PPaDS MMXXI

Matúš Jókay, Château de la Mûre
matus.jokay@stuba.sk

uim.fei.stuba.sk/predmet/i-ppds

konzultácie elektronicky

mail, discord

Francesco Pierfederici

Distributed Computing with Python

Packt Publishing Ltd.

April 2016

ISBN 978-1-78588-969-1

PPaDS – Definície

- Paralelný výpočet

  Paralelný výpočet je súčasné využitie viacerých (t.j. viac než 1) výpočtových uzlov na dosiahnutie cieľa výpočtu

- Distribuovaný výpočet

  Distribuovaný výpočet je súčasné využitie viacerých (t.j. viac než 1) výpočtových uzlov na dosiahnutie cieľa výpočtu

PPaDS - Definície

- **Paralelný** výpočet (výpočtový uzol = **CPU**)

  Paralelný výpočet je súčasné využitie viacerých (t.j. viac než 1) výpočtových uzlov na dosiahnutie cieľa výpočtu

- **Distribuovaný** výpočet (výpočtový uzol = **PC**)

  Distribuovaný výpočet je súčasné využitie viacerých (t.j. viac než 1) výpočtových uzlov na dosiahnutie cieľa výpočtu

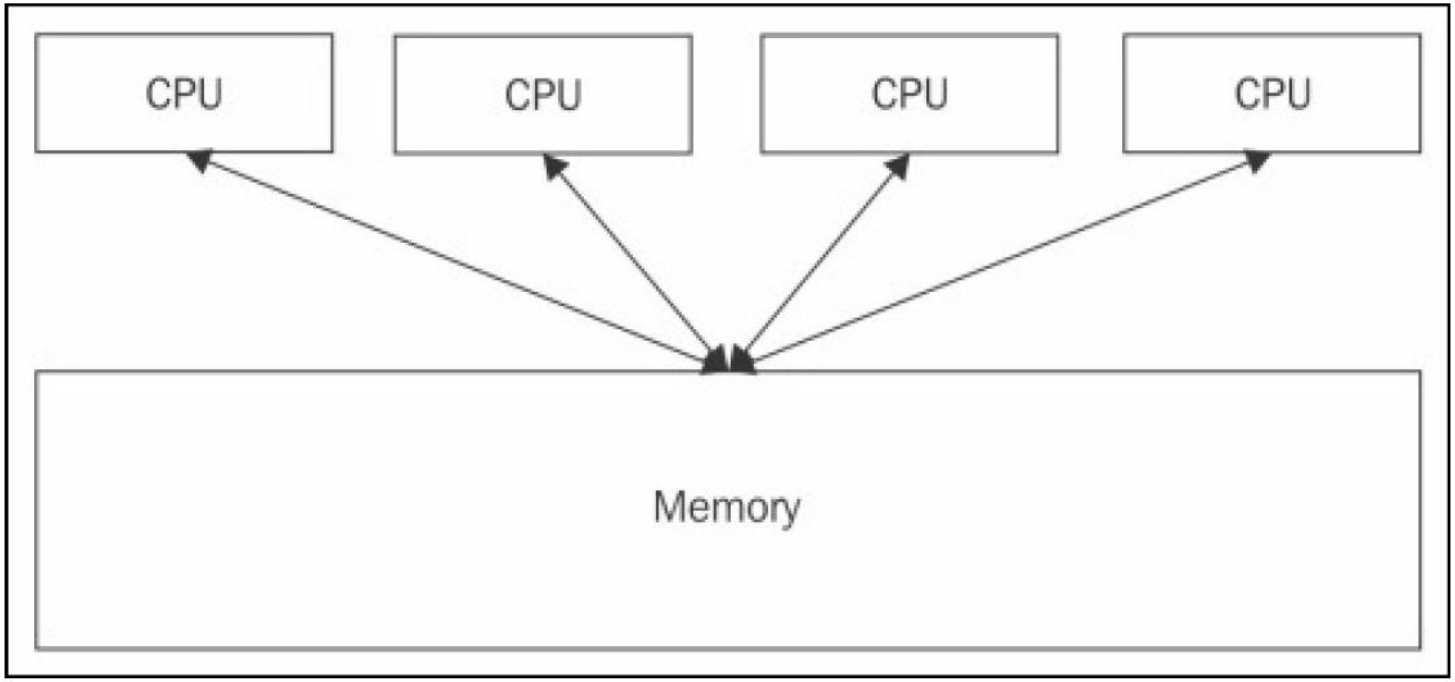
Vývoj distribuovanej aplikácie

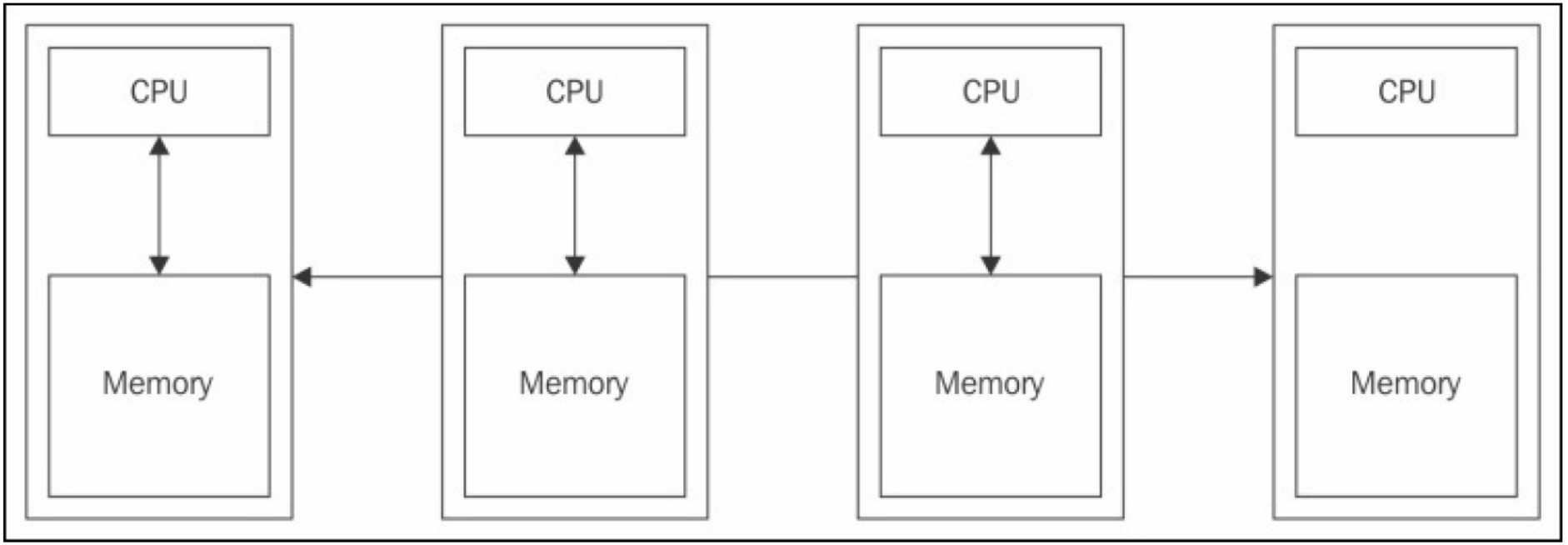
1. Vývoj jednovláknovej (jednoprocesovej) aplikácie
2. Vývoj multiprocesovej aplikácie (nie multivláknovej, hoci v závislosti od implementačného prostredia to môže byť jeden z medzikrokov)
3. Vývoj distribuovanej aplikácie

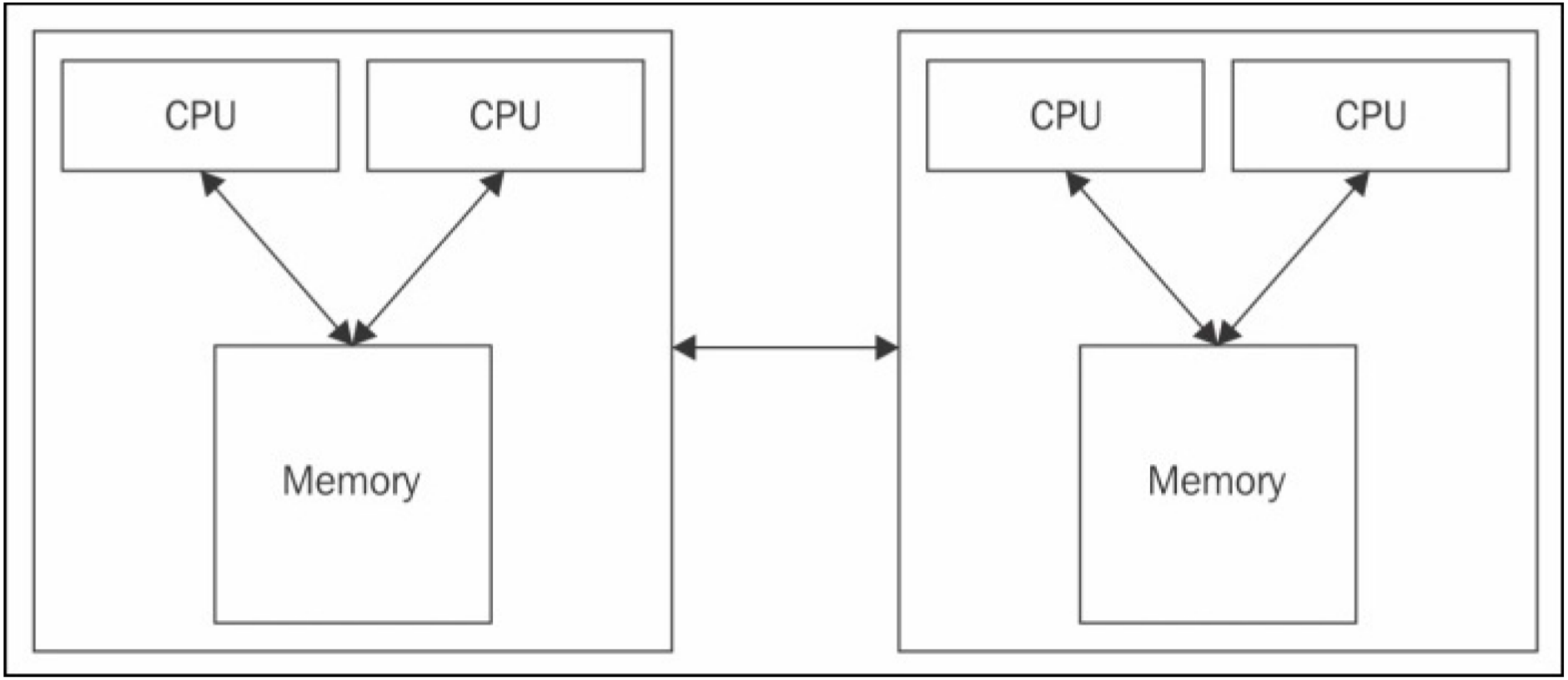
PPaDS – pozor na údaje!

- Skutočným úzkym hrdlom výpočtov zväčša (závisí od typu aplikácie) bývajú samotné údaje, nie CPU

- Niekedy stačí zdieľaný súborový systém (napr. NFS na unixových systémoch), inokedy zdieľaná databáza alebo posielanie správ…


CPU
CPU
CPU
CPU
Memory


CPU
CPU
CPU
CPU
Memory
Memory
Memory
Memory



Hybridná architektúra pamäte

Asynchrónne programovanie

Asynchrónne programovanie

- Pre naše účely budeme program rozdeľovať na úlohy (tasky), z ktorých každá vykonáva nejakú operáciu (napr. delenie, získanie údajov z db, vypisovanie na monitor atď…)

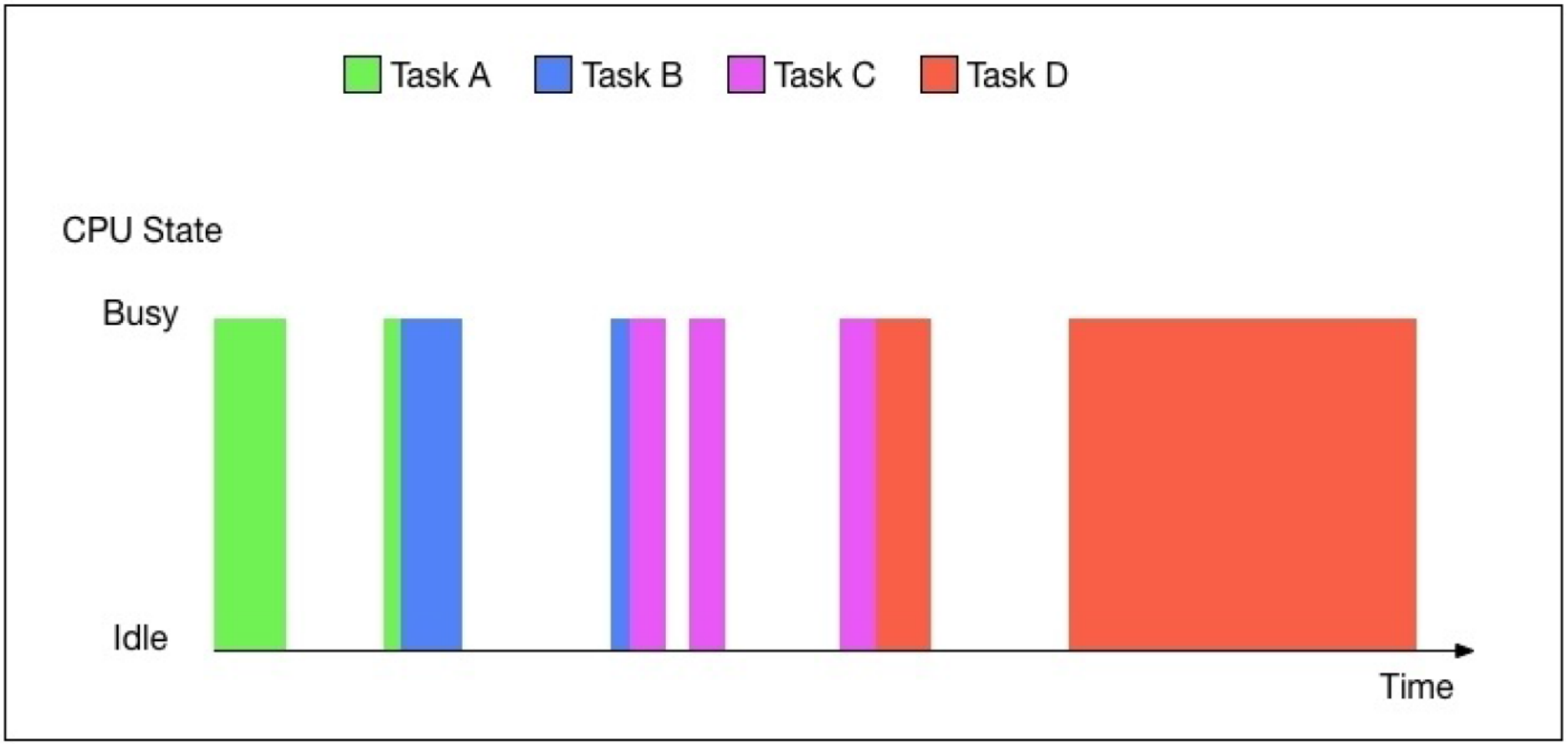
Asynchrónne programovanie

- Pre naše účely budeme program rozdeľovať na úlohy (tasky), z ktorých každá vykonáva nejakú operáciu (napr. delenie, získanie údajov z db, vypisovanie na monitor atď…)
- Pre jednoduchosť budeme považovať funkcie programu za takéto úlohy (tasky)

Asynchrónne programovanie

- Príklad: majme program, ktorý vykonáva 4 úlohy – A, B, C a D
- Každá z úloh vykonáva nejaký výpočet a pristupuje k I/O zariadeniam

- Nasledujúci graf znázorňuje beh týchto 4 úloh na jednom jadre CPU



- Počas I/O úlohy CPU je v stave **nečinnosti**!!!
- Rôzne komponenty majú rôznu rýchlosť prístupu aj prenosovú kapacitu (disk, sieť...)
- Počas tejto doby je CPU nečinná a nevyužitá

Asynchrónne programovanie

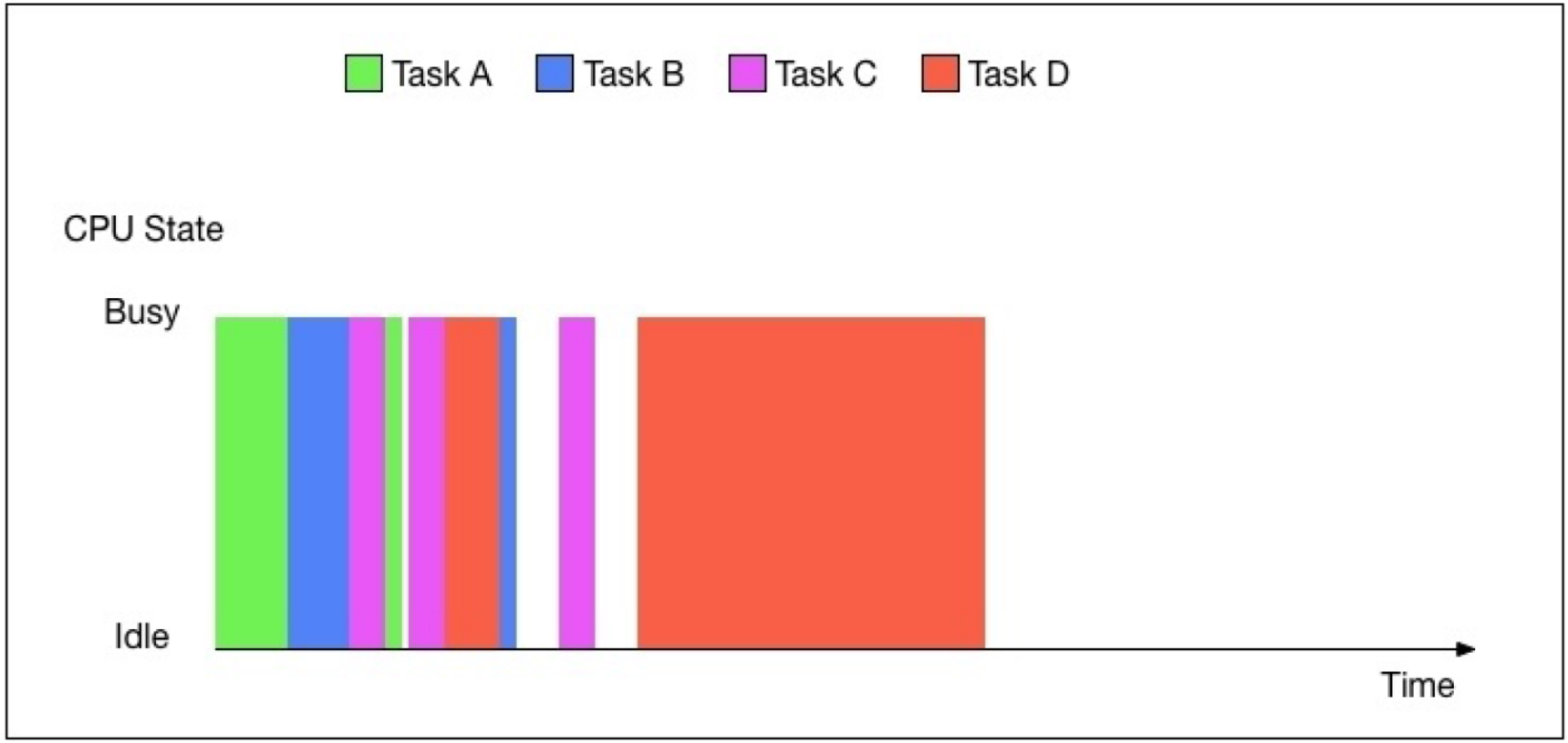
- Ideálne by bolo také usporiadanie úloh, aby v čase nečinnosti (čakania na I/O) jednej úlohy mohla využívať CPU iná úloha

Asynchrónne programovanie

- Ideálne by bolo také usporiadanie úloh, aby v čase nečinnosti (čakania na I/O) jednej úlohy mohla využívať CPU iná úloha
- A práve o toto ide v prípade asynchrónneho (alebo inak aj udalosťami riadeného (event-driven)) programovania

Asynchrónne programovanie

- Ideálne by bolo také usporiadanie úloh, aby v čase nečinnosti (čakania na I/O) jednej úlohy mohla využívať CPU iná úloha
- A práve o toto ide v prípade asynchrónneho (alebo inak aj udalosťami riadeného (event-driven)) programovania
- Predošlý príklad so štyrmi úlohami, ale pomocou asynchrónneho prístupu, vidno na nasledujúcom obrázku



- Aj v tomto prípade sa úlohy spúšťajú sekvenčne
- Ale neblokujú počas I/O čakania CPU

Asynchrónne programovanie

- Asynchrónne programovanie != multivláknové

Asynchrónne programovanie

- Asynchrónne programovanie != multivláknové
  - Pri multivláknovom OS rozhoduje (plánovač), ktoré vlákno bude čo vykonávať, a ovšem podľa požadovaných I/O automaticky dokáže odložiť vykonávanie vlákien, ktoré čakajú na dokončenie I/O

Asynchrónne programovanie

- Asynchrónne programovanie != multivláknové
  - Pri multivláknovom OS rozhoduje (plánovač), ktoré vlákno bude čo vykonávať, a ovšem podľa požadovaných I/O automaticky dokáže odložiť vykonávanie vlákien, ktoré čakajú na dokončenie I/O
  - Pri asynchrónnom programovaní sa úloha samotná rozhoduje (v zastúpení programátora, ovšem), že sa vzdá CPU, pokým nebude dokončená želaná I/O operácia

Asynchrónne programovanie

- Asynchrónne programovanie != multivláknove
  - Pri multivláknovom OS sa úlohy môžu vykonávať paralelne
  - Pri asynchrónnom programovaní sa úlohy stále vykonávajú sekvenčne! Ide však o efektívnejšie využitie CPU

Koprogram (coroutine)

- Podľa wikipédie sú koprogramy všeobecnejšie než podprogramy (procedúry, funkcie, metódy), nakoľko majú viac vstupných bodov, pozastavení a obnovení výpočtu v rôznych miestach svojej definície.

Koprogram (coroutine)

- Životný cyklus podprogramov je riadený zásobníkom (posledne zavolaný podprogam urobí návrat ako prvý)

- Životný cyklus koprogamu závisí výhradne na použití

Koprogram (coroutine)

- Podprogram má iba jeden vstupný bod (začiatok, pri jeho zavolaní), a je možné iba raz sa z neho vrátiť
- Koprogramy sa môžu vracať viackrát (v Pythone príkazom *yield*)

Koprogram (coroutine)

- Podprogram má iba jeden vstupný bod (začiatok, pri jeho zavolaní), a je možné iba raz sa z neho vrátiť
- Koprogramy sa môžu vracať viackrát (v Pythone príkazom *yield*)
  - Začiatkom koprogramu je vstupný bod pri jeho volaní
  - Ďalšie vstupné body sú dané príkazom *yield*

Koprogram (coroutine)

- Ako to funguje v praxi

Koprogram (coroutine)

- Ako to funguje v praxi
- Pri prvom zavolaní koprogramu sa začne vykonávať od svojho začiatku podobne ako podprogram

Koprogram (coroutine)

- Ako to funguje v praxi
- Pri prvom zavolaní koprogramu sa začne vykonávať od svojho začiatku podobne ako podprogram
- Akonáhle narazí na príkaz *yield*, vráti výsledok a odovzdá riadenie do volajúceho (pod/ko) programu podobne, ako príkaz *return* pri klasickom podprograme

Koprogram (coroutine)

- Avšak pri ďalšom volaní toho istého koprogramu (v pripade Pythonu pomocou metódy *next*) sa koprogram nezačne vykonávať od začiatku, ale od príkazu, ktorý sa nachádza bezprostredne za posledným vykonaným príkazom *yield*

Koprogram (coroutine)

- Je vhodný skôr k obnoveniu vykonávania, nie k reštartovaniu (od začiatku tela svojej definície)
- Je schopný udržiavať medzi jednotlivými volaniami
  - Stav
  - Premenné (podobne ako *uzávery* v Pythone)
  - Bod aktuálneho vykonávania
  - Výsledok nemusí byť vrátený na konci vetvy kódu

Koprogram (coroutine)

- Na pochopenie *koprogramov* v Pythone je nutné chápať *generátory*, a k nim potrebujeme vedomosti o *iterátoroch* ;)

Koprogram (coroutine)

- Na pochopenie *koprogramov* v Pythone je nutné chápať *generátory*, a k nim potrebujeme vedomosti o *iterátoroch* ;)
- Generátory sú tiež zovšeobecnením podprogramov, ale sú obmedzenejšie než koprogramy

Koprogram (coroutine)

- Na pochopenie *koprogramov* v Pythone je nutné chápať *generátory*, a k nim potrebujeme vedomosti o *iterátoroch* ;)
- Generátory sú tiež zovšeobecnením podprogramov, ale sú obmedzenejšie než koprogramy
  - Podobne ako koprogramy umožňujú viac návratov

Koprogram (coroutine)

- Na pochopenie *koprogramov* v Pythone je nutné chápať *generátory*, a k nim potrebujeme vedomosti o *iterátoroch* ;)
- Generátory sú tiež zovšeobecnením podprogramov, ale sú obmedzenejšie než koprogramy
  - Podobne ako koprogramy umožňujú viac návratov
  - Nemôžu však kontrolovať miesto, v ktorom bude pokračovať vykonávanie kódu po zavolaní *yield*

Koprogram (coroutine)

- Na pochopenie *koprogramov* v Pythone je nutné chápať *generátory*, a k nim potrebujeme vedomosti o *iterátoroch* ;)
- Generátory sú tiež zovšeobecnením podprogramov, ale sú obmedzenejšie než koprogramy
  - Podobne ako koprogramy umožňujú viac návratov
  - Nemôžu však kontrolovať miesto, v ktorom bude pokračovať vykonávanie kódu po zavolaní *yield*
  - Vracajú kontrolu vždy na miesto, z ktorého boli zavolané

1 Iterovanie v Pythone

- Úfajme, že sme dostatočne oboznámení s *iterovaním* rôznych objektov v Pythone (reťazce, zoznamy, n-tice, súbory…)

1 Iterovanie v Pythone

- Úfajme, že sme dostatočne oboznámení s *iterovaním* rôznych objektov v Pythone (reťazce, zoznamy, n-tice, súbory…)

```
>>> for i in range(3):
...     print(i)
...
0
1
2
>>> for line in open('exchange_rates_v1.py'):
...     print(line, end='')
...
#!/usr/bin/env python3
import itertools
import time
import urllib.request
…
```

1 Iterovanie v Pythone

- Prečo je možné iterovať nielen reťazce a zoznamy, ale aj komplexnejšie objekty?

1 Iterovanie v Pythone

- Prečo je možné iterovať nielen reťazce a zoznamy, ale aj komplexnejšie objekty?
- Dôvodom je rozhranie Pythonu, tzv. iteračný protokol (*iteration protocol*)

1 Iterovanie v Pythone

- Prečo je možné iterovať nielen reťazce a zoznamy, ale aj komplexnejšie objekty?
- Dôvodom je rozhranie Pythonu, tzv. iteračný protokol (*iteration protocol*)
- Objekt, ktorý implementuje metódy __*iter*__ a __*next*__, sa stáva iterátorom

1 Iterovanie v Pythone

- Prečo je možné iterovať nielen reťazce a zoznamy, ale aj komplexnejšie objekty?
- Dôvodom je rozhranie Pythonu, tzv. iteračný protokol (*iteration protocol*)
- Objekt, ktorý implementuje metódy __*iter*__ a __*next*__, sa stáva iterátorom
  - Prvá z metód (__iter__) vracia objekt, ktorý je možné pomocou metódy __next__ iterovať

1 Iterovanie v Pythone

- Prečo je možné iterovať nielen reťazce a zoznamy, ale aj komplexnejšie objekty?
- Dôvodom je rozhranie Pythonu, tzv. iteračný protokol (*iteration protocol*)
- Objekt, ktorý implementuje metódy __*iter*__ a __*next*__, sa stáva iterátorom
  - Prvá z metód (__iter__) vracia objekt, ktorý je možné pomocou metódy __next__ iterovať
  - Druhá metóda (__next__) vracia pri sekvenčnom volaní generované prvky pekne jeden za druhým

1 Iterovanie v Pythone

```
class MyIterator(object):
    def __init__(self, xs):
        self.xs = xs

    def __iter__(self):
        return self

    def __next__(self):
        if self.xs:
            return self.xs.pop(0)
        else:
            raise StopIteration

for i in MyIterator([0, 1, 2]):
    print(i)
```

1 Iterovanie v Pythone

- Predošlý kód vypíše na výstup

```
0
1
2
```

1 Iterovanie v Pythone

- Predošlý kód vypíše na výstup

```
0
1
2
```

- Pre lepšie pochopenie protokolu môžeme “manuálne” prelistovať objekt, ktorý implementuje protokol iterovania

1 Iterovanie v Pythone

- Predošlý kód vypíše na výstup

```
0
1
2
```

- Pre lepšie pochopenie protokolu môžeme “manuálne” prelistovať objekt, ktorý implementuje protokol iterovania

```
itrtr = MyIterator([3, 4, 5, 6])

print(next(itrtr))
print(next(itrtr))
print(next(itrtr))
print(next(itrtr))

print(next(itrtr))
```

1 Iterovanie v Pythone

- A výstup:

```
3
4
5
6
Traceback (most recent call last):
  File "iteration.py", line 32, in <module>
    print(next(itrtr))
  File "iteration.py", line 19, in __next__
  raise StopIteration
StopIteration
```

1 Iterovanie v Pythone

- Najprv vytvoríme inštanciu triedy *MyIterator*, a potom pomocou volania *next*() postupne získavame hodnoty postupnosti, ktorú je tento objekt schopný generovať

1 Iterovanie v Pythone

- Najprv vytvoríme inštanciu triedy *MyIterator*, a potom pomocou volania *next*() postupne získavame hodnoty postupnosti, ktorú je tento objekt schopný generovať
- Po vyčerpaní prvkov sa generuje výnimka *StopIteration*

1 Iterovanie v Pythone

- Najprv vytvoríme inštanciu triedy *MyIterator*, a potom pomocou volania *next*() postupne získavame hodnoty postupnosti, ktorú je tento objekt schopný generovať
- Po vyčerpaní prvkov sa generuje výnimka *StopIteration*
- Cyklus *for* v Pythone využíva presne tento mechanizmus: volá *next*() na iterátore, ktorý na začiatku získa pomocou volania *iter*(), a zachytáva výnimku *StopIteration*, aby vedel, kedy je koniec iterovania

2 Generátor v Pythone

- Je volateľný (angl. *callable*) objekt (funkcia), ktorý vracia inštanciu generátora (iterator); ten miesto okamžitého vrátenia celého výsledku postupne generuje čiastkové výsledky

2 Generátor v Pythone

- Je volateľný (angl. *callable*) objekt (funkcia), ktorý vracia inštanciu generátora (iterator); ten miesto okamžitého vrátenia celého výsledku postupne generuje čiastkové výsledky
- Na generovanie čiastkového výsledku sa používa kľúčové slovo *yield*

2 Generátor v Pythone

- Je volateľný (angl. *callable*) objekt (funkcia), ktorý vracia inštanciu generátora (iterator); ten miesto okamžitého vrátenia celého výsledku postupne generuje čiastkové výsledky
- Na generovanie čiastkového výsledku sa používa kľúčové slovo *yield*

```
def mygenerator(n):
    while n:
        n -= 1
        yield n
```

2 Generátor v Pythone

```
def mygenerator(n):
    while n:
        n -= 1
        yield n

if __name__ == '__main__':
    for i in mygenerator(3):
        print(i)
```

2 Generátor v Pythone

```
def mygenerator(n):
    while n:
        n -= 1
        yield n

if __name__ == '__main__':
    for i in mygenerator(3):
        print(i)
```

Výstup:

```
2
1
0
```

2 Generátor v Pythone

- V čom je rozdiel medzi obyčajnou funkciou (metódou) a generátorom?

2 Generátor v Pythone

- V čom je rozdiel medzi obyčajnou funkciou (metódou) a generátorom?
- V použití kľúčového slova *yield*

2 Generátor v Pythone

- Zavolanie (funkcie) generátora negeneruje žiaden objekt postupnosti, ale tzv. objekt generátora (generátorový objekt, ktorý je iterátor)

2 Generátor v Pythone

- Zavolanie (funkcie) generátora negeneruje žiaden objekt postupnosti, ale tzv. objekt generátora (generátorový objekt, ktorý je iterátor)

```
>>> from generators import mygenerator
>>> mygenerator(5)
<generator object mygenerator at 0x101267b48>
```

2 Generátor v Pythone

- Na aktivovanie generátorového iterátora je potrebné na ňom vyvolať metódu *next*()

2 Generátor v Pythone

- Na aktivovanie generátorového iterátora je potrebné na ňom vyvolať metódu *next*()

```
>>> g = mygenerator(2)
>>> next(g)
1
>>> next(g)
0
>>> next(g)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

2 Generátor v Pythone

- Každé volanie *next*() produkuje práve jednu hodnotu z postupnosti generovanej generátorovým iterátorom

2 Generátor v Pythone

- Každé volanie *next*() produkuje práve jednu hodnotu z postupnosti generovanej generátorovým iterátorom
- Po vyprázdnení množiny prvkov, z ktorých sa generuje postupnosť, sa vyvolá výnimka *StopIteration*

2 Generátor v Pythone

- Každé volanie *next*() produkuje práve jednu hodnotu z postupnosti generovanej generátorovým iterátorom
- Po vyprázdnení množiny prvkov, z ktorých sa generuje postupnosť, sa vyvolá výnimka *StopIteration*
- Generátorové **funkcie** sú v podstate zjednodušeným zápisom iterátorov – odpadá nutnosť definovať **triedu** s metódami __*iter*__ a __*next*__

2 Generátor v Pythone

- Po vyprázdnení objektu generátora (keď už generuje výnimku *StopIteration*) ho už nevieme použiť na opätovné generovanie postupnosti

2 Generátor v Pythone

- Po vyprázdnení objektu generátora (keď už generuje výnimku *StopIteration*) ho už nevieme použiť na opätovné generovanie postupnosti

- Ak chceme znovu získať postupnosť generovanú generátorovým objektom, musíme získať nový objekt generátora a iterovať ho

3 Koprogramy v Pythone

3 Koprogramy v Pythone

- Výraz *yield* môžeme použiť aj na pravej strane priradenia, na znak toho, že chceme prvky nie generovať, ale spracovávať (konzumovať)

3 Koprogramy v Pythone

- Výraz *yield* môžeme použiť aj na pravej strane priradenia, na znak toho, že chceme prvky nie generovať, ale spracovávať (konzumovať)
- Ak takéto niečo použijeme vo funkcii/metóde Pythonu, vytvoríme tzv. koprogram

3 Koprogramy v Pythone

- Výraz *yield* môžeme použiť aj na pravej strane priradenia, na znak toho, že chceme prvky nie generovať, ale spracovávať (konzumovať)
- Ak takéto niečo použijeme vo funkcii/metóde Pythonu, vytvoríme tzv. koprogram
- Koprogram v Pythone je, **VEĽMI zjednodušene** napísané, typ funkcie, ktorá môže prerušiť a znovu obnoviť svoje vykonávanie na miestach s výskytom kľúčového slova *yield* na pravej strane priradenia

3 Koprogramy v Pythone

- Koprogramy v Pythone nie sú jednoduchými generátormi! (aj keď sa v nich používa to isté kľúčové slovo *yield*)

- Dôvody:
  - Koprogramy nie sú spojené s iteráciou
  - Generátorové objekty hodnoty produkujú, koprogramy ich konzumujú

3 Koprogramy v Pythone

- Koprogramy využívajú na svoju činnosť nasledovné 3 mechanizmy

3 Koprogramy v Pythone

- Koprogramy využívajú na svoju činnosť nasledovné 3 mechanizmy
  1. *yield*()
  2. *send*()
  3. *close*()

3 Koprogramy v Pythone

- Koprogramy využívajú na svoju činnosť nasledovné 3 mechanizmy
  1. *yield*() – uspí vykonávanie koprogramu do najbližšieho vyvolania metódy *send*() nad objektom koprogramu
  2. *send*()
  3. *close*()

3 Koprogramy v Pythone

- Koprogramy využívajú na svoju činnosť nasledovné 3 mechanizmy
  1. *yield*() – uspí vykonávanie koprogramu do najbližšieho vyvolania metódy *send*() nad objektom koprogramu
  2. *send*() – slúži na poslanie údajov na spracovanie koprogramu (teda zároveň „prebudí“ koprogram)
  3. *close*()

3 Koprogramy v Pythone

- Koprogramy využívajú na svoju činnosť nasledovné 3 mechanizmy
  1. *yield*() – uspí vykonávanie koprogramu do najbližšieho vyvolania metódy *send*() nad objektom koprogramu
  2. *send*() – slúži na poslanie údajov na spracovanie koprogramu (teda zároveň „prebudí“ koprogram)
  3. *close*() – ukončí koprogram (pošle mu výnimku *GeneratorExit*)

```
def complain_about(substring):
    print('Please talk to me!')
    try:
        while True:
            text = (yield)
            if substring in text:
                print('Oh no: I found a %s again!'
                      % (substring))
    except GeneratorExit:
        print('Ok, ok: I am quitting.')
```

```
def complain_about(substring):
    print('Please talk to me!')
    try:
        while True:
            text = (yield)
            if substring in text:
                print('Oh no: I found a %s again!'
                      % (substring))
    except GeneratorExit:
        print('Ok, ok: I am quitting.')
```

- Nami definovaný koprogram spracúva jeden argument (reťazec)

```
def complain_about(substring):
    print('Please talk to me!')
    try:
        while True:
            text = (yield)
            if substring in text:
                print('Oh no: I found a %s again!'
                      % (substring))
    except GeneratorExit:
        print('Ok, ok: I am quitting.')
```

- Nami definovaný koprogram spracúva jeden argument (reťazec)
- Po vypísaní úvodnej správy začne vykonávať nekonečný cyklus vnorený do *try-except* bloku

```
def complain_about(substring):
    print('Please talk to me!')
    try:
        while True:
            text = (yield)
            if substring in text:
                print('Oh no: I found a %s again!'
                      % (substring))
    except GeneratorExit:
        print('Ok, ok: I am quitting.')
```

- Nami definovaný koprogram spracúva jeden argument (reťazec)
- Po vypísaní úvodnej správy začne vykonávať nekonečný cyklus vnorený do *try-except* bloku
- Po zachytení výnimky *GeneratorExit* činnosť koprogramu (po výpise správy) končí

```
def complain_about(substring):
    print('Please talk to me!')
    try:
        while True:
            text = (yield)
            if substring in text:
                print('Oh no: I found a %s again!'
                      % (substring))
    except GeneratorExit:
        print('Ok, ok: I am quitting.')
```

- Nami definovaný koprogram spracúva jeden argument (reťazec)
- Po vypísaní úvodnej správy začne vykonávať nekonečný cyklus vnorený do *try-except* bloku
- Po zachytení výnimky *GeneratorExit* činnosť koprogramu (po výpise správy) končí
- Telo cyklu je jednoduché – pomocou *yield* získame údaje, ktoré uložíme do premennej *text* a následne ich spracujeme

3 Spustenie koprogramu

```
>>> from coroutines import complain_about
>>> c = complain_about('Ruby')
>>> next(c)
Please talk to me!
>>> c.send('Test data')
>>> c.send('Some more random text')
>>> c.send('Test data with Ruby somewhere in it')
Oh no: I found a Ruby again!
>>> c.send('Stop complaining about Ruby or else!')
Oh no: I found a Ruby again!
>>> c.close()
Ok, ok: I am quitting.
```

3 Spustenie koprogramu

- Zavolaním *complain_about('Ruby')* sa vytvorí objekt koprogramu (nič iné sa neudeje!)

3 Spustenie koprogramu

- Zavolaním *complain_about('Ruby')* sa vytvorí objekt koprogramu (nič iné sa neudeje!)
- Až následným zavolaním metódy *next*() nad týmto objektom sa spustí vykonávanie koprogramu (vidíme to na základe výpisu úvodnej správy)

3 Spustenie koprogramu

- Zavolaním *complain_about('Ruby')* sa vytvorí objekt koprogramu (nič iné sa neudeje!)
- Až následným zavolaním metódy *next*() nad týmto objektom sa spustí vykonávanie koprogramu (vidíme to na základe výpisu úvodnej správy)
- V cykle vykonávanie koprogramu dosiahne riadok *text = (yield)*, ktorý preruší vykonávanie funkcie

3 Spustenie koprogramu

- Zavolaním *complain_about('Ruby')* sa vytvorí objekt koprogramu (nič iné sa neudeje!)
- Až následným zavolaním metódy *next*() nad týmto objektom sa spustí vykonávanie koprogramu (vidíme to na základe výpisu úvodnej správy)
- V cykle vykonávanie koprogramu dosiahne riadok *text = (yield)*, ktorý preruší vykonávanie funkcie
- Koprogram obnoví svoju činnosť, akonáhle dostane údaje na spracovanie pomocou metódy *send*()

3 Spustenie koprogramu

- Každé zavolanie metódy *send*() posunie vykonávanie kódu v koprograme na ďalšie *yield*

3 Spustenie koprogramu

- Každé zavolanie metódy *send*() posunie vykonávanie kódu v koprograme na ďalšie *yield*
- Koprogram je možné ukončiť zavolaním metódy *close*(), ktorá generuje výnimku *GeneratorExit*

3 Spustenie koprogramu

- Každé zavolanie metódy *send*() posunie vykonávanie kódu v koprograme na ďalšie *yield*
- Koprogram je možné ukončiť zavolaním metódy *close*(), ktorá generuje výnimku *GeneratorExit*
- Ak by sme v ukážkovom kóde vynechali blok *try-catch*, nespracuje sa výnimka *GeneratorExit*; koprogram iba skončí svoju činnosť

3 Spustenie koprogramu

- Každé zavolanie metódy *send*() posunie vykonávanie kódu v koprograme na ďalšie *yield*
- Koprogram je možné ukončiť zavolaním metódy *close*(), ktorá generuje výnimku *GeneratorExit*
- Ak by sme v ukážkovom kóde vynechali blok *try-catch*, nespracuje sa výnimka *GeneratorExit*; koprogram iba skončí svoju činnosť (**skúste v rámci cvičenia**)

3 Spustenie koprogramu

- Po ukončení činnosti koprogramu pomocou *close*() objekt koprogramu síce ostáva jestvovať, ale nemožno mu ani poslať údaje pomocou *send*(), ani obnoviť jeho vykonávanie pomocou metódy *next*()

3 Spustenie koprogramu

- Po ukončení činnosti koprogramu pomocou *close*() objekt koprogramu síce ostáva jestvovať, ale nemožno mu ani poslať údaje pomocou *send*(), ani obnoviť jeho vykonávanie pomocou metódy *next*()

```
>>> c.close()
>>> c.send('This will crash')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>> next(c)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

3 Spustenie koprogramu

- Pri použití koprogramu je vždy nutné (po vytvorení objektu koprogramu) použiť metódu *next*() na jeho aktivovanie

3 Spustenie koprogramu

- **Pri použití koprogramu je vždy nutné (po vytvorení objektu koprogramu) použiť metódu *next*() na jeho aktivovanie**

3 Spustenie koprogramu

- Pri použití koprogramu je vždy nutné (po vytvorení objektu koprogramu) použiť metódu *next*() na jeho aktivovanie
- Pri častom používaní koprogramov to môže byť dosť otravné, preto je možné v Pythone využiť **dekorátory** na obídenie nutnosti vyvolávania *next*() po vytvorení objektu koprogramu (aby sa predišlo napríklad vynechaniu tohto volania zo zábudlivosti)

```
>>> def coroutine(fn):
...     def wrapper(*args, **kwargs):
...         c = fn(*args, **kwargs)
...         next(c)
...         return c
...     return wrapper
...
>>> @coroutine
... def complain_about2(substring):
...     print('Please talk to me!')
...     while True:
...         text = (yield)
...         if substring in text:
...             print('Oh no: I found a %s again!'
...                   % (substring))
...
>>> c = complain_about2('JavaScript')
Please talk to me!
>>> c.send('Test data with JavaScript somewhere in it')
Oh no: I found a JavaScript again!
>>> c.close()
```

4 Záver

4 Záver

- Výhody asynchrónneho programovania pomocou rozšírených generátorov v Pythone:
  1. Šetrenie pamäťou
  2. Vyťaženie CPU pri I/O operáciách

4 Záver

- Výhody asynchrónneho programovania pomocou rozšírených generátorov v Pythone:
  1. Šetrenie pamäťou
  2. Vyťaženie CPU pri I/O operáciách

- Nevýhody:
  1. Vyššia zložitosť kódu
  2. Nutnosť vlastného plánovača behu koprogramov

4 Záver

- Na čo si dať pozor pri písaní koprogramov

- Vhodné iba pre aplikácie, kde sa veľa času trávi v I/O operáciách
- Treba sa vyhýbať funkciám, ktoré sú blokujúce (tým sa stráca zmysel asynchrónneho kódu)
- Vzhľadom na zložitosť kódu je nutné s rozvahou navrhovať celú aplikáciu!

4 Záver

- „Pravé“ koprogramy sa do jazyka Python zaviedli až vo verzii 3.5
- Na ich definíciu a použitie sa pozrieme v ďalšej prednáške o týždeň

Príklad na cvičení

- V danom vstupnom súbore program vyhľadá počet výskytov slova na vstupe programu

- Ide o kombináciu príkazov ‘grep’ a ‘wc’ ;)
- Napr. ‘grep –o zivot vesmir.txt | wc –l’ (prepínač –o programu grep znamená, že každý výskyt slova ‘zivot’ v súbore ‘vesmir.txt’ sa zobrazí na nový riadok)
- Reálne sa nespúšťajú príkazy/programy OS...

Príklad na cvičení

- V danom vstupnom súbore program vyhľadá počet výskytov slova na vstupe programu

- Nepôjde o paralelné spustenie viacerých vlákien!
- Pôjde o sekvenčné vykonávanie kódu
- Zaujímavosťou je to, že tok vykonávania nie je riadený klasicky (zásobníkom). Na to si treba pri asynchrónnom programovaní zvyknúť...

Doplnková literatúra

http://www.dabeaz.com/coroutines/Coroutines.pdf