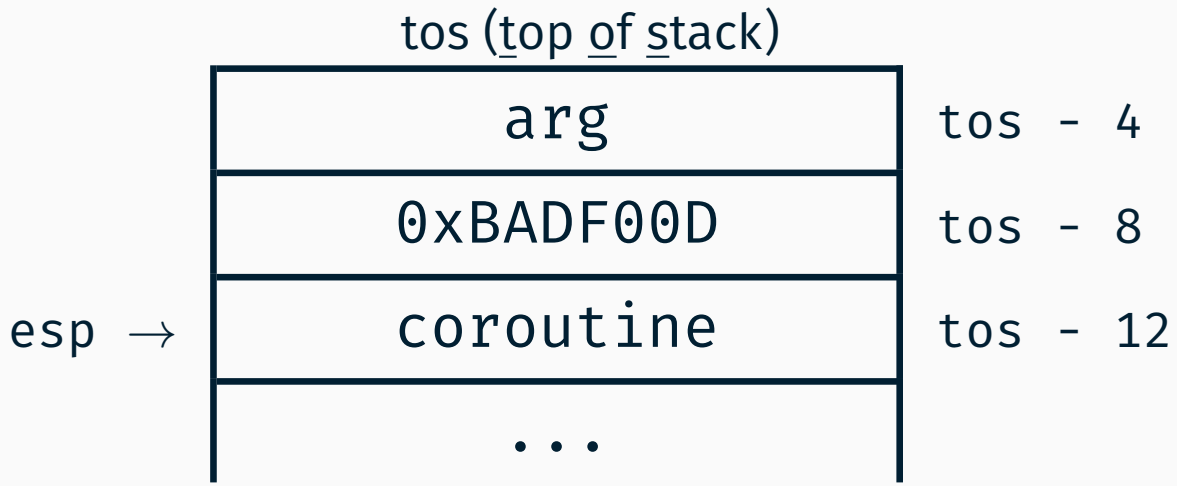
```
void coroutine(void *arg){ ... }
```



was passiert nun bei `toc_go`/`toc_switch`?

Stack aufsetzen

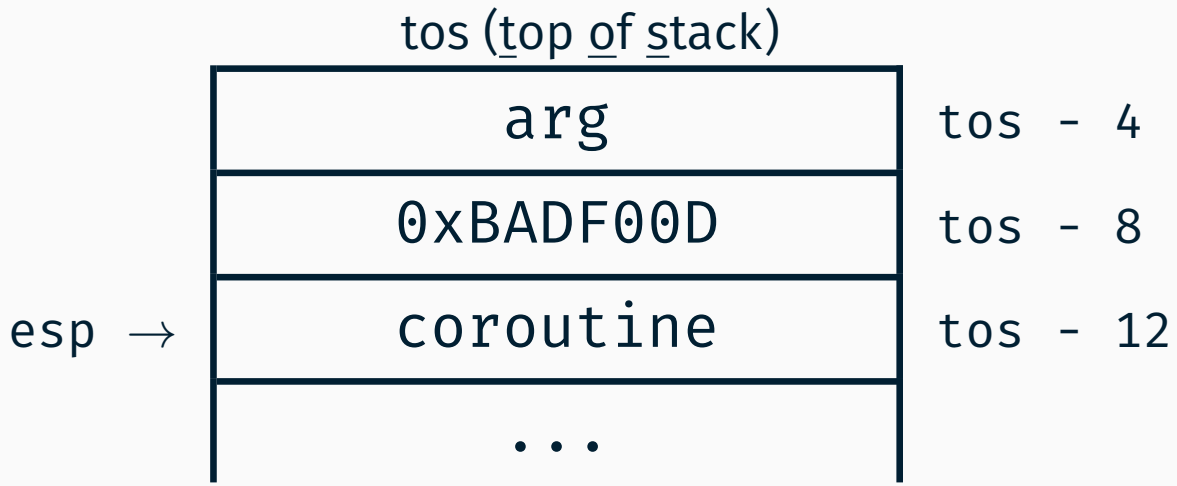
```
void coroutine(void *arg){ ... }
```



was passiert wenn `coroutine` abgearbeitet ist?

Stack aufsetzen

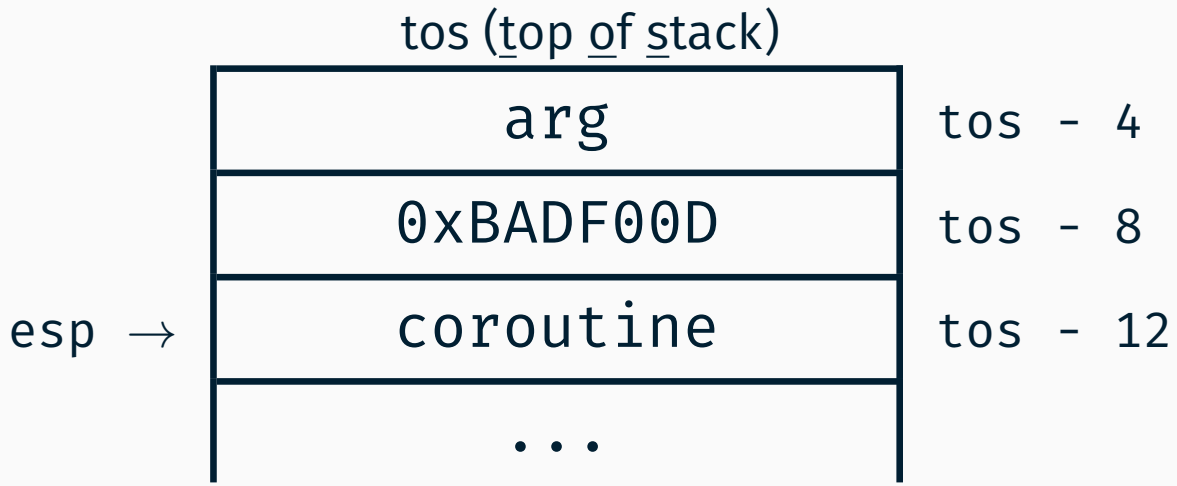
```
void coroutine(void *arg){ ... }
```



muss in `toc_settle` implementiert werden

Stack aufsetzen

```
void coroutine(void *arg){ ... }
```

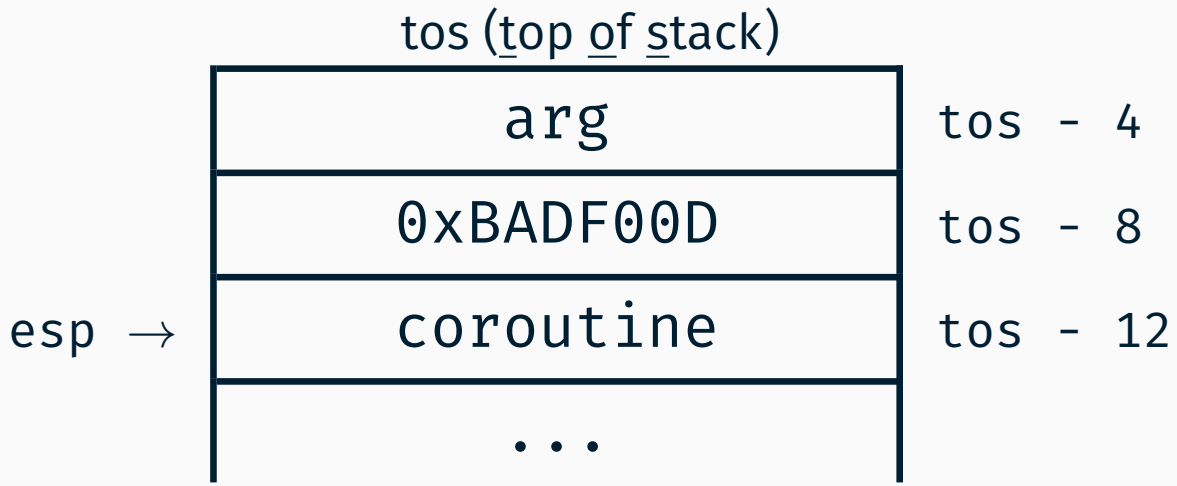


muss in `toc_settle` implementiert werden

- in C[++] (nicht Assembler)

Stack aufsetzen

```
void coroutine(void *arg){ ... }
```

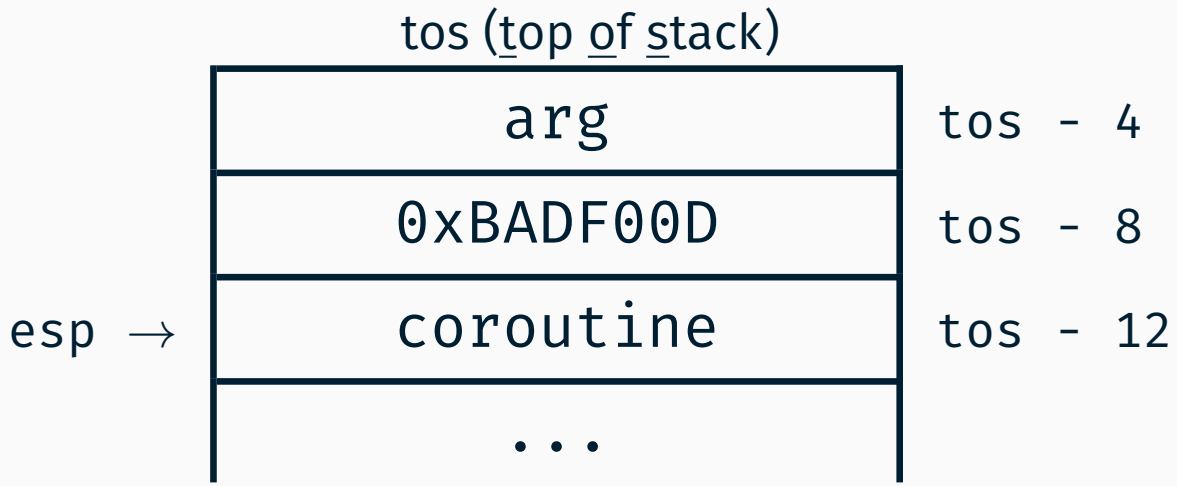


muss in `toc_settle` implementiert werden

- in C[++] (nicht Assembler)
- statisch übergebenen Speicher als Stack aufsetzen
  `void **esp = (void **) tos;`

Stack aufsetzen

```
void coroutine(void *arg){ ... }
```

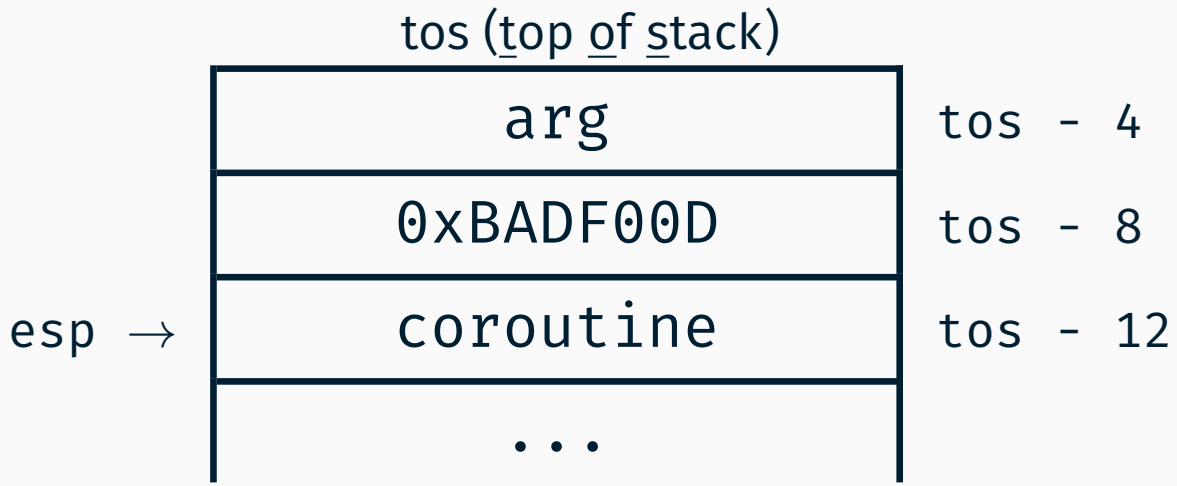


muss in `toc_settle` implementiert werden

- in C[++] (nicht Assembler)
- statisch übergebenen Speicher als Stack aufsetzen
  `void **esp = (void **) tos;`
  Pointerarithmetik ist toll: `*(--esp) = arg;`

Stack aufsetzen

```
void coroutine(void *arg){ ... }
```



muss in `toc_settle` implementiert werden

- in C[++] (nicht Assembler)
- statisch übergebenen Speicher als Stack aufsetzen
  `void **esp = (void **) tos;`
  Pointerarithmetik ist toll: `*(--esp) = arg;`
- `esp` in Kontextstruktur `toc` setzen

Umsetzung in OOStuBS/MPStuBS

Threads

```
class Thread {
    struct toc regs;
  public:
    Thread(void* tos);
    virtual void action() = 0;
}
```

Threads

```
class Thread {
    struct toc regs;
  public:
    Thread(void* tos);
    virtual void action() = 0;
}
```

Adresse einer virtuellen Member-Funktion nicht (ohne weiteres) ermittelbar

Threads

```
class Thread {
    struct toc regs;
  public:
    Thread(void* tos);
    virtual void action() = 0;
}
```

Adresse einer virtuellen Member-Funktion nicht (ohne weiteres) ermittelbar

```
void kickoff (Thread* t){
  t->action();
}
```

Scheduler

Scheduler verwaltet Koroutinen zentral

```
class Scheduler {
    Queue<Thread> routines;
  public:
    void resume();
};
```

Scheduler

Scheduler verwaltet Koroutinen zentral

```
class Scheduler {
    Queue<Thread> routines;
  public:
    void resume();
};
```

OOStuBS immer synchrone Aufrufe
$\rightarrow$ Konsistenz gesichert

Scheduler

Scheduler verwaltet Koroutinen zentral

```
class Scheduler {
    Queue<Thread> routines;
  public:
    void resume();
};
```

OOStuBS immer synchrone Aufrufe
$\rightarrow$ Konsistenz gesichert

MPStuBS Synchronisation notwendig

Dispatcher

Synchronisation auf Epilogebene

Dispatcher

Synchronisation auf Epilogebene

```
main(){
  ...
  guard.enter();
  scheduler.schedule();
}
```

Dispatcher

Synchronisation auf Epilogebene

```
main(){
  ...
  guard.enter();
  scheduler.schedule();
}
```

```
toc_go:
  ...
  ret
```

Dispatcher

Synchronisation auf Epilogebene

```
main(){
  ...
  guard.enter();
  scheduler.schedule();
}
```

```
toc_go:
  ...
  ret
```

```
kickoff(Thread* t){
  guard.leave();
  t->action();
}
```

Syntaxunterschiede bei x86-Assembler

Intel:

```
mov ebx, 0x17
```

(Ziel, Quelle)

`objdump -M intel`

AT&T:

```
mov $0x17, %ebx
```

(Quelle, Ziel)

Standard bei `objdump`

Fragen?

Nächste Woche (13. & 15. Dezember)
Abgabe von Aufgabe 3 im Huber-CIP