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 5. prosince 2023

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 libovolně proloží příkazy 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 je nedeterministický. Například u výše uvedeného programu nevíme, zda je po skončení programu hodnota proměnné `x` jedna, nebo dva.

Výraz je z hlediska vyhodnocování **atomický**, pokud

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

Příkaz přiřazení ***variable*** `=` ***expression*** je z hlediska vykonávání **atomický**, pokud

1. výraz ***expression*** je z hlediska vyhodnocování atomický a proměnná ***variable*** není čtena ani měněna v jiném procesu
2. nebo ***expression*** neobsahuje žádnou proměnnou, která může být změněna v jiném procesu.

Příkazy v procesech v paralelním programu musí být atomické. Například oba příkazy v následujícím programu jsou atomické.

x = None	
x = 1	x = 2

K práci s paralelním vykonáváním používáme knihovnu `co` 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)
```

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 není atomický:

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

Převedeme jej na dva atomické příkazy:

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 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`

jsou příkazy `x = x + 1` atomické.

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) => value
```

odebere hodnotu ze začátku fronty a vrátí ji. V případě, že je fronta prázdná, čeká dokud fronta nebude prázdná.

Obsluhy obou zpráv jsou atomické.

Předání hodnoty pomocí fronty:

q = Queue()	
⋮	value = q.get()
q.put(1)	⋮

K tisku v paralelním programu je potřeba používat funkci `safe_print`, která vytiskne zadané hodnoty podobně jako vestavěná funkce `print`. Například:

q = Queue()	
q.put(1)	safe_print(q.get())
q.put(2)	safe_print(q.get())

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 vytvořených procesů 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)
```