



Betriebssysteme und nebenläufige Programmierung

SoSe 2026

Roland Wismüller
Betriebssysteme / verteilte Systeme
roland.wismueller@uni-siegen.de
Tel.: 0271/740-4050, Büro: H-B 8404

Stand: 20. März 2026

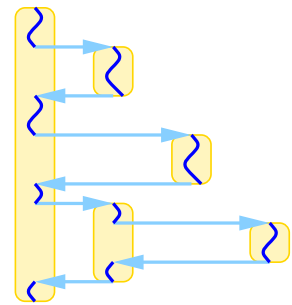


Betriebssysteme und nebenläufige Programmierung

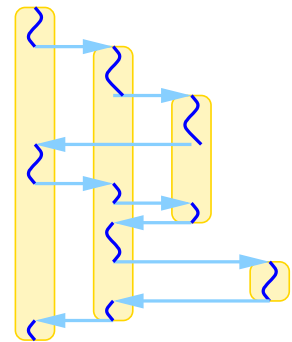
SoSe 2026

6 Koroutinen und asynchrone Programmierung

- ➔ Normaler Prozeduraufruf: asymmetrisch
 - ➔ Aufrufer sichert Rückkehradresse
 - ➔ aufgerufene Prozedur läuft von Anfang bis Ende
 - ➔ Aufrufer macht nach Aufrufstelle weiter



- ➔ **Koroutinen** (Conway 1963): symmetrisch
 - ➔ kein Aufruf, sondern Wechsel in andere Koroutine
 - ➔ diese Koroutine macht ab letzter „Unterbrechungsstelle“ weiter
 - ➔ statt Rückkehr wieder Wechsel in beliebige andere Koroutine



Koroutinen vs. Threads

- ➔ Gemeinsamkeiten:
 - ➔ Ausführung wechselt zwischen Koroutinen bzw. Threads hin und her
 - ➔ Fortsetzung erfolgt immer an der letzten „Unterbrechungsstelle“
 - ➔ während der Unterbrechung bleibt der Zustand (lokale Variable) gespeichert
- ➔ Wesentlicher Unterschied:
 - ➔ bei Koroutinen erfolgt der Wechsel kooperativ, an genau festgelegten Stellen
 - ➔ der zu speichernde Zustand kann daher minimiert werden
 - ➔ Koroutinen sind verzahnt, aber nicht wirklich nebenläufig



Realisierung von Koroutinen

- ➔ Über Fortsetzungen (*continuations*)
- ➔ Eine Fortsetzung repräsentiert den Rest der Ausführung eines unterbrochenen Kontrollflusses
 - Fortsetzungsadresse (Befehlszähler)
 - lokale Variablen (inkl. Register)
 - der Koroutine selbst, sowie ggf. von aufgerufenen normalen Prozeduren
- ➔ Jede Koroutine benötigt daher ihren eigenen Keller
- ➔ Koroutinen-Wechsel entspricht Wechsel des Kellers und „Rücksprung“
 - Register werden dabei durch den Compiler im Keller gesichert

6.2 Die Protothread-Bibliothek



- ➔ Extrem leichtgewichtige „Thread“-Implementierung im Benutzermodus für die Programmiersprache C
 - in Form von Makros
- ➔ Realisiert Koroutinen ohne Keller
 - d.h. lokale Variable werden beim Koroutinen-Wechsel nicht gesichert
 - im Bedarfsfall `static` Variablen verwenden
 - möglich, solange Koroutinen nicht mehrfach instantiiert werden
 - der Zustand einer Koroutine besteht damit nur aus der Fortsetzungsadresse (Programmzähler)
- ➔ Einsatz in ressourcenbeschränkten Systemen (z.B. eingebetteten Systemen, Sensornetzen, etc.)

Anmerkungen zu Folie 307:

Mehr Informationen, inkl. den Quellcode der Bibliothek, finden Sie auf der Webseite <http://dunkels.com/adam/pt>.

307-1

6.2 Die Protothread-Bibliothek ...



(Animierte Folie)

Beispiel

```
PT_THREAD(thr(struct pt *pt, int no)) {
    PT_BEGIN(pt);
    printf("Thr %d: A\n", no);
    PT_YIELD(pt);
    printf("Thr %d: B\n", no);
    PT_YIELD(pt);
    printf("Thr %d: C\n", no);
    PT_END(pt);
}

int main() {
    struct pt pt1, pt2;
    PT_INIT(&pt1);
    PT_INIT(&pt2);
    while(PT_SCHEDULE(thr(&pt1, 1) & thr(&pt2, 2)));
}
```

Deklaration der Koroutine

In dieser Struktur wird der Zustand der Koroutine gespeichert

Koroutinen-Wechsel (wechselt immer zum Aufrufer der Koroutine, hier **main**)

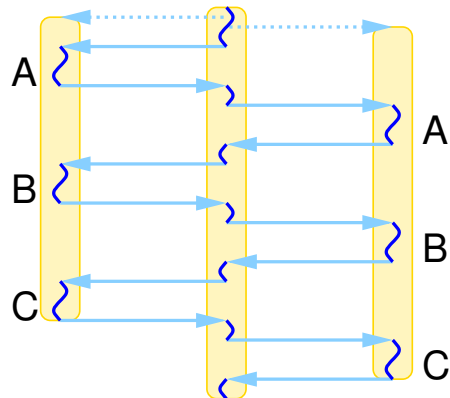
Initialisierung von zwei Koroutinen

Beide Koroutinen abwechselnd vorantreiben

Anmerkungen zu Folie 308:

Ablauf:

thr(1) main thr(2)



308-1

6.2 Die Protothread-Bibliothek ...



(Animierte Folie)

Beispiel nach Expansion der Makros (vereinfacht)

```
bool thr(struct pt *pt, int no) {
```

```
    switch(pt->lc) {  
        case 0:
```

```
            printf("Thr %d: A\n", no);
```

```
            pt->lc = 9;
```

```
            return false;
```

```
        case 9:
```

```
            printf("Thr %d: B\n", no);
```

```
            pt->lc = 11;
```

```
            return false;
```

```
        case 11:
```

```
            printf("Thr %d: C\n", no);
```

```
    }
```

```
    return true;
```

```
}
```

← **PT_BEGIN(pt)**

Switch-Anweisung setzt an der in pt gespeicherten Stelle fort

← **PT_YIELD(pt)**

Sichert Fortsetzungsadresse und kehrt zum Aufrufer zurück

← **PT_END(pt)**

Kehrt (endgültig) zurück



Weiteres Beispiel: Erzeuger/Verbraucher-Problem

```
PT_THREAD(producer(struct pt *pt)) {
    static int i;
    PT_BEGIN(pt);
    for (i=0; i<N; i++) {
        PT_SEM_WAIT(pt, &empty);
        insertItem(produce());
        PT_SEM_SIGNAL(pt, &full);
    }
    PT_END(pt);
}
```

- ➔ Schleifenvariable muß als `static` deklariert werden
 - ➔ die Werte lokaler Variablen werden beim Koroutinenwechsel nicht gesichert
- ➔ Kein wechselseitiger Ausschluß nötig



Weiteres Beispiel: Erzeuger/Verbraucher-Problem ...

```
PT_THREAD(consumer(struct pt *pt)) {
    static int i;
    PT_BEGIN(pt);
    for (i=0; i<N; i++) {
        PT_SEM_WAIT(pt, &full);
        consume(removeItem());
        PT_SEM_SIGNAL(pt, &empty);
    }
    PT_END(pt);
}
```

- ➔ `PT_SEM_WAIT` entspricht `P()` Operation auf einem Semaphor
- ➔ `PT_SEM_SIGNAL` entspricht `V()` Operation auf einem Semaphor
- ➔ `full` und `empty` sind aber lediglich `int`-Variable



Beispiel nach Makro-Expansion

```
bool consumer(struct pt *pt) {
    static int i;
    switch (pt->lc) {
    case 0:
        for (i=0; i<N; i++) {
            pt->lc = 94;
            case 94:
                if (full <= 0)
                    return false;
                full--;
                consume(removeItem());
                empty++;
        }
    }
    ...
}
```

← PT_SEM_WAIT(pt, &full)

Semaphor P-Operation
kehrt zum Aufrufer zurück
statt zu blockieren => Polling

← PT_SEM_SIGNAL(pt, &empty)

Semaphor V-Operation
zählt lediglich Zähler hoch

6.3 Asynchrone Programmierung



- ➔ Viele Programme (insbes. netzwerkbasierte Clients und Server) sind stark E/A-lastig
 - ➔ Programm blockiert die meiste Zeit, um auf E/A zu warten
- ➔ Wunsch: Warten auf E/A soll nebenläufig erfolgen können
 - ➔ z.B. unabhängige E/A Aufträge gleichzeitig ausführen
 - ➔ beschleunigt Programmausführung auch auf einer CPU
- ➔ Beispiel im Folgenden:
 - ➔ Web-Browser muss zwei verschiedene HTML-Dokumente laden, um eine Webseite darzustellen
 - ➔ HTML-Anfrage kann 'Redirect' zurückliefern
 - ➔ dann ist eine neue Anfrage notwendig
 - ➔ d.h., Anfragen können auch voneinander abhängen

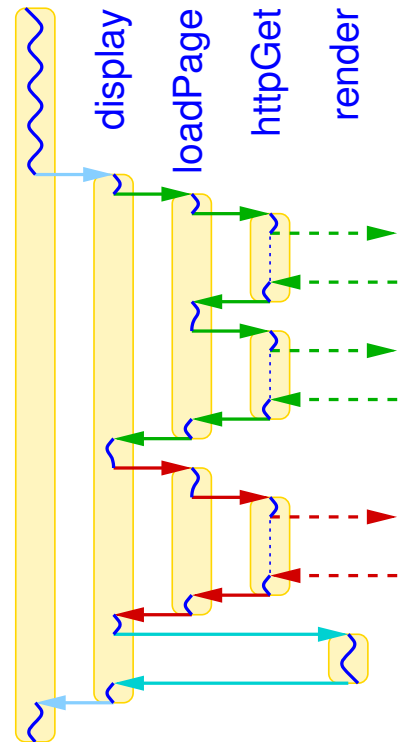
6.3.1 Sequentielle Realisierung



- ➔ Lade beide Seiten nacheinander
- ➔ Im Beispiel soll bei der ersten Seite zunächst ein 'Redirect' kommen

```
void display() {  
    String p1 = loadPage("...1");  
    String p2 = loadPage("...2");  
    render(p1, p2);  
}  
  
String loadPage(String url) {  
    String p = httpGet(url);  
    String target = checkRedirect(p);  
    if (target != null)  
        return httpGet(target);  
    return p;  
}
```

Zeitlicher Ablauf:



Anmerkungen zu Folie 314:

- ➔ Der Übersicht halber werden für den ersten Aufruf von `loadPage()` grüne, für den zweiten Aufruf rote Linien verwendet.
- ➔ Die gestrichelten Linien stellen die HTTP-Anfrage an den Server bzw. dessen Antwortnachricht dar.
- ➔ Man erkennt, dass das Programm (d.h., der ausführende Thread) längere Zeit blockiert ist und damit viel Zeit benötigt.
- ➔ Insbesondere könnte das Programm beschleunigt werden, wenn das Laden der beiden Seiten (also die Aufrufe der Methode `loadPage()`) nebenläufig erfolgen könnte.

6.3.2 Realisierung mit Threads

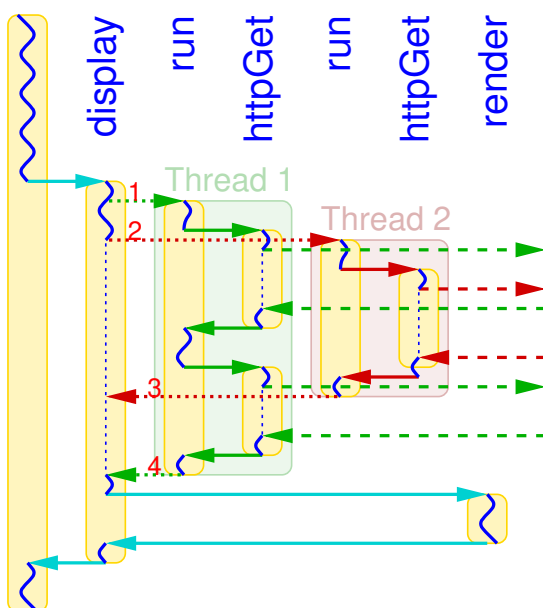


(Animierte Folie)

```
void display() {  
    HttpThread t1 = new HttpThread("...1"); t1.start();  
    HttpThread t2 = new HttpThread("...2"); t2.start();  
    t1.join(); t2.join();  
    render(t1.res, t2.res);  
}  
  
class HttpThread extends Thread {  
    String res, url;  
    HttpThread(String url) { this.url = url; }  
    void run() {  
        res = httpGet(url);  
        String target = checkRedirect(res);  
        if (target != null)  
            res = httpGet(target);  
    }  
}
```

Anmerkungen zu Folie 315:

Zeitlicher Ablauf des Programms:



Die gepunkteten Linien stellen die Thread-erzeugung und die Benachrichtigung über die Beendigung eines Threads dar.

1. Thread t1 wird erzeugt.
2. Thread t2 wird erzeugt.
3. Thread t2 terminiert.
4. Thread t1 terminiert.



Diskussion

- ➔ Laden der beiden Seiten erfolgt nebenläufig
 - ➔ während des Wartens auf die erste Seite kann weitergearbeitet werden
- ➔ Relativ hoher Overhead zum Erzeugen der Threads
 - ➔ Systemaufrufe!
- ➔ Ressourcenverbrauch der Threads (insbes. Kellerspeicher) bei Servern problematisch
 - ➔ ggf. Tausende von nebenläufigen Anfragen!
- ➔ Ggf. Synchronisation der Threads erforderlich
- ➔ Komplexe Änderung der Programmstruktur notwendig

6.3.3 Asynchrone Programmierung



- ➔ Basis: Nutzung **asynchroner** Methodenaufrufe
- ➔ Idee: aufgerufene Methode kann zum Aufrufer zurückkehren, **bevor** ihre Aufgabe beendet ist
 - ➔ Aufgabe wird „im Hintergrund“ weiter bearbeitet
- ➔ Aufrufer kann jetzt nebenläufig weiterarbeiten
 - ➔ z.B. weitere asynchrone Methoden aufrufen
- ➔ Nach Abschluss der Aufgabe müssen aber i.a. weitere Aktionen durchgeführt werden
- ➔ Dazu zwei Alternativen:
 - ➔ Aufrufer kann bei Bedarf explizit auf Abschluss warten
 - ➔ bei Abschluss der Aufgabe wird automatisch eine vorgegebene Aktion (Methode) gestartet
 - ➔ z.B. in Form eines Callbacks

Anmerkungen zu Folie 317:

Ein grundsätzliches Problem bei der asynchronen Programmierung ist die Behandlung von Fehlern bzw. Exceptions. Auf die dafür verfügbaren Konzepte und Konstrukte wird im Folgenden aber nicht näher eingegangen.

317-1

6.3.4 Asynch. Programmierung mit Callbacks



(Animierte Folie)

```
void display() {
    String[] pages = new String[2];
    loadPage("...1", p -> { pages[0] = p; });
    loadPage("...2", p -> { pages[1] = p; });
    while ((pages[0] == null) || (pages[1] == null));
    render(pages[0], pages[1]);
}

void loadPage(String url, Callback cb) {
    httpGet(url, p -> { load1(p, cb); });
}

void load1(String p, Callback cb) {
    String target = checkRedirect(p);
    if (target != null)
        httpGet(target, cb);
    else
        cb.invoke(p);
}
```

Anmerkungen zu Folie 318:

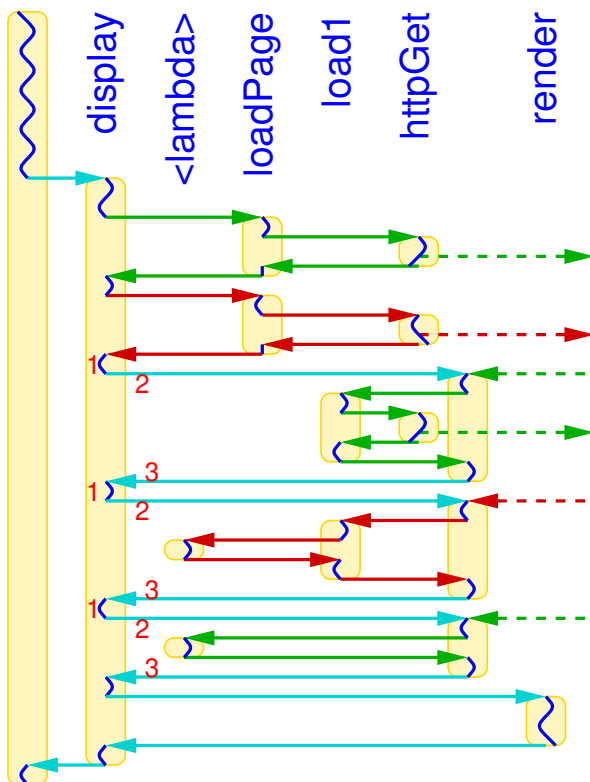
- ➔ Mit der Syntax `p -> { ... }` (einem sog. Lambda-Ausdruck) kann in Java eine anonyme Funktion erzeugt werden. Da Java nur Objekte kennt, wird dies so realisiert, daß ein Objekt erzeugt wird, dessen Klasse nur eine Methode (nämlich die durch den Lambda-Ausdruck definierte Funktion) hat.
- ➔ Damit das Beispiel funktioniert, müsste in Java noch explizit eine Schnittstelle für den Typ der Lambda-Ausdrücke definiert werden (ein sog. *Functional Interface*):

```
interface Callback {  
    void invoke(String s);  
}
```

- ➔ Ein grundsätzliches Problem bei der Verwendung von Callbacks ist das gleichzeitige Warten auf die Beendigung zweier asynchroner Aktivitäten. Im Beispiel ist dies in der Methode `display()` der Einfachheit halber durch aktives Warten gelöst.

318-1

Zeitlicher Ablauf des Programms:



1. Aktives Warten auf beide Ergebnisse.
2. Die Funktion `httpGet` muss es so einrichten, daß beim Eintreffen der Antwort eine Unterbrechung des aktuell ausgeführten Programmcodes ausgelöst und eine zu `httpGet` gehörige Behandlungsrouting aufgerufen wird.
[Prinzipiell ist der Signalmechanismus hier verwendbar. Da in einem Signalhandler viele Operationen (z.B. Speicheranforderung, Synchronisation) nicht sicher ausgeführt werden können, kann der Callback dabei aber nicht direkt vom Signalhandler aufgerufen werden.]
3. Rückkehr aus der Behandlungsroutine.

318-2



Diskussion

- ➔ Keine Verwendung von Threads nötig
 - z.B., wenn keine Threadunterstützung vorhanden ist
- ➔ Verwendung von Callbacks führt zu sehr unübersichtlicher Programmstruktur
- ➔ Bearbeitung im Hintergrund muß geeignet realisiert werden
 - z.B. mit Interrupt- oder Signalhandlern, ggf. auch mit Threads
- ➔ Übergang in die „synchrone Welt“ schwierig
 - benötigt eine Möglichkeit, auf den Aufruf eines Callbacks zu warten
 - im Beispiel mit aktivem Warten gelöst (unschön)

6.3.5 Futures und Promises



- ➔ **Future**: Variable, deren Wert erst in der Zukunft verfügbar wird
 - sobald die zugehörige Berechnung beendet ist
- ➔ **Promise**: Funktion/Berechnung, die den Wert des *Futures* setzt
 - oft mit *Future* gleichgesetzt
- ➔ Unterscheidung explizites / implizites *Future*
 - explizit: vor der Verwendung des Werts muß explizit gewartet werden, bis dieser berechnet wurde
 - implizit: Compiler erzeugt Synchronisation, wo nötig
- ➔ *Future* kann (per Referenz) als Argument etc. weitergegeben werden, ohne auf den Wert zu warten
- ➔ Ggf. auch Aufruf einer Berechnung mit dem noch nicht vorhandenen Wert möglich (**Promise Chaining**):
 - Ergebnis ist dann wieder ein *Future*

(Animierte Folie)

Explizite Futures mit blockierendem Warten

```
import java.util.concurrent.*;

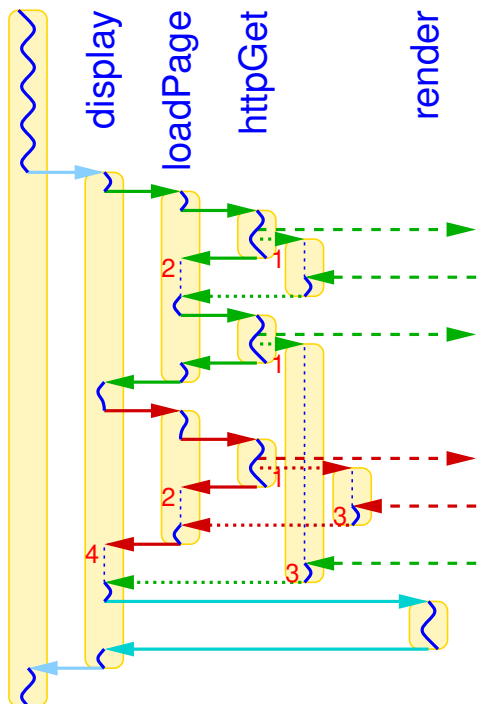
void display() {
    Future<String> p1 = loadPage("...1");
    Future<String> p2 = loadPage("...2");
    render(p1.get(), p2.get());
}

Future<String> loadPage(String url) {
    String p = httpGet(url).get();
    String target = checkRedirect(p);
    if (target != null)
        return httpGet(target);
    return CompletableFuture.completedFuture(p);
}
```

Anmerkungen zu Folie 321:

Die Schnittstelle `Future` und die Klasse `CompletableFuture` stellt Java im Paket `java.util.concurrent` zur Verfügung.

Zeitlicher Ablauf des Programms:



Im Beispiel wurde angenommen, daß `httpGet()` jeweils einen neuen Thread erzeugt, der das *Future* dann zu einem späteren Zeitpunkt ausfüllt. Die senkrechten gepunkteten Linien zeigen an, eine Blockierung des jeweiligen Threads an. Die waagerechten gepunkteten Linien stellen wieder die Threaderzeugung und die Benachrichtigung bei der Terminierung dar.

1. `httpGet()` erzeugt einen neuen Thread, der auf die Antwortnachricht des Servers wartet und diese in das *Future* schreibt.
2. `loadPage()` blockiert in `get()` solange, bis das Ergebnis von `httpGet()` verfügbar ist.
3. Das *Future* wird beschrieben.
4. `display()` blockiert in `p1.get()` bis der Wert von `p1` verfügbar ist.

322-1

6.3.5 Futures und Promises ...



(Animierte Folie)

Futures mit Promise Chaining

```

void display() {
    Future<String> p1 = loadPage("...1");
    Future<String> p2 = loadPage("...2");
    render(p1.get(), p2.get());
}

Future<String> loadPage(String url) {
    // Anmerkung: so in Java NICHT möglich!!
    return httpGet(url).then(p -> { load1(p); });
}

Future<String> load1(String p) {
    String target = checkRedirect(p);
    if (target != null)
        return httpGet(target);
    return CompletableFuture.completedFuture(p);
}

```


6.3.6 Async / Await (Koroutinen)



- ➔ Unterstützung asynchroner Funktionsaufrufe durch Sprache / Compiler
- ➔ Schlüsselwort `async` kennzeichnet asynchrone Funktion
 - ➔ liefert automatisch ein *Future* zurück
- ➔ Schlüsselwort `await` „wartet“, bis der Wert verfügbar ist
 - ➔ bedeutet hier: Funktion (Koroutine!) kehrt erst einmal zurück, neuer Aufruf, wenn Wert (voraussichtlich) zur Verfügung steht
 - ➔ benötigt einen *Executor*, der regelmäßiges Polling der Koroutinen durchführt
- ➔ Programmstruktur sehr ähnlich zu sequentiellm Programm
 - ➔ aber: zeitlicher Ablauf trotzdem sehr komplex
- ➔ Realisierung ohne Threads möglich (z.B. in Rust)

6.3.6 Async / Await (Koroutinen) ...



(Animierte Folie)

```
// Anmerkung: so in Java NICHT möglich!!
void display() {
    Future<String> p1 = loadPage("...1");
    Future<String> p2 = loadPage("...2");
    Future<String[]> pp = join(p1, p2);
    String[] p = block_on(pp);
    render(p[0], p[1]);
}

async String loadPage(String url) {
    String p = await httpGet(url);
    String target = checkRedirect(p);
    if (target != null)
        return await httpGet(target);
    return p;
}
```

Anmerkungen zu Folie 325:

Java unterstützt kein *Async/Await*! Das Beispiel ist an die Sprache Rust angelehnt.

Die gepunkteten Pfeil nach rechts zeigen die Erzeugung eine Koroutine an. In Rust wird an dieser Stelle noch kein Code der Koroutine ausgeführt. Koroutinen werden in Rust in Zustandsautomaten übersetzt, deren Zustände sich i.W. den Ausführungszustand der Koroutine merken.

- ➔ Die Weiterführung einer Koroutine wird durch den Aufruf einer Methode `poll()` auf der Koroutine angestoßen (strichpunktierter Pfeil nach rechts).
- ➔ Die Koroutine läuft dann solange, bis sie
 - auf ein Ereignis warten muß. In diesem Fall gibt `poll()` den Status `Pending` zurück (strichpunktierter Pfeil nach links). Zudem erhält der Aufrufer über ein Argument von `poll()` eine Benachrichtigung, wenn er die Methode das nächste Mal aufrufen soll (damit wird unnötiges *Busy Waiting* vermieden).
- oder
 - ihr Ende erreicht. In diesem Fall gibt `poll()` den Status `Ready` zurück (gepunkteter Pfeil nach links).
- ➔ Das Polling wird vom einem *Executor* durchgeführt, der im Beispiel über die Funktion `block_on()` aufgerufen wird.

325-1

Mehr Informationen zu *Async/Await* finden Sie u.a. unter

- ➔ <https://en.wikipedia.org/wiki/Async/await>
- ➔ <https://os.phil-opp.com/async-await>
- ➔ <https://rust-lang.github.io/async-book>
- ➔ <https://medium.com/@KevinBGreene/async-programming-in-rust-part-3-futures-and-async-await-b508f7e44abc>

325-2

- ➔ `aio_error()` erlaubt Abfrage des Ausführungszustands
 - ➔ inkl. Fehlerstatus
- ➔ `aio_suspend()` blockiert den Aufrufer, bis mindestens einer der spezifizierten Aufträge beendet ist
- ➔ Mit `lio_listio()` kann eine ganze Liste nebenläufiger Aufträge gestartet werden
 - ➔ zur Einsparung von Systemaufrufen

Anmerkungen zu Folie 327:

- ➔ Das Öffnen und Schließen von Dateien erfolgt nach wie vor über synchrone Systemaufrufe, da diese Funktionen nicht relevant für die *Performance* sind.
- ➔ Die genannten Funktionen sind in Linux als Bibliothek realisiert. Die eigentlichen Systemaufrufe sind im Detail anders, realisieren aber prinzipiell dieselbe Funktion.
- ➔ Ein Tutorial zu asynchroner E/A unter Linux finden Sie unter <https://developer.ibm.com/articles/l-async>