Grundlagen der Softwareentwicklung II

8. Nebenläufigkeit

Version 1.1

Prof. Dr. Michael Munz
Fakultät für Mechatronik und Medizintechnik
Technische Hochschule Ulm

THU
Technische
Hochschule
Ulm

Lernziele

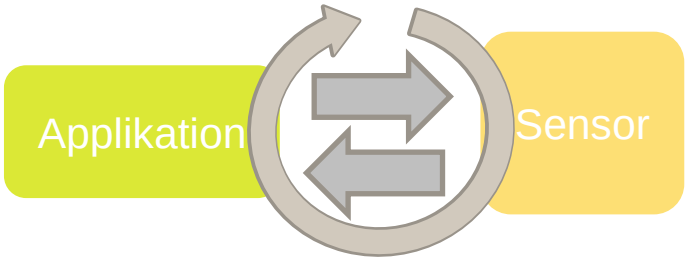
- Bedeutung der Begriffe Nebenläufigkeit und Parallelität kennen
- Fallstricke und Probleme kennen und vermeiden
- Techniken zur sicheren nebenläufigen Programmierung in Python anwenden können
- Einsatzfelder für nebenläufige Programmierung erkennen

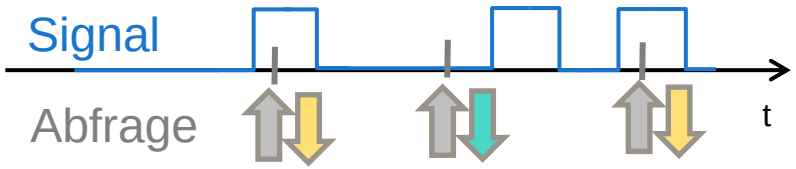
Motivation: Möglichkeiten zum Auslesen von Sensordaten

1. **Regelmäßige Abfrage** (*aktives Warten, polling*)
   → Ständiges Auslesen von Statuswerten

Nachteile:

- Hohe CPU-Last
- Messungen können „verpasst“ werden



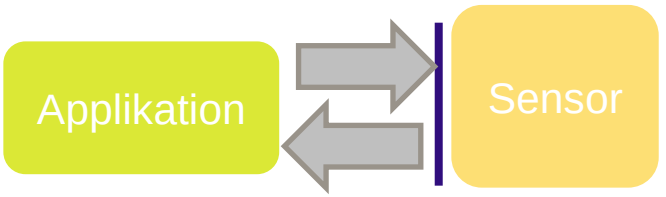


2. **Blockierendes Warten** (*passives Warten*)

→ Leseaufruf kehrt erst zurück, wenn Daten vorliegen

Nachteil:

- Programmausführung wird angehalten



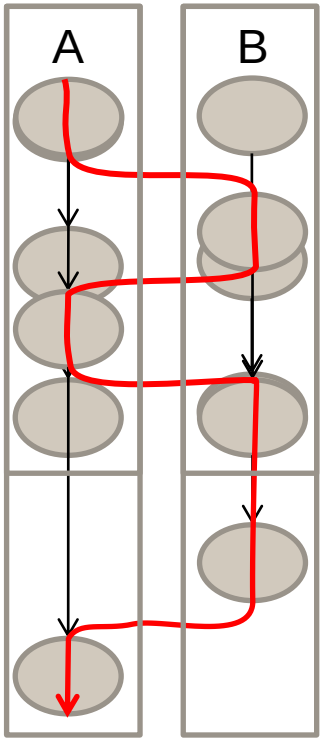
Nebenläufigkeit vs. Parallelität

Definition: Nebenläufigkeit

Zwei Aktivitäten A und B heißen nebenläufig, wenn sie unabhängig voneinander ausgeführt werden können, d.h. wenn sie keine kausale Abhängigkeit voneinander besitzen.

→ Es ist egal, in welcher Reihenfolge die Verarbeitungsschritte von Aktivität A und B ausgeführt werden

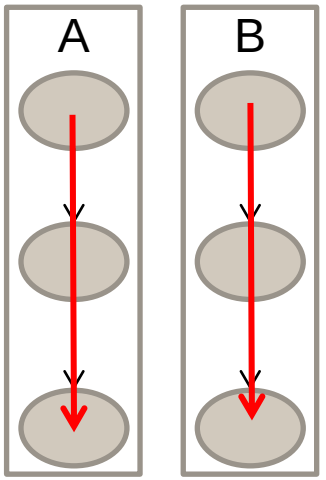
Beispiel: *Auslesen von Sensordaten und Ändern von Parametern*



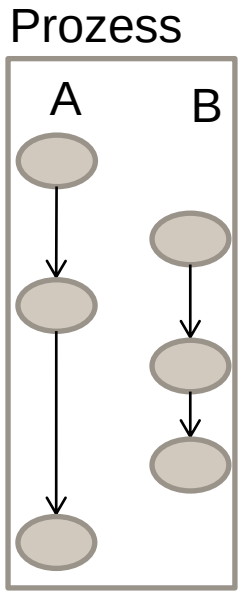
Definition: Parallelität

Zwei Aktivitäten A und B heißen parallel, wenn sie nebenläufig sind und zeitgleich ausgeführt werden können.

Beispiel*: Sensor 1 misst Druck, Sensor 2 misst gleichzeitig Helligkeit*



Begriffe lokaler Parallelität



Prozess

- Ist ein in Ausführung befindliches Programm
  (die *Dynamik* des Programms)
- Verwaltet vom Betriebssystem
- Besitzt eigenen Speicherbereich und Betriebsmittel

Thread *(engl. Faden, Strang)*

- Folge von Verarbeitungsschritten innerhalb eines Prozesses
- Jeder Prozess besitzt mindestens einen Thread
- Innerhalb eines Prozesses teilen sich Threads alle Ressourcen
  (d.h. Speicher und Betriebsmittel)
- „Leichtgewichtiger Prozess“

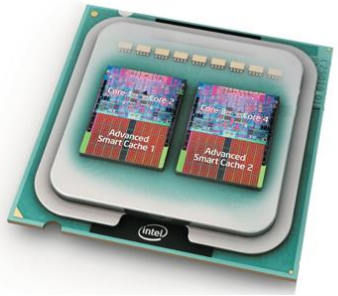
→ *Threads ermöglichen Nebenläufigkeit innerhalb eines Programms*

Umschaltung zwischen Threads oder Prozessen erfolgt automatisch durch eine **Scheduler** und ist **nicht-deterministisch**

Realisierung von Parallelität

- **Echte Parallelität auf Systemebene**: Zeitgleiche Ausführung von Nebenläufigkeiten durch getrennte Recheneinheiten, aber gemeinsamer Speicher (shared memory)
  - **Multiprozessorsystem:**
    Mehrere Prozessoren im System
    (z.B. Serversysteme)

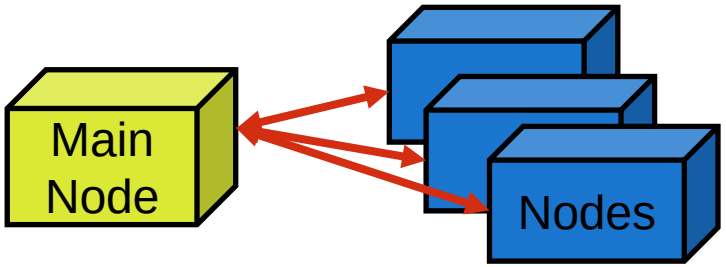
  - **Mehrkernprozessorsystem:**
    Dualcode, Quadcore, Hexacore-Systeme,…
    (z.B. Intel i7: 2-6 Kerne pro CPU)



- **Quasi-Parallelität:** Ausführung mehrerer Threads oder Prozesse durch:
  - Umschalten zwischen Threads und Prozessen (Multithreading, Multitasking)
  - Parallele Ausführung von Befehlen auf verschiedenen CPU-Kernen (Multicore-CPU) oder in verschiedenen Teilbereichen der CPU (Hyperthreading-Technologie)

Realisierung von Parallelität (2)

- **Verteilte Parallelität (Multi-Node):**
  Verbindung mehrerer Computer über Netzwerk, getrennter Speicher
  Beispiel: Google Webdienste: verteilt über mindestens 16 Zentren (wahrscheinlich mehr), pro Zentrum 45 Container mit jeweils 1.160 Rechnern



- **Distributed Computing:**
  Verbindung vieler Computer durch lose Kopplung, heterogene Computerarchitekturen (OS, Prozessoren, Leistung, etc.) und eventuell räumliche Verteilung → ***Grid Computing***

Threads in Python

- Modul threading
- Objektorientierter Aufbau: Klasse Thread
- Erzeugung:
  `t1 = Thread(target=<Funktionsname>, args=(<Parameter>))`
- `Starten: t1.start()`

Beispiel → threads.py

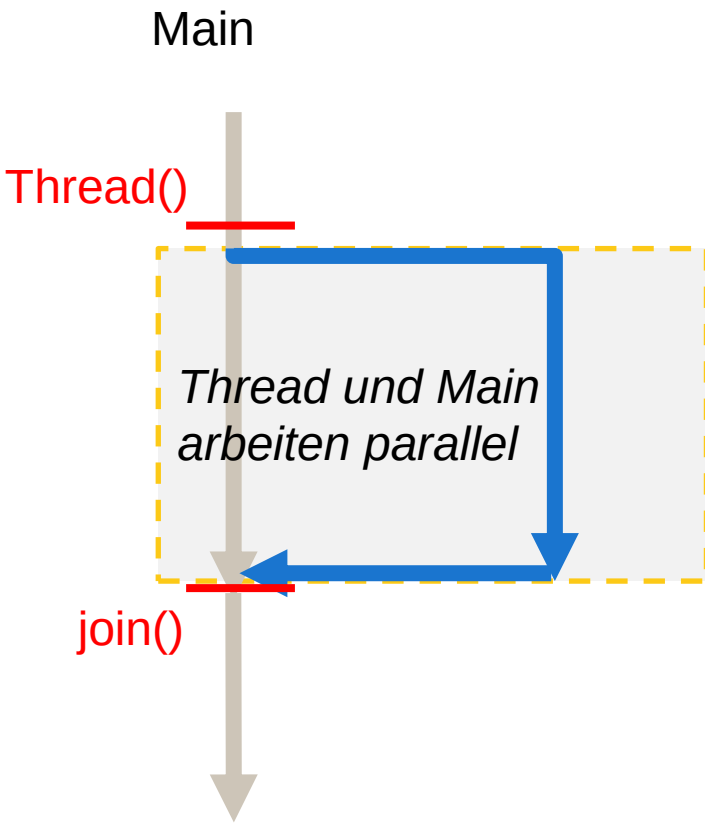
Beachte:

- Ein Thread wird beendet, wenn die Target-Methode verlassen wird
- Wenn alle Threads abgearbeitet sind, beendet sich das Hauptprogramm
- Die Reihenfolge der Abarbeitung der Threads ist zufällig

Warten auf Threads

- Anwendungsfall: Hauptprogramm soll blockieren, bis ein Thread (oder mehrere) beendet sind
- Realisierung: Methode join() von Thread
- Aufruf:

```
t1 = Thread(targetMethod)
t1.start()
…
t1.join()
```



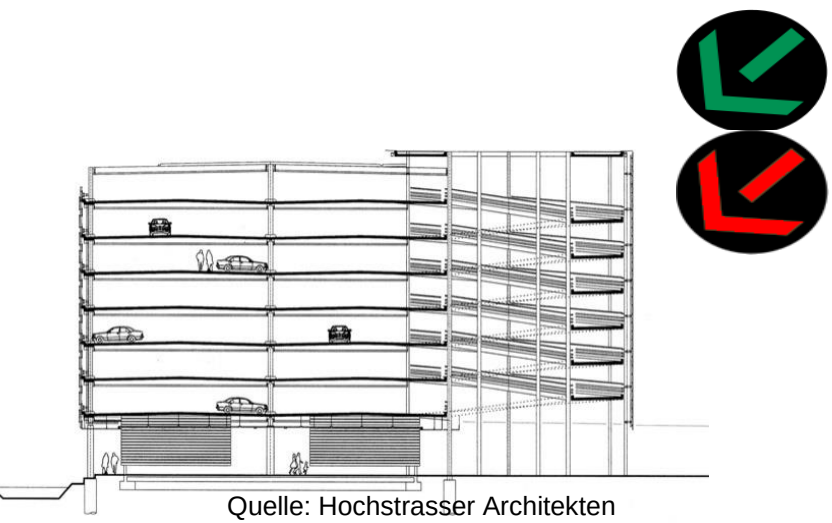
Beispiel → threads_join.py

Beispiel: Stockwerkbelegung im Parkhaus

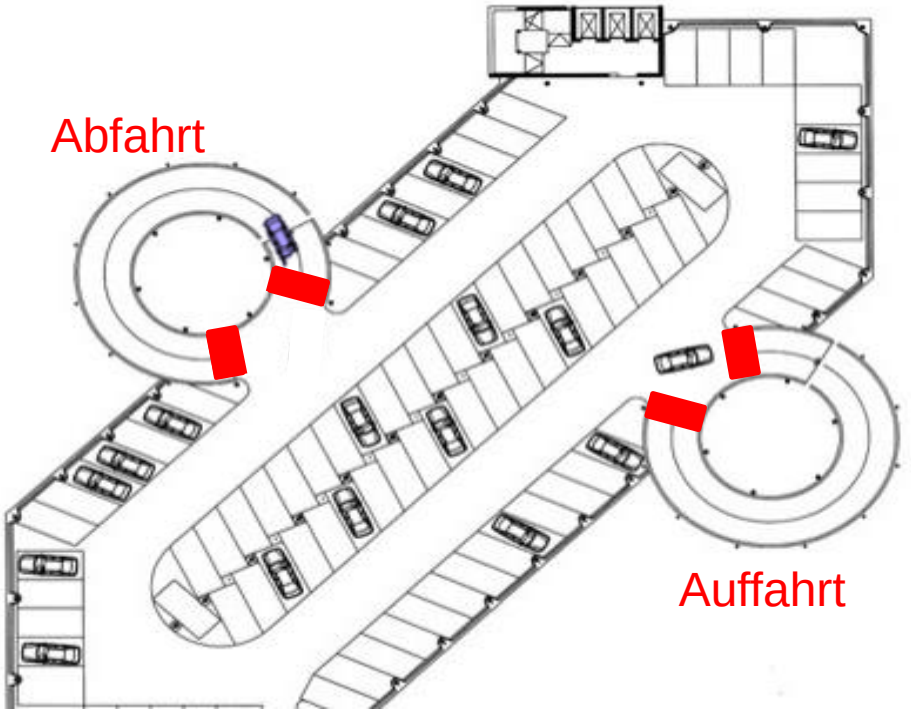
System zur Bestimmung der Fahrzeuganzahl auf einem Stockwerk

- Auswertesystem: Zählen der ein- und ausfahrenden Fahrzeuge
- Sensorik: je eine Lichtschranke in jeder Ein- und Ausfahrt des Stockwerks

→ „Gleichzeitiges“ Warten auf Signale der vier Lichtschranken



Quelle: Hochstrasser Architekten

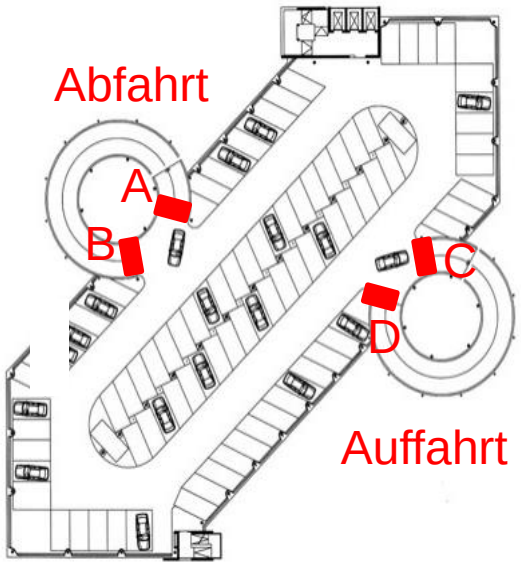


Beispiel: Stockwerkbelegung im Parkhaus

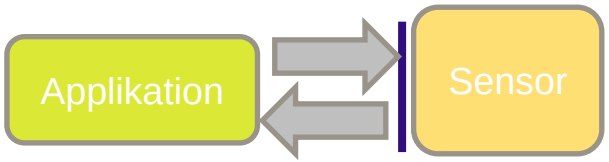
Erzeugen und Starten von 4 Threads (2 dargestellt):

```
def enter_from_above():
    while (True):
        # Warte auf Lichtschrankensignal
        wait_for_light_barrier("enter_from_above")
        # Erhöhe den Zähler
        update_car_counter(car_counter + 1)

def exit_above():
    while (True):
        # Warte auf Lichtschrankensignal
        wait_for_light_barrier("exit_above")
        # Verringere den Zähler
        update_car_counter(car_counter - 1)
[…]
# Globale Variablen
car_counter = 300
car_counter_correct = car_counter
# Threads starten
t_enter_from_above = Thread(target=enter_from_above)
t_enter_from_below = Thread(target=enter_from_below)
t_enter_from_above.start()
t_enter_from_below.start()
[…]
```



Quelle: Hochstrasser Architekten



Beispiel → thread_parkhaus.py

Beispiel: Stockwerkbelegung im Parkhaus (2)

Nebenläufige Abarbeitung der Aktivitäten:

```
def enter_from_above():
    while (True):
        # Warte auf Lichtschranke
        wait_for_light_barrier(
                  "enter_from_above")
        # Erhöhe den Zähler
        update_car_counter(car_counter + 1)
```

```
def exit_above():
    while (True):
        # Warte auf Lichtschrankensignal
        wait_for_light_barrier(
                              "exit_above")
        # Verringere den Zähler
        update_car_counter(car_counter - 1)
```

→ Funktioniert das Programm wie erwartet?

Beispielausgabe (initial 300 Fahrzeuge):

```
Aktueller Zählerstand: 297 (richtig: 300, Differenz: -3)
Aktueller Zählerstand: 295 (richtig: 292, Differenz: 3)
Aktueller Zählerstand: 293 (richtig: 290, Differenz: 3)
Aktueller Zählerstand: 285 (richtig: 285, Differenz: 0)
Aktueller Zählerstand: 273 (richtig: 291, Differenz: -18)
```

Problemursache: nebenläufige Speicherzugriffe

- Threads teilen sich Ressourcen (Speicher und Betriebsmittel) und daher auch **Variablen** und **Datenstrukturen**
- Die Umschaltung zwischen Threads erfolgt **nicht-deterministisch**

  → Wann und wo die Threads unterbrochen werden, ist unklar

  → Unterbrechung auch zwischen Lesen und Schreiben möglich!

Unser Beispiel:

```
def enter_from_above():
    while (True):
        wait_for_light_barrier(
                    "enter_from_above")
        update_car_counter(car_counter + 1)
```

A2 A1

```
def exit_above():
    while (True):
        wait_for_light_barrier(
                                "exit_above")
        update_car_counter(car_counter - 1)
```

B2 B1

```
def update_car_counter(new_value):
    sleep(0.00001)
    global car_counter
         car_counter = new_value
```

→ *Race Condition*

Nebenläufige Speicherzugriffe

Merke: Gleichzeitige Schreib-/ Lesezugriffe auf die gleiche Ressource aus unterschiedlichen Threads müssen **koordiniert** werden (**Synchronisation**).

→ Ansonsten kann es zum ***Lost-Update-Problem*** kommen

→ Auch bei Anweisungen in einer Zeile, z.B.:

```
car_counter = car_counter + 1
```

Lösung: Erzwingen des ***wechselseitigen Ausschlusses*** der Threads innerhalb eines bestimmten Bereiches

Definition: Kritischer Bereich

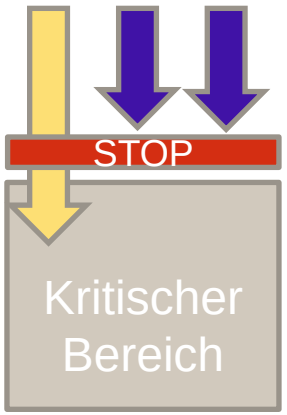
Bereich, in dem mindestens zwei nebenläufige Aktivitäten auf die gleiche Ressource zugreifen und mindestens eine davon schreibt.

Wechselseitiger Ausschluss (Mutex)

Mutex (***mut****ual* ***ex****clusion*): Oberbegriff für Verfahren zur Realisierung von wechselseitigem Ausschluss

Eigenschaften:

- Der von einer Mutex geschützte, kritische Bereich ist für andere Threads nicht zugänglich
- Wenn der kritische Bereich bereits von einem Thread betreten wurde, warten andere Threads
- Verhindert
  - Änderungen der Daten während des Lesens
  - Unterbrechung während der Bearbeitung von Datensätzen, deren Konsistenz garantiert sein muss
  - Verwendung inkonsistenter Daten
- Realisierung mit Hilfe einer **atomaren** Systemfunktion



Realisierung kritischer Bereiche: Monitor

- Die einfachste Variante einer Mutex-Technik
- **Prüft** und **sperrt** (atomar!) den kritischen Bereich
- Immer im Zusammenhang mit einem Lock-Objekt:

```
// Lock-Objekt anlegen
lock_obj = Lock()
[…]
lock_obj.acquire()
kritischer Bereich
lock_obj.release()
unkritischer Bereich
```

Realisierung kritischer Bereiche: Monitor (2)

Definiere gemeinsames Sperrobjekt (global):

```
[…]
counter_lock_obj = Lock()
[…]
```

Prüfung und Sperren der kritischen Abschnitte mit Hilfe des Lock-Objektes:

```
def enter_from_above():
    while (True):
        wait_for_light_barrier(
                    "enter_from_above")
        counter_lock_obj.acquire()
        update_car_counter(car_counter + 1)
        counter_lock_obj.release()
```

```
def exit_above():
    while (True):
        wait_for_light_barrier(
                                "exit_above")
        counter_lock_obj.acquire()
        update_car_counter(car_counter - 1)
        counter_lock_obj.release()
```

Nachteile von Locks

Locks besitzen auch Nachteile:

Performance-Verlust: Locks führen durch die atomare Ausführung der Prüf-Funktionen zu einem Performance-Verlust

→ Lösung: den kritischen Bereich so klein wie möglich wählen

Verklemmungs-Problematik (*Deadlock*)

- Wird ein Release vergessen, so wird dieser kritische Bereich nicht mehr freigegeben
  → das Programm blockiert (für immer…)

Lösung in Python: with-Statement

→ acquire() und release() wird beim Betreten und Verlassen des with-Abschnitts automatisch ausgeführt

```
with counter_lock_obj:
    update_car_counter(car_counter + 1)
```

Deadlocks bei mehreren Ressourcen

- Deadlocks können auch auftreten, wenn in einem Programm mehr als ein Lock verwendet wird
  - mehr als ein kritischer Bereich
  - mehr als eine exklusiv genutzte Ressource
  - …

Beispiel → threads_monitor.py

- Daher möglichst die Ressourcen immer in der gleichen Reihenfolge allokieren!

Weiter Einsatzgebiete für nebenläufige Programmierung

Auswahl:

Blockierende Systemaufrufe: Behandlung mehrerer blockierender Schnittstellen

Beispiel: *Warten auf die Signalisierung mehrerer Lichtschranken*

Ereignisbehandlung: Programm muss jederzeit auf Ereignisse reagieren können

Beispiel: *Abschaltung bei Temperatur-Überschreitung, Mouse-Klicks auf Benutzeroberflächen*

Wartezeitoptimierung: Parallelisierung unterschiedlich schneller Operationen

Beispiel: *EKG-Ausdruck und gleichzeitige Signalaufnahme*

Algorithmenbeschleunigung: Parallelisierung von Algorithmen

Beispiel: *Berechnung von Matrixoperationen in der Bildverarbeitung*

…