KATEDRA INFORMATIKY
UNIVERZITA PALACKÉHO V OLOMOUCI
www.inf.upol.cz

Jan Laštovička
https://www.inf.upol.cz/lide/jan-lastovicka
jan.lastovicka@upol.cz
17. listopadu 12, 771 46 Olomouc

Úvod do programovacích stylů ◇ poznámky k přednášce

12. Paralelní programování

verze z 11. prosince 2024

Příkaz

```
global variable
```

použitý na začátku těla funkce deklaruje, že proměnná *variable* je **globální**. Nastavení hodnoty proměnné *variable* vede k změně globální proměnné *variable*.

Například uvažujme program:

```
x = 0

def f():
    x = 1

def g():
    global x
    x = 1
```

Zavolání funkce `f` neovlivní hodnotu globální proměnné `x`:

```
>>> f()
>>> x
0
```

Naproti tomu zavolání funkce `g` vede ke změně globální proměnné `x`:

```
>>> g()
>>> x
1
```

Vezměme programy *process*1, ..., *process*n a program *globals*. Zápis:

globals		
*process*1	...	*process*n

nazveme **paralelním programem**. Programy *process*1, ..., *process*n se nazývají **procesy**. Program *globals* definuje globální proměnné, které procesy používají.

Například:

x = None	
x = 1	x = 2

Paralelní program

globals		
*process*1	...	*process*n

se vykoná tak, že se nejprve vykoná program ***globals*** a následně se vykoná libovolné proložení příkazů procesů ***process***1, ..., ***process***n. Konkrétní posloupnost provedených příkazů se nazývá **historie** paralelního programu.

Například vykonání:

x = None	
x = 1	x = 2

povede buď na historii:

```
x = None
x = 2
x = 1
```

nebo na historii:

```
x = None
x = 1
x = 2
```

Výsledek vykonání paralelního programu může být **nedeterministický**. Tím myslíme, že po skončení jeho vykonávání nemusíme vědět, jakou hodnotu má nějaká proměnná. Například u výše uvedeného programu nevíme, zda je po skončení programu hodnota proměnné `x` jedna, nebo dva.

K tisku v paralelním programu je potřeba používat funkci `safe_print`, která vytiskne zadané hodnoty stejně jako vestavěná funkce `print`, ale navíc zajistí, aby tisk byl atomický. Například:

`safe_print(12)`	`safe_print(34)`

Výraz se **vyhodnotí atomicky**, pokud

1. obsahuje nejvýše jednu proměnnou ***variable***, která může být změněna v jiném procesu,
2. a obsahuje ji nejvýše jednou.

Například v programu:

x = 0 y = 0 z = 0	
x = 1 z = 1	y = 1 safe_print(x + y) safe_print(x + x) safe_print(x + z)

se výraz `x + y` vyhodnotí atomicky, protože pouze proměnná `x` je měněna v jiném procesu a výraz ji obsahuje pouze jednou. Výraz `x + x` se nevyhodnotí atomicky. Výraz sice obsahuje jedinou proměnnou, která je měněna v jiném procesu, ale obsahuje ji dvakrát. Ani výraz `x + z` se atomicky nevyhodnotí, protože obsahuje dvě proměnné, které jsou měněny v jiném procesu.

Příkaz přiřazení ***variable*** `=` ***expression*** se **vykoná atomicky**, pokud

1. se výraz ***expression*** vyhodnotí atomicky a proměnná ***variable*** není čtena ani měněna v jiném procesu
2. nebo ***expression*** neobsahuje žádnou proměnnou, která je měněna v jiném procesu.

Například v programu:

x = 0 y = 0 z = 1 a = 0	
x = x + z a = x + x	y = x + 1 x = y + z

se příkaz `y = x + 1` vykoná atomicky, protože výraz `x + 1` se vyhodnotí atomicky a proměnná `y` není čtena ani měněna v jiném procesu. Příkaz `x = y + z` se také vykoná atomicky. Zde `y + z` neobsahuje žádnou proměnnou, která je měněna v jiném procesu. Příkaz `x = x + 1` se atomicky nevykoná vykoná. Výraz `x + 1` se sice vyhodnotí atomicky, ale `x` je čtena nebo měněna v jiném procesu a výraz `x + 1` obsahuje proměnnou, která je měněna v jiném procesu. Ani příkaz `a = x + x` se nevykoná atomicky, protože výraz `x + x` se nevyhodnotí atomicky.

Budeme požadovat, aby příkazy v procesech paralelního programu se vykonávaly atomicky. Například oba příkazy v následujícím programu jsou atomické.

x = None	
x = 1	x = 2

K práci s paralelním vykonáváním používáme knihovnu `co` (zkratka za *concurrent*) nelézající se v souboru `co.py`. Knihovna se importuje příkazem:

```
from co import *
```

Funkce z knihovny `co`:

```
co_call(function1, ..., functionn) => None
```

paralelně zavolá funkce *function*1, ..., *function*n bez parametrů a čeká až volání všech funkcí skončí.

Program:

globals		
*process*1	...	*process*n

můžeme zapsat kódem:

```
globals

def p1:
     process1

⋮

def pn:
     processn

co_call(p1, ..., pn)
```

kde v každé funkci deklarujeme všechny globální proměnné, které funkce mění.

Například:

x = None	
x = 1	x = 2

zapíšeme jako:

```
x = None

def p1():
    global x
    x = 1

def p2():
    global x
    x = 2

co_call(p1, p2)
```

Funkce `random_sleep` způsobí pozastavení vykonávání procesu na zadaný počet vteřin. Výchozí hodnota parametru je $0,01$. Pro zvýšení pravděpodobnosti některých historií přidáme náhodné čekání:

```
x = None

def p1():
    global x
    random_sleep()
    x = 1


def p2():
    global x
    random_sleep()
    x = 2

co_call(p1, p2)
```

Po vykonání programu nevíme, zda:

```
>>> x
1
```

nebo:

```
>>> x
2
```

Příkaz `x = x + 1` v následujícím programu se nevykoná atomicky:

`x = 0`	
`x = x + 1`	`x = x + 1`

Příkaz, který se nevykoná atomicky, musíme rozdělit na více příkazů, které se atomicky vykonají. Například příkazy z předchozího programu rozdělíme na:

`x = 0`	
`tmp1 = x + 1` `x = tmp1`	`tmp2 = x + 1` `x = tmp2`

Proměnné `tmp1` a `tmp2` jsou lokální. Jaké jsou všechny možné historie programu? Jaké jsou možné hodnoty proměnné `x` po skončení programu?

Možná historie programu:

```
x = 0
tmp1 = x + 1
tmp2 = x + 1
x = tmp1
x = tmp2
```

vedoucí na hodnotu jedna.

Protože jsou proměnné `tmp1` a `tmp2` lokální, můžeme je přejmenovat tak, aby se jmenovali stejně:

`x = 0`	
`tmp = x + 1` `x = tmp`	`tmp = x + 1` `x = tmp`

U historie pak ale musíme u každého příkazu uvést, který proces ho vykonal:

```
   x = 0
1  tmp = x + 1
2  tmp = x + 1
1  x = tmp
2  x = tmp
```

Pro testování zopakujeme v každém procesu inkrementaci stokrát:

```
x = 0

def p():
    global x
    for i in range(100):
        random_sleep()
        tmp = x + 1
        random_sleep()
        x = tmp

co_call(p, p)
print(x)
```

Jaká čísla může program vytisknout?

Uzamčením můžeme vynutit vykonání části kódu atomicky. **Zámek** je hodnota, která se vytváří voláním `make_lock()` funkce z knihovny `co`. Uzamčení bloku *`block`* zámkem *`lock`* provedeme následovně.

```
with lock:
    block
```

Přesněji pokud chceme část kódu *block* provést atomicky, musíme ho a všechny části kódu, které v jiných procesech čtou nebo mění proměnné v *block* uzamknout stejným zámkem.

Například v programu:

```
x = 0
lock = make_lock()
```

`with lock:` `    x = x + 1`	`with lock:` `    x = x + 1`

se příkazy `x = x + 1` vykonají atomicky.

Globální proměnnou můžeme využít k předání hodnoty mezi procesy. Například:

```
x = None
```

⋮
`x = 1`
	`while x == None:`
	`    pass`
	⋮

Pro pohodlnější komunikaci mezi procesy používáme **frontu**. Frontu vytvoříme instanciací třídy `Queue` z knihovny `co`. Frontě můžeme zaslat následující dvě zprávy. Zpráva

```
queue.put(value) => None
```

přidá hodnotu na konec fronty. Zpráva

```
queue.get() => value
```

odebere hodnotu ze začátku fronty a vrátí ji. V případě, že je fronta prázdná, čeká dokud fronta nebude prázdná. Obsluhy obou zpráv se vykonají atomicky.

Předání hodnoty pomocí fronty:

```
q = Queue()
```

⋮ `q.put(1)`	`value = q.get()` ⋮

Frontu můžeme samozřejmě použít k předání více hodnot:

```
q = Queue()
```

`q.put(1)`	`safe_print(q.get())`
`q.put(2)`	`safe_print(q.get())`

Zprávu `get` můžeme zaslat i s argumentem rovným hodnotě `False`:

```
queue.get(False) => value
```

v takovém případě se vyvolá chyba `QueueEmpty` v případě, že je fronta prázdná. Například:

`q = Queue()`	

`q.put(1)`	`result = None`
	`while result == None:`
	`  try:`
	`  result = q.get(False)`
	`  except QueueEmpty:`
	`  pass`
	⋮

Funkce

```
start_process(function, arg1, ..., argn) => process
```

knihovny `co` vytvoří a vrátí proces, ve kterém proběhne volání funkce *function* s argumenty *arg*1, ..., *arg*n.

Funkce knihovny `co`:

```
join_process(process) => None
```

čeká, než proces skončí. Na skončení všech procesů vytvořených program je potřeba čekat.

Například:

`x = None`	
`x = 1`	`x = 2`

je možné zapsat i takto:

```
x = None

def p():
    global x
    random_sleep()
    x = 1

process = start_process(p)
random_sleep()
x = 2
join_process(process)
```

Otázky a úkoly na cvičení

1. Určete možné historie následujícího paralelního programu.

`x = 1` `y = 1`	
`x = 2` `x = 3`	`y = x`

2. Určete možné hodnoty proměnné `y` po skončení vykonávání programu z předchozího úkolu.

3. Rozhodněte, zda se příkazy v následujících programech vykonají atomicky a v záporném případě příkazy rozložte na atomické.

(a)

`x = 1` `y = 1`	
`x = x + 1`	`y = y + 1`

(b)

`x = 1` `y = 2`	
`x = y`	`y = x`

(c)

`x = 0` `y = 0`	
`x = 1`	`y = x + x`

4. V programech z předchozího úkolu určete možné hodnoty sdílených proměnných.

5. V programech z třetího úkolu pomocí zámků zajistěte atomické vykonání příkazů, které by se jinak atomicky nevykonaly.

6. Spusťte paralelně dva procesy, které budou tisknout čísla od nuly po devítku.

7. Vytvořte dva procesy. První druhému postupně pošle čísla od jedné do desíti včetně. Druhý proces obdržená čísla sečte a vytiskne výsledek.

8. Upravte program z předchozího úkolu tak, aby druhý proces nevěděl, kolik čísel má obdržet.

9. Vytvořte tři procesy, kde první dva procesy postupně posílí čísla mezi jednou a devíti včetně třetímu procesu. Třetí proces obdržená čísla sčítá a nakonec vytiskne jejich součet.

10. Vytvořte tři procesy, kde první druhému postupně posílá čísla mezi jednou a devíti včetně. Druhý počítá druhé mocniny obdržených čísel a posílá je třetímu procesu, který přijatá čísla tiskne.

11. Vytvořte aplikaci, s dlouhým výpočtem v jiném procesu. Uživatele informujte z kolika procent je výpočet hotový.