

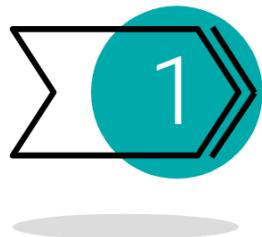
# Unidad 9: Multiprocessing

[Imprimir en PDF](#)

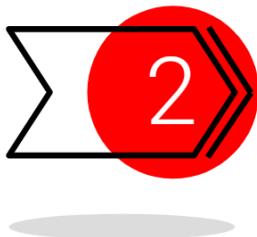
## Contenido

- Contenido de la unidad
- Definición de multiprocessing
- Multiprocessing versus Threading
- Ventajas y Desventajas
- Programación - paso a paso
- Programación - unificado
- Taller y Quiz

## Contenido de la unidad



Definición de  
Multiprocessing



Multiprocessing  
versus  
Threading



Ventajas y  
Desventajas

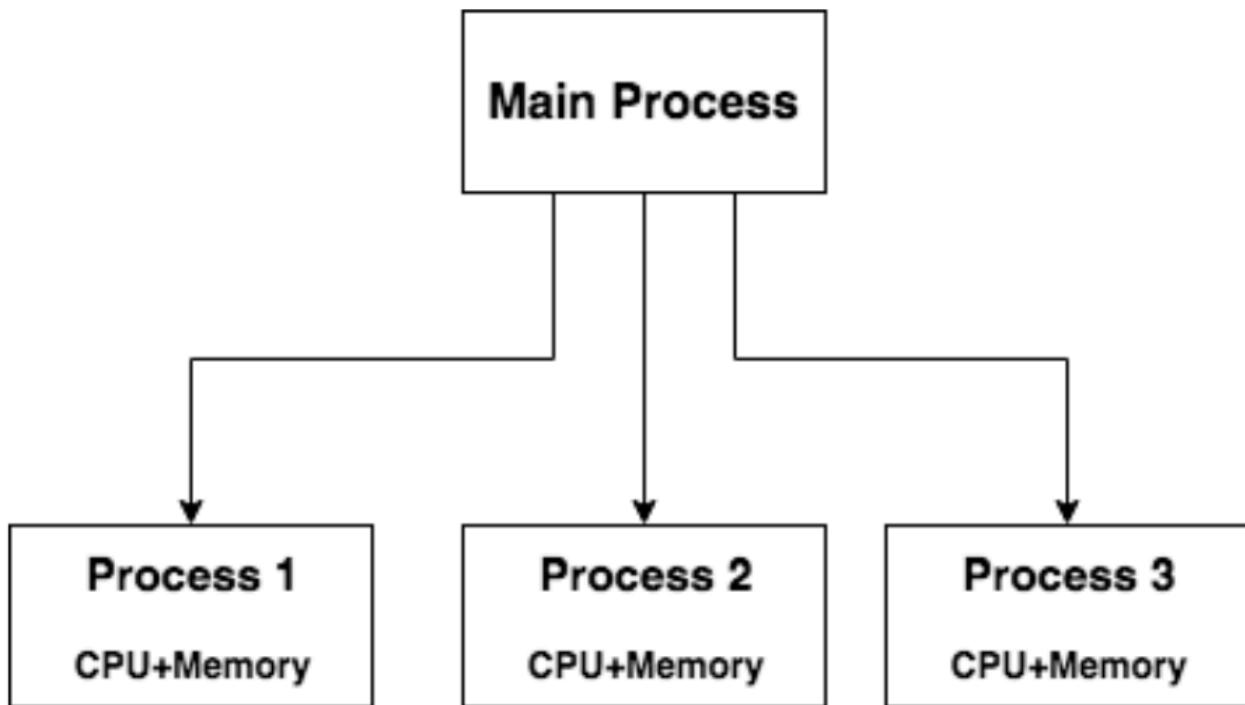


Programación

## Definición de multiprocessing

- Multiprocessing es una técnica utilizada en programación para dividir un programa en varios procesos independientes que se ejecutan en paralelo en un procesador con múltiples núcleos.
- Cada proceso tiene su propia memoria y se ejecuta de forma independiente, lo que permite una mayor eficiencia y rendimiento en la ejecución del programa.

## Representación gráfica

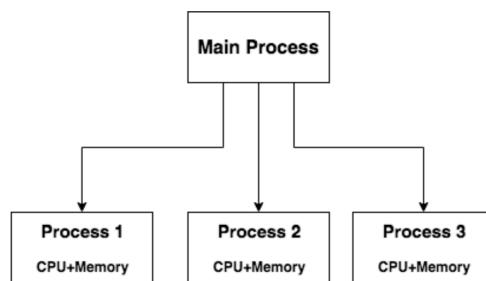


<https://hiteshmishra708.medium.com/multiprocessing-in-python-c6735fa70f3f>

# Multiprocessing versus Threading

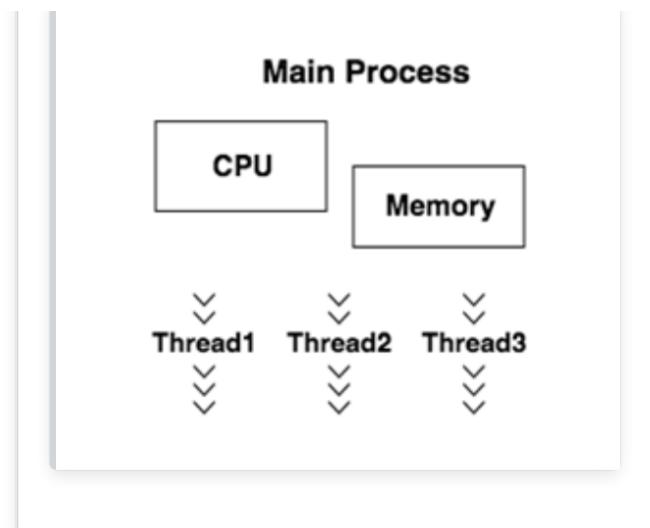
A diferencia de Threading, que utiliza hilos para dividir la tarea en diferentes subprocesos, Multiprocessing utiliza procesos que se ejecutan en núcleos de procesador separados, lo que permite una verdadera ejecución en paralelo.

## Multiprocessing



<https://hiteshmishra708.medium.com/multiprocessing-in-python-c6735fa70f3f>

## Multithreading



# Ventajas y Desventajas

## Ventajas

### 1. Mayor eficiencia y rendimiento

Multiprocessing permite la ejecución de varias tareas en paralelo, lo que acelera el procesamiento del programa y mejora su rendimiento.

### 2. Mayor capacidad de procesamiento

Los procesos se ejecutan en núcleos de procesador separados, lo que significa que el programa puede aprovechar completamente el potencial de la CPU para mejorar su capacidad de procesamiento.

### 3. Mayor estabilidad del programa

Los procesos se ejecutan de forma independiente, lo que significa que si un proceso falla, los demás pueden seguir ejecutándose sin problemas.

### 4. Mayor facilidad para la gestión de recursos

Multiprocessing gestiona los recursos del sistema de forma más efectiva, lo que significa que los programas pueden ser más eficientes en términos de memoria y uso de la CPU.

## Desventajas

### 1. Mayor consumo de recursos

La creación de procesos adicionales puede consumir más memoria y recursos del sistema, lo que puede afectar el rendimiento del programa.

### 2. Mayor complejidad del código

El uso de Multiprocessing puede requerir una mayor complejidad del código, lo que significa que el programa puede ser más difícil de entender y mantener.

### 3. Problemas de sincronización

La sincronización entre procesos puede ser un problema en Multiprocessing, lo que puede generar errores en el programa.

### 4. Problemas de comunicación

La comunicación entre procesos puede ser más compleja en Multiprocessing, lo que puede generar problemas de compatibilidad y de rendimiento.

## Programación - paso a paso

- ¿Cómo ejecutar funciones en paralelo utilizando la librería multiprocessing de Python?

**i Nota****Veamos cómo se programa**

Ver el Notebook «1 - Multiprocessing.ipynb»

- ¿Cómo asignar y obtener los nombres de los procesos?

**i Nota****Veamos cómo se programa**

Ver el Notebook «2 - Multiprocessing.ipynb»

- ¿Cómo ejecutar un proceso en segundo plano?

**i Nota****Veamos cómo se programa**

- Ver el Notebook «3 - Multiprocessing.ipynb»
- Ver el Python «3ejecutar\_proceso\_segundo\_plano.py»

- ¿Cómo matar un proceso?

**i Nota****Veamos cómo se programa**

Ver el Notebook «4 - Multiprocessing.ipynb»

- ¿Cómo usar un proceso en una subclase?

**i Nota****Veamos cómo se programa**

Ver el Notebook «5 - Multiprocessing.ipynb»

- ¿Cómo intercambiar objetos entre procesos utilizando una cola?

**Nota****Veamos cómo se programa**

Ver el Notebook «6 - Multiprocessing.ipynb»

- ¿Cómo intercambiar objetos entre procesos utilizando una tubería?

**Nota****Veamos cómo se programa**

Ver el Notebook «7 - Multiprocessing.ipynb»

- ¿Cómo sincronizar procesos?

**Nota****Veamos cómo se programa**

- Ver el Notebook «8 - Multiprocessing.ipynb»
- Ver el Python «8sincronizar\_procesos\_barrier.py»

- ¿Cómo sincronizar procesos utilizando un administrador de procesos (manager)?

**Nota****Veamos cómo se programa**

Ver el Notebook «9 - Multiprocessing.ipynb»

- ¿Cómo ejecutar funciones en paralelo utilizando procesos con Pool?

**i Nota****Veamos cómo se programa**

Ver el Notebook «10 - Multiprocessing.ipynb»

# Programación - unificado

**💡 Truco**[Acceder a Todo en uno](#)

O ver el Notebook «Multiprocessing.ipynb»

**💡 Truco**[Acceder a Jupyter files](#)**💡 Truco**[Acceder a Python files](#)

# Taller y Quiz

**💡 Truco**Ver la subsección **Evaluemos lo aprendido!!**

# 1 - Multiprocessing.ipynb

## Contenido

- 1 - Multiprocessing.ipynb
- 1 - ¿Cómo ejecutar funciones en paralelo utilizando la librería multiprocessing de Python?

## Librería Multiprocessing de Python

La librería Multiprocessing de Python es una biblioteca estándar que permite la creación de procesos en paralelo en una computadora con múltiples núcleos o CPUs. Proporciona una interfaz de programación para crear y administrar procesos de manera fácil y eficiente, lo que permite acelerar la ejecución de programas que realizan tareas intensivas en cómputo.

- La librería multiprocessing de Python permite ejecutar funciones en paralelo utilizando procesos.
- Para ello, se utiliza la función Process() de la librería multiprocessing.
- La función Process() recibe como argumentos el nombre de la función que se desea ejecutar y los argumentos de la función.
- La función Process() devuelve un objeto de tipo Process que se puede almacenar en una lista.
- Para ejecutar la función, se utiliza el método start() del objeto Process.
- Para esperar a que termine la ejecución de la función, se utiliza el método join() del objeto Process.
- En el ejemplo, se crea una función mi\_fun que imprime el número del proceso en el que se ejecuta.
- Se crea una lista Process\_jobs que almacena los objetos Process.
- Se crea un bucle for que crea 5 procesos, los almacena en la lista Process\_jobs, y los ejecuta.

- Se crea un bucle for que espera a que terminen los procesos almacenados en la lista Process\_jobs.

```
import multiprocessing

def mi_fun(i):
    print ('Función llamada en el proceso: %s \n' %i)
    return

if __name__ == '__main__':
    Process_jobs = []
    for i in range(5):
        p = multiprocessing.Process(target=mi_fun, args=(i,))
        Process_jobs.append(p)
        p.start()
    for p in Process_jobs:
        p.join()
```

Función llamada en el proceso: 0  
Función llamada en el proceso: 2  
Función llamada en el proceso: 3  
Función llamada en el proceso: 1

Función llamada en el proceso: 4

# 2 - Multiprocessing.ipynb

## Contenido

- 2 - Multiprocessing.ipynb
- 2 - ¿Cómo asignar y obtener los nombres de los procesos?

## Librería Multiprocessing de Python

La librería Multiprocessing de Python es una biblioteca estándar que permite la creación de procesos en paralelo en una computadora con múltiples núcleos o CPUs. Proporciona una interfaz de programación para crear y administrar procesos de manera fácil y eficiente, lo que permite acelerar la ejecución de programas que realizan tareas intensivas en cómputo.

- El nombre del proceso se puede obtener utilizando la función `current_process()` de la librería `multiprocessing`.
- El nombre del proceso se puede asignar utilizando el argumento `name` de la función `Process()` de la librería `multiprocessing`.

```
import multiprocessing
import time

def foo():
    name = multiprocessing.current_process().name
    print ("Inicio del proceso llamado %s \n" %name)
    time.sleep(2)
    print ("Final del proceso llamado %s \n" %name)

if __name__ == '__main__':
    # Crear procesos
    process_with_default_name1 = multiprocessing.Process(target=foo)
    process_with_default_name2 = multiprocessing.Process(target=foo)
    process_with_name = multiprocessing.Process(name='MeNombraron', target=foo)

    # Ejecutar procesos
    process_with_name.start()
    process_with_default_name1.start()
    process_with_default_name2.start()

    # Esperar finalización de los procesos
    process_with_name.join()
    process_with_default_name1.join()
    process_with_default_name2.join()
```

```
Inicio del proceso llamado MeNombraron

Inicio del proceso llamado Process-4

Inicio del proceso llamado Process-5

Final del proceso llamado MeNombraron

Final del proceso llamado Process-4
Final del proceso llamado Process-5
```

# 3 - Multiprocessing.ipynb

## Contenido

- 3 - Multiprocessing.ipynb
- 3 - ¿Cómo ejecutar un proceso en segundo plano? - **Ejecutar Python File**

## Librería Multiprocessing de Python

La librería Multiprocessing de Python es una biblioteca estándar que permite la creación de procesos en paralelo en una computadora con múltiples núcleos o CPUs. Proporciona una interfaz de programación para crear y administrar procesos de manera fácil y eficiente, lo que permite acelerar la ejecución de programas que realizan tareas intensivas en cómputo.

- Los procesos en el modo NO\_background\_process tienen una salida, por lo que el proceso demoníaco finaliza automáticamente después de que finaliza el programa principal para evitar la persistencia de los procesos en ejecución.

```
import multiprocessing
import time

def foo():
    name = multiprocessing.current_process().name
    print ("Inicio del proceso llamado %s" %name)
    time.sleep(2)
    print ("Final del proceso llamado %s" %name)

if __name__ == '__main__':
    background_process = multiprocessing.Process(name='background_process', target=foo)
    background_process.daemon = True

    NO_background_process = multiprocessing.Process(name='NO_background_process', target=foo)
    NO_background_process.daemon = False

    background_process.start()
    NO_background_process.start()
    #background_process.join()
    #NO_background_process.join()
    print ("Final del principal")
```

```
Final del principal
Inicio del proceso llamado background_process
Inicio del proceso llamado NO_background_process
```

# 4 - Multiprocessing.ipynb

## Contenido

- 4 - Multiprocessing.ipynb
- 4 - ¿Cómo matar un proceso?

## Librería Multiprocessing de Python

La librería Multiprocessing de Python es una biblioteca estándar que permite la creación de procesos en paralelo en una computadora con múltiples núcleos o CPUs. Proporciona una interfaz de programación para crear y administrar procesos de manera fácil y eficiente, lo que permite acelerar la ejecución de programas que realizan tareas intensivas en cómputo.

- El método `is_alive()` del objeto `Process` devuelve `True` si el proceso está en ejecución.
- El proceso se puede matar utilizando el método `terminate()` del objeto `Process`.
- El método `terminate()` envía una señal al proceso.
- El método `exitcode` del objeto `Process` devuelve el código de salida del proceso.
- Código de salida del proceso:
  - «== 0»: Esto significa que no se produjo ningún error
  - «> 0»: Esto significa que el proceso tuvo un error y salió de ese código
  - «< 0»: Esto significa que el proceso se eliminó con una señal de `-1 * ExitCode`

```
import multiprocessing
import time

def foo():
    print('Inicio función')
    time.sleep(1)
    print('Final función')

if __name__ == '__main__':
    p = multiprocessing.Process(target=foo)
    print('Proceso antes ejecución:', p, p.is_alive())

    p.start()
    print('Proceso ejecutandose:', p, p.is_alive())
    #p.join()
    p.terminate()
    print('Proceso terminado:', p, p.is_alive())

    time.sleep(1)
    print('Proceso terminado 1 segundo:', p, p.is_alive())

    print('Código de salida del proceso:', p.exitcode)
```

# 5 - Multiprocessing.ipynb

## Contenido

- 5 - Multiprocessing.ipynb
- 5 - ¿Cómo usar un proceso en una subclase?

## Librería Multiprocessing de Python

La librería Multiprocessing de Python es una biblioteca estándar que permite la creación de procesos en paralelo en una computadora con múltiples núcleos o CPUs. Proporciona una interfaz de programación para crear y administrar procesos de manera fácil y eficiente, lo que permite acelerar la ejecución de programas que realizan tareas intensivas en cómputo.

- La librería multiprocessing de Python permite crear procesos utilizando subclases.
- Para ello, se crea una clase que hereda de la clase Process de la librería multiprocessing.
- En el ejemplo, se crea una clase MyProcess que hereda de la clase Process.
- La clase MyProcess sobrescribe el método run() de la clase Process.
- La clase MyProcess llama al método run() de la clase Process.
- Se crea una lista jobs que almacena los objetos MyProcess.
- Se crea un bucle for que crea 5 procesos, los almacena en la lista jobs, y los ejecuta.
- Se crea un bucle for que espera a que terminen los procesos almacenados en la lista jobs.

```
import multiprocessing

class MyProcess(multiprocessing.Process):

    def run(self):
        print ('Ejecución en %s \n' %self.name)
        return

    if __name__ == '__main__':
        jobs = []

        for i in range(5):
            p = MyProcess()
            jobs.append(p)
            p.start()

        for p in jobs:
            p.join()
```

Ejecución en MyProcess-56  
Ejecución en MyProcess-55

Ejecución en MyProcess-58  
Ejecución en MyProcess-57

Ejecución en MyProcess-59

# 6 - Multiprocessing.ipynb

## Contenido

- 6 - Multiprocessing.ipynb
- 6 - ¿Cómo intercambiar objetos entre procesos utilizando una cola?

## Librería Multiprocessing de Python

La librería Multiprocessing de Python es una biblioteca estándar que permite la creación de procesos en paralelo en una computadora con múltiples núcleos o CPUs. Proporciona una interfaz de programación para crear y administrar procesos de manera fácil y eficiente, lo que permite acelerar la ejecución de programas que realizan tareas intensivas en cómputo.

- La librería multiprocessing de Python permite intercambiar objetos entre procesos utilizando una cola.
- Para ello, se utiliza la clase Queue() de la librería multiprocessing.
- La clase Queue() recibe como argumento el tamaño máximo de la cola.
- La clase Queue() tiene los métodos put() y get() para añadir y obtener objetos de la cola.
- En el ejemplo, se crea una clase producer que hereda de la clase Process de la librería multiprocessing.
- La clase producer sobrescribe el método run() de la clase Process.
- La clase producer añade objetos a la cola utilizando el método put() de la clase Queue.
- La clase producer imprime el tamaño de la cola utilizando el método qsize() de la clase Queue.
- Se crea una clase consumer que hereda de la clase Process de la librería multiprocessing.
- La clase consumer sobrescribe el método run() de la clase Process.
- La clase consumer obtiene objetos de la cola utilizando el método get() de la clase Queue.
- La clase consumer imprime los objetos obtenidos de la cola.
- La clase consumer imprime un mensaje si la cola está vacía.

```
import multiprocessing
import random
import time

class producer(multiprocessing.Process):
    def __init__(self, queue):
        multiprocessing.Process.__init__(self)
        self.queue = queue

    def run(self):
        for i in range(5):
            item = random.randint(0, 256)
            self.queue.put(item)
            print("Proceso Producer: item %d agregado a la cola %s" % (item, self.queue))
            time.sleep(0.5)
            print("El tamaño de la cola es %s" % self.queue.qsize())

class consumer(multiprocessing.Process):
    def __init__(self, queue):
        multiprocessing.Process.__init__(self)
        self.queue = queue

    def run(self):
        while True:
            if (self.queue.empty()):
                print("La cola está vacía")
                break
            else:
                time.sleep(1)
                item = self.queue.get()
                print('Proceso Consumer: item %d sacado de %s \n' % (item, self.queue))
                time.sleep(0.5)

if __name__ == '__main__':
    queue = multiprocessing.Queue()
    process_producer = producer(queue)
    process_consumer = consumer(queue)
    process_producer.start()
    process_consumer.start()
    process_producer.join()
    process_consumer.join()
```

```
Proceso Producer: item 56 agregado a la cola producer-60
El tamaño de la cola es 1
Proceso Producer: item 23 agregado a la cola producer-60
Proceso Consumer: item 56 sacado de consumer-61

El tamaño de la cola es 1
Proceso Producer: item 162 agregado a la cola producer-60
El tamaño de la cola es 2
Proceso Producer: item 87 agregado a la cola producer-60
El tamaño de la cola es 3
Proceso Producer: item 140 agregado a la cola producer-60
Proceso Consumer: item 23 sacado de consumer-61

El tamaño de la cola es 3
Proceso Consumer: item 162 sacado de consumer-61

Proceso Consumer: item 87 sacado de consumer-61
Proceso Consumer: item 140 sacado de consumer-61

La cola está vacía
```

# 7 - Multiprocessing.ipynb

## Contenido

- 7 - Multiprocessing.ipynb
- 7 - ¿Cómo intercambiar objetos entre procesos utilizando una tubería?

## Librería Multiprocessing de Python

La librería Multiprocessing de Python es una biblioteca estándar que permite la creación de procesos en paralelo en una computadora con múltiples núcleos o CPUs. Proporciona una interfaz de programación para crear y administrar procesos de manera fácil y eficiente, lo que permite acelerar la ejecución de programas que realizan tareas intensivas en cómputo.

- La librería multiprocessing de Python permite intercambiar objetos entre procesos utilizando una tubería.
- Para ello, se utiliza la clase Pipe() de la librería multiprocessing.
- La clase Pipe() tiene los métodos send() y recv() para añadir y obtener objetos de la tubería.
- Después de iniciar los procesos y establecer las tuberías entre ellos, se cierran los extremos de lectura de pipe\_1 y pipe\_2 en el proceso principal usando pipe\_1[0].close() y pipe\_2[0].close().
- Esto se hace para indicar que el proceso principal ya no está utilizando los extremos de lectura de las tuberías y para que los procesos secundarios puedan continuar funcionando sin interrupciones.
- Luego, se utiliza un bucle try-except para recibir los objetos enviados por el proceso secundario a través de pipe\_2.
- El bucle continúa hasta que se produce un EOFError, lo que indica que ya no hay más objetos en la tubería.

- En cada iteración del bucle, se llama a pipe\_2[1].recv() para recibir el objeto enviado por el proceso secundario y se imprime en la consola usando print().
- En el código, el índice [0] se utiliza para hacer referencia al extremo de escritura y el índice [1] se utiliza para hacer referencia al extremo de lectura.

```
import multiprocessing

def create_items(pipe):
    output_pipe, _ = pipe
    for item in range(10):
        output_pipe.send(item)
    output_pipe.close()

def multiply_items(pipe_1, pipe_2):
    close, input_pipe = pipe_1
    close.close()
    output_pipe, _ = pipe_2
    try:
        while True:
            item = input_pipe.recv()
            output_pipe.send(item * item)
    except EOFError:
        output_pipe.close()

if __name__ == '__main__':
    pipe_1 = multiprocessing.Pipe(True)
    process_pipe_1 = multiprocessing.Process(target=create_items, args=(pipe_1))
    process_pipe_1.start()

    pipe_2 = multiprocessing.Pipe(True)
    process_pipe_2 = multiprocessing.Process(target=multiply_items, args=(pipe_1, pipe_2))
    process_pipe_2.start()

    pipe_1[0].close()
    pipe_2[0].close()

    try:
        while True:
            print(pipe_2[1].recv())
    except EOFError:
        print("Final de la comunicación")
```

```
0  
1  
4  
9  
16  
25  
36  
49  
64  
81
```

Final de la comunicación

# 8 - Multiprocessing.ipynb

## Contenido

- 8 - Multiprocessing.ipynb
- 8 - ¿Cómo sincronizar procesos? - **Ejecutar Python File**

## Librería Multiprocessing de Python

La librería Multiprocessing de Python es una biblioteca estándar que permite la creación de procesos en paralelo en una computadora con múltiples núcleos o CPUs. Proporciona una interfaz de programación para crear y administrar procesos de manera fácil y eficiente, lo que permite acelerar la ejecución de programas que realizan tareas intensivas en cómputo.

- La sincronización de procesos se puede lograr utilizando:
  - la clase Barrier de la librería multiprocessing.
  - la clase Lock de la librería multiprocessing.

```
import multiprocessing
from multiprocessing import Barrier, Lock, Process
from time import time
from datetime import datetime

def test_with_barrier(synchronizer, serializer):
    name = multiprocessing.current_process().name
    synchronizer.wait()
    now = time()

    with serializer:
        print("Proceso %s ----> %s \n" %(name,datetime.fromtimestamp(now)))

    #serializer.acquire()
    #print("Proceso %s ----> %s" %(name,datetime.fromtimestamp(now)))
    #serializer.release()

def test_without_barrier():
    name = multiprocessing.current_process().name
    now = time()
    print("Proceso %s ----> %s \n" %(name ,datetime.fromtimestamp(now)))

if __name__ == '__main__':
    synchronizer = Barrier(2)
    serializer = Lock()
    Process(name='p1 - prueba_con_barrera',target=test_with_barrier,args=(sync
    Process(name='p2 - prueba_con_barrera',target=test_with_barrier,args=(sync
    Process(name='p3 - prueba_sin_barrera',target=test_without_barrier).start(
    Process(name='p4 - prueba_sin_barrera',target=test_without_barrier).start(
```

Proceso p1 - prueba\_con\_barrera ----> 2023-05-07 16:41:32.844288

Proceso p2 - prueba\_con\_barrera ----> 2023-05-07 16:41:32.846685

Proceso p3 - prueba\_sin\_barrera ----> 2023-05-07 16:41:32.854381

Proceso p4 - prueba\_sin\_barrera ----> 2023-05-07 16:41:32.872981

# 9 - Multiprocessing.ipynb

## Contenido

- 9 - Multiprocessing.ipynb
- 9 - ¿Cómo sincronizar procesos utilizando un administrador de procesos (manager)? - **Ejecutar Python File**

## Librería Multiprocessing de Python

La librería Multiprocessing de Python es una biblioteca estándar que permite la creación de procesos en paralelo en una computadora con múltiples núcleos o CPUs. Proporciona una interfaz de programación para crear y administrar procesos de manera fácil y eficiente, lo que permite acelerar la ejecución de programas que realizan tareas intensivas en cómputo.

- La sincronización de procesos se puede lograr utilizando un administrador de procesos (manager).

```
import multiprocessing
import time

def worker(dictionary, key, item):
    print(key, item, "\n")
    time.sleep(2)
    dictionary[key] = item

if __name__ == '__main__':
    mgr = multiprocessing.Manager()
    dictionary = mgr.dict()
    jobs = [multiprocessing.Process(target=worker, args=(dictionary, i, i*2))

            start_time = time.time()
            for j in jobs:
                j.start()
            for j in jobs:
                j.join()

            end_time = time.time()
            print ('Results:', dictionary)

    print("El tiempo de ejecución fue:", end_time - start_time, "segundos")
```

0 01  
22

43

64  
8  
5

10  
6

127  
14  
8  
16

9 18

Results: {0: 0, 1: 2, 2: 4, 3: 6, 4: 8, 5: 10, 6: 12, 7: 14, 8: 16, 9: 18}  
El tiempo de ejecución fue: 2.2683022022247314 segundos

# 10 - Multiprocessing.ipynb

## Contenido

- 10 - Multiprocessing.ipynb
- 10 - ¿Cómo ejecutar funciones en paralelo utilizando procesos con Pool?

## Librería Multiprocessing de Python

La librería Multiprocessing de Python es una biblioteca estándar que permite la creación de procesos en paralelo en una computadora con múltiples núcleos o CPUs. Proporciona una interfaz de programación para crear y administrar procesos de manera fácil y eficiente, lo que permite acelerar la ejecución de programas que realizan tareas intensivas en cómputo.

- La función Pool() recibe como argumento el número de procesos que se desea crear.
- Para ejecutar la función, se utiliza el método map() del objeto Pool.
- Para esperar a que terminen la ejecución de las funciones, se utiliza el método close() y el método join() del objeto Pool.
- Cuando se crea la instancia de multiprocessing.Pool con processes=4, se están creando 4 procesos en paralelo para ejecutar la función function\_square en cada uno de ellos, utilizando los elementos de la lista inputs como entrada.
- Cada proceso recibe un subconjunto de los datos y ejecuta la función sobre ellos de manera independiente y concurrente con los otros procesos.
- Al final, se recopilan los resultados y se devuelven como una lista en el orden en que se completaron.

```
import multiprocessing

def function_square(data):
    result = data*data
    return result

if __name__ == '__main__':
    inputs = list(range(0,100))
    pool = multiprocessing.Pool(processes=4)
    pool_outputs = pool.map(function_square, inputs)

    pool.close()
    pool.join()
    print ('Pool      :', pool_outputs)
```

```
Pool      : [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 25
```

# Cargar una imagen en C

[Imprimir en PDF](#)

## Contenido

- Instalemos algunas librerías útiles.
- Ejemplo de como se deben escribir los programas en c para cargar imágenes
- Estrategia paralela de filtrado.
- Ahora con OpenMP

## Instalemos algunas librerías útiles.

```
!sudo apt-get update  
!sudo apt-get install libpng-dev  
!sudo apt-get install libjpeg-dev
```

0% [Working]

Hit:1 http://archive.ubuntu.com/ubuntu jammy InRelease

0% [Waiting for headers] [Connecting to security.ubuntu.com] [Waiting for heade

Get:2 https://cloud.r-project.org/bin/linux/ubuntu jammy-cran40/ InRelease [3,6:

0% [Waiting for headers] [Connecting to security.ubuntu.com] [Connecting to ppa

Get:3 https://developer.download.nvidia.com/compute/cuda/repos/ubuntu2204/x86\_6:

0% [Waiting for headers] [Connecting to security.ubuntu.com (91.189.91.83)] [Co  
0% [Waiting for headers] [Connecting to security.ubuntu.com (91.189.91.83)] [Co

Get:4 http://archive.ubuntu.com/ubuntu jammy-updates InRelease [119 kB]

Get:5 http://security.ubuntu.com/ubuntu jammy-security InRelease [110 kB]

Get:6 http://archive.ubuntu.com/ubuntu jammy-backports InRelease [109 kB]

Get:7 https://cloud.r-project.org/bin/linux/ubuntu jammy-cran40/ Packages [47.6

Get:8 https://developer.download.nvidia.com/compute/cuda/repos/ubuntu2204/x86\_6:

Get:9 http://archive.ubuntu.com/ubuntu jammy-updates/main amd64 Packages [1,535

Get:10 http://security.ubuntu.com/ubuntu jammy-security/restricted amd64 Package

Hit:11 https://ppa.launchpadcontent.net/c2d4u.team/c2d4u4.0+/ubuntu jammy InRele

Get:12 http://archive.ubuntu.com/ubuntu jammy-updates/restricted amd64 Packages

Get:13 http://archive.ubuntu.com/ubuntu jammy-updates/universe amd64 Packages [:

Get:14 http://security.ubuntu.com/ubuntu jammy-security/universe amd64 Packages

Get:15 http://security.ubuntu.com/ubuntu jammy-security/main amd64 Packages [1,:

Get:16 http://archive.ubuntu.com/ubuntu jammy-backports/universe amd64 Packages

Hit:17 https://ppa.launchpadcontent.net/deadsnakes/ppa/ubuntu jammy InRelease

Hit:18 https://ppa.launchpadcontent.net/graphics-drivers/ppa/ubuntu jammy InRele

Hit:19 https://ppa.launchpadcontent.net/ubuntugis/ppa/ubuntu jammy InRelease

Fetched 9,186 kB in 1s (8,125 kB/s)

Reading package lists... Done

Reading package lists... Done

Building dependency tree... Done

Reading state information... Done

libpng-dev is already the newest version (1.6.37-3build5).

0 upgraded, 0 newly installed, 0 to remove and 19 not upgraded.

Reading package lists... Done

Building dependency tree... Done

Reading state information... Done

libjpeg-dev is already the newest version (8c-2ubuntu10).

libjpeg-dev set to manually installed.

0 upgraded, 0 newly installed, 0 to remove and 19 not upgraded.

# Ejemplo de como se deben escribir los programas en c para cargar imágenes

Este código permite cargar cualquier tipo de imagen .PNG. Se le carga de manera dinámica la imagen en el momento de la ejecución por linea de comando.

```
%%writefile cargar_imagen.c
#include <png.h>
#include <stdio.h>
#include <stdlib.h>

// Función para leer la imagen PNG y obtener su tamaño
void read_png_file(char *filename, int *width, int *height) {
    FILE *file = fopen(filename, "rb");
    if (!file) abort();

    png_structp png = png_create_read_struct(PNG_LIBPNG_VER_STRING, NULL, NULL,
                                              NULL);
    if (!png) abort();

    png_infop info = png_create_info_struct(png);
    if (!info) abort();

    if (setjmp(png_jmpbuf(png))) abort();

    png_init_io(png, file);

    png_read_info(png, info);

    *width = png_get_image_width(png, info);
    *height = png_get_image_height(png, info);

    // Aquí podrías expandir el código para leer los datos de los píxeles si lo
    // necesitas
    fclose(file);
}

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "Uso: %s <ruta de la imagen>\n", argv[0]);
        return 1;
    }

    int width, height;
    read_png_file(argv[1], &width, &height);

    printf("Tamaño de la imagen: %d x %d\n", width, height);

    return 0;
}
```

Overwriting cargar\_imagen.c

Compilamos el archivo en C con las respectivas librerías.

```
!gcc cargar_imagen.c -o imagen -lpng
```

```
!ls
```

```
cargar_imagen.c  gato.png  imagen  imagen.jpg  sample_data
```

Ejecutamos el archivo que se genera del programa

y le pasamos como argumento la ruta de la imagen

```
!wget -O imagen.png https://w7.pngwing.com/pngs/36/440/png-transparent-homer-si  
!wget -O imagen.jpeg https://w7.pngwing.com/pngs/36/440/png-transparent-homer-s
```

```
--2023-11-30 10:38:28-- https://w7.pngwing.com/pngs/36/440/png-transparent-home-resolving-w7.pngwing.com(w7.pngwing.com)... 172.67.165.106, 104.21.73.185, 2600  
Resolving w7.pngwing.com (w7.pngwing.com)... 172.67.165.106, 104.21.73.185, 2600  
Connecting to w7.pngwing.com (w7.pngwing.com)|172.67.165.106|:443... connected.  
HTTP request sent, awaiting response... 200 OK  
Length: 33266 (32K) [image/png]  
Saving to: 'imagen.png'
```

```
imagen.png      0%[                ] 0  ---KB/s  
imagen.png 100%[=====] 32.49K  ---KB/s in 0.01s
```

```
2023-11-30 10:38:29 (2.95 MB/s) - 'imagen.png' saved [33266/33266]
```

```
--2023-11-30 10:38:29-- https://w7.pngwing.com/pngs/36/440/png-transparent-home-resolving-w7.pngwing.com(w7.pngwing.com)... 172.67.165.106, 104.21.73.185, 2600  
Resolving w7.pngwing.com (w7.pngwing.com)... 172.67.165.106, 104.21.73.185, 2600  
Connecting to w7.pngwing.com (w7.pngwing.com)|172.67.165.106|:443... connected.  
HTTP request sent, awaiting response... 200 OK  
Length: 33266 (32K) [image/png]  
Saving to: 'imagen.jpeg'
```

```
imagen.jpeg 100%[=====] 32.49K  ---KB/s in 0.01s
```

```
2023-11-30 10:38:29 (2.77 MB/s) - 'imagen.jpeg' saved [33266/33266]
```

```
!./imagen "/content/imagen.png"
```

```
Tamaño de la imagen: 920 x 719
```

## Estrategia paralela de filtrado.

Para este vamos a aplicarle el filtro de sobel (vertical y horizontal) a la imagen y usando una estrategia paralela.

Primero veamos como es el código de manera secuencial para este caso

```

%%writefile filtro_image.c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#define STB_IMAGE_IMPLEMENTATION //Librerias para cargar imagen
#include "stb_image.h"
#define STB_IMAGE_WRITE_IMPLEMENTATION //libreria para guardar imagenes.
#include "stb_image_write.h"

void aplicar_filtro_bordes(unsigned char *input, unsigned char *output, int width, int height) {
    int kernel_x[3][3] = {{-1, 0, 1}, {-2, 0, 2}, {-1, 0, 1}};
    int kernel_y[3][3] = {{1, 2, 1}, {0, 0, 0}, {-1, -2, -1}};

    for (int y = 1; y < height - 1; y++) {
        for (int x = 1; x < width - 1; x++) {
            float gx = 0, gy = 0;
            for (int ky = -1; ky <= 1; ky++) {
                for (int kx = -1; kx <= 1; kx++) {
                    int p = input[(y + ky) * width + (x + kx)];
                    gx += p * kernel_x[ky + 1][kx + 1];
                    gy += p * kernel_y[ky + 1][kx + 1];
                }
            }
            int magnitude = (int)sqrt(gx * gx + gy * gy);
            magnitude = magnitude > 255 ? 255 : magnitude;
            output[y * width + x] = (unsigned char)magnitude;
        }
    }
}

int main() {
    int width, height, channels;
    unsigned char *img = stbi_load("imagen.jpeg", &width, &height, &channels, 1);
    if (img == NULL) {
        printf("Error al cargar la imagen\n");
        return 1;
    }

    unsigned char *output_img = malloc(width * height * sizeof(unsigned char));
    if (output_img == NULL) {
        printf("No se pudo asignar memoria para la imagen de salida\n");
        stbi_image_free(img);
        return 1;
    }

    clock_t start, end;
    double cpu_time_used;

    start = clock();
    aplicar_filtro_bordes(img, output_img, width, height, channels);
    end = clock();
    cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;

    stbi_write_jpg("imagen_con_bordes.jpg", width, height, 1, output_img, 100);
}

```

```
    printf("Tiempo de ejecución: %f segundos\n", cpu_time_used);

    stbi_image_free(img);
    free(output_img);

    return 0;
}
```

Writing filtro\_image.c

## Descargamos los archivos de las librerías externas

```
!wget https://raw.githubusercontent.com/nothings/stb/master/stb_image.h
!wget https://raw.githubusercontent.com/nothings/stb/master/stb_image_write.h
```

```
--2023-11-30 10:38:46-- https://raw.githubusercontent.com/nothings/stb/master/
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 185.199.109.1
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|185.199.109.1|:443...
HTTP request sent, awaiting response... 200 OK
Length: 284733 (278K) [text/plain]
Saving to: 'stb_image.h'


```

```
stb_image.h      0%[                    ]          0  --.-KB/s
stb_image.h      100%[=====] 278.06K  --.-KB/s    in 0.04s
```

```
2023-11-30 10:38:46 (6.97 MB/s) - 'stb_image.h' saved [284733/284733]
```

```
--2023-11-30 10:38:46-- https://raw.githubusercontent.com/nothings/stb/master/
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 185.199.109.1
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|185.199.109.1|:443...
HTTP request sent, awaiting response... 200 OK
Length: 71221 (70K) [text/plain]
Saving to: 'stb_image_write.h'


```

```
stb_image_write.h 100%[=====] 69.55K  --.-KB/s    in 0.02s
```

```
2023-11-30 10:38:47 (3.00 MB/s) - 'stb_image_write.h' saved [71221/71221]
```

## Verificamos que se descarguen los archivos.

```
!ls -l
```

```
total 456
-rw-r--r-- 1 root root 1034 Nov 30 10:30 cargar_imagen.c
-rw-r--r-- 1 root root 2028 Nov 30 10:38 filtro_image.c
-rwxr-xr-x 1 root root 16696 Nov 30 10:27 imagen
-rw-r--r-- 1 root root 33266 Feb 18 2020 imagen.jpeg
-rw-r--r-- 1 root root 33266 Feb 18 2020 imagen.png
drwxr-xr-x 1 root root 4096 Nov 28 14:27 sample_data
-rw-r--r-- 1 root root 284733 Nov 30 10:38 stb_image.h
-rw-r--r-- 1 root root 71221 Nov 30 10:38 stb_image_write.h
```

Se compila el programa con las respectivas banderas para tomar las librerías

-lm es para que compile la librería de funciones matemática como sqrt()

```
!gcc filtro_image.c -o filtro_image -lm
```

```
!./filtro_image
```

Tiempo de ejecución: 0.039270 segundos

Ahora con OpenMP

Descargamos las librerías útiles.

```
!sudo apt-get update
!sudo apt-get install build-essential
```

```
0% [Working]
```

```
Hit:1 https://cloud.r-project.org/bin/linux/ubuntu jammy-cran40/ InRelease  
0% [Connecting to archive.ubuntu.com (91.189.91.81)] [Waiting for headers] [Con  
Hit:2 https://developer.download.nvidia.com/compute/cuda/repos/ubuntu2204/x86_6  
0% [Waiting for headers] [Waiting for headers] [Connected to ppa.launchpadconte  
Hit:3 http://archive.ubuntu.com/ubuntu jammy InRelease  
Hit:4 http://archive.ubuntu.com/ubuntu jammy-updates InRelease  
Get:5 http://security.ubuntu.com/ubuntu jammy-security InRelease [110 kB]  
Hit:6 http://archive.ubuntu.com/ubuntu jammy-backports InRelease  
Hit:7 https://ppa.launchpadcontent.net/c2d4u.team/c2d4u.0+/ubuntu jammy InRele  
Hit:8 https://ppa.launchpadcontent.net/deadsnakes/ppa/ubuntu jammy InRelease  
Hit:9 https://ppa.launchpadcontent.net/graphics-drivers/ppa/ubuntu jammy InRele  
Hit:10 https://ppa.launchpadcontent.net/ubuntugis/ppa/ubuntu jammy InRelease  
Get:11 http://security.ubuntu.com/ubuntu jammy-security/main amd64 Packages [1,  
Get:12 http://security.ubuntu.com/ubuntu jammy-security/restricted amd64 Package  
Fetched 2,868 kB in 2s (1,758 kB/s)  
Reading package lists... Done  
Reading package lists... Done  
Building dependency tree... Done  
Reading state information... Done  
build-essential is already the newest version (12.9ubuntu3).  
0 upgraded, 0 newly installed, 0 to remove and 17 not upgraded.
```

## Escribimos el programa usando openMP.

Note que la diferencia respecto al secuencial es la línea:

```
#pragma omp parallel for collapse(2)
```

1. **#pragma omp parallel for** es una directiva de OpenMP que se utiliza para paralelizar un bucle for. Hace que el bucle se ejecute en varios hilos en paralelo, lo que puede acelerar significativamente el procesamiento si tienes un procesador de múltiples núcleos.
2. **collapse(2)**: Este modificador especifica que los dos bucles for anidados más cercanos (en este caso, los bucles que iteran sobre **y** y **x**) deben colapsarse en un solo bucle iterativo para fines de paralelización. Esto es útil cuando cada bucle por sí solo no proporciona suficientes iteraciones para aprovechar completamente las capacidades de paralelización, pero combinados, ofrecen un número significativo de tareas independientes que pueden ser distribuidas entre los hilos disponibles.

Verifique que el bucle externo itera a través de las filas (y) de la imagen y el bucle interno a través de las columnas (x). Al usar collapse(2), se trata la combinación de estas dos dimensiones como un único conjunto de iteraciones, lo que permite una distribución más eficiente de las iteraciones entre los hilos de OpenMP.

```

%%writefile tu_programa.c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <omp.h>
#define STB_IMAGE_IMPLEMENTATION
#include "stb_image.h"
#define STB_IMAGE_WRITE_IMPLEMENTATION
#include "stb_image_write.h"

void aplicar_filtro_bordes(unsigned char *input, unsigned char *output, int width, int height, int channels) {
    int kernel_x[3][3] = {{-1, 0, 1}, {-2, 0, 2}, {-1, 0, 1}};
    int kernel_y[3][3] = {{1, 2, 1}, {0, 0, 0}, {-1, -2, -1}};

    #pragma omp parallel for collapse(2)
    for (int y = 1; y < height - 1; y++) {
        for (int x = 1; x < width - 1; x++) {
            float gx = 0, gy = 0;
            for (int ky = -1; ky <= 1; ky++) {
                for (int kx = -1; kx <= 1; kx++) {
                    int p = input[(y + ky) * width + (x + kx)];
                    gx += p * kernel_x[ky + 1][kx + 1];
                    gy += p * kernel_y[ky + 1][kx + 1];
                }
            }
            int magnitude = (int)sqrt(gx * gx + gy * gy);
            magnitude = magnitude > 255 ? 255 : magnitude;
            output[y * width + x] = (unsigned char)magnitude;
        }
    }
}

int main() {
    int width, height, channels;
    unsigned char *img = stbi_load("imagen.jpeg", &width, &height, &channels, 1);
    if (img == NULL) {
        printf("Error al cargar la imagen\n");
        return 1;
    }

    unsigned char *output_img = malloc(width * height * sizeof(unsigned char));
    if (output_img == NULL) {
        printf("No se pudo asignar memoria para la imagen de salida\n");
        stbi_image_free(img);
        return 1;
    }

    clock_t start, end;
    double cpu_time_used;

    start = clock();
    aplicar_filtro_bordes(img, output_img, width, height, channels);
    end = clock();
    cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
}

```

```
    stbi_write_jpg("imagen_con_bordes.jpg", width, height, 1, output_img, 100);

    printf("Tiempo de ejecución: %f segundos\n", cpu_time_used);

    stbi_image_free(img);
    free(output_img);

    return 0;
}
```

Overwriting tu\_programa.c

```
!gcc tu_programa.c -o tu_programa -fopenmp -lm
```

```
!./tu_programa
```

Tiempo de ejecución: 0.060102 segundos

# Ejercicio de procesamiento de imágenes en paralelo

## Contenido

- Librerías para imágenes
- Funciones útiles para cargar y guardar imágenes.
- Carguemos una imagen
- Veamos como se hace un filtro a pedal (sin estrategias paralelas)
- Apliquemos el filtro a la imagen
- Ahora revisemos una estrategia paralela. Multiprocessing

## Librerías para imágenes

```
!pip install Pillow numpy
```

```
Requirement already satisfied: Pillow in /usr/local/lib/python3.10/dist-packages  
Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-packages
```

## Funciones útiles para cargar y guardar imágenes.

```
import numpy as np
from PIL import Image
import time
from IPython.display import display # Importar para visualización

def cargar_imagen(ruta):
    return Image.open(ruta).convert('L')

def guardar_imagen(imagen, ruta):
    imagen.save(ruta)
```

## Carguemos una imagen

```
!wget -O imagen.jpg https://previews.123rf.com/images/aprillrain/aprillrain22110001/123RF22110001_100000000-Image-of-a-spring-rainfall-with-drops-falling-downward-.jpg
!ls
```

```
--2023-11-29 22:37:07-- https://previews.123rf.com/images/aprillrain/aprillrain22110001/123RF22110001_100000000-Image-of-a-spring-rainfall-with-drops-falling-downward-.jpg
Resolving previews.123rf.com (previews.123rf.com)... 18.66.255.80, 18.66.255.37
Connecting to previews.123rf.com (previews.123rf.com)|18.66.255.80|:443... connected
HTTP request sent, awaiting response... 200 OK
Length: 323060 (315K) [image/jpeg]
Saving to: 'imagen.jpg'
```

```
  0%[                                         ] 0      --.-KB/s
  100%[=====] 315.49K  --.-KB/s    in 0.08s
```

```
2023-11-29 22:37:07 (3.86 MB/s) - 'imagen.jpg' saved [323060/323060]
```

```
imagen_con_bordes.jpg  imagen.jpg  sample_data
```

```
path = "/content/imagen.jpg"
img = cargar_imagen(path)
img
```



## Veamos como se hace un filtro a pedal (sin estrategias paralelas)

Los «**kernel**s», también conocidos como matrices de convolución o filtros, son fundamentales en el procesamiento de imágenes y la visión por computadora. **Son matrices pequeñas** que se utilizan para aplicar efectos como desenfoque, nitidez, detección de bordes, entre otros, a una imagen.

# ¿Qué es un Kernel?

Un kernel es una matriz de  $N \times N$

(donde N es usualmente un número impar como 3, 5, 7, etc.) que se aplica a cada píxel y sus vecinos en una imagen. Cada elemento de un kernel está asociado con un píxel vecino. Al multiplicar estos elementos con los píxeles vecinos y sumarlos, se obtiene un nuevo valor para el píxel original. Este proceso se repite para cada píxel de la imagen.

## ¿Cómo Funcionan los Kernels en el Filtrado de Imágenes?

### Desplazamiento sobre la Imagen:

El kernel se «desliza» sobre la imagen, posicionándose sucesivamente sobre cada píxel de la imagen.

En cada posición, los píxeles debajo del kernel se corresponden con los elementos de la matriz del kernel.

### Operación de Convolución:

Para cada píxel bajo el kernel, se multiplica su valor por el valor correspondiente en el kernel. Luego, se suman todos estos productos. Este total es el nuevo valor del píxel central en la imagen filtrada.

### Normalización (Opcional):

En algunos casos, los valores del kernel se normalizan (por ejemplo, asegurándose de que sumen 1) para mantener el rango de intensidad de la imagen.

```
def aplicar_filtro_bordes(imagen):
    # Convertir imagen a un array de numpy
    pixels = np.array(imagen)

    # Kernel de Sobel para bordes verticales
    kernel = np.array([[1, 2, 1], [0, 0, 0], [-1, -2, -1]])

    # Preparar el array de salida
    bordes = np.zeros_like(pixels)

    # Aplicar el filtro de Sobel solo en la dirección vertical
    for i in range(1, pixels.shape[0]-1):
        for j in range(1, pixels.shape[1]-1):
            gy = np.sum(np.multiply(pixels[i-1:i+2, j-1:j+2], kernel))
            bordes[i, j] = min(255, np.abs(gy))

    return Image.fromarray(bordes)
```

## Apliquemos el filtro a la imagen

```
imagen = cargar_imagen(path)

# Cargar y aplicar filtro
inicio = time.time()
imagen_bordes = aplicar_filtro_bordes(imagen)
fin = time.time()

# Imprimir tiempo de ejecución
print(f"Tiempo de ejecución: {fin - inicio} segundos")

# Visualizar la imagen
display(imagen_bordes)
```

Tiempo de ejecución: 16.858123302459717 segundos



```
# Guardar imagen resultante
path_bordes = 'imagen_con_bordes.jpg'
guardar_imagen(imagen_bordes, path_bordes)
```

## Ahora revisemos una estrategia paralela. Multiprocessing

```
import numpy as np
from PIL import Image
import time
from IPython.display import display
from multiprocessing import Pool, cpu_count
```

## Cambiemos un poco la lógica

```
def aplicar_filtro_seccion(args):
    pixels, inicio, fin, ancho = args
    # Kernel de Sobel para bordes verticales
    kernel_y = np.array([[1, 2, 1], [0, 0, 0], [-1, -2, -1]])

    bordes_seccion = np.zeros_like(pixels[inicio:fin, :]) ## Ahora lo que va a

    for i in range(1, fin - inicio - 1):
        for j in range(1, ancho - 1):
            gy = np.sum(np.multiply(pixels[i-1+inicio:i+2+inicio, j-1:j+2], ke
            bordes_seccion[i, j] = min(255, np.abs(gy))

    return bordes_seccion
```

```
# Esta función se encarga de gestionar los procesos y los "pedazos" de imagen

def aplicar_filtro_bordes_multiprocessing(imagen):
    pixels = np.array(imagen)
    alto, ancho = pixels.shape
    num_procesos = cpu_count()
    print("numero de procesos", num_procesos)

    # Dividir la imagen en secciones con superposición
    seccion_alto = alto // num_procesos
    secciones = []
    for i in range(num_procesos):
        inicio = i * seccion_alto
        fin = (i + 1) * seccion_alto if i != num_procesos - 1 else alto
        if i != 0:
            inicio -= 1 # Superposición para cubrir bordes
        secciones.append((pixels, inicio, fin, ancho))

    # Crear un pool de procesos y aplicar el filtro a cada sección
    with Pool() as pool:
        resultados = pool.map(aplicar_filtro_seccion, secciones)

    # Combinar los resultados
    bordes = np.vstack(resultados)

    return Image.fromarray(bordes)
```

```
# Cargar y aplicar filtro
imagen = Image.open(path).convert('L')

inicio = time.time()
imagen_bordes = aplicar_filtro_bordes_multiprocessing(imagen)
fin = time.time()

# Guardar imagen resultante
ruta_imagen_bordes = 'imagen_con_bordes.jpg'
imagen_bordes.save(ruta_imagen_bordes)

# Imprimir tiempo de ejecución
print(f"Tiempo de ejecución: {fin - inicio} segundos")

# Visualizar la imagen
display(imagen_bordes)
```

```
numero de procesos 2
Tiempo de ejecución: 17.785150051116943 segundos
```

