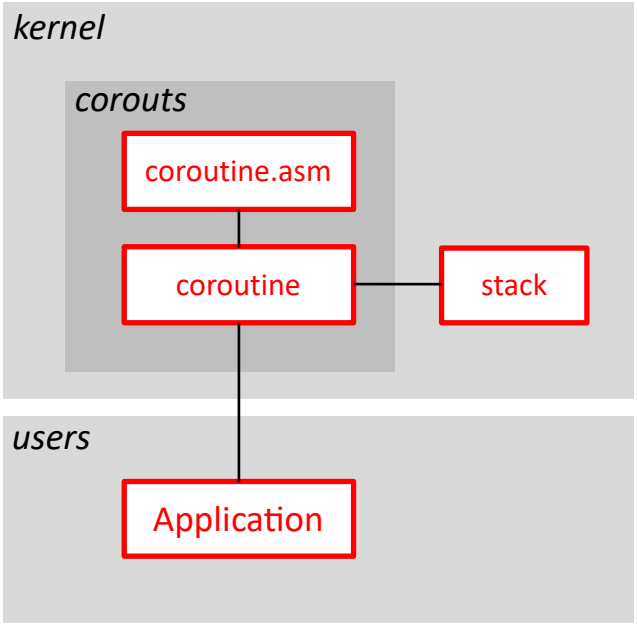
# Betriebssystem-Entwicklung

## Implementierung von Koroutinen in Rust

Michael Schöttner

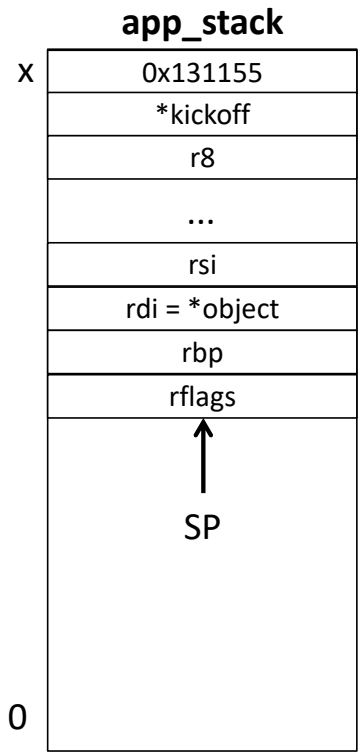
# Überblick der relevanten Dateien





# coroutine_prepare_stack

- Hier wird der Stack für den ersten Aufruf vorbereitet
  - Es werden alle Register gesichert
  - `*kickoff` dient als Rücksprungadresse und als Einstieg in die Koroutine
  - `kickoff` ist in `coroutine.rs` implementiert und erwartet als Parameter einen Zeiger auf das Koroutinen-Objekt, das ist hier `*object`
  - `0x13155` ist nur ein Dummy-Rücksprungadresse die nie verwendet wird
- `sp` wird in `Coroutine::context` gesichert



# Starten der ersten Koroutine

Heinrich Heine
Universität Düsseldorf

- Aufruf der Funktion `start` des `Coroutine`-Objektes

```
extern "C" fn _coroutine_start  (stack_ptr: usize);

pub fn start (c: *mut Coroutine) {
    unsafe {
        _coroutine_start((*cor).stack_ptr);
    }
}
```

- Hier wird dann `_couroutine_start` gerufen, eine Assembler-Routine
  - Diese schaltet auf den präparierten Stack um
  - Lädt die Prozessorregister mit den auf dem Stack gesicherten Inhalten
  - Macht dann einen Rücksprung der bei `kickoff` landet
  - Der Parameter `*object` muss für `kickoff` im Register `rdi` stehen (1. Parameter); das passt bereits durch den präparierten Stack

# Koroutinen-Wechsel



- Wird durchgeführt durch Aufrufen von `coroutine::switch2next`
  - Hiermit kann die aktive Koroutine einen Wechsel auslösen (auf die Nächste in der Kette)
  - Jedes Koroutinen-Objekt speichert `next`
- Das eigentliche Umschalten erfolgt in der Assembler-Funktion `_coroutine_switch`

```
extern "C" {
    fn _coroutine_switch (now_stack_ptr: *mut usize,
                          then_stack: usize);
}

pub fn switch2next (now: *mut Coroutine) {
    unsafe {
        _coroutine_switch( &mut (*now).stack_ptr,
                           (*(*now).next).stack_ptr);id,
                         );
    }
}
```

# Koroutinen-Wechsel (2)

Heinrich Heine
Universität Düsseldorf

- `_coroutine_switch` ist eine Assembler-Routine:
  - Sichert die Registerinhalte des Aufrufers auf dessen Stack und speichert dann die Adresse des zuletzt belegten Stackeintrages in `now_stack_ptr`
  - Anschließend wird der Stack umgeschaltet auf `then_stack`
  - Nun werden die Register geladen, mit den Inhalten die auf dem Stack gespeichert sind
  - Am Ende erfolgt ein Rücksprung mit `ret`, wodurch die neue Koroutine fortgesetzt wird

- Wird das erste Mal auf eine Koroutine umgeschaltet, die nicht mit `start` aktiviert wurde, sondern durch `switch2next`, so funktioniert das `ret` hier genauso wie bei `_coroutine_start` und man landet in `kickoff`

- Ansonsten landet der `ret` in `switch2next` und von dort aus geht es zurück zu der Stelle wo die Koroutine freiwillig die CPU abgegeben hat

# Koroutinen – Übersicht 1/2

Heinrich Heine
Universität Düsseldorf

**corouts_demo.rs**

```
pub fn run() {

}
```

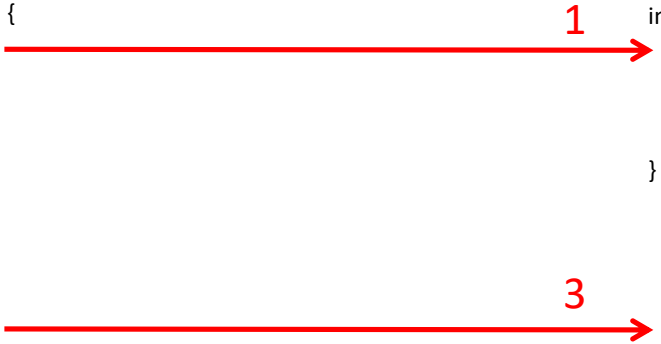
**coroutine.rs**

```
// externe Funktionen in coroutine.asm
extern "C" {
   fn _coroutine_start(stack_ptr: usize);
   fn _coroutine_switch(now_stack_ptr: *mut usize, then_stack: usize);
}

impl Coroutine {
   pub fn new(my_cid: usize, my_entry: extern "C" fn(*mut Coroutine)) -> Box<Coroutine> {
      ...
      corout.coroutine_prepare_stack();
      corout
   }

   fn coroutine_prepare_stack (&mut self) {
   }

   pub fn start (cor: *mut Coroutine) {
      unsafe {
         _coroutine_start((*cor).stack_ptr);     }
   }
```



# Koroutinen – Übersicht 2/2

Heinrich Heine
Universität Düsseldorf

**coroutine_demo.rs**

```
extern "C" fn coroutine_loop_entry(myself: *mut coroutine::Coroutine) {
    loop {

        switch2next ();
    }
}
```


**coroutine.rs**

```
impl Coroutine {

    pub fn switch2next (now: *mut Coroutine) {

    }


}
```