hhu

hhu
Heinrich Heine
Universität Düsseldorf

Betriebssystem-Entwicklung

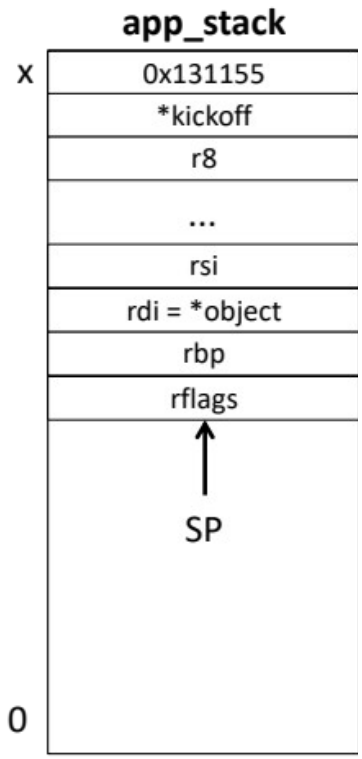
Implementierung von Koroutinen in Rust

Michael Schöttner

Coroutine::prepare_stack

Heinrich Heine
Universität Düsseldorf

- 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 `coroutine` (`object` im Bild)
  - `0x13155` ist nur ein Dummy-Rücksprungadresse die nie verwendet wird
- `stack_ptr` (`SP` im Bild) wird in `Coroutine` gesichert



Starten der ersten Koroutine

Heinrich Heine
Universität Düsseldorf

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

```
unsafe extern "C" fn coroutine_start (stack_ptr: usize) {
    naked_asm!(...)
}

pub fn start (&mut self) {
   unsafe {
      coroutine_start(self.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 `coroutine` muss für `kickoff` im Register `rdi` stehen (1. Parameter); das passt bereits durch den präparierten Stack

Koroutinen-Wechsel

Heinrich Heine
Universität Düsseldorf

- Wird durchgeführt durch Aufrufen von `Coroutine::switch`
  - 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`

```
unsafe extern "C" fn coroutine_switch(
    current_stack_ptr: *mut usize, next_stack: usize
) {
    naked_asm!(...)
}

pub fn switch(&mut self) {
   unsafe {
      coroutine_switch(&mut self.stack_ptr, (*self.next).stack_ptr);
   }
}
```

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 `current_stack_ptr`
  - Anschließend wird der Stack umgeschaltet auf `next_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 `switch`, so funktioniert das `ret` hier genauso wie bei `coroutine_start` und man landet in `kickoff`

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