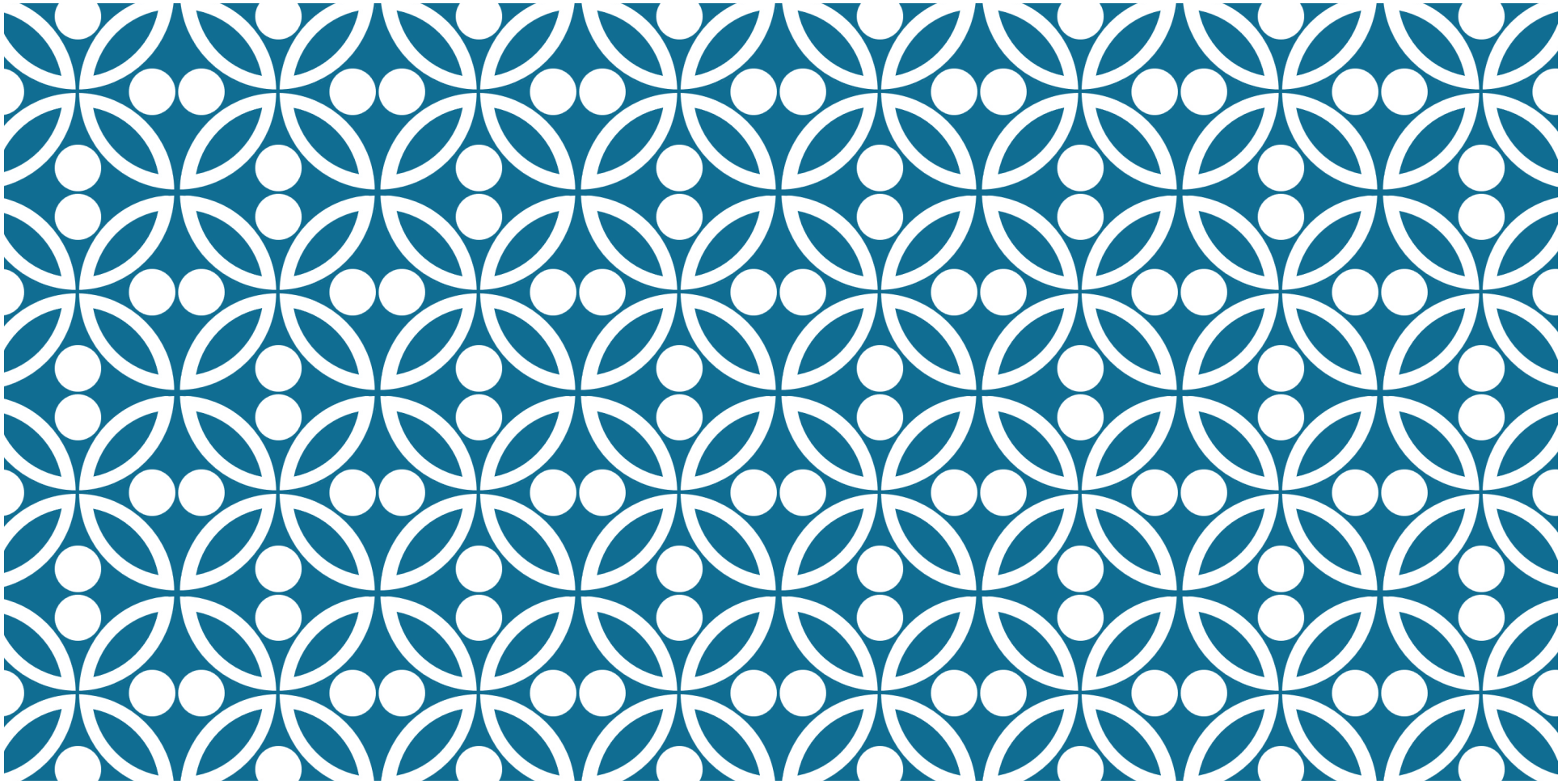
AULA 4



ORDENANDO AS COMUNICAÇÕES

USANDO QUEUE PARA GERAR QUATRO STRINGS ALEATÓRIAS EM PARALELO.

```
import multiprocessing as mp
import random
import string
random.seed(123)
# Define an out queue
out = mp.Queue()
# define a example function
def rand_string(length, out):
# Generates a random string of
numbers and chars
rand_str = ''.join(random.choice(
string.ascii_lowercase
+ string.ascii_uppercase
+ string.digits)
for i in range(length))
out.put(rand_str)
```

```
# Setup a list of processes to run
processes = [mp.Process(target=rand_string,
args=(5, out)) for x in range(4)]
# Run processes
for p in processes:
p.start()
# Exit the completed processes
for p in processes:
p.join()
# Get process results from the out queue
results = [out.get() for p in processes]
print(results)
```

COMO RECUPERAR RESULTADOS EM UMA ORDEM ESPECÍFICA

A ordem dos resultados obtidos não precisa necessariamente corresponder à ordem dos processos (na lista de processos).

Como eventualmente usamos o método .get() para recuperar os resultados da Fila sequencialmente, a ordem em que os processos são concluídos determina a ordem de nossos resultados.

Por exemplo, se o segundo processo terminou antes do primeiro processo, a ordem das sequências na lista de resultados também poderia ter sido

['PQpqM', 'yzQfA', 'SHZYV', 'PSNkD']

em vez de ['yzQfA', 'PQpqM', 'SHZYV', 'PSNkD']

Se nossa aplicação exigisse que recuperássemos os resultados em uma ordem específica, uma possibilidade seria nos referirmos ao atributo ._identity dos processos.

Nesse caso, poderíamos simplesmente usar os valores de nosso objeto range como argumento de posição. O código modificado seria:

ORDENANDO OS RESULTADOS

```
# Define an output queue
out = mp.Queue()
# Define a example function
def rand_string(length, pos, output):
  """ Generates a random string of numbers, lower- and uppercase chars. """
  rand_str = ''.join(random.choice(
  string.ascii_lowercase
  + string.ascii_uppercase
  + string.digits)
  for i in range(length)):
    out.put((pos, rand_str))
  # Setup a list of processes that we want to run
  processes = [mp.Process(target=rand_string, args=(5, x, out)) for x in range(4)]
  # Run processes
  for p in processes:
    p.start()
  # Exit the completed processes
  for p in processes:
    p.join()
  # Get process results from the output queue
  results = [out.get() for p in processes]
  print(results)
```

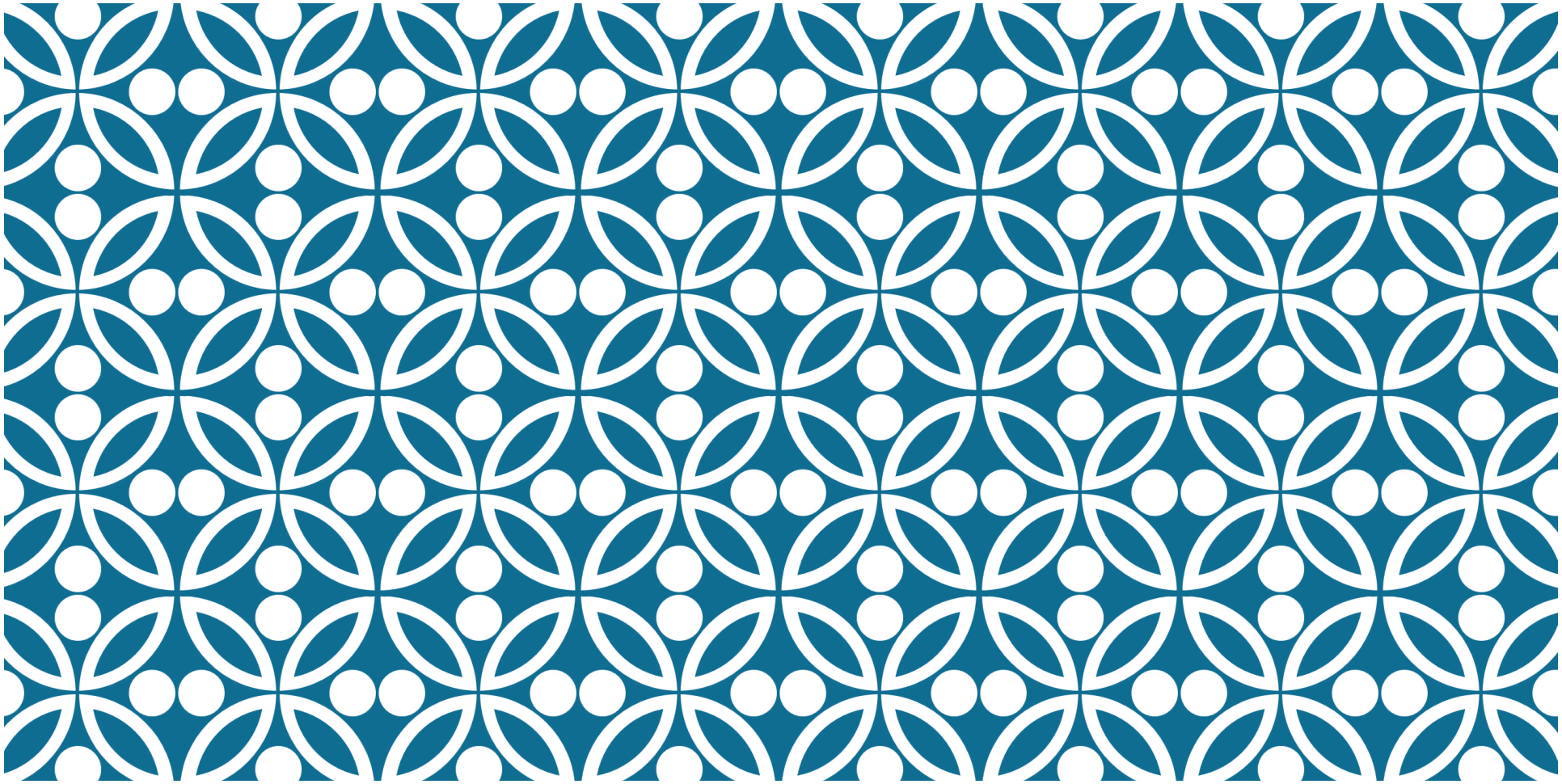
Saída:
[(0, 'h5hoV'), (1, 'fvdmN'), (2, 'rxGX4'), (3, '8hDJj')]

COMO RECUPERAR RESULTADOS EM UMA ORDEM ESPECÍFICA

Para ter certeza de que recuperamos os resultados em ordem, podemos simplesmente classificar os resultados e, opcionalmente, nos livrar do argumento de posição:

```
results.sort()
results = [r[1] for r in results]
print(results)
```

Uma maneira mais simples de manter uma lista ordenada de resultados é usar as funções Pool.apply e Pool.map, as quais discutiremos na próxima seção.



POOL DE PROCESSOS

A CLASSE POOL

Outra abordagem mais conveniente para tarefas simples de processamento paralelo é fornecida pela classe Pool.

Existem quatro métodos que são particularmente interessantes:

- Pool.apply
- Pool.map
- Pool.apply_async
- Pool.map_async

Os métodos Pool.apply e Pool.map são basicamente equivalentes às funções de mapear e aplicar embutidas do Python.

A CLASSE POOL

Antes de chegarmos às variantes assíncronas dos métodos Pool, vamos dar uma olhada em um exemplo simples usando Pool.apply e Pool.map.

O método pode subdividir a entrada (map) ou não (apply)

Aqui, vamos definir o número de processos para 4, o que significa que a classe Pool permitirá apenas 4 processos em execução ao mesmo tempo.

```
def cube(x):
return x**3
```

```
pool = mp.Pool(processes=4)
results = [pool.apply(cube,
args=(x,)) for x in range(1,7)]
print(results)
```

```
[1, 8, 27, 64, 125, 216]
```

```
def cube(x):
return x**3
```

```
pool = mp.Pool(processes=4)
results = pool.map(cube, range(1,7))
print(results)
```

```
[1, 8, 27, 64, 125, 216]
```

A CLASSE POOL

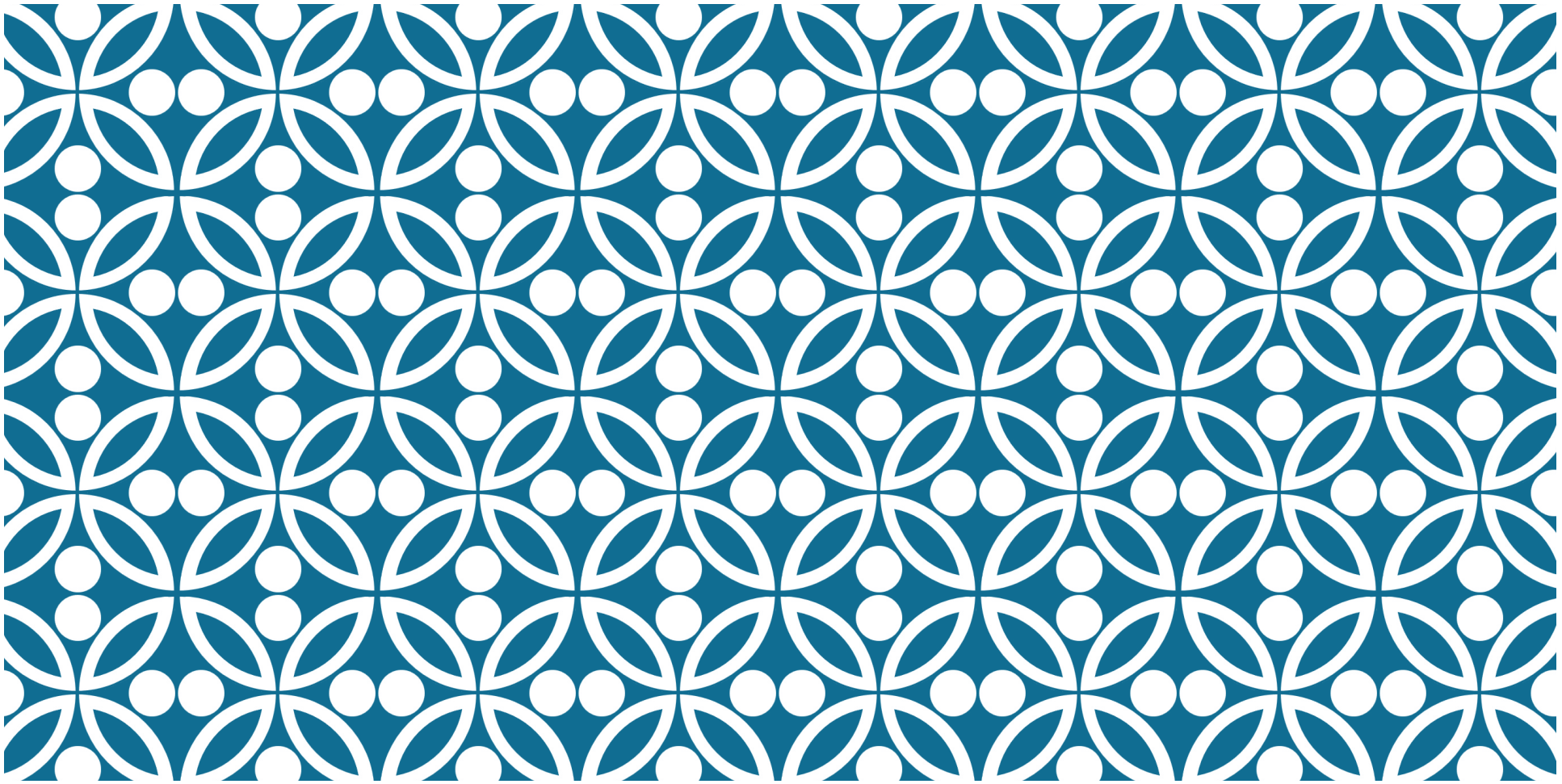
O Pool.map e o Pool.apply bloquearão o programa principal até que todos os processos sejam concluídos, o que é bastante útil se quisermos obter resultados em uma ordem específica para determinados aplicativos.

Em contraste, as variantes assíncronas enviarão todos os processos de uma vez e recuperarão os resultados assim que forem concluídos.

Mais uma diferença é que precisamos usar o método get após a chamada apply_async() para obter os valores de retorno dos processos finalizados.

```
pool = mp.Pool(processes=4)
results = [pool.apply_async(cube, args=(x,)) for x in range(1,7)]
output = [p.get() for p in results]
print(output)
```

```
[1, 8, 27, 64, 125, 216]
```



SINALIZANDO ENTRE PROCESSOS USANDO **EVENT**

SINALIZANDO ENTRE PROCESSOS

A classe Event fornece uma maneira simples de comunicar informações de estado entre processos.

Um evento pode ser alternado entre estados definidos e não definidos.

Os usuários do objeto de evento podem esperar que ele mude de não configurado para definido, usando um valor de tempo limite opcional.

Quando wait() expira, retorna sem erro. O chamador é responsável por verificar o estado do evento usando is_set().

EVENT

is_set()

- Retorna verdadeiro se e somente se o sinalizador interno for verdadeiro.

set()

- Defina o sinalizador interno como true. Todos os processos que esperam que isso se torne verdade são despertados. Processos que chamam wait() quando o sinalizador é true, não serão bloqueados.

clear()

- Redefinir o sinalizador interno para false. Subseqüentemente, os segmentos que chamam wait() serão bloqueados até que set() seja chamado para definir o sinalizador interno como true novamente.

wait(timeout=None)

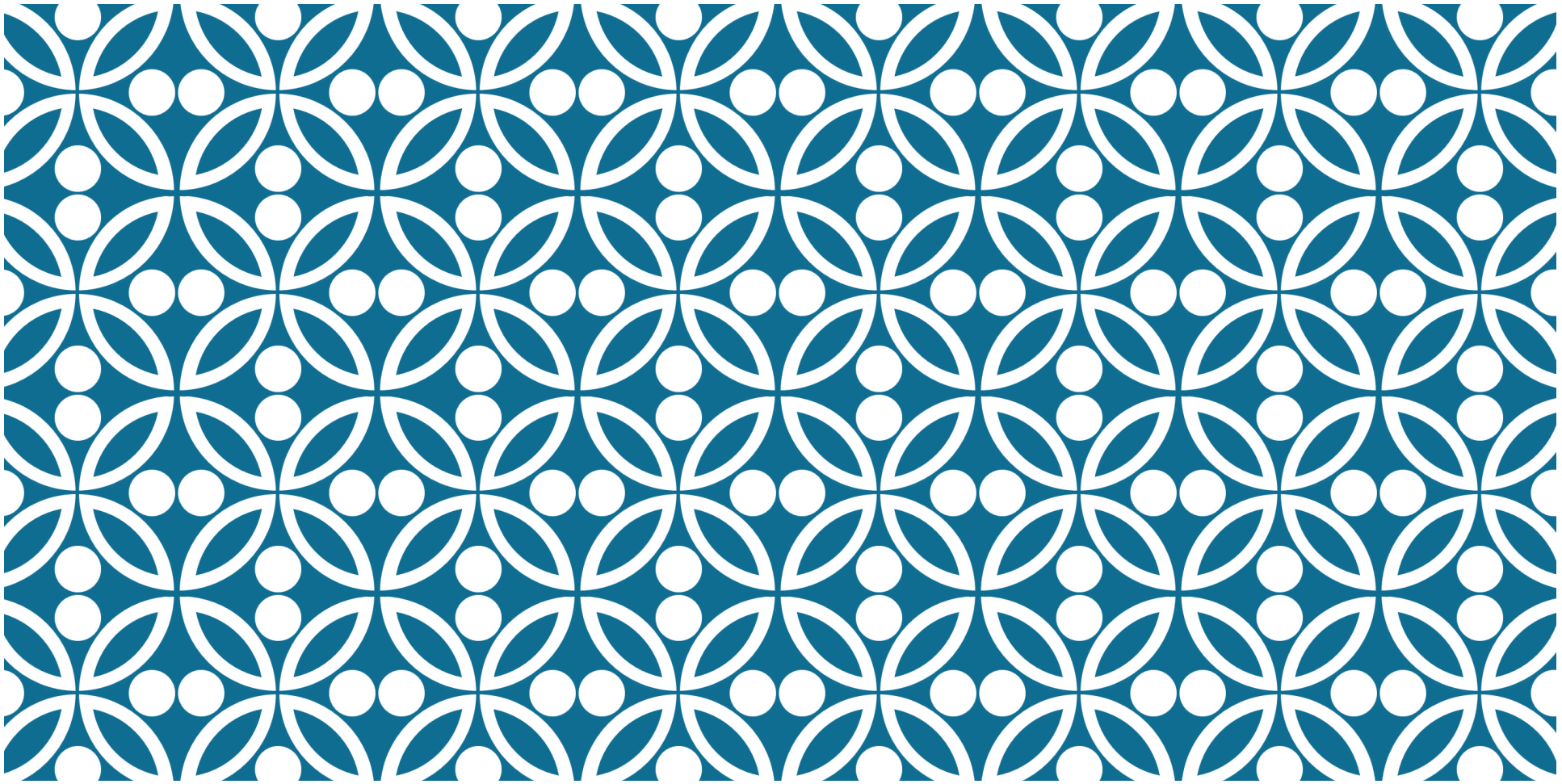
- Bloqueie até que o sinalizador interno seja verdadeiro. Se o sinalizador interno for verdadeiro na entrada, retorne imediatamente. Caso contrário, bloqueie até que outro processo chame set() para definir o sinalizador como true ou até que ocorra o tempo limite opcional.
- Quando o argumento timeout está presente e não é None, ele deve ser um número ponto flutuante especificando um tempo limite para a operação em segundos (ou frações dele).

SINALIZANDO ENTRE PROCESSOS

```
import multiprocessing
import time

def wait_event(e):
  """Wait for the event to be set before doing anything"""
  print 'wait_for_event: starting'
  e.wait()
  print 'wait_event: e.is_set()->', e.is_set()
def wait_event_timeout(e, t):
  """Wait t seconds and then timeout"""
  print 'wait_for_event_timeout: starting'
  e.wait(t)
  print 'wait_event_timeout: e.is_set()->', e.is_set()

if __name__ == '__main__':
  e = multiprocessing.Event()
  w1 = multiprocessing.Process(name='block', target=wait_event, args=(e,))
  w1.start()
  w2 = multiprocessing.Process(name='non-block', target=wait_event_timeout, args=(e, 2))
  w2.start()
  print 'main: waiting before calling Event.set()'
  time.sleep(3)
  e.set()
  print 'main: event is set'
```

Saída:

```
main: waiting before calling Event.set()
wait_for_event: starting
wait_for_event_timeout: starting
wait_for_event_timeout: e.is_set()-> False
main: event is set
wait_for_event: e.is_set()-> True
```



SINCRONIZANDO PROCESSOS USANDO **BARRIER**

SINCRONIZANDO PROCESSOS COM BARRIER

Essa classe fornece uma primitiva de sincronização simples para uso por um número fixo de processos que precisam aguardar um ao outro.

Cada um dos threads tenta passar a barreira chamando o método wait() e irá bloquear até que todos os threads tenham feito suas chamadas wait(). Neste ponto, os processos são liberados simultaneamente.

A barreira pode ser reutilizada qualquer número de vezes para o mesmo número de threads.

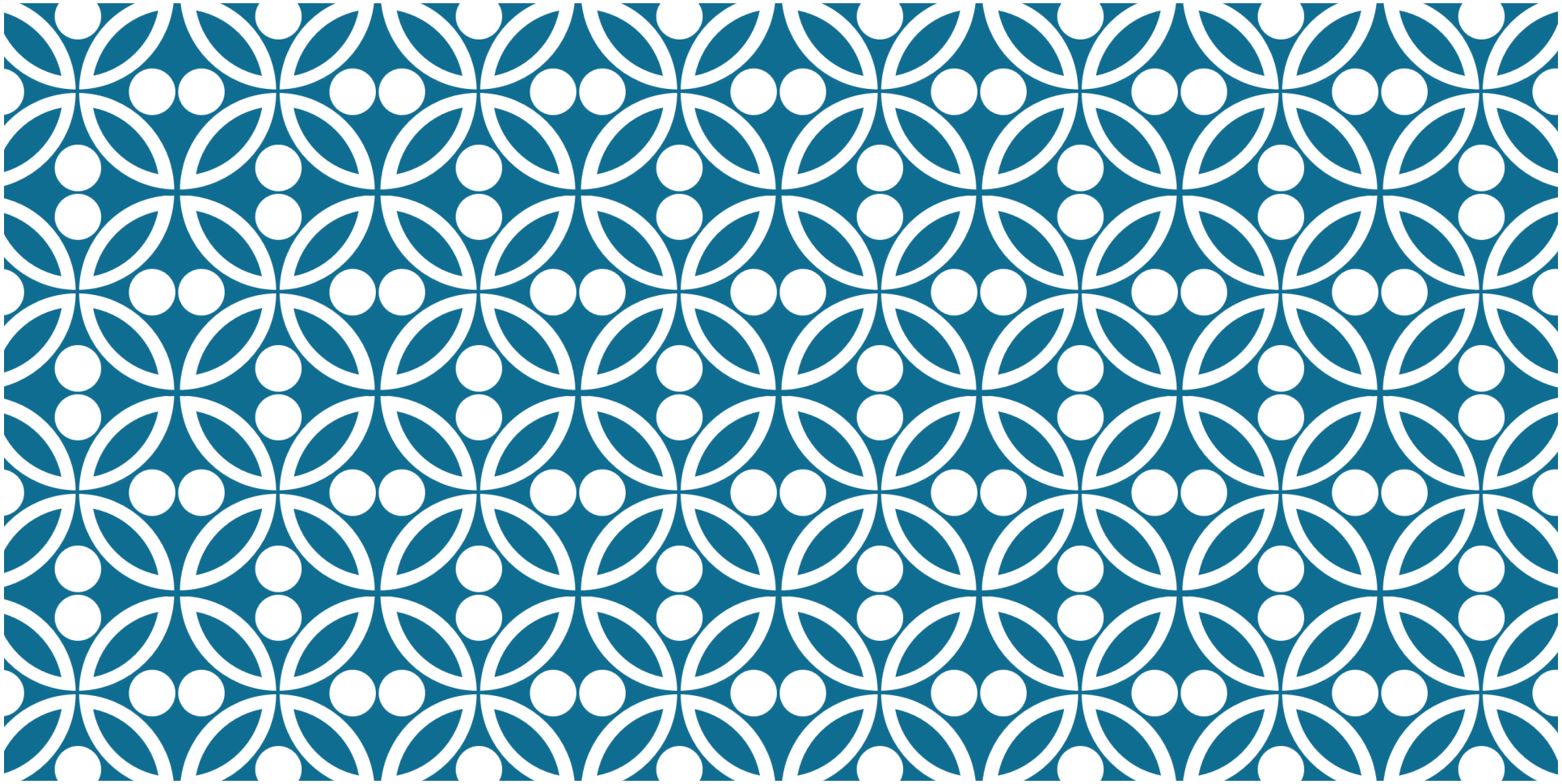
Presente apenas na versão 3.3.

CONTROLANDO ACESSO A RECURSOS

```
import multiprocessing
import sys

def server(b):
  start_server()
  b.wait()
  while True:
    connection = accept_connection()
    process_server_connection(connection)

def client(b):
  b.wait()
  while True:
    connection = make_connection()
    process_client_connection(connection)

b = multiprocessing.Barrier(2, timeout=5)
s = multiprocessing.Process(target=server, args=(b,))
c = multiprocessing.Process(target=client, args=(b,))
s.start()
c.start()
s.join()
c.join()
```

In this example, the messages printed to the console may be jumbled together if the two processes do not synchronize their access of the output stream with the lock.

Saída:
Lock acquired via with
Lock acquired directly



DEMAIS FUNÇÕES DIVERSAS

FUNÇÕES DIVERSAS

multiprocessing.active_children()

- Return list of all live children of the current process.
- Calling this has the side effect of “joining” any processes which have already finished.

multiprocessing.cpu_count()

- Return the number of CPUs in the system.
- This number is not equivalent to the number of CPUs the current process can use. The number of usable CPUs can be obtained with len(os.sched_getaffinity(0))
- May raise NotImplementedError.
- See also: os.cpu_count()

FUNÇÕES DIVERSAS

run()

- Method representing the process's activity.
- You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

start()

- Start the process's activity.
- This must be called at most once per process object. It arranges for the object's run() method to be invoked in a separate process.

join([timeout])

- If the optional argument timeout is None (the default), the method blocks until the process whose join() method is called terminates. If timeout is a positive number, it blocks at most timeout seconds. Note that the method returns None if its process terminates or if the method times out. Check the process's exitcode to determine if it terminated.
- A process can be joined many times.
- A process cannot join itself because this would cause a deadlock. It is an error to attempt to join a process before it has been started.

FUNÇÕES DIVERSAS

is_alive()

- Return whether the process is alive.
- Roughly, a process object is alive from the moment the start() method returns until the child process terminates.

terminate()

- Terminate the process. On Unix this is done using the SIGTERM signal; on Windows TerminateProcess() is used. Note that exit handlers and finally clauses, etc., will not be executed.
- Note that descendant processes of the process will not be terminated – they will simply become orphaned.
- Atenção: If this method is used when the associated process is using a pipe or queue then the pipe or queue is liable to become corrupted and may become unusable by other process. Similarly, if the process has acquired a lock or semaphore etc. then terminating it is liable to cause other processes to deadlock.

Note that the start(), join(), is_alive(), terminate() and exitcode methods should only be called by the process that created the process object.