

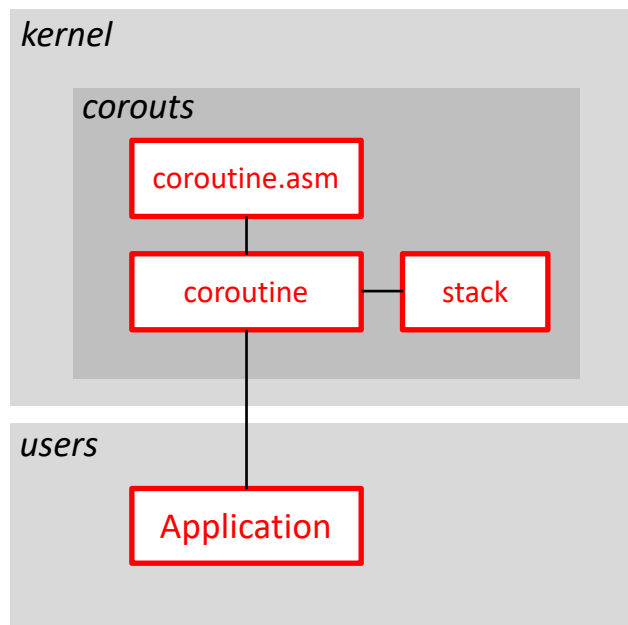


Betriebssystem- Entwicklung

Implementierung von Koroutinen in Rust

Michael Schöttner

Überblick der relevanten Dateien



- Anwendungen nutzen Koroutinen indem sie den Trait `CoroutineEntry` implementieren

```
pub trait CoroutineEntry {  
    // Einstiegsfunktion der Koroutine  
    fn run(&mut self, object: *mut Coroutine);  
}
```

- Bevor eine Anwendung die erste Koroutine startet müssen diese vorbereitet werden
 - Für jede Koroutine muss ein Objekt mit `Box::new` angelegt werden
 - Der Stack wird in der Funktion `Coroutine::new` automatisch angelegt
 - Anschließend müssen die Koroutinen-Objekte mithilfe `Coroutine::set_next` zyklisch miteinander verkettet werden
 - Zum Schluss kann die erste Koroutine mit `Coroutine::start` angestossen werden

- Beispiel: Einstiegsfunktion `run` einer Koroutine
- `corouts_demo.rs` (Auszug)

```
struct MyCoroutine {  
}  
  
impl coroutine::CoroutineEntry for MyCoroutine {  
  
    fn run(&mut self, object: *mut coroutine::Coroutine) {  
        // Code der Coroutine  
    }  
  
}
```

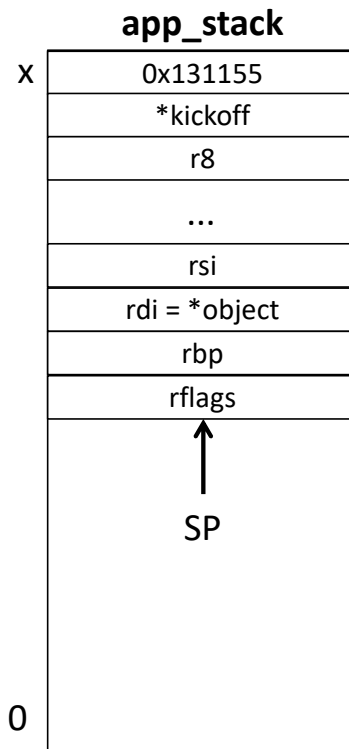
■ Anlegen der Koroutinen-Objekte und Starten der ersten Koroutine

```
pub fn init_demo() {  
    let c1 = Box::new( MyCoroutine {} );  
    let mut corout1 = coroutine::Coroutine::new(1, c1);  
  
    let c2 = Box::new( MyCoroutine {} );  
    let mut corout2 = coroutine::Coroutine::new(2, c2);  
  
    corout1.set_next( corout2.get_raw_pointer() );  
    corout2.set_next( corout1.get_raw_pointer() );  
  
    coroutine::Coroutine::start( corout1.get_raw_pointer() );  
}
```

- Beim Anlegen des Koroutinen-Objektes wird `new` aufgerufen
 - Hier wird der Stack angelegt; das Koroutinen-Objekt
 - Anschließend wird in `coroutine_prepare_stack` der Stack für den ersten Aufruf präpariert

```
impl Coroutine {  
  
    pub fn new(mycid: u64, myentry: Box<dyn CoroutineEntry>)-> Box<Coroutine> {  
        let mystack = stack::Stack::new(4096);  
  
        let mut corout = Box::new( Coroutine{ cid: mycid,  
                                              context: 0,  
                                              stack: mystack,  
                                              entry: myentry,  
                                              next: ptr::null_mut(),  
                                              } );  
  
        corout.coroutine_prepare_stack();  
        corout  
    }  
}
```

- 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`
 - `0x131155` ist nur ein Dummy-Rücksprungadresse die nie verwendet wird
- SP wird in `Coroutine::context` gesichert



■ Aufruf der Funktion `start` des Coroutine-Objektes

```
extern "C" fn _coroutine_start (now_coroutine_struct: *mut c_void);

pub fn start (c: *mut Coroutine) {
    unsafe {
        _coroutine_start(c as *mut c_void);
    }
}
```

■ 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

- 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_coroutine_struct: *mut c_void,  
                          then_coroutine_struct: *mut c_void);  
}  
  
pub fn switch2next (now: *mut Coroutine) {  
    unsafe {  
        _coroutine_switch( now as *mut c_void,  
                           ((*now).next) as *mut c_void,  
                           );  
    }  
}
```

- `_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.context`
 - Anschließend wird der Stack umgeschaltet auf `then.context`
 - Nun werden die Register geladen, mit den Inhalten die auf dem Stack gespeichert sind
 - Am Ende erfolgt ein Rücksprung mit `ret`, wodurch die Koroutine `then` fortgesetzt wird
- Wird das erste Mal auf eine Koroutine umgeschaltet, die nicht mit `start` aktiviert wurde, 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

startup.rs

```
fn aufgabe4() {  
    let c1 = Box::new( MyCoroutine {} );  
    let mut corout1 = coroutine::Coroutine::new(1, c1);  
  
    let c2 = Box::new( MyCoroutine {} );  
    let mut corout2 = coroutine::Coroutine::new(2, c2);  
  
    corout1.set_next( corout2.get_raw_pointer() );  
    corout2.set_next( corout1.get_raw_pointer() );  
  
    coroutine::Coroutine::start( corout1.get_raw_pointer() );  
}
```

coroutine.rs

```
// externe Funktionen in coroutine.asm  
extern "C" {  
    fn _coroutine_start (now_coroutine_struct: *mut c_void);  
    fn _coroutine_switch (now_coroutine_struct: *mut c_void,  
                          then_coroutine_struct: *mut c_void);  
}  
  
impl Coroutine {  
    pub fn new(mycid: u64, myentry: Box<dyn CoroutineEntry>)-> Box<Coroutine> {  
        ...  
        corout.coroutine_prepare_stack();  
        corout  
    }  
  
    fn coroutine_prepare_stack (&mut self) {  
    }  
}
```

1



2



startup.rs

```
fn aufgabe4() {  
    ....  
    coroutine::Coroutine::start( corout1.get_raw_pointer() );  
}
```

coroutine_demo.rs

```
struct MyCoroutine {  
}  
  
impl coroutine::CoroutineEntry for MyCoroutine {  
    fn run(&mut self, object: *mut coroutine::Coroutine) {  
        // Code der Coroutine  
        switch2next ();  
    }  
}
```

coroutine.rs

```
impl Coroutine {  
    pub fn start (c: *mut Coroutine) {  
        unsafe {  
            _coroutine_start(c as *mut c_void);  
        }  
    }  
  
    pub fn switch2next (now: *mut Coroutine) {  
        unsafe {  
            _coroutine_switch( now as *mut c_void,  
                               ((*now).next) as *mut c_void,  
                               );  
        }  
    }  
}
```

3

4