PPaDS MMXX

Matus Jokay, C-503, matus.jokay@stuba.sk
Roderik Ploszek, C-512, roderik.ploszek@stuba.sk

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

konzultacie dohodou

Francesco Pierfederici

Distributed Computing with Python

Packt Publishing Ltd.

April 2016

ISBN 978-1-78588-969-1

PPaDS - Definicie

- Paralelny vypocet

  Paralelny vypocet je sucasne vyuzitie viacerych (t.j. viac nez 1) vypoctovych uzlov na dosiahnutie ciela vypoctu

- Distribuovany vypocet

  Distribuovany vypocet je sucasne vyuzitie viacerych (t.j. viac nez 1) vypoctovych uzlov na dosiahnutie ciela vypoctu

PPaDS - Definicie

- **Paralelny** vypocet (vypoctovy uzol = **CPU**)

  Paralelny vypocet je sucasne vyuzitie viacerych (t.j. viac nez 1) vypoctovych uzlov na dosiahnutie ciela vypoctu

- **Distribuovany** vypocet (vypoctovy uzol = **PC**)

  Distribuovany vypocet je sucasne vyuzitie viacerych (t.j. viac nez 1) vypoctovych uzlov na dosiahnutie ciela vypoctu

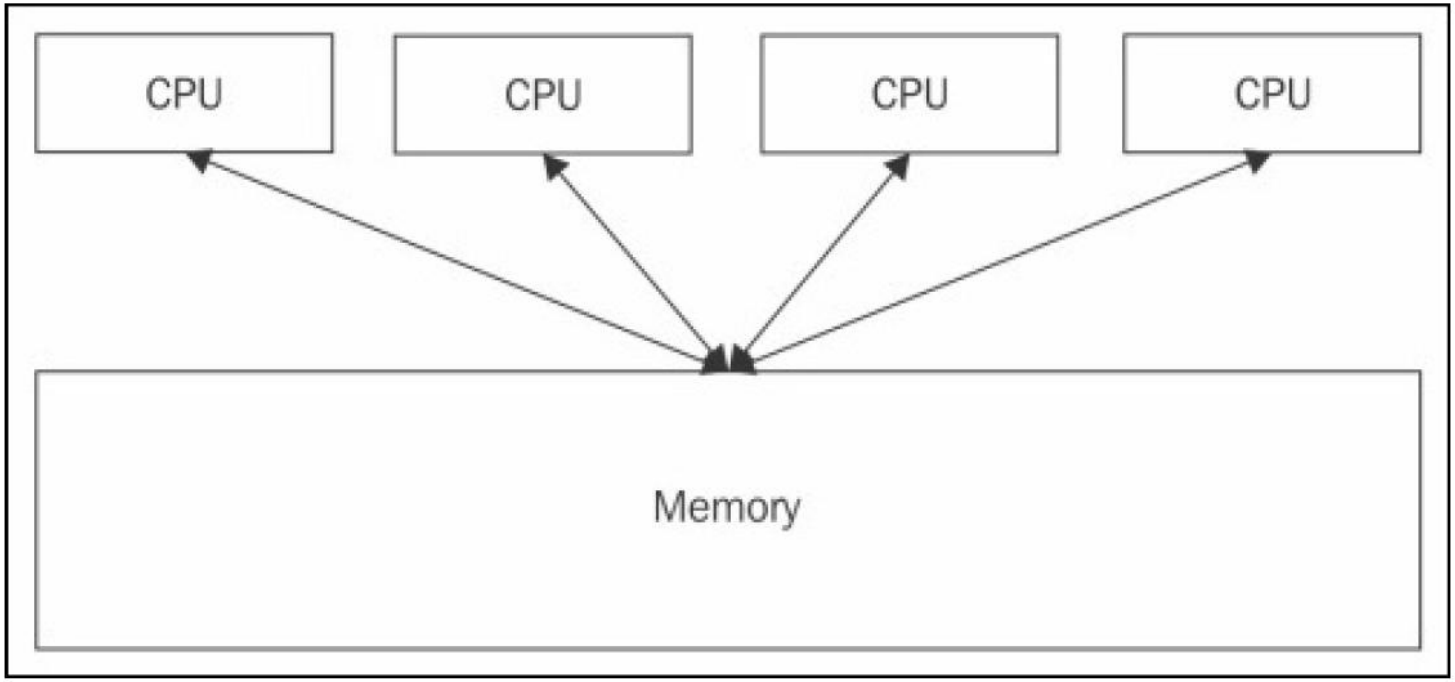
Vyvoj distribuovanej aplikacie

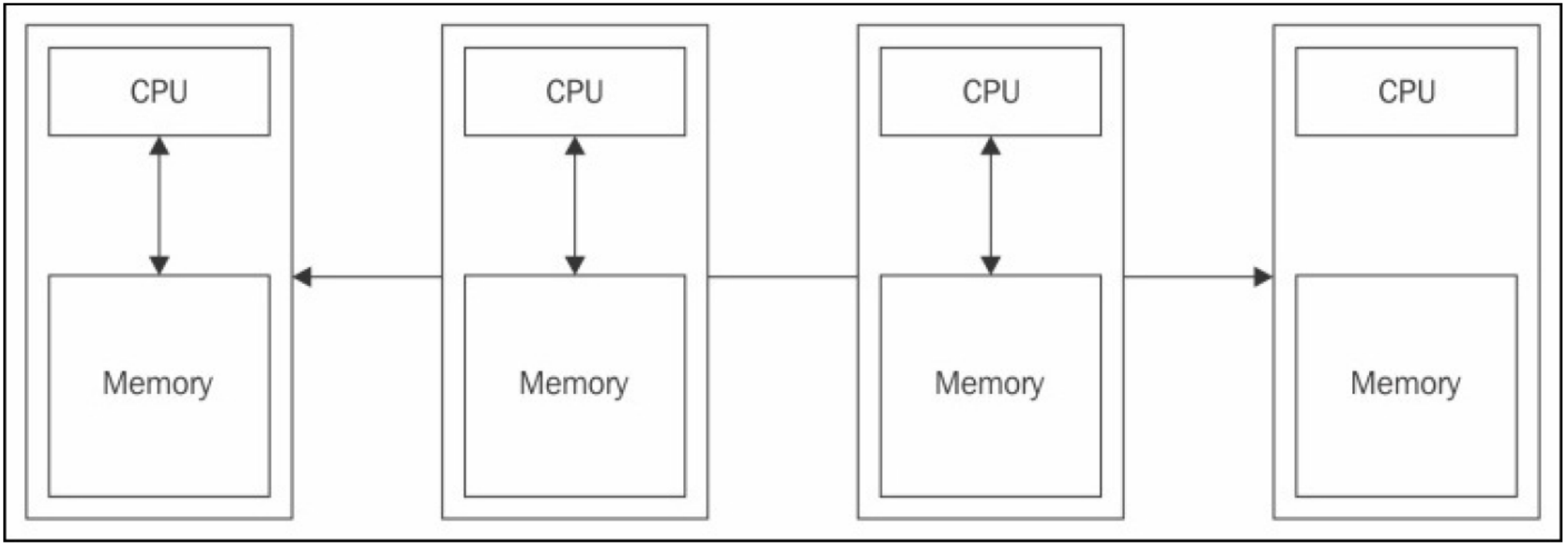
1. Vyvoj jednovlaknovej (jednoprocesovej) aplikacie
2. Vyvoj multiprocesovej (nie multivlaknovej, hoci v zavislosti od implementacneho prostredia to moze byt jeden z medzikrokov) aplikacie
3. Vyvoj distribuovanej aplikacie

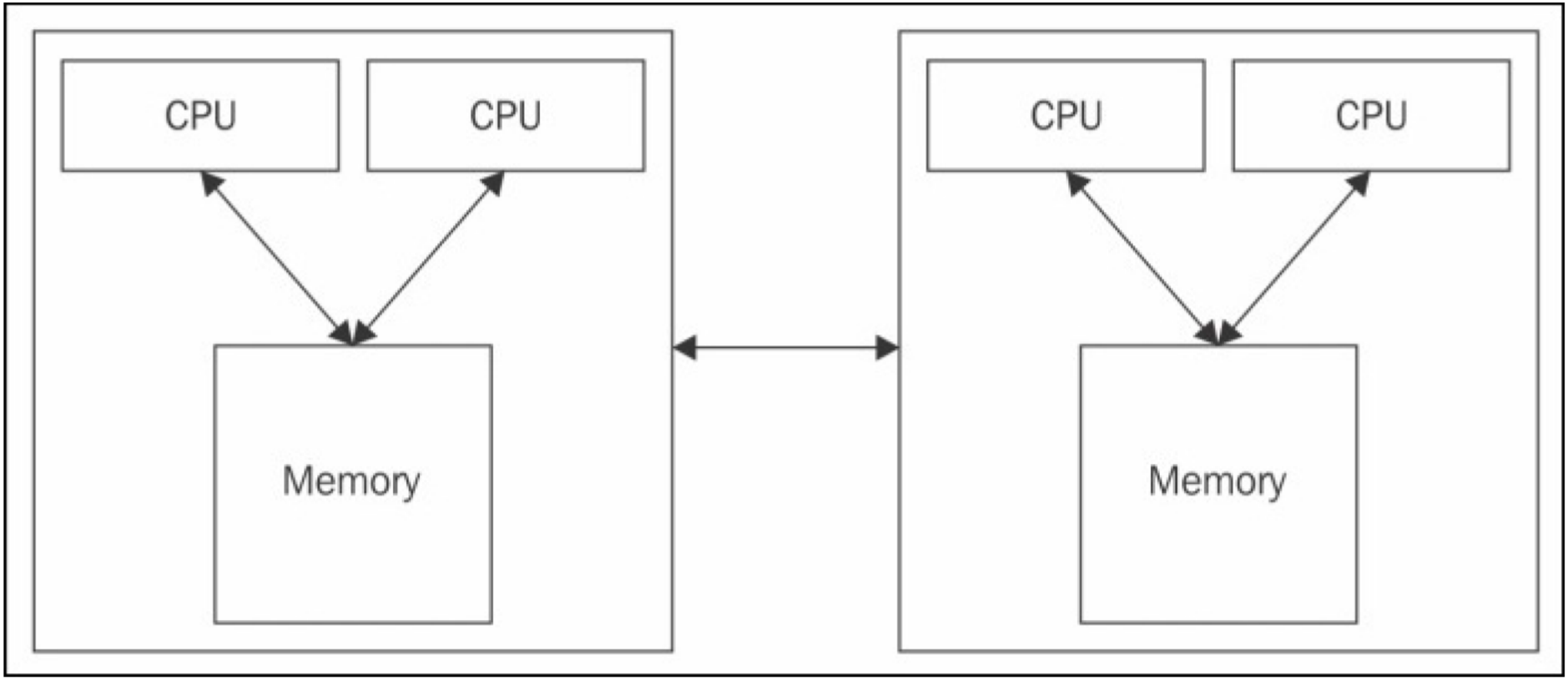
PPaDS – pozor na udaje!

- Skutocnym uzkym hrdlom vypoctov zvacsa (zavisi od typu aplikacie) byvaju samotne udaje, nie CPU

- Niekedy staci zdielany suborovy system (napr. NFS na unixovych systemoch), inokedy zdielana databaza alebo posielanie sprav…


CPU
CPU
CPU
CPU
Memory


CPU
CPU
CPU
CPU
Memory
Memory
Memory
Memory



Hybridna architektura pamate

Asynchronne programovanie

Asynchronne programovanie

- Pre nase ucely budeme program rozdelovat na ulohy (tasky), z ktorych kazda vykonava nejaku operaciu (napr. delenie, ziskanie udajov z db, vypisovanie na monior atd…)

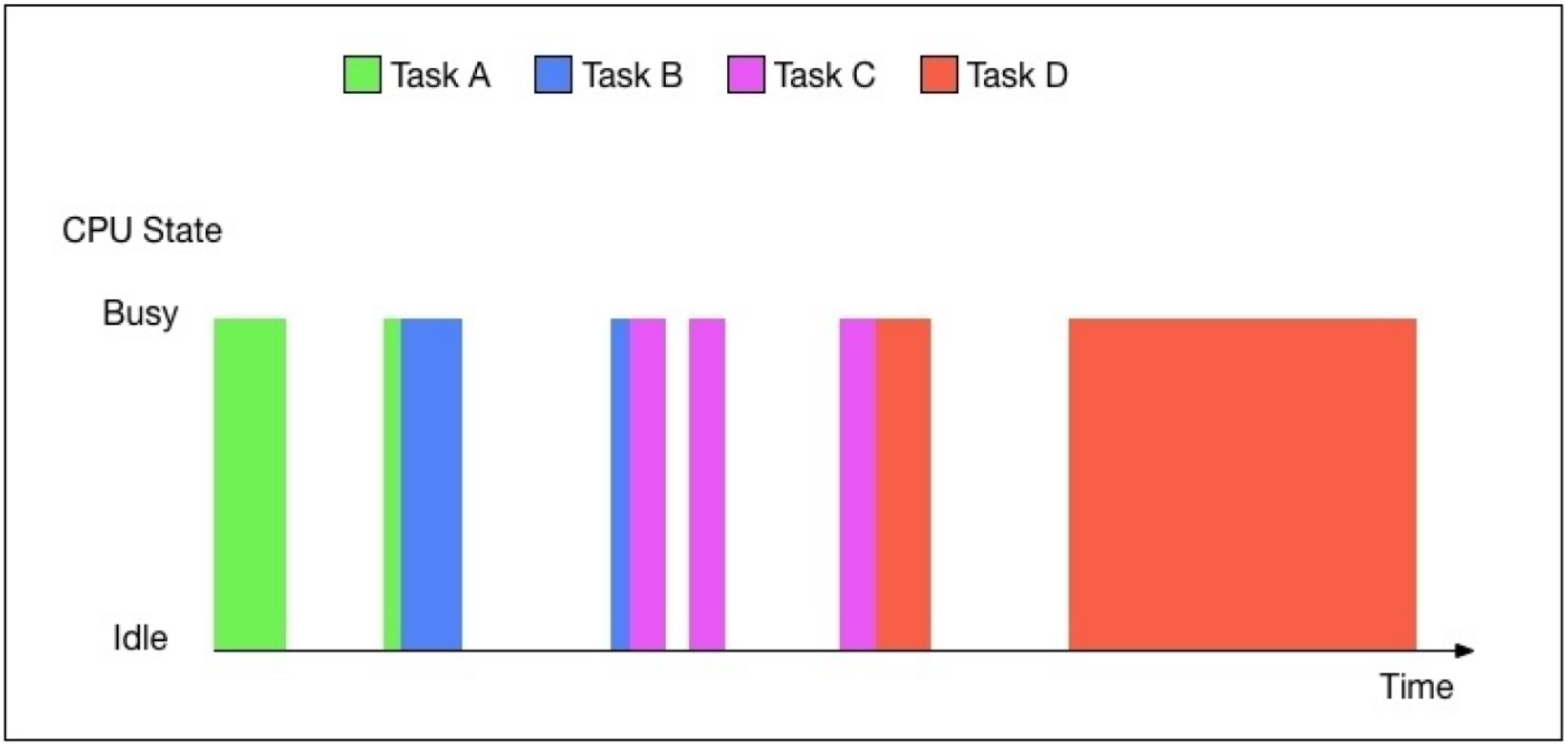
Asynchronne programovanie

- Pre nase ucely budeme program rozdelovat na ulohy (tasky), z ktorych kazda vykonava nejaku operaciu (napr. delenie, ziskanie udajov z db, vypisovanie na monior atd…)
- Pre jednoduchost budeme povazovat funkcie programu za taketo ulohy (tasky)

Asynchronne programovanie

- Priklad: majme program, ktory vykonava 4 ulohy – A, B, C a D
- Kazda z uloh vykonava nejaky vypocet a pristupuje k I/O zariadeniam

- Nasledujuci graf znazornuje beh tychto 4 uloh na jednom jadre



- Pocas I/O ulohy CPU je v stave necinnosti!
- Rozne komponenty maju roznu rychlost pristupu aj prenosovu kapacitu (disk, siet...)
- Podla tejto doby je CPU necinne a nevyuzite

Asynchronne programovanie

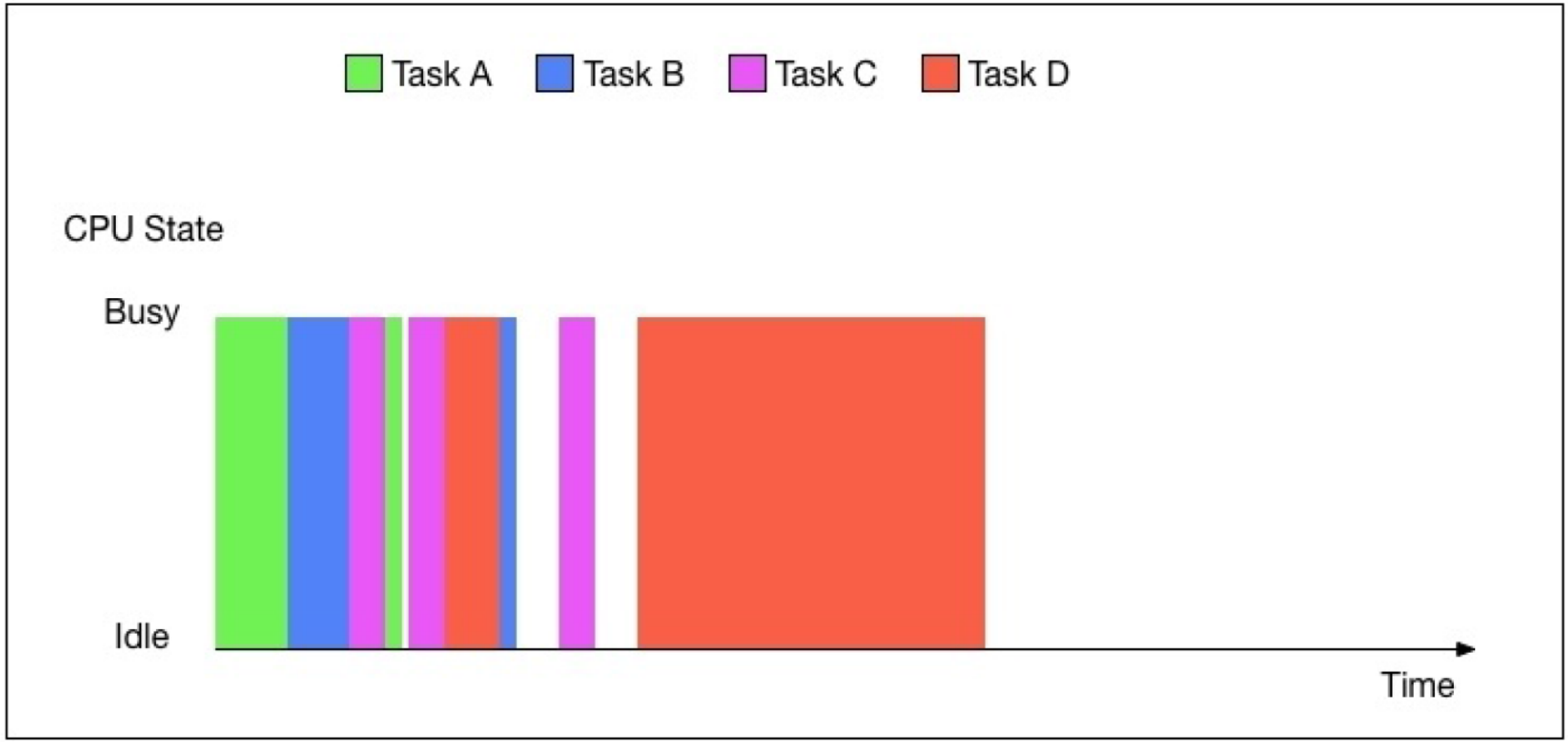
- Idealne by bolo take usporiadanie uloh, aby v case necinnosti (cakania na I/O) jednej ulohy mohla vyuzivat CPU ina uloha

Asynchronne programovanie

- Idealne by bolo take usporiadanie uloh, aby v case necinnosti (cakania na I/O) jednej ulohy mohla vyuzivat CPU ina uloha
- A prave o toto ide v pripade asynchronneho (alebo inak aj udalostami riadeneho (event-driven)) programovania

Asynchronne programovanie

- Idealne by bolo take usporiadanie uloh, aby v case necinnosti (cakania na I/O) jednej ulohy mohla vyuzivat CPU ina uloha
- A prave o toto ide v pripade asynchronneho (alebo inak aj udalostami riadeneho (event-driven)) programovania
- Predosly priklad so styrmi ulohami, ale pomocou asynchronneho pristupu, vidno na nasledujucom obrazku



- Aj v tomto pripade sa ulohy spustaju sekvencne
- Ale neblokuju pocas I/O cakania CPU

Asynchronne programovanie

- Asynchronne programovanie != multivlaknove

Asynchronne programovanie

- Asynchronne programovanie != multivlaknove
  - Pri multivlaknovom OS rozhoduje (planovac), ktore vlakno bude co vykonavat, a ovsem podla pozadovanych I/O automaticky dokaze odlozit vykonavanie vlakien, ktore cakaju na dokoncenie I/O

Asynchronne programovanie

- Asynchronne programovanie != multivlaknove
  - Pri multivlaknovom OS rozhoduje (planovac), ktore vlakno bude co vykonavat, a ovsem podla pozadovanych I/O automaticky dokaze odlozit vykonavanie vlakien, ktore cakaju na dokoncenie I/O
  - Pri asynchronnom programovani sa uloha samotna rozhoduje (v zastupeni programatora, ovsem), ze sa vzda CPU, pokym nebude dokoncena I/O operacia!

Asynchronne programovanie

- Asynchronne programovanie != multivlaknove
  - Pri multivlaknovom OS sa ulohy mozu vykonavat paralelne
  - Pri asynchronnom programovani sa ulohy stale vykonavaju sekvencne! Ide vsak o efektivnejsie vyuzitie CPU

Koprogram (coroutine)

- Podla wikipedie su koprogramy vseobecnejsie nez podprogramy (procedury, funkcie, metody), nakolko maju viac vstupnych bodov, pozastaveni a obnoveni vypoctu v roznych miestach svojej definicie.

Koprogram (coroutine)

- Zivotny cyklus podprogramov je riadeny zasobnikom (posledne zavolany podprogam urobi navrat ako prvy)

- Zivotny cyklus koprogamu zavisi vyhradne na pouziti

Koprogram (coroutine)

- Podprogram ma iba jeden vstupny bod (zaciatok, pri jeho zavolani), a je mozne iba raz sa z neho vratit
- Koprogramy sa mozu vracat viac krat (v pythone prikazom *yield*)

Koprogram (coroutine)

- Podprogram ma iba jeden vstupny bod (zaciatok, pri jeho zavolani), a je mozne iba raz sa z neho vratit
- Koprogramy sa mozu vracat viac krat (v pythone prikazom *yield*)
  - Zaciatkom koprogramu je vstupny bod pri jeho volani
  - Nasledujuce vstupne body su dane prikazom *yield*

Koprogram (coroutine)

- Ako to funguje v praxi

Koprogram (coroutine)

- Ako to funguje v praxi
- Pri prvom zavolani koprogramu sa zacne vykonavat od svojho zaciatku podobne ako podprogram

Koprogram (coroutine)

- Ako to funguje v praxi
- Pri prvom zavolani koprogramu sa zacne vykonavat od svojho zaciatku podobne ako podprogram
- Akonahle narazi na prikaz *yield*, vrati vysledok a odovzda riadenie do volajuceho (pod/ko) programu podobne, ako prikaz *return* pri klasickom podprograme

Koprogram (coroutine)

- Avsak pri dalsom volani toho isteho koprogramu (v pripade Pythonu pomocou metody *next*) sa koprogram nezacne vykonavat od zaciatku, ale od prikazu, ktory sa nachadza bezprostredne za poslednym vykonanym prikazom *yield*

Koprogram (coroutine)

- Je vhodny skor k obnoveniu vykonavania, nie k restartovaniu (od zaciatku tela svojej definicie)
- Je schopny udrziavat medzi jednotlivymi volaniami
  - Stav
  - Premenne (podobne ako uzavery v Pythone)
  - Bod aktualneho vykonavania
  - Vysledok nemusi byt vrateny na konci vetvy kodu

Koprogram (coroutine)

- Na pochopenie koprogramov v Pythone je nutne chapat generatory, a k nim potrebujeme vedomosti o iteratoroch ;)

Koprogram (coroutine)

- Na pochopenie koprogramov v Pythone je nutne chapat generatory, a k nim potrebujeme vedomosti o iteratoroch ;)
- Generatory su tiez zovseobecnenim podprogramov, ale su obmedzenejsie nez koprogramy

Koprogram (coroutine)

- Na pochopenie koprogramov v Pythone je nutne chapat generatory, a k nim potrebujeme vedomosti o iteratoroch ;)
- Generatory su tiez zovseobecnenim podprogramov, ale su obmedzenejsie nez koprogramy
  - Podobne ako koprogramy umoznuju viac navratov

Koprogram (coroutine)

- Na pochopenie koprogramov v Pythone je nutne chapat generatory, a k nim potrebujeme vedomosti o iteratoroch ;)
- Generatory su tiez zovseobecnenim podprogramov, ale su obmedzenejsie nez koprogramy
  - Podobne ako koprogramy umoznuju viac navratov
  - Nemozu vsak kontrolovat miesto, v ktorom bude pokracovat vykonavanie kodu po zavolani *yield*

Koprogram (coroutine)

- Na pochopenie koprogramov v Pythone je nutne chapat generatory, a k nim potrebujeme vedomosti o iteratoroch ;)
- Generatory su tiez zovseobecnenim podprogramov, ale su obmedzenejsie nez koprogramy
  - Podobne ako koprogramy umoznuju viac navratov
  - Nemozu vsak kontrolovat miesto, v ktorom bude pokracovat vykonavanie kodu po zavolani *yield*
  - Vracaju kontrolu vzdy na miesto, z ktoreho boli zavolane

1 Iterovanie v Pythone

- Ufajme, ze sme dostatocne oboznameni s iterovanim roznych objektov v Pythone (retazce, zoznamy, n-tice, subory…)

1 Iterovanie v Pythone

- Ufajme, ze sme dostatocne oboznameni s iterovanim roznych objektov v Pythone (retazce, zoznamy, n-tice, subory…)

```
>>> 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

- Preco je mozne iterovat nielen retazce a zoznamy, ale aj komplexnejsie objekty?

1 Iterovanie v Pythone

- Preco je mozne iterovat nielen retazce a zoznamy, ale aj komplexnejsie objekty?
- Dovodom je rozhranie Pythonu, tzv. iteracny protokol (iteration protocol)

1 Iterovanie v Pythone

- Preco je mozne iterovat nielen retazce a zoznamy, ale aj komplexnejsie objekty?
- Dovodom je rozhranie Pythonu, tzv. iteracny protokol (iteration protocol)
- Objekt, ktory implementuje metody __*iter*__ a __*next*__, sa stava iteratorom

1 Iterovanie v Pythone

- Preco je mozne iterovat nielen retazce a zoznamy, ale aj komplexnejsie objekty?
- Dovodom je rozhranie Pythonu, tzv. iteracny protokol (iteration protocol)
- Objekt, ktory implementuje metody __*iter*__ a __*next*__, sa stava iteratorom
  - Prva z metod (__iter__) vracia objekt, ktory je mozne pomocou metody __next__ iterovat

1 Iterovanie v Pythone

- Preco je mozne iterovat nielen retazce a zoznamy, ale aj komplexnejsie objekty?
- Dovodom je rozhranie Pythonu, tzv. iteracny protokol (iteration protocol)
- Objekt, ktory implementuje metody __*iter*__ a __*next*__, sa stava iteratorom
  - Prva z metod (__iter__) vracia objekt, ktory je mozne pomocou metody __next__ iterovat
  - Druha z metod (__next__) vracia pri sekvencnom volani prvky postupnosti pekne jeden za druhym

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

- Predosly kod vypise na vystup

```
0
1
2
```

1 Iterovanie v Pythone

- Predosly kod vypise na vystup

```
0
1
2
```

- Pre lepsie pochopenie protokolu mozeme “manualne” prelistovat objekt, ktory implementuje protokol iterovania

1 Iterovanie v Pythone

- Predosly kod vypise na vystup

```
0
1
2
```

- Pre lepsie pochopenie protokolu mozeme “manualne” prelistovat objekt, ktory 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 vystup:

```
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 vytvorime instanciu triedy *MyIterator*, a potom pomocou volania *next*() postupne ziskavame hodnoty postupnosti, ktoru je tento objekt schopny generovat

1 Iterovanie v Pythone

- Najprv vytvorime instanciu triedy *MyIterator*, a potom pomocou volania *next*() postupne ziskavame hodnoty postupnosti, ktoru je tento objekt schopny generovat
- Po vycerpani prvkov sa generuje vynimka *StopIteration*

1 Iterovanie v Pythone

- Najprv vytvorime instanciu triedy *MyIterator*, a potom pomocou volania *next*() postupne ziskavame hodnoty postupnosti, ktoru je tento objekt schopny generovat
- Po vycerpani prvkov sa generuje vynimka *StopIteration*
- Cyklus *for* v Pythone vyuziva presne tento mechanizmus: vola *next*() na iteratore, ktory na zaciatku ziska pomocou volania *iter*(), a zachytava vynimku *StopIteration*, aby vedel, kedy je koniec iterovania

2 Generator v Pythone

- Je volatelny (angl. *callable*) objekt (funkcia), ktory vracia instanciu generatora (iterator); ten miesto okamziteho vratenia celeho vysledku postupne generuje ciastkove vysledky

2 Generator v Pythone

- Je volatelny (angl. *callable*) objekt (funkcia), ktory vracia instanciu generatora (iterator); ten miesto okamziteho vratenia celeho vysledku postupne generuje ciastkove vysledky
- Na generovanie ciastkoveho vysledku sa pouziva klucove slovo *yield*

2 Generator v Pythone

- Je volatelny (angl. *callable*) objekt (funkcia), ktory vracia instanciu generatora (iterator); ten miesto okamziteho vratenia celeho vysledku postupne generuje ciastkove vysledky
- Na generovanie ciastkoveho vysledku sa pouziva klucove slovo *yield*

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

2 Generator v Pythone

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

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

2 Generator v Pythone

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

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

Vystup:

```
2
1
0
```

2 Generator v Pythone

- V com je rozdiel medzi obycajnou funkciou (metodou) a generatorom?

2 Generator v Pythone

- V com je rozdiel medzi obycajnou funkciou (metodou) a generatorom?
- V pouziti klucoveho slova *yield*

2 Generator v Pythone

- Zavolanie (funkcie) generatora negeneruje ziaden objekt postupnosti, ale tzv. objekt generatora (generatorovy objekt, ktory je iterator)

2 Generator v Pythone

- Zavolanie (funkcie) generatora negeneruje ziaden objekt postupnosti, ale tzv. objekt generatora (generatorovy objekt, ktory je iterator)

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

2 Generator v Pythone

- Na aktivovanie generatoroveho iteratora je potrebne na nom vyvolat metodu *next*()

2 Generator v Pythone

- Na aktivovanie generatoroveho iteratora je potrebne na nom vyvolat metodu *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 Generator v Pythone

- Kazde volanie *next*() produkuje prave jednu hodnotu z postupnosti generovanej generatorovym iteratorom

2 Generator v Pythone

- Kazde volanie *next*() produkuje prave jednu hodnotu z postupnosti generovanej generatorovym iteratorom
- Po vyprazdneni mnoziny prvkov, z ktorych sa generuje postupnost, sa vyvola vynimka *StopIteration*

2 Generator v Pythone

- Kazde volanie *next*() produkuje prave jednu hodnotu z postupnosti generovanej generatorovym iteratorom
- Po vyprazdneni mnoziny prvkov, z ktorych sa generuje postupnost, sa vyvola vynimka *StopIteration*
- Generatorove **funkcie** su v podstate zjednodusenym zapisom iteratorov – odpada nutnost definovat **triedu** s metodami __*iter*__ a __*next*__

2 Generator v Pythone

- Po vyprazdneni objektu generatora (ked uz generuje vynimku *StopIteration*) ho uz nevieme pouzit na opatovne generovanie postupnosti

2 Generator v Pythone

- Po vyprazdneni objektu generatora (ked uz generuje vynimku *StopIteration*) ho uz nevieme pouzit na opatovne generovanie postupnosti

- Ak chceme znovu ziskat postupnost generovanu generatorovym objektom, musime ziskat novy objekt generatora a iterovat ho

3 Koprogramy v Pythone

3 Koprogramy v Pythone

- Vyraz *yield* mozeme pouzit aj na pravej strane priradenia, na znak toho, ze chceme prvky nie generovat, ale spracovavat

3 Koprogramy v Pythone

- Vyraz *yield* mozeme pouzit aj na pravej strane priradenia, na znak toho, ze chceme prvky nie generovat, ale spracovavat
- Ak taketo nieco pouzijeme vo funkcii/metode Pythonu, vytvorime tzv. koprogram

3 Koprogramy v Pythone

- Vyraz *yield* mozeme pouzit aj na pravej strane priradenia, na znak toho, ze chceme prvky nie generovat, ale spracovavat
- Ak taketo nieco pouzijeme vo funkcii/metode Pythonu, vytvorime tzv. koprogram
- Koprogram v Pythone je, **VELMI zjednodusene** napisane, typ funkcie, ktora moze prerusit a znovu obnovit svoje vykonavanie na miestach s vyskytom klucoveho slova *yield* na pravej strane priradenia

3 Koprogramy v Pythone

- Koprogramy v Pythone nie su jednoduchymi generatormi! (aj ked sa v nich pouziva to iste klucove slovo *yield*)

- Dovody:
  - Koprogramy nie su spojene s iteraciou
  - Generatorove objekty hodnoty produkuju, koprogramy ich konzumuju

3 Koprogramy v Pythone

- Koprogramy vyuzivaju na svoju cinnost nasledovne 3 mechanizmy

3 Koprogramy v Pythone

- Koprogramy vyuzivaju na svoju cinnost nasledovne 3 mechanizmy
  1. *yield*()
  2. *send*()
  3. *close*()

3 Koprogramy v Pythone

- Koprogramy vyuzivaju na svoju cinnost nasledovne 3 mechanizmy
  1. *yield*() – uspi vykonavanie koprogramu do najblizsieho vyvolania metody *send*() nad objektom koprogramu
  2. *send*()
  3. *close*()

3 Koprogramy v Pythone

- Koprogramy vyuzivaju na svoju cinnost nasledovne 3 mechanizmy
  1. *yield*() – uspi vykonavanie koprogramu do najblizsieho vyvolania metody *send*() nad objektom koprogramu
  2. *send*() – sluzi na poslanie udajov na spracovanie koprogramu (teda zaroven prebudi koprogram)
  3. *close*()

3 Koprogramy v Pythone

- Koprogramy vyuzivaju na svoju cinnost nasledovne 3 mechanizmy
  1. *yield*() – uspi vykonavanie koprogramu do najblizsieho vyvolania metody *send*() nad objektom koprogramu
  2. *send*() – sluzi na poslanie udajov na spracovanie koprogramu (teda zaroven prebudi koprogram)
  3. *close*() – ukonci koprogram (posle mu vynimku *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 definovany koprogram spracuva jeden argument (retazec)

```
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 definovany koprogram spracuva jeden argument (retazec)
- Po vypisani uvodnej spravy zacne vykonavat nekonecny cyklus vnoreny 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 definovany koprogram spracuva jeden argument (retazec)
- Po vypisani uvodnej spravy zacne vykonavat nekonecny cyklus vnoreny do *try-except* bloku
- Po zachyteni vynimky *GeneratorExit* cinnost koprogramu (po vypise spravy) konci

```
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 definovany koprogram spracuva jeden argument (retazec)
- Po vypisani uvodnej spravy zacne vykonavat nekonecny cyklus vnoreny do *try-except* bloku
- Po zachyteni vynimky *GeneratorExit* cinnost koprogramu (po vypise spravy) konci
- Telo cyklu je jednoduche – pomocou *yield* ziskame udaje, ktore ulozime do premennej *text* a nasledne 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

- Zavolanim *complain_about('Ruby')* sa vytvori objekt koprogramu (nic ine sa neudeje!)

3 Spustenie koprogramu

- Zavolanim *complain_about('Ruby')* sa vytvori objekt koprogramu (nic ine sa neudeje!)
- Az naslednym zavolanim metody *next*() nad tymto objektom sa spusti vykonavanie koprogramu (vidime to na zaklade vypisu uvodnej spravy)

3 Spustenie koprogramu

- Zavolanim *complain_about('Ruby')* sa vytvori objekt koprogramu (nic ine sa neudeje!)
- Az naslednym zavolanim metody *next*() nad tymto objektom sa spusti vykonavanie koprogramu (vidime to na zaklade vypisu uvodnej spravy)
- V cykle vykonavanie koprogramu dosiahne riadok *text = (yield)*, ktory prerusi vykonavanie funkcie

3 Spustenie koprogramu

- Zavolanim *complain_about('Ruby')* sa vytvori objekt koprogramu (nic ine sa neudeje!)
- Az naslednym zavolanim metody *next*() nad tymto objektom sa spusti vykonavanie koprogramu (vidime to na zaklade vypisu uvodnej spravy)
- V cykle vykonavanie koprogramu dosiahne riadok *text = (yield)*, ktory prerusi vykonavanie funkcie
- Koprogram obnovi svoju cinnost, akonahle dostane udaje na spracovanie pomocou metody *send*()

3 Spustenie koprogramu

- Kazde zavolanie metody *send*() posunie vykonavanie kodu v koprograme na dalsie *yield*

3 Spustenie koprogramu

- Kazde zavolanie metody *send*() posunie vykonavanie kodu v koprograme na dalsie *yield*
- Koprogram je mozne ukoncit zavolanim metody *close*(), ktora generuje vynimku *GeneratorExit*

3 Spustenie koprogramu

- Kazde zavolanie metody *send*() posunie vykonavanie kodu v koprograme na dalsie *yield*
- Koprogram je mozne ukoncit zavolanim metody *close*(), ktora generuje vynimku *GeneratorExit*
- Ak by sme v ukazkovom kode vynechali blok *try-catch*, nespracuje sa vynimka *GeneratorExit*; koprogram iba skonci svoju cinnost

3 Spustenie koprogramu

- Kazde zavolanie metody *send*() posunie vykonavanie kodu v koprograme na dalsie *yield*
- Koprogram je mozne ukoncit zavolanim metody *close*(), ktora generuje vynimku *GeneratorExit*
- Ak by sme v ukazkovom kode vynechali blok *try-catch*, nespracuje sa vynimka *GeneratorExit*; koprogram iba skonci svoju cinnost (**skuste v ramci cvicenia**)

3 Spustenie koprogramu

- Po ukonceni cinnosti koprogramu pomocou *close*() objekt koprogramu sice ostava jestvovat, ale nemozno mu ani poslat udaje pomocou *send*(), ani obnovit jeho vykonavanie pomocou metody *next*()

3 Spustenie koprogramu

- Po ukonceni cinnosti koprogramu pomocou *close*() objekt koprogramu sice ostava jestvovat, ale nemozno mu ani poslat udaje pomocou *send*(), ani obnovit jeho vykonavanie pomocou metody *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 pouziti koprogramu je vzdy nutne (po vytvoreni objektu koprogramu) pouzit metodu *next*() na jeho aktivovanie

3 Spustenie koprogramu

- **Pri pouziti koprogramu je vzdy nutne (po vytvoreni objektu koprogramu) pouzit metodu *next*() na jeho aktivovanie**

3 Spustenie koprogramu

- Pri pouziti koprogramu je vzdy nutne (po vytvoreni objektu koprogramu) pouzit metodu *next*() na jeho aktivovanie
- Pri castom pouzivani koprogramov to moze byt dost otravne, preto je mozne v Pythone vyuzit **dekoratory** na obidenie nutnosti vyvolavania *next*() po vytvoreni objektu koprogramu (aby sa predislo napriklad vynechaniu tohto volania zo zabudlivosti)

```
>>> 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 Zaver

4 Zaver

- Vyhody asynchronneho programovania pomocou rozsirenych generatorov:
  1. Setrenie pamatou
  2. Vytazenie CPU pri I/O operaciach

4 Zaver

- Vyhody asynchronneho programovania pomocou rozsirenych generatorov:
  1. Setrenie pamatou
  2. Vytazenie CPU pri I/O operaciach

- Nevyhody:
  1. Vyssia zlozitost kodu
  2. Nutnost vlastneho planovaca behu koprogramov

4 Zaver

- Na co si dat pozor pri pisani koprogramov

- Vhodne iba pre aplikacie, kde sa vela casu travi v I/O operaciach
- Treba sa vyhybat funkciam, ktore su blokujuce (tym sa straca zmysel asynchronneho kodu)
- Vzhladom na zlozitost kodu je nutne s rozvahou navrhovat celu aplikaciu!

4 Zaver

- „Prave“ koprogramy sa do jazyka Python zaviedli az vo verzii 3.5
- Na ich definiciu a pouzitie sa pozrieme v dalsej prednaske o tyzden

Priklad na cviceni

- V danom vstupnom subore program vyhlada pocet vyskytov slova na vstupe programu

- Ide o kombinaciu prikazov ‘grep’ a ‘wc’ ;)
- Napr. ‘grep –o zivot vesmir.txt | wc –l’ (prepinac –o programu grep znamena, ze kazdy vyskyt slova ‘zivot’ v subore ‘vesmir.txt’ sa zobrazi na novy riadok)
- Realne sa nespustaju prikazy systemu...

Priklad na cviceni

- V danom vstupnom subore program vyhlada pocet vyskytov slova na vstupe programu

- Nepojde o paralelne spustenie viacerych vlakien!
- Pojde o sekvencne vykonavanie kodu
- Zaujimavostou je to, ze tok vykonavania nie je riadeny klasicky (zasobnikom). Na to si treba pri asynchronnom programovani zvyknut...