Modul: Programmierung B-PRG
Grundlagen der Programmierung 1

V25 Nebenläufige Programme mit Python

Prof. Dr. Detlef Krömker
Professur für Graphische Datenverarbeitung
Institut für Informatik
Fachbereich Informatik und Mathematik (12)

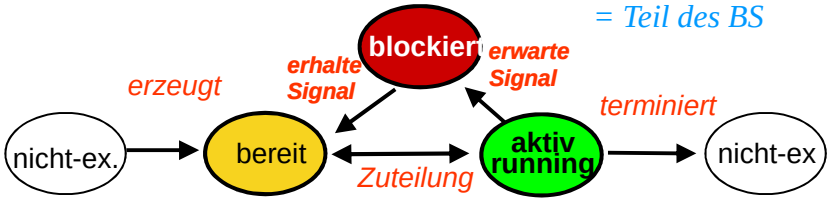
Rückblick:
Prozessmodelle -- Gewichtsklassen

- **schwergewichtiger Prozess** (*heavyweight process*)
  - Prozessinstanz und Benutzeradressraum bilden eine Einheit
  - Prozesswechsel ; **zwei** Adressraumwechsel: AR x ⇒ BS ⇒ AR y
  - ”klassischer“ UNIX Prozess
- **leichtgewichtiger Prozess** (*lightweight process*)
  - Prozessinstanz und Adressraum sind voneinander entkoppelt
  - Prozesswechsel; **einen** Adressraumwechsel: AR x ⇒ BS ⇒ AR x
  - Das ist ein "Kernfaden" (engl. kernel thread): Thread auf Kernebene
- **federgewichtiger Prozess** (*featherweight process*)
  - Prozessinstanzen und Adressraum bilden eine Einheit
  - Prozesswechsel ; **kein** Adressraumwechsel: AR x ⇒ AR x
  - Benutzerfaden (engl. user thread): Faden auf Benutzerebene

Prozeßzustände

Dispatcheraktionen

= Teil des BS



Prozesse warten in einer (Prioritäts-)Warteschlange ...

- auf den Prozessor (*bereit*)
- auf Daten des I/O-Geräts (*blockiert*)
- auf eine Nachricht (*blockiert*)
- auf ein Zeitsignal (*blockiert*)

Inhalt

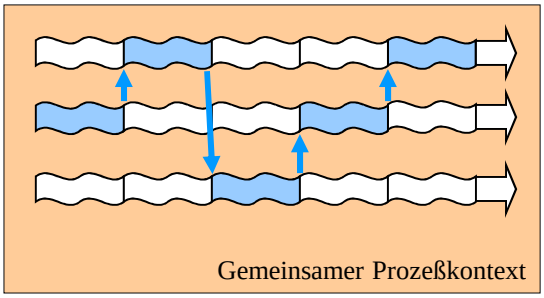
- **Thread – Was ist das eigentlich?**
- **Implementierung von Nebenläufigkeit**
  - **Threads**

  Umgang mit konkurrierenden Zugriffen
  - **Locks**
  - **(Queues)**
- **Deadlocks**

Zum Teil basieren diese Folien auf Vorlagen von Prof. Dr. Rüdiger Brause, Ifl Frankfurt und einem Vortrag von Stefan Schwarzer, Chemnitzer Linuxtage 2013 Chemnitz, Deutschland, 2013-03-17

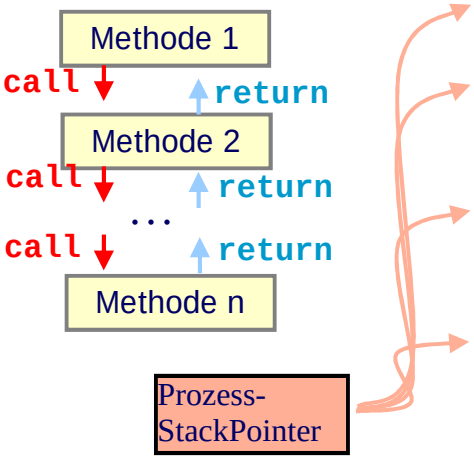
Threads ("Programm-Fäden", Coroutinen)

- **Kennzeichen: gemeinsamer** Prozeßkontext
  (Speicher-Addressbereich, Dateien (*file handles*), Ressourcen)
- asynchroner, paralleler, unterschiedlicher Programmverlauf (eigener *Stack*)



Nebenläufigkeit bei Methoden ?

- Methoden

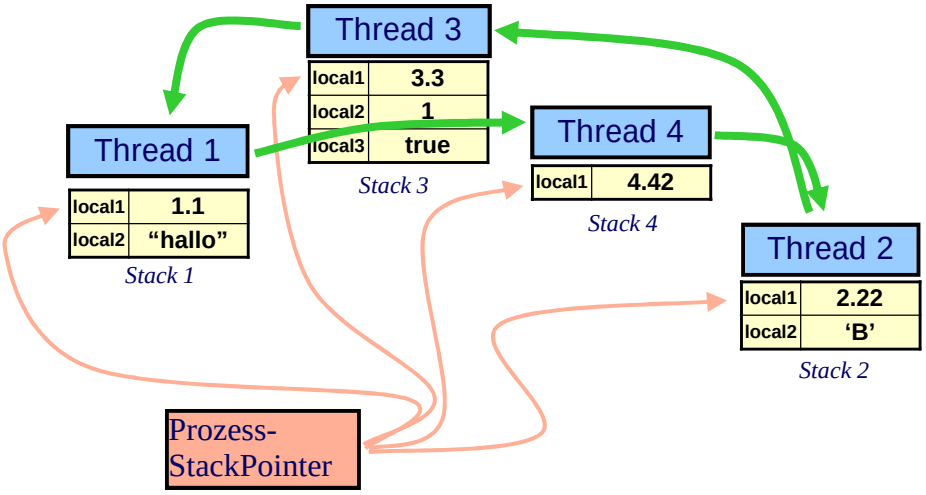


gemeinsamer Stack:
temp., lokale Variable

local1	1.1
local2	'B'

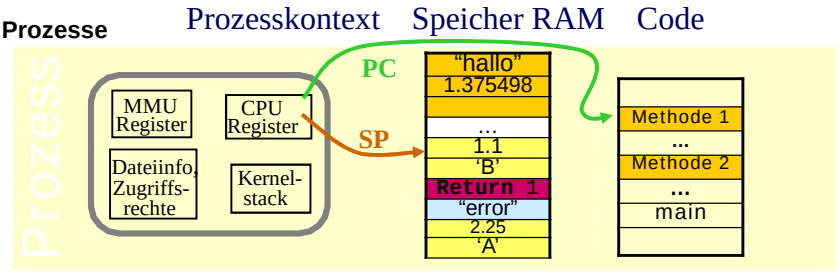
Nebenläufigkeit

- Coroutinen und Threads

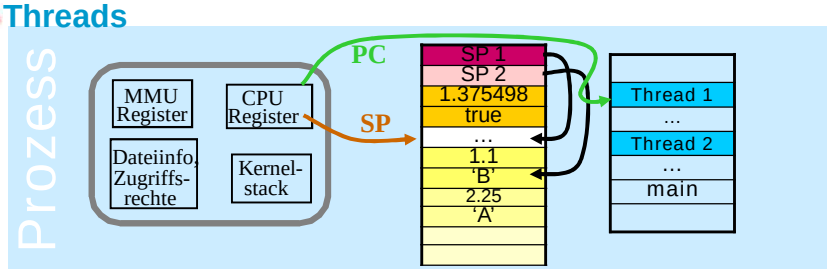


Nebenläufigkeit

- **Prozesse**



- **Threads**



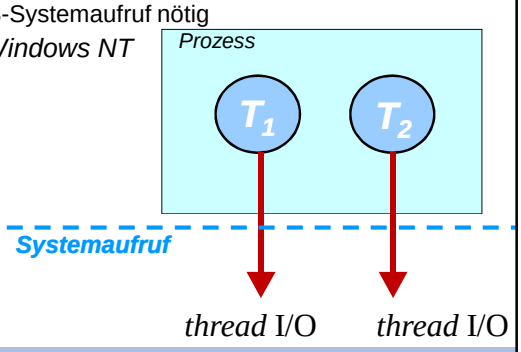
Thread- Typen: *heavyweight threads = lightweight process*

kontrolliert vom Betriebssystem *(z.B. Windows NT)*

Vorteil: Unabhängiger I/O aller threads T_1, T_2

Nachteil: langsamer BS-Systemaufruf nötig

das sind „***fibers***“ *in Windows NT*

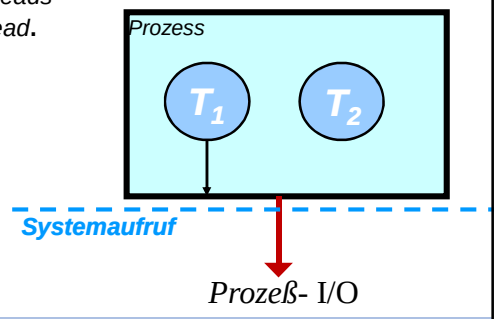


Thread- Typen: *lightweight threads = featherweight process)*

kontrolliert vom Benutzerprogramm *(z.B. Unix-Bibliothek)*

Vorteil: sehr schneller *thread*-Wechsel T_1, T_2

Nachteil: Blockieren aller *threads*
bei I/O-Warten von einem *thread***.**



Also:
Prozessmodelle -- Gewichtsklassen

- **schwergewichtiger Prozess** (*heavyweight process*)
  - Prozessinstanz und Benutzeradressraum bilden eine Einheit
  - Prozesswechsel ; **zwei** Adressraumwechsel: AR x ⇒ BS ⇒ AR y
  - ”klassischer“ UNIX Prozess
- **leichtgewichtiger Prozess** (*lightweight process*)
  - Prozessinstanz und Adressraum sind voneinander entkoppelt
  - Prozesswechsel; **einen** Adressraumwechsel: AR x ⇒ BS ⇒ AR x
  - Das ist ein "Kernfaden" (engl. kernel thread): Thread auf Kernebene
- **federgewichtiger Prozess** (*featherweight process*)
  - Prozessinstanzen und Adressraum bilden eine Einheit
  - Prozesswechsel ; **kein** Adressraumwechsel: AR x ⇒ AR x
  - Benutzerfaden (engl. user thread): Faden auf Benutzerebene

Wann ist Nebenläufigkeit sinnvoll?

1. **CPU-intensive Aufgaben** – dann, wenn man sie auf mehrere Prozessorkerne verteilen kann.
2. **Ein-/Ausgabe intensive Aufgaben** – Während Daten ausgetauscht werden, können andere Programmteile weiterlaufen.
3. **Interaktiven Programmen** wg. Bedienbarkeit – Während ein GUI-Programm eine Hintergrundaufgabe ausführt, soll es für den Nutzer bedienbar bleiben.

Begrifflichkeiten

- **Nebenläufigkeit / Concurrency:** mehrere Ausführungsstränge – aber nicht unbedingt (echt) gleichzeitige Ausführung.
- **Parallelism**: (echt) gleichzeitige Ausführung von Code (beispielsweise auf verschiedenen CPU-Kernen)
- **Atomare Operation / Atomic Operation**: ein Vorgang, der nicht von einem anderen Thread oder Prozess unterbrochen werden **kann**
- **Race Condition**: Ein kritischer Wettlauf (auch *race hazard*) ist eine Konstellation, in der das Ergebnis einer Operation vom zeitlichen Verhalten bestimmter anderer Operationen abhängt: Threads oder Prozesse kommen sich also gegenseitig in die Quere.

Grundproblem für Race Conditions

- Konkurrierende Zugriffe auf Ressourcen aus zwei oder mehr Ausfürungssträngen heraus müssen abgesichert werden, um RaceConditions zu verhindern.
- Konkurrierend ist insbesondere alles, was den Zustand einer Ressource ändert. Unterscheidung in lesende und schreibende Zugriffe ist mitunter irreführend.
  - Beispiel: Lesender Zugriff auf eine Datei ändert den Dateizeiger.
- Ressourcen sind zum Beispiel:
  - Einzelwerte und Datenstrukturen im Speicher
  - Dateien
  - Sockets
  - Bildschirm/Fenster

Multithreading

- Nebenläufigkeit innerhalb eines Prozesses
- Modul `threading` (in der Standardbibliothek) nicht `_thread` (low-Level Bibliothek)
- Die Ausführungsstränge können auf Daten im Speicher zugreifen.
- Achtung: Bei CPython kommt das **Global Interpreter Lock (GIL)** zum Tragen. Das GIL verhindert die **parallele Ausführung** von Python-Code. Es wird **aber** bei I/O-Operationen freigegeben.
- Auch C-Erweiterungen können das GIL freigeben.
- Das GIL begrenzt also nur bei CPU-begrenzten Abläufen; bei I/O-begrenzten Abläufen ist es eher unproblematisch.

Multithreading-Beispiel (1)

Beispiel ist von Tung Le Trong. Danke.

```
from threading import Thread, Semaphore

def main():  # starting with  not synchronized
    t1 = Thread(target=thread_no_sem, args=("Betriebs",))
    t2 = Thread(target=thread_no_sem, args=("systeme\n",))
    # start threads
    t1.start(); t2.start()
    # wait threads to finish
    t1.join();  t2.join()

def thread_no_sem(text):
        for i in range(5):
                print(text, end="")
```

Der Konstruktor `Thread()`

Die Klasse `Thread()` repräsentiert eine Aktivität die in als **leichtgewichtiger Prozess** (*lightweight process = heavyweight thread)*) läuft

Zwei Wege, diese Aktivität näher zu spezifizieren:

- Ein callable-Objekt (Funktion, Methode) zu übergeben oder
- überschreiben der Methode `run() in der Unterklasse.`

Nach dem Erzeugen des Objektes muss die Aktivität noch mit der Methode `start()` gestartet werden.

Andere Threads können die Thread's `join()` Methode rufen. Dies halt den rufenden Thread solange an, bis der gerufene Thread terminiert.

Argumente des Konstruktors

- `class threading.Thread(group=None, target=None, name=None, args=(), kwargs={}, *, daemon=None)`

This constructor should always be called with keyword arguments. Arguments are:

- `group` should be None; reserved for future extension when a ThreadGroup class is implemented.
- `target` is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.
- `name` is the thread name. By default, a unique name is constructed of the form "Thread-N" where N is a small decimal number.
- `args` is the argument tuple for the target invocation. Defaults to ().
- `kwargs` is a dictionary of keyword arguments for the target invocation. Defaults to {}.
- If not None, `daemon` explicitly sets whether the thread is daemonic. If None (the default), the daemonic property is inherited from the current thread.
- If the subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

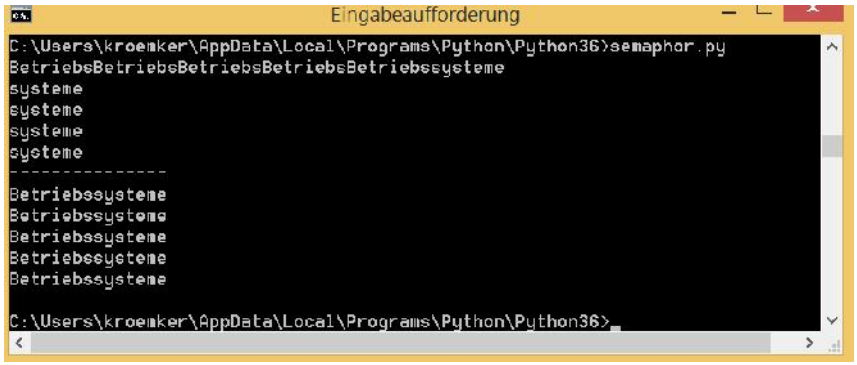
Multithreading-Beispiel (2)

```
# prosecution of the main method
print("--------------")  # again, now synchronized

    a, b = Semaphore(1), Semaphore(0)
    t1 = Thread(target=thread, args=("Betriebs", a, b))
    t2 = Thread(target=thread, args=("systeme\n", b, a))
    t1.start(); t2.start()
    t1.join();  t2.join()

def thread(text, a, b):
    for i in range(5):
         a.acquire()
         print(text, end="")
         b.release()

if __name__ == '__main__':
        main()
```


Die Semaphore "Lock Objects"

- Dies ist ein low level-Synchronisations-Primitiv.
- Dieses lock hat zwei Zustände: “locked” oder “unlocked”. Default ist “unlocked”.
- Es hat zwei Methoden: `acquire()` und `release()`.
- Im "locked" Zustand: `acquire()`blockiert den Thread, bis eine `release()Methode` in einem anderen Thread dieses Semaphor freigibt.
-

Dieses Programm gestartet liefert: (in der Betriebssystem-Konsole)



Deadlock

Prinzip:

- Ein Deadlock entsteht, wenn sich Ausführungsstränge (Threads) gegenseitig Ressourcen vorenthalten.
- Ausführungsstränge in diesem Sinn können Threads auch Prozesse sein.

Beispiel:

- zwei Threads lesen aus einer Eingabedatei und schreiben in eine Ausgabedatei.
- Dabei sind beide Dateien mit je einem Lock gesichert.

Locks vs. Queues

Solange ein Lock gehalten wird, kann kein anderer Thread den gesicherten Code-Abschnitt ausführen.

Threads müssen also aufeinander warten.

Keine Parallelisierung des gesicherten Code-Abschnitts

Gefahr von Deadlocks (umso wahrscheinlicher, je länger die beteiligten Locks gehalten werden)

Queues reduzieren diese Probleme, weil das implizite Lock (beziehungsweise Locks bei getrennt gesicherten "Enden") nur kurzzeitig gehalten wird.

Die Verwendung von **Queues** erleichtert es, die Funktionsweise des nebenläufigen Codes zu verstehen (später mehr dazu).

Ein Hinweis:

- Die mitgelieferte Entwicklungsumgebung von Python – IDLE – ist **nicht** thread-sicher, so sind zum Beispiel die Ausgabefunktionen nicht synchronisiert.
- Insbesondere bringt die gleichzeitige Ausführung von print-Anweisungen IDLE zum Absturz!

→ **Eigene Programme, die Threads benutzen, von der Kommandozeile aus starten!**

Umsetzungen von Nebenläufigkeit in Python Multiprocessing

- Nebenläufigkeit zwischen verschiedenen Prozessen
- Modul `multiprocessing` (in der Standardbibliothek)
- Datenaustausch zwischen Prozessen erfolgt über Nachrichten oder
- über Shared Memory
- Bei Austausch von Nachrichten müssen diese serialisiert werden (in Python meist mit dem pickle-Modul). Achtung: **Serialisierung ist Zusatzaufwand.**
- Vorteil bei Multiprocessing: keine prinzipiellen Einschränkungen der gleichzeitigen Ausführung, auch nicht bei CPU-Begrenzung
- Als Mischform sind natürlich auch mehrere Prozesse mit einem oder mehreren Threads möglich.

Semaphore: Python

- **`Class threading.Semaphore(v)`**
  Datenstruktur mit zählendem Semaphor **`s`** + Thread-*Warteschlange*

  Methoden

  - **`acquire()`** — *Wenn s>0, dekrementiere s .*
    Wenn s=0, blockiere Thread in Warteschlange.
  - **`release()`** — *Wenn s>0, inkrementiere s .*
    Wenn s=0 und ein Thread wartet, hole einen Thread aus der Warteschlange.

- **`Class multiprocessing.Semaphore(v)`**
  Geeignet für Prozesse und Multiprozessormaschinen

- ”Prozesse“ erhalten Eingabedaten und/oder erzeugen Ausgabedaten.”
- Prozesse“ können beispielsweise auch Python-Threads sein.
- Daten¨ubertragung zwischen Prozessen durch Nachrichten-Übertragung (Message Passing).
  - Kein gemeinsamer Zustand
  - Keine Race Conditions
  - Varianten
    - Concurrent Sequential Processes (CSP)
    - Dataflow
    - Flow-Based Programming
- Python-Bibliotheken: PyCSP, PyF, DAGPype, . . .

Umsetzungen von Nebenläufigkeit in Python GUI

- Event Loop: Schleife ("Main Loop“) registriert Ereignisse (Beispiele: Mausklick, Tastatur Klick), z.B. bei TKinter, PyGTK, PyQt, wxPython.
- Je nach Ereignis wird ein ”Handler“ dafür aufgerufen, der die Verarbeitung des Ereignisses übernimmt.
- Die Kontrolle geht (spätestens) danach an die Hauptschleife zurück.
- Programmfluss möglicherweise schwer zu überblicken, wenn die Handler voneinander abhängen.

Eine Event Loop dient der Steuerung von Nebenläufigkeit, nicht von Parallelität (Ausnutzung mehrerer CPU-Kerne).

Zusätzliche Handhabung von Threads oder Prozessen wird nicht vereinfacht.

Fragen ?

und (hoffentlich) Antworten

Ausblick für Montag:

Was noch fehlt:

Interrupts und Exceptions

und, herzlichen Dank für Ihre Aufmerksamkeit!