



## **Guião de aula Sincronização de processos**

---

### **1. Introdução**

Na última aula estudámos mecanismos que permitem comunicação entre processos: a memória partilhada e a troca de mensagens. Uma das conclusões dessa aula foi a de que a memória partilhada tinha algumas vantagens, em particular a rapidez e facilidade de acesso ao espaço partilhado, mas tinha uma desvantagem importante: a necessidade de implementar mecanismos de sincronização para acesso a essa zona de memória. De facto, o acesso concorrente à memória partilhada, com vários processos a acederem de forma assíncrona às variáveis aí armazenadas, pode criar incoerências nos dados. Nesta aula vamos estudar dois mecanismos de sincronização que nos permitem evitar estes problemas: os *mutexes* (ou trincos) e os semáforos.

#### **1.1. Motivação**

Comecemos por dar um exemplo de um programa cujo resultado não é exatamente o que se pretende devido à existência de acesso não sincronizado a uma região de memória partilhada. Neste programa são criados 500 processos, e cada um deles vai incrementar a variável partilhada uma única vez. Assim, seria de esperar que no final a variável partilhada armazenasse o número 500. Mas não é isso que acontece, como vamos poder observar. O código deste programa, `syncProb.py`, é apresentado de seguida.

```
from multiprocessing import Process, Value

partilhada = Value("i", 0)

def incrementa():
    partilhada.value += 1

processos=[]

for i in range(500):
    NewP = Process(target=incrementa)
    processos.append(NewP)
    NewP.start()
for p in processos:
    p.join()

print (partilhada.value)
```

O resultado de várias execuções deste programa é apresentado de seguida.

```
$ python syncProb.py
495
$ python syncProb.py
500
$ python syncProb.py
497
```

Como se pode verificar, o valor final apresentado no ecrã é variável, e não 500, como seria, à partida, esperado. A justificação é o facto de os 500 processos acederem à variável partilhada de forma concorrente. Para o programa funcionar devidamente, seria necessário que apenas um processo de cada vez pudesse manipular a variável partilhada. Ou seja, os processos deveriam estar sincronizados de alguma forma quando estão a aceder a essa sua secção crítica. Quando um processo está a executar a secção crítica, nenhum outro processo a deve aceder. Como se pode ver no código acima, no exemplo apresentado (assim como nos restantes que serão apresentados neste fascículo), a secção crítica é a que está a **vermelho**. É nesta secção que se manipula a variável partilhada.

## 1.2. *Mutexes*

A forma mais simples de resolver o problema da secção crítica apresentado anteriormente é através do uso de um *mutex* (ou *lock*). Um *mutex* (**mutual exclusion**) assegura exclusão mútua no acesso a uma região crítica. Esta propriedade garante que, se um processo está a executar na secção crítica, mais nenhum processo pode estar a executar nessa secção.

O módulo `multiprocessing` do Python oferece esta ferramenta de sincronização através da função `Lock()`. Quando queremos proteger o acesso a uma região crítica agarramos o (fazemos o `acquire` do) *lock mutex*:

```
Lock.acquire()
```

Quando saímos da região crítica, libertamos o (fazemos o `release` do) *mutex*:

```
Lock.release()
```

O código apresentado a seguir, `syncProbMutex.py`, resolve o problema que tínhamos anteriormente.

```
from multiprocessing import Process, Value, Lock

partilhada = Value("i", 0)
mutex = Lock()

def incrementa():
    mutex.acquire()
    partilhada.value += 1
    mutex.release()

processos=[]

for i in range(500):
    NewP = Process(target=incrementa)
    processos.append(NewP)
    NewP.start()
for p in processos:
    p.join()

print (partilhada.value)
```

O resultado de várias execuções deste programa é apresentado de seguida.

```
$ python syncProb.py
500
$ python syncProb.py
500
$ python syncProb.py
500
```

Como se pode verificar, o resultado já é o esperado, tendo sido resolvido o problema de sincronização.

### 1.3. Semáforos

O segundo mecanismo de sincronização que vamos estudar nesta aula é o semáforo. Um semáforo é uma ferramenta mais sofisticada do que um *mutex*. Além de permitir garantir exclusão mútua no acesso a um recurso partilhado, tal como o *mutex*, o semáforo pode resolver outros problemas de sincronização entre processos.

O semáforo é uma variável inteira que, tal como o *mutex*, só pode ser acedida através de duas operações, `acquire()` e `release()` (considerando a API do módulo `multiprocessing` do Python). A operação `acquire()` testa se o semáforo tem o valor zero. Se tiver, bloqueia. Se não tiver, decrementa o valor do semáforo e prossegue para a secção crítica. A função `release()` incrementa o valor do semáforo.

O semáforo permite assim controlar o acesso a um (ou vários) recurso(s) partilhado(s). Sempre que um processo quiser aceder a um recurso partilhado, executa a operação `acquire()`. Se o recurso estiver disponível, então o semáforo tem um valor positivo. O valor do semáforo é decrementado, e o processo pode então aceder ao recurso. Quando o semáforo tiver o valor zero, então isso significa que o recurso está indisponível. Nesse momento o processo bloqueia até que o semáforo passe a ter um valor positivo. Isso acontece quando um processo deixa de utilizar o recurso partilhado. Nesse momento o processo executa a operação `release()`, incrementando o semáforo. O processo que está à espera que o recurso fique disponível vai ser notificado de que o semáforo já tem um valor positivo, sai do estado de bloqueio, decrementa o semáforo e executa a sua secção crítica.

Os semáforos podem ser inicializados com qualquer número inteiro positivo (incluindo o zero). Esta possibilidade de inicialização é o que garante flexibilidade a esta ferramenta e o que a torna distinta do *mutex*:

- Se inicializado a 1, o semáforo pode servir para oferecer exclusão mútua (funcionando como um *mutex*).
- Inicializado com um número inteiro  $n$  maior do que 1 permite-se controlar o acesso a  $n$  instâncias de um recurso.
- E, se inicializado a zero, permite outras formas de sincronização entre processos (por exemplo, garantir que uma dada secção de código Y só corre depois de uma outra secção X).

Vamos agora dar um exemplo de como a utilização de semáforos nos permite resolver o problema apresentado no início deste guião. Para tal, vamos alterar o programa principal de acordo com o programa `syncProbSem.py`, apresentado a seguir (as alterações são apresentadas a vermelho).

```

from multiprocessing import Process, Value, Semaphore

partilhada = Value("i",0)
sem = Semaphore(1)

def incrementa():
    sem.acquire()
    partilhada.value += 1
    sem.release()

processos=[]

for i in range(500):
    NewP = Process(target=incrementa)
    processos.append(NewP)
    NewP.start()
for p in processos:
    p.join()

print (partilhada.value)

```

Este semáforo é inicializado com o valor 1 (já que só há um recurso a ser partilhado, funcionando assim como um *mutex*).

O resultado da execução deste programa é apresentado a seguir.

```

$ python syncProb.py
500
$ python syncProb.py
500
$ python syncProb.py
500

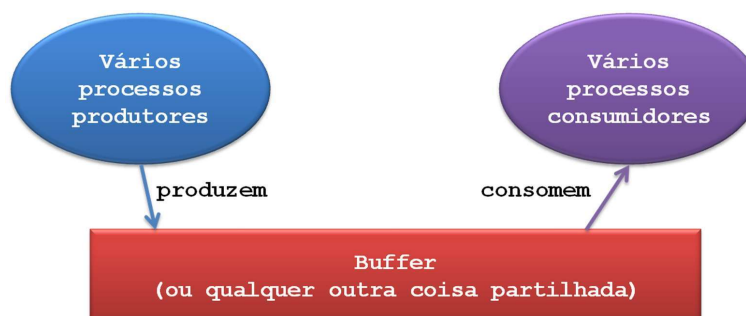
```

Mais uma vez, o problema de sincronização foi resolvido.

## 2. Exercícios propostos

- 2.1. Considere o programa no ficheiro `SynchBasico.py`. Corra o programa e verifique que este não está a executar normalmente. Utilize um semáforo para resolver o problema de sincronização.

Para ilustrar o potencial dos semáforos como mecanismo de sincronização, vamos mostrar como se pode resolver um problema clássico de sincronização – o problema do produtor-consumidor – usando semáforos. Neste problema, um produtor é um processo que produz informação. O processo que consome essa informação é o consumidor.





A solução inclui a criação de uma região de memória que será compartilhada pelo produtor e pelo consumidor. O produtor e o consumidor têm de estar sincronizados, de forma que o consumidor não tente consumir algo que o produtor ainda não produziu.

Na solução abaixo (solução 1) usamos um *buffer* com dimensão 1. O semáforo *empty* indica o número de *slots* livres (1). Se houver *slots* livres, o produtor pode produzir mais elementos. O semáforo *full* indica o número de *slots* preenchidos (ainda não consumidos, portanto). Se houver *slots* preenchidos, o consumidor pode consumir mais elementos.

```
from multiprocessing import Process, Array, Semaphore
import random, time

MAX_SIZE = 1
buffer = Array("i", MAX_SIZE)
empty = Semaphore(MAX_SIZE) #Inicialmente, MAX_SIZE posicoes livres
full = Semaphore(0) #Inicialmente, 0 posicoes ocupadas

def produtor():
    while True:
        nextProduced = random.randint(1,100)
        empty.acquire()      #Ha' posicoes livres?
        buffer[0] = nextProduced
        full.release()       #Informo que há nova posicao ocupada
        print ("+++Produzi ", str(nextProduced))
        time.sleep(random.randint(0,3)) #descanso um pouco

def consumidor():
    while True:
        
        nextConsumed = buffer[0]
        
        print ("---Consumi ", str(nextConsumed))
        time.sleep(random.randint(0,3)) #descanso um pouco

prod = Process(target=produtor)
cons = Process(target=consumidor)

prod.start()
cons.start()
prod.join()
cons.join()
```

## 2.2. Complete o exemplo anterior com as duas instruções omissas.

De seguida é apresentada a solução 2 para o problema do produtor-consumidor. Neste caso a variável *buffer* é um *buffer* circular que armazena *MAX\_SIZE* inteiros. De igual forma, o produtor e o consumidor têm de estar sincronizados, de forma que o consumidor não tente consumir algo que o produtor ainda não produziu. O semáforo *empty* indica o número de *slots* livres (*MAX\_SIZE*). Se houver *slots* livres, o produtor pode produzir mais elementos. O semáforo *full* indica o número de *slots* preenchidos (ainda não consumidos, portanto). Se houver *slots* preenchidos, o consumidor pode consumir mais elementos.

Nesta solução, o produtor começa por esperar que haja um *slot* vazio para poder produzir um novo elemento. Se houver, coloca na posição corrente dada pelo seu índice o novo elemento e incrementa o índice. Como *buffer* é circular, este

índice tem de dar a volta quando chega ao último elemento. O consumidor, por seu lado, espera que haja *slots* cheios, com informação ainda não consumida. Quando houver pode consumir o elemento da posição corrente e, tal como o produtor, termina incrementando o seu índice e garantindo simultaneamente que ele dá a volta.

```
from multiprocessing import Process, Array, Semaphore
import random, time

MAX_SIZE = 5
buffer = Array("i", MAX_SIZE)

empty = Semaphore(MAX_SIZE) #Inicialmente, MAX_SIZE posicoes livres
full = Semaphore(0)         #Inicialmente, 0 posicoes ocupadas

def produtor():
    inPosition = 0
    while True:
        nextProduced = random.randint(1,100)
        empty.acquire() #Ha' posicoes livres?
        buffer[inPosition] = nextProduced
        temp = inPosition
        inPosition = (inPosition + 1) % MAX_SIZE
        full.release() #Informo que há nova posicao ocupada
        print ("+++Produzi ", str(nextProduced), " na posicao ", str(temp))
        time.sleep(random.randint(0,3)) #descanso um pouco

def consumidor():
    outPosition = 0
    while True:
        full.acquire()
        temp = outPosition
        outPosition = (outPosition + 1) % MAX_SIZE
        print ("---Consumi ", str(buffer[temp]), " na posicao ", str(temp))
        time.sleep(random.randint(0,3)) #descanso um pouco

prod = Process(target=produtor)
cons = Process(target=consumidor)

prod.start()
cons.start()

prod.join()
cons.join()
```

### 2.3. Complete o exemplo anterior com as instruções omissas.

O resultado da execução deste programa, `ProdutorConsumidor.py`, é apresentado a seguir. O facto de o consumidor consumir na mesma ordem em que o produtor produz os novos elementos colocados no *buffer* é demonstrativo da correta resolução do problema.

```
$ python ProdutorConsumidor.py
+++Produzi 13 na posicao 0
---Consumi 13 na posicao 0
+++Produzi 77 na posicao 1
---Consumi 77 na posicao 1
+++Produzi 72 na posicao 2
+++Produzi 7 na posicao 3
```

```

---Consumi 72 na posicao 2
+++Produzi 20 na posicao 4
---Consumi 7 na posicao 3
---Consumi 20 na posicao 4
+++Produzi 65 na posicao 0
---Consumi 65 na posicao 0
+++Produzi 100 na posicao 1
---Consumi 100 na posicao 1
+++Produzi 73 na posicao 2
+++Produzi 93 na posicao 3
---Consumi 73 na posicao 2
---Consumi 93 na posicao 3
+++Produzi 24 na posicao 4
+++Produzi 67 na posicao 0
---Consumi 24 na posicao 4
+++Produzi 94 na posicao 1

(...)

```

Até agora, apenas foram utilizados um produtor e um consumidor (implementado pelo código do programa `ProdutorConsumidor.py`). No entanto, esta solução pode ser adaptada para o caso em que podem existir vários produtores e vários consumidores.

- 2.4. Analisando o código do **`ProdutorConsumidor.py`** e querendo adaptá-lo para vários produtores e vários consumidores, qual a variável que tem de ser partilhada entre os vários produtores? E qual a variável que tem de ser partilhada entre os vários consumidores?

Altere o código de modo a ter 2 produtores, 2 consumidores e as respetivas variáveis partilhadas.

- 2.5. Com a implementação anterior é possível ter os 2 produtores a alterar o seu índice ao mesmo tempo. Caso ocorra esta situação a variável `inPosition` pode ter valores incorretos. Resolva este problema identificando a secção crítica do código dos produtores.
- 2.6. O mesmo ocorre com os consumidores. Resolva também este problema.
- 2.7. Normalmente as soluções apresentadas na bibliografia usam o mesmo semáforo para resolver os dois problemas anteriores.
  - a) É possível usar 2 semáforos diferentes? Justifique
  - b) Quais as vantagens de usar 2 semáforos diferentes? Justifique.

### 3. Exercícios complementares

- 3.1. Altere o programa anterior (resultante da pergunta 2.6) de forma a incluir um segundo buffer, `controlBuffer`. Este novo *buffer* deve ter o mesmo tamanho do original e deve armazenar informação que permita saber se cada uma das posições do *buffer* original está livre ou ocupado. Quando o produtor escreve na posição `i` de `myBuffer`, a posição correspondente do `controlBuffer` deve ser colocada a 1. Quando o

consumidor ler um determinado *slot*, então esse inteiro deverá ser colocado a 0. O programa deve mostrar o conteúdo dos dois *buffers* no ecrã cada vez que houver produção/consumo de informação.

3.2. Partindo do exercício anterior, altere o programa de modo que:

- a) O produtor gere tantos números quanto a capacidade do *buffer* (pode experimentar com valores de *buffer* maiores do que o atual). O consumidor só deverá começar a consumir quando o produtor encher totalmente o *buffer* (deve ser usado um semáforo para contemplar esta situação).
- b) Quando o buffer estiver cheio, o consumidor deve consumir o conteúdo do *buffer* por ordem crescente. Isto é, deve começar por consumir o número mais baixo que está no *buffer*, avançar para o segundo mais baixo, etc., até consumir o número mais alto.

3.3. Altere uma das versões do programa anterior de forma que o consumidor consuma os valores na ordem inversa em que o produtor os produz. O consumidor só deverá começar a consumir quando o produtor encher totalmente o *buffer* (deve ser usado um semáforo para contemplar esta situação).

3.4. Resolver o problema do leitor/escritor utilizando semáforos.

#### **4. Bibliografia e outro material de apoio**

[1] Mark Lutz, “Programming Python, 4th edition, December 2010 (capítulo 5)