Betriebssysteme und nebenläufige Programmierung

SoSe 2025

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

Stand: 1. April 2025

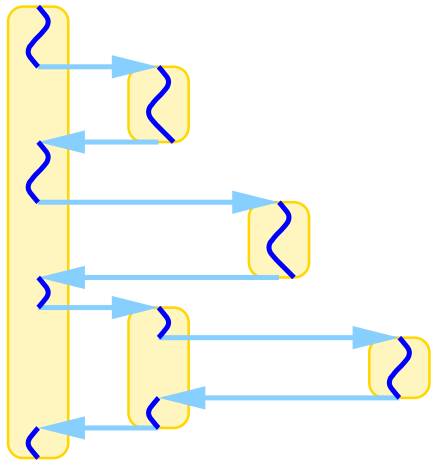
Betriebssysteme und nebenläufige Programmierung

SoSe 2025

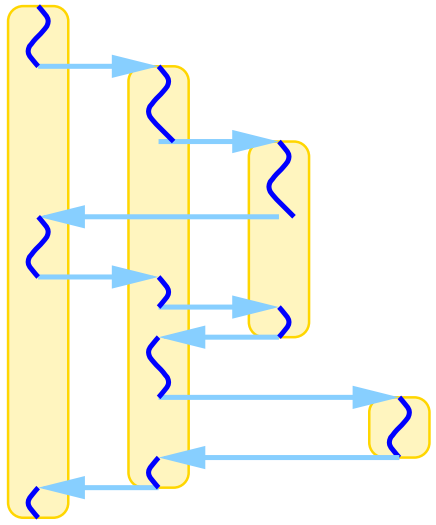
6 Koroutinen und asynchrone Programmierung

6.1 Koroutinen

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



- **Koroutinen** (Convay 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 „Unterbrechungs-stelle“
  - 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 ressourcesbeschränkten Systemen (z.B. eingebetteten Systemen, Sensornetzen, etc.)

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)));
}
```

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

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)));
}
```

In dieser Struktur wird der Zustand der Koroutine gespeichert

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)));
}
```

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

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)));
}
```

Initialisierung von zwei Koroutinen

Beispiel

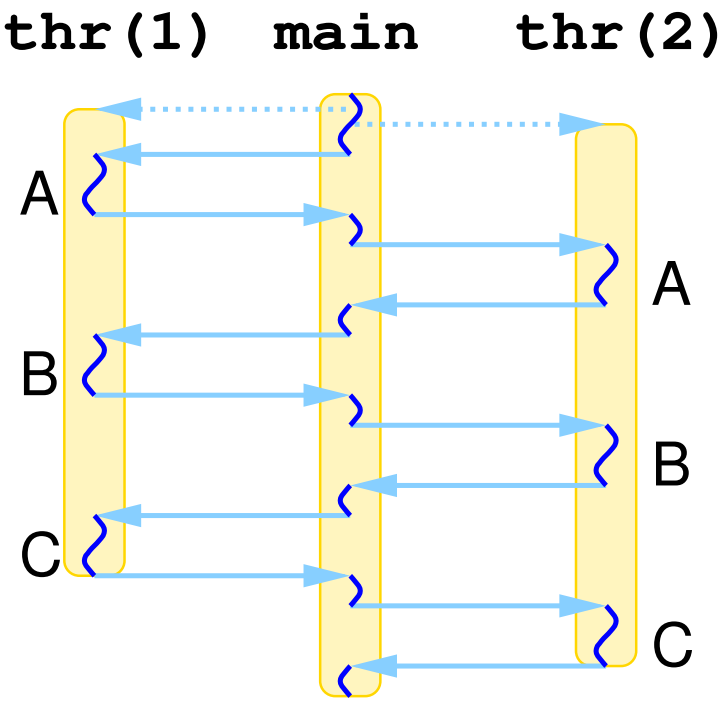
```
PT_THREAD(thr(struct pt *pt, int no)) {
  PT_BEGIN(pt);
  printf("Thr %d: A\n", no);
  PT_YIELD(pt);
  printf("Thr %d: B\n", no);
  PT_YIELD(pt);
  printf("Thr %d: C\n", no);
  PT_END(pt);
}
int main() {
  struct pt pt1, pt2;
  PT_INIT(&pt1);
  PT_INIT(&pt2);
  while(PT_SCHEDULE(thr(&pt1, 1) & thr(&pt2, 2)));
}
```

Beide Koroutinen abwechselnd vorantreiben

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)));
}
```

Ablauf:



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;
}
```

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

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_YIELD(pt)

Sichert Fortsetzungsadresse und kehrt zum Aufrufer zurück

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_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

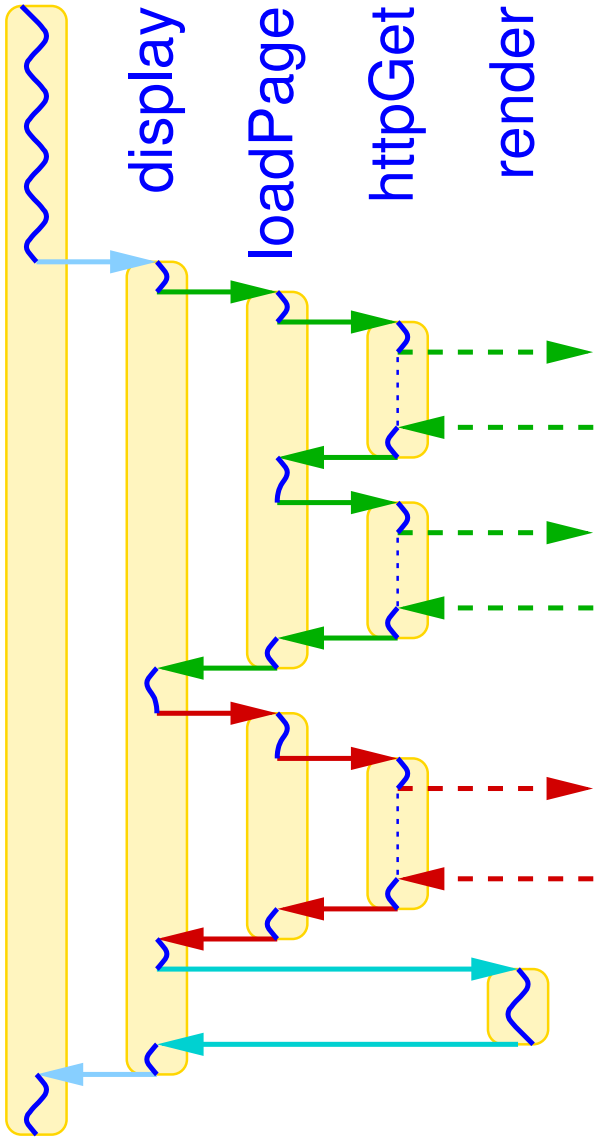
- Viele Programme (insbes. netzwerkbasierte Clients und Server) sind stark E/A-lastig
  - Programm blockiert die meiste Zeit, um auf E/A zu warten
- Wunsch: Warten auf E/A soll nebenläufig erfolgen können
  - z.B. unabhängige E/A Aufträge gleichzeitig ausführen
  - beschleunigt Programmausführung auch 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:

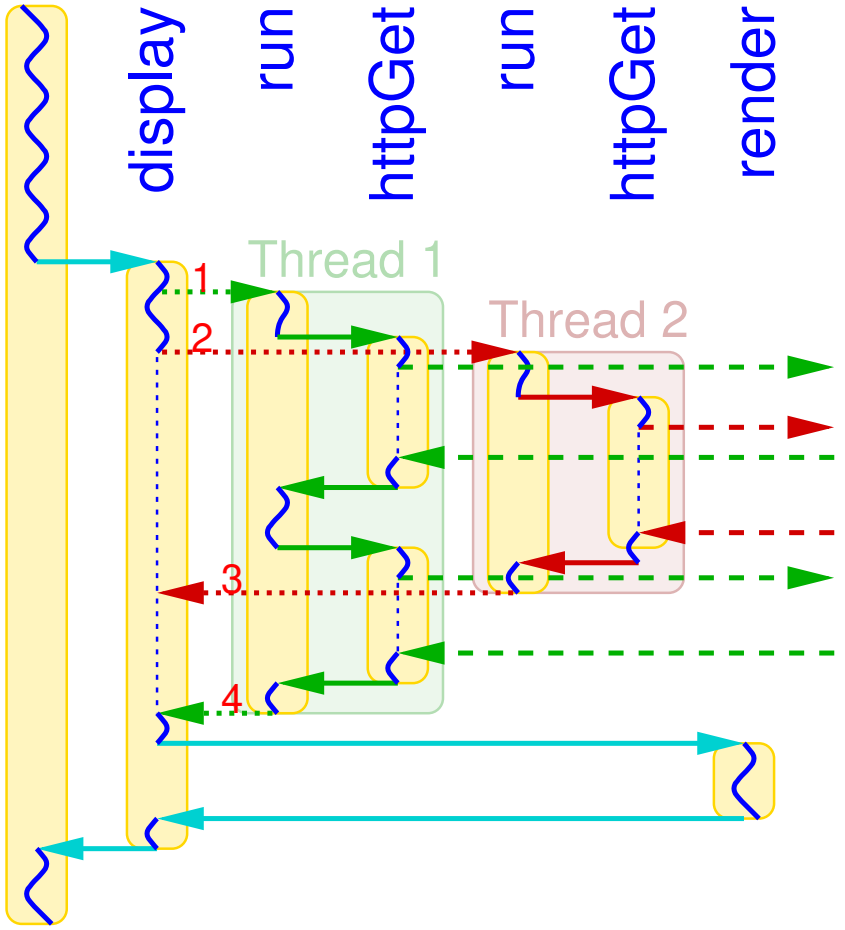


6.3.2 Realisierung mit Threads

```
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);
  }
}
```

6.3.2 Realisierung mit Threads

```
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
  void run() {
    res = httpGet(url);
    String target = checkRedire
    if (target != null)
      res = httpGet(target);
  }
}
```



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

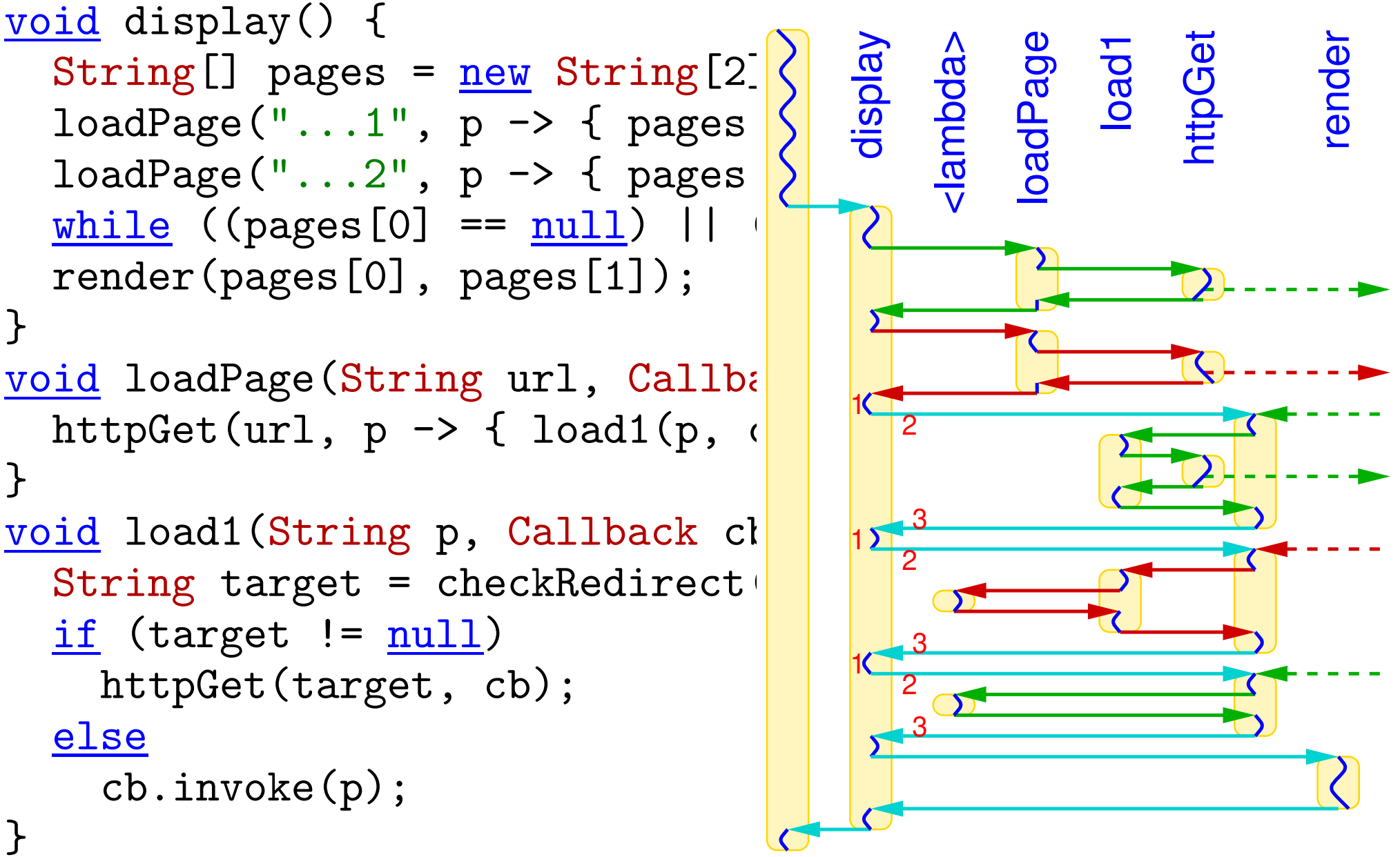
6.3.3 Asynchrone Programmierung

- Basis: Nutzung **a**synchroner 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

6.3.4 Asynch. Programmierung mit Callbacks

```
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);
}
```

6.3.4 Asynch. Programmierung mit Callbacks



```
void display() {
  String[] pages = new String[2]
  loadPage("...1", p -> { pages
  loadPage("...2", p -> { pages
  while ((pages[0] == null) ||
  render(pages[0], pages[1]);
}
void loadPage(String url, Callba
  httpGet(url, p -> { load1(p, c
}
void load1(String p, Callback cb
  String target = checkRedirect
  if (target != null)
    httpGet(target, cb);
  else
    cb.invoke(p);
}
```

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/Berechung, die den Wert des *Future*s 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 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*

Explizite *Futures* mit blockierendem Warten

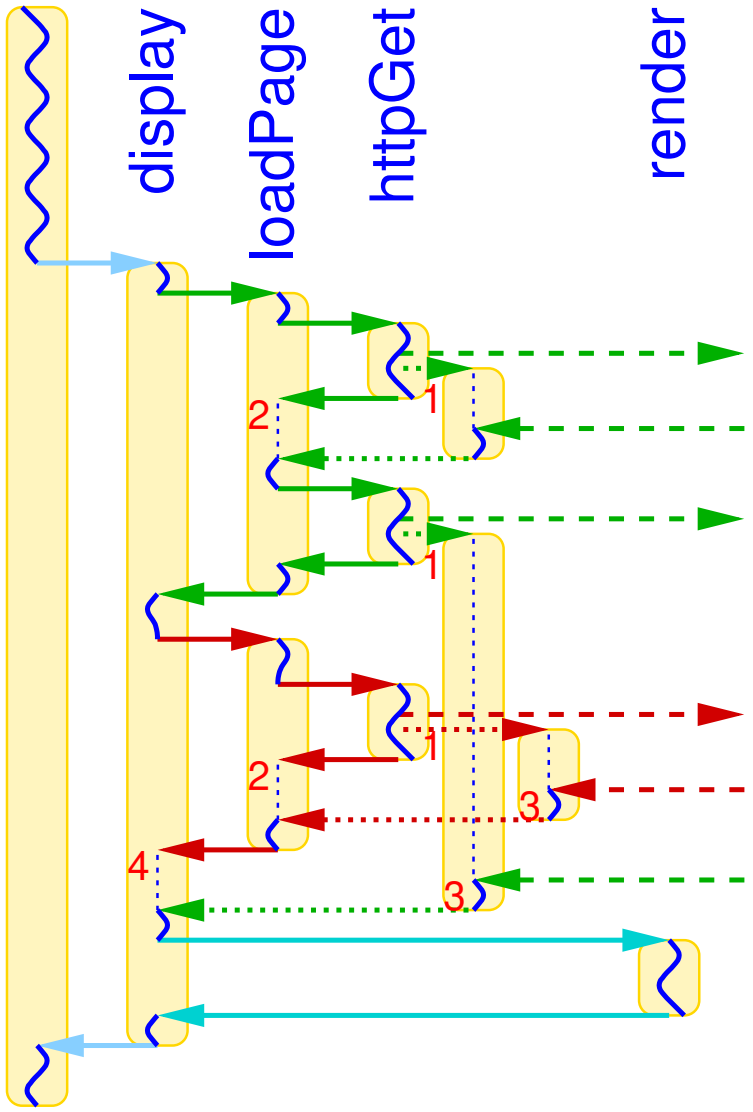
```
import java.util.concurrent.*;

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);
}
```

Explizite *Futures* mit blockierendem Warten

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

void display() {
  Future<String> p1 = loadPage("...
  Future<String> p2 = loadPage("...
  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.complete
}
```

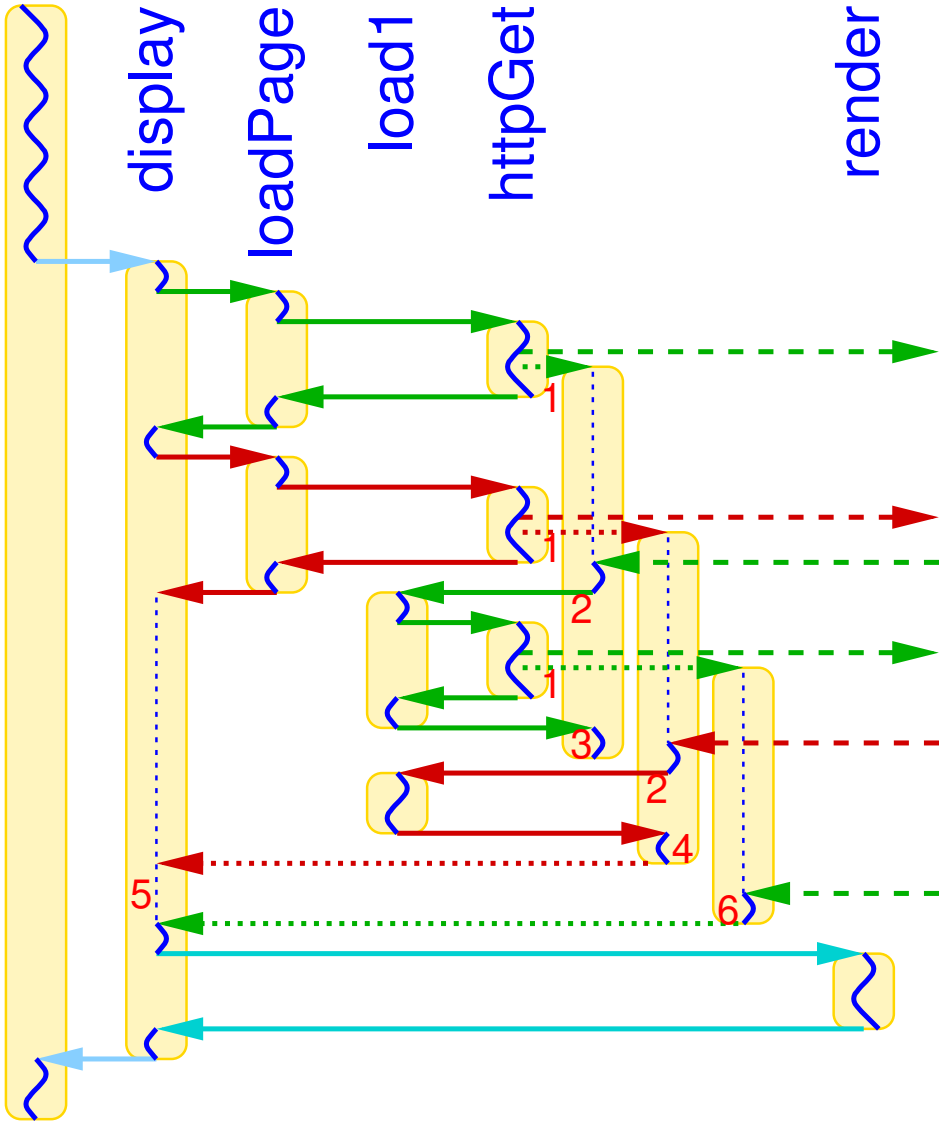


Futures mit Promise Chaining

```
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);
}
```

Futures mit Promise Chaining

```
void display() {
  Future<String> p1 = loadPage
  Future<String> p2 = loadPage
  render(p1.get(), p2.get());
}
Future<String> loadPage(String
  // Anmerkung: so in Java NICHT möglich
  return httpGet(url).then(p -
}
Future<String> load1(String p)
  String target = checkRedirec
  if (target != null)
    return httpGet(target);
  return CompletableFuture.com
}
```



6.3.6 Async / Await (Koroutinen)

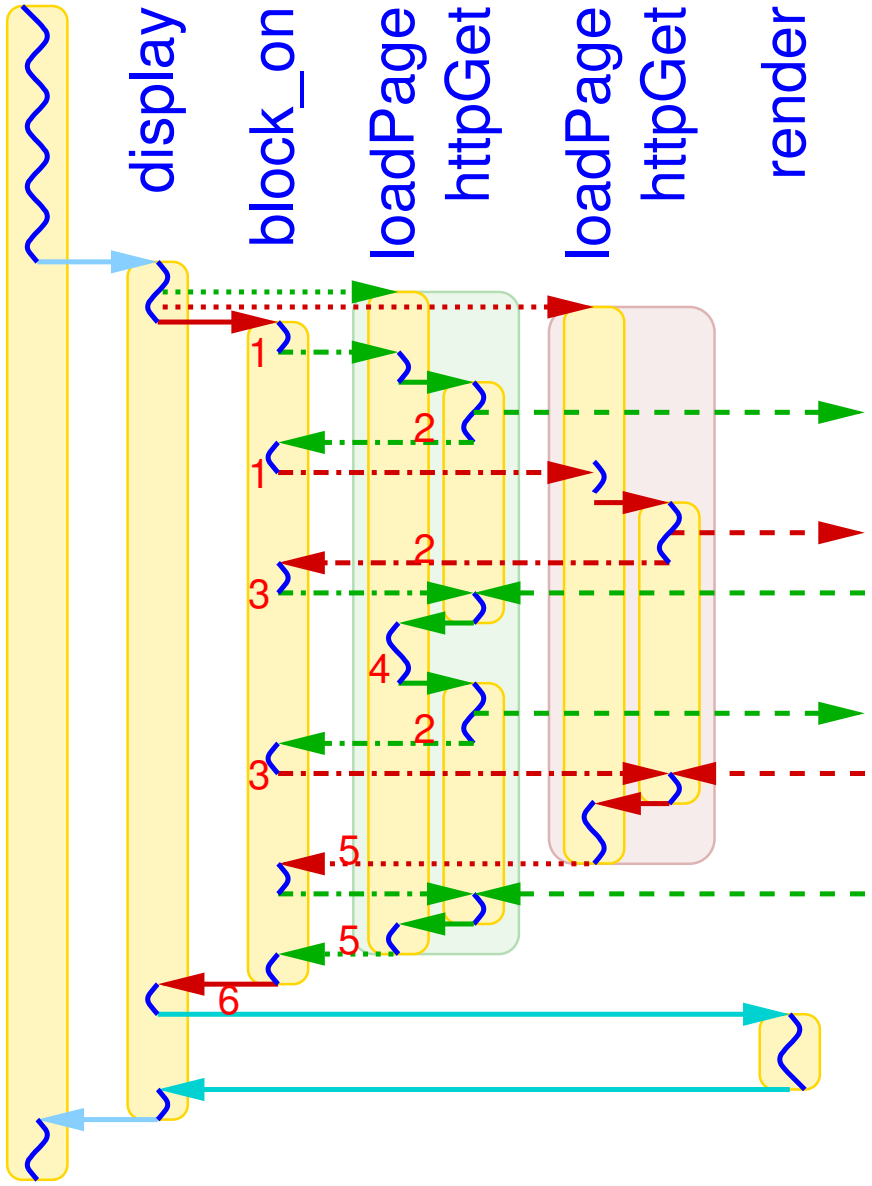
- Unterstützung asynchroner Funktionsaufrufe durch Sprache / Compiler
- Schlüsselwort `async` kennzeichet asynchrone Funktion
  - liefert automatisch ein *Future* zurück
- Schlüsselwort `await` „wartet“, bis der Wert verfügbar ist
  - bedeutet hier: Funktion (Koroutine!) kehrt erst einmal zurück, neuer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

6.3.6 Async / Await (Koroutinen) ...

```
// 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;
}
```

6.3.6 Async / Await (Koroutinen) ...

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



6.4 Asynchrone Ein-/Ausgabe in POSIX

- Asynchrone Alternativen für die blockierenden Systemaufrufe `read()` und `write()`
  - Zugriff auf Dateien und Geräte, Netzwerkkommunikation
- `aio_read()` und `aio_write()` starten Lese- bzw. Schreib-Aufträge, ohne auf Beendigung zu warten
- Auftrag enthält u.a.:
  - Dateideskriptor
  - Offset in der Datei
    - kein globaler Dateizeiger wegen nebenläufiger Operationen
  - Quell- bzw. Zielpuffer mit Länge
  - Benachrichtigungs-Methode: keine Benachrichtigung, Signal, oder Aufruf einer Funktion in einem separaten Thread

6.4 Asynchrone Ein-/Ausgabe in POSIX ...

- `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