UNIVERSIDADE DA CORUÑA

epef

Programación multihilo y multiproceso

Máster Universitario en Informática Industrial y Robótica

José Antonio Becerra Permuy
Alma María Mallo Casdelo

Contenido

1. Procesos e hilos.
2. Implementación de hilos en Python.
3. Implementación de multiproceso en Python.
4. Intercambio de datos entre procesos.
5. Sincronización (de procesos e hilos).

UNIVERSIDADE DA CORUÑA

epef

Procesos e hilos

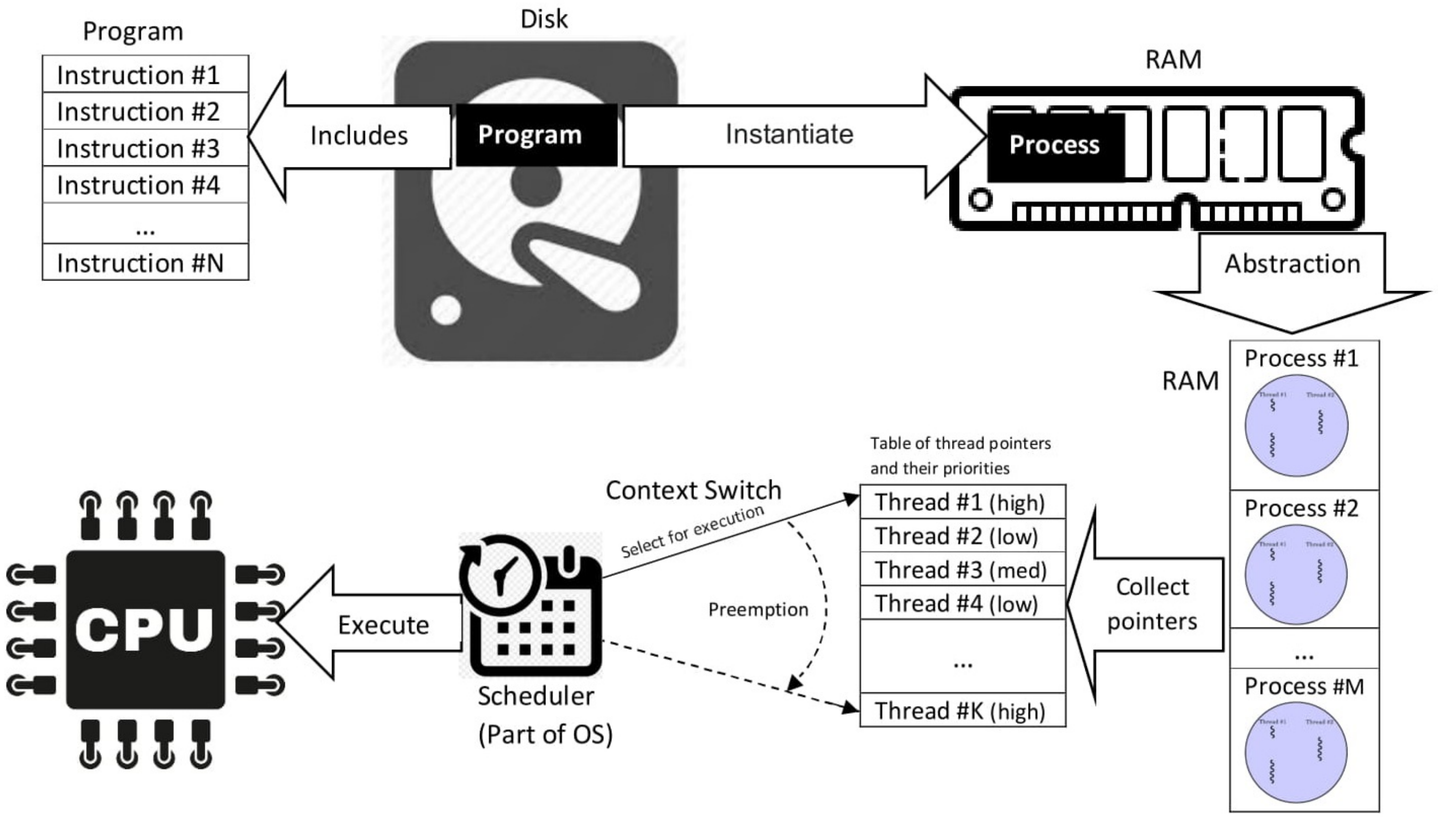
Procesos e hilos

- Un proceso es una instancia en ejecución de un programa.
- Consta de:
  - Un espacio en memoria RAM que contiene las instrucciones (en código máquina) del programa, la memoria reservada por él (*heap*) y la pila (*stack*).
  - Descriptores de recursos del S.O. en uso (ficheros, sockets…).
  - Atributos (identificador, permisos, propietario…).
  - Contexto: registros, tabla de páginas de memoria RAM…

Procesos e hilos

- Un proceso puede ejecutar bloques de código simultáneamente => hilos de ejecución (*threads*):
  - El S.O. puede tener soporte específico para ellos o no => *kernel level threads* vs. *user level threads*.
  - Comparten el espacio de memoria => es necesario proteger las variables ante accesos concurrentes.

Procesos e hilos



https://en.wikipedia.org/wiki/Process_(computing)

Implementación de hilos en Python

Implementación: hilos

- Módulo `threading` (gestión de hilos de forma explícita):
  - Clase `Thread`.
  - Especialización de la clase `Thread`.
- Módulo `concurrent.futures` (reparto dinámico de tareas entre un conjunto de *futures*, hilos o procesos):
  - Clase `ThreadPoolExecutor`.

Implementación: hilos

- Se puede utilizar el módulo `threading` o `concurrent.futures`.
- Implementación del intérprete CPython muy limitada: solo un hilo puede ejecutar *bytecode* en un instante de tiempo (GIL) => No se usa más de un núcleo.
- Importantes excepciones en las que se deshabilita el GIL:
  - Cuando hay que hacer varias tareas de E/S intensivas (se deshabilita el GIL mientras se espera el resultado de una operación de E/S).
  - Cuando se usan llamadas a librerías de cálculo intensivo implementadas en otros lenguajes. Ej: NumPy

Implementación: hilos

- A partir de Python 3.13 (como muy pronto).
- API de subintérpretes disponible directamente en Python, cada uno con su GIL.
- Los diferentes subintérpretes se comportarán como diferentes *kernel level threads* y los threads en un mismo subintérprete como diferentes *user level threads*.
- Compartir datos entre diferentes subintérpretes se hará mediante `interpreters.Queue`, y el objeto, en realidad, será compartido o copiado dependiendo del tipo de dato.
- Opciones:
  - `interpreters`
  - `concurrent.futures.InterpreterPoolExecutor`.

API (1/2): hilos

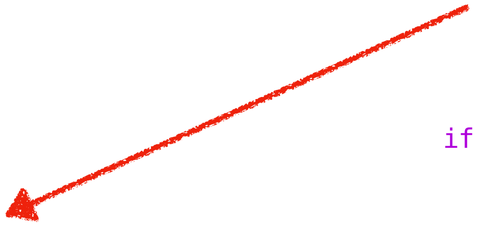
- Usando `threading`, dos formas:
  - Construir un objeto `Thread` pasando un método / función al constructor.
  - Crear una clase que herede de `Thread` e implementar el método `run()`.
- Métodos principales:
  - `start()` => inicia el thread.
  - `join()` => espera hasta que finaliza el thread.

Ejemplo pasando función al constructor de Thread

```python
from threading import Thread
from timeit import timeit
import random


def calculo(n):
    for _ in range(n):
        random.random()


def single_thread(counter, n_loops):
    for _ in range(n_loops):
        calculo(counter)
```

```python
def multi_thread(counter, n_loops):
    threads = []
    for _ in range(n_loops):
        thread = Thread(target=calculo, args=(counter,))
        thread.start()
        threads.append(thread)
    for thread in threads:
        thread.join()
```

```python
def main():
    t1 = timeit("single_thread(80000000, 4)", globals=globals(),
         number=1)
    print(f"Tiempo de ejecución usando 1 thread: {t1:.3f}")
    t2 = timeit("multi_thread(80000000, 4)", globals=globals(),
         number=1)
    print(f"Tiempo de ejecución usando 4 threads: {t2:.3f}")


if __name__ == "__main__":
    main()
```



Ejemplo:
Tiempo de ejecución usando 1 thread: 10.242
Tiempo de ejecución usando 4 threads: 10.277

Ejemplo con clase heredada de Thread

```python
from threading import Thread
from timeit import timeit
import random
```

```python
class MiThread(Thread):
    def __init__(self, counter, **kwargs):
        super().__init__(**kwargs)
        self.counter = counter

    def run(self):
        calculo(self.counter)
```

```python
def calculo(n):
    for _ in range(n):
        random.random()
```

```python
def single_thread(counter, n_loops):
    for _ in range(n_loops):
        calculo(counter)
```

```python
def multi_thread(counter, n_loops):
    threads = []
    for _ in range(n_loops):
        thread = MiThread(counter)
        thread.start()
        threads.append(thread)
    for thread in threads:
        thread.join()
```

```python
def main():
    t1 = timeit("single_thread(80000000, 4)", globals=globals(),
         number=1)
    print(f"Tiempo de ejecución usando 1 thread: {t1:.3f}")
    t2 = timeit("multi_thread(80000000, 4)", globals=globals(),
         number=1)
    print(f"Tiempo de ejecución usando 4 threads: {t2:.3f}")
```

```python
if __name__ == "__main__":
    main()
```

Ejemplo:
Tiempo de ejecución usando 1 thread: 10.248
Tiempo de ejecución usando 4 threads: 10.272

API (2/2): hilos

- Usando `concurrent.futures`:
  - Construir un objeto `ThreadPoolExecutor` especificando el número de hilos deseado.
  - Interesa cuando el número total de tareas es superior al de hilos que se quieren utilizar.
  - El parámetro `max_workers` representa el número máximo de hilos a utilizar y, desde Python 3.8, vale por defecto `min(32, os.cpu_count() + 4)`.
- Métodos principales:
  - `submit()` => añade una función / método al pool para su ejecución.
  - `map()` => aplica una función / método a todos los elementos de un objeto iterable.
  - `shutdown()` => finaliza el pool. El parámetro `cancel_futures` controla si se espera por las tareas que no han finalizado o no. No es necesario si el objeto se usa con `with`.

Ejemplo con ThreadPoolExecutor

```python
from concurrent.futures import ThreadPoolExecutor
from timeit import timeit
import random


def calculo(n):
    for _ in range(n):
        random.random()


def single_thread(counter, n_loops):
    for _ in range(n_loops):
        calculo(counter)


def multi_thread_submit(counter, n_loops):
    with ThreadPoolExecutor(max_workers=n_loops) as pool:
        for _ in range(n_loops):
            pool.submit(calculo, counter)


def multi_thread_map(counter, n_loops):
    with ThreadPoolExecutor(max_workers=n_loops) as pool:
        pool.map(calculo, [counter] * n_loops)
```

```python
def main():
    t = timeit("single_thread(80000000, 4)", globals=globals(),
        number=1)
    print(f"Tiempo de ejec. usando 1 thread: {t:.3f}")
    t = timeit("multi_thread_submit(80000000, 4)",
        globals=globals(), number=1)
    print(f"Tiempo de ejec. usando 4 threads y submit: {t:.3f}")
    t = timeit("multi_thread_map(80000000, 4)",
        globals=globals(), number=1)
    print(f"Tiempo de ejec. usando 4 threads y map: {t:.3f}")


if __name__ == "__main__":
    main()
```

Ejemplo:
Tiempo usando 1 thread: 10.259
Tiempo usando 4 threads y submit: 10.298
Tiempo usando 4 threads y map: 10.293

Consideraciones uso de hilos en Python

- ¿Quieren decir los resultados anteriores que no vale la pena usar de momento hilos de ejecución en Python? => ¡En absoluto!

- Los ejemplos anteriores no tienen ningún tipo de E/S. Ni para almacenamiento, ni interacción con usuario, ni red de datos, ni acceso a BD…

  - De haber E/S de algún tipo, sí que compensa usar hilos.

Control de número de threads en NumPy

```python
import time
import numpy as np
from timeit import timeit
from threadpoolctl import threadpool_limits


def multiplica_matrices(m1, m2, n_threads=0):
    if n_threads:
        with threadpool_limits(limits=n_threads):
            t = timeit("m1 @ m2", globals=locals(), number=100)
            print(f"Tiempo con {n_threads} threads: {t:.3f}")
    else:
        t = timeit("m1 @ m2", globals=locals(), number=100)
        print(f"Tiempo sin especificar nº de threads: {t:.3f}")


def main():
    rng = np.random.default_rng(time.time_ns())
    m1 = rng.random([2000, 2000])
    m2 = rng.random([2000, 2000])
    multiplica_matrices(m1, m2, 1)
    multiplica_matrices(m1, m2, 2)
    multiplica_matrices(m1, m2, 4)
    multiplica_matrices(m1, m2, 8)
    multiplica_matrices(m1, m2, 16)
    multiplica_matrices(m1, m2)
```

```python
if __name__ == "__main__":
    main()
```

- Usamos el módulo `threadpoolctl` (hay que instalarlo con pip) para limitar el número de threads que utilizará internamente NumPy. Existen otros métodos.
- Ejemplos de ejecución en una CPU con 8+2 núcleos:

```
Tiempo con 1 threads: 32.983
Tiempo con 2 threads: 16.971
Tiempo con 4 threads: 8.983
Tiempo con 8 threads: 5.616
Tiempo con 16 threads: 10.312
Tiempo sin especificar nº de threads: 9.403
```

- **NumPy tiende a utilizar automáticamente tantos threads como núcleos tiene la CPU pero** NO está garantizado y **eso no tiene por qué ser la mejor opción** (en el ejemplo no todos los núcleos son iguales, si no que 2 son más lentos).

Implementación de multiproceso en Python

Implementación: multiproceso

- Módulo `multiprocessing`:
  - Clase `Process`.
  - Especialización de la clase `Process`.
- Módulo `concurrent.futures`:
  - Clase `ProcessPoolExecutor`.

Implementación: multiproceso

- Se puede utilizar el módulo `multiprocessing` o `concurrent.futures`.
- Cada proceso tiene su espacio de memoria, así que no afecta la restricción del GIL, pero los objetos ejecutados tienen que soportar la serialización con el módulo `pickle`.
- Comparado con hilos:
  - Más consumo de memoria.
  - Más dificultad para compartir datos.
  - Más facilidad para aprovechar los diferentes núcleos.
- Nota: `multiprocessing` y `concurrent.futures` no funcionan en Jupyter, usar `multiprocess` y `pathos.multiprocessing` en su lugar (usan `dill` en vez de `pickle`, que es el origen del problema).

API (1/2): multiproceso

- Usando `multiprocess` como reemplazo de `multiprocessing`, dos formas:
  - Construir un objeto `Process` pasando un método / función al constructor.
  - Crear una clase que herede de `Process` e implementar el método `run()`.
- Métodos principales:
  - `start()` => inicia el proceso.
  - `join()` => espera hasta que finaliza el proceso.

Ejemplo pasando función al constructor de Process

```python
from timeit import timeit
import random
from multiprocess import Process


def calculo(n):
    for _ in range(n):
        random.random()


def single_process(counter, n_loops):
    for _ in range(n_loops):
        calculo(counter)
```

```python
def multi_process(counter, n_loops):
    procesos = []
    for _ in range(n_loops):
        proceso = Process(target=calculo, args=(counter,))
        proceso.start()
        procesos.append(proceso)
    for proceso in procesos:
        proceso.join()
```

```python
def main():
    t = timeit("single_process(80000000, 4)", globals=globals(),
        number=1)
    print(f"Tiempo de ejecución usando 1 proceso: {t:.3f}")
    t = timeit("multi_process(80000000, 4)", globals=globals(),
        number=1)
    print(f"Tiempo de ejecución usando 4 procesos: {t:.3f}")


if __name__ == "__main__":
    main()
```

Ejemplo:
Tiempo de ejecución usando 1 proceso: 10.756
Tiempo de ejecución usando 4 procesos: 2.824

Ejemplo con clase heredada de Process

```python
from timeit import timeit
import random
from multiprocess import Process
```

```python
class MiProceso(Process):
    def __init__(self, counter, **kwargs):
        super().__init__(**kwargs)
        self.counter = counter

    def run(self):
        calculo(self.counter)
```

```python
def calculo(n):
    for _ in range(n):
        random.random()
```

```python
def single_process(counter, n_loops):
    for _ in range(n_loops):
        calculo(counter)
```

```python
def multi_process(counter, n_loops):
    procesos = []
    for _ in range(n_loops):
        proceso = MiProceso(counter)
        proceso.start()
        procesos.append(proceso)
    for proceso in procesos:
        proceso.join()
```

```python
def main():
    t = timeit("single_process(80000000, 4)", globals=globals(),
        number=1)
    print(f"Tiempo de ejecución usando 1 proceso: {t:.3f}")
    t = timeit("multi_process(80000000, 4)", globals=globals(),
        number=1)
    print(f"Tiempo de ejecución usando 4 procesos: {t:.3f}")


if __name__ == "__main__":
    main()
```

Ejemplo:
Tiempo de ejecución usando 1 proceso: 10.517
Tiempo de ejecución usando 4 procesos: 2.773

API (2/2): multiproceso

- Usando `pathos.multiprocessing` como reemplazo de `concurrent.futures`, dos formas:
  - Construir un objeto `ProcessPool` especificando el número de procesos concurrentes deseado.
  - Interesa cuando el número total de tareas es superior al de procesos simultáneos que se quieren utilizar.
  - El parámetro `nodes` representa el número máximo de procesos a utilizar y su valor por defecto es igual al número de núcleos.
- Métodos principales:
  - `apipe()` => añade una función / método al pool para su ejecución asíncrona.
  - `amap()` => aplica una función / método a todos los elementos de un objeto iterable.
  - En ambos casos, se devuelve un objeto sobre el que se puede llamar al método `get()` para obtener el resultado o simplemente esperar a que finalicen los procesos.

Ejemplo con pathos.multiprocessing.ProcessPool

```python
from timeit import timeit
import random
from pathos.multiprocessing import ProcessPool


def calculo(n):
    for _ in range(n):
        random.random()


def single_process(counter, n_loops):
    for _ in range(n_loops):
        calculo(counter)


def multi_process_apipe(counter, n_loops):
    with ProcessPool(nodes=n_loops) as pool:
        results = []
        for _ in range(n_loops):
            result = pool.apipe(calculo, counter)
            results.append(result)
        for result in results:
            result.get()


def multi_process_amap(counter, n_loops):
    with ProcessPool(nodes=n_loops) as pool:
        results = pool.amap(calculo, [counter] * n_loops)
        results.get()


def main():
    t = timeit("single_process(80000000, 4)", globals=globals(),
        number=1)
    print(f"Tiempo de ejec. usando 1 proceso: {t:.3f}")
    t = timeit("multi_process_apipe(80000000, 4)”,
        globals=globals(), number=1)
    print(f"Tiempo de ejec. usando 4 procesos y pipe: {t:.3f}")
    t = timeit("multi_process_amap(80000000, 4)”,
        globals=globals(), number=1)
    print(f"Tiempo de ejec. usando 4 procesos y map: {t:.3f}")


if __name__ == "__main__":
    main()
```

Ejemplo:

```
Tiempo usando 1 proceso: 10.836
Tiempo usando 4 procesos y pipe asínc.: 2.793
Tiempo usando 4 procesos y map asínc.: 2.768
```

UNIVERSIDADE DA CORUÑA

epef

Intercambio de datos entre procesos

Multiproceso: intercambio de datos

- De nuevo, reemplazamos `multiprocessing` por `multiprocess`.
- Explícito:
  - `multiprocess.Pipe`.
  - `multiprocess.Queue`.
- Implícito:
  - `multiprocess.Value`.
  - `multiprocess.Array`.
  - `multiprocess.Manager`.

Multiproceso: intercambio de datos explícito

- Se puede utilizar `multiprocess.Pipe` o `multiprocess.Queue`:
  - `multiprocess.Pipe`:
    - Devuelve un par de objetos que se pueden utilizar para enviar / recibir datos con `send()` / `recv()` únicamente por dos procesos (uno por cada extremo del canal de comunicación).
    - Si el buffer de la tubería se llena, `send()` se convierte en bloqueante.
  - `multiprocess.Queue`:
    - Usa internamente `multiprocess.Pipe`.
    - Una misma cola puede ser usada por tantos procesos como se quiera, a cambio se pierde eficiencia en la comunicación.
    - Se utiliza como `queue.Queue` => los procesos implicados pueden enviar / recibir datos poniendo y quitando datos de la cola con `put()` y `get()`.

Ejemplo con multiprocess.Pipe

```python
from timeit import timeit
import random
from multiprocess import Process, Pipe


def calculo(n, conn):
    suma = 0
    for _ in range(n):
        suma += random.random()
    conn.send(suma)


def multi_process(n_elem, n_procesos):
    procesos = []
    pipes = []
    n_elem_proceso = n_elem // n_procesos
    for _ in range(n_procesos):
        me_conn, other_conn = Pipe()
        proceso = Process(target=calculo,
                          args=(n_elem_proceso, other_conn))
        proceso.start()
        procesos.append(proceso)
        pipes.append(me_conn)
    for proceso in procesos:
        proceso.join()
    resultado = 0
    for conn in pipes:
        resultado += conn.recv()


def main():
    t = timeit("multi_process(320000000, 1)", globals=globals(),
        number=1)
    print(f"Tiempo de ejecución usando 1 proceso: {t:.3f}")
    t = timeit("multi_process(320000000, 4)", globals=globals(),
        number=1)
    print(f"Tiempo de ejecución usando 4 procesos: {t:.3f}")


if __name__ == "__main__":
    main()
```

Ejemplo:

```
Tiempo de ejecución usando 1 proceso: 11.579
Tiempo de ejecución usando 4 procesos: 3.050
```

Ejemplo con multiprocess.Queue

```python
from timeit import timeit
import random
from multiprocess import Process, Queue


def calculo(n, sumas):
    suma = 0
    for _ in range(n):
        suma += random.random()
    sumas.put(suma)


def multi_process(n_elem, n_procesos):
    procesos = []
    n_elem_proceso = n_elem // n_procesos
    sumas = Queue()
    for _ in range(n_procesos):
        proceso = Process(target=calculo,
                  args=(n_elem_proceso, sumas))
        proceso.start()
        procesos.append(proceso)
    for proceso in procesos:
        proceso.join()
    resultado = 0
    while not sumas.empty():
        resultado += sumas.get()


def main():
    t = timeit("multi_process(320000000, 1)", globals=globals(),
        number=1)
    print(f"Tiempo de ejecución usando 1 proceso: {t:.3f}")
    t = timeit("multi_process(320000000, 4)", globals=globals(),
        number=1)
    print(f"Tiempo de ejecución usando 4 procesos: {t:.3f}")


if __name__ == "__main__":
    main()
```

Ejemplo:

```
Tiempo de ejecución usando 1 proceso: 11.432
Tiempo de ejecución usando 4 procesos: 3.022
```

Multiproceso: intercambio de datos implícito

- Se puede utilizar `multiprocess.Value`, `multiprocess.Array` o `multiprocess.Manager`:
  - `multiprocess.Value`:
    - Permite acceder a un dato compartido. Es necesario indicar el tipo de dato (https://docs.python.org/3/library/array.html#module-array) y utilizar un `lock` para cualquier operación no atómica. Se accede al valor con el atributo `value`.
  - `multiprocess.Array`:
    - Similar a `Value` pero para un array. Mismas consideraciones. Se accede a cada elemento como en una lista o un array de NumPy.
  - `multiprocess.Manager`:
    - Soporta Value, Array, diccionarios, listas y colas, así como los mecanismos de sincronización habituales. Se pueden crear nuevos managers para soportar otros tipos de datos. Los procesos pueden estar en ordenadores diferentes (usando los argumentos `address` y `authkey` del constructor).

Ejemplo con multiprocess.Value

```python
from timeit import timeit
import random
from multiprocess import Process, Value


def calculo(n, sumas):
    suma = 0
    for _ in range(n):
        suma += random.random()
    with sumas.get_lock():
        sumas.value += suma


def multi_process(n_elem, n_procesos):
    procesos = []
    n_elem_proceso = n_elem // n_procesos
    sumas = Value("d", 0.0)
    for _ in range(n_procesos):
        proceso = Process(target=calculo,
                  args=(n_elem_proceso, sumas))
        proceso.start()
        procesos.append(proceso)
    for proceso in procesos:
        proceso.join()
    print(sumas.value)
```

```python
def main():
    t = timeit("multi_process(320000000, 1)", globals=globals(),
        number=1)
    print(f"Tiempo de ejecución usando 1 proceso: {t:.3f}")
    t = timeit("multi_process(320000000, 4)", globals=globals(),
        number=1)
    print(f"Tiempo de ejecución usando 4 procesos: {t:.3f}")


if __name__ == "__main__":
    main()
```

Ejemplo:

```
Tiempo de ejecución usando 1 proceso: 11.525
Tiempo de ejecución usando 4 procesos: 3.041
```

Ejemplo con multiprocess.Array

```python
from timeit import timeit
import random
from multiprocess import Process, Array


def calculo(n, sumas, process_id):
    suma = 0
    for _ in range(n):
        suma += random.random()
    sumas[process_id] = suma


def multi_process(n_elem, n_procesos):
    procesos = []
    n_elem_proceso = n_elem // n_procesos
    sumas = Array("d", n_procesos)
    for i in range(n_procesos):
        proceso = Process(target=calculo,
                  args=(n_elem_proceso, sumas, i))
        proceso.start()
        procesos.append(proceso)
    for proceso in procesos:
        proceso.join()
    suma = 0
    for i in sumas:
        suma += i


def main():
    t = timeit("multi_process(320000000, 1)", globals=globals(),
        number=1)
    print(f"Tiempo de ejecución usando 1 proceso: {t:.3f}")
    t = timeit("multi_process(320000000, 4)", globals=globals(),
        number=1)
    print(f"Tiempo de ejecución usando 4 procesos: {t:.3f}")


if __name__ == "__main__":
    main()
```

Ejemplo:

```
Tiempo de ejecución usando 1 proceso: 11.511
Tiempo de ejecución usando 4 procesos: 3.036
```

Ejemplo con multiprocess.Manager

```python
from timeit import timeit
import random
from multiprocess import Process, Manager


def calculo(n, sumas, lock):
    suma = 0
    for _ in range(n):
        suma += random.random()
    with lock:
        sumas.value += suma


def multi_process(n_elem, n_procesos):
    procesos = []
    n_elem_proceso = n_elem // n_procesos
    with Manager() as manager:
        sumas = manager.Value("d", 0.0)
        lock = manager.Lock()
        for _ in range(n_procesos):
            proceso = Process(target=calculo,
                              args=(n_elem_proceso, sumas, lock))
            proceso.start()
            procesos.append(proceso)
        for proceso in procesos:
            proceso.join()
        print(sumas)


def main():
    t = timeit("multi_process(320000000, 1)", globals=globals(),
               number=1)
    print(f"Tiempo de ejecución usando 1 proceso: {t:.3f}")
    t = timeit("multi_process(320000000, 4)", globals=globals(),
               number=1)
    print(f"Tiempo de ejecución usando 4 procesos: {t:.3f}")


if __name__ == "__main__":
    main()
```

Ejemplo:

```
Tiempo de ejecución usando 1 proceso: 11.178
Tiempo de ejecución usando 4 procesos: 3.054
```

UNIVERSIDADE DA CORUÑA

epef

Sincronización (de procesos e hilos)

Sincronización

- En los hilos es necesaria para acceder a variables susceptibles de ser accedidas por más de un hilo, pero en los procesos también puede tener sentido para controlar el acceso a recursos hardware compartidos.

- Todas las clases están definidas tanto en `threading` como en `multiprocessing`:

  - `Lock()` y `RLock()`.

  - `Semaphore()`.

  - `Condition()`.

  - `Event()`.

  - `Barrier()`.

Sincronización: Lock

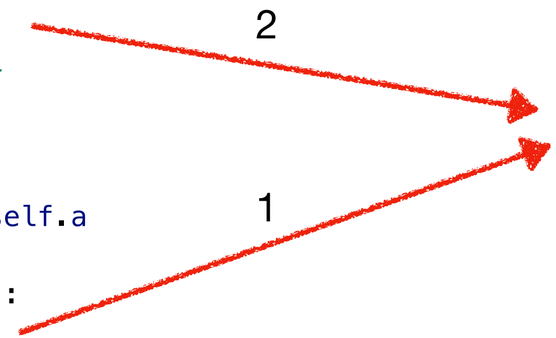
- `Lock()`.
  - Se utiliza para proteger un bloque de código de forma que no pueda ser ejecutado simultáneamente por más de un hilo / proceso, por ejemplo porque implica el acceso a un objeto compartido.
  - `acquire()` => Espera a que el bloqueo esté liberado, bloquea y continua la ejecución.
  - `release()` => Libera el bloqueo (notificándolo a un hilo / proceso) y continua la ejecución.
  - Se puede poner el bloque dentro de un `with` en vez de llamar a `acquire()` y `release()`.
- `RLock()`.
  - *Reentrant Lock*. Similar a `Lock` pero admite varias llamadas a `acquire()` dentro de un mismo hilo / proceso sin haber llamado a `release()` entre ellas.

Ejemplo con Lock

```python
import time
import numpy
import threading


class numero:
    def __init__(self):
        self._valor = 0

    @property
    def valor(self):
        return self._valor

    @valor.setter
    def valor(self, valor):
        self._valor = valor


def sum_random_vector(n_elem, suma, lock):
    rng = numpy.random.default_rng(time.time_ns())
    vector = rng.random(n_elem)
    suma_local = numpy.sum(vector)
    nombre = threading.current_thread().name
    print(f"Soy {nombre} y mi suma es {suma_local}")
    with lock:
        suma.valor += suma_local


def multi_thread(n_elem, n_loops):
    threads = []
    suma = numero()
    lock = threading.Lock()
    for _ in range(n_loops):
        thread = threading.Thread(target=sum_random_vector,
            args=(n_elem, suma, lock))
        thread.start()
        threads.append(thread)
    for thread in threads:
        thread.join()
    print(f"La suma total es {suma.valor}")


def main():
    multi_thread(1000000, 4)


if __name__ == "__main__":
    main()
```

Ejemplo con RLock

```
"""
https://stackoverflow.com/a/16568426
"""

import threading


class X:
    def __init__(self):
        self.a = 1
        self.b = 2
        self.lock = threading.RLock()

    def cambia_a(self):
        with self.lock:
            self.a += 1

    def cambia_b(self):
        with self.lock:
            self.b += self.a

    def cambia_ab(self):
        with self.lock:
            self.cambia_a()
            self.cambia_b()
```



El bloqueo lo ejecuta el mismo hilo

Sincronización: Semaphore

- `Semaphore(valor)`.
  - Se utiliza para proteger un bloque de código de forma que no pueda ser ejecutado simultáneamente por más de un número prefijado de hilos / procesos (`valor`), para lo cual mantiene un contador interno. Su uso suele estar asociado a algún recurso con capacidad limitada (ej: conexiones simultáneas a un servidor).
  - `acquire()` => Si `valor` es mayor que 0, lo decrementa y continúa la ejecución. Si no, espera a que el bloqueo sea liberado, decrementa `valor` y continua la ejecución.
  - `release()` => Incrementa `valor`, libera el bloqueo (notificándolo a un hilo / proceso) y continua la ejecución.
  - Se puede poner el bloque dentro de un `with` en vez de llamar a `acquire()` y `release()`.

Sincronización: Condition

- `Condition()`.
  - Se utiliza para proteger un bloque de código de forma que no se ejecute hasta que se cumpla una condición, en cuya evaluación interviene el contenido de algún objeto compartido cuyo acceso simultáneo tiene que ser evitado. Internamente usa un `Lock` para bloquear.
  - `acquire()` => Espera a que el bloqueo esté liberado, bloquea y continua la ejecución.
  - `release()` => Libera el bloqueo (notificándolo a un hilo / proceso) y continua la ejecución.
  - `wait_for(condicion)` => Ejecuta `acquire()`, comprueba el valor de `condicion` y:
    - Si `True` => Continua el programa.
    - Si `False` => Ejecuta `wait()`, que hace `release()` y queda a la espera de que otro hilo / proceso ejecute `notify()` y `release()`. En cuanto eso ocurre, ejecuta `acquire()` y vuelve a comprobar el valor de `condicion`.
  - `notify()` => Despierta uno de los hilos / procesos que están esperando en `wait()`.
  - Se puede poner el bloque dentro de un `with` en vez de llamar a `acquire()` y `release()`.

Sincronización: Event

- `Event()`.
  - Se utiliza para notificar algo a un hilo / proceso. Internamente contiene un *flag*.
  - `set()` => Cambia el flag a `True`. Todos los hilos / procesos que estaban en `wait()` son notificados.
  - `clear()` => Cambia el flag a `False`. Todos los hilos / procesos que después llamen a `wait()` quedarán bloqueados hasta el siguiente `set()`.
  - `wait()` => Espera a que el flag sea `True` y después continúa.

Sincronización: Barrier

- `Barrier(n_integrantes)`.
  - Se utiliza para sincronizar explícitamente en un punto del código un conjunto de hilos / procesos.
  - `wait()` => Espera a que todos los integrantes de la barrera entren en espera. En ese momento, todos despiertan y continúan su ejecución.

Referencias

- https://docs.python.org/3/library/threading.html
- https://docs.python.org/3/glossary.html#term-global-interpreter-lock
- https://peps.python.org/pep-0734/
- https://docs.python.org/3/library/concurrent.futures.html
- https://pypi.org/project/threadpoolctl/
- https://docs.python.org/3/library/multiprocessing.html
- https://multiprocess.readthedocs.io/en/latest/
- https://pypi.org/project/dill/