FRIEDRICH-SCHILLER-
UNIVERSITÄT
JENA

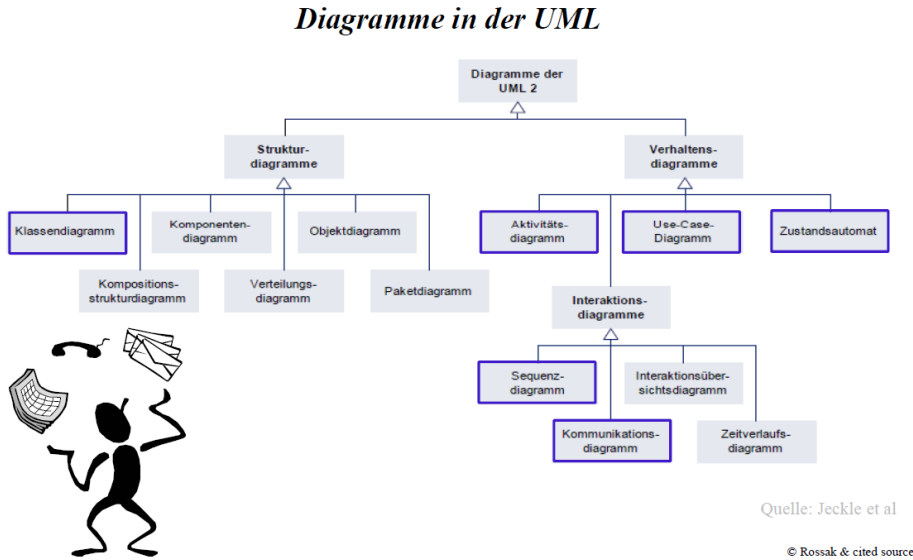
Einführung in Python

7. Vorlesung

Threads und Multiprocessing

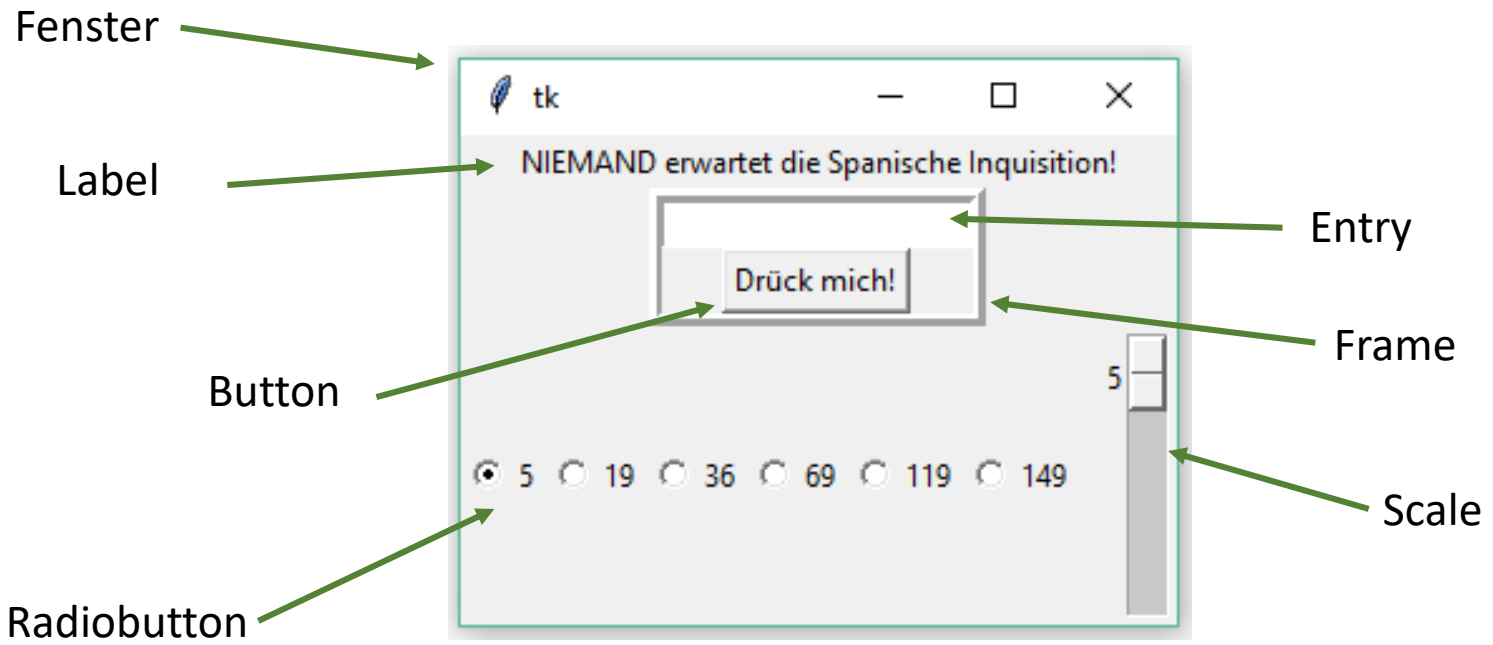
Wiederholung letztes Mal

- Objektorientierte Software-Entwicklung ist ein komplexer Prozess
- Wird oft in verschiedene Phasen eingeteilt um ein gegebenes Problem zu modellieren, spezifizieren und anschließend zu implementieren
- Die UML bietet eine Vielzahl an Formalismen und Diagrammen zur Beschreibung von Softwareprojekten



Wiederholung letztes Mal

- GUIs erlauben eine multimediale, asynchrone interaktive Kommunikation mit dem Computer
- Eine GUI besteht aus einem Fenster und Widgets die in einer Master-Slave-Hierarchie angeordnet sind



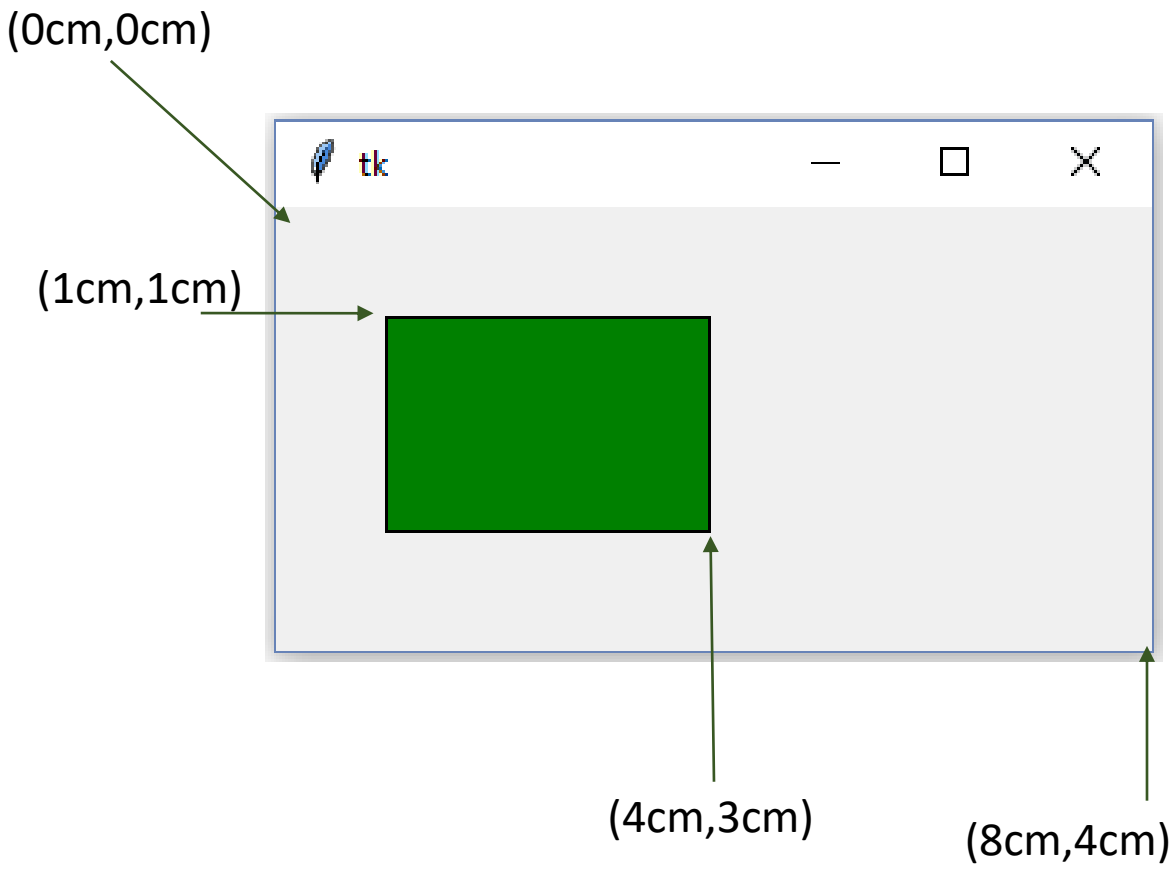
Wiederholung letztes Mal

- Tkinter ist die Standardbibliothek für GUI-Programmierung in Python
- Es gibt verschiedene Layoutmanager zur Positionierung der Widgets in einem Fenster
- Events lösen ein bestimmtes Systemverhalten aus und werden durch Event-Sequenzen beschrieben und an Eventhandler gebunden

<*Modifizierer*- Typ -*Qualifizierer*>

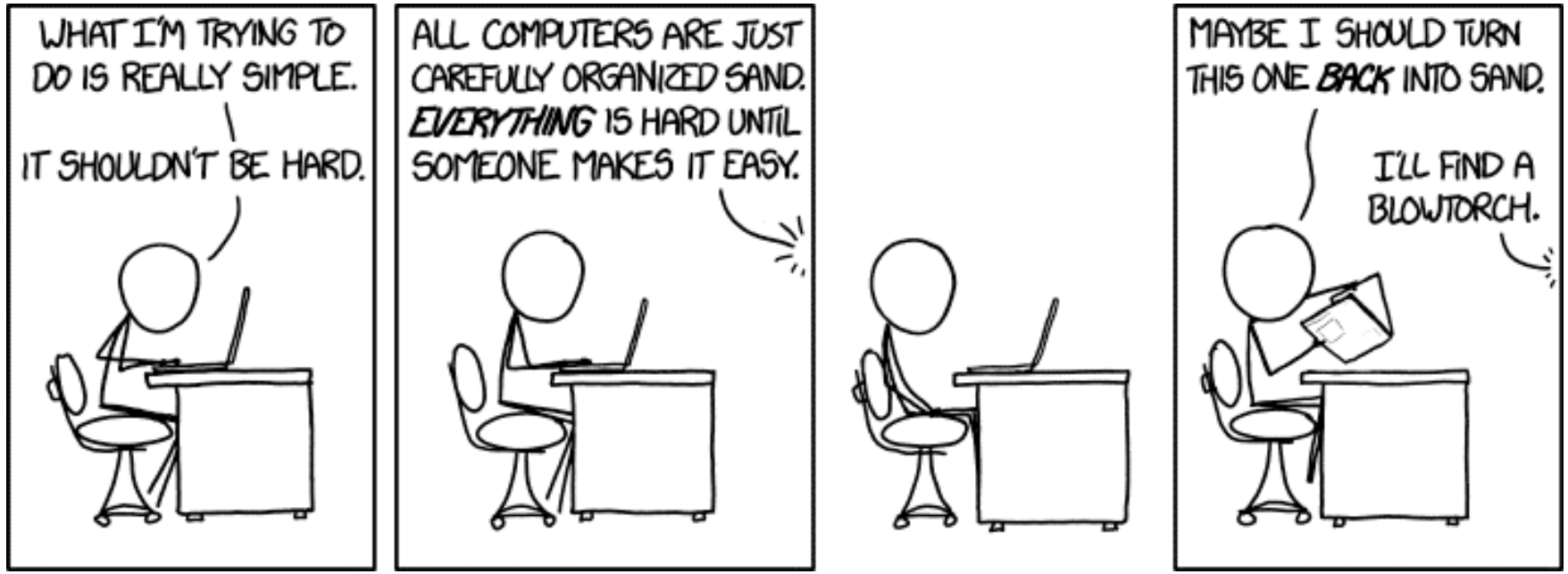
Wiederholung letztes Mal

- Mit der Klasse Canvas von Tkinter lassen sich Grafiken in GUIs erstellen und als Grafikdatei abspeichern

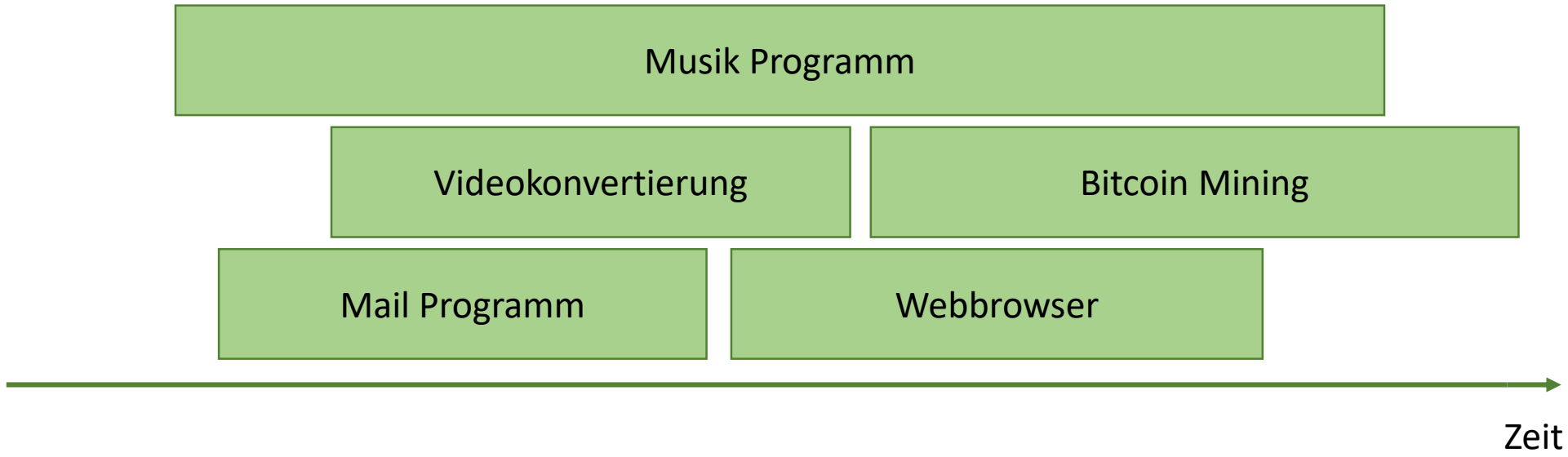


- Mit der Klasse PhotoImage lassen sich auch Pixelgrafiken einbinden und manipulieren

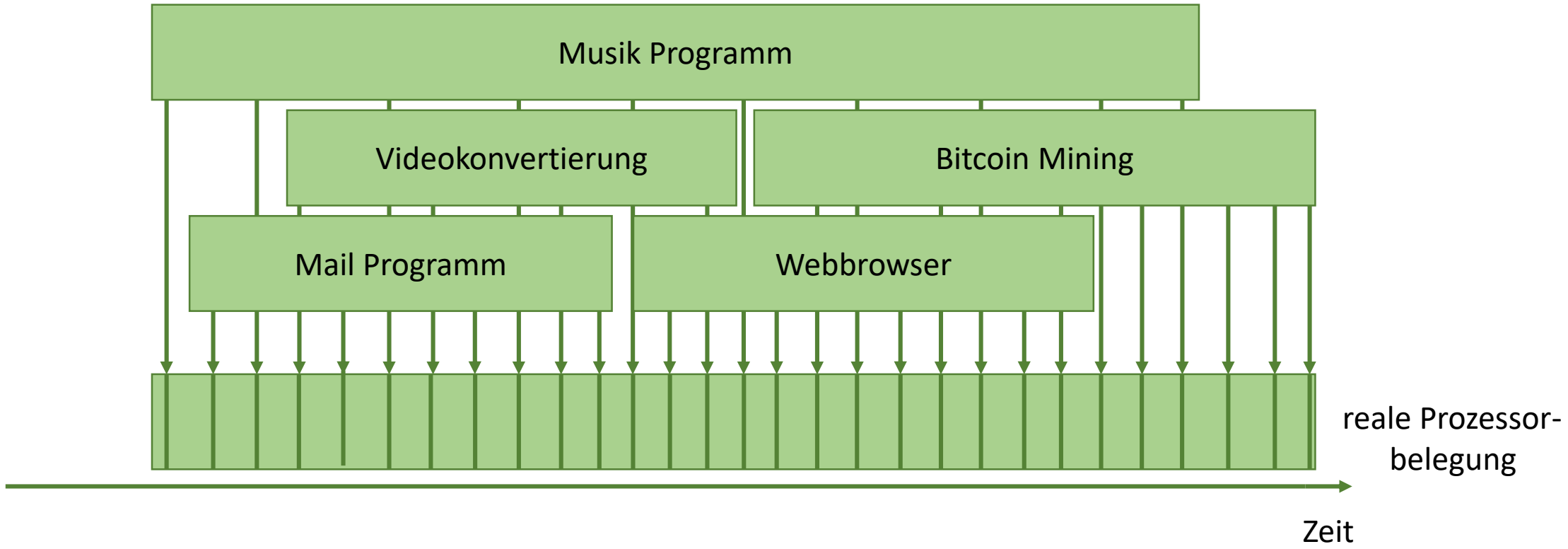
Prozesse und Threads



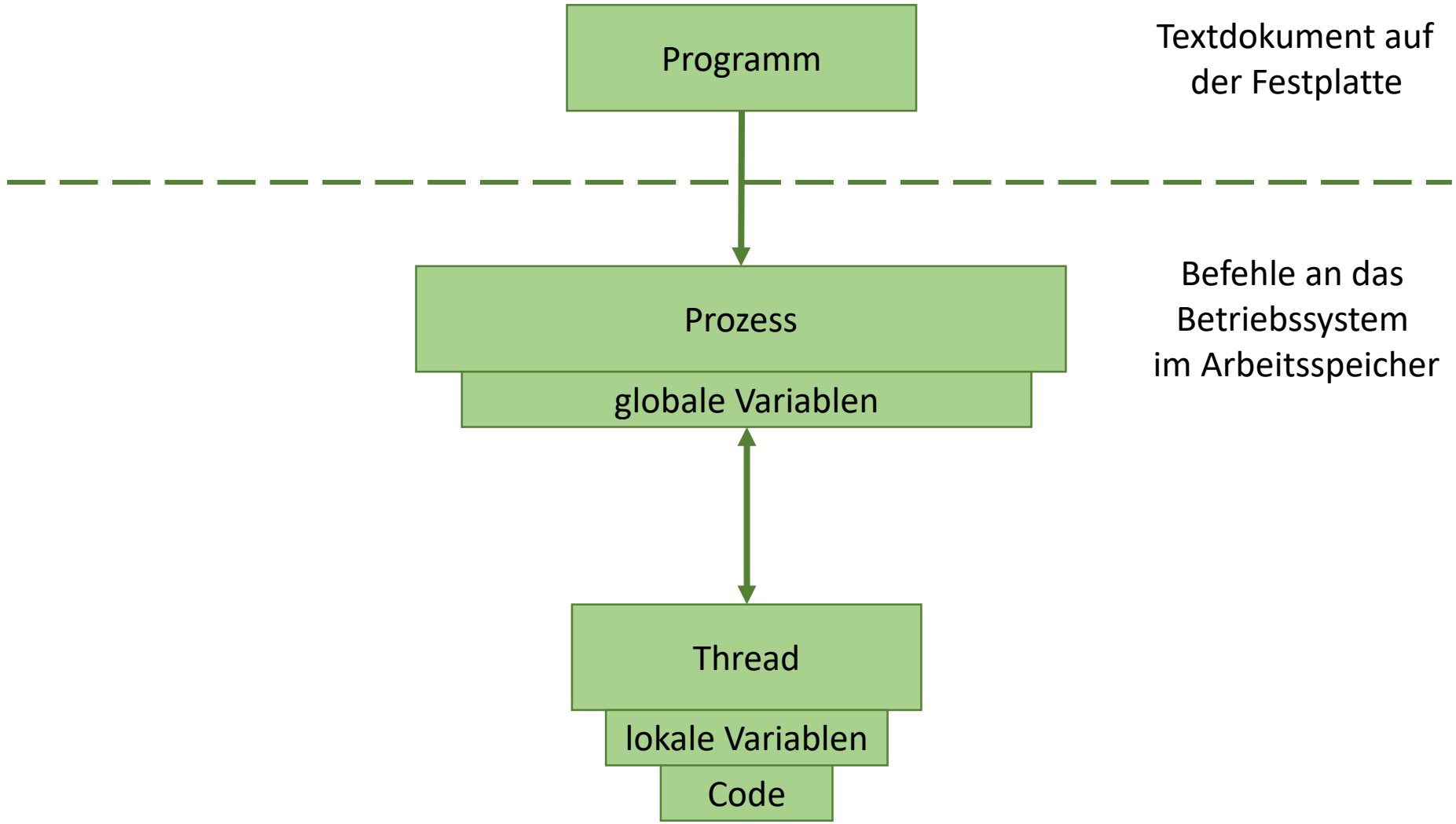
Parallele Programmierung



Parallele Programmierung



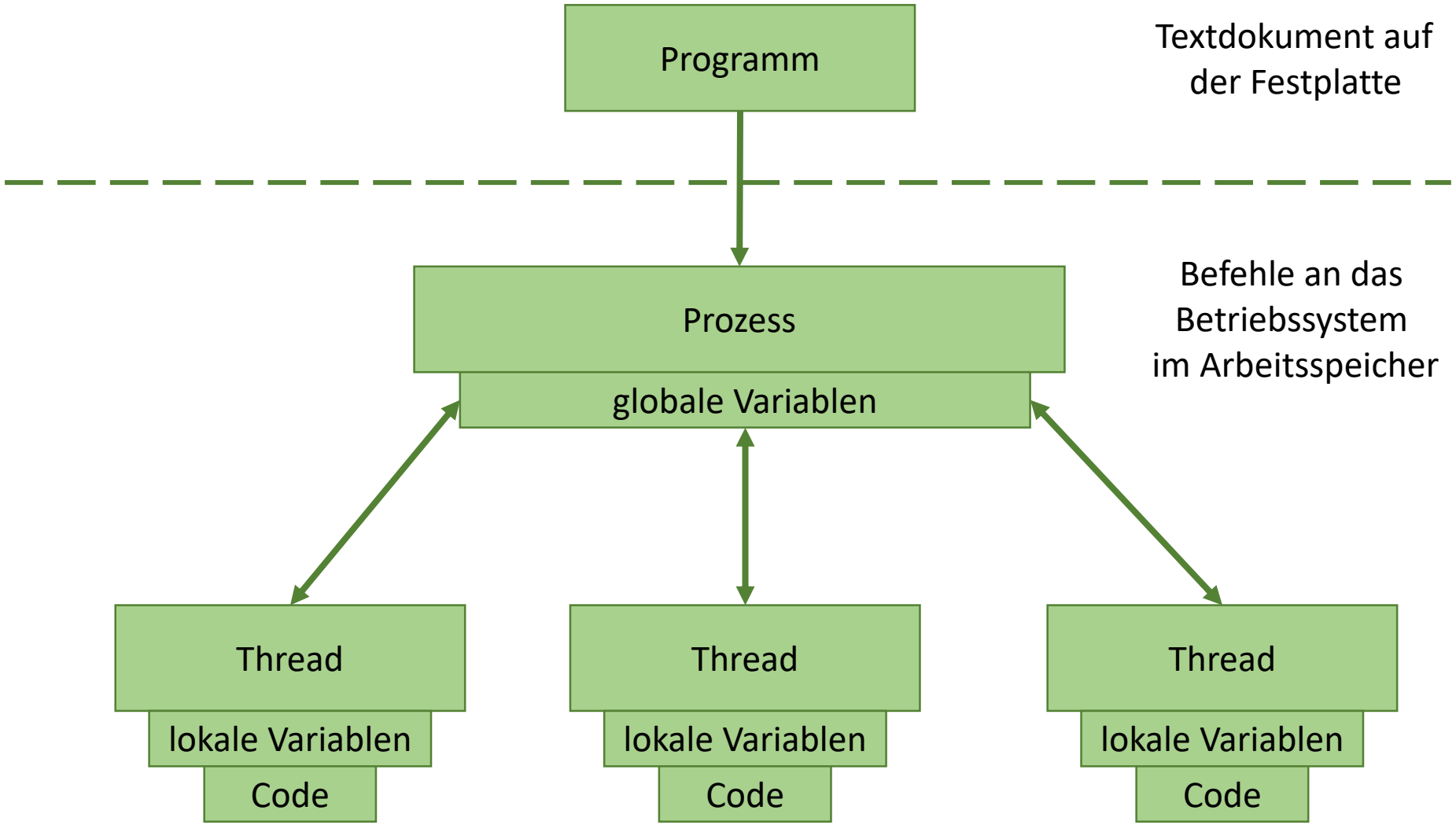
Ein Prozess


Programm
Textdokument auf
der Festplatte
Prozess
globale Variablen
Befehle an das
Betriebssystem
im Arbeitsspeicher
Thread
lokale Variablen
Code

Ein Prozess

- Threads sind (Teil)Prozesse und Prozesse sind Programme in Ausführung
- Jeder Prozess hat eine ID und einen abgeschotteten Bereich im Arbeitsspeicher (Prozessumgebung)
- Ein Prozess ist von der Zeit abhängig: Er wird erzeugt, nimmt verschiedene Zustände an und stirbt (in der Regel) irgendwann

Ein Prozess


Programm
Textdokument auf der Festplatte
Prozess
globale Variablen
Befehle an das Betriebssystem im Arbeitsspeicher
Thread
lokale Variablen
Code
Thread
lokale Variablen
Code
Thread
lokale Variablen
Code

Ein Prozess

- Jeder Prozess besitzt mindestens einen Thread (sich selbst), er kann aber auch mehrere andere Threads starten

- Threads werden vom Betriebssystem (scheinbar) gleichzeitig ausgeführt

- Threads die zu einem Prozess gehören, teilen sich den gleichen Speicherbereich (haben Zugriff auf alle globalen Variablen)

Threads in Python

- In Python gibt es zwei verschiedene Implementierungen für Threads:
  - Das Modul *thread* (Python 2) bzw. *_thread* (Python 3) betrachtet Threads als Funktionen
  - Das Modul *threading* implementiert Threads als eigenständige Objekte und erlaubt komplexere parallele Verarbeitungen

Das thread-Modul

- Die Funktion *start_new_thread()* erlaubt es einzelne Funktionen in separaten Threads zu starten
- Die Funktion hat als Wiedergabewert den Namen (die ID) des neu erzeugten Threads
- Endet die Funktion, wird der dazugehörige Thread automatisch gelöscht

start_new_thread(function, args)

Referenz auf die auszuführende Funktion

Tupel mit den Argumenten der Funktion

Das thread-Modul – Beispiel Zähler

```
from time import *
from tkinter import *

def zählen():
    global f
    for i in range(11):
        f.zahl.set(str(i))
        sleep(1)

f = Tk()
f.zahl = StringVar()
f.l = Label(f, textvariable=f.zahl)
f.b = Button(f, command=zählen, text='Los!')
f.l.pack()
f.b.pack()
```

Das thread-Modul – Beispiel Zähler

```
import _thread

def zählen():
    global f
    for i in range(11):
        f.zahl.set(str(i))
        sleep(1)
def zählen_threaded():
    _thread.start_new_thread(zählen, ())

f = Tk()
f.zahl = StringVar()
f.l = Label(f, textvariable=f.zahl)
f.b = Button(f, command=zählen_threaded, text='Los!')
f.l.pack()
f.b.pack()
f.mainloop()
```

Das thread-Modul – Beispiel Heron

```
from _thread import start_new_thread

def heron(a, b):
    old, new, eps = 1, 1, 0.0000001
    while True:
        old, new = new, (new + a/new) / 2.0
        print(f'{b}: {old} {new}')
        if abs(new - old) < eps:
            break
    return(new)

start_new_thread(heron,(99,'Thread 1'))
start_new_thread(heron,(999, ,'Thread 2'))
start_new_thread(heron,(1733, ,'Thread 3'))
start_new_thread(heron,(17334, ,'Thread 4'))
input()
```

Das thread-Modul – Beispiel Heron

```
from _thread import start_new_thread

numThreads = 0
def heron(a, b):
    global numThreads
    numThreads += 1
    #Code wie vorher
    numThreads -= 1
    return(new)

start_new_thread(heron,(99,'Thread 1'))
start_new_thread(heron,(999, ,'Thread 2'))
start_new_thread(heron,(1733, ,'Thread 3'))
start_new_thread(heron,(17334, ,'Thread 4'))

while numThreads > 0:
       pass
```

Das thread-Modul – Lock-Objekte

- Mit *Lock*-Objekten lassen sich kritische Bereiche (*critical sections*) eines Programms markieren

- Solche markierten Codeabschnitte werden so ausgeführt als ob sie atomar wären, d. h. sie können nicht in Teilschritte aufgespaltet werden, sondern müssen als Ganzes abgearbeitet werden bevor ein anderer Thread weitermachen darf

- Lock-Objekte werden mit der Funktion *thread.allocate_lock()* erzeugt

Das thread-Modul – Beispiel Heron

```
from _thread import start_new_thread, allocate_lock

numThreads = 0
threadStarted = False
lock = allocate_lock()
def heron(a, b):
    lock.acquire()
    global numThreads, threadStarted
    numThreads += 1
    threadStarted = True
    lock.release()
    #Code wie vorher
    lock.acquire()
    numThreads -= 1
    lock.release()
    return(new)

start_new_thread(heron,(99,))
start_new_thread(heron,(999,))
start_new_thread(heron,(1733,))
start_new_thread(heron,(17334,))

While (not threadStarted) or (numThreads > 0):
        pass
```

Das thread-Modul – Lock-Objekte

- Globale Variablen auf die mehrere Threads zugreifen können, sollten immer mit *critical sections* geschützt werden, um schwer reproduzierbar und lokalisierbare Fehler zu vermeiden

- Wenn mehrere Lock-Objekte benutzt werden, kann es passieren das sich ein Programm im so genannten *Deadlock* aufhängt, weil zwei geblockte Threads gegenseitig aufeinander warten

Das Modul threading

- *threading* ist die objektorientierte Schnittstelle für Threads, d. h. Instanzen eigener Klassen kann man die Eigenschaften von Threads geben

- Die Klasse *Thread* aus threading besitzt zwei wichtige Methoden:
  - *start()*: Anlegen und starten des Threads
  - *run()*: ‘Inhalt‘ des Threads, also der auszuführende Code

Das Modul threading

- Ähnlich zum Modul *_threads* lassen sich auch einzelne Funktionen in einem Thread-Objekte starten und ausführen

```
import time
from threading import Thread

def schlafen(i):
    print(f'Thread {i} schläft für 5 Sekunden')
    time.sleep(5)
    print('Thread {i} ist aufgewacht')

for i in range(10):
    t = Thread(target=schlafen, args=(i,))
    t.start()
```

Das Modul threading – Beispiel Primzahl

```
import threading

class Primzahl(threading.Thread):
    def __init__(self, zahl):
        threading.Thread.__init__(self)
        self.zahl = zahl

    def run(self):
        counter = 2
        while counter**2 < self.zahl:
            if self.zahl % counter == 0:
                print(f'{self.zahl} ist keine Primzahl, da {self.zahl} = {counter}'\
                      f' * {self.zahl/counter}')
                return()
            counter += 1
        print(f'{self.zahl} ist eine Primzahl')
```

Das Modul threading – Beispiel Primzahl

- Mit der Thread-Methode *join()* zwingt man den Hauptprozess so lange mit seiner Beendung zu warten, bis der Thread der damit ‘markiert‘ wurde, beendet wurde

```
threads = []
While True:
    eingabe = int(input('Zahl: '))
    if eingabe < 1:
        break

    thread = Primzahl(eingabe)
    threads.append(thread)
    thread.start()

for x in threads:
    x.join()
```

Das Modul threading – Locking

- Ähnlich zum Modul *_threads* lassen sich auch Lock-Objekte erzeugen im kritische Bereiche zu markieren

- Mit der Funktion *threading.Lock()* erzeugt man ein neues Lock-Objekt, welches wie vorher die zwei Funktionen *acquire()* und *release()* besitzt

- Lock-Objekte sind mit dem *with*-Schlüsselwort kompatibel

```
lock = threading.Lock()

with lock:
    #Do some stuff
```

Das Modul threading – Wieder Beispiel Heron

```
import threading

numThreads = 0
threadStarted = False
lock = threading.Lock()
def heron(a, b):
    with lock:
        global numThreads, threadStarted
        numThreads += 1
        threadStarted = True

    # Heron Code wie vorher
    with lock:
        numThreads -= 1
    return(new)

threading.thread(heron,(99,))
threading.thread(heron,(999,))
threading.thread(heron,(1733,))
threading.thread(heron,(17334,))

While (not threadStarted) or (numThreads > 0):
        pass
```

Das Modul threading

- Problem: Wenn sehr viele Threads auf einmal laufen, so können sich diese gegenseitig verlangsamen, da sich alle die gleiche Rechenzeit teilen müssen

- Lösung: Wir begrenzen die Zahl an gleichzeitig arbeitenden Threads und lassen neue Aufgaben automatisch an freie Threads zuweisen

Worker-Threads und Queues

- Queues (Warteschlangen) werden in Python durch das gleichnamige Modul zur Verfügung gestellt

- Queues haben drei wichtige Methoden:
  - *put()*: Hinzufügen einer neuen Aufgabe in die Queue
  - *get()*: Die nächste Aufgabe wird aus der Queue ausgegeben
  - *task_done()*: Ist ein Thread mit einer Aufgabe fertig, teilt er dies der Queue mit, damit sie die verarbeitete Aufgabe entfernen kann

Worker-Threads und Queues

```
import threading
import queue

class Primzahl(threading.Thread):
    ergebnis = {}
    lock = threading.Lock()
    q = queue.Queue()
                                #__init__ Methode wie vorher
    def run(self):
        while True:
            zahl = Primzahl.q.get()  #Anfragen der nächsten Aufgabe aus der Queue; hier nächste Zahl
            erg = self.isPrime(zahl)
            with Primzahl.lock:
                Primzahl.ergebnis[zahl] = erg
                Primzahl.q.task_done()

    def isPrime(self, zahl):
        counter = 2
        while counter**2 < zahl:
            if zahl % counter == 0:
                return(f'{zahl} ist keine Primzahl, da {zahl} = {counter} * {zahl/counter}')
            counter += 1
        return('Primzahl‘)

#Fortsetzung des Codes auf der nächsten Folie
```

Worker-Threads und Queues

```
threads = [Primzahl() for i in range(2)]          #Erzeugen von 2 Primzahl-Objekten
for x in threads:
    x.setDaemon(True)
    x.start()                #Starten der Primzahl-Objekte, d.h. ihre run-Methoden werden ausgeführt

While True:
    eingabe = int(input('Zahl: '))
    if eingabe < 0:
        break
    elif eingabe == 0:
        print('Status')
        with Primzahl.lock:
            for key in Primzahl.ergebnis:
                print(f‘{key} -- {Primzahl.ergebnis.get(key, ‘In Arbeit‘)}\n')

    Primzahl.q.put(eingabe)                       #Die nächste Zahl wird in die Queue gesteckt

Primzahl.q.join()          #So lange es noch Threads gibt die Aufgaben aus der Queue bearbeiten,
                           #wird der Hauptprozess nicht beendet
```

- Queues sind synchronisierte Objekte, d. h. sie kümmern sich selbst darum das sicher auf sie zugegriffen wird

Weitere threading Funktionen

Name	Funktion
threading.active_count()	Gibt die Anzahl der laufenden Threads eines Prozesses wieder.
threading.current_thread()	Gibt die Referenz auf das Thread-Objekt des dazugehörigen (Sub)Prozesses wieder,
threading. enumerate()	Gibt eine Liste aller aktiven Threads zurück, einschließlich des Hauptprozesses und Demon-Threads und ausschließlich nicht gestarteter Threads.
threading. local()	Gibt den kompletten lokalen Namensraum eines Threads zurück.
threading.Event()	Erzeugt ein Event-Objekt zur Steuerung von Threads.
threading.Timer()	Erzeugt ein Timer-Objekt zur Steuerung von Threads.

Thread-Events

- Thread-Events bieten eine einfach Möglichkeit für Threads miteinander zu kommunizieren

- Die Funktion *threading.Event()* erzeugt ein Event-Objekt, auf das alle Threads zugreifen können:
  - *set()*: Versetzt das Event in den aktiven Modus.
  - *clear()*: Versetzt das Event in den inaktiven Modus.
  - *wait()*: Ein Thread wartet darauf das ein Event in den aktiven Modus wechselt.

- Ein Thread, der die *wait*-Methode eines Event-Objekts aufruft, wird so lange unterbrochen, bis ein anderer Thread das Event mit *set()* auslöst

Thread-Events

```
import threading

def function1(x, event1, event2):
    #Do some cool stuff
    event1.set()                        #Aktiviere das Objekt event1
    event2.wait()                       #Warte das function2 das Objekt event2 aktiviert
    #Do more cool stuff in sync with thread2

def function2(y, event1, event2):
    #Do some cool stuff
    event2.set()                        #Aktiviere das objekt event2
    event1.wait()                       #Warte das funtion1 das Objekt event1 aktiviert
    #Do more cool stuff in sync with thread1

event1 = threading.Event()
event2 = threading.Event()

thread1 = threading.Thread(target=function1, (x, event1, event2))
thread2 = threading.Thread(target=function2, (y, event1, event2))
thread1.start()
thread2.start()
```

Thread-Conditions

- Mit Thread-Conditions können sich Threads gezielt ansprechen, die sich gemeinsame Ressourcen teilen

- Ein Conditions-Objekt funktioniert ähnlich zu einem Lock-Objekt und hat ähnliche Funktionen:
  - *wait()*: Ein Thread wartet darauf von einem anderem Thread geweckt zu werden.
  - *notify(n=1)*: Ein Thread weckt *n* Threads die auf eine bestimmte Condition warten.
  - *notifyAll()*: Ein Thread weckt alle Threads die auf eine bestimmte Condition warten.
  - *acquire()*: Wie bei Lock-Objekten.
  - *release()*: Wie bei Lock-Objekten.

Thread-Conditions

```
import threading

def function1(x, cond):
    with cond:
        cond.wait() #function1 wartet bis eine andere Funktion das Objekt cond aktiviert


def function2(y, event1, event2):
    with cond:
        cond.notifyAll() #Alle Funktionen die auf cond warten, können nun weitermachen

condition = threading.Condition()

thread1 = threading.Thread(target=function1, (x, condition))
thread2 = threading.Thread(target=function2, (y, condition))
thread1.start()
thread2.start()
```

Thread-Timer

- Die *threading.Timer()*-Funktion lässt eine Funktion wie durch *threading.Thread()* in einem neuem Thread starten, allerdings zeitverzögert

threading.Timer(interval, function, args)

Thread-Timer

- Als Zeitintervall können sowohl Interger- als auch Float-Variablen übergeben werden

```
import time, threading

def wecker(zeit):
    print('KLINGELING!')
    print(f'Der Wecker wurde um {zeit} Uhr gestellt.')
    print(f'Es ist nun {time.strftime("%H:%M:%S")} Uhr')

timer = threading.Timer(10.19, wecker, [time.strftime('%H:%M:%S')])
timer.start()
```

Thread-Timer

- Die Methode *cancel()* lässt einen Timer auch wieder komplett abbrechen, solange er noch nicht ausgelöst wurde

```
import time, threading

def wecker(zeit):
    print('KLINGELING!')
    print(f'Der Wecker wurde um {zeit} Uhr gestellt.')
    print(f'Es ist nun {time.strftime("%H:%M:%S")} Uhr')

timer = threading.Timer(10, wecker, [time.strftime('%H:%M:%S')])
timer.start()

time.sleep(5)
timer.cancel()
```

Multiprocessing

Multiprocessing

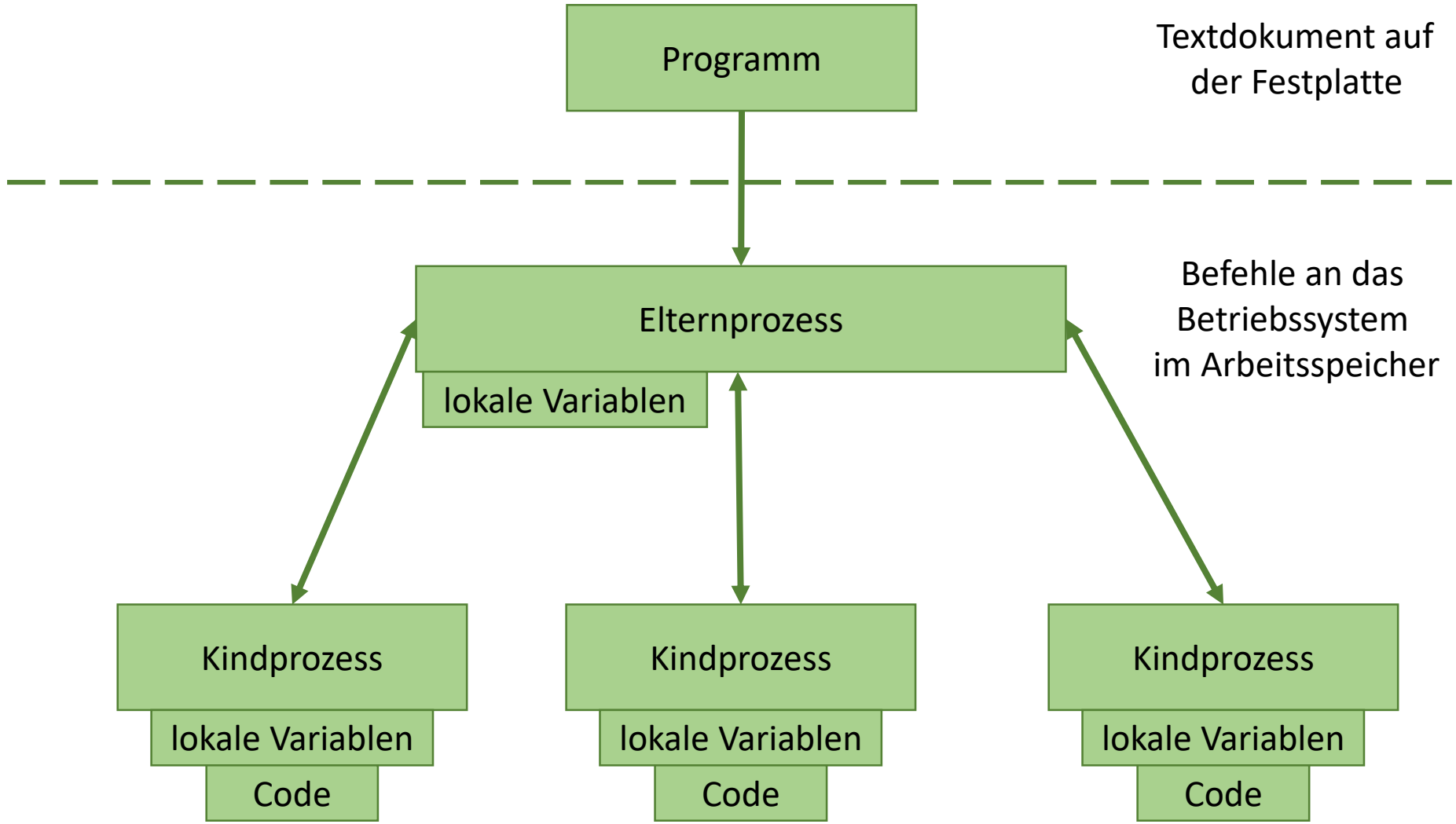
- Da Threads Teilprozesse sind, sind sie immer abhängig von dem dazugehörigem Hauptprozess
  → Endet dieser, enden alle Threads *oder* der Hauptprozess kann erst enden wenn alle seine Threads beendet wurden

- Threads werden nicht (immer) gezielt auf mehrere vorhandene Prozessoren verteilt, damit ist echtes paralleles arbeiten nicht möglich

- Echte Prozesse werden automatisch vom Betriebssystem auf alle vorhandenen Prozessoren gleichmäßig verteilt

- Kinderprozesse teilen sich keinen gemeinsamen globalen Namensraum mit ihrem Elternprozess und *können* nach seiner Beendigung weiterlaufen

Ein Prozess


Programm
Textdokument auf
der Festplatte
Elternprozess
lokale Variablen
Befehle an das
Betriebssystem
im Arbeitsspeicher
Kindprozess
lokale Variablen
Code
Kindprozess
lokale Variablen
Code
Kindprozess
lokale Variablen
Code

Multiprocessing

- Das Modul *multiprocessing* ist fast exakt wie *threading* aufgebaut, aber statt Threads erzeugt es echte Prozesse

- Die meisten Funktionen und Prinzipien der *threading.Thread*-Objekte gelten auch für die *multiprocessing.Process*-Objekte

```
import time, multiprocessing

def schlafen(i):
    print('Prozess {} schläft für 5 Sekunden'.format(i))
    time.sleep(5)
    print('Prozess {} ist aufgewacht'.format(i))

for i in range(10):
    t = multiprocessing.Process(target=schlafen, args=(i,))
    t.start()
```

Multiprocessing

- Wie in *threading* lassen sich Lock-, Conditions- und Timer-Objekte erstellen, um Prozesse miteinander zu synchronisieren

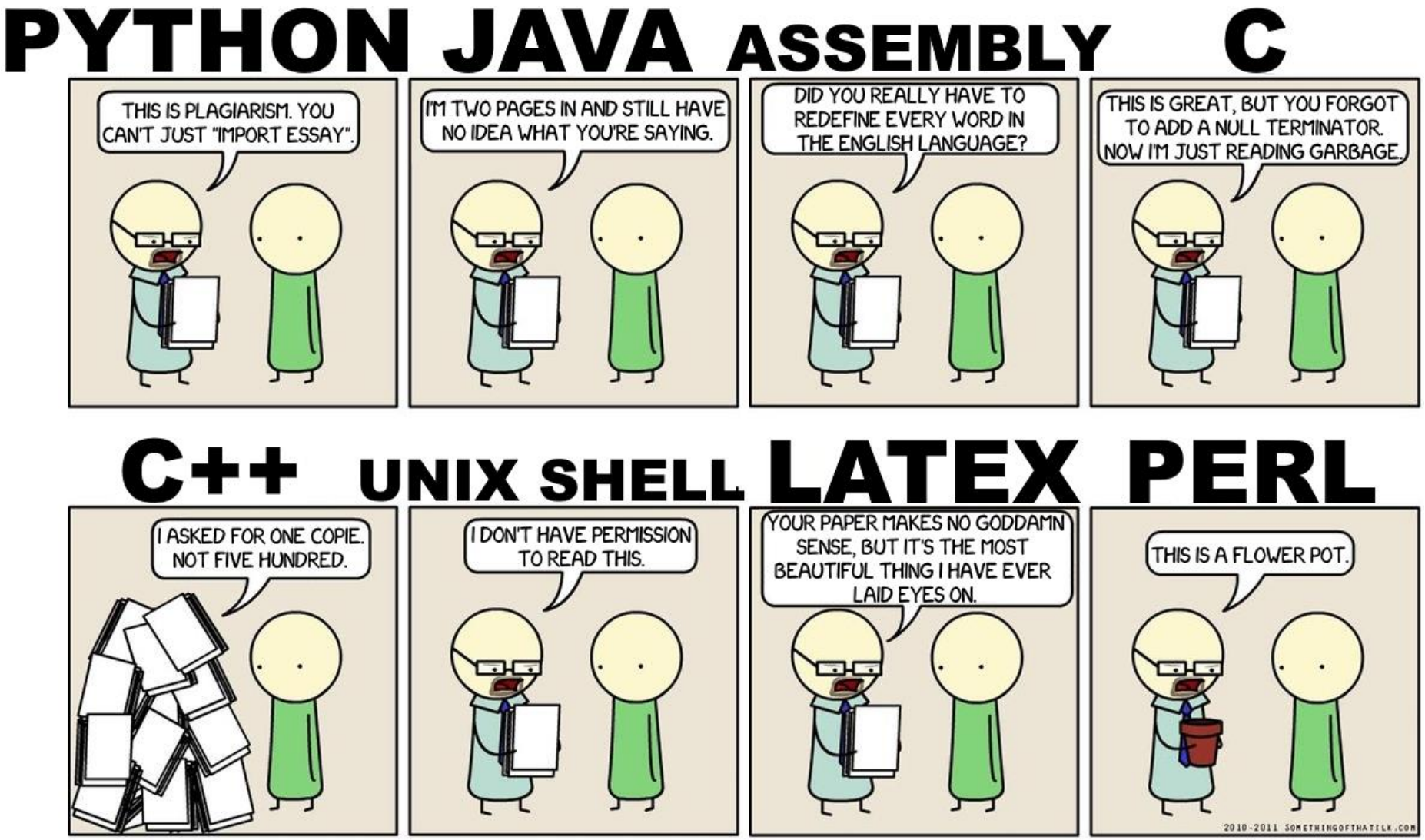
- Mit dem Attribut *daemon* lassen sich Prozesse in Dämon-Prozesse umwandeln, d. h. sie laufen unabhängig von ihrem Elternprozess

```
t = multiprocessing.Process(target=function, args=(x,))
t.Daemon = True
```

- Auch Prozess-Objekte kennen die *join()*-Methode um auf ihre gemeinsame Beendigung zu warten

Multiprocessing

- Mit der *terminate()*-Methode lassen sich Prozesse gezielt beenden
- Wie Threads lassen sich auch Prozesse in Queues zusammenfassen, um die Anzahl an gleichzeitig laufenden Kindprozessen zu limitieren
- Eigene Klassen können von *multiprocessing.Process* erben und verfügen damit über eine *run()*- und eine *start()*-Funktion
- Es ist möglich einen künstlichen globalen Namensraum anzulegen in dem sich Kindprozesse bestimmte Variablen aus dem Elternprozess teilen können


PYTHON
THIS IS PLAGIARISM. YOU CAN'T JUST "IMPORT ESSAY".
JAVA
I'M TWO PAGES IN AND STILL HAVE NO IDEA WHAT YOU'RE SAYING.
ASSEMBLY
DID YOU REALLY HAVE TO REDEFINE EVERY WORD IN THE ENGLISH LANGUAGE?
C
THIS IS GREAT, BUT YOU FORGOT TO ADD A NULL TERMINATOR. NOW I'M JUST READING GARBAGE.
C++
I ASKED FOR ONE COPIE. NOT FIVE HUNDRED.
UNIX SHELL
I DON'T HAVE PERMISSION TO READ THIS.
LATEX
YOUR PAPER MAKES NO GODDAMN SENSE, BUT IT'S THE MOST BEAUTIFUL THING I HAVE EVER LAID EYES ON.
PERL
THIS IS A FLOWER POT.
2010-2011 SOMETHINGOFTHATILK.COM