

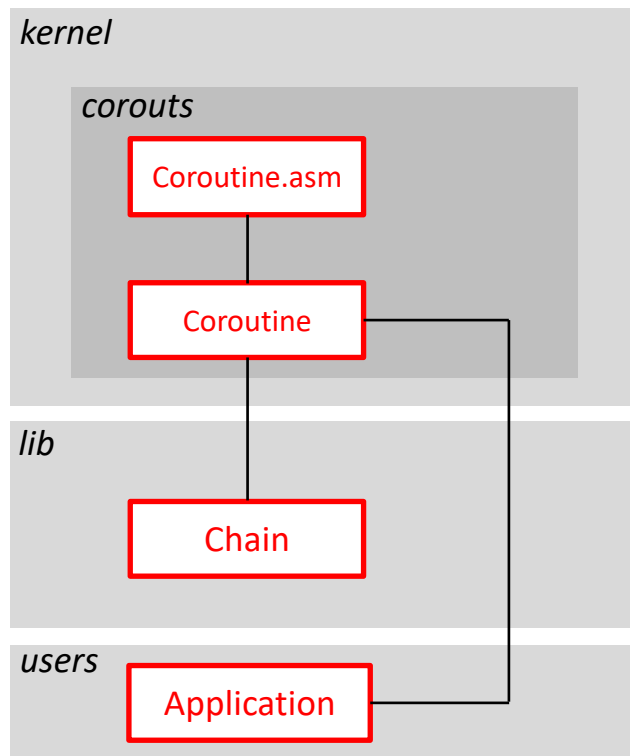


Betriebssystem- Entwicklung

Implementierung von Koroutinen in C++

Michael Schöttner

Überblick der relevanten Dateien



- Anwendungen nutzen Koroutinen indem sie die Klasse `Coroutine` erweitern
- Hierdurch werden mehrere Methoden vererbt
 - `setNext` dient der Verkettung der Koroutinen (siehe unten)
 - `start` stösst die erste Koroutine an
 - `switch2next` schaltet zur nächsten Koroutine um
- Bevor eine Anwendung die erste Koroutine startet müssen diese vorbereitet werden
 - Für jede Koroutine muss ein Objekt angelegt werden und ein Stack an den Konstruktor übergeben werden
 - Der Stack kann dynamisch über die eigene Heap-Verwaltung angelegt werden (empfohlen)
 - Oder als globale Variable (ein Array vom Datentyp `uint64_t`)
 - Anschließend müssen die Koroutinen-Objekte zyklisch miteinander verkettet werden
 - Zum Schluss kann die erste Koroutine angestossen werden

Koroutine in einer Anwendung nutzen

- Anwendungs-Klasse schreiben, die `Coroutine` erweitert
- Beispiel: `Application.h` (Auszug)

```
class Application : public Coroutine {  
  
public:  
    // Initialisieren der Koroutine  
    Application (uint64_t *stack) : Coroutine (stack);  
  
    void run ();          // Start-Methode (wird indirekt gerufen)  
};
```

- Beispiel: `Application.cc` (Auszug)

```
void Application::run () {  
    // Arbeit erledigen  
    switch2next (); // CPU abgeben (auf naechste Koroutine umschalten)  
}
```

■ Instanzieren der Anwendungs-Klasse in `main.cc`

```
// Stack fuer die Koroutine
unsigned int app1_stack[1024];
unsigned int app2_stack[1024];

int main() {
    // App instanzieren; Koroutine wird dadurch initialisiert
    // Wir übergeben den Zeiger auf das Ende des Arrays.
    // Der Stack wächst von hohen zu niedrigen Adressen!
    Application app1 ( &app1_stack[1024] );
    Application app2 ( &app2_stack[1024] );

    app1.setNext(&app2);
    app2.setNext(&app1);

    // 1. Koroutine starten
    app1.start ();
}
```

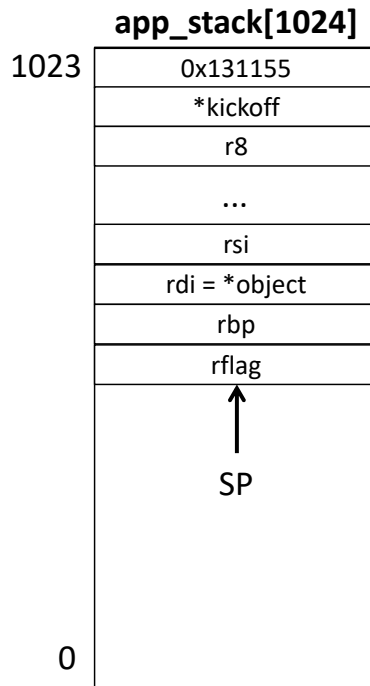
- Der Konstruktor von `Application` ruft aufgrund der Klassenvererbung den Konstruktor der Klasse `Coroutine`

```
Coroutine::Coroutine (uint64_t *stack) {  
    Coroutine_init (&context, stack, kickoff, this);  
}
```

- `Coroutine_init` ist eine C-Funktion (in `Couroutine.cc`), welche den Stack für das erste Umschalten auf die Coroutine präpariert (siehe nächste Seite)
- `Context` ist eine Instanzvariable und speichert den zuletzt genutzten Stackeintrag
- `kickoff` wird nachher beschrieben

```
void Coroutine_init ( uint64_t *context, uint64_t *stack,  
                     void (*kickoff) (Coroutine*),  
                     void *object  
                     );
```

- 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.cc` 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 `context` gesichert



- Aufruf der Methode `start` der App-Instanz, die Coroutine erweitert

```
extern "C" void Coroutine_start (struct CoroutineState* context);  
  
void Coroutine::start () {  
    Coroutine_start(&context);  
}
```

- Hier wird dann `Coroutine_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` für `kickoff` muss im Register `rdi` stehen (1. Parameter); das passt 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)
 - `Coroutine` erweitert `Chain` und erbt damit einen `next`-Zeiger
- Das eigentliche Umschalten erfolgt in der Assembler-Funktion `Coroutine_switch`

```
extern "C"
{
    void Coroutine_switch (void *context_now, void *context_then);
}

void Coroutine::switch2next () {
    Coroutine_switch (&this->context, &(((Coroutine*)next)->context));
}
```

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

main.cc

```
// Instanz für einen Anwendungsthread anlegen  
unsigned int app1_stack[1024];  
unsigned int app2_stack[1024];
```

```
int main() {  
    Application app1 ( &app1_stack[1024] );  
    Application app2 ( &app2_stack[1024] );  
  
    app1.setNext(&app2);  
    app2.setNext(&app1);  
  
    app1.start ();  
}
```

Application.cc

```
void Application::run () {  
  
    // Arbeit erledigen  
  
    switch2next ();  
}
```

Coroutine.cc

```
// externe Funktionen in Coroutine.asm
```

```
extern "C" {  
    void Coroutine_start (struct CoroutineState* context);  
    void Coroutine_switch (void *context_now, void *context_then);  
}
```

```
Coroutine::Coroutine (uint64_t *stack) {  
    Coroutine_init (&context, stack, kickoff, this);  
}
```

```
void Coroutine_init ( uint64_t *context, uint64_t *stack,  
    void (*kickoff)(Coroutine*), void *object );
```

```
void Coroutine::start () {  
    Coroutine_start(&context);  
}
```

```
void Coroutine::switch2next () {  
    Coroutine_switch (&this->context, &(((Coroutine*)next)->context));  
}
```

1

2

3

4