Inhalt

- Sperrsynchronisation
- Verklemmungen
- Reihenfolgensynchronisation mit Bedingungsvariablen

Synchronisation

Wenn nebenläufige Programme gemeinsame Betriebsmittel verwenden, muss der Zugriff synchronisiert erfolgen.

Gemeinsame Betriebsmittel sind z.B.:

- Daten (gemeinsame Speicherbereiche, DB-Records, Dateien, ...)
- Geräte (I/O-Geräte, Sensoren/Aktoren, ...)
- Software (Treiber, Collection-Klassen, ...)
  - ➔ siehe *thread safety*

Es wird unterschieden zwischen:

- **Sperrsynchronisation** und
- **Reihenfolgensynchronisation**

Sperrsynchronisation

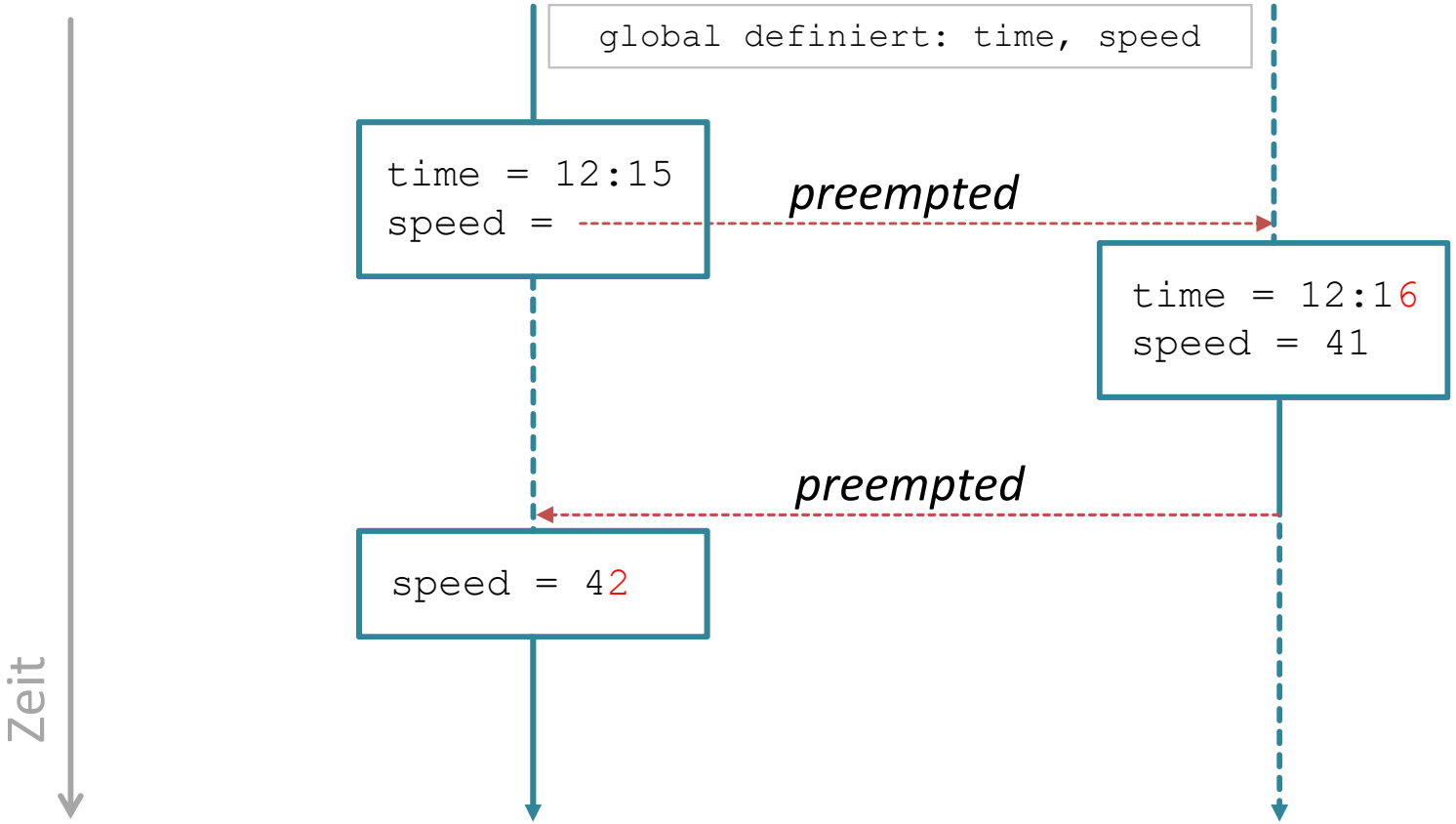
(auch: **wechselseitiger Ausschluss**, **mutual exclusion**)

- Die Sperrsynchronisation stellt sicher, dass ein bestimmter Programmabschnitt nur von einer nebenläufigen Programmeinheit ohne Unterbrechung* („atomar“) ausgeführt wird.
- Weitere Zugriffe auf diesen **kritischen Abschnitt** (critical section) werden blockiert und gepuffert.
- Die Abarbeitung des Puffers erfolgt systemabhängig, z.B. entsprechend dem aktuellen Scheduling.
- Die Sperrsynchronisation definiert keine Abarbeitungsreihenfolge.

*: der Prozess/thread kann durchaus unterbrochen werden

Problemstellung zur Sperrsynchronisation

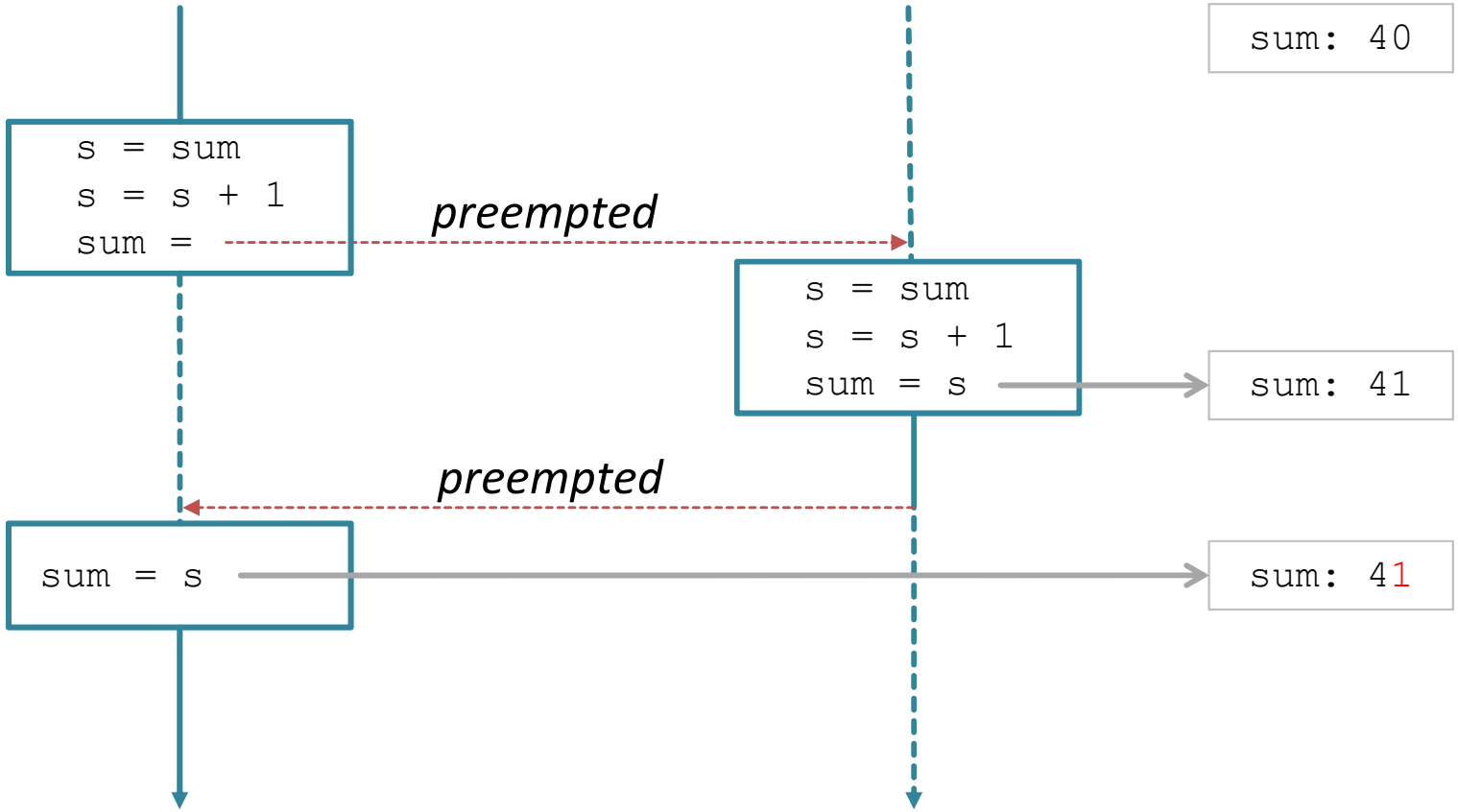
a) mehrere nebenläufige Einheiten können Datensätze (*time* und *speed*) in gemeinsamen Speicher schreiben:



Ergebnis:
time = 12:16
speed = 42

Der resultierende Datenbestand ist inkonsistent (*nicht zusammen passend*).

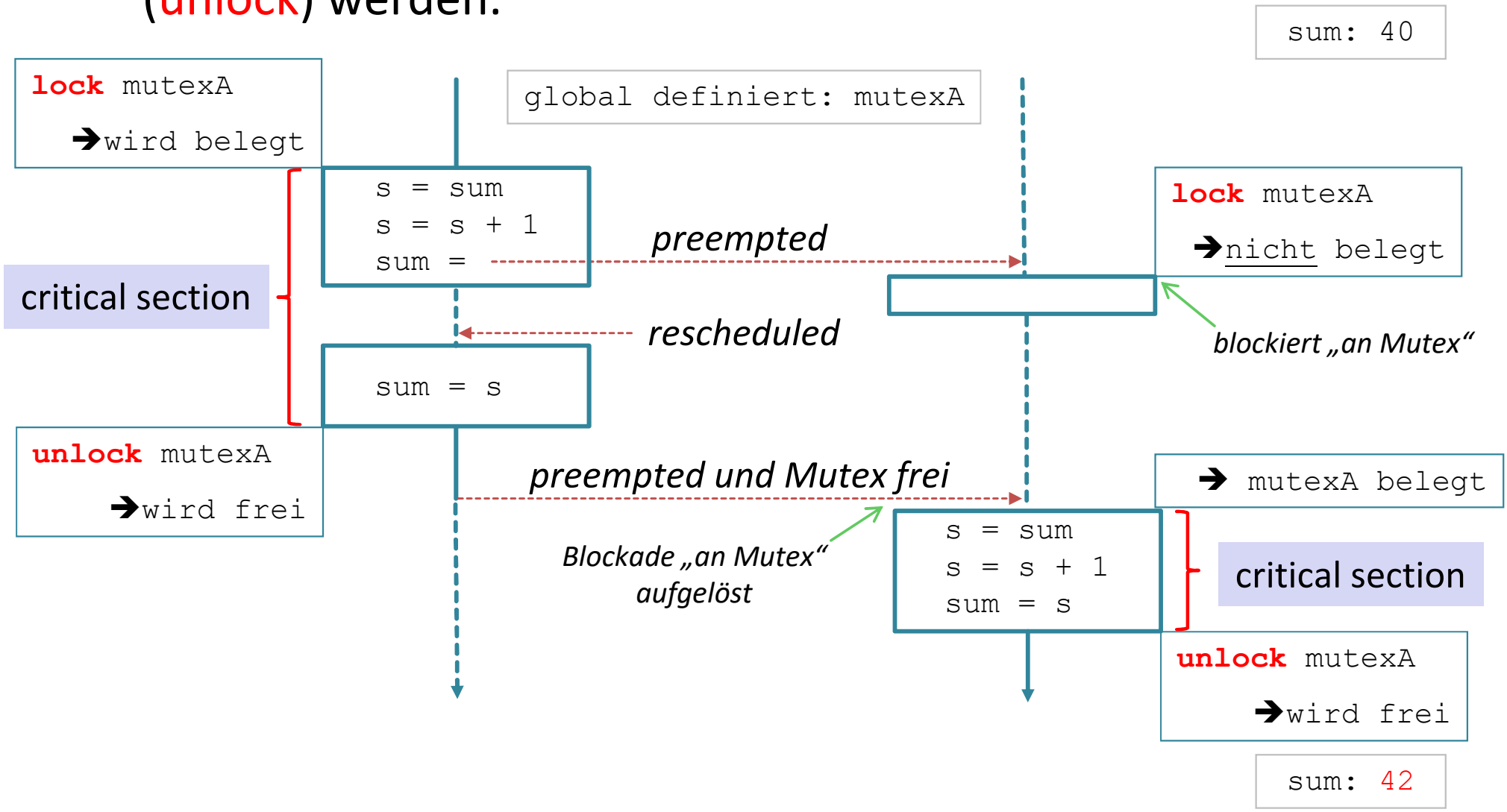
b) mehrere nebenläufige Einheiten lesen und schreiben gemeinsame Datenbereiche (globale Variable *sum*):



Das Ergebnis hängt von der zeitlichen Abfolge der Bearbeitung ab (race condition).

Lösung

Der kritische Abschnitt wird mit einer „Sperrvariablen“ (Mutex) geschützt. Ein Mutex kann belegt (lock) und freigegeben (unlock) werden.



- Sperrvariablen (Mutexe) sind selbst kritische Abschnitte und können nicht mit „Bordmitteln“ selbst erstellt werden (Interruptsperren, test-and-set, ...).
- Sperrvariablen sind nur Vereinbarungen.
- Es wird keine tatsächliche Sperre errichtet.
- Der wechselseitige Ausschluss ist nur gewährleistet, wenn alle den Vereinbarungen folgen:
  ➔ ohne *lock-Anforderung* besteht immer Zugriff

Probleme mit Sperrvariablen:

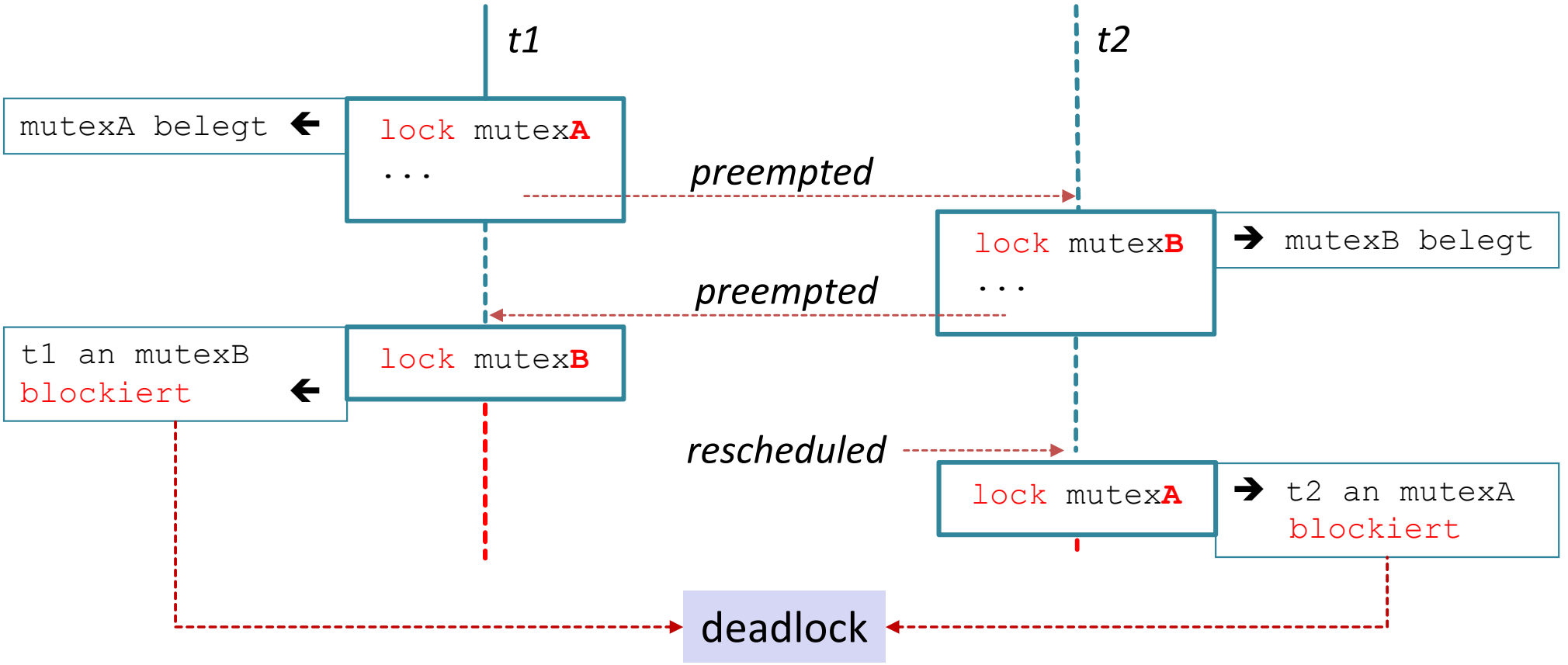
- lock/unlock wird leicht vergessen
- die Fehler treten evtl. nur sporadisch auf
- fehlerhafte lock/unlock Aufrufsequenzen führen zu Verklemmungen (deadlocks).

Verklemmungen (deadlocks)

- Eine Verklemmung liegt vor, wenn nebenläufige Aktivitäten auf die Freigabe von resources (z.B. Mutexe) warten, die nur von den Wartenden selbst freigegeben werden können.
- Vermeiden kann man Verklemmungen durch „sauberes Softwaredesign“ (Petri-Netze)
  ➔ leider ist das Verfahren bzw. die Modellierung selbst sehr komplex und damit fehleranfällig.
- Betriebssysteme bzw. system calls können nur beschränkt deadlocks erkennen.
- Bei sicherheitskritischen Systemen werden watch-dog Schaltungen eingesetzt.

Typische deadlock-Situation

Zwei threads t1 und t2 verwenden zwei Mutexe in unterschiedlicher Folge:

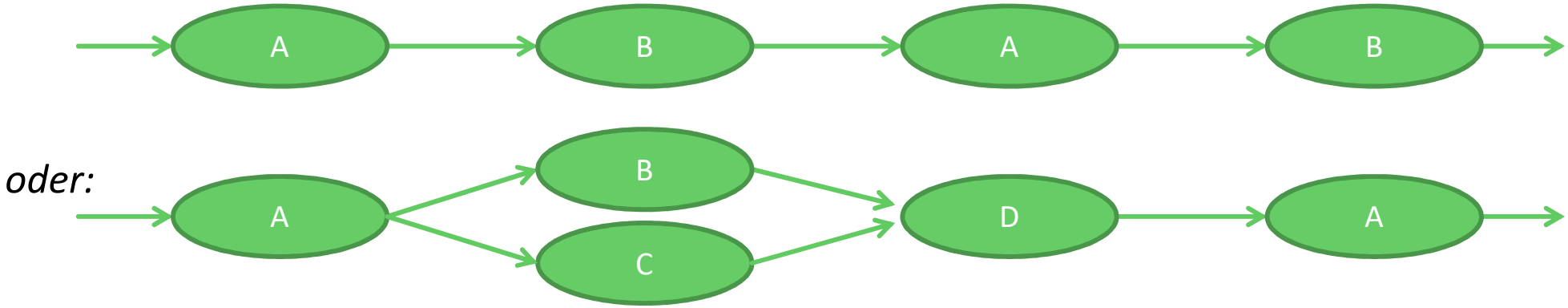


mehrere Mutexe:
➔ immer identische lock/unlock Sequenz verwenden

Demo: V04Beispiele `no01`, `no02`, `no03`

Reihenfolgensynchronisation (Kooperation)

- Anders als bei der Sperrsynchronisation wird hier die Reihenfolge nebenläufiger Aktivitäten erzwungen.
- Nebenläufige Aktivitäten kooperieren z.B.:
  - wenn sie auf Daten anderer Aktivitäten warten
  - wenn sie auf gemeinsame zeitliche Bedingungen warten
- Sprachmittel sind: Bedingungsvariablen, Semaphore, ...
- Abhängigkeitsdiagramme veranschaulichen kooperative Aktivitäten, z.B.:



A, B, ... : nebenläufige Aktivitäten

Bedingungsvariablen (condition variables)

Es wird gewartet (*condition-wait*), bis eine beliebige global definierte Bedingung von einer anderen nebenläufigen Aktivität erfüllt ist (*condition-signal*).

➔ Bedingungsvariablen erfordern einen **Mutex**

Prinzip:

```
global definiert: condVar
```

wartende Aktivität:

```
-lock mutex
-teste "globale Bedingung":
    false: warte an condVar
           und unlock mutex
-evtl.Bearbeitung des krit.
 Bereichs
-unlock mutex
```

signalisierende Aktivität:

```
-lock mutex
-bearbeite condVar ...
-teste "globale Bedingung":
    true: Signal an condVar
-evtl.Bearbeitung des krit.
 Bereichs
-unlock mutex
```

POSIX pthread condition variables

- Eine Bedingungsvariable vom Typ **`pthread_cond_t`** hat eine Warteschlange für wartende threads.
- Die Operation **`pthread_cond_wait`** erfordert als Parameter:
  - eine Bedingungsvariable **`condVar`** und
  - einen Mutex **`mutX`**

  wait führt folgende Operation atomisch aus:
  ➔ aufrufenden thread an **`condVar`** blockieren (Warteschlange) UND **`mutX`** freigeben (unlock)
- Die Operation **`pthread_cond_signal(condVar)`** „weckt einen der an **`condVar`** wartenden threads auf“. Dieser kann aber nur im Besitz des Mutex den kritischen Bereich weiter ausführen. Daher:
  signal führt folgende Operation aus:
  ➔ es wird ein thread aus der **`condVar`**-Warteschlange in die Warteschlange des Mutex dieser **`condVar`** überführt.

pthread_cond_wait und pthread_cond_signal

```
global definiert: mutX, condVar, globDat
```

t1

```
lock mutX
while(!globDat)
    wait(condVar, mutX)
...
unlock mutX
```

```
t1 blockiert in
condVar-Queue und
unlock mutX
```

```
t1 von
condVar-Queue in
mutX-Queue
```

t2

```
lock mutX
globDat = ...
if(globDat)
    signal(condVar)
...
unlock mutX
```

Demo: V04Beispiele no04