Echtzeitsysteme

2. Nebenläufigkeit (*Concurrency*)

Prof. Dr. Roland Dietrich

Nebenläufigkeit und Parallelität

- **Nebenläufigkeit (*Concurrency*)**: Notationen und Techniken
  - Für die **potentiell parallele Ausführung** von Aktivitäten
    - Wir tun so, als wäre parallele Ausführung tatsächlich möglich
  - Die Lösung der dabei entstehenden **Synchronisation**s- und **Kommunikation**sprobleme
- Möglich auf verschiedenen Abstraktionsebenen
  - Modelle (z.B. UML-Aktivitätsdiagramme, endliche Automaten)
  - Programmiersprachen
- **Parallelität (*Parallelism*)**
  - Parallele (gleichzeitige) Ausführung mehrere Aktivitäten
  - Implementierung von echter Parallelität ist nur auf einer geeigneten Realisierungsplattform (HW und SW) möglich
  - Die Realisierungsplattform ist bei der Betrachtung von Nebenläufigkeit zunächst irrelevant
- Frage: Nutzt Nebenläufigkeit etwas ohne echte Parallelität?

- **Was nützt Parallelität?**
  - **Amdahl's Gesetz** beschreibt die Beziehung zwischen Anzahl verfügbarer Prozessoren und der dadurch möglichen Beschleunigung von Algorithmen durch Parallelisierung

P = Anteil eines Algorithmus (Codes), der Parallelisiert werden kann

N = Anzahl der zur Verfügung stehenden Prozessoren

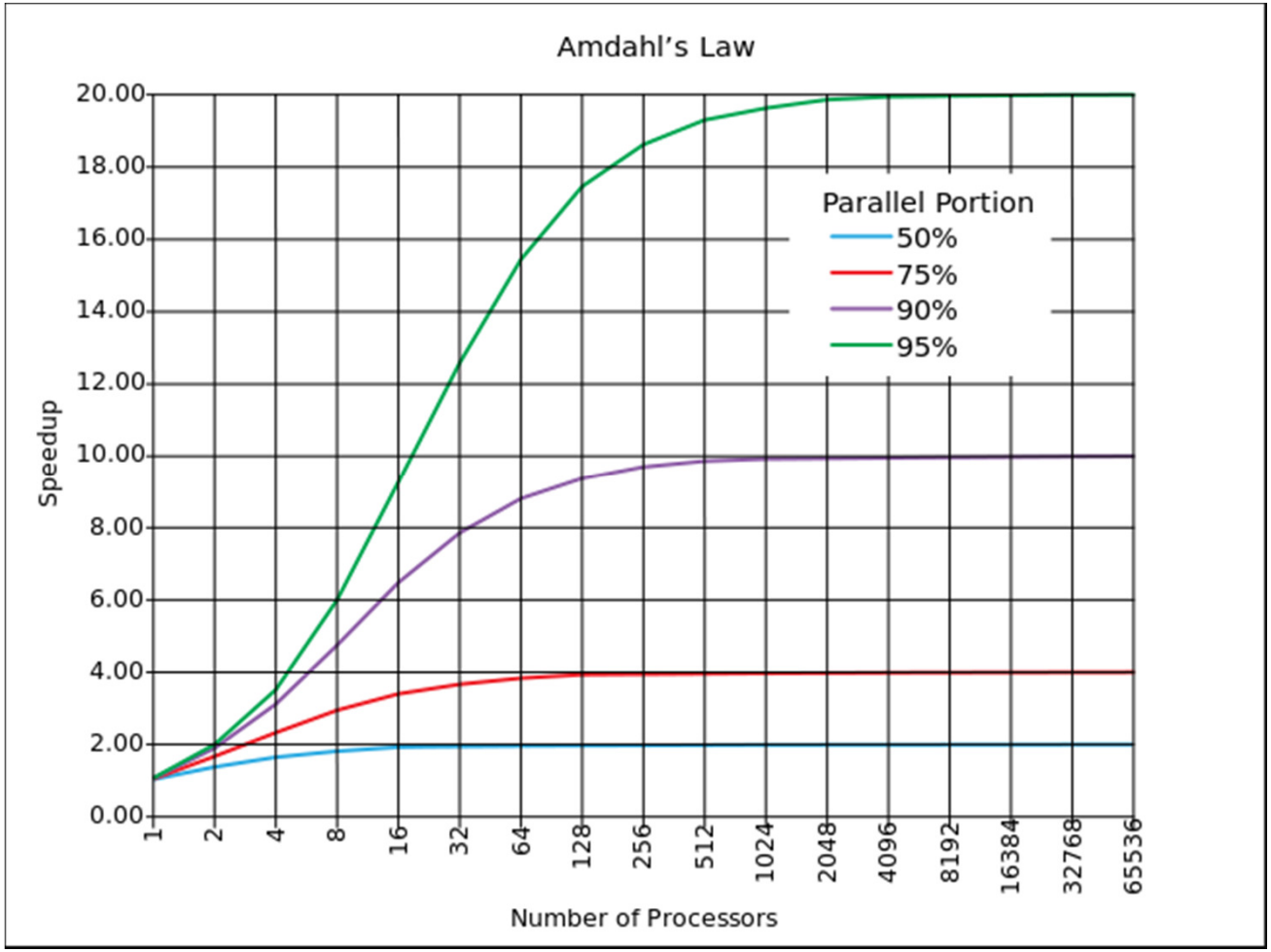
Maximale Beschleunigung $S(N) = \frac{1}{(1-P) + \frac{P}{N}}$

Beispiel:

P = 1/2, N = 4	S(N) = 1.6
P = 1/2	
P = 1/2, N = 100	S(N) = 1.98
P = 1/2, N → ∞	S(N) = 2

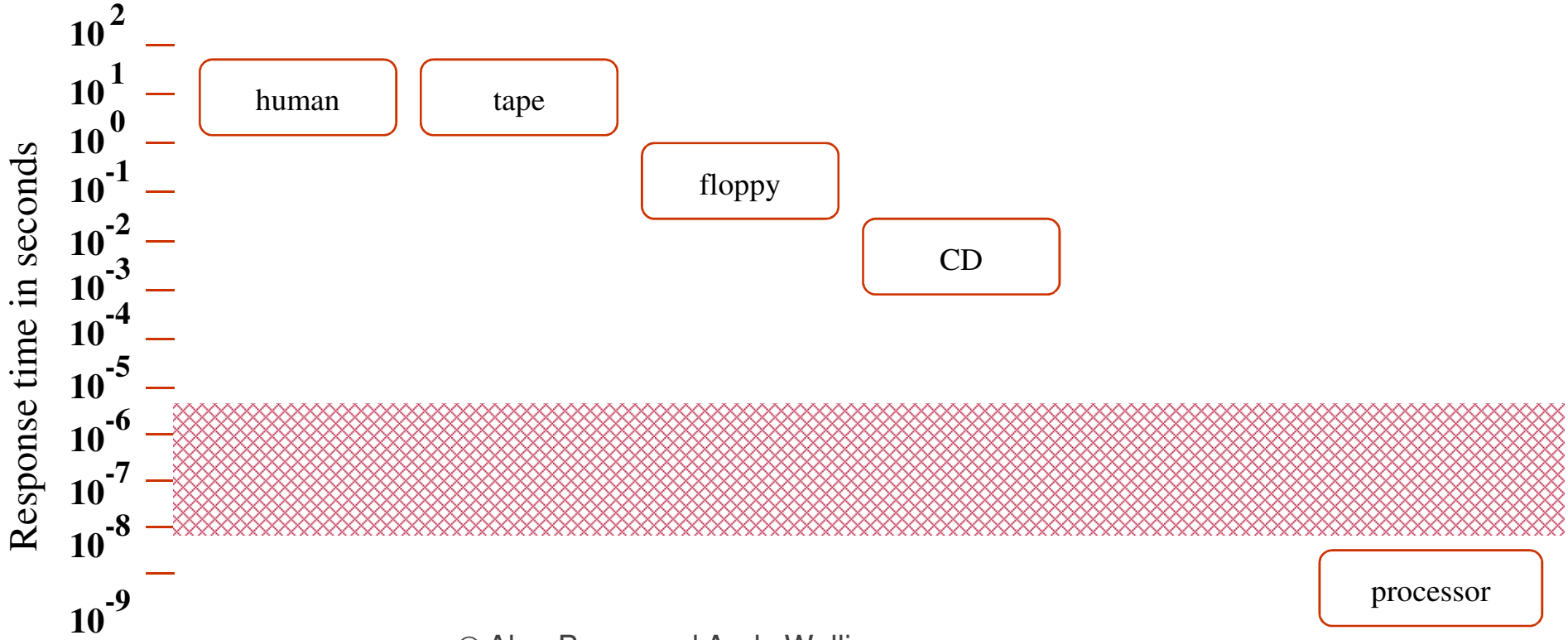
Nebenläufigkeit und Parallelität

- Was nützt Parallelität?

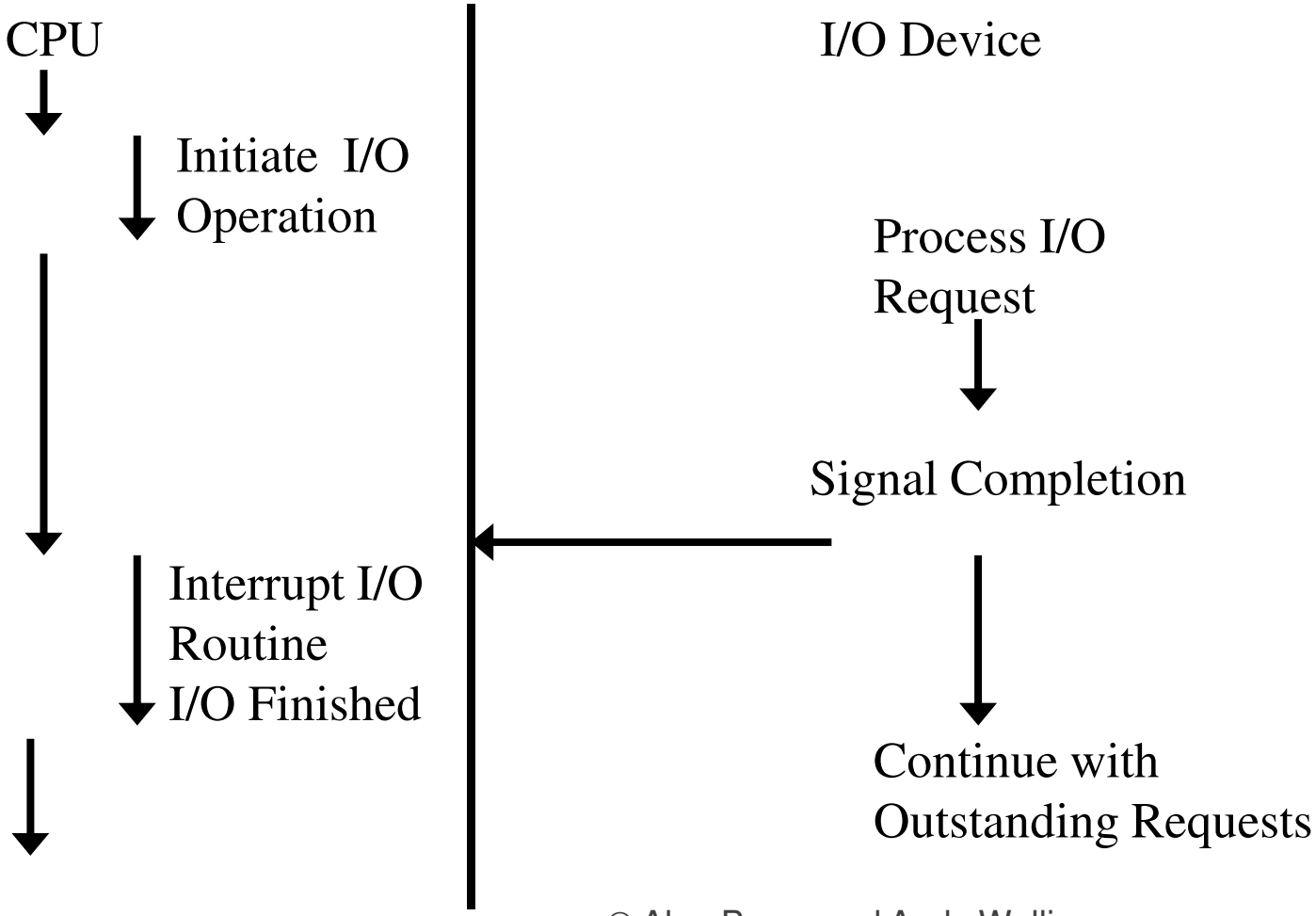


[Quelle: http://en.wikipedia.org/wiki/Amdahl's_law]

- **Warum Nebenläufigkeit?**
  - Um den Prozessor gut auszunutzen
    - E/A-Operation dauern wesentlich länger als Prozessor-Operationen



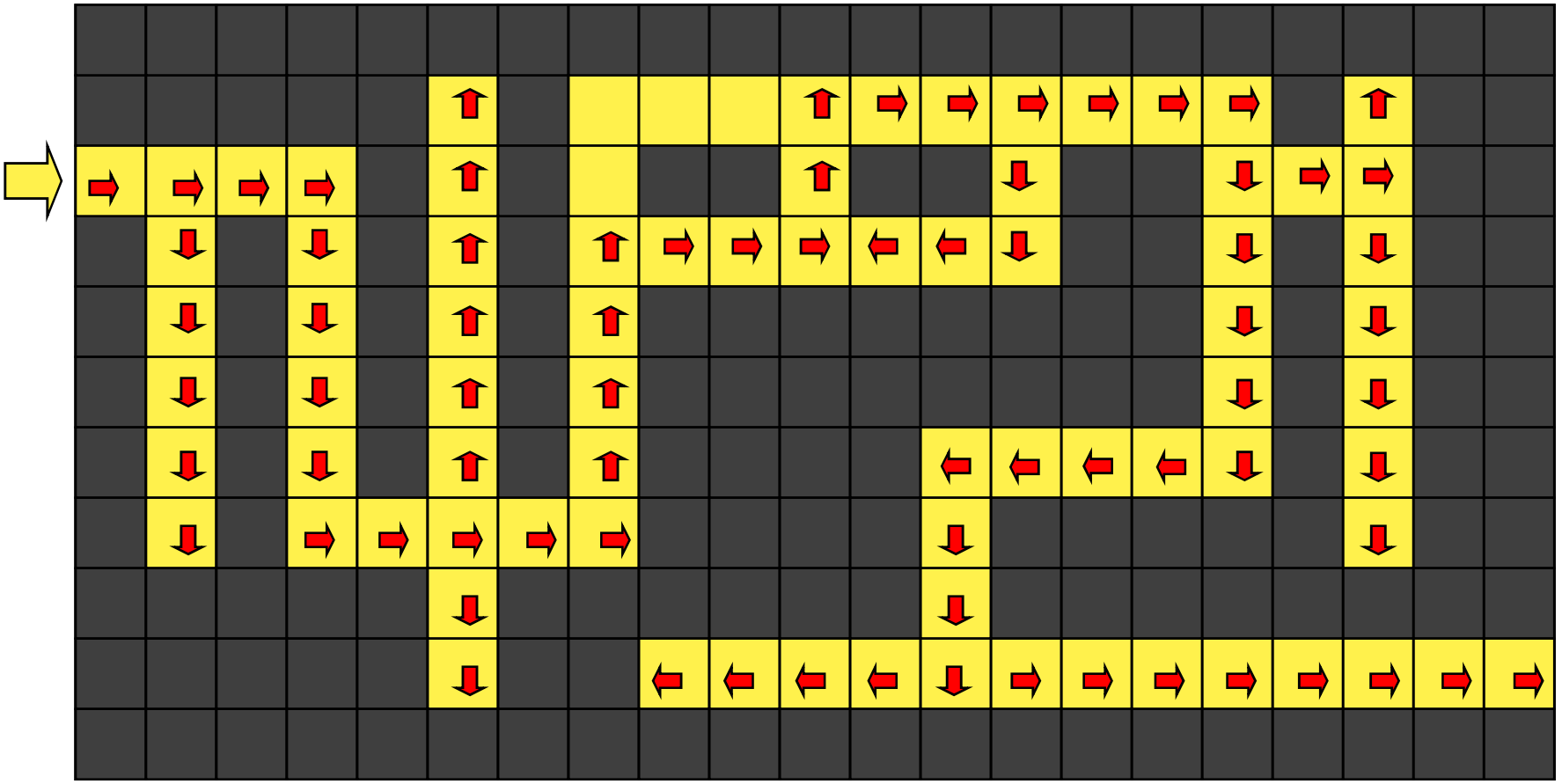
- **Warum Nebenläufigkeit?**
  - Nebenläufigkeit zwischen CPU und E/A-Geräten



© Alan Burns and Andy Wellings

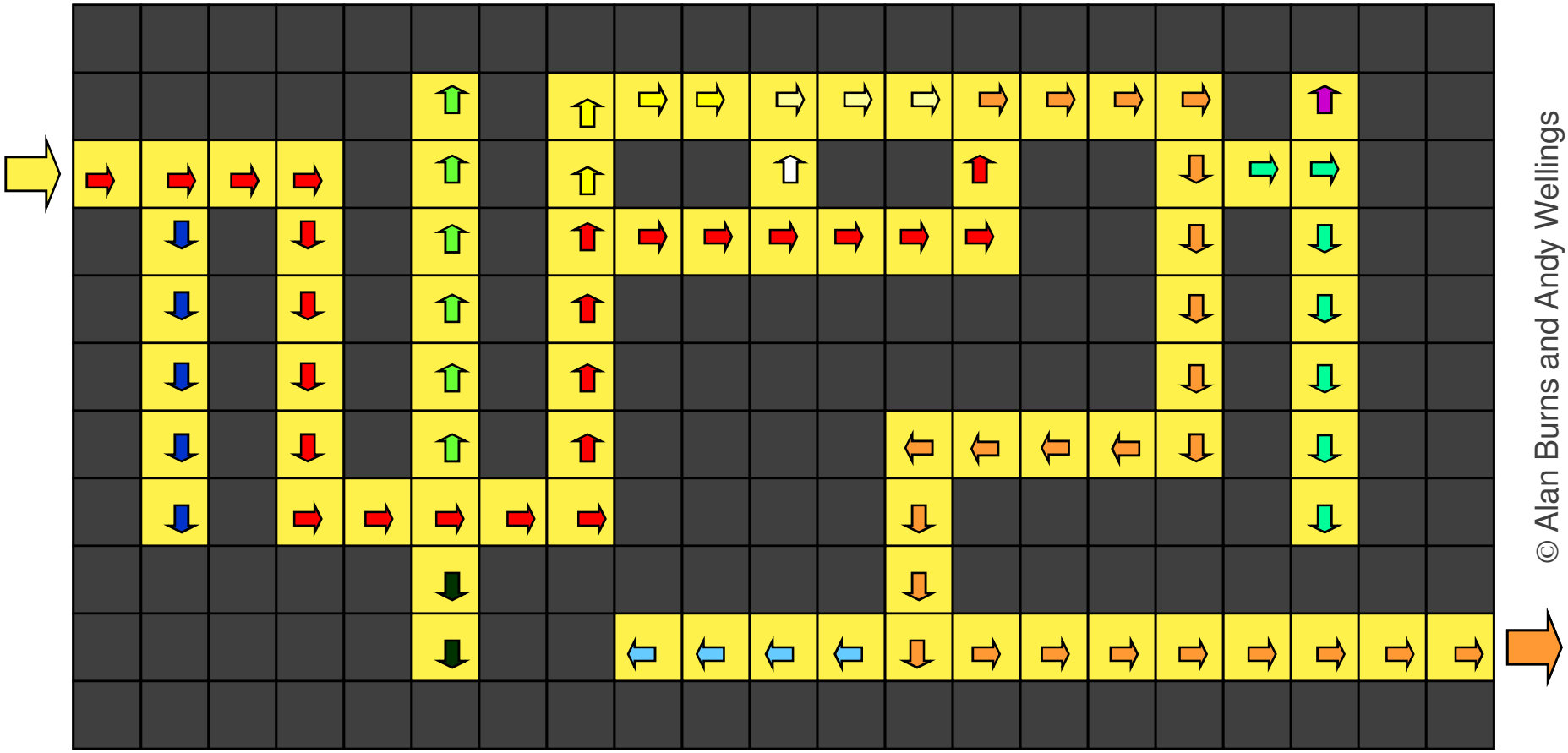
Nebenläufigkeit und Parallelität

- **Warum Nebenläufigkeit?**
  - Um potentielle Parallelverarbeitung auszudrücken
    → Grundlage für echte Parallelisierung
  - Beispiel: Finde den Weg durch ein Labyrinth – sequentielle Lösung



- **Warum Nebenläufigkeit?**
  - Beispiel: Finde den Weg durch ein Labyrinth – nebenläufige Lösung

    Jede Pfeilfolge einer Farbe entspricht einem unabhängigen Prozess



Nebenläufigkeit und Parallelität

- **Warum Nebenläufigkeit?**
  - Die reale Welt ist parallel!
  - Insbesondere eingebettete und Echtzeitsysteme
    - In der Regel müssen mehrere Geräte gleichzeitig gesteuert werden
      - Roboter
      - Fließband
      - Kameras
      - Motoren
      - ...
  - Wenn sich die parallelen Strukturen der realen Welt in den Programmstrukturen widerspiegeln, sind die Programme
    - verständlicher
    - wartbarer
    - zuverlässiger

Prozesse und Tasks

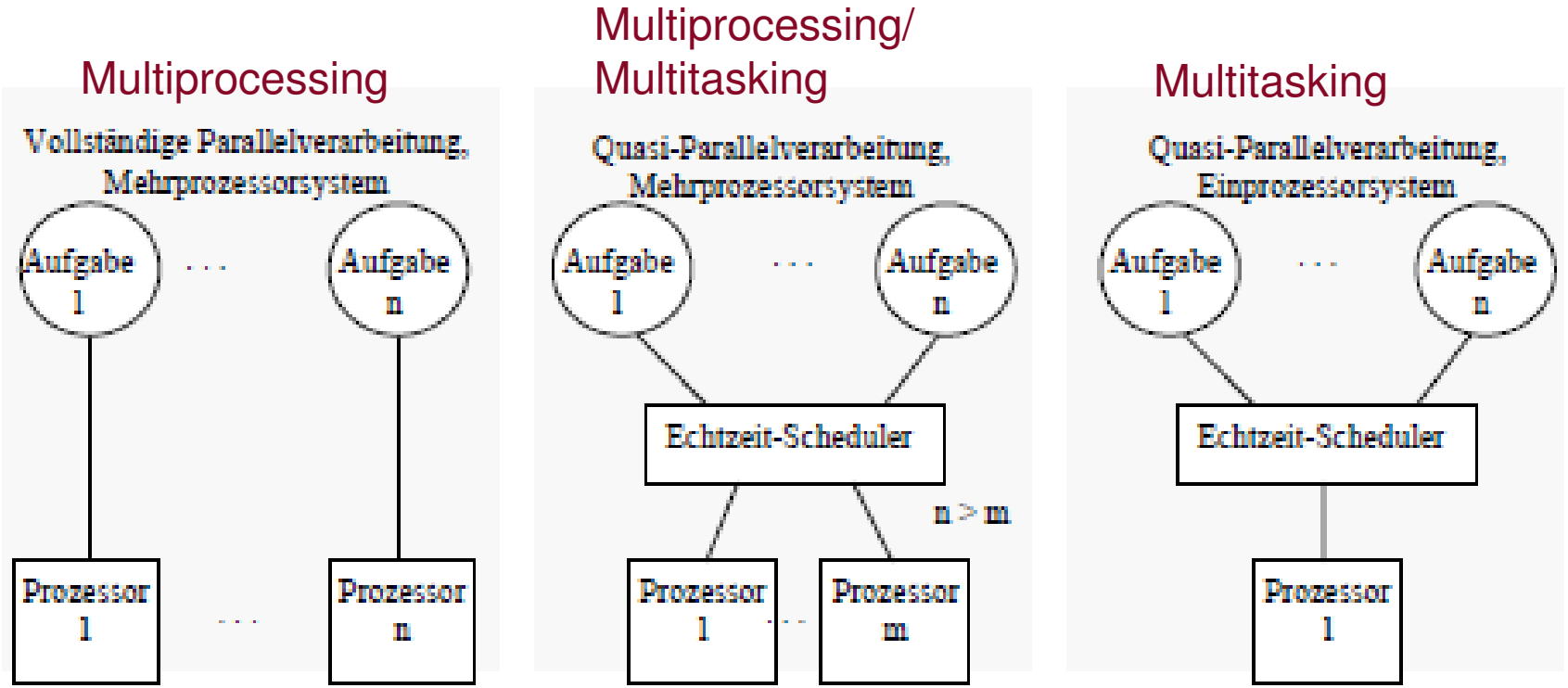
- Ein Rechenprozess, kurz **Prozess** ist ein Programm in Ausführung
  - besteht aus einer Menge von n unabhängigen, sequentiellen *Tasks*
  - Jede *Task* kann potentiell gleichzeitig mit anderen ausgeführt werden
  - Alle Tasks eines Prozesses teilen sich den selben Adressraum (*shared memory*)
- Eine **Task** ist eine einzelne, sequentielle Folge von Programmschritten in Ausführung
  - Synonyme: *Thread* („Ausführungsfaden“), leichtgewichtiger Prozess
- Anmerkungen
  - Klassische sequentielle Programmiersprachen (Fortran, C, C++, COBOL) ermöglichen nur die Programmierung einer Task (d.h. Prozess = Task)
  - Moderne Programmiersprachen (Ada, Java, C#) ermöglichen es, in einem Programm (= Prozess) mehrere Tasks zu erzeugen und zu starten
  - Moderne Betriebssysteme stellen die dafür benötigten Dienste in APIs zur Verfügung

Prozesse und Tasks

- ***Multiprogramming/Multitasking*** [Nach Burns und Wellings 2009]
  (Nebenläufige Verarbeitung)
  - Mehrere Tasks werden auf einem Prozessor ausgeführt (quasi-parallel, nebenläufig)
  - Festzulegen, wann welche Task den Prozessor „bekommt“ ist eine wesentliche Aufgabe des Betriebssystems (→ *Task Scheduling*)
  - Bei Echtzeitsystemen sind hierbei die Zeitbedingungen zu beachten (→ *Real-Time Task Scheduling*)
- ***Multiprocessing*** (Parallelverarbeitung)
  - Die Ausführung mehrere Tasks wird auf mehrere Prozessoren verteilt
  - Die Prozessoren können auf einen gemeinsamen Speicher zugreifen
  - Falls die Zahl der Prozessoren kleiner ist als die der Tasks, ist auch hier (Echtzeit-) Scheduling erforderlich (→ Multitasking)
- ***Distributet Processing*** (Verteilte Verarbeitung)
  - Die Ausführung mehrerer Tasks wird auf mehrere Prozessoren verteilt, die auf keinen gemeinsamen Speicher zugreifen können

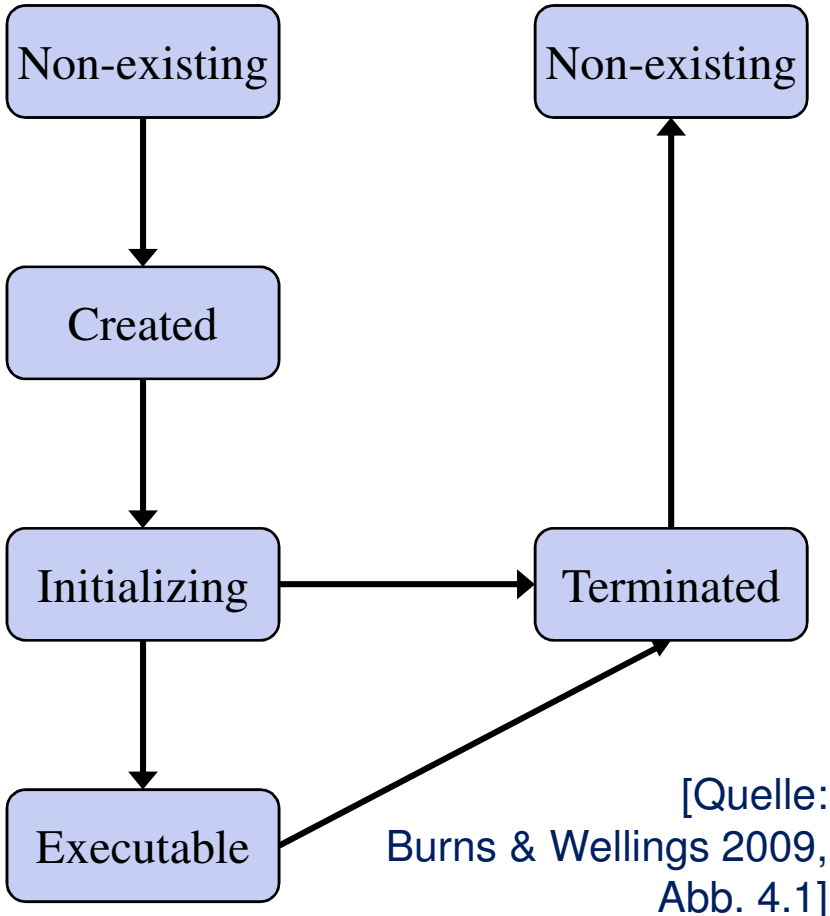
Prozesse und Tasks

- Multitasking und Multiprocessing
  - „Quasi-Parallelverarbeitung“ = Nebenläufigkeit



[Quelle: Wörn & Brinkschulte 2005, Abb. 5.7]

- **Task-Zustände**
  - Beachte:
    - Tasks können während der Initialisierung fehlschlagen
    - *Executable:* die Task ist bereit zur Ausführung, falls ein Prozessor dafür zur Verfügung steht (nicht unbedingt *executing*)
    - Es kann Tasks geben, die nicht terminieren (d.h. sie bleiben im Zustand *executable*)
    - Die Task-Verwaltung, d.h. die Ausführung der Zustandswechsel ist Aufgabe eine **Laufzeitsystems** (*Run-Time Support System, RTSS, Run-Time Kernel*)



[Quelle: Burns & Wellings 2009, Abb. 4.1]

    - Wenn mehre Tasks im Zustand *executable* sind, muss das Laufzeitsystem auch organisieren, wann welche Task den Prozessor bekommt (→ *Task-Scheduling*)

Tasks in Programmiersprachen

- Erforderliche **Programmier-Konstrukte** für Nebenläufigkeit
  - Definition nebenläufiger Aktivitäten (Tasks/Threads/Prozesse)
  - Synchronisationsmechanismen zwischen nebenläufigen Tasks
  - Kommunikationsmechanismen zwischen nebenläufigen Tasks
- Interaktionsverhalten
  - **Unabhängige Tasks** (*independent*):
    - Keine Kommunikation
    - Keine Synchronisation
  - **Kooperierende Tasks** (*cooperating*)
    - Kommunizieren und synchronisieren ihre Aktivitäten, um eine gemeinsame Aufgabe zu erfüllen
  - **Konkurrierende Tasks** (*competing*)
    - Müssen sich Ressorucen teilen
      - z.B. E/A-Geräte, Sensoren, Aktoren, …

Tasks in Programmiersprachen

- Task-Struktur
  - **Statische Tasks:** Die Anzahl der Tasks (zur Laufzeit!) ist zur Übersetzungszeit bekannt und fest
  - **Dynamische Tasks:** Tasks können zur Laufzeit erzeugt werden
- Task-Ebenen
  - **Geschachtelte Tasks:** Tasks können in anderen Programmelementen enthalten sein, insbesondere können Tasks andere Tasks als Sub-Strukturen enthalten
  - **Flache Tasks:** Tasks gibt es nur auf der obersten Ebene der Programmstruktur
- Beispiele:

Sprache	*Task-Struktur*	*Task-Ebenen*
Concurrent Pascal	statisch	flach
Occam2	statisch	geschachtelt
C/POSIX	dynamisch	flach
Ada, Java, C#	dynamisch	geschachtelt

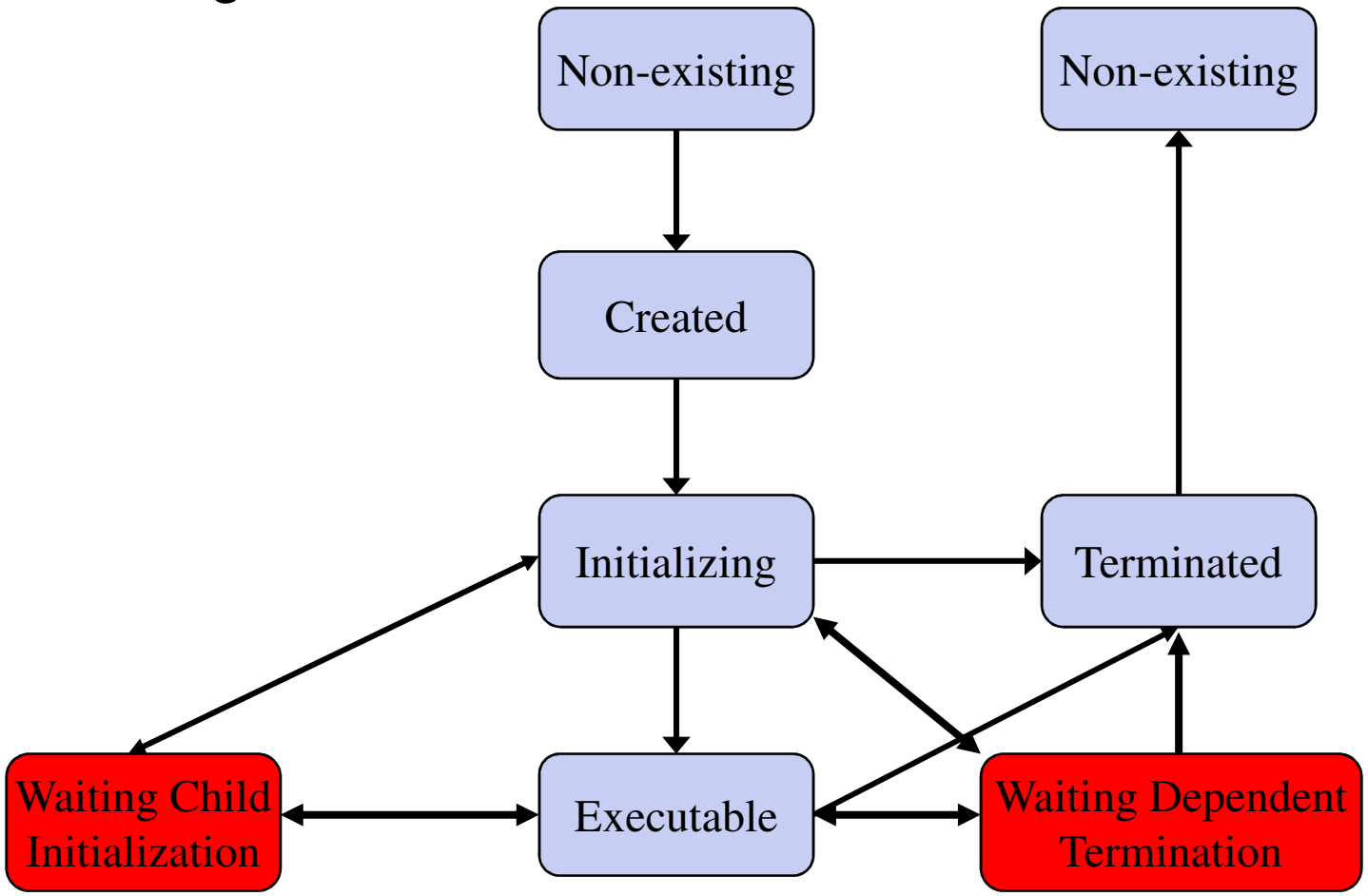
Tasks in Programmiersprachen

- Granularität der Nebenläufigkeit
  - **Grobgranular:** wenige Tasks, lange Task-Dauer
    - z.B. neue Tasks definieren und starten (Ada, C/POSIX, Java, C#)
  - **Feingranular:** viele Tasks, kurze Task-Dauer
    - z.B. Parallel-Anweisungen (Occam2)
- Initialisierung von Tasks
  - Über Parameter (analog Funktionen)
  - Explizite Kommunikation nach Start
- Terminierung von Tasks
  - wenn der Task-Rumpf (Funktionsrumpf, *body*) terminiert
  - wenn eine Terminierungs-Anweisung ausgeführt wird („Suizid“)
  - Abbruch durch eine explizite Aktion eines anderen Tasks („Mord“)
  - Auftreten einer unbehandelten Fehlersituation
  - nie (Tasks sind Endlosschleifen)
  - wenn sie nicht länger gebraucht werden

- Geschachtelte Tasks
  - Falls Schachtelung möglich ist, können Hierarchien von Tasks erzeugt werden
  - Beziehungen:
    - **Elter/Kind** (*parent/child*):
      - Elter(nteil) ist verantwortlich für die Erzeugung des Kinds
      - Die Ausführung des Elters kann verzögert werden, während das Kind initialisiert wird
    - **Wächter/Abhängiger** (*guardian/dependant*):
      - Ein Wächter ist beeinflusst von der Terminierung der abhängigen Tasks
      - Abhängigkeit besteht vom Wächter selbst oder von einem inneren Block
      - Ein Wächter darf einen Block erst verlassen, wenn alle von dem Block abhängigen Tasks terminiert haben
      - Ein Wächter kann erst terminieren, wenn alle abhängigen Tasks terminiert haben
      - Ein Elter einer Task kann gleichzeitig Wächter sein
  - Durch die Beziehungen können sich neue Task-Zustände ergeben

Tasks in Programmiersprachen

- Task-Zustände bei Elter/Kind- und Wächter/Abhängigkeit - Beziehungen



[Quelle: Burns & Wellings 2009,Abb. 4.2]

Tasks in Programmiersprachen

- Programmierkonstrukte für Nebenläufigkeit
  - **Fork and Join:**
    - Ein Fork-Aufruf spezifiziert eine Routine, die nebenläufig zum Aufrufer gestartet wird (Der Fork-Aufrufer wird zum Elter)
    - Ein Join-Aufruf lässt den Aufrufer warten, bis die nebenläufige Routine beendet ist (der Join-Aufrufer wird zum Wächter).

```
function F return is ...;
procedure P;
   ...
   C:= fork F; -- F wird nebenläufig zu P gestartet
   ...
   J:= join C; -- P muss Terminierung von F abwarten
   ...
end P;
```

      - Bei `join C` kann es sein, dass F noch nicht fertig ist (d.h. P muss noch warten)
      - Bei der Terminierung von F kann es sein, dass `join C` noch nicht erreicht ist (d.h. F muss noch warten).
    - Fork and Join gibt es z.B. in Real-Time Posix

Tasks in Programmiersprachen

- Programmierkonstrukte für Nebenläufigkeit
  - **Cobegin (parbegin/par):**
    - Nebenläufige Ausführung einer Menge von Anweisungen

```
cobegin
  S1;
  S2;
  S3;
  .
  .
  .
  Sn;
coend;
```

      - `S1,…,Sn` werden nebenläufig ausgeführt
      - Die Anweisung ist terminiert, wenn alle S_i terminiert haben
      - Jedes S_i kenn eine beliebige Anweisung der Sprache sein

    - Cobegin gibt es in frühen nebenläufigen Programmiersprachen wie Concurrent Pascal oder Occam2

Tasks in Programmiersprachen

- Programmierkonstrukte für Nebenläufigkeit
  - **Explizite Taskdeklaration:**
    - Eine Task wird durch eine entsprechende Deklaration als ein Ablauf deklariert, der nebenläufig zu allem anderen auszuführen ist.

```
task body Thread is
begin
   . . . // Anweisungen der Task
end;
```

    - Dadurch ist nicht deklariert, wann die Task gestartet wird!
    - Sprachen mit expliziter Task-Deklaration können Tasks explizit oder implizit erzeugen
    - Heute Standard in Programmiersprachen mit Nebenläufigkeit (C/Real-Time POSIX, Ada, Java)

Tasks in Programmiersprachen

- Zusammenfassung (Variationen von möglichen Taskmodellen)
  - Struktur
    - statische oder dynamische Tasks
  - Ebene
    - geschachtelt, flach
  - Initialisierung
    - mit oder ohne Parameterübergabe
  - Granularität
    - fein- oder grobgranular
  - Terminierung
    - Natürlich (wenn sie fertig ist), Selbst-Terminierung („Suizid“), Abbruch durch andere („Mord“), unbehandelte Ausnahme, nie, wenn sie nicht mehr gebraucht wird
  - Programmierkonstrukte für Nebenläufigkeit
    - Fork/Join
    - Cobegin
    - Explizite Task-Deklaration

Tasks in Ada

- Task-Deklaration
  - Tasks können als Typen deklariert werden (→ mehrere Instanzen derselben Task können erzeugt werden)
  - Tasks können als einmalige Instanzen ohne Typ deklariert werden („anonyme Tasks“)
  - Eine Task besteht aus einer Spezifikation und einem Rumpf (*body*)
    - diese sind syntaktisch getrennt
  - Die Spezifikation definiert u.a. einen Task-/Tasktypnamen und Parameter zur Initialisierung (optional)
- Erzeugung von Tasks (d.h. sie starten nebenläufig)
  - Implizit, sobald der Gültigkeitsbereich ihrer Deklaration betreten wir
  - Explizit durch Verwendung von „access“-Typen (Pointer) und den „new“-Operator

- **Beispiel**: Implizit erzeugte, anonyme Tasks [Burns & Wellings 2009]

```
procedure Example1 is
  task A;
  task B;
  task body A is
    -- local declarations for task A
  begin
    -- sequence of statement for task A
  end A;

  task body B is
    -- local declarations for task B
  begin
    -- sequence of statements for task B
  end B;
begin
        -- tasks A and B start their executions before
        -- the first statement of the procedure's sequence
        -- of statements.
        ...
end Example1; -- the procedure does not terminate until
              -- tasks A and B have terminated
```

Deklarationsteil von *Example1*

Anweisungsteil (Block) von *Example1*

- **Beispiel**: Task-Typen

```
declare  -- Beginn eines (anonymen) Blocks:
         -- erst die Deklarationen

   task type A_Type;
   task type B_Type;

   A: A_Type;
   B: B_Type;

   task body A_Type is ... begin ... end A_Type;
   task body B_Type is ... begin ... end B_Type;

   -- Typdeklaration: Ein Feld von Tasks
   type ATaskField is array(1..100) of A_Type;

   -- Typdefinitioin: Ein Verbund (Struktur)
   type Mixture is record
     Index: Integer;
     Action: B_Type;  -- eine Task als Komponente
   end record;

   F: ATaskField;
   M: Mixture;

begin  ... end;  -- Anweisungen des Blocks
```

- **Beispiel:** Roboterarm
  - Ein Roboterarm kann sich in 3 Dimensionen (X,Y,Z) bewegen
  - Die Bewegung wird in jeder Dimension durch einen eigenen Motor ausgeführt
  - Jeder Motor wird gesteuert durch eine separate Ada-Task
    - Es gibt einen Tasktyp `Control` mit einem Parameter für `Dim` für die Dimension
    - Die Task besteht aus einer Endlosschleife
      (zu Beginn ist die aktuelle Position eine Grundposition)
      - Bewege Roboterarm an die aktuelle Position
      - Bestimme relative Bewegung
      - Aktuelle Position = aktuelle Position + relative Bewegung
    - Es werden 3 Tasks dieses Typs für die Dimensionen X, Y, und Z nebenläufig gestartet
  - Prozeduren zum bewegen des Roboterarms und zum Bestimmen der relativen Bewegung (erforderliche Positionsveränderung)
    - `Move_Arm(D: Dimension; C: Integer);`
    - `New_Setting(D: Dimension) return Integer;`

Tasks in Ada

- **Beispiel:** Roboterarm

```
package Arm_Support is

type Dimension is
       (Xplane, Yplane, Zplane);

procedure Move_Arm(D: Dimension;
                   C: Integer);
--- Moves the arm to C

function New_Setting(D: Dimension;)
                  return Integer;
--- Returns a new required
--- relative position

end Arm_Support;
```

```
with Arm_Support; use Arm_Support;
Procedure main is
  task type Control(Dim : Dimension);
  C1 : Control(Xplane);
  C2 : Control(Yplane);
  C3 : Control(Zplane);
  task body Control is
    Position : Integer; -- absolute position
    Setting : Integer;  -- relative movement
  begin
    Position := 0;      -- rest position
    loop
      Setting = New_Setting (Dim);
      Position := Position + Setting;
      Move_Arm (Dim, Position);
    end loop;
  end Control;
begin
  null;
end Main;
```

- **Beispiel:** Explizit erzeugte Tasks [Burns & Wellings 2009]
  - *Hinweis:*
    - Pointer werden in Ada mit dem Schlüsselwort **`access`** deklariert;

```
type PointerTyp is access BezugsTyp;
```

    - Wenn `Q` eine Task-Access-Variable (Task-Pointer) ist, bezeichnet `Q.all` die Task, auf die `Q` zeigt.

```
procedure Example2 is
  task type T; task body T is ...
  type TPtr is access T;  -- Typ TPtr ist "Zeiger auf" T
  P : TPtr;
  Q : TPtr := new T;
  . . .
begin
  ...             -- Q.all wird gestartet (Task vom Typ T)
  P := new T;     -- P.all wird gestartet (Task vom Typ T)
  Q := new T;
                  -- Q.all ist neu, das "alte" Q.all läuft
                  -- als anonyme Task weiter!
  ...
end Example2;
```

Tasks in Ada

- Wann terminieren Tasks?
  - Wenn die Anweisungen des Rumpfs beendet sind
  - Wenn ein spezielles „Terminate“-Statement erreicht wird (Spezielle Alternative in einer Select-Anweisung, siehe später)
  - Eine Task kann alle anderen Tasks in ihrem Gültigkeitsbereich abbrechen
    - Von diesen abhängige Tasks werden ebenfalls abgebrochen
- Eltern und Wächter/Master – Blöcke (vgl. 2-17)
  - Elter (**parent**) einer Task C, ist die Task P, die C erzeugt hat; C ist Kind (child) von P
    - P muss während der Aktivierung von C warten
  - Der **Master-Block** M einer abhängigen (*dependent*) Task D kann erst beendet werden, wenn D beendet ist
    - Ein Master block entspricht einem „Wächter“
  - In vielen Fällen sind Elter und Master dieselben Tasks
    - Aber: ein Master muss keine Task sein, es kann ein beliebiger Block sein

- Master – Blöcke
  - Beispiel: Interne Masterblöcke (anderen Tasks untergeordnet)
    - Die Task die den internen Masterblock ausführt, erzeugt die abhängige Task `Dependant` und ist damit Elter (*parent*).
    - Nicht die Eltern-Task, sondern der interne Masterblock kann nicht beendet werden, bevor `Dependant` nicht beendet ist.

```
declare -- interner MASTER block
   -- Lokale Deklarationen
   task Dependant;
   task body Dependant is begin ... end;
begin -- MASTER block
   ...
end; -- MASTER block
```

Tasks in Ada

- Master – Blöcke
  - Beispiel: Dynamische Tasks
    - Elter ist die Task , die die Task erzeugt hat (d.h. die die new-Anweisung enthält)
    - Master ist der Block, dessen Deklarationsteil den Zeiger-Typ definiert!

```
declare
   task type Dependant;
   type Dependant_Ptr is access Dependant;
   A : Dependant_Ptr;
   task body Dependant is begin ... end;
begin
   ...
   declare
      B : Dependant;
      C : Dependant_Ptr := new Dependant;
   begin
      A := C;
   end;
end;
```

Hier muss gewartet werden, bis `B` terminiert, aber nicht C.all;
C.all kann noch aktiv sein, obwohl C.all nicht mehr sichtbar ist;
Auf die Task kann immer noch über A zugegriffen werden.

Tasks in C/Real-Time POSIX

- In einem Betriebssystem-Prozess (Programm in Ausführung) können mehrere ***Threads*** enthalten sein.
  - Alle Threads laufen im selben Adressraum
  - Alle Threads können auf denselben Speicherbereich zugreifen
  - Vergleichbar mit Ada-Tasks
  - Programmierung und Bearbeitung von Threads über API-Funktionen
    → Schnittstellen siehe [*Beispiele zu Kapitel 2*]
    → `#include <pthread.h>`
- „Concurrency-Level“:
  - Wie viele Threads kann der Betriebssystem-Kern nebenläufig ausführen?
  - Wird in der Regel dem Betriebssystem überlassen!
  - Info an das Betriebssystem welches Concurrency-Level benötigt wird: `pthread_setconcurrency(int level)`
    - Muss aber nicht vom BS berücksichtigt werden!

Tasks in C/Real-Time POSIX

- Thread-Datentypen
  - **`pthread_t`**: Jeder Thread wird identifiziert durch einen Identifikator dieses Typs (definiert als Ganzzahl, „Handle“)
    - Ein Thread kann seine eigene ID bestimmen:

```
pthread_t pthread_self(void);
```

  - **`pthread_attr_t`**: Struktur, die verschiedene Thread-Attribute beinhaltet
    - Beispiele für Thread-Attribute:
      - *detach state*: legt fest, ob Thread-Daten über die Terminierung des Threads hinaus zugreifbar sind (default) oder nicht
      - *stack size* (in Bytes, default = 1 MB)
      - *sceduling policy* und *sceduling parameter* (später mehr!)
    - Die Attribute eines Threads können durch API-Funktionen beeinflusst werden (vgl. Beispiel 1), z.B.:

```
int pthread_attr_init(pthread_attr_t *attr);
```

→ *Initialisieren mit Default-Werten*

```
int pthread_attribute_setstacksize(pthread_attr_t *attr,
                                   size_t stacksize);
```

Tasks in C/Real-Time POSIX

- Erzeugen von Threads

```
int pthread_create(
        pthread_t *thread,
        pthread_attr_t *attr,
        void *(*start_routine)(void *),
        void *arg);
```

  - `thread` zeigt nach Aufruf auf die ID des erzeugten Threads
  - `attr` zeigt auf die vorgesehenen Thread-Attribute
  - `start_routine` zeigt auf die Funktion, die der Thread ausführen soll
  - `arg` zeigt auf ein Feld von Argumenten, die der Start-Funktion übergeben werden sollen.

- Erzeugen von Threads – Beispiel

```
pthread_attr_t tattr; /* Variable für Thread-Attribute */
pthread_t tid;        /* Variable für Thread-ID */

/* Diese Funktion soll als Thread gestartet werden */
void *start_routine(void *arg) { ... } ;

void *arg; /* sollte auf die Argumente der
              Start-Routine zeigen */
int ret;   /* Für Rückgabewerte */

/* Thread-Attribute initialisieren mit default-Werten */
ret = pthread_attr_init(&tattr);

/* Thread erzeugen und starten */
ret = pthread_create(&tid, &tattr, start_routine, arg);
```

Tasks in C/Real-Time POSIX

- Terminieren von Threads
  - Natürlich: nach Rückkehr aus der Start-Funktion
  - Mord: ein Thread kann von außen beendet werden mit

```
int pthread_cancel(pthread_t thread);
```

  - Suizid: ein Thread kann sich selbst beenden durch Aufruf von

```
void pthread_exit(void *staus_ptr)
```

Beispiel:

```
int status = ... ; /* Status-Wert schreiben */
pthread_exit(&status); /* Beenden mit Status-Übergabe */
```

- Warten auf Threads: **Join**

```
int pthread_join(thread_t tid, void **status);
```

  - Der aufrufende Thread wird angehalten, solange bis der Thread mit der ID `tid` terminiert
  - Falls der Thread `tid` mit `pthread_exit(&status)` beendet wurde, steht in `status` ein Pointer auf den von diesem geschriebenen Status-Wert

- Threads „aufräumen“
  - Wann wird der von einem Thread belegete Speicherplatz inklusive aller Thread-Informationen freigegeben („Detached“)?
  - Abhängig vom *Detach-State* (→ Thread-Attribute)
    - **Detached**: unmittelbar nach Terminierung
      - Nach Terminierung des Threads sind Thread-Informationen nicht mehr verfügbar
      - Kein Join auf den Thread mehr möglich!
    - **Nondetached** (**joinable**):
      - nach einem Join auf dem Thread und Terminierung des Threads

- Threads aufräumen
  - Festlegen des Detach-Status bei Task-Erzeugung
    - Default-Detach-Status ist *non-detached (joinable)*
    - Makros für den Detach-Status

```
PTHREAD_CREATE_DETACHED
PTHREAD_CREATE_JOINABLE
```

    - Explizites setzten des Detach-Status bei Thread-Erzeugung über Thread-Attribute:

```
ret = pthread_attr_init(&tattr);
ret = pthread_attr_setdetachstate(&tattr,
                                  PTHREAD_CREATE_DETACHED);
ret = pthread_create(&tid, &tattr, start_routine, arg);
```

    - Dynamisch (zur Laufzeit des Threads) Detach-Status auf *detached* setzen:

```
ret = pthread_detach(tid)
```

- **Beispiel:** Roboterarm (vgl. S. 2-27)

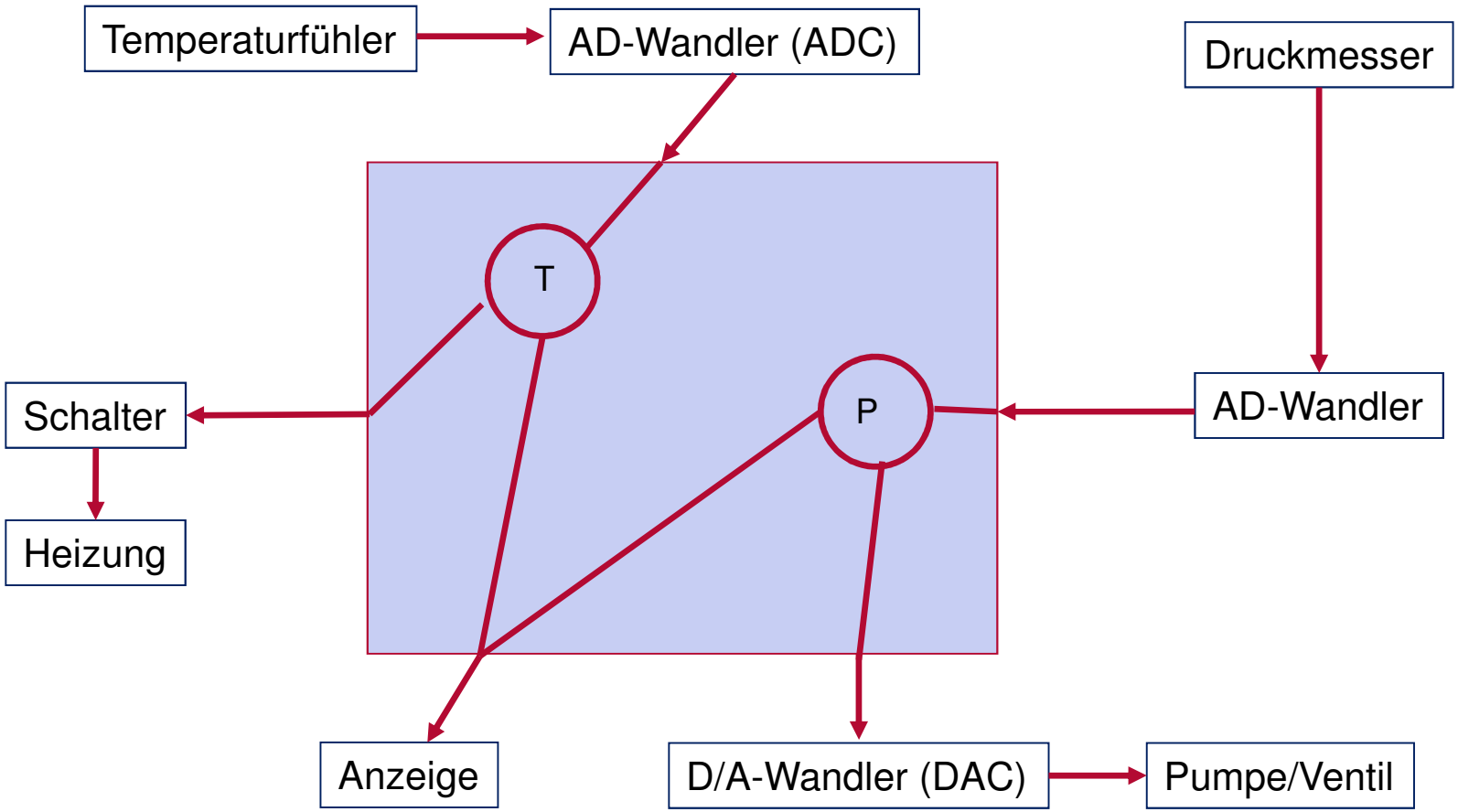
```
#include <pthread.h>

pthread_attr_t attributes;
pthread_t xp, yp, zp;

typedef enum {xplane, yplane, zplane} dimension;

int new_setting(dimension D);
void move_arm(dimension D, int P);

void controller(dimension *dim) { /* function to start als threads */
  int position, setting;

  position = 0;
  while (1) {
    move_arm(*dim, position);
    setting = new_setting(*dim);
    position = position + setting;
  }
  /* note, no call to pthread_exit, process does not terminate */
}
```

- **Beispiel:** Roboterarm

```
#include <stdlib.h>
int main() {
  dimension X, Y, Z;  void *result;

  X = xplane; Y = yplane; Z = zplane;

  if(pthread_attr_init(&attributes) != 0)
    /* set default attributes */
    exit(EXIT_FAILURE);
  if(pthread_create(&xp, &attributes, (void *)controller, &X) != 0)
    exit(EXIT_FAILURE);
  if(pthread_create(&yp, &attributes,(void *)controller, &Y) != 0)
    exit(EXIT_FAILURE);
  if(pthread_create(&zp, &attributes,(void *)controller, &Z) != 0)
    exit(EXIT_FAILURE);
  pthread_join(xp, (void **)&result); /* block main program */

  exit(EXIT_FAILURE); /* error exit, program should not terminate */
}
```

Beispiel

- Ein einfaches eingebettetes System [Burns & Wellings 2005, Kap. 4.8]
  - Ziel: Temperatur und Druck in einem chemischen Prozess in definierten Grenzen halten.



Beispiel

- Mögliche Software-Architekturen
  - Sequentielle Lösung:
    - Ein einziges Programm, welches die logische Nebenläufigkeit von T und P ignoriert

    →Keine Unterstützung durch ein Betriebssystem erforderlich
  - Nebenläufigkeit mit Betriebssystem-Aufrufen
    - T und P werden in einer sequentiellen Programmiersprache geschrieben (als separate Programme oder separate Funktionen in einem Programm)
    - Betriebssystem-Aufrufe werden benutzt um T und P nebenläufig zu starten
  - Nebenläufigkeit in der Programmiersprache
    - Ein einziges, nebenläufiges Programm, das die logische Struktur von T und P erhält

    →Keine Betriebssystem-Unterstützung im Programm erforderlich

    →Aber ein entsprechendes Laufzeitsystem, das Nebenläufigkeit unterstützt.

Beispiel

- Hilfspakete (1) – Tatentypen und Ein-/Ausgabe

```
package Data_Types is
   type Temp_Reading is new Integer range 10..500;
   type Pressure_Reading is new Integer range 0..750;
   type Heater_Setting is (On, Off);
   type Pressure_Setting is new Integer range 0..9;
end Data_Types;

with Data_Types; use Data_Types;
package IO is
   procedure Read(TR : out Temp_Reading); -- from ADC
   procedure Read(PR : out Pressure_Reading);
   procedure Write(HS : Heater_Setting);-- to switch
   procedure Write(PS : Pressure_Setting); -- to DAC
   procedure Write(TR : Temp_Reading);  -- to screen
   procedure Write(PR : Pressure_Reading);-- to screen
end IO;
```

Typ-definitionen

Prozeduren für den Daten-Austausch mit der Umgebung

Beispiel

- Hilfspakete (2) – Steuerungs-Prozeduren

```
with Data_Types; use Data_Types;
package Control_Procedures is
   -- procedures for converting a reading into
   -- an appropriate setting for output.
   procedure Temp_Convert(TR : Temp_Reading;
                          HS : out Heater_Setting);
   procedure Pressure_Convert(PR : Pressure_Reading;
                              PS : out Pressure_Setting);
end Control_Procedures;
```

Beispiel

- Sequentielle Lösung

```
with Data_Types; use Data_Types; with IO; use IO;
with Control_Procedures;  use Control_Procedures;
procedure Controller is
   TR : Temp_Reading;    PR : Pressure_Reading;
   HS : Heater_Setting; PS : Pressure_Setting;
begin
   loop
      Read(TR);     -- from ADC
      Temp_Convert(TR,HS); -- convert reading to setting
      Write(HS);    -- to switch
      Write(TR);    -- to screen
      Read(PR);     -- as above for pressure
      Pressure_Convert(PR,PS);
      Write(PS);
      Write(PR);
   end loop; -- infinite loop
end Controller;
```

Beispiele

- Probleme der sequentiellen Lösung
  - Temperatur und Druck werden mit der selben Frequenz gemessen
    - Könnte nicht den zeitlichen Anforderungen entsprechen
    - Kann man durch Verwendung von Zählern und `if`-Anweisungen verbessern
  - Auch dann: Falls Steuerungs-Prozeduren (`Temp_Convert` und `Pressure_Convert`) zu lange dauern, könnte die nächste wichtige Messung zu spät kommen
    - Lösung: die Prozeduren in einzelne Anweisungen auf-splitten und mit den Lese-Anweisungen verzahnen.
  - Während man auf Eingaben wartet (z.B. Temperatur) ist der Rest (z.B. Druck lesen) blockiert
  - Insbesondere: Ein Systemfehler beim Temperatur-Lesen führt dazu, dass kein Druck mehr gemessen wird

Beispiel

- Verbesserte Sequentielle Lösung
  - Zwei zusätzliche Boole'sche Funktionen im Pakckage `IO`:
    - **function** Ready_Temp **return** Boolean **is** ...
    - **function** Ready_Pres **return** Boolean **is** ...

```
procedure Controller is
  TR : Temp_Reading;    PR : Pressure_Reading;
  HS : Heater_Setting; PS : Pressure_Setting;
Begin
  loop
    if Ready_Temp then
      Read(TR); Temp_Convert(TR,HS);
      Write(HS);  Write(TR);
    end if;
    if Ready_Pres then
      Read(PR); Pressure_Convert(PR,PS);
      Write(PS);  Write(PR);
    end if;
  end loop;
end Controller;
```

Beispiel

- Bewertung der verbesserten sequentiellen Version
  - Die Lösung ist zuverlässiger
    - Warten auf Temperaturwerte bremst das Lesen von Druckwerten nicht aus und umgekehrt
  - Problematisch:
    - Das Programm verbringt vermutlich die meiste Zeit in Schleifen, in denen nur die Eingabegeräte abgefragt werden ((„polling“, „busy wait“)
    - „Beschäftigungstherapie“ für den Prozessor: andere Prozesse werden ausgebremst
- **Hauptkritik an der sequentiellen Lösung**
  - Die Tatsache, dass die Druck- und Temperatur-Steuerungszyklen **unabhängig** voneinander ablaufen, wird im Programm nicht angemessen berücksichtigt

  →Druck- und Temperatursteuerung sollten als **nebenläufige Tasks** programmiert werden!

Beispiel

- Nebenläufige Lösung mit Betriebssystemaufrufen
  - Beispielsweise mit einer Real-Time POSIX-ähnlichen ADA-Schnittstelle für Betriebssystemaufrufe (***O**perating **S**ystem **I**nterface*)

```
package OSI is
   type Thread_ID is private;
   type Thread is access procedure;

   function Create_Thread(Code : Thread)
                  return Thread_ID;
   -- other subprograms
   procedure Start(ID : Thread_ID);
private
    type Thread_ID is ...;
end OSI;
```

Beispiel

- Nebenläufige Lösung mit Betriebssystem-Aufrufen
  - Ein Package `Processes` für die beiden nebenläufigen Prozesse:

```
package Processes is
   procedure Temp_Controller;
   procedure Pressure_Controller;
end Processes;

with...; -- Data_Types, IO, Control_Procedures
package body Processes is
   procedure Temp_Controller is
      TR : Temp_Reading;  HS : Heater_Setting;
   begin
      loop
         Read(TR); Temp_Convert(TR,HS);
         Write(HS); Write(TR);
      end loop;
   end Temp_Controller;
```

Beispiel

- Nebenläufige Lösung mit Betriebssystem-Aufrufen
  - Ein Package `Processes` für die beiden nebenläufigen Prozesse (Forts.):

```
procedure Pressure_Controller is
  PR : Pressure_Reading;
  PS : Pressure_Setting;
begin
  loop
    Read(PR);
    Pressure_Convert(PR,PS);
    Write(PS);
    Write(PR);
  end loop;
end Pressure_Controller;
end Processes;
```

Beispiel

- Nebenläufige Lösung mit Betriebssystem-Aufrufen
  - Hauptprozedur: erzeugt und startet zwei nebenläufige Prozesse für Temeperatur und Druck:

```
with OSI, Processes; use OSI, Processes;
procedure Controller is
  TC, PC : Thread_ID;
begin
    TC := Create_Thread(Temp_Controller'Access);
    PC := Create_Thread(Pressure_Controller'Access);
   Start(TC);
   Start(PC);
end Controller;
```

  - **Vorteil:** zuverlässiger, Prozesse können sich nicht „ausbremsen“, kein „busy waiting“
  - **Nachteil:** Immer noch nicht gut lesbar!
    - Es ist schwer zu erkennen, welche Prozeduren „normal“ sind, und welche als nebenläufige Tasks gedacht sind.

Beispiel

- Nebenläufige Lösung mit Ada-Tasking

```
with ...
procedure Controller is

   task Temp_Controller;
   task Pressure_Controller;

   task body Temp_Controller is
      TR : Temp_Reading;
      HS : Heater_Setting;
   begin
      loop
         Read(TR);
         Temp_Convert(TR,HS);
         Write(HS);
         Write(TR);
      end loop;
   end Temp_Controller;
```

```
   task body Pressure_Controller is
      PR : Pressure_Reading;
      PS : Pressure_Setting;
   begin
      loop
         Read(PR);
         Pressure_Convert(PR,PS);
         Write(PS);
         Write(PR);
      end loop;
   end Pressure_Controller;

begin
   null;   -- Temp_Controller and
           -- Pressure_Controller
           -- have started their
           -- executions
end Controller;
```

Fazit

- Gründe für **Nebenläufigkeit als Programmiersprachen-Konzept**
  - Programm sind leichter lesbar und wartbar
  - Unterschiedliche Betriebssysteme realisieren Nebenläufigkeit auf verschiedene Arten → Nebenläufigkeit in der Programmiersprache macht Programme portabler
  - Manche eingebetteten Prozessoren haben gar kein Betriebssystem
- Gründe für **Nebenläufigkeit mit Hilfe von Betriebssystem-Aufrufen**
  - Unterschiedliche Sprachen realisieren Nebenläufigkeit auf verschiedene Arten → Programme aus unterschiedlicher Sprachen können einfacher auf einem Betriebssystem integriert werden.
  - Es könnte schwierig sein, das Nebenläufigkeitskonzept einer Programmiersprache auf einem BS effizient zu implementieren
  - Es gibt heute Betriebssystem-Standards wie POSIX, die ebenfalls zu portablen Programmen führen

Literatur

[Burns & Wellings 2009] Alan Burns, Andy Wellings: *Real-Time Systems and Programming Languages. Ada, Real-Time Java and C/Real-Time POSIX*. Addison Wesley, 2009.

[Wörn & Brinkschulte 2005] Heinz Wörn, Uwe Brinkschulte: *Echtzeitsysteme*. Springer, 2005.

[Zöbel 2008] Dieter Zöbel: *Echtzeitsysteme. Grundlagen der Planung*. Springer, 2008.