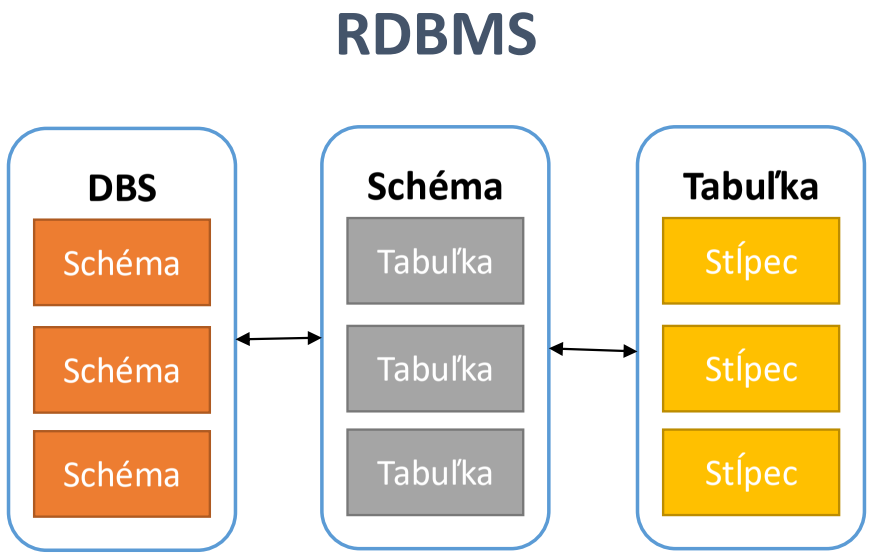
Python a jazyk SQL

- Znalosť SQL je žiadaná pri vývojárskych pozíciách
- SQL je často využívaný pri vývoji programov využívajúce databázy
- Dáta musíme niekde uchovávať
- V SQL sa stretávame s rôznymi operáciami nad dátami v databázach SELECT, INSERT, UPDATE, DELETE, JOIN (LEFT, RIGHT, INNER, FULL OUTER JOIN)
- Používame rôzne operátory AND, OR, NOT, IN, LIKE, UNION, EXISTS, ...
- Často sa využívajú **relačné databázy**
- Systémy, ktoré pracujú nad relačnými databázami sa nazývajú relational database managment system (**RDBMS**) alebo systémy správy relačných databáz, napr. **MySql, MariaDB, Microsoft SQL, SQL Lite, PostgreSQL**
- Existujú aj nerelačné databázy, napr. MongoDB (Document Based DBS)

Python a jazyk SQL

- PostgreSQL DBS rovnako populárna ako MySQL DBS
- PostgreSQL ponúka viac funkcií ako MySQL, zároveň je o niečo pomalšia ako MySQL
- PostgreSQL umožňuje správu používateľov, každému používateľovi môžeme definovať rôzne prístupové práva
- Používateľov vieme začleniť do skupín a skupine delegovať práva
- MySQL umožňuje pracovať s používateľskými účtami, každému účtu definovať práva v rôznych databázach, v rôznych tabuľkách



Každý RDMBS môže mať niekoľko databáz, každá databáza môže mať niekoľko schém, každá schéma môže mať niekoľko tabuliek, každá tabuľka môže mať niekoľko stĺpcov

Manažment pamäte

- Existujú 2 hlavné typy pamätí:
  - Pamäťový zásobník (Stack memory)
  - Voľná pamäť (heap memory - RAM)

- **Pamäťový zásobník** je špeciálna oblasť pamäte RAM, ktorý vytvára dátové štruktúry, v ktorých sú uložené aktívne (volané) funkcie a lokálne premenné
- **Voľná pamäť** je špeciálna oblasť v pamäti RAM. Veľkosť voľnej pamäte je zvyčajne väčšia ako pamäťového zásobníka (možnosť ukladať viac položiek). Objekty tried sú uložené v RAM pamäti

Manažment pamäte

Zásobníková pamäť	Voľná pamäť
Malá veľkosť	Väčšia veľkosť
Rýchly prístup	Pomalý prístup
Ukladanie volaných funkcií a lokálnych premenných	Ukladá objekty (Python zvykne odstrániť objekty a zostane prázdne miesto v pamäti, stane sa fragmentovanou)
Žiadna fragmentácia	Môže nastať fragmentácia

Manažment pamäte

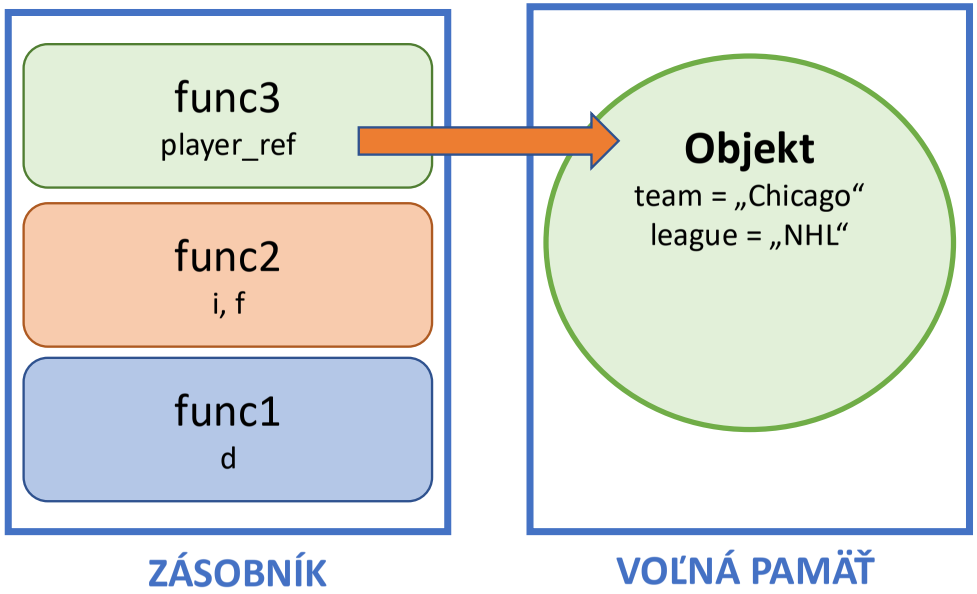
```
def func1():
    d = 10
    func2()

def func2(i):
    f = 30.0
    func3()

def func3()
    player_ref = Player()
```

Volaním funkcií sa do zásobníku, odspodu, ukladajú vrstvy nasledovne: func1 spolu s lokálnou premennou d, func2 s lokálnymi premennými i,f a func3 s premennou - referenciou player_ref na Objekt

```
Class Player:
    team = “Chicago“
    league = “NHL“
```



Manažment pamäte

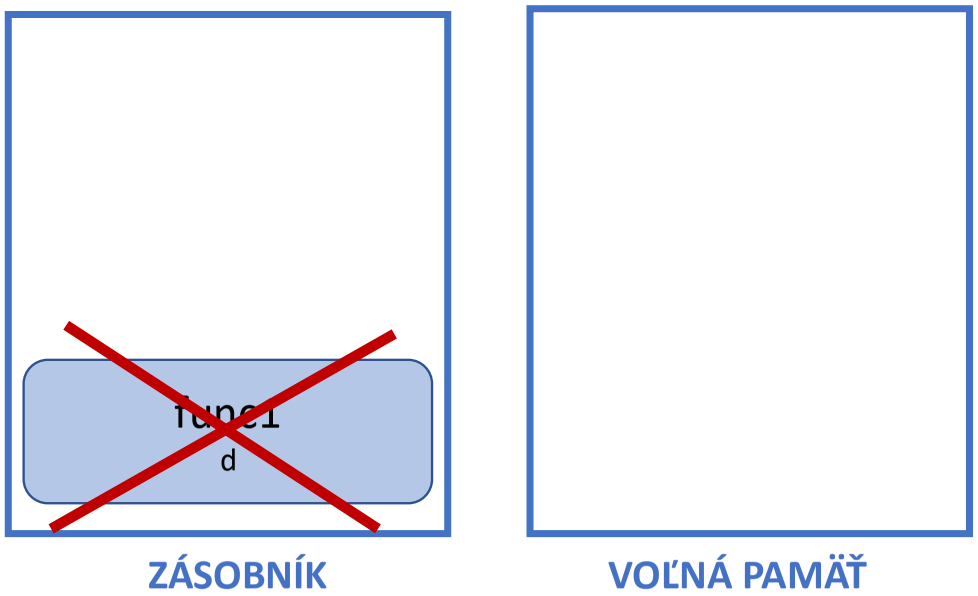
```
def func1():
    D = 10
    func2()

def func2(i):
    f = 30.0
    func3()

def func3()
    player_ref = Player()
```

Ak funkcia 3 skončila, referencia na objekt zanikla, Python odstráni Objekt z voľnej pamäte procesom **Garbage collection**, sekvenčný tok sa **postupne** vracia späť, jednotlivé vrstvy sú zo zásobníka mazané

```
Class Player:
   team = “Chicago“
   league = “NHL“
```



Garbage collection (GC)

- V Pythone sa nemusíme znepokojovať s nevyužívanými objektami vo voľnej pamäti (heap memory/RAM)
- Ak neexistuje referencia zo zásobníka do voľnej pamäte na objekt, takýto objekt sa stáva vhodným objektom pre proces **Garbage collection**
- **Garbage collection** (zbierka odpadu) je proces, ktorý odstráni nevyužité objekty bez referencie z voľnej pamäte (heap memory)
- Každá premenná v Pythone má referenciu na objekt
- Objekt môže mať niekoľko ďalších referencií

  a = [„Jonathan“, 19, False]

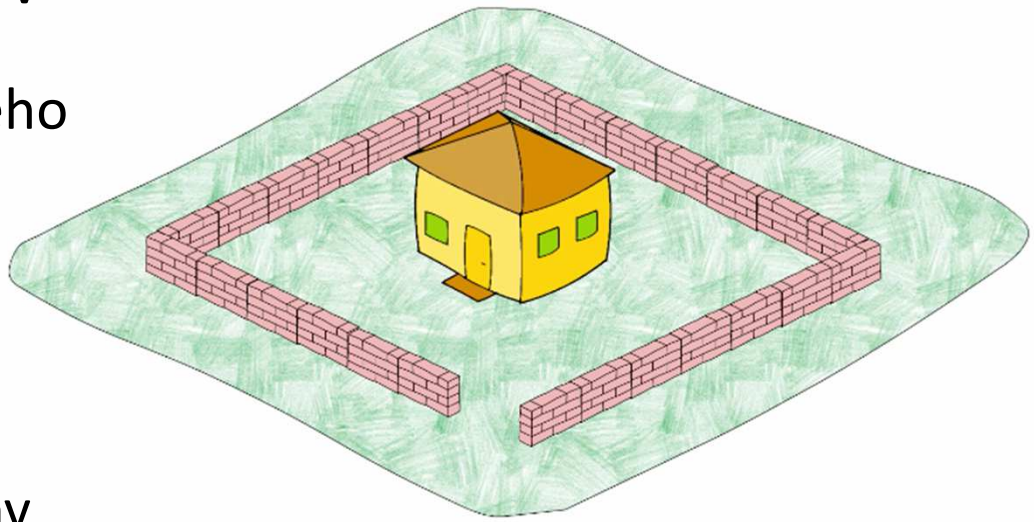
  b = a
- V tomto prípade existujú 2 referencie (a, b) na 1 objekt (list)
- Každý objekt v Pythone má extra pole – **počítadlo referencií**, ktoré rastie alebo klesá v závislosti, či pointer pre daný objekt je vytvorený, alebo zmazaný. Pre príklad je počítadlo referencií rovný 2.
- Ak počítadlo referencií je **rovné nule**, **garbage collection** môže odstrániť objekt z voľnej pamäte (heap memory)
- **del** kľúčové slovo dekrementuje počet referencií v počítadle referencií na daný objekt (**del** a – zrušená referencia premennej **a** na objekt v heap memory), inými slovami **del** odstráni definíciu premennej

Garbage collection (GC)

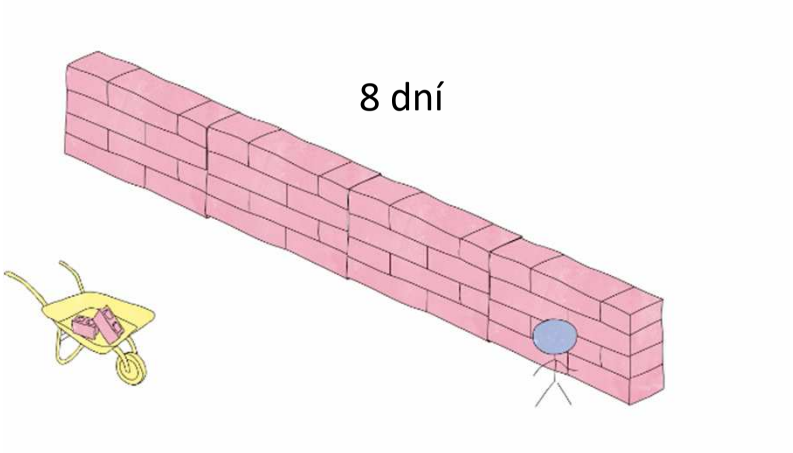
- Všetky premenné v Pythone sú objekty
  a = „Brandon“
  b = 25
  c = 16.25
  d = True
- Premenné – referencie sú uložené v zásobníku (stack memory), ktoré odkazujú na objekt vo voľnej pamäti (heap memory)
- Ak zmažeme premennú – referenciu na objekt, zmaže sa zo zásobníka, ale zmaže sa i referencia na objekt v heap memory. Objekt v heap memory sa stáva vhodným „kandidátom“ na odstránenie pre proces **Garbage collection**
- Ak platí:
  a = [10, “Patrick“, True, 356.2]
  b = [10, “Patrick“, True, 356.2]
- **a** a **b** odkazujú na rozdielne objekty v heap memory

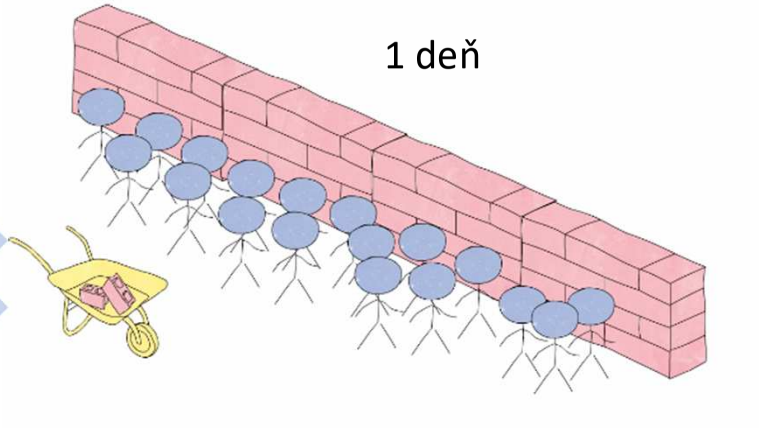
Paralelné programovanie

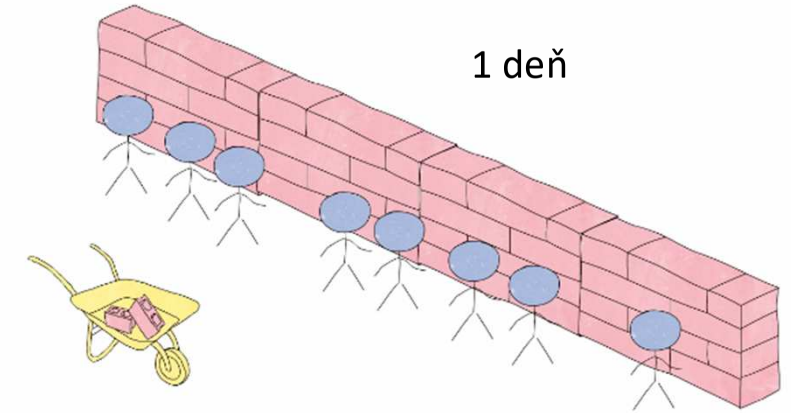
- **Paralelné programovanie** alebo paralelný prístup predstavuje využitie viacerých dostupných zdrojov na vyriešenie úlohy v rýchlejšom čase
- Máme postaviť oplotenie okolo rodinného domu
  - 1 robotník bude pracovať 8 dní
  - 2 robotníci budú pracovať 4 dni
  - 4 robotníci budú pracovať 2 dni
  - 8 robotníkov bude pracovať 1 deň
  - 30 robotníkov bude pracovať 1 deň
- V určitom bode rýchlosť vyriešenia úlohy nezmeníme, vplýva na to viacero faktorov
- Dobré vedieť rozlíšiť, kedy pracovať paralelne a kedy sekvenčne



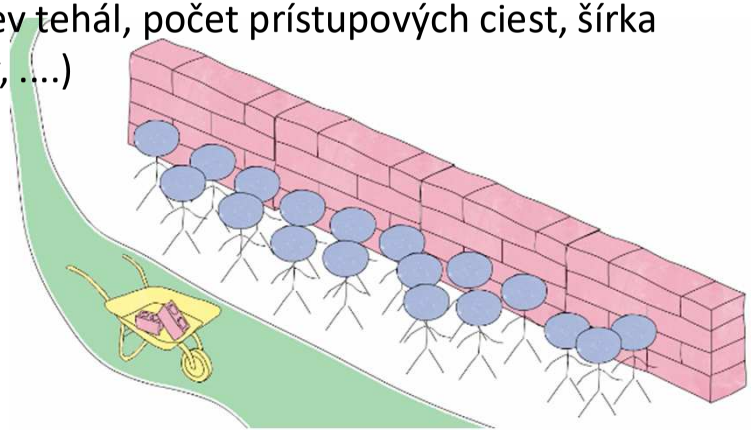
Paralelné programovanie







V určitom momente vplyvom viacerých faktorov, rýchlosť vyriešenia úlohy nezmeníme (ukladanie vrstiev tehál, počet prístupových ciest, šírka cesty, ...)



Paralelné programovanie

- **Aká je motivácia paralelného programovania ?**
  - Defaultne programovacie jazyky sú sekvenčné, to znamená spúšťajú príkazy a bloky príkazov jeden za druhým (riadok za riadkom)

```
def main_method():
    initialize_arrays()
    download_stocks()
    initialize_time_series_models()
    make_prediction()
```

  - V uvedenom príklade (jednovláknová aplikácia), kde metódy sú volané jedna za druhou, používateľ musí čakať, kým skončí jedna metóda, aby bola spustená druhá
  - Čo nie je optimálne riešenie. Časovo náročné operácie (download_stocks()) môžu zapríčiniť zamrznutie aplikácie a používateľ nemusí vedieť, čo sa deje

Paralelné algoritmy - využitie

- V modelovaní:
  - Formácií galaxií
  - Pohybov planét
  - Klimatických zmien
  - Dopravných špičiek
  - Pohybe tektonických dosiek
  - Počasia
- Big data, Data mining
- Umelá inteligencia
- Pokročila virtuálna realita

Paralelné programovanie

- Poznáme:
  - **Sekvenčné algoritmy** – vykonáva sa jedna úloha za druhou (jeden príkaz za druhým)
  - **Paralelné algoritmy** – vykoná sa niekoľko rôznych úloh na rôznych procesoroch, alebo rôznych jadrách, súčasne, skombinujú sa výstupy

Paralelné algoritmy	Sekvenčné algoritmy
• Nájdenie prvočísel v intervale • Vytvoriť z intervalu podmnožiny. V podmnožinách nájdeme prvočísla sekvenčne, výsledky spojíme do jedného výstupu • Big data – rozdeliť big data do podmnožín, vykonať matematické operácie, zlúčiť výsledky do jedného	• numerické operácie • Každý nasledujúci krok sa **spolieha** na výstup predchádzajúceho kroku

- **Motivácia paralelizmu**: zvýšiť výkon algoritmu pri zložitých operáciach vytvorením paralelného algoritmu. Dijkstrov algoritmus nájdenia najkratšej cesty medzi dvoma uzlami

Paralelné programovanie

- **Aká je motivácia paralelného programovania ?**
  - Zmysel je oddeliť niekoľko úloh. Jedna alebo viac úloh, ktoré sú časovo náročné, môžu výrazne ovplyvniť čas spustenia iných úloh.
    - **Príklad**: aplikácia na prácu s akciovými trhmi pravidelne sťahuje dáta z webovej lokality (nech Yahoo Finance)
      - Trvá **2 – 3** minúty kým sa stiahnu všetky dáta, ale my nechceme na 2 – 3 minúty zapríčiniť zamrznutie aplikácie

      ✓ Riešenie: vytvoríme samostatné, oddelené vlákno pre operáciu sťahovania dát z webovej lokality a počas behu procedúry používateľ vie v aplikácii realizovať akékoľvek iné operácie

      Bežný príklad vo Windows OS – kopírovanie veľkého súboru – operácia beží v samostatnom vlákne, čím nezapríčini zamrznutie celého operačného systému

Paralelné programovanie

- **Aká je motivácia paralelného programovania ?**

sťahovanie obrázkov z webu — **VLÁKNO #1**

kopírovanie súborov,
analýza obsahu — **VLÁKNO #2**

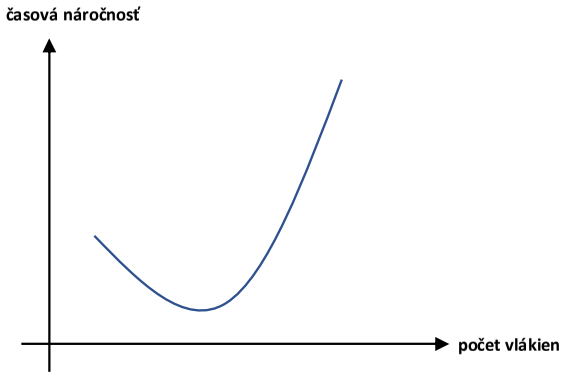
zložité výpočty
(simulácie, zložité
matematické výpočty) — **VLÁKNO #3**

Paralelné programovanie

- **Aká je motivácia paralelného programovania**
  - **zvýšiť výkon algoritmu (zlepšiť responzívnosť aplikácie)** pri zložitých operáciach vytvorením paralelného algoritmu. Dijkstrov algoritmus nájdenia najkratšej cesty medzi dvoma uzlami
  - **mnoho problémov** je tak **veľkých** alebo **zložitých**, že je nepraktické alebo nemožné ich vyriešiť pomocou sekvenčného programu, najmä vzhľadom na obmedzenú pamäť počítača. Vieme vytvoriť aplikáciu i s časovo náročnými operáciami, ktoré môžu bežať súbežne bez zamrznutia aplikácie. **Príklad**: kopírovanie veľkého súboru v OS, webové vyhľadávače/databázy spracúvajúce milióny transakcií každú sekundu
  - **lepšie využiť hardvér** zariadenia: Moderné počítače, vrátane notebookov, sú vo svojej architektúre paralelné s viacerými procesormi/jadrami. Paralelný softvér je špeciálne určený pre paralelný hardvér s viacerými jadrami, vláknami. Vo väčšine prípadov sériové programy bežiace na moderných počítačoch „plytvajú“ potenciálnym výpočtovým výkonom. **Defaultne**, každý Python program je jednovláknový (dôsledok GIL): môže existovať niekoľkojadrový procesor, ktorého potenciál nevyužijeme

Paralelné programovanie

- **Problémy paralelných algoritmov:**
  - **Komunikácia medzi vláknami**: cena komunikácie medzi vláknami naprieč jadrami je príliš veľká. Paralelné programy sú občas **pomalšie** ako sekvenčné. Napr. pre malé databázy (výsledky z rôznych vlákien z rôznych jadier musíme spojiť do jedného). Pre databázy s desiatkami miliónov záznamov je to samozrejme odlišné
  - **Load balance**: Úlohy musia byť rozdelené **rovnomerne** medzi procesormi !!! Zoberme si výpočet faktoriálu, čísel 1 – 1000. Nemôžeme rozdeliť interval na podmnožiny 1 – 500 a 501 – 1000, a tieto podmnožiny spustiť na dvoch procesoroch, pretože to nie je správne. Prvý procesor skončí oveľa skôr ako druhý
  - Paralelné algoritmy je veľmi, veľmi zložité implementovať (ľahké zaniesť chybami, ťažké detekovať chyby)



ZÁSADA: pre malé problémy je zbytočné použiť viacvláknový prístup

Paralelné algoritmy - vlastnosti

- **Komplexnosť**: Všeobecne sú paralelné aplikácie oveľa zložitejšie ako bežné aplikácie, možno až rádovo. Nielenže je spustených viac vlákien súčasne, ale je medzi nimi aj tok údajov

- **Výpočtové nároky:** hlavným zámerom paralelného programovania je znížiť výpočtový čas, ale aby sa to dosiahlo, je potrebné viac času CPU. Napríklad paralelný kód, ktorý beží za 1 hodinu na 8 procesoroch, v skutočnosti využíva 8 hodín času CPU. Obdobne množstvo požadovanej pamäte môže byť väčšie pre paralelné algoritmy ako pre bežné sekvenčné algoritmy, kvôli potrebe replikovania údajov. V prípade krátkodobých paralelných programov môže dôjsť k zníženiu výkonu v porovnaní s podobnou sevkenčnou implementáciou. Režijné náklady spojené s nastavením paralelného prostredia, vytvorením úlohy, komunikáciou a ukončením úlohy môžu tvoriť významnú časť celkového času vykonania pre krátke behy

Paralelné algoritmy - vlastnosti

- **Škálovateľnosť**: Schopnosť škálovania paralelného algoritmu je výsledkom viacerých vzájomne súvisiacich faktorov. Obyčajne pod pojmom škálovateľnosť sa rozumie pridanie ďalších procesorov do algoritmu. Algoritmus môže mať však svoje limity škálovateľnosti. V určitom okamihu pridanie ďalších zdrojov spôsobí zníženie výkonu programu, čo je bežná situácia s mnohými paralelnými aplikáciami. Pri škálovateľnosti zohráva významnú úlohu i samotný hardvér (komunikačná šírka pásma, dostupná pamäť pre dané zariadenie, rýchlosť procesora). Rôzne knižnice používané v programe môžu nezávisle ovplyvniť škálovateľnosť paralelnej aplikácie.

Procesy

- Multithreading je schopnosť procesora (CPU) vykonať niekoľko vlákien súbežne
- Procesy a vlákna sú nezávislé postupnosti vykonania
- Proces je inštancia spusteného programu
  - Ak spustíme SW alebo webový prehliadač, všetko sú to nezávislé procesy
  - OS priraďuje odlišné registre, zásobníkovú pamäť a voľnú /RAM pamäť (heap memory) pre každý jeden proces osobitne
- Pri vetvení procesov sa kopíruje celá pamäť z pôvodného procesu (zložitejší ako vlákno, používa dvakrát toľko pamäte). Zároveň sa spotrebuje viac času, pretože pri vetvení procesu sa kopírujú všetko, čo je uložené v pamäti

PROCES

Vlákna

- Vlákno je „odľahčený“ proces
  - Je to jednotka vykonania určitej úlohy v rámci daného procesu (proces môže mať niekoľko vlákien)
  - Každé vlákno v procese zdieľa pamäť, zdroje, programátori sa musia vysporiadať so súbežným programovaním a multithreadingom (viacvláknovým prístupom)
  - Vlákno je odľahčený proces, nekopíruje sa celý pamäťový priestor, dáta v pamäti, potreba menej času, ušetrená pamäť

Vytvorenie nového vlákna vyžaduje menej zdrojov, je „lacnejšie“, ako vytvorenie nového procesu

PROCES

Zdroje (registre, heap memory (RAM))
Vlákno #1 (stack memory) Vlákno #2 (stack memory) . . . Vlákno #n – k (stack memory)

Každé vlákno má vlastnú zásobníkovú pamäť, zdieľa voľnú pamäť, pričom dáta v zdieľanej časti pamäte musia byť konzistentné. Z toho dôvodu je nutné vysporiadať sa so synchronizáciou

CPU bound vs. I/O bound proces

- **CPU bound** (na procesor viazané) procesy – proces vyžadujúci veľa procesorového času na vykonanie úlohy (matematické operácie, grafické operácie, editácia videa, šifrovanie/dešifrovanie správ). Procesy strávia najviac času spracovania úlohy v procesore
- **I/O bound** (viazané na vstup/výstup) procesy – protiklad CPU bound procesov, procesy strávia najviac času čakania na určitý vstup/výstup. Napríklad kopírovanie, prenos súborov, čítanie z disku, sťahovanie dát z webu, čakanie na používateľa kým nevloží určitý typ dát, webové servre a ich procesy čakajúca na požiadavky klientov

- **Python vlákna nevyužívať** pre **CPU bound procesy**, nemá zmysel, nezvýšime rýchlosť aplikácie. **I/O bound procesy** – **má zmysel** použiť vlákna, zvýši sa rýchlosť aplikácie, vlákna strávia veľa času čakania na vstup/výstup. Ostatné vlákna môžu byť použité pre procesor. Iné jazyky (Java, C++), nemajú toto obmedzenie !!!

Procesy

- Procesy v Pythone spúšťame metódou start(), ktorá má 3 módy:
  - **Spawn**: dostupné pre Windows, Unix, Mac OS
  - **Fork**: dostupné len pre Unix platformy
  - **Forkserver**: dostupné len pre Unix platformy

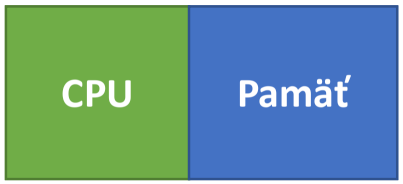
- **Rozdiel medzi Spawn a fork**
  - **Fork** skopíruje celú pamäť (spotreba väčšej pamäte), rýchlejšie spustenie procesu ako pri spawn. Default mód pre Unix platformy.
  - **Spawn** nekopíruje celú pamäť - nekopíruje kľúčové mená súborov a iné, ktoré nie sú potrebné v nových procesoch, čo je zároveň i menšia optimalizácia – ušetrenie mála pamäte. Trvá dlhšie spustiť proces spawn módom. Spawn je default mód pre Windows, Mac OS.

- Forkserver: **hybrid** medzi Fork a spawn. Snaží sa ušetriť pamäť a zároveň byť rýchly

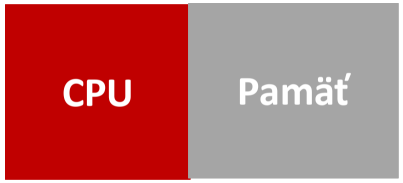
TJ|EvB1

Procesy

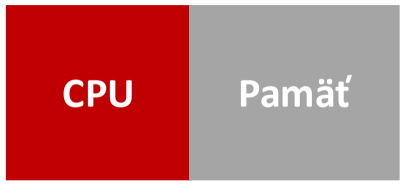
Rodičovský server, z ktorého sa vytvára fork server, zapíše separátny proces so separátnou pamäťou



Fork server, z ktorého sú volané ďalšie forky nových procesov



Fork server sa snaží len klonovať fork server pamäť, nie pamäť rodičovského procesu (ušetrí pamäť). Prvýkrát volaním procesu je trochu pomalší, potrebuje spustiť Fork server mód



Vlákna a procesy

Vlákno:

- Rovnaká pamäť
- „Odľahčený“ proces
- Žiadna izolácia
- Implementovaný GIL v Pythone
- Lacnejšie vytvoriť vlákno

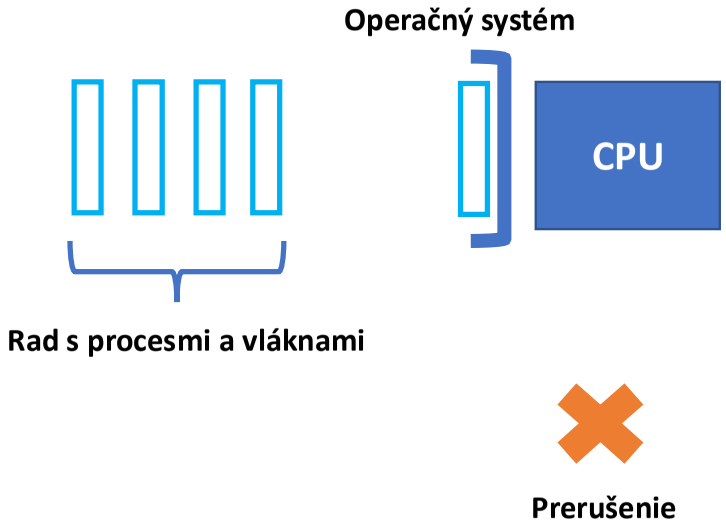
Proces:

- Separátna pamäť
- Náročnejší na zdroje
- Izolovaný
- Nie je implementovaný GIL v Pythone
- Drahšie vytvoriť proces

Vlákna a procesy

(zmena kontextu)

- Ako určiť, kedy sa má vykonať konkrétny proces alebo konkrétne vlákno v procesore ?
  - Toto spravuje operačný systém, ktorý je zodpovedný cez scheduler (plánovač) za plánovanie vykonávania jednotlivých procesov a vlákien
  - Procesy v jednoprocesorovom CPU čakajú na vykonanie v rade za sebou
  - Operačný systém priraďuje/prepína proces, vlákno procesoru jeden za druhým, čo sa nazýva zmena kontextu
  - Zmena kontextu nastáva v malých časových úsekoch, alebo
  - Ak nastane prerušenie v procese, napr. proces čaká na vstup, alebo výstup z iného procesu, scheduler proces vymení s iným pripraveným procesom v rade, aktuálny proces zaradí na koniec radu pripravených (čakajúcich) procesov
  - Zmenou kontextu – prepínanie jednej úlohy za druhou vyvoláva u používateľa dojem, že sa jedná o paralelné spracovanie požiadavky, napriek tomu, že aplikácia beží na jednom CPU s jedným jadrom
  - Čas, ktorý strávi OS na výmenu procesov v procesore, zmena kontextu, je veľmi rýchla operácia, ale pri tisíckach procesoch alebo vláknach v rade, je to plytvanie času systému (nestrávime čas vykonávaním programu)



Paralelné programovanie vs. súbežné (konkurentné) programovanie

- **Paralelné programovanie a súbežné (konkurentné) programovanie:**
  - Ak používame viaceré vlákna (multithreading) – **nie je** nutné ich spustiť **paralelne**, závisí od konkrétneho OS alebo zariadenia
  - **Súbežné programovanie**: ak máme zariadenie s 1 procesorom, s 1 jadrom, vieme zabezpečiť viacvláknový beh aplikácie, ktorý je riadený pomocou **time-slicing** v 1 jadre. Čo môže byť **pomalšie**, ako keby sme implementovali sekvenčný, jednovláknový prístup. **Jedno** vlákno v danom čase !

    Viacvláknová aplikácia – spustíme vlákno 1 na krátky časový úsek, potom spustíme vlákno 2 na krátky časový úsek, potom vlákno 3, ..., vlákno n. Zatiaľ čo jedno vlákno beží, ostatné vlákna čakajú na OS a procesor.

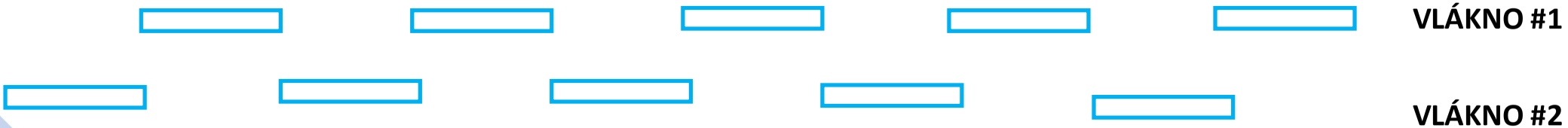
  - **Paralelné programovanie (spustenie)**: vieme spustiť viac vlákien **v rovnakom čase** vo viacerých jadrách alebo na viacerých procesoroch !!!

    Jeden procesor (jadro) spracuje vlákno 1, druhý procesor (jadro) spracuje vlákno 2, tretí procesor (tretie jadro) spracuje vlákno 3, ... Všetky vlákna sú spustené **v rovnakom čase**. Počas implementácie paralelného algoritmu sa pýtame systému, koľko procesorov, alebo jadier má k dispozícii

Súbežné programovanie
(time slicing)

- Multithreading a algoritmus časového úseku (time-slicing algorithm)
  - Predpokladajme, že máme **k** vlákien (viac ako 1 vlákno v našej aplikácii)
  - Majme 1 jadrové CPU, ktoré sa musí vysporiadať s **k** vláknami našej aplikácie
    - Prístup 1: použitie time-slicing algoritmu

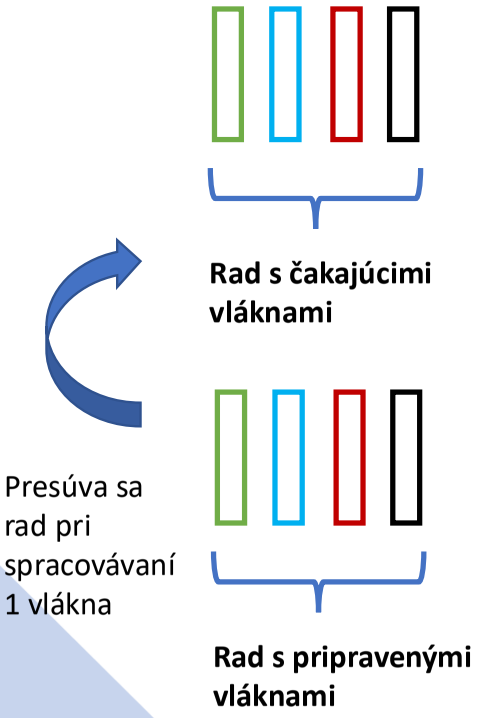
Časový úsek behu vlákna alebo procesu v 1 jadrovom CPU nazýva time-slicing alebo algoritmus krájania času. Vykonáva ho Scheduler operačného systému

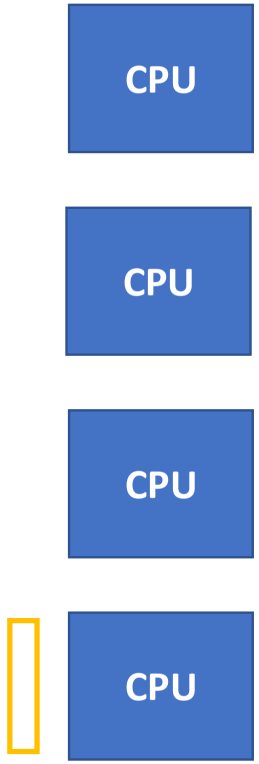


„Viacvláknový prístup“ (s time-slicing)

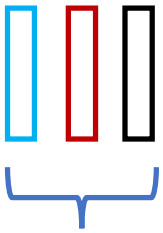
CPU spustí **vlákno #1** pre malý časový úsek, potom **vlákno #2**, potom znova **vlákno #1**, následne **vlákno #2** a tak ďalej...

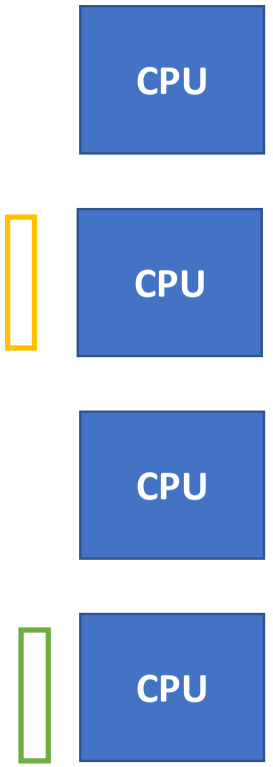
Viac vlákien vs. Paralelné algoritmy





Python prístup k paralelizmu. Ak je 1 vlákno priradené CPU, celý rad pripravených vlákien sa presúva do radu s čakajúcimi vláknami. Keď sa vlákno z CPU uvoľní, vlákna sa presunú späť do radu vlákien pripravených na spustenie. Dôsledok **GIL**.





Bežný prístup paralelizmu vlákien v jazykoch ako **Java, C++**

Global Interpreter Lock (GIL)

- **Global Interpreter Lock (GIL)** je mutex (alebo zámok), ktorý povolí len jednému vláknu držať kontrolu nad Python interpreterom v danom časovom úseku
- To znamená iba **jedno vlákno** môže byť spustené v určitom časovom úseku
- GIL zabraňuje viacvláknovým aplikáciam využiť všetky výhody viacprocesorových systémov
- V **Pythone nie je reálne paralelné programovanie** (spustenie programu)
- Ak chceme spustiť program na viacprocesorových systémoch paralelne, namiesto paralelných vlákien, **musíme** využiť **paralelné procesy**
- Procesy nie sú tak efektívne ako vlákna (vlákno je odľahčená verzia procesu)
- Pre daný proces môže existovať niekoľko vlákien
- Prepínanie medzi vláknami je už celkom nákladné
- Prepínanie medzi procesmi je ešte nákladnejšie
- Používanie procesov pre paralelné programovanie v Pythone nie je najlepšie riešenie
- Môžeme získať 2, 3-krát vyšší výkon aplikácie napísanej v Pythone, ale
- ak sa zameriame na jazyky C++ alebo Java, vieme dosiahnuť 20 – 50 násobný vyšší výkon ako v Pythone, pretože v týchto jazykoch neexistuje GIL, čo znamená, že vieme získať plnú výhodu multiprocesorových systémov za pomoci viacerých vlákien

Prečo bol GIL implementovaný

- **Python** existuje už od dôb, kedy operačné systémy nemali koncept vlákien
- Python bol navrhnutý tak, aby sa dal ľahko používať, a aby bol vývoj rýchlejší
- Postupom času čoraz viac vývojárov ho začalo používať a stal sa obľúbeným pre mnohé komunity
- **GIL umožňuje zvýšiť výkon programov** s jedným vláknom, pretože je jediným zámkom, ktorý nastavuje pravidlo, že vykonanie ľubovoľného Python bytecode vyžaduje získanie tohto zámku. Takto sa zabráni deadlockom (existuje vždy iba jeden zámok, ktorý je distribuovaný rôznym Python objektom)
- Jednovláknové programy sú v Pythone rýchlejšie ako viacvláknové, pretože účinnosťou GIL sa viacvláknová aplikácia stane v určitom čase jednovláknovou
- GIL nebol v Python verzii 3 odstránený z dôvodu:
  - Spomaleniu už existujúcich jednovláknových aplikácií
  - Aplikácie napísané v Pythone v3 by boli pomalšie ako aplikácie z Pythonu v2
  - Nie je jednoduché odstrániť GIL

na obídenie GIL, používať paralelizmus procesov alebo iný intepreter (Jython, IronPython, ..., kde nie je implementovaný GIL). Použitím paralelizmu vlákien, bez uvedených postupov, **nedosiahneme paralelné správanie aplikácie !