

Nebenläufigkeit in Python

Concurrency in Python

Dozent: Prof. Dr. Michael Eichberg
Kontakt: michael.eichberg@dhbw.de
Version: 1.0

Folien: <https://delors.github.io/ds-nebenlaeufigkeit-in-python/folien.de.rst.html>
<https://delors.github.io/ds-nebenlaeufigkeit-in-python/folien.de.rst.html.pdf>
Fehler melden: <https://github.com/Delors/delors.github.io/issues>

Hintergrund

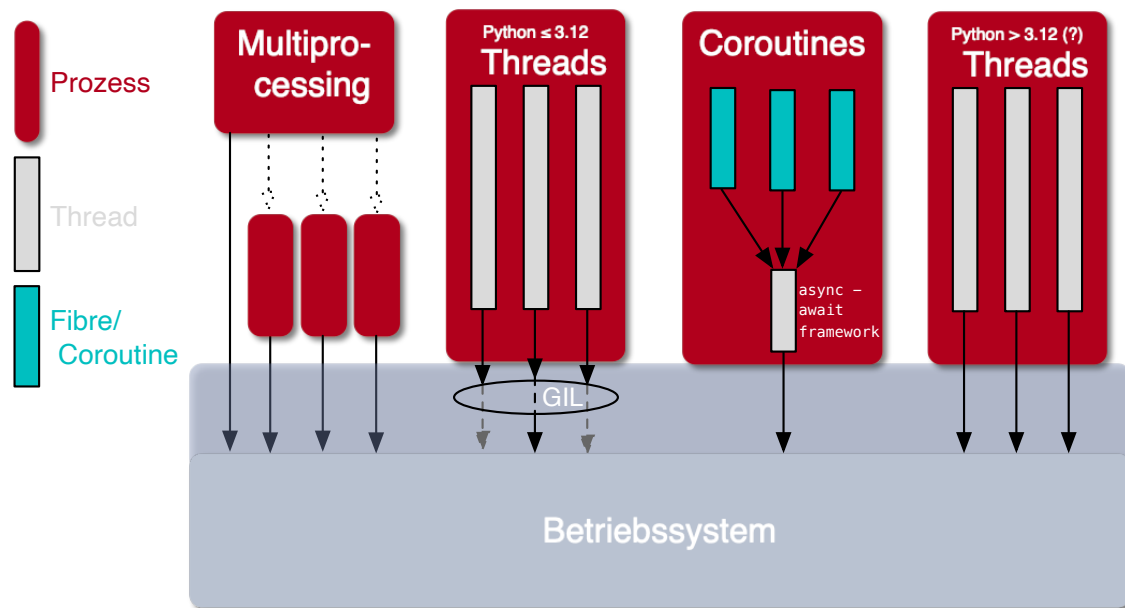
Nebenläufigkeit ist notwendig, um:

- die Reaktionsfähigkeit von Anwendungen zu verbessern
- eine bessere Auslastung von (Mehrkern-)prozessoren zu erreichen

Ein gutes Verständnis von nebenläufiger Programmierung ist für die Entwicklung von verteilten Anwendungen unerlässlich, da Server immer mehrere Anfragen gleichzeitig bearbeiten.

1. Nebenläufigkeitsmodelle in Python

Prozesse vs. Threads vs. Coroutines in Python



- Prozesse sind voneinander isoliert und können nur über explizite Mechanismen miteinander kommunizieren (z. B. Pipes und Queues); Prozesse teilen sich *nicht* denselben Adressraum.
- Alle Threads eines Prozesses teilen sich denselben Adressraum. Python Threads sind vom Betriebssystem unterstützte Threads, die direkt vom Betriebssystem verwaltet werden. Python (d. h. der Standardinterpreter CPython bis (mind.) einschließlich Version 3.12) führt aber immer nur einen Thread aus aufgrund des *Global Interpreter Locks* (GIL).

Der GIL existiert(e) insbesondere, da dadurch die Implementierung von Python einfacher wurde (z. B. kann problemlos *Reference Counting* verwendet werden und Probleme mit externen Bibliotheken sind auch minimiert.)

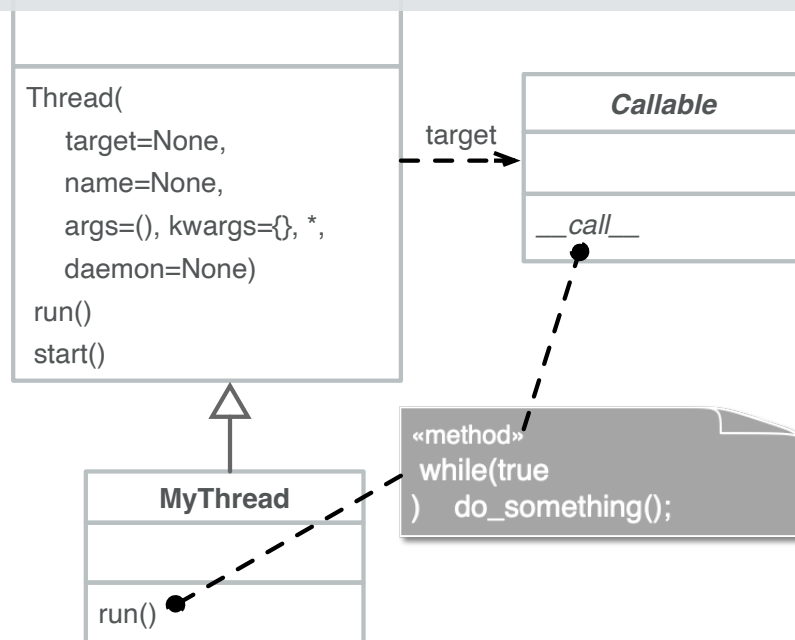
Andere Python-Implementierungen (wie Jython und IronPython) haben keinen GIL und können daher mehrere Threads (echt) parallel ausführen.

- *Coroutines* (auch *Fibres*) nutzen immer kooperatives Multitasking. D. h. ein Fibre gibt die Kontrolle an eine andere Fibre explizit ab. (Früher wurden *Fibres* auch als *Green Threads* bezeichnet.) Diese sind für das Betriebssystem unsichtbar.

Coroutines erfordern explizite Unterstützung in den Bibliotheken. Alle auf Koroutinen basierenden Tasks werden in von der Event-Loop verwaltet und von einem einzigen Thread ausgeführt.

Nebenläufigkeit in Python

Die `Process` API bietet eine vergleichbare Schnittstelle.



- Threads werden in Python über die vordefinierte Klasse `threading.Thread` bereitgestellt.
- Alternativ kann ein `Callable` an ein Thread-Objekt übergeben werden.
- Threads beginnen ihre Ausführung erst, wenn die `start`-Methode in der Thread-Klasse aufgerufen wird. Die `Thread.start`-Methode ruft die `run`-Methode auf. Ein direkter Aufruf der `run`-Methode führt nicht zu einer nebenläufigen Ausführung.
- Der aktuelle Thread kann mittels der statischen Methode `Thread.currentThread()` ermittelt werden.
- Ein Thread wird beendet, wenn die Ausführung seiner `run`-Methode entweder normal oder als Ergebnis einer unbehandelten Ausnahme endet.
- Python unterscheidet *User-Threads* und *Daemon-Threads*.

Daemon-Threads sind Threads, die allgemeine Dienste bereitstellen und normalerweise nie beendet werden. Jeder Thread, der eine Endlosschleife ausführt sollte als Daemon-Thread gekennzeichnet werden bei Erzeugung.

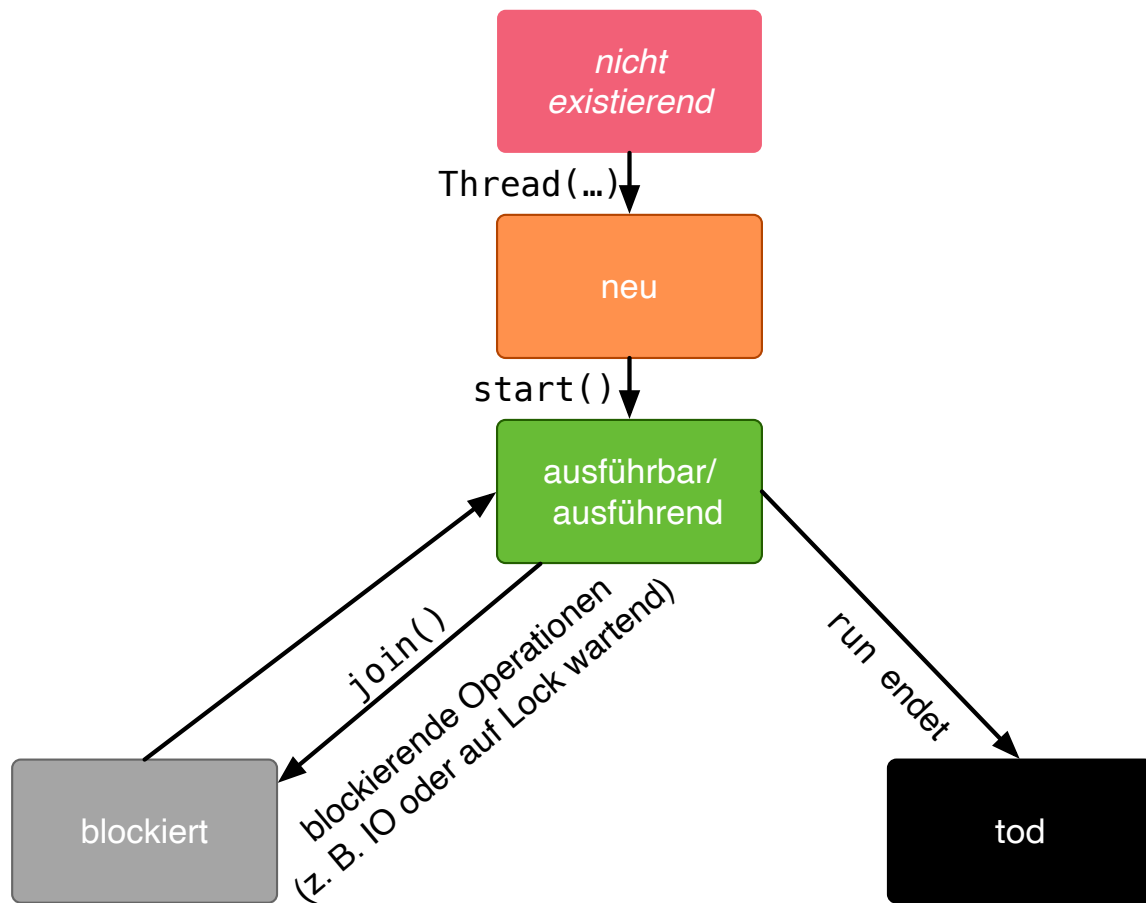
Wenn alle Benutzer-Threads beendet sind, werden die Daemon-Threads automatisch beendet, und das Hauptprogramm endet.

Der Thread kann beim Erzeugen als Daemon-Thread gekennzeichnet werden, indem der Parameter `daemon` auf `True` gesetzt wird.

Inter-Thread/Process-Koordination

- Ein `Thread/Process` kann (mit oder ohne Zeitüberschreitung) auf die Beendigung eines anderen `Threads/Processes` (des Ziels) warten, indem er die `join`-Methode für das `Thread/Process`-Objekt des Ziels aufruft.
- Mit der Methode `is_alive` kann ein Thread feststellen, ob der Ziel-Thread beendet wurde.

Python Thread States



Beispiel: Multiprocessing - „IO-Bound“

```
1 import time
2 from multiprocessing \
3     import Process, current_process
4
5 def busy_sleep():
6     time.sleep(10)
7
8 print(current_process().name)
9
10 if __name__ == '__main__':
11     p1 = Process(target=busy_sleep)
12     p2 = Process(target=busy_sleep)
13     p1.start()
14     p2.start()
15     p1.join()
16     p2.join()
```

\$ `time ./processes_sleep.py`

MainProcess

Process-2

Process-1

`./processes_sleep.py`

0.07s user

0.02s system

0% cpu

10.070 total

Beispiel: Multiprocessing - CPU-Bound

```
import time
from multiprocessing \
    import Process, current_process

def computation():
    j = 1
    for i in range(100*1000*1000):
        j += (i/j)
    print("Done:"+str(j))

print(current_process().name)

if __name__ == '__main__':
    p1 = Process(target=computation)
    p2 = Process(target=computation)
    p1.start()
    p2.start()
    p1.join()
    p2.join()
```

```
$ time ./processes_computation.py
MainProcess
Process-1
Process-2
Done:100000000.0
Done:100000000.0
./processes_computation.py
 5.60s user
 0.02s system
194% cpu
 2.899 total
```

Hinweise

Je nach Betriebssystem werden die Kindprozesse ggf. anders ausgeführt (`fork` oder `spawn`). Linux/Posix bietet die beste Unterstützung gefolgt von MacOS und Windows.

Beispiel: Threading - „IO-Bound“

```
#!/usr/bin/env python3
import time
from threading import Thread, current_thread
```

```
def busy_sleep():
    # ts_print(current_thread().name)
    time.sleep(10)
```

```
if __name__ == '__main__':
    t1 = Thread(target=busy_sleep)
    t2 = Thread(target=busy_sleep)
    t1.start()
    t2.start()
    t1.join()
    t2.join()
```

```
$ time ./threads_sleep.py
0.02s user
0.01s system
0% cpu
10.188 total
```

Beispiel: Threading - CPU-Bound

```
#!/usr/bin/env python3
import time
from threading \
    import Thread, current_thread

def computation():
    ts_print(current_thread().name)
    j = 1
    for i in range(100*1000*1000):
        j += (i/j)
    ts_print("Done:"+str(j))

if __name__ == '__main__':
    t1 = Thread(target=computation)
    t2 = Thread(target=computation)
    t1.start()
    t2.start()
    t1.join()
    t2.join()

$ time ./threads_computation.py 16:10:15
Thread-1 (computation)
Thread-2 (computation)
Done:100000000.0
Done:100000000.0
Done.
./threads_computation.py
5.27s user
0.02s system
96% cpu
5.450 total
```

Beispiel: Coroutines

```
#!/usr/bin/env python3
import asyncio

async def busy_sleep(id):
    print(f"Task {id} started")
    await asyncio.sleep(10)
    print(f"Task {id} completed")

async def main():
    t1 = asyncio.create_task(busy_sleep(1))
    t2 = asyncio.create_task(busy_sleep(2))

    print("Both initialized.")
    await t1
    await t2
    print("Done.")



if __name__ == '__main__':
    asyncio.run(main())

$ time ./async.py
Both initialized.
Task 1 started
Task 2 started
Task 1 completed
Task 2 completed
Done.
./async.py
 0.05s user
 0.01s system
 0% cpu
10.063 total
```

-
- Beide Tasks werden von dem gleichen Thread ausgeführt. Der Thread gibt „die Kontrolle an die Event-Loop ab“, wenn er auf eine entsprechende blockierende Methode trifft. Die Event-Loop kann dann die Kontrolle an einen anderen Task übergeben.
 - Warten (`await`) ist nur möglich in asynchronen Methoden (`async def`).
 - `asyncio.run(<fn>)` startet die Event-Loop und führt die übergebene asynchrone Methode aus.
 - Die Verwendung von Koroutinen erfordert explizite Unterstützung in den Bibliotheken.

2. Sperren und Bedingungsvariablen

Synchronisation mit Hilfe von *Sperren*

- Zugriff auf gemeinsam genutzte Ressourcen muss synchronisiert werden, um  *Race Conditions* ( *Wettlaufsituationen*) zu vermeiden.
(Unabhängig davon ob Threads echt parallel oder nur scheinbar parallel ausgeführt werden.)
- Eine *Sperre* (`Lock`) ist ein Objekt, das es erlaubt Code im wechselseitigen Ausschluss (engl. *mutual exclusion*) auszuführen.
D. h. ein Thread blockiert, wenn er versucht eine Sperre zu erwerben, die bereits von einem anderen Thread gehalten wird.
- Der Code, der von einer Sperre geschützt wird, wird als kritischer Abschnitt bezeichnet.

Eine *Race Condition* liegt vor, wenn der Zustand eines (Software-)Systems von der Abfolge oder dem Zeitpunkt anderer unkontrollierbarer Ereignisse abhängt. Eine Race Condition führt ggf. zu unerwarteten oder inkonsistenten Ergebnissen.

Verwendung von *Sperren*^[1]

- Am Anfang des kritischen Abschnitts wird die Sperre angefordert mit `<Lock>.acquire()`.
- Am Ende des kritischen Abschnitts wird die Sperre freigegeben mit `<Lock>.release()`.
- Um sicherzustellen, dass eine gehaltene Sperre immer aufgehoben wird, sollte `try-finally` oder ein passendes `with`-Statement verwendet werden. (Lock implementiert z. B. das Protokoll von *Context-Managern*)

```
lock = Lock()
lock.acquire()
try:
    # critical section
finally:
    lock.release()
lock = Lock()

with lock:
    # critical section
```

^[1] Die APIs von `threading` und `multiprocessing` sind in weiten Teilen vergleichbar.

Beispiel: Thread-safe Shared Counter

```
from threading import Thread, Lock
```

```
class SharedCounter:
```

```
    def __init__(self):  
        self._value = 0  
        self.lock = Lock()
```

```
    def value(self):  
        return self._value
```

```
# Thread-sichere Implementierungen  
# von increment und decrement
```

```
def increment(self):  
    self.lock.acquire()  
    try:  
        self._value += 1  
    finally:  
        self.lock.release()
```

```
def decrement(self):  
    with self.lock:  
        self._value -= 1
```

Warnung

Code, der eine konkrete Sperre erzeugt, anfordert und freigibt, sollte immer lokal sein; d.h. nicht über die Code-basis verteilt sein. Auch wenn es möglich ist eine Instanz eines Locks weiterzureichen und sperren in einer Methode anzufordern und in einer anderen Methode freizugeben, so ist dies eine schlechte Praxis, da es zu ((sehr,) sehr) schwer zu findenden Fehlern führen kann.

Sperren und komplexe Rückgabewerte

```
from threading import Thread, Lock
```

```
class SharedCoordinate:
```

```
    def __init__(self, x, y):  
        self.x = x  
        self.y = y  
        self.lock = Lock()
```

```
    def update(self, x, y):  
        self.lock.acquire()  
        try:  
            self.x = x  
            self.y = y  
        finally:  
            self.lock.release()
```

```
    def value(self):  
        with self.lock:  
            return (self.x, self.y)
```

Beide Methoden müssen synchronisiert werden, damit es nicht dazu kommen kann, dass man einen ungültigen Zustand beobachten kann. Ein ungültiger Zustand wäre ein paar Koordinaten, die nicht zusammengehören. Z. B. wenn der Wert x von einem Aufruf kommt (update(100,100)) und der Wert y von einem anderen (update(200,200)); d.h. der Wert, den `value` zurückliefert: 100, 200 wäre.

Bedingte Synchronisation

- drückt eine Bedingung für die Reihenfolge der Ausführung von Operationen aus.
- z. B. können Daten erst dann aus einem Puffer entfernt werden, wenn Daten in den Puffer eingegeben wurden.
- Python unterstützt optionale Bedingungs-Variablen (Instanzen von `Condition`), mit den klassischen Methoden `wait` und `notify` bzw. `notify_all`.

Diese Methoden erlauben es auf bestimmte Bedingungen zu warten und andere Threads zu benachrichtigen, wenn sich die Bedingung geändert hat.

Programmierung mit `Conditions`

- Die Methoden `wait` und `notify(_all)` können nur verwendet werden, wenn die Sperre gehalten wird; andernfalls wird eine `RuntimeError` ausgelöst.
- Die `wait`-Methode blockiert immer den aufrufenden Thread und gibt die mit dem Objekt verbundene Sperre frei.
- Die `notify(n=1)`-Methode weckt (mind.) n wartende Threads auf. Welcher Thread aufgeweckt wird, ist nicht spezifiziert.
`notify` gibt die Sperre nicht frei; daher muss der aufgeweckte Thread warten, bis er die Sperre erhalten kann, bevor er fortfahren kann.
- Um alle wartenden Threads aufzuwecken, muss die Methode `notify_all` verwendet werden.
Warten die Threads aufgrund unterschiedlicher Bedingungen, so ist immer `notify_all` zu verwenden.
- Wenn kein Thread wartet, dann haben `notify` und `notify_all` keine Wirkung.

Wichtig

Wenn ein Thread aufgeweckt wird, kann er nicht davon ausgehen, dass seine Bedingung erfüllt ist! Die Bedingung ist immer in einer Schleife zu prüfen und der Thread muss ggf. wieder in den Wartezustand versetzen.

Beispiel: Implementation eines *BoundedBuffer*

■ Ein *BoundedBuffer* hat (z. B.) traditionell zwei Bedingungsvariablen:

- *not_full* und
- *not_empty*.

In diesem Fall würde gelten, dass, wenn ein Thread auf eine Bedingung wartet, kein anderer Thread auf die andere Bedingung warten kann, da sich die Bedingungen gegenseitig ausschließen.

Beispiel: Synchronisation mit Bedingungsvariablen

```
from threading \
    import Condition, Lock

class BoundedBuffer:

    def __init__(self, capacity):
        self.capacity = capacity
        self.buffer = []
        self.lock = Lock()
        self.not_empty = Condition(self.lock)
        self.not_full = Condition(self.lock)

    def put(self, item):
        with self.not_full:
            while len(self.buffer) == \
                self.capacity:
                self.not_full.wait()
            self.buffer.append(item)
            self.not_empty.notify()

    def get(self):
        with self.not_empty:
            while len(self.buffer) == 0:
                self.not_empty.wait()
            item = self.buffer.pop(0)
            self.not_full.notify()
            return item
```

Beispiel: Synchronisation mit nur einer Bedingung

Im Folgenden sehen wir eine Implementierung mit nur einer Bedingungsvariablen, um bestimmte Synchronisationsfehler demonstrieren zu können.

```
1 from threading import Thread, Lock, Condition
2
3 class BoundedBuffer:
4
5     def __init__(self, capacity):
6         self.capacity = capacity
7         self.buffer = []
8         self.lock = Lock()
9         self.not_used = Condition(self.lock)
10
11     ...
12
13     def put(self, item):
14         with self.not_used:
15             while len(self.buffer) == self.capacity:
16                 self.not_used.wait()
17             self.buffer.append(item)
18             self.not_used.notify_all() # notify_all() !
19
20     def get(self):
21         with self.not_used:
22             while len(self.buffer) == 0:
23                 self.not_used.wait()
24             item = self.buffer.pop(0)
25             self.not_used.notify_all() # notify_all() !
26             return item
```

Fehlersituation, die bei der Verwendung von `notify` (statt `notify_all`) auftreten könnte.

```
bb = BoundedBuffer(1);
p1 = Thread(target=lambda: bb.put(1)); p2 = Thread(target=lambda: bb.put(2))
c1 = Thread(target=lambda: bb.get()); c2 = Thread(target=lambda: bb.get())
c1.start(); c2.start(); p1.start(); p2.start();
```

	Aktionen	(Änderung des) Zustand(s) des Buffers	Auf die Sperre (Lock) wartend	An der Bedingung wartend
1	c1:bb.get() , c2:bb.get(), p1:bb.put(), p2:bb.put()	empty	{c2,p1,p2}	{c1}
2	c2:bb.get()	empty	{p1,p2}	{c1,c2}
3	p1:bb.put(1)	empty → not empty	{p2,c1}	{c2}
4	p2:bb.put(2)	not empty	{c1}	{c2,p2}
5	c1:bb.get()	not empty → empty	{c2}	{p2}
6	c2:bb.get()	empty	∅	{c2,p2}

In Schritt 5 wurde (z. B.)- aufgrund des Aufrufs von `notify` durch `c1` - der Thread `c2` aufgeweckt - anstatt des Threads `p2`. Der aufgeweckte Thread `c2` prüft die Bedingung (Schritt 6) und stellt fest, dass der Puffer leer ist. Er geht wieder in den Wartezustand. Jetzt warten sowohl ein Thread, der ein Wert schreiben möchte, als auch ein Thread, der einen Wert lesen möchte.

Best Practices in Hinblick auf Synchronisation

- Code, der eine Sperre hält (🔒 *Lock*) sollte so kurz (zeitlich) wie möglich gehalten werden.
(D. h. der Code zwischen `Lock.acquire()` und `Lock.release()`)
- Verschachtelte Anforderungen von Sperren sollten vermieden werden, da die äußere Sperre nicht freigegeben wird, wenn man an der Inneren wartet. Dies kann leicht zum Auftreten eines Deadlocks führen.

- Wenn zwei (oder mehr) Threads bzw. Prozesse auf die gleichen Ressourcen in unterschiedlicher Reihenfolge zugreifen und entsprechende Sperren halten bzw. anfordern, kann es zu einem Deadlock kommen.

Zu Beachten

Ressourcen immer in der gleichen Reihenfolge sperren, um Deadlocks zu vermeiden.

Sperren (d. h. `Locks`) in Verbindung mit Bedingungsvariablen sind nur eine Möglichkeit, um die Synchronisation von Threads zu ermöglichen. Es ist jedoch ein sehr häufiges Modell. (Alternativen sind zum Beispiel: *Semaphoren*, *Nachrichtenübermittlung*)

3. Ausgewählte Aspekte der Nebenläufigkeit

Thread-lokaler Speicher

Thread-lokaler Speicher (`threading.local()`) ermöglicht es, dass jeder Thread eine lokale Kopie einer bestimmten Variable hat

```
import time
import threading

stop = False # shared global variable
local_data = threading.local()

def f(v):
    setattr(local_data, "value", 0)
    while(not stop):
        print(local_data.value)
        local_data.value += v
        time.sleep(1)

# "main" thread
t1 = threading.Thread(target=f, args=(1,))
t2 = threading.Thread(target=f, args=(-1,))
t1.start()
t2.start()
time.sleep(3);
print("Attributes of local_data: " + \
      str(local_data.__dict__.keys()))
stop = True
print("Stop set to True.")
t1.join()
t2.join()
```

```
$ ./ThreadLocal.py
0
0
-1
1
-2
2
Attributes of local_data: []
Stop set to True. Waiting for threads to finish.
```

Reentrant Locks

- *Reentrant Locks* (`RLock`) sind Sperren, die von demselben Thread mehrmals erworben werden können.
- Implementierungen: `threading.RLock` oder `multiprocessing.RLock`.

Thread-/ProcessPools

- *ThreadPool* und *ProcessPool* bieten eine höherwertige Abstraktion, um eine große Anzahl von Aufgaben nebenläufig zu verarbeiten.
 - Beide erben von `concurrent.futures.Executor`; zentrale Methoden:
 - `submit(fn, *args, **kwargs)`: Fügt eine Aufgabe hinzu und gibt ein `Future`-Objekt zurück.
- Auf Futures sind die Hauptfunktionen:
- `done()`: Gibt zurück, ob die Aufgabe abgeschlossen ist.
 - `result(timeout=None)`: Gibt das Ergebnis zurück, wenn die Aufgabe abgeschlossen ist; blockiert ggf..
 - `map(func, *iterables, timeout=None, chunksize=1)`: Führt die Funktion für jedes Element in `iterables` aus und gibt die Ergebnisse in der Reihenfolge zurück, in der sie abgeschlossen wurden.

Motivation: Nachrichtenaustausch

- Locks haben das große Potential eigentlich nebenläufige Programme effektiv zu serialisieren (und zu verlangsamen).
- Prozesse nutzen keinen gemeinsamen Adressraum.
- Eine Möglichkeit auf Locks weitgehend zu verzichten ist der Nachrichtenaustausch.

Generell ist der Austausch zwischen Prozessen über `Queues`, `Pipes` und (explizitem) `SharedMemory` möglich; d. h. in diesen Fällen ist Inter-Prozess-Kommunikation (*Interprocess Communication* (*IPC*)) notwendig.

Queues

`queue.Queue` oder `multiprocessing.JoinableQueue`

Die grundlegenden Methoden von `Queues` sind:

- `Queue(maxsize=0)`
Erzeugt eine neue Queue-Instanz welche `maxsize` Elemente speichern kann. 0 bedeutet, dass die Queue unendlich groß ist.
(Pythons `Queue` realisiert einen *Bounded Buffer*.)
- `put(item)` : Fügt ein Element in die Queue ein.
- `get()` : Entfernt und gibt das erste Element aus der Queue zurück.
- `task_done()` : Signalisiert, dass ein Element aus der Queue *abgearbeitet* wurde.
- `join()` : Blockiert bis alle Elemente aus der Queue *abgearbeitet* wurde.

Beispiel - Verwendung von queues für Thread-Sichere Konsolenausgabe

Setup

```
import threading
from queue import Queue

print_queue = Queue()

def ts_print(msg):
    print_queue.put(msg)

def print_handler():
    while True:
        msg = print_queue.get()
        # there will ever be only one thread
        print(msg)
        print_queue.task_done()
```

Verwendung

```
Thread(target=print_handler, daemon=True).start()
:
# <thread 1:> ts_print("Hello")
:
# <thread 2:> ts_print("World")
:
print_queue.join()
```

Hinweise

- nur ein Thread darf die print_queue abarbeiten
- wir müssen überall ``ts_print`` verwenden

Verwendung von queues für die Kommunikation zwischen Prozessen

```
1 from random import randint
2 from multiprocessing import current_process, Process, JoinableQueue as MPQueue
3 from threading import Thread
4 from queue import Queue as TQueue
5 import time
6
7 def print_queue_handler(print_queue):
8     while True:
9         msg = print_queue.get()
10        print(msg)
11        print_queue.task_done()
12
13 def read_from_ip_queue(ip_queue, print_queue): # ip =(here) interprocess
14     while True:
15         msg = ip_queue.get()
16         print_queue.put(msg)
17         ip_queue.task_done()
18
19 def f(c_to_p_ip_queue):
20     time.sleep(randint(1, 3)) # just some fuzzing
21
22     c_to_p_ip_queue.put("I'm alive: " + current_process().name)
23
24     time.sleep(randint(1, 3)) # just some fuzzing
25
26     c_to_p_ip_queue.put("Hell World from " + current_process().name)
27
28 if __name__ == "__main__":
29     print_queue = TQueue()
30     c_to_p_ip_queue = MPQueue()
31     p1 = Process(target=f, args=(c_to_p_ip_queue,))
32     p1.start()
33     p2 = Process(target=f, args=(c_to_p_ip_queue,))
34     p2.start()
35     Thread(
36         target=read_from_ip_queue,
37         args=(c_to_p_ip_queue, print_queue, ),
38         daemon=True,
39     ).start()
40     Thread(target=print_queue_handler, args=(print_queue,), daemon=True).start()
41     c_to_p_ip_queue.join()
42     print_queue.join()
43     p2.join()
44     p1.join()
```


Thread Safety - Voraussetzung

Damit eine Klasse thread-sicher ist, muss sie sich in einer single-threaded Umgebung korrekt verhalten.

D. h. wenn eine Klasse korrekt implementiert ist, dann sollte keine Abfolge von Operationen (Lesen oder Schreiben von öffentlichen Feldern und Aufrufen von öffentlichen Methoden) auf Objekten dieser Klasse in der Lage sein:

- das Objekt in einen ungültigen Zustand versetzen,
- das Objekt in einem ungültigen Zustand zu beobachten oder
- eine der Invarianten, Vorbedingungen oder Nachbedingungen der Klasse verletzen.

Die Klasse muss das korrekte Verhalten auch dann aufweisen, wenn auf sie von mehreren Threads aus zugegriffen wird.

- Unabhängig vom *Scheduling* oder der Verschachtelung der Ausführung dieser Threads durch die Laufzeitumgebung,
- Ohne zusätzliche Synchronisierung auf Seiten des aufrufenden Codes.

Dies hat zur Folge, dass Operationen auf einem thread-sicheren Objekt für alle Threads so erscheinen als ob die Operationen in einer festen, global konsistenten Reihenfolge erfolgen würden.

Da sich Prozesse den Adressraum mit Threads nicht teilen, ist es nicht möglich, dass ein Prozess den Speicher eines anderen Prozesses direkt manipuliert. Dies bedeutet jedoch nicht, dass keine Inter-Prozess-Koordination notwendig ist. Insbesondere wenn auf gemeinsame Ressourcen - wie zum Beispiel die Konsole - zugegriffen wird, ist eine Koordination notwendig.

Thread Safety Level

Immutable  *Unveränderlich:*

Die Objekte sind konstant und können nicht geändert werden.

Thread-sicher: Die Objekte sind veränderbar, unterstützen aber nebenläufigen Zugriff, da die Methoden entsprechende Sperren und Bedingungen verwenden.

Bedingt Thread-sicher:

All solche Objekte bei denen jede einzelne Operation thread-sicher ist, aber bestimmte Sequenzen von Operationen eine externe Synchronisierung erfordern können.

Thread-kompatibel:

Alle Objekte die keinerlei Synchronisierung aufweisen. Der Aufrufer kann die Synchronisierung jedoch ggf. extern übernehmen.

Thread-hostile „Thread-schädlich“:

Objekte, die nicht thread-sicher sind und auch nicht thread-sicher gemacht werden können, da sie zum Beispiel globalen Zustand manipulieren.

Ein Beispiel bzgl. *bedingt Thread-sicher* wäre die Verwendung eines Iterators, bei dem die Methoden für sich genommen thread-sicher sind, aber die Iteration über die Elemente als ganzes zusätzliche Synchronisation erfordert, damit die Ergebnisse konsistent sind.

Ein Beispiel für eine *thread-schädliche* Klasse (Code) wäre eine Klasse, die auf eine globale Variable zugreift bzw. globalen Zustand ändert, der von mehreren Threads verwendet wird, ohne dass eine Synchronisierung stattfindet.

Warnung

Wenn Nebenläufigkeit nicht richtig umgesetzt wird, dann kann dies nicht nur zu schwer zu findenden Fehlern führen sondern auch **zu langsam(er)en Programmen**.

Im Allgemeinen sollte Parallelisierung auf *höchstmöglicher Ebene* erfolgen.

Threads und Prozesse nicht terminieren

Warnung

Auch wenn es technisch möglich ist Threads und Prozesse explizit zu terminieren (z. B. durch `Process.terminate()`) so sollte man darauf verzichten.

Das Hauptproblem sind nicht freigegebene Locks und Ressourcen, die sich in einem inkonsistenten Zustand befinden können.

Auch in anderen Programmiersprachen sollte man niemals Threads oder Prozesse explizit terminieren.

Warnung

Nebenläufigkeit macht *nichts* einfacher! Entwickle und teste immer erst eine single-threaded Version des Programms.

Übung

Implementieren Sie einen einfachen *DelayedBuffer*, der es ermöglicht Aufgaben (d. h. Objekte vom Typ `Callable`) erst nach einer bestimmten Zeit auszuführen. Die Klasse muss zwei Funktionen zur Verfügung stellen:

```
submit(self, delay, fn, *args, **kwargs):
```

Die Funktion `fn` wird nach `delay` Sekunden ausgeführt wobei `delay` vom Typ `Float` ist. `args` und `kwargs` sind die Argumente, die an `fn` übergeben werden.

```
join(self):
```

Wartet bis alle Aufgaben abgearbeitet wurden.

Im folgenden sehen Sie eine mögliche Verwendung des Puffers:

```
buffer = DelayedBuffer()
buffer.submit(100 / 1000, ts_print, "Hello ", **{"end": "", "flush": True})
buffer.submit(1000 / 1000, ts_print, "World!")
buffer.submit(500 / 1000, ts_print, "of the ", **{"end": "", "flush": True})
buffer.submit(200 / 1000, ts_print, "from ", **{"end": "", "flush": True})
buffer.submit(300 / 1000, ts_print, "the other side ", **{"end": "", "flush": True})
# ggf. await buffer.join() im Falle von Koroutinen
buffer.join()
print("Done.")
```

3.1. Implementation mit Threads

Implementieren Sie die Klasse `DelayedBuffer` mit Hilfe von `Threads` (und ggf. `Queues` bzw. `Locks`).

Implementieren Sie `ts_print` als Thread-sichere Variante von `print`.

3.2. Implementation mit Threadpool

Implementieren Sie die Klasse `DelayedBuffer` mit Hilfe eines `concurrent.futures.ThreadPool`s (und ggf. `Queues` bzw. `Locks`).

Implementieren Sie `ts_print` als Thread-sichere Variante von `print`. Wählen Sie ggf. eine andere Implementierung als in der vorherigen Aufgabe.

3.3. Implementation mit Koroutinen

Implementieren Sie die Klasse `DelayedBuffer` mit Hilfe von Koroutinen (und ggf. `asyncio.Queue`s).