

INE5410 – Programação Concorrente

Unidade III – Tecnologias para Programação Concorrente

3.2 - Concorrência em Linguagens Orientadas a Objetos

Prof. Frank Siqueira
frank.siqueira@ufsc.br



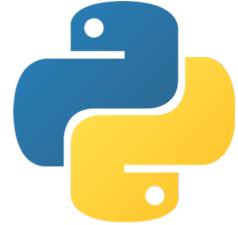
Conteúdo

- Concorrência em Linguagens OO
- Concorrência em Python
 - Threads
 - Processos
 - Executores
 - Controle de Concorrência
 - Locks
 - Semáforos
 - Mecanismos de Comunicação

Concorrência em Linguagens OO

- Grande parte das linguagens OO suporta concorrência nativamente em suas libs/APIs
- Em geral, há classes que representam Threads, Mutexes, Semáforos, etc.
 - Java: classe Thread; instrução synchronized; classes Lock, Semaphore, ... do pacote java.util.concurrent
 - Go: goroutines; Mutex, RWMutex, ... do pacote sync
 - C++: classes std::thread, std::lock, std::mutex, std::counting_semaphore, std::binary_semaphore, ...
 - C#: classes Thread, Monitor, Semaphore do namespace System.Threading; instrução lock(Object)
 - ...

Concorrência em Python



- Um novo processo do SO é criado sempre que executamos um programa Python
 - Inicialmente o processo terá uma thread (*main*)
 - Mais threads podem ser criadas pelo código do programa ao longo da execução
 - Apenas uma thread do processo executa por vez
- O *runtime* do Python limita a concorrência entre as threads do programa
 - Cada processo possui uma estrutura de sincronização chamada Global Interpreter Lock (GIL)
 - Somente a thread que tiver acesso a essa estrutura pode executar

Concorrência em Python

- O *Global Interpreter Lock (GIL)* impede que as threads de um processo Python executem simultaneamente em núcleos diferentes
 - Isso é feito por meio de um semáforo binário
 - Essa estratégia foi adotada devido à forma como o gerenciamento de memória do Python foi implementado
 - O coletor de lixo do Python executa em exclusão mútua com as threads do processo

Concorrência em Python

- O *Global Interpreter Lock (GIL)* é liberado quando:
 - A thread executa uma operação de I/O
 - Outra thread aguarda a execução
 - Indicado pela variável de condição *gil_drop_request*
 - A thread em execução verifica algumas condições (quantum de tempo, chamadas bloqueantes, etc.)
 - O interpretador Python coordena a troca da thread em execução

Concorrência em Python

- Por que usar threads em Python se não há paralelismo real entre elas?
 - Execução sequencial x concorrente (*multithread*)
 - Suponha que a thread T1 interage com o usuário enquanto T2 realiza cálculos complexos
 - Em uma execução sequencial, teríamos o seguinte:



- Com suporte a *multithreading* podemos reduzir a ociosidade e fazer melhor uso da CPU:



Threads

- Módulo `threading`
 - Contém classes necessárias para uso de threads

```
import threading
```

- Classe `Thread`
 - Representa uma thread em Python
 - Recebe um *target* no construtor que indica a função que será executada pela Thread

```
from threading import Thread
def codigo_thread():
    ...
thread1 = Thread(target=codigo_thread)
```

Threads

- O construtor da classe Thread

```
class Thread ( group=None, target=None,  
               name=None, *args=(),  
               **kwargs={}, daemon=None)
```

- **group**: usado para criação de grupos de threads
- **target**: função ou lambda executado pela thread
- **name**: nome da thread (por padrão, “Thread-N”)
- **args**: tupla de argumentos enviados à thread
- **kwargs**: dicionário de argumentos enviados à thread
- **daemon**: indica se a thread é um *daemon*; faz com que ela seja encerrada quando todas as threads não-*daemon* do programa terminarem a execução

Threads

- Métodos úteis da classe `threading.Thread`
 - `run()`: código executado pela thread (pode ser sobrescrito ou alterado com o parâmetro *target* do construtor)
 - `start()`: inicia a execução da thread
 - `is_alive()`: informa se a thread está viva (ou seja, já foi iniciada e ainda não terminou)
 - `join()`: aguarda que a thread termine de executar
- Atributos da classe `threading.Thread`
 - `daemon`: indica se a thread é um *daemon*
 - `name`: nome da thread

Threads

- Comportamento das Threads
 - O método `start()` inicia a execução da thread
 - O método `join()` aguarda o fim da thread

```
from threading import Thread
def codigo_thread():
    ...
thread1 = Thread(target=codigo_thread)
thread2 = Thread(target=codigo_thread)
thread1.start()      # inicia a thread1
thread2.start()      # inicia a thread2
...                  # threads em execução
thread1.join()       # aguarda fim da thread1
thread2.join()       # aguarda fim da thread2
```

Threads

- Exemplo: Programa que executa duas ações
 - interage com usuário (E/S)
 - realiza cálculo (CPU intensivo)

```
from threading import Thread

def le_nome():
    nome = input('Informe o seu nome: ')
    print('Olá', nome, '! Obrigado por testar o programa.')

def calcula():
    print('Cálculo iniciado...')
    [ (x**2) for x in range(10000000) ]
    print('Cálculo concluído.')
```

Threads

- Exemplo: execução sequencial

```
le_nome()  
calcula()
```

- Exemplo: execução concorrente

```
thread1 = Thread(target=le_nome)  
thread2 = Thread(target=calcula)  
thread1.start() # inicia a thread1  
thread2.start() # inicia a thread2  
thread1.join()  # aguarda fim da thread1  
thread2.join()  # aguarda fim da thread2
```

Threads

- Exemplo: execução sequencial x concorrente

```
from threading import Thread
def le_nome():
    nome = input("Informe o seu nome: ")
    print('Olá', nome, '! Obrigado por testar o programa.')
def calcula():
    print('Cálculo iniciado...')
    [(x**2) for x in range(10000000)]
    print('Cálculo concluído.')
```

Execução Sequencial do Exemplo

```
le_nome()
calcula()

Informe o seu nome: Frank
Olá Frank ! Obrigado por testar o programa.
Cálculo iniciado...
Cálculo concluído.
```

Execução Concorrente do Exemplo

```
thread1 = Thread(target=le_nome)
thread2 = Thread(target=calcula)
thread1.start()# inicia a thread1
thread2.start()# inicia a thread2
thread1.join()# aguarda fim da thread1
thread2.join()# aguarda fim da thread2

Cálculo iniciado...
Informe o seu nome: Frank
Olá Frank ! Obrigado por testar o programa.
Cálculo concluído.
```

Threads

- Exemplo: Criação de threads em um laço
 - Podemos usar a mesma função para criar várias threads que executam o mesmo código

```
t = []                      # lista de threads
for i in range(3):
    t.append(Thread(target=calcula))
    t[i].start()

for i in range(3):
    t[i].join()
```

Threads

- Exemplo: Criação de threads em um laço

```
▶ t = []
  for i in range(3):
    t.append(Thread(target=calcula))
    t[i].start()

  for i in range(3):
    t[i].join()
```

```
⇨ Cálculo iniciado...
Cálculo iniciado...
Cálculo iniciado...
Cálculo concluído.
Cálculo concluído.
Cálculo concluído.
```

Threads

- Exemplo: Passando argumentos às threads
 - Podemos passar argumentos à função target
 - Isso pode levar a comportamentos inesperados!

```
def calcula_exp(i):
    print('Cálculo iniciado para o valor', i)
    [ (x**i) for x in range(10000000) ]
    print('Cálculo concluído para o valor', i)

t = []
for i in range(3):
    t.append(Thread(target=calcula_exp(i)))
    t[i].start()

for i in range(3):
    t[i].join()
```

Threads

- Exemplo: Passando argumentos às threads

```
def calcula_exp(i):
    print('Cálculo iniciado para o valor', i)
    [(x**i) for x in range(10000000)]
    print('Cálculo concluído para o valor', i)

t = []
for i in range(3):
    t.append(Thread(target=calcula_exp(i)))
    t[i].start()

for i in range(3):
    t[i].join()
```

```
Cálculo iniciado para o valor 0
Cálculo concluído para o valor 0
Cálculo iniciado para o valor 1
Cálculo concluído para o valor 1
Cálculo iniciado para o valor 2
Cálculo concluído para o valor 2
```

- Conclusão: se passarmos o iterador do laço para as threads elas serão executadas sequencialmente!

Threads

- Exemplo: Passando argumentos às threads
 - Solução 1: passar os argumentos explicitamente no construtor da Thread

```
def calcula_exp(i):
    print('Cálculo iniciado para o valor', i)
    [(x**i) for x in range(10000000)]
    print('Cálculo concluído para o valor', i)

t = []
for i in range(3):
    t.append(Thread(target=calcula_exp, args(i,)))
    t[i].start()

for i in range(3):
    t[i].join()
```

Threads

- Exemplo: Passando argumentos às threads

```
t = []
for i in range(3):
    t.append(Thread(target=calcula_exp, args=(i,)))
    t[i].start()

for i in range(3):
    t[i].join()
```

```
Cálculo iniciado para o valor 0
Cálculo iniciado para o valor 1
Cálculo iniciado para o valor 2
Cálculo concluído para o valor 0
Cálculo concluído para o valor 1
Cálculo concluído para o valor 2
```

- Conclusão: se passarmos um argumento explícito na criação da Thread haverá concorrência na execução!

Threads

- Exemplo: Passando argumentos às threads
 - Solução 2: passar um lambda como target no lugar da função

```
def calcula_exp(i):
    print('Cálculo iniciado para o valor', i)
    [(x**i) for x in range(10000000)]
    print('Cálculo concluído para o valor', i)

t = []
for i in range(3):
    t.append(Thread(target=lambda: calcula_exp(i)))
    t[i].start()

for i in range(3):
    t[i].join()
```

Threads

- Exemplo: Passando argumentos às threads



```
t = []
for i in range(3):
    t.append(Thread(target=lambda: calcula_exp(i)))
    t[i].start()

for i in range(3):
    t[i].join()
```

```
Cálculo iniciado para o valor 0
Cálculo iniciado para o valor 1
Cálculo iniciado para o valor 2
Cálculo concluído para o valor 0
Cálculo concluído para o valor 1
Cálculo concluído para o valor 2
```

- Conclusão: o uso do lambda permitiu a execução das threads de forma concorrente!

Threads

- Exemplo: Criando Threads com Herança

```
from threading import Thread
from time import sleep

class MyThread(Thread):          # subclasse de Thread
    def __init__(self, name, sleep_time): # construtor recebe sleep_time
        self.sleep_time = sleep_time      # tempo que a thread irá dormir
        super().__init__(name=name)       # chama construtor da superclasse

    def run(self):                     # método executado pela thread
        print(self.name + ' iniciada...\n', end='')
        sleep(self.sleep_time)
        print(self.name + ' concluída.\n', end='')

t = []                           # lista com instâncias de MyThread
for i in range(3):
    t.append(MyThread(name='MyThread-' + str(i), sleep_time=i+3))
    t[i].start()
    if t[i].is_alive():            # verifica se a thread está viva
        print(t[i].name + ' está ativa.\n', end='')

for i in range(3):
    t[i].join()
    print(t[i].name + ' terminou.\n', end='')
```

Threads

- Exemplo: Criando Threads com Herança

```
▶ from threading import Thread
from time import sleep

class MyThread(Thread):          # subclasse de Thread
    def __init__(self, name, sleep_time): # construtor recebe sleep_time
        self.sleep_time = sleep_time      # tempo que a thread irá dormir
        super().__init__(name=name)       # chama construtor da superclasse

    def run(self):                    # método executado pela thread
        print(self.name + ' iniciada...\n', end='')
        sleep(self.sleep_time)
        print(self.name + ' concluída.\n', end='')

t = []                          # lista com instâncias de MyThread
for i in range(3):
    t.append(MyThread(name='MyThread-' + str(i), sleep_time=i+3))
    t[i].start()
    if t[i].is_alive():           # verifica se a thread está viva
        print(t[i].name + ' está ativa.\n', end='')

for i in range(3):
    t[i].join()
    print(t[i].name + ' terminou.\n', end='')

MyThread-0 iniciada...
MyThread-0 está ativa.
MyThread-1 iniciada...
MyThread-1 está ativa.
MyThread-2 iniciada...
MyThread-2 está ativa.
MyThread-0 concluída.
MyThread-0 terminou.
MyThread-1 concluída.
MyThread-1 terminou.
MyThread-2 concluída.
MyThread-2 terminou.
```

Threads

- Funções úteis do módulo `threading`:
 - `active_count()`: retorna o número de threads ativas no momento invocação
 - `current_thread()`: retorna o objeto Thread em execução naquele momento
 - `enumerate()`: retorna uma lista de todos os objetos Thread ativos (não inclui threads terminadas e que não tenham sido inicializadas)
 - `main_thread()`: retorna o objeto Thread correspondente à thread principal do processo

Processos

- No Python, somente uma thread de cada processo tem acesso ao processador
- Para que haja paralelismo real, precisamos criar vários processos independentes
 - Processos requerem a alocação de mais recursos em comparação com threads
 - Como processos possuem áreas de memória independentes, precisam interagir por meio de mecanismos de comunicação inter-processos

Processos

- Módulo `multiprocessing`
 - Permite criar processos e ter paralelismo real

- Classe `Process`

```
class Process ( group=None, target=None,  
                name=None, *args=(),  
                **kwargs={}, *, daemon=None)
```

- `group`: usado para criação de grupos de processos
- `target`: função ou lambda executado pelo processo
- `name`: nome do processo (*default*: “Process-N”)
- `args`: tupla de argumentos enviada ao processo
- `kwargs`: dicionário de args. enviado ao processo
- `daemon`: indica se o processo é um *daemon*

Processos

- Principais métodos da classe `Process`
 - `start()`: inicia a execução do processo
 - `join()`: aguarda o fim da execução do processo
 - `run()`: contém o código executado pelo processo
 - `is_alive()`: indica se o processo está em execução
 - `terminate()`: encerra a execução do processo
- Atributos da classe `Process`
 - `name`: nome atribuído ao processo
 - `pid`: identificador do processo

Processos

- Exemplo: criação de processos

```
from multiprocessing import Process
from time import sleep

def temporizador(proc, tempo):
    print('[%s] Iniciada contagem de %i segundos' % (proc, tempo))
    while tempo:
        print('[%s] Tempo restante: %i segundo(s)' % (proc, tempo))
        sleep(1)
        tempo-=1
    print('[%s] Tempo esgotado.' % proc)

if __name__ == '__main__': # caso for o programa principal
    proc1 = Process(target=temporizador, args=('Proc1', 10))
    proc2 = Process(target=temporizador, args=('Proc2', 5))
    proc1.start()
    proc2.start()
    proc1.join()
    proc2.join()
```

Processos

- Exemplo: criação de processos

```
▶ from multiprocessing import Process
from time import sleep

def temporizador(proc, tempo):
    print('[%s] Iniciada contagem de %i segundos' % (proc, tempo))
    while tempo:
        print('[%s] Tempo restante: %i segundo(s)' % (proc, tempo))
        sleep(1)
        tempo-=1
    print('[%s] Tempo esgotado.' % proc)

if __name__ == '__main__': # caso for o programa principal
    proc1 = Process(target=temporizador, args=('Proc1', 10))
    proc2 = Process(target=temporizador, args=('Proc2', 5))
    proc1.start()
    proc2.start()
    proc1.join()
    proc2.join()

[Proc1] Iniciando temporizador de 10 segundos
[Proc1] Tempo restante: 10 segundo(s)
[Proc2] Iniciando temporizador de 5 segundos
[Proc2] Tempo restante: 5 segundo(s)
[Proc1] Tempo restante: 9 segundo(s)
[Proc2] Tempo restante: 4 segundo(s)
[Proc1] Tempo restante: 8 segundo(s)
[Proc2] Tempo restante: 3 segundo(s)
[Proc1] Tempo restante: 7 segundo(s)
[Proc2] Tempo restante: 2 segundo(s)
[Proc1] Tempo restante: 6 segundo(s)
[Proc2] Tempo restante: 1 segundo(s)
[Proc1] Tempo restante: 5 segundo(s)
[Proc2] Tempo esgotado.
[Proc1] Tempo restante: 4 segundo(s)
[Proc1] Tempo restante: 3 segundo(s)
[Proc1] Tempo restante: 2 segundo(s)
[Proc1] Tempo restante: 1 segundo(s)
[Proc1] Tempo esgotado.
```

Processos

- Exemplo: cancelando um processo

```
from multiprocessing import Process
from time import sleep

def temporizador(proc, tempo):
    print('[%s] Iniciada contagem de %i segundos' % (proc, tempo))
    while tempo:
        print('[%s] Tempo restante: %i segundo(s)' % (proc, tempo))
        sleep(1)
        tempo-=1
    print('[%s] Tempo esgotado.' % proc)

if __name__ == '__main__': # caso for o programa principal
    proc = Process(target=temporizador, args=('Temporizador', 10))
    proc.start()

    resp = input('Digite qualquer tecla para finalizar a execução: ')
    if proc.is_alive():
        proc.terminate()
```

Processos

- Exemplo: cancelando um processo

```
▶ from multiprocessing import Process
  from time import sleep

  def temporizador(proc, tempo):
      print('[%s] Iniciada contagem de %i segundos' % (proc, tempo))
      while tempo:
          print('[%s] Tempo restante: %i segundo(s)' % (proc, tempo))
          sleep(1)
          tempo-=1
      print('[%s] Tempo esgotado.' % proc)

  if __name__ == '__main__': # caso for o programa principal
      proc = Process(target=temporizador, args=('Temporizador', 10))
      proc.start()

      resp = input('Digite qualquer tecla para finalizar a execução: ')

      if proc.is_alive():
          proc.terminate()
  
```

▶ [Temporizador] Iniciada contagem de 10 segundos
[Temporizador] Tempo restante: 10 segundo(s)
[Temporizador] Tempo restante: 9 segundo(s)
[Temporizador] Tempo restante: 8 segundo(s)
[Temporizador] Tempo restante: 7 segundo(s)
[Temporizador] Tempo restante: 6 segundo(s)
[Temporizador] Tempo restante: 5 segundo(s)
Digite qualquer tecla para finalizar a execução:

Executores

- O módulo `concurrent.futures` permite criar executores de tarefas concorrentes
 - A classe abstrata `Executor` permite gerenciar um grupo de tarefas trabalhadoras (*workers*) que executa as funções que lhe forem submetidas
 - Os resultados produzidos pelas funções podem ser obtidos por meio de objetos `Future`
 - Implementações da classe abstrata `Executor`:
 - `ThreadPoolExecutor`: um executor de threads
 - `ProcessPoolExecutor`: um executor de processos

Executores

- Principais métodos da classe `Executor`:
 - `submit(fn, /, *args, **kwargs)`: submete uma chamada a `fn(*args, **kwargs)` para ser executada; retorna um `Future` com o qual é possível obter o resultado da execução
 - `shutdown(wait=True, *, cancel_futures=False)`: encerra o executor; por padrão, aguarda o fim da execução das tarefas e não cancela os futures associados às tarefas executadas

Executores

- O construtor da classe `ThreadPoolExecutor`

```
class ThreadPoolExecutor(max_workers=None,  
                        thread_name_prefix='',  
                        initializer=None, initargs=())
```

- `max_workers`: define o número de threads a serem criadas; valor *default*: `min(32, os.cpu_count() + 4)`
- `thread_name_prefix`: prefixo que forma o nome das threads do pool, seguido de um número
- `initializer`: define a função executada pelas threads criadas; pode ser informado posteriormente
- `initargs`: tupla de argumentos passados para a função; por padrão é uma tupla vazia

Executores

- Exemplo: uso do ThreadPoolExecutor

```
from concurrent.futures import ThreadPoolExecutor  
  
executor = ThreadPoolExecutor(max_workers=2)  
for i in range(3):  
    executor.submit(calcula)  
  
executor.shutdown()
```

- Forma alternativa: uso da cláusula `with`
 - Faz o `shutdown()` automaticamente

```
from concurrent.futures import ThreadPoolExecutor  
  
with ThreadPoolExecutor(max_workers=2) as executor:  
    for i in range(3):  
        executor.submit(calcula)
```

Executores

- Exemplo: uso do ThreadPoolExecutor
 - Como o *pool* possui 2 *workers*, são executadas até 2 chamadas concorrentes à função calcula()



```
from concurrent.futures import ThreadPoolExecutor

executor = ThreadPoolExecutor(max_workers=2)
for i in range(3):
    executor.submit(calcula)
executor.shutdown()
```

Cálculo iniciado...
Cálculo iniciado...
Cálculo concluído.
Cálculo iniciado...
Cálculo concluído.
Cálculo concluído.



```
from concurrent.futures import ThreadPoolExecutor

with ThreadPoolExecutor(max_workers=2) as executor:
    for i in range(3):
        executor.submit(calcula)
```



Cálculo iniciado...
Cálculo iniciado...
Cálculo concluído.
Cálculo iniciado...
Cálculo concluído.
Cálculo concluído.

Executores

- Exemplo: obtendo o resultado de um Future
 - O método `result()` retorna o resultado da execução de uma tarefa já concluída

```
from math import factorial
from concurrent.futures import ThreadPoolExecutor

futs = []
with ThreadPoolExecutor(max_workers=5) as executor:
    for i in range(100,120):
        futs.append(executor.submit(factorial,i))

for j in range(20):
    try:
        result = futs[j].result()
    except Exception as exc:
        print('Exceção calculando o fatorial de', 100+j)
    else:
        print('Fatorial de', 100+j, '=', futs[j].result())
```

Executores

- Exemplo: obtendo o resultado com Future

```
▶ from math import factorial
  from concurrent.futures import ThreadPoolExecutor

  futs = []
  with ThreadPoolExecutor(max_workers=5) as executor:
    for i in range(100,120):
      futs.append(executor.submit(factorial,i))

  for j in range(20):
    try:
      result = futs[j].result()
    except Exception as exc:
      print('Exceção calculando o fatorial de', 100+j)
    else:
      print('Fatorial de', 100+j, '=', futs[j].result())

  □ Fatorial de 100 = 9332621544394415268169923885626670049071596826438162
  Fatorial de 101 = 9425947759838359420851623124482936749562312794702543
  Fatorial de 102 = 9614466715035126609268655586972595484553559050596594
  Fatorial de 103 = 9902900716486180407546715254581773349090165822114492
  Fatorial de 104 = 1029901674514562762384858386476504428305377245499907
  Fatorial de 105 = 1081396758240290900504101305800329649720646107774902
  Fatorial de 106 = 1146280563734708354534347384148349428703884874241396
  Fatorial de 107 = 1226520203196137939351751701038733888713156815438294
  Fatorial de 108 = 1324641819451828974499891837121832599810209360673358
  Fatorial de 109 = 1443859583202493582204882102462797533793128203133960
  Fatorial de 110 = 1588245541522742940425370312709077287172441023447356
  Fatorial de 111 = 1762952551090244663872161047107075788761409536026565
  Fatorial de 112 = 1974506857221074023536820372759924883412778680349753
  Fatorial de 113 = 2231192748659813646596607021218715118256439908795221
  Fatorial de 114 = 2543559733472187557120132004189335234812341496026552
  Fatorial de 115 = 2925093693493015690688151804817735520034192720430535
  Fatorial de 116 = 3393108684451898201198256093588573203239663555699420
  Fatorial de 117 = 3969937160808720895401959629498630647790406360168322
  Fatorial de 118 = 4684525849754290656574312362808384164392679504998620
  Fatorial de 119 = 5574585761207605881323431711741977155627288610948358
```

Executores

- Exemplo: iterando sobre uma lista de Future
 - A função `concurrent.futures.as_completed()` obtém os Future das tarefas concluídas

```
from concurrent.futures import as_completed

def calc_fat(n):
    return (n, factorial(n))

with ThreadPoolExecutor(max_workers=5) as exec:
    futs = {exec.submit(calc_fat, i): i for i in range(100,120)}

for fut in as_completed(futs):
    try:
        (n, fat) = fut.result()
    except Exception as exc:
        print('Exceção calculando o fatorial:', exc)
    else:
        print('Fatorial de', n, '=', fat)
```

Executores

- Exemplo: iterando sobre uma lista de Future

```
from concurrent.futures import as_completed

def calc_fat(n):
    return (n, factorial(n))

with ThreadPoolExecutor(max_workers=5) as exec:
    futs = {exec.submit(calc_fat, i): i for i in range(100,120)}

for fut in concurrent.futures.as_completed(futs):
    try:
        (n, fat) = fut.result()
    except Exception as exc:
        print('Exceção calculando o fatorial:', exc)
    else:
        print('Fatorial de', n, '=', fat)

Fatorial de 101 = 9425947759838359420851623124482936749562312794702543768
Fatorial de 114 = 2543559733472187557120132004189335234812341496026552301
Fatorial de 109 = 1443859583202493582204882102462797533793128203133960291
Fatorial de 113 = 223119274865981364659660702121871511825643990879522131
Fatorial de 103 = 990290071648618040754671525458177334909016582211449248
Fatorial de 100 = 933262154439441526816992388562667004907159682643816214
Fatorial de 119 = 557458576102760588132343171174197715562728861094835817
Fatorial de 111 = 1762952551090244663872161047107075788761409536026565516
Fatorial de 106 = 114628056373470835453434738414834942870388487424139673
Fatorial de 105 = 1081396758240290900504101305800329649720646107774902579
Fatorial de 107 = 122652020319613793935175170103873388871315681543829450
Fatorial de 108 = 132464181945182897449989183712183259981020936067335806
Fatorial de 118 = 468452584975429065657431236280838416439267950499862031
Fatorial de 115 = 292509369349301569068815180481773552003419272043053514
Fatorial de 116 = 339310868445189820119825609358857320323966355569942077
Fatorial de 102 = 961446671503512660926865558697259548455355905059659464
Fatorial de 110 = 158824554152274294042537031270907728717244102344735632
Fatorial de 112 = 197450685722107402353682037275992488341277868034975337
Fatorial de 117 = 3969937160808720895401959629498630647790406360168322301
Fatorial de 104 = 102990167451456276238485838647650442830537724549990721
```

Conteúdo

- Concorrência em Linguagens OO
- Concorrência em Python
 - Threads
 - Processos
 - Executores
 - **Controle de Concorrência**
 - Locks
 - Semáforos
 - Mecanismos de Comunicação

Controle de Concorrência

- A ausência de paralelismo real em aplicações Python não impede que ocorram condições de corrida no código
 - Podem ocorrer inconsistências nos dados acessados concurrentemente pelas Threads
 - A lógica do programa pode não funcionar conforme esperado

Controle de Concorrência

- Exemplo: condição de corrida

```
from threading import Thread

def gera_nf():
    global cont_nf
    prox_nf = cont_nf + 1 # calcula o numero da NF
    print('Gerando NF no. ' + str(prox_nf) + '\n', end='')
    cont_nf += 1 # atualiza o contador de NFs

cont_nf = 0
threads = [Thread(target=gera_nf) for i in range(10)]
for thread in threads:
    thread.start()
for thread in threads:
    thread.join()
```

Controle de Concorrência

- Exemplo: condição de corrida

```
▶ from threading import Thread

def gera_nf():
    global cont_nf
    prox_nf = cont_nf + 1    # calcula o numero da NF
    print('Gerando NF no. ' + str(prox_nf) + '\n', end='')
    cont_nf += 1              # atualiza o contador de NFs

cont_nf = 0
threads = [Thread(target=gera_nf) for i in range(10)]
for thread in threads:
    thread.start()
for thread in threads:
    thread.join()

Gerando NF no. 1
Gerando NF no. 2
Gerando NF no. 3
Gerando NF no. 3
Gerando NF no. 5
Gerando NF no. 6
Gerando NF no. 6
Gerando NF no. 8
Gerando NF no. 9
Gerando NF no. 10
```

Controle de Concorrência

- A classe `Lock` do módulo `threading` implementa um mutex/lock
 - Métodos da classe `Lock`:
 - `Lock()`: *factory* que cria uma instância de `Lock`
 - `acquire(blocking=True, timeout=None)`: trava o *lock* se estiver livre; caso contrário, bloqueia até que ele seja liberado (exceto se *blocking* = *False*); o tempo de bloqueio pode ser limitado definindo um *timeout*
 - `release()`: libera o *lock*; se ele não estiver bloqueado, gera a exceção `RuntimeError`; não precisa ser chamado pela mesma thread que fez o `acquire()`
 - `locked()`: retorna um booleano indicando se o *lock* está ou não bloqueado

Controle de Concorrência

- Exemplo: uso do Lock

```
from threading import Thread, Lock

def gera_nf_com_lock():
    global cont_nf
    lock.acquire()
    prox_nf = cont_nf + 1 # calcula o numero da NF
    print('Gerando NF no. ' + str(prox_nf) + '\n', end='')
    cont_nf += 1 # atualiza o contador de NFs
    lock.release()

cont_nf = 0
lock = Lock()
threads = [Thread(target=gera_nf_com_lock) for i in range(10)]
for thread in threads:
    thread.start()
for thread in threads:
    thread.join()
```

Controle de Concorrência

- Exemplo: uso do Lock

```
from threading import Thread, Lock

def gera_nf_com_lock():
    global cont_nf
    lock.acquire()
    prox_nf = cont_nf + 1    # calcula o numero da NF
    print('Gerando NF no. ' + str(prox_nf) + '\n', end='')
    cont_nf += 1              # atualiza o contador de NFs
    lock.release()

cont_nf = 0
lock = Lock()
threads = [Thread(target=gera_nf_com_lock) for i in range(10)]
for thread in threads:
    thread.start()
for thread in threads:
    thread.join()
```

```
Gerando NF no. 1
Gerando NF no. 2
Gerando NF no. 3
Gerando NF no. 4
Gerando NF no. 5
Gerando NF no. 6
Gerando NF no. 7
Gerando NF no. 8
Gerando NF no. 9
Gerando NF no. 10
```

Controle de Concorrência

- Exemplo: uso do **Lock** com **with**
 - O **Lock** é travado automaticamente ao entrar no bloco de código e liberado no final

```
from threading import Thread, Lock

def gera_nf_com_with():
    global cont_nf
    with lock:
        prox_nf = cont_nf + 1 # calcula o numero da NF
        print('Gerando NF no. ' + str(prox_nf) + '\n', end='')
        cont_nf += 1 # atualiza o contador de NFs
```

Controle de Concorrência

- A classe `RLock` do módulo `threading` implementa um lock/mutex reentrante
 - A thread que obtém o `Lock` pode fazer N chamadas a `acquire()`
 - Para liberá-lo, deve chamar `release()` N vezes

```
from threading import RLock

rlock = RLock()
...
rlock.acquire()      # 1o acquire
...
rlock.acquire()      # 2o acquire
...
rlock.release()      # 1o release; ainda falta 1
...
rlock.release()      # 2o release; lock foi liberado
```

Controle de Concorrência

- A classe **Condition** permite definir condições para execução de threads
 - Uma **Condition** é sempre associada a um **Lock**
 - É possível associar várias condições ao mesmo *lock*
 - Threads podem aguardar que uma condição seja satisfeita para prosseguir com a sua execução
 - Threads que aguardam uma condição devem ser notificadas quando esta condição for alterada
 - Condições são úteis para evitar espera ocupada

Controle de Concorrência

- Métodos da classe `Condition`:
 - `Condition(lock=None)`: cria uma condição e a associa a um objeto `Lock` (cria um novo *lock* se não informado)
 - `acquire()`: trava o *lock* associado à condição
 - `release()`: libera o *lock* associado à condição
 - `wait(timeout=None)`: bloqueia a thread até que a condição seja satisfeita ou o *timeout* se esgote
 - `notify(n=1)`: libera *n* threads que aguardam pela condição para que prossigam com a execução
 - `notify_all()`: libera a execução de todas as threads que aguardam pela condição

Controle de Concorrência

- Exemplo: produtor-consumidor

```
from time import sleep
from random import randint
from threading import Thread, Lock, Condition

def produtor():
    ...

def consumidor():
    ...

buffer = []
tam_buffer = 3
lock = Lock()
lugar_no_buffer = Condition(lock)
item_no_buffer = Condition(lock)
produtor = Thread(target=produtor)
consumidor = Thread(target=consumidor)
produtor.start()
consumidor.start()
produtor.join()
consumidor.join()
```

Controle de Concorrência

- Exemplo: produtor-consumidor (cont.)

```
def produtor():
    global buffer
    for i in range(10):
        sleep(randint(0,2))           # fica um tempo produzindo...
        item = 'item ' + str(i)
        with lock:
            if len(buffer) == tam_buffer:
                print('>>> Buffer cheio. Produtor irá aguardar.')
                lugar_no_buffer.wait()    # aguarda que haja um lugar no buffer
        buffer.append(item)
        print('Produzido %s (há %i itens no buffer)' % (item,len(buffer)))
        item_no_buffer.notify()

def consumidor():
    global buffer
    for i in range(10):
        with lock:
            if len(buffer) == 0:
                print('>>> Buffer vazio. Consumidor irá aguardar.')
                item_no_buffer.wait()    # aguarda que haja um item para consumir
            item = buffer.pop(0)
            print('Consumido %s (há %i itens no buffer)' % (item,len(buffer)))
            lugar_no_buffer.notify()
        sleep(randint(0,2))           # fica um tempo consumindo...
```

Controle de Concorrência

- Exemplo: produtor-consumidor

```
from time import sleep
from random import randint
from threading import Thread, Lock, Condition

def produtor():
    global buffer
    for i in range(10):
        sleep(randint(0,2))           # fica um tempo produzindo...
        item = 'item ' + str(i)
        with lock:
            if len(buffer) == tam_buffer:
                print('>>> Buffer cheio. Produtor irá aguardar.')
                lugar_no_buffer.wait()   # aguarda que haja lugar no buffer
            buffer.append(item)
            print('Produzido %s (há %i itens no buffer)' % (item,len(buffer)))
            item_no_buffer.notify()

def consumidor():
    global buffer
    for i in range(10):
        with lock:
            if len(buffer) == 0:
                print('>>> Buffer vazio. Consumidor irá aguardar.')
                item_no_buffer.wait()   # aguarda que haja um item para consumir
            item = buffer.pop(0)
            print('Consumido %s (há %i itens no buffer)' % (item,len(buffer)))
            lugar_no_buffer.notify()
            sleep(randint(0,2))       # fica um tempo consumindo...

buffer = []
tam_buffer = 3
lock = Lock()
lugar_no_buffer = Condition(lock)
item_no_buffer = Condition(lock)
produtor = Thread(target=produtor)
consumidor = Thread(target=consumidor)
produtor.start()
consumidor.start()
produtor.join()
consumidor.join()
```

```
Produzido item 0 (há 1 itens no buffer)
Produzido item 1 (há 2 itens no buffer)
Consumido item 0 (há 1 itens no buffer)
Produzido item 2 (há 2 itens no buffer)
Consumido item 1 (há 1 itens no buffer)
Produzido item 3 (há 2 itens no buffer)
Consumido item 2 (há 1 itens no buffer)
Produzido item 4 (há 2 itens no buffer)
Produzido item 5 (há 3 itens no buffer)
Consumido item 3 (há 2 itens no buffer)
Consumido item 4 (há 1 itens no buffer)
Consumido item 5 (há 0 itens no buffer)
>>> Buffer vazio. Consumidor irá aguardar.
Produzido item 6 (há 1 itens no buffer)
Produzido item 7 (há 2 itens no buffer)
Produzido item 8 (há 3 itens no buffer)
>>> Buffer cheio. Produtor irá aguardar.
Consumido item 6 (há 2 itens no buffer)
Consumido item 7 (há 1 itens no buffer)
Produzido item 9 (há 2 itens no buffer)
Consumido item 8 (há 1 itens no buffer)
Consumido item 9 (há 0 itens no buffer)
```

Controle de Concorrência

- Semáforo
 - Implementado pela classe `threading.Semaphore`
 - Métodos:
 - `Semaphore(value=1)`: constrói um semáforo e inicia o contador com o valor especificado (valor *default* 1)
 - `acquire(blocking=True, timeout=None)`: decrementa o contador se > 0 ; caso contrário, bloqueia (exceto se *blocking = False*) até que ocorra um `release()` ou o *timeout* esgote
 - `release(n=1)`: incrementa o contador do semáforo com o valor *n* (valor *default* 1)

Controle de Concorrência

- Exemplo: uso da classe Semaphore

```
from time import sleep
from threading import Thread, Semaphore

def usa_licenca(i):
    sem.acquire()
    print('Thread %i está usando uma licença\n' % i, end=' ')
    sleep(2) # usa a licença durante um tempo...
    print('Thread %i acabou de usar uma licença\n' % i, end=' ')
    sem.release()

num_licencias = 2
sem = Semaphore(num_licencias)
threads = [Thread(target=usa_licenca, args=(i,)) for i in range(10)]
for thread in threads:
    thread.start()
for thread in threads:
    thread.join()
```

Controle de Concorrência

- Exemplo: uso da classe Semaphore

```
▶ from time import sleep
from threading import Thread, Semaphore

def usa_licenca(i):
    sem.acquire()
    print('Thread %i está usando uma licença\n' % i, end=' ')
    sleep(2)    # usa a licença durante um tempo...
    print('Thread %i acabou de usar uma licença\n' % i, end=' ')
    sem.release()

num_licencias = 2
sem = threading.Semaphore(num_licencias)
threads = [Thread(target=usa_licenca,args=(i,)) for i in range(10)]
for thread in threads:
    thread.start()
for thread in threads:
    thread.join()
```

↳ Thread 0 está usando uma licença
Thread 1 está usando uma licença
Thread 0 acabou de usar uma licença
Thread 1 acabou de usar uma licença
Thread 2 está usando uma licença
Thread 3 está usando uma licença
Thread 2 acabou de usar uma licença
Thread 3 acabou de usar uma licença
Thread 4 está usando uma licença
Thread 5 está usando uma licença
Thread 4 acabou de usar uma licença
Thread 5 acabou de usar uma licença
Thread 7 está usando uma licença
Thread 6 está usando uma licença
Thread 7 acabou de usar uma licença
Thread 6 acabou de usar uma licença
Thread 8 está usando uma licença
Thread 9 está usando uma licença
Thread 9 acabou de usar uma licença
Thread 8 acabou de usar uma licença

Controle de Concorrência

- Classe `threading.BoundedSemaphore`
 - Implementa um semáforo cujo contador não pode ultrapassar o valor inicial especificado na sua criação
 - As assinaturas dos métodos são idênticas às da classe `Semaphore`
 - Caso ocorra uma tentativa de incrementar o contador acima do valor limite, é lançada a exceção `ValueError`

Mecanismos de Comunicação

- Threads de um mesmo processo podem trocar dados por meio de variáveis globais
 - Isso é possível porque elas compartilham o mesmo espaço de endereçamento do processo
 - Com isso, a troca de dados entre elas é mais simples e rápida
- Já os processos precisam usar algum mecanismo para comunicação
 - Isso faz com que as trocas de dados entre processos sejam mais lentas e caras

Mecanismos de Comunicação

- Existem diversas formas de estabelecer a comunicação entre processos em Python
- Comunicação entre processos rodando na mesma máquina pode ser feita com:
 - Canais de comunicação
 - Filas de mensagens
 - Memória compartilhada
- Para processos em máquinas diferentes, é necessário usar algum mecanismo de comunicação remota
 - Sockets TCP ou UDP, Thrift, gRPC, middlewares de mensagens, web services, etc.

Mecanismos de Comunicação

- Canais de Comunicação



Mecanismos de Comunicação

- A classe `Pipe` do módulo `multiprocessing` representa um canal de comunicação
 - `Pipe(duplex=True)` cria um canal de comunicação bidirecional (ou unidirecional, se `duplex=False`)
 - São retornados dois objetos `Connection`, `conn1` e `conn2`, que representam as extremidades do pipe
 - Se o canal for unidirecional, `conn1` só pode receber dados e `conn2` só pode enviar
 - Dados são enviados com `Connection.send(Object)`
 - O método `Connection.recv()` aguarda até que algum dado seja enviado e retorna o que for recebido

Mecanismos de Comunicação

- Exemplo: comunicação usando Pipe

```
from multiprocessing import Process, Pipe

def calc(conn):
    (op, i, j) = conn.recv()
    if op == 'soma':
        print('Executando operação:', op)
        conn.send(i + j)
    else:
        conn.send('Operação desconhecida.')
    conn.close()

if __name__ == '__main__':
    conn1, conn2 = Pipe(duplex=True)
    p = Process(target=calc, args=(conn2,))
    p.start()
    conn1.send(['soma', 100, 200])
    res = conn1.recv()
    print('Resultado:', res)
    p.join()
```

Mecanismos de Comunicação

- Exemplo: comunicação usando Pipe

```
▶ from multiprocessing import Process, Pipe

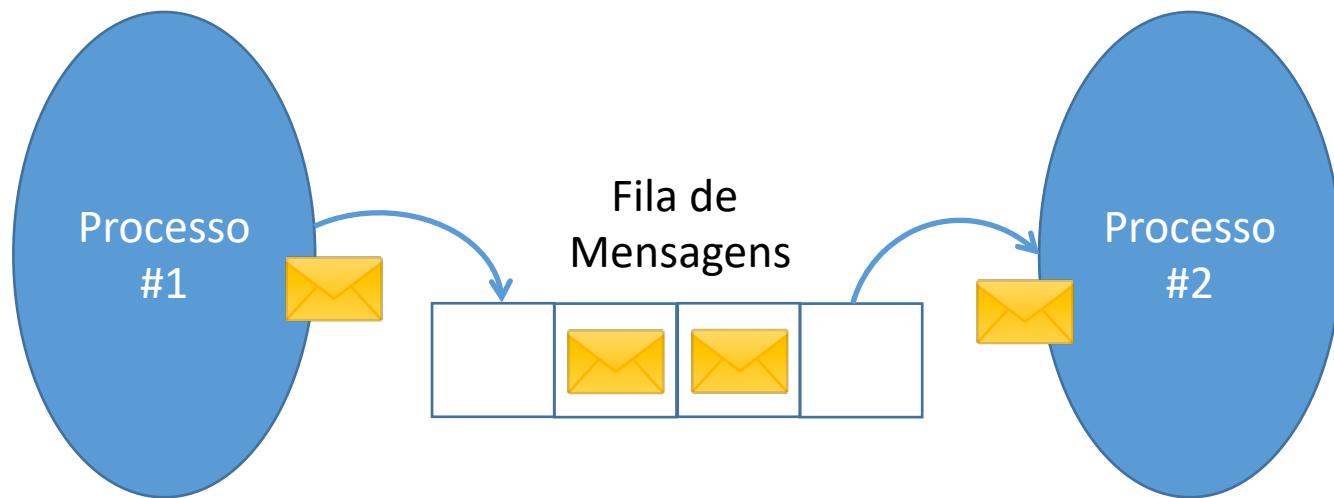
def calc(conn):
    (op, i, j) = conn.recv()
    if op == 'soma':
        print('Executando operação:', op)
        conn.send(i + j)
    else:
        conn.send('Operação desconhecida.')
    conn.close()

if __name__ == '__main__':
    conn1, conn2 = Pipe(duplex=True)
    p = Process(target=calc, args=(conn2,))
    p.start()
    conn1.send(['soma', 100, 200])
    res = conn1.recv()
    print('Resultado:', res)
    p.join()
```

```
Executando operação: soma
Resultado: 300
```

Mecanismos de Comunicação

- Filas de Mensagens



Mecanismos de Comunicação

- A classe `Queue` do módulo `multiprocessing` implementa uma fila de mensagens
 - A fila representada pelo objeto `Queue` pode ser compartilhada entre processos
 - A fila utiliza uma thread, um pipe e alguns locks/semáforos na sua implementação
 - O construtor `Queue([maxsize])` cria uma fila cujo tamanho pode ser limitado por `maxsize`
 - A fila é encerrada chamando `Queue.close()` seguido de `Queue.join_thread()` que encerra a thread usada na implementação

Mecanismos de Comunicação

- Principais métodos da classe `Queue`:
 - `put(obj, block=True, timeout=None)`: coloca a “mensagem” `obj` na fila; se a fila estiver cheia, bloqueia por padrão até que haja lugar na fila ou que o *timeout* informado esgote
 - `get(block=True, timeout=None)`: retira uma mensagem da fila e a retorna; se a fila estiver vazia, bloqueia por padrão até que chegue uma mensagem ou o *timeout* informado esgote
 - `qsize()`: informa o número de mensagens na fila
 - `empty()`: indica se a fila está vazia
 - `full()`: informa se a fila está cheia

Mecanismos de Comunicação

- Exemplo: uso da fila de mensagens

```
from multiprocessing import Process, Queue

def consumidor(fila):
    while fila.empty() == False:
        msg = fila.get()
        print('Mensagem recebida:', msg)

if __name__ == '__main__':
    fila = Queue()
    cons = Process(target=consumidor, args=(fila,))
    for i in range (1,10):
        msg = 'Teste ' + str(i)
        print('Enviando mensagem: %s' % msg)
        fila.put(msg)
    cons.start()
    cons.join()
    fila.close()
    fila.join_thread()
```

Mecanismos de Comunicação

- Exemplo: uso da fila de mensagens

```
▶ from multiprocessing import Process, Queue

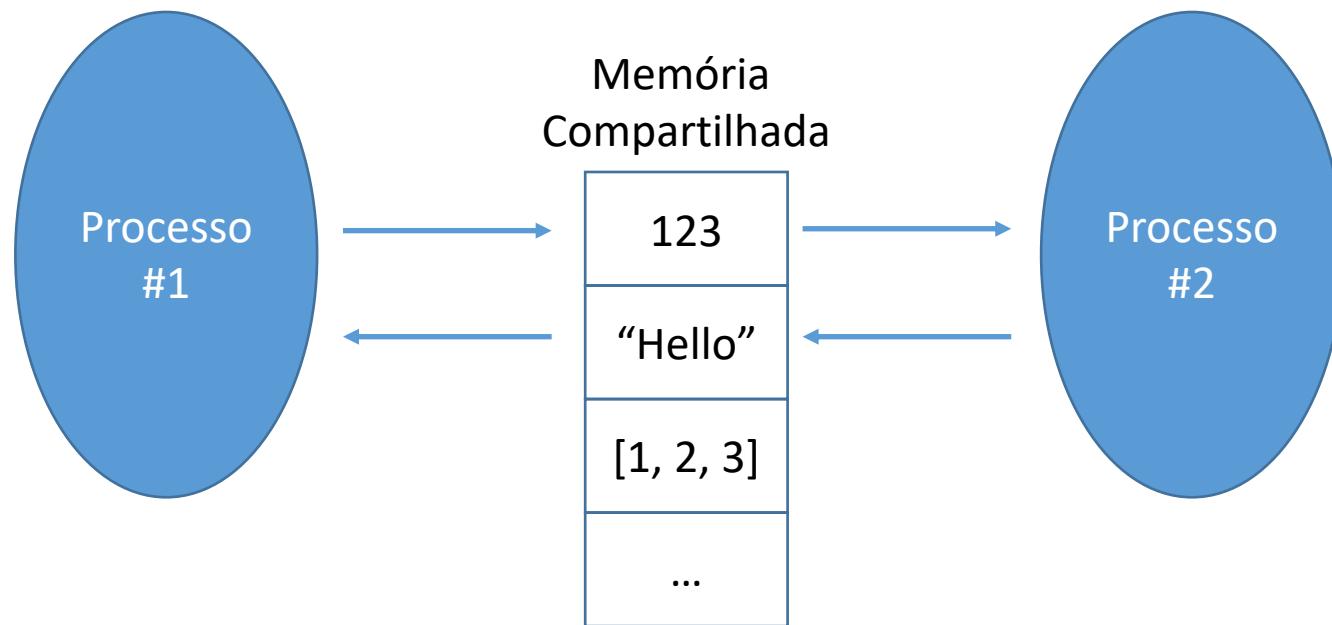
def consumidor(fila):
    while fila.empty() == False:
        msg = fila.get()
        print('Mensagem recebida:', msg)

if __name__ == '__main__':
    fila = Queue()
    cons = Process(target=consumidor, args=(fila,))
    for i in range (1,10):
        msg = 'Teste ' + str(i)
        print('Enviando mensagem: %s' % msg)
        fila.put(msg)
    cons.start()
    cons.join()
    fila.close()
    fila.join_thread()
```

```
⇨ Enviando mensagem: Teste 1
Enviando mensagem: Teste 2
Enviando mensagem: Teste 3
Enviando mensagem: Teste 4
Enviando mensagem: Teste 5
Enviando mensagem: Teste 6
Enviando mensagem: Teste 7
Enviando mensagem: Teste 8
Enviando mensagem: Teste 9
Mensagem recebida: Teste 1
Mensagem recebida: Teste 2
Mensagem recebida: Teste 3
Mensagem recebida: Teste 4
Mensagem recebida: Teste 5
Mensagem recebida: Teste 6
Mensagem recebida: Teste 7
Mensagem recebida: Teste 8
Mensagem recebida: Teste 9
```

Mecanismos de Comunicação

- Memória Compartilhada



Mecanismos de Comunicação

- As classes `Value` e `Array` do módulo `multiprocessing` nos permitem compartilhar memória entre processos
 - `Value(type, *args, lock=True)` permite criar um objeto do tipo `type`; os `args` são usados para inicialização, se necessário; um `lock` pode ser usado para evitar acesso concorrente
 - `Array(type, size_or_initializer, *, lock=True)` cria um array de elementos do tipo `type`, que terá o tamanho ou será inicializado conforme indicado por `size_or_initializer`, e que pode ter um `lock` para evitar acesso concorrente

Mecanismos de Comunicação

- Exemplo: uso de memória compartilhada

```
from multiprocessing import Process, Value, Array

def double(v, a):
    v.value = v.value * 2
    for i in range(len(a)):
        a[i] = a[i] * 2

if __name__ == '__main__':
    val = Value('d', 1.99)
    arr = Array('i', range(5))
    print('Valor Inicial:', val.value)
    print('Array Inicial:', arr[:])
    p = Process(target=double, args=(val, arr))
    p.start()
    p.join()
    print('Valor Final:', val.value)
    print('Array Final:', arr[:])
```

Mecanismos de Comunicação

- Exemplo: uso de memória compartilhada

```
▶ from multiprocessing import Process, Value, Array

def double(v, a):
    v.value = v.value * 2
    for i in range(len(a)):
        a[i] = a[i] * 2

if __name__ == '__main__':
    val = Value('d', 1.99)
    arr = Array('i', range(5))
    print('Valor Inicial:', val.value)
    print('Array Inicial:', arr[:])
    p = Process(target=double, args=(val, arr))
    p.start()
    p.join()
    print('Valor Final:', val.value)
    print('Array Final:', arr[:])
```

```
Valor Inicial: 1.99
Array Inicial: [0, 1, 2, 3, 4]
Valor Final: 3.98
Array Final: [0, 2, 4, 6, 8]
```

Referências

- Nguyen, Quan. Mastering Concurrency in Python: Create faster programs using concurrency, asynchronous, multithreading, and parallel programming. Packt Publishing; 1st ed., 2018.
- Mendizabal, Odorico. Material de INE5410 – Programação Concorrente – Semestre 2020/1.
- Python 3.9.0 Documentation:
<https://docs.python.org/3/library/>

INE5410 – Programação Concorrente

Unidade III – Tecnologias para Programação Concorrente

3.2 - Concorrência em Linguagens Orientadas a Objetos

Prof. Frank Siqueira
frank.siqueira@ufsc.br

