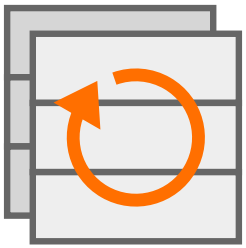


Operating System Group

TUHH

Technische Universität Hamburg

BSB – Übung 4: Kontextwechsel

Yannick Loeck

2022-06-01

Inhalt

TUHH
Technische Universität Hamburg

BSB - HÜ4

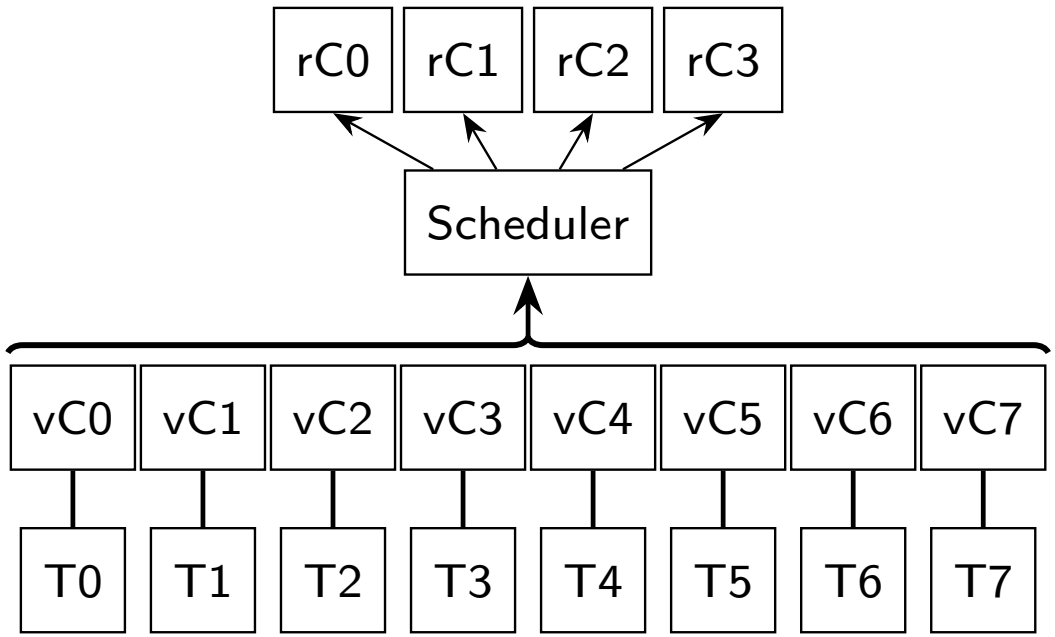
Mehrere Threads auf einer CPU ausführen.

Worum geht es in der Übung heute:

- Kooperativer Kontextwechsel
- Threads, Scheduler, Dispatcher

Aufteilung einer CPU

- Ziel: Ausführung mehrerer unabhängiger Aufgaben auf einer CPU
- “Aufgabe”: Thread mit Kontext
- Jeder Thread bekommt vom OS virtuelle CPU
- vCPU bietet selbes Interface wie echte CPU
- OS wechselt zwischen vCPUs



Aufteilung einer CPU: Multiplexing

TUHH
Technische Universität Hamburg

Räumlich

- Register auf vCPUs aufteilen
- Mehrere Zustände gleichzeitig

Zeitlich

- Threads bekommen CPU für gewisse Zeit
- OS wechselt zwischen Thread und Zuständen
- Quasiparallelität

Threadwechsel: Architektur

- Kooperativ: Thread initiiert Wechsel zu anderem Thread
- Scheduler verwaltet Thread-Liste
- Dispatcher führt tatsächliche Umschaltung durch
- Wechsel durch Austauschen des Kontexts
- Für Anwendungen nicht erkennbar dass sie als Thread laufen

Kontextwechsel


```
void context_switch(Context* current, Context* next);
```

Was brauchen wir um zu einem Thread zu wechseln?

- Register
  - Flüchtige Register (eax, ecx, edx)
  - CPU-Statuswort (eflags)
  - Program Counter (eip)
  - Nichtflüchtige Register (esp, ebp, edi, esi, ebx)
- Speicher:
  - Globale Variablen
  - Heap
  - Stackbereich (ein Stack pro Thread)

Kontextwechsel

TUHH
Technische Universität Hamburg

```
void context_switch(Context* current, Context* next);
```

Was brauchen wir um zu einem Thread zu wechseln?

- Register
  - Flüchtige Register (eax, ecx, edx) ⇒ compiler sichert
  - CPU-Statuswort (eflags) ⇒ compiler sichert
  - Program Counter (eip)
  - Nichtflüchtige Register (esp, ebp, edi, esi, ebx)
- Speicher:
  - Globale Variablen
  - Heap
  - Stackbereich (ein Stack pro Thread)

Kontextwechsel


```
void context_switch(Context* current, Context* next);
```

Was brauchen wir um zu einem Thread zu wechseln?

- Register
  - Flüchtige Register (eax, ecx, edx) ⇒ compiler sichert
  - CPU-Statuswort (eflags) ⇒ compiler sichert
  - Program Counter (eip) ⇒ call-Instruktion sichert
  - Nichtflüchtige Register (esp, ebp, edi, esi, ebx)
- Speicher:
  - Globale Variablen
  - Heap
  - Stackbereich (ein Stack pro Thread)

Kontextwechsel

TUHH
Technische Universität Hamburg

```
void context_switch(Context* current, Context* next);
```

Was brauchen wir um zu einem Thread zu wechseln?

- Register
  - Flüchtige Register (eax, ecx, edx) ⇒ compiler sichert
  - CPU-Statuswort (eflags) ⇒ compiler sichert
  - Program Counter (eip) ⇒ call-Instruktion sichert
  - Nichtflüchtige Register (esp, ebp, edi, esi, ebx)
- Speicher:
  - Globale Variablen ⇒ geteilt
  - Heap ⇒ haben wir nicht
  - Stackbereich (ein Stack pro Thread)

Kontextwechsel

TUHH
Technische Universität Hamburg

```
void context_switch(Context* current, Context* next);
```

Was brauchen wir um zu einem Thread zu wechseln?

- Register
  - Flüchtige Register (eax, ecx, edx) ⇒ compiler sichert
  - CPU-Statuswort (eflags) ⇒ compiler sichert
  - Program Counter (eip) ⇒ call-Instruktion sichert
  - Nichtflüchtige Register (esp, ebp, edi, esi, ebx) ⇒ müssen wir sichern
- Speicher:
  - Globale Variablen ⇒ geteilt
  - Heap ⇒ haben wir nicht
  - Stackbereich (ein Stack pro Thread) ⇒ über esp gesichert

Funktion für Kontextwechsel


```
void context_switch(Context* current, Context* next);
```

- In Assembler implementieren
- nasm mit Intel Syntax (`<OP> <ZIEL>, <QUELLE>`)
- Nicht-flüchtige Register in current sichern (`mov`)
- Nicht-flüchtige Register aus next wiederherstellen
- Mit ret zurückkehren

Erster Start eines Threads

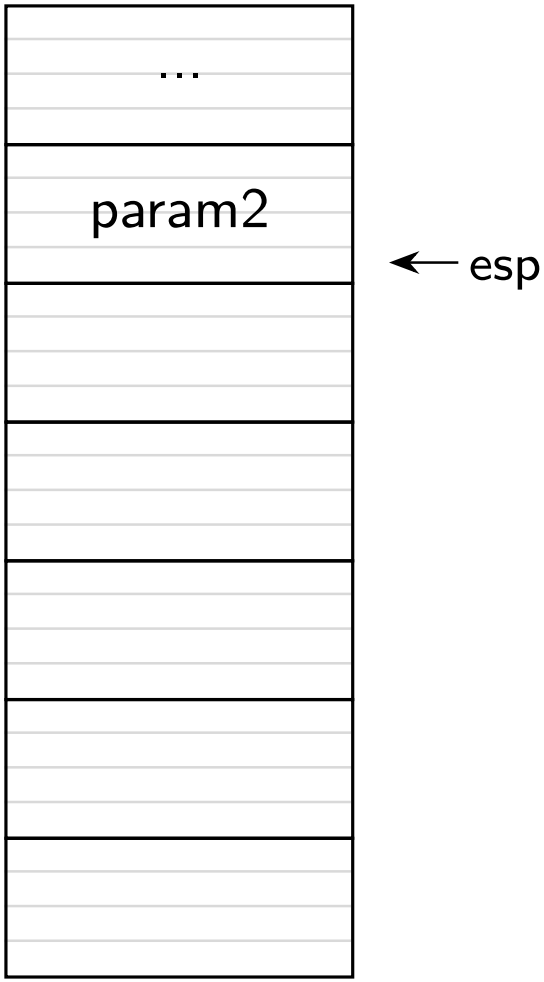
TUHH
Technische Universität Hamburg

- Wenn ein Thread noch nie lief, was sollen wir wiederherstellen?
- Lösung: Zustand für neuen Thread nachbauen
- 2 Datenstrukturen die befüllt werden müssen:
- Stack
- Context
  - Stack-Pointer esp soll auf neuen Stack zeigen
  - Nichtflüchtige Register mit 0 initialisieren

Stack

Funktionen mit Parametern - Stackansicht

Stack

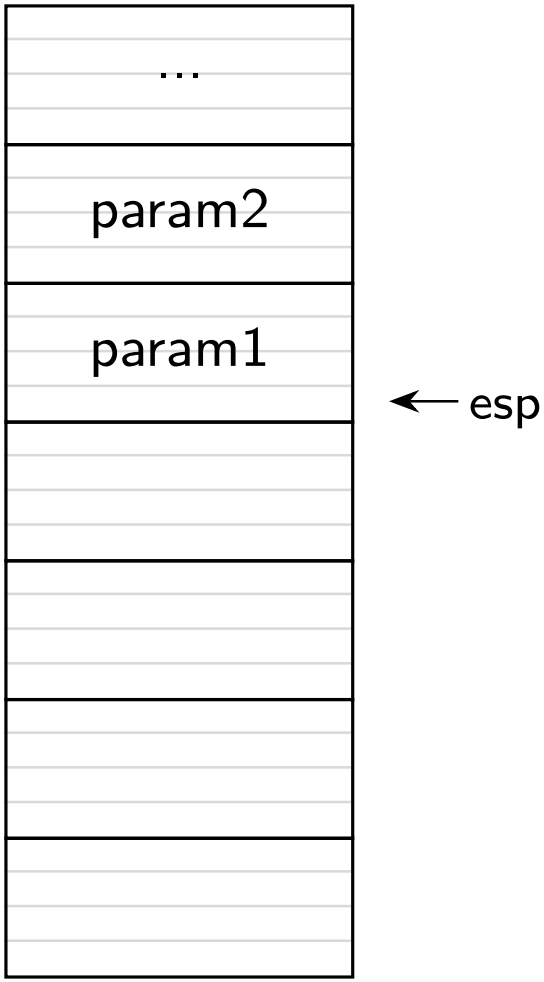


Programm

```
push param2
```

Funktionen mit Parametern - Stackansicht

Stack

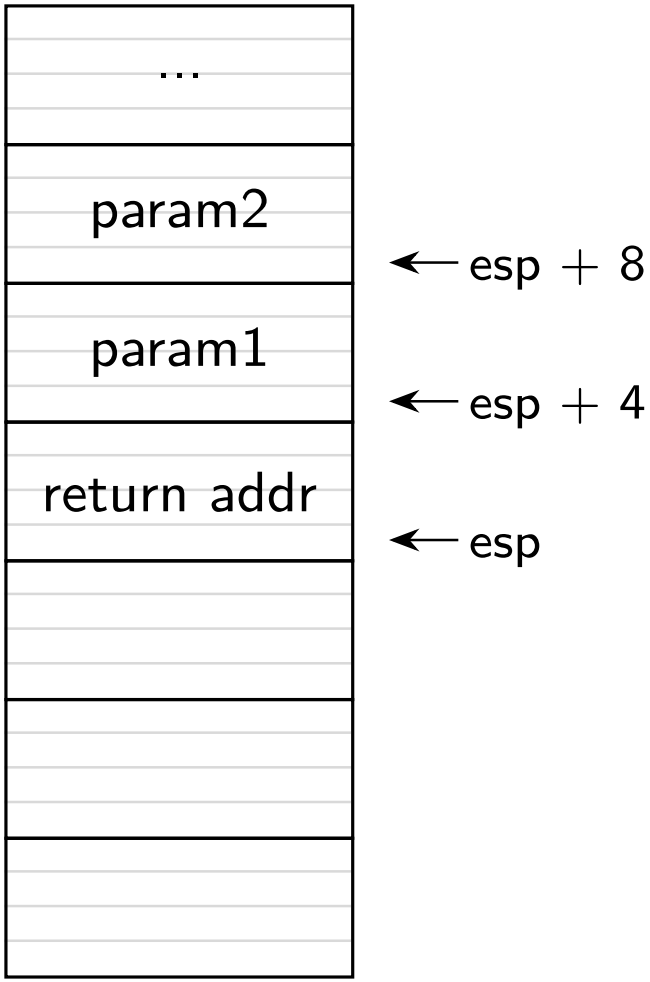


Programm

```
push param2
push param1
```

Funktionen mit Parametern - Stackansicht

Stack



Programm

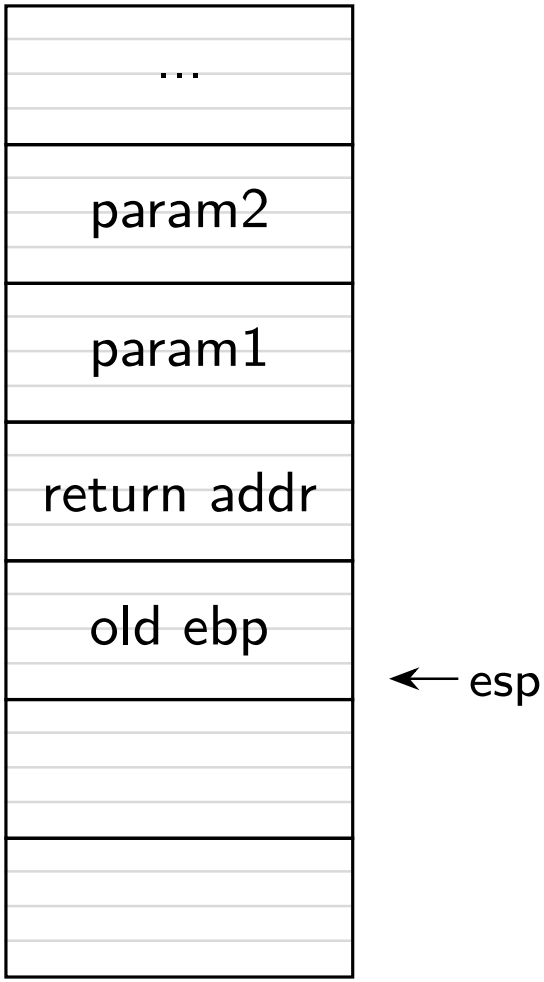
```
push param2
push param1
call func
```

`return addr:`

`func:`

Funktionen mit Parametern - Stackansicht

Stack



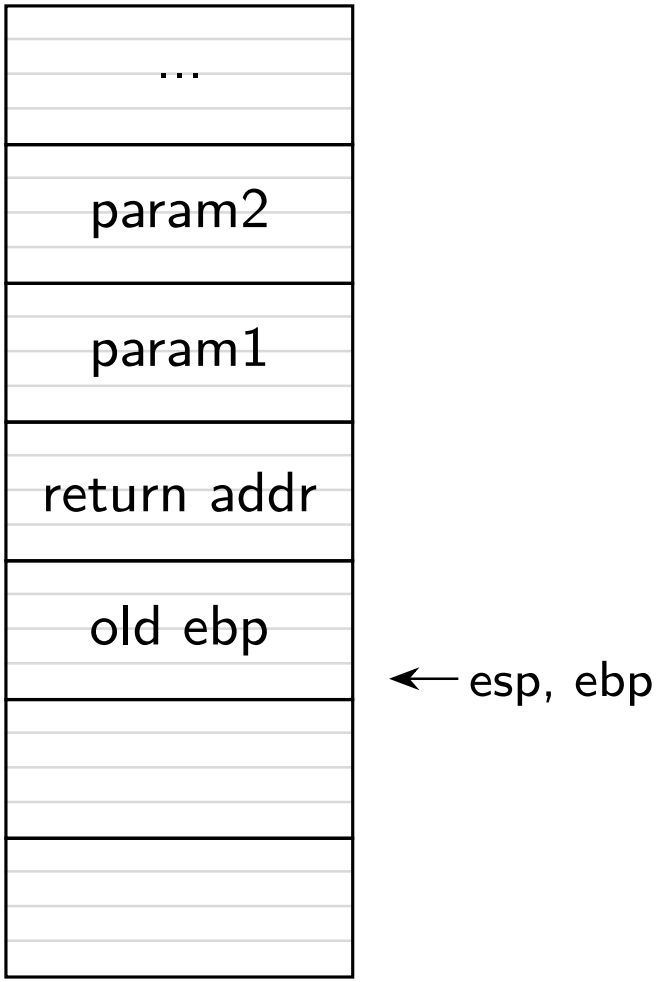
Programm

```
                push param2
                push param1
                call func
return addr:

        func:   push ebp
```

Funktionen mit Parametern - Stackansicht

Stack



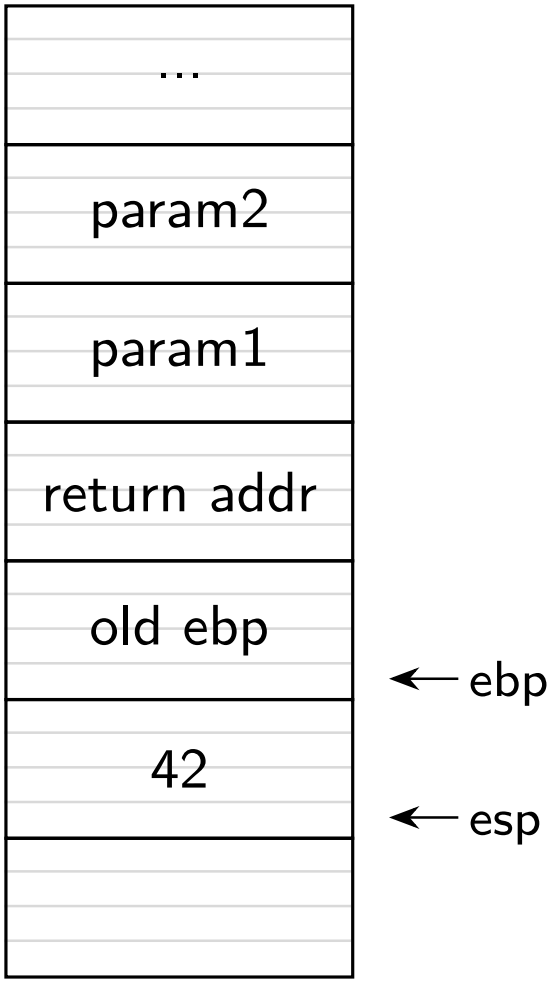
Programm

```
                push param2
                push param1
                call func
return addr:

        func:   push ebp
                mov ebp, esp
```

Funktionen mit Parametern - Stackansicht

Stack



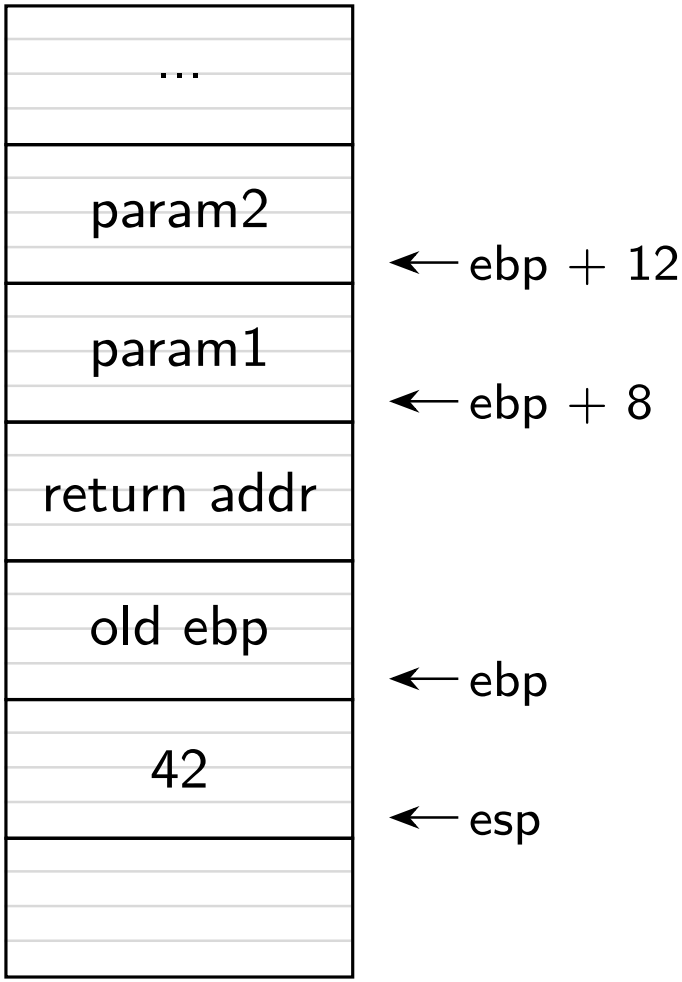
Programm

```
                push param2
                push param1
                call func
return addr:

        func:   push ebp
                mov ebp, esp
                push ebx
```

Funktionen mit Parametern - Stackansicht

Stack

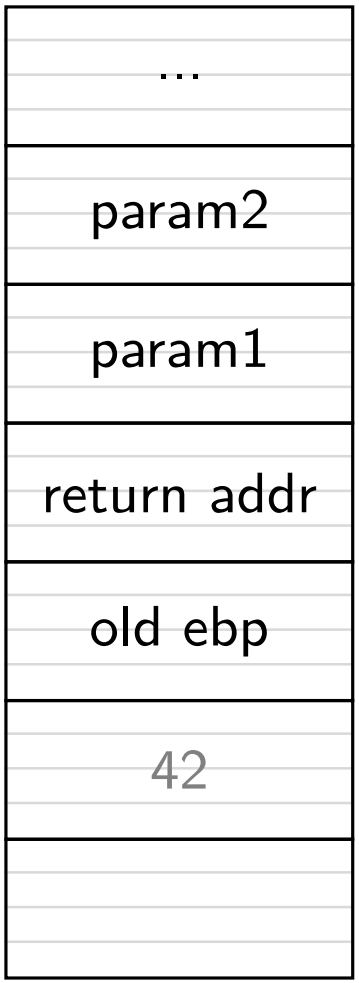


Programm

```
                    push param2
                    push param1
                    call func
return addr:

func:               push ebp
                    mov ebp, esp
                    push ebx
```

Funktionen mit Parametern - Stackansicht

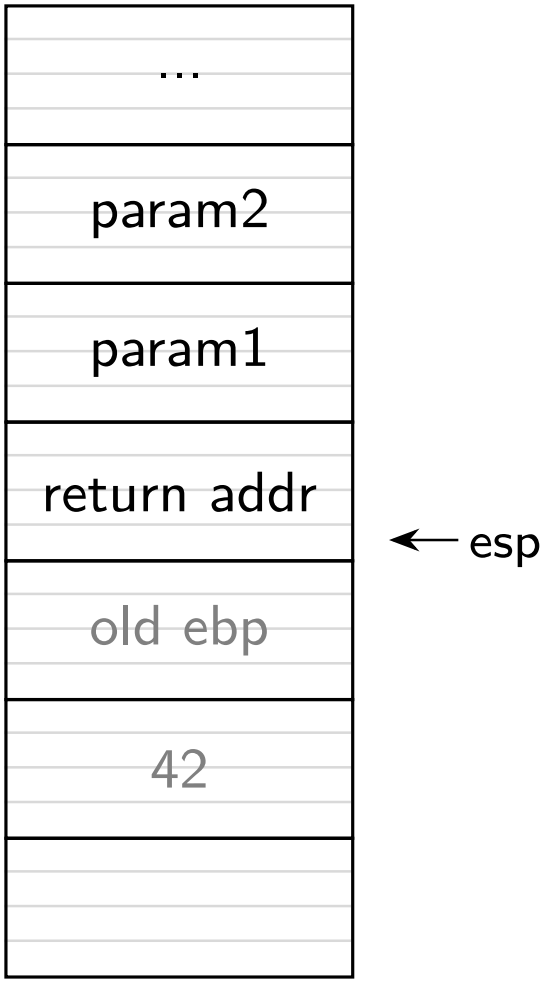


Programm

```
                    push param2
                    push param1
                    call func
return addr:

            func:   push ebp
                    mov ebp, esp
                    push ebx
                    mov esp, ebp
```

Funktionen mit Parametern - Stackansicht

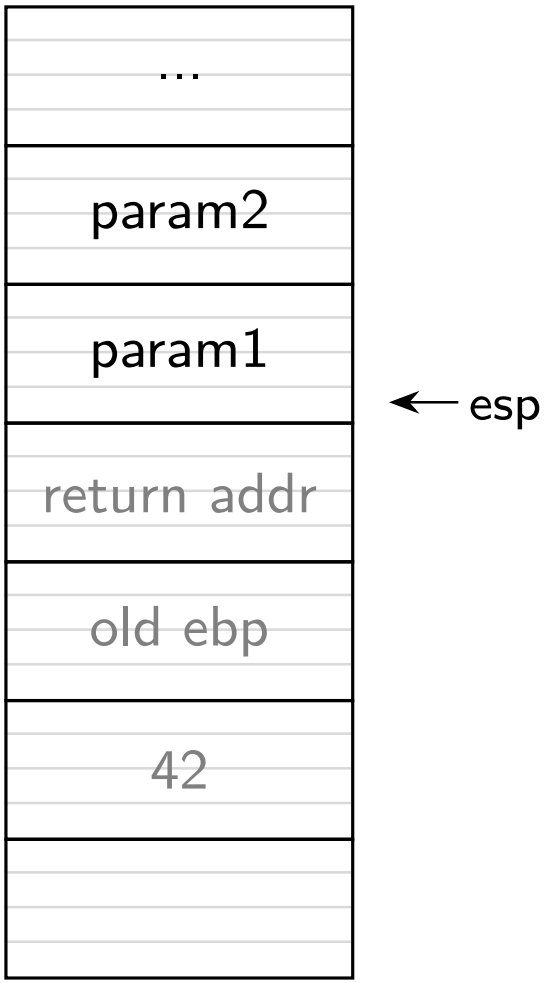


Programm

```
                    push param2
                    push param1
                    call func
return addr:

            func:   push ebp
                    mov ebp, esp
                    push ebx
                    mov esp, ebp
                    pop ebp
```

Funktionen mit Parametern - Stackansicht



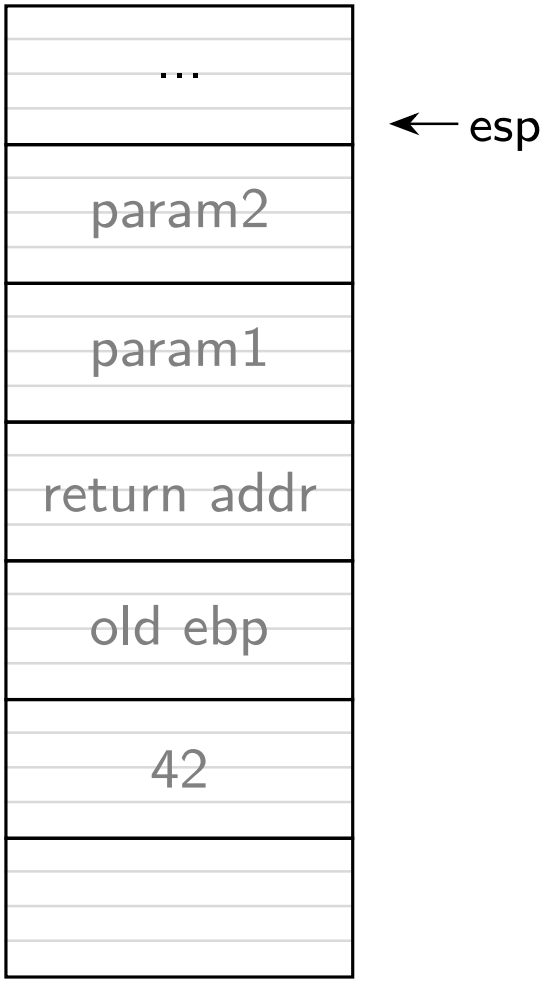
Programm

```
                    push param2
                    push param1
                    call func
return addr:

func:               push ebp
                    mov ebp, esp
                    push ebx
                    mov esp, ebp
                    pop ebp
                    ret
```

Funktionen mit Parametern - Stackansicht

Stack



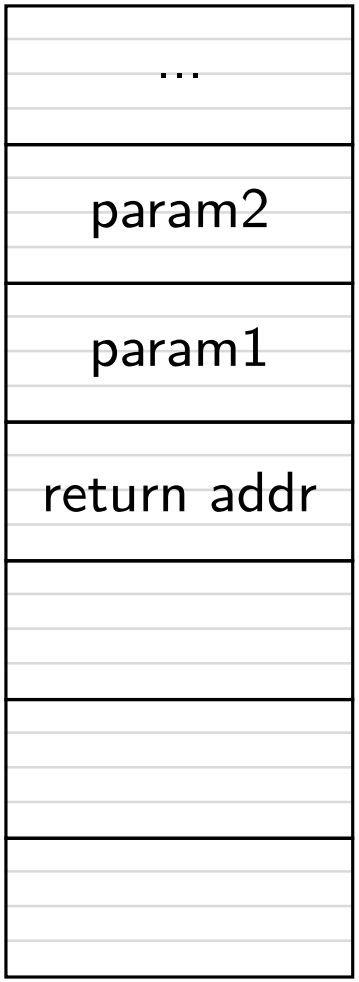
Programm

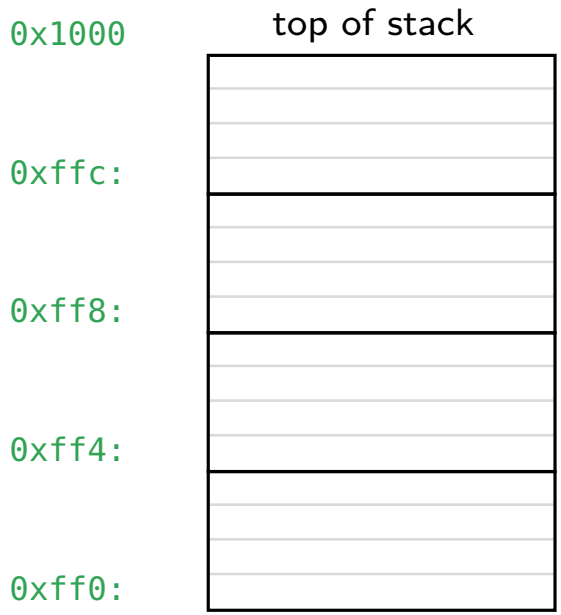
```
                    push param2
                    push param1
                    call func
return addr:        add esp, 8

func:               push ebp
                    mov ebp, esp
                    push ebx
                    mov esp, ebp
                    pop ebp
                    ret
```

Stack

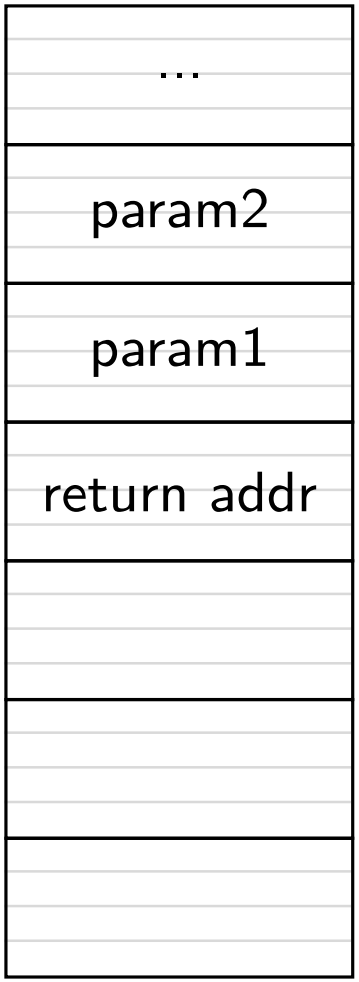
```
void kickoff(Thread* thr);
```

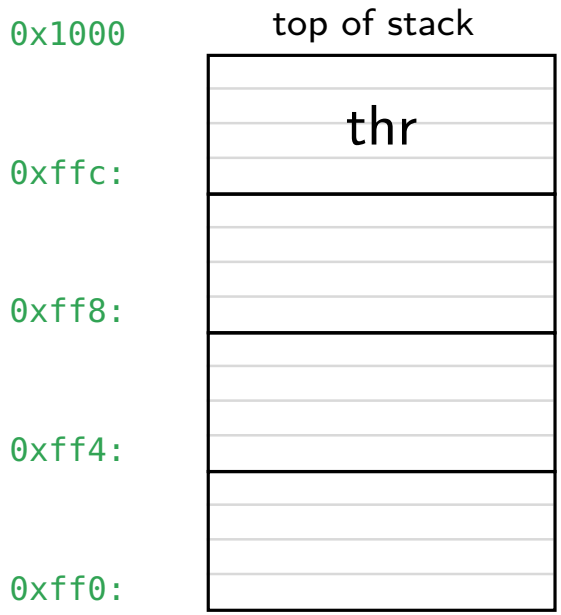




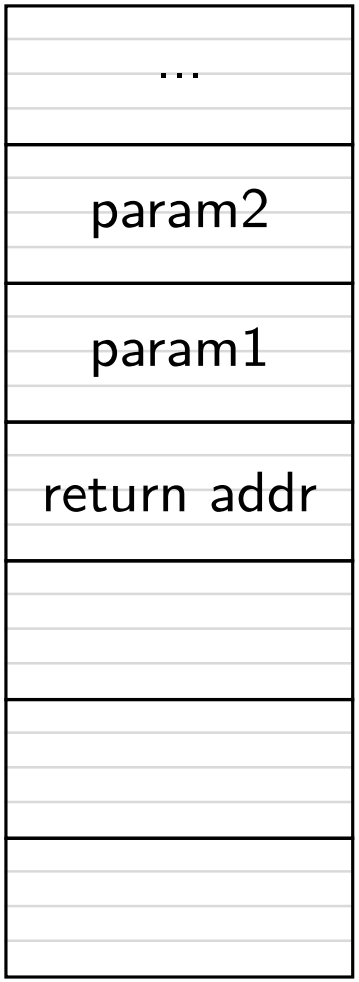
Stack

```
void kickoff(Thread* thr);
```

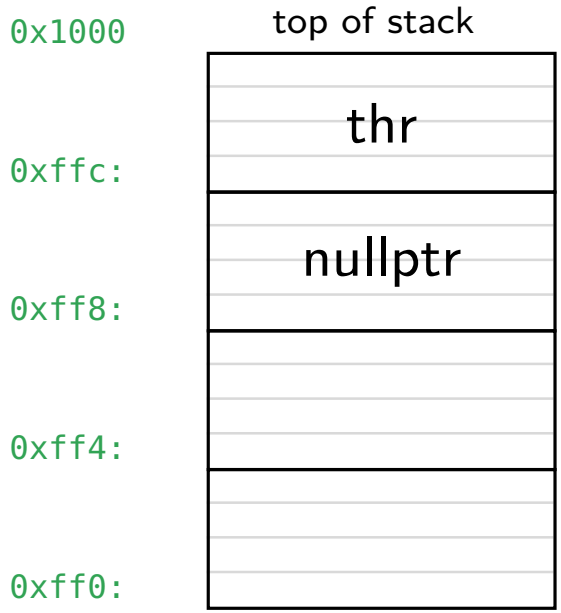




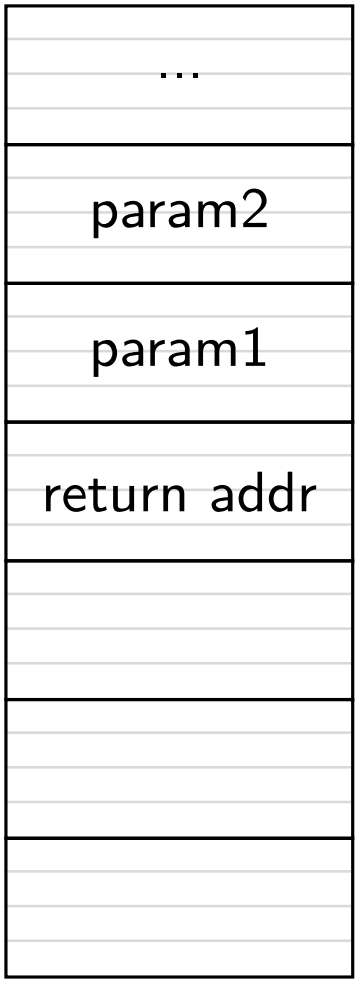
Stack



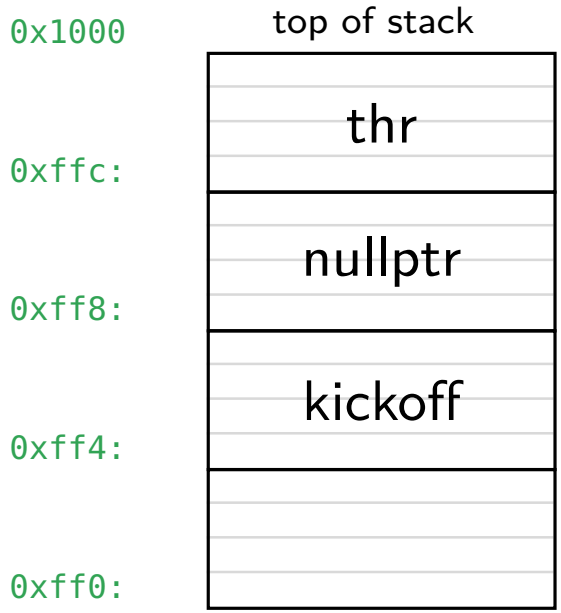
- Erwartetes Stack-Layout



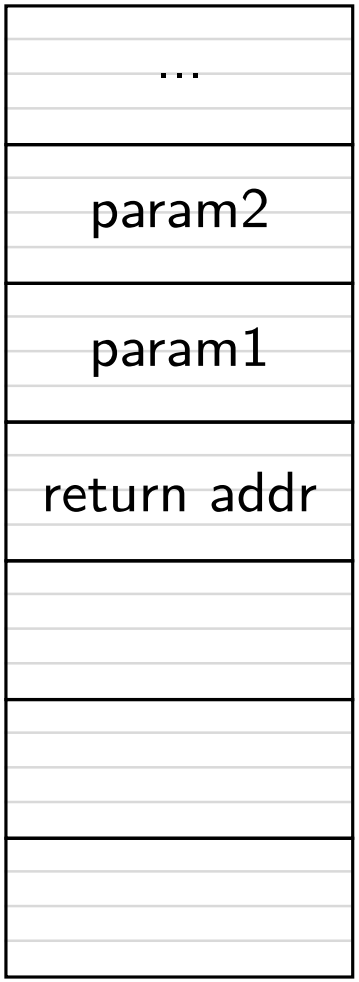
Stack



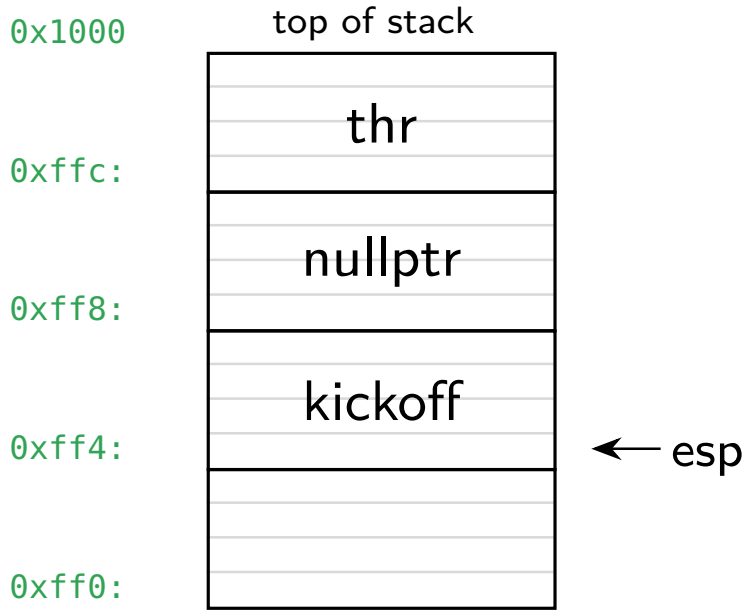
- Erwartetes Stack-Layout



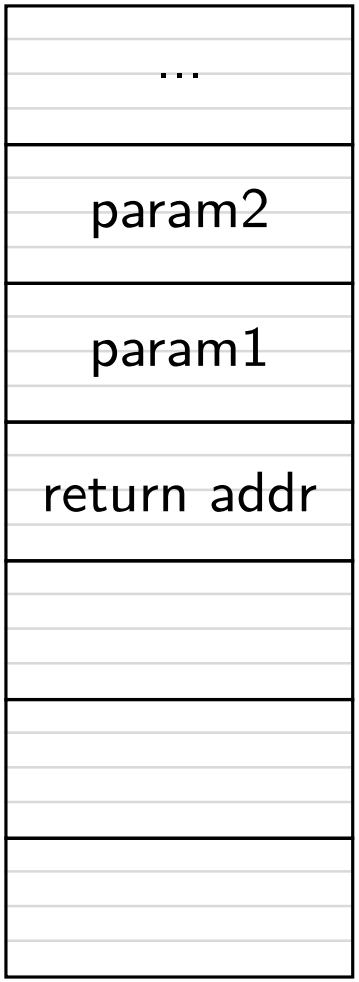
Stack



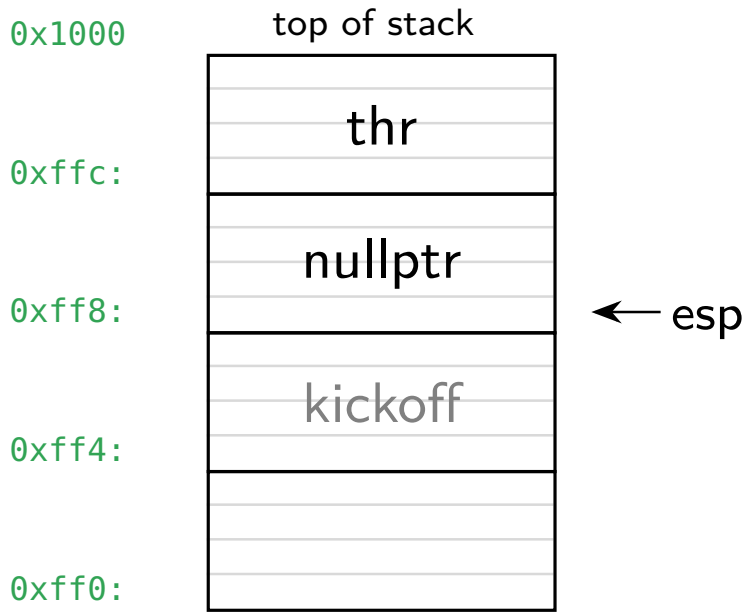
- Aufruf über `ret` statt `call`



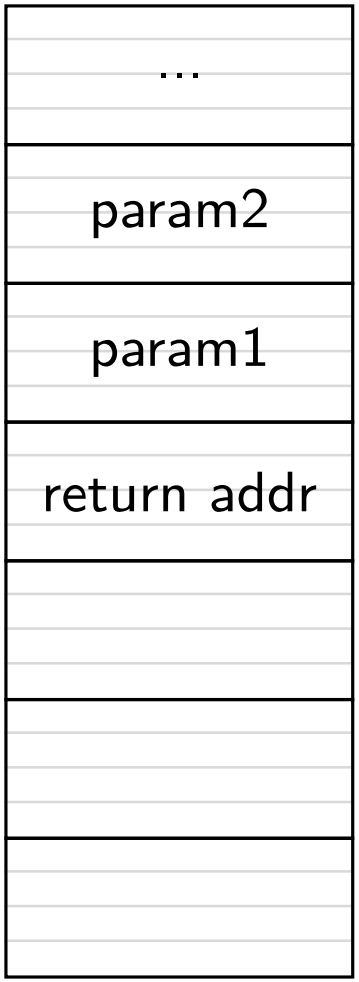
Stack



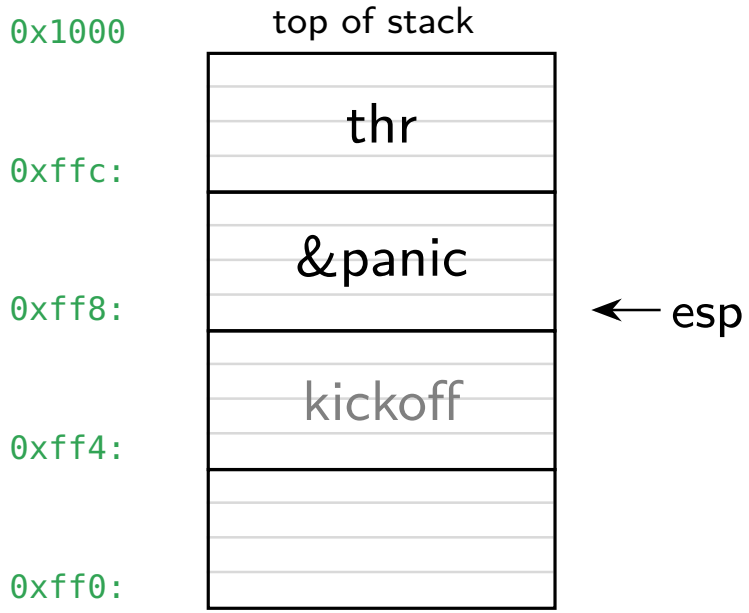
- Jetzt sind wir in `kickoff`



Stack



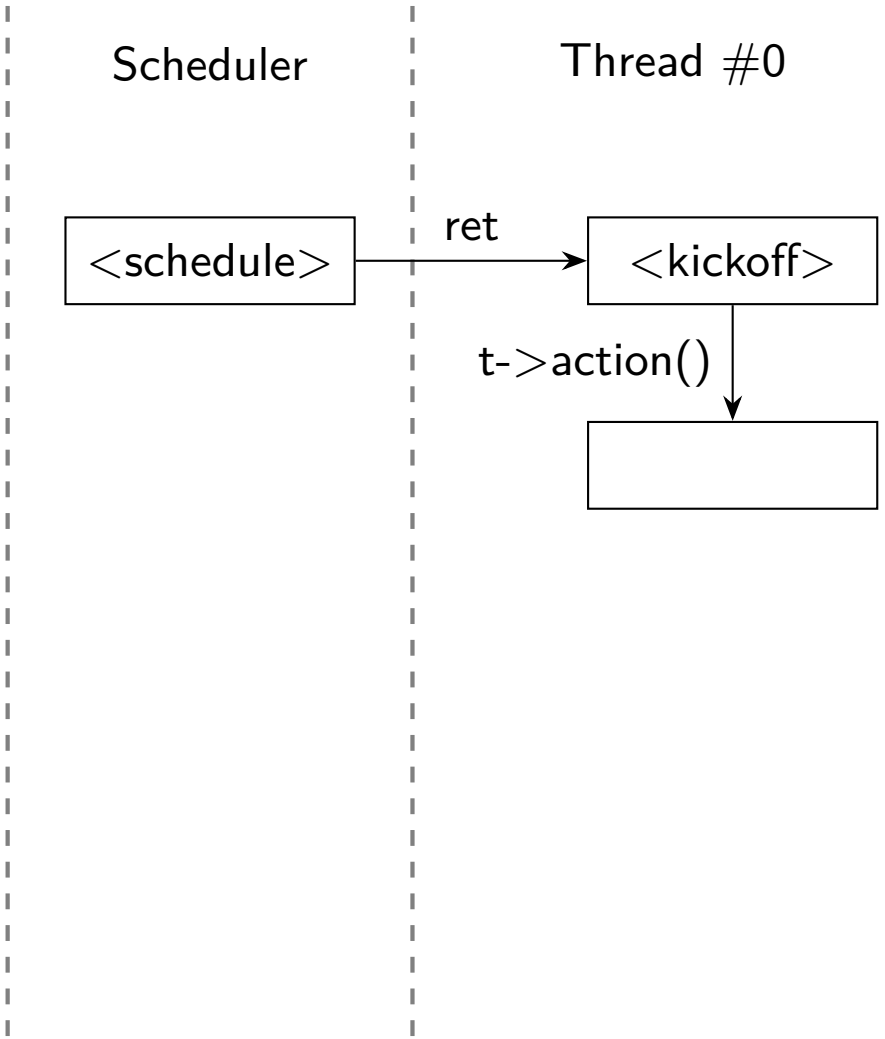
- Kein return von `kickoff`



Kontrollfluss beim Threadwechsel

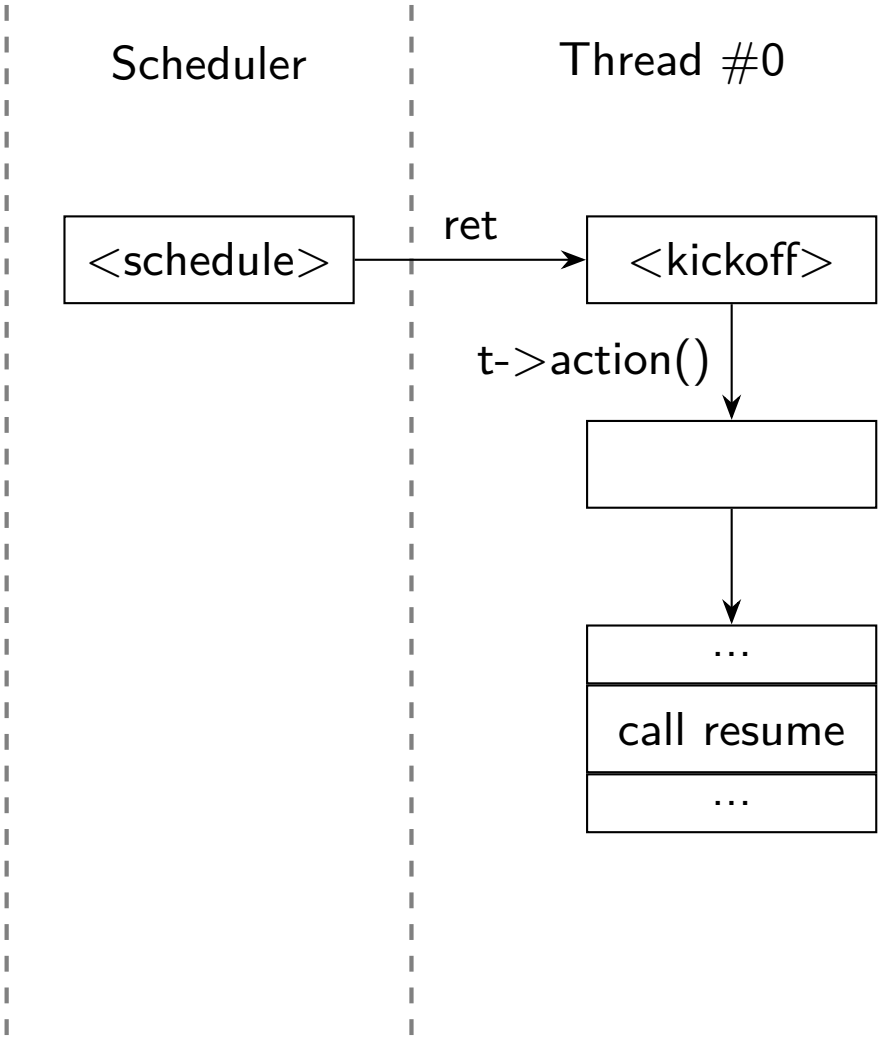
- Start des Schedulers

Kontrollfluss beim Threadwechsel



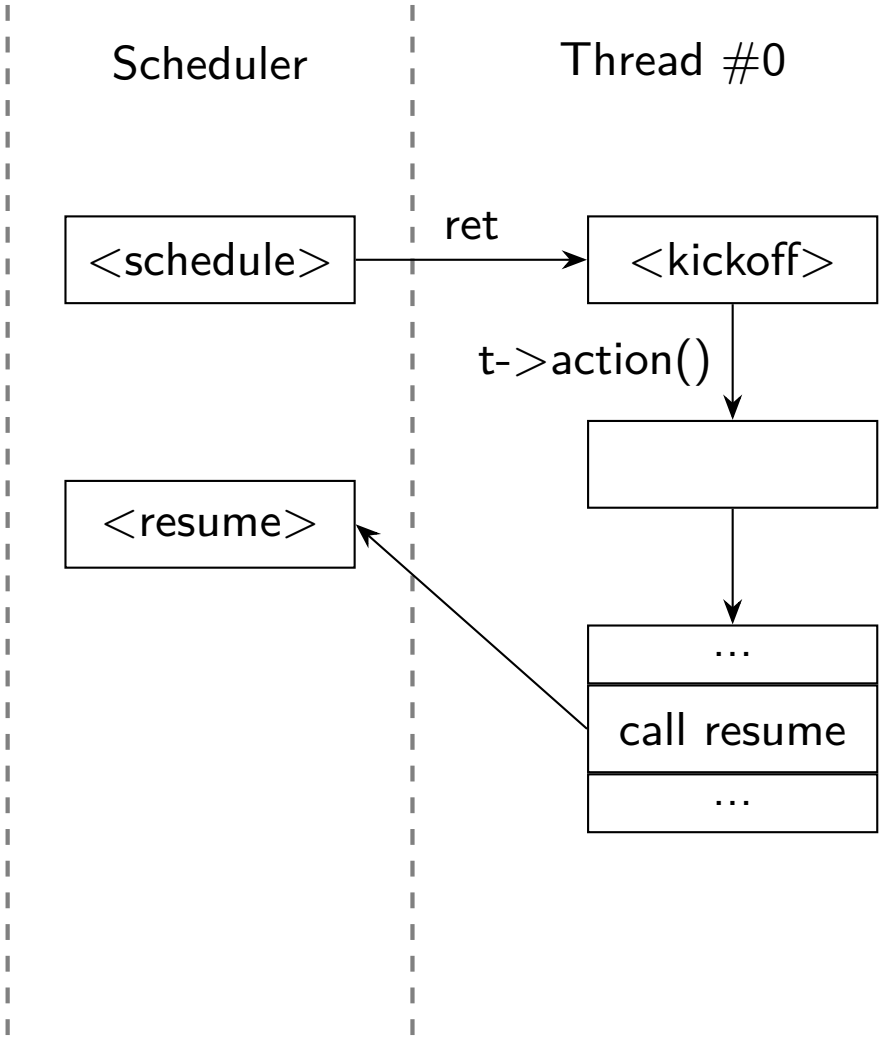
- Erster Start eines Threads

Kontrollfluss beim Threadwechsel

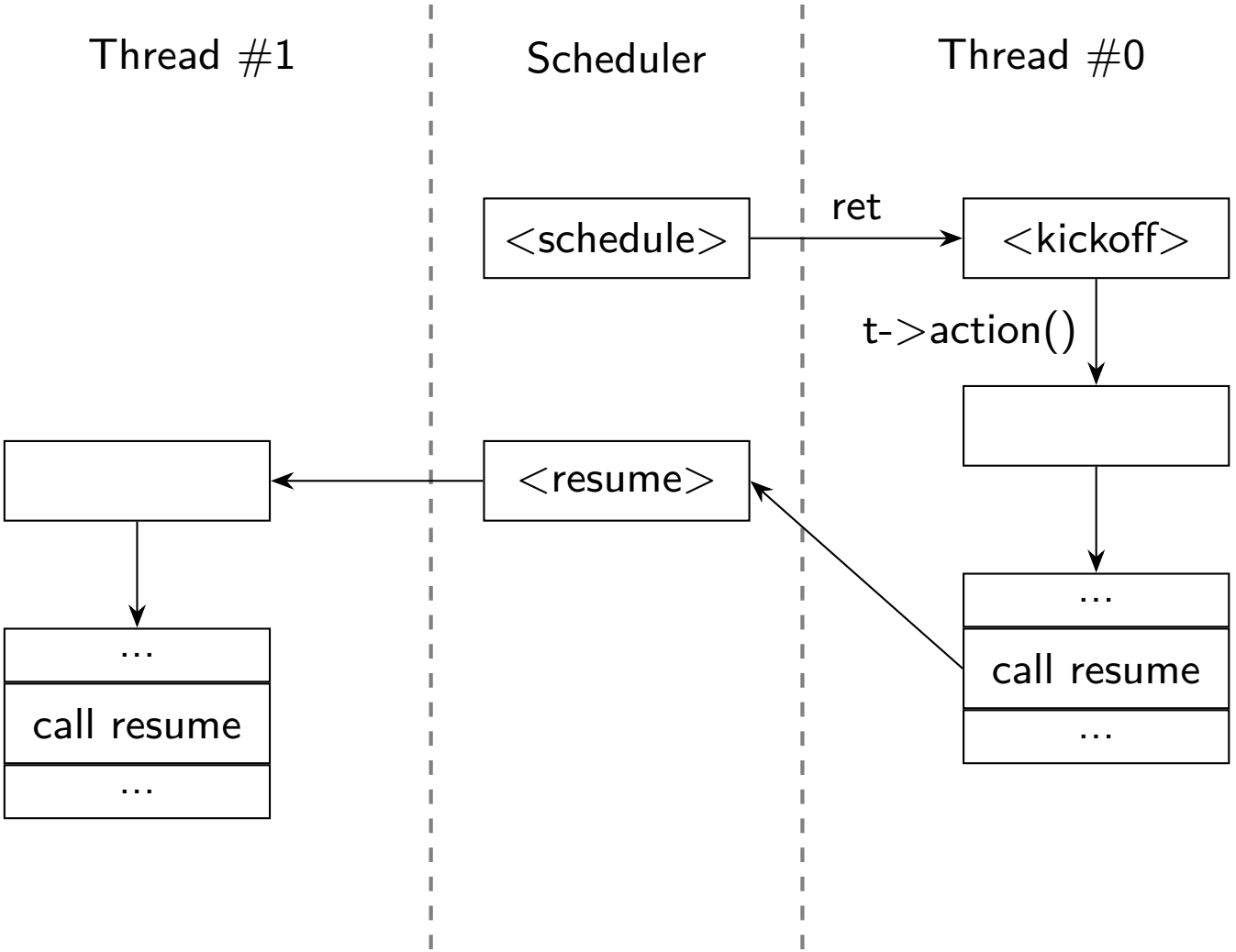


- Thread fordert selbst Scheduler an

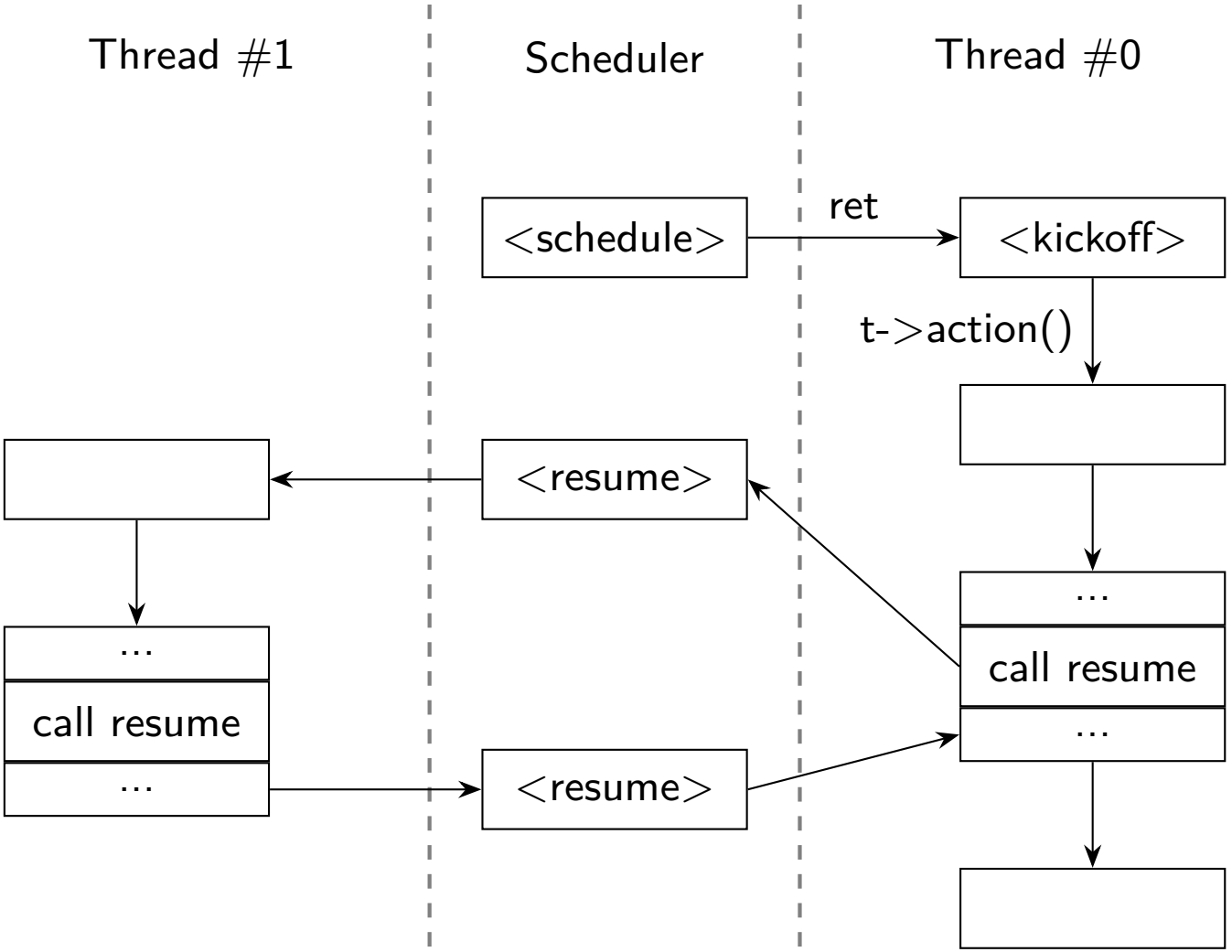
Kontrollfluss beim Threadwechsel



Kontrollfluss beim Threadwechsel

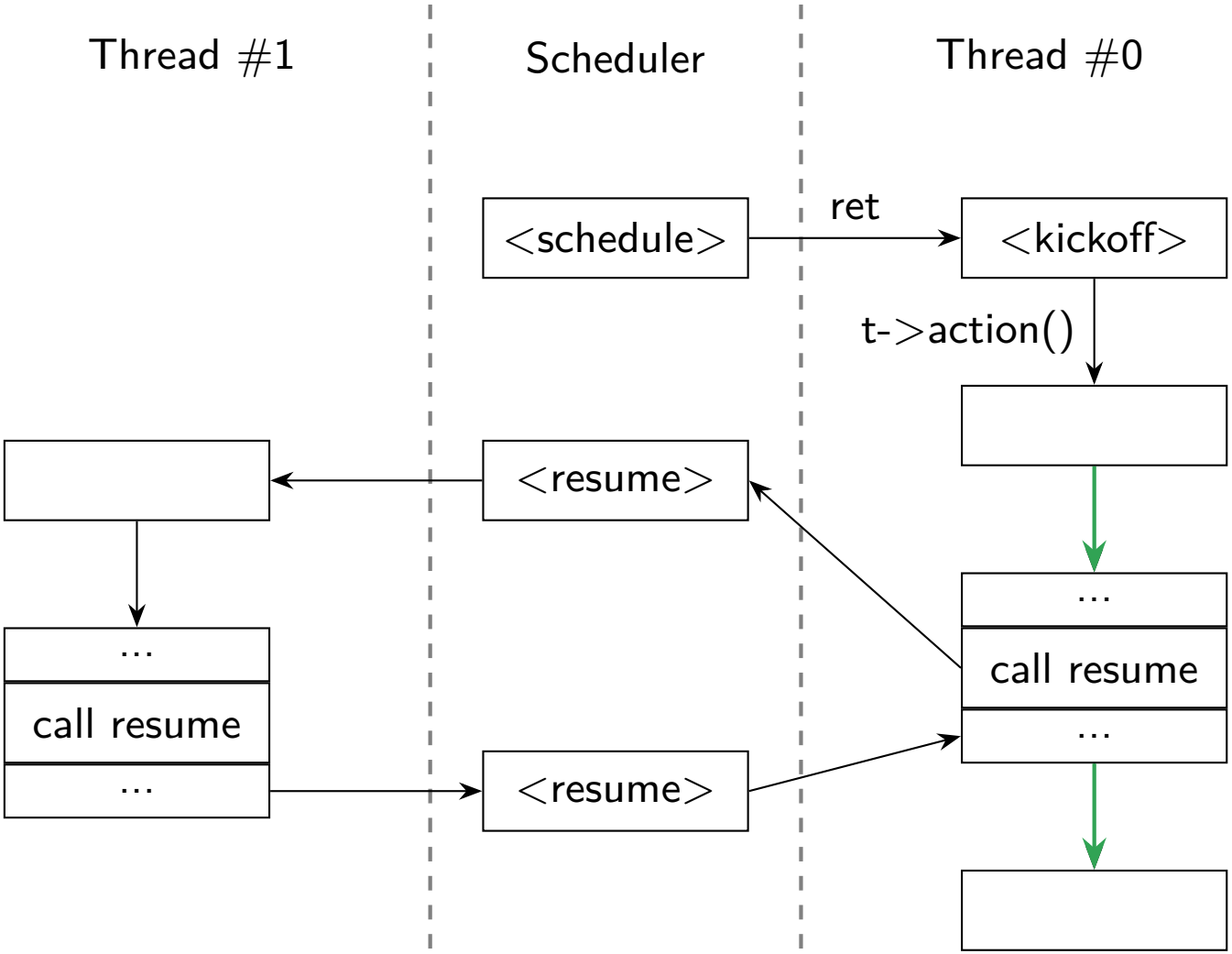


Kontrollfluss beim Threadwechsel



- Fortsetzung des ersten Threads

Kontrollfluss beim Threadwechsel



- Grün: Kontrollfluss aus Perspektive des Threads

Implementierung

Thread

TUHH
Technische Universität Hamburg

- Enthält Kontext
- `prepareContext`: Initialisiert Stack und Register
  (für Arithmetik `void*` auf `void**` casten)
- `kickoff`: Hilfsfunktion für ersten Start von Thread
- `go`: Startet ersten Thread auf CPU
- `resume`: Führt Kontextwechsel durch
- `action`: Ausführungscode von Thread

Für Thread-Liste im Scheduler

Enthält `queue_link[1]`

Dispatcher

TUHH
Technische Universität Hamburg

- Wrapper um Funktionen aus `Thread`
- Führt Scheduler-Entscheidungen aus
- life-Pointer: Pointer auf aktiven Thread (pro CPU)
- `go`: Nur für ersten zu startenden Thread
- `dispatch`: Führt Kontextwechsel durch

Scheduler

TUHH
Technische Universität Hamburg

- Entscheidet über nachfolgende Threads
- Verwaltet ready queue (geschützt auf $E\frac{1}{2}$)
- Entkopplung vom Dispatcher

- `schedule`: Startet scheduling (ein mal pro CPU)
- `ready`: Hängt Thread an ready queue
- `exit`: Erlaubt Thread sich selbst zu beenden
- `kill`: Entfernt Thread aus queue bzw. markiert als *dying*
- `resume`: Dispatched nächsten Thread aus der Queue

Testanwendungen

TUHH
Technische Universität Hamburg

- Mehrere Threads erstellen, abwechselnd laufen lassen
- Ein 4096B Stack pro Thread (globale Variable)
- Klasse `Application` von `Thread` erben lassen
- **Achtung!** Top-of-Stack übergeben

- Am besten nach jedem Schritt testen
  1. `context_launch`, `context_switch`
  2. `Dispatcher::go`, `Dispatcher::dispatch`
  3. `Scheduler::schedule`, `Scheduler::resume` (**Scheduling auf E$\frac{1}{2}$!**)