Unidad 6
Concurrencia y Paralelismo

Prof: Lic. Rosales Pablo (prosales@unpata.edu.ar)
Materia: Programación Orientada a Objetos
Año: 2024

Problemas

Rendimiento: tareas lentas y de mucho consumo de procesamiento.

- **Simulaciones**: clima, mecánica de fluidos, dinámicas moleculares, muchas ecuaciones matemáticas sobre datasets históricos.
- **Procesamiento de imágenes y videos:** procesamiento en cada pixel de cálculos complejos. (renderizados)
- **Análisis de datos**: modelos de aprendizaje automático (grandes datasets), clustering, procesamiento de lenguaje natural.

Mejora en capacidad de respuesta: todas las tareas de un Sistema Operativo (SO) deben de mantenerse receptivas a las peticiones del usuario.

Problemas

Recursos limitados: el tiempo de ejecución en CPU y la memoria es limitado y debe ser aprovechado de manera eficiente.

Bloqueos: el sistema debe poder continuar con ejecuciones aun cuando se estén esperando por búsquedas a discos externos o por entradas de usuario o peticiones por internet.

Escalabilidad: problemas que aumentan en tamaño pueden llegar a ser fácilmente escalables con la incorporación de más HW y software paralelizable.

Proceso

Es un programa en ejecución. Tiene su propio espacio en memoria con su código ejecutable, variables, la pila de llamadas a funciones,etc..

Son independientes y aislados de otros procesos, no comparten espacio en memoria.

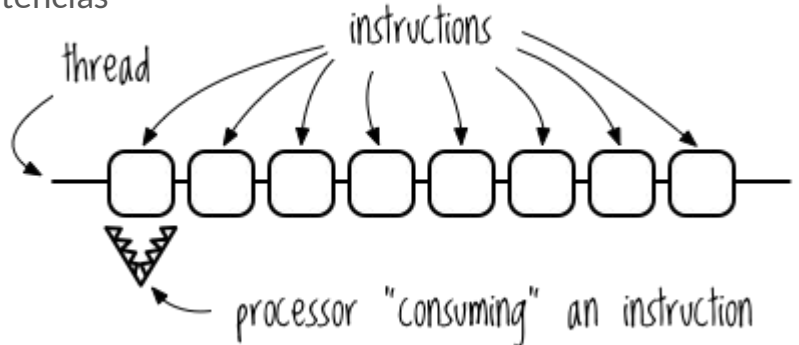
Para qué procesos se comuniquen entre si es necesario implementar procesos complejos como IPC, que involucran el manejo de sockets, cola de mensajes, etc..

Hilo (Thread)

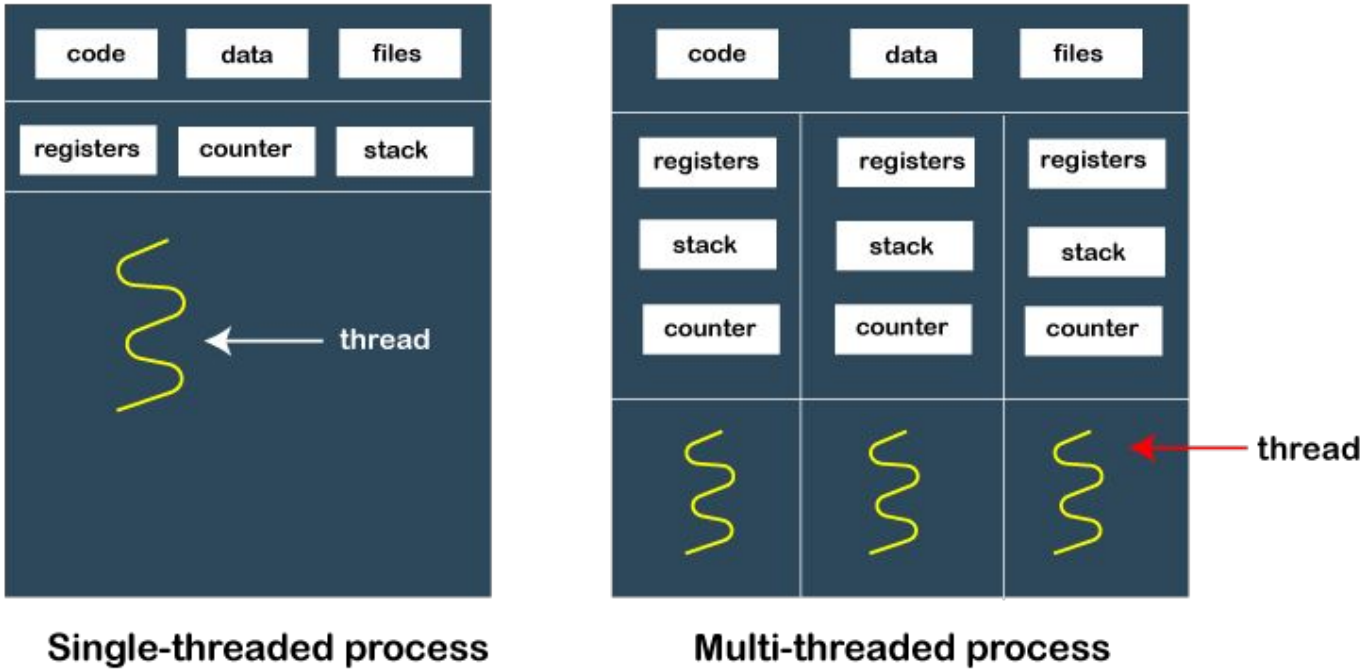
Unidad pequeña de ejecución dentro de un proceso.

Comparten espacio de memoria con otros hilos del mismo proceso, pueden acceder a variables y recursos compartidos.

La comunicación entre los hilos se logra fácilmente al compartir memoria, pero se tiene que tener especial cuidado al acceder a esta, se generan inconsistencias



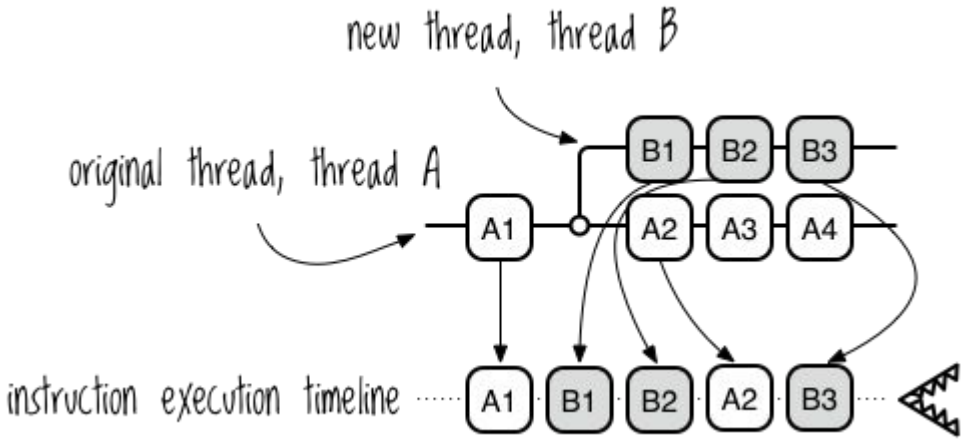
Proceso vs Hilo



Concurrencia

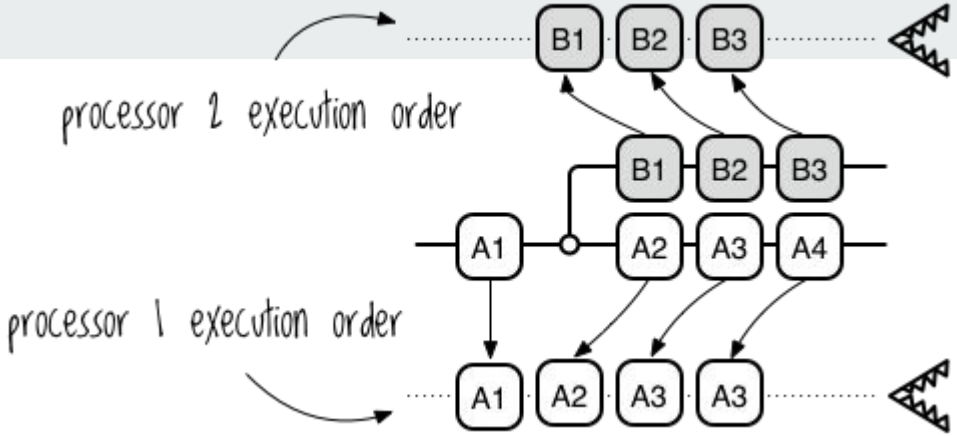
La concurrencia es la ejecución coordinada de procesos para simular una ejecución al mismo tiempo en el SO.

Varias tareas se inician y ejecutan compartiendo un tiempo de ejecución asignado por el SO para entrar al procesador. La entrada y salida de cada tarea es tan rápida que para nuestra percepción es simultánea.



fuente: https://www.braveclojure.com/concurrency/

Paralelismo



La posibilidad de poder ejecutar tareas o procesos al mismo tiempo, en lugar de ejecutarlo secuencialmente. Permite aprovechar al máximo los recursos de HW mejorando la velocidad y eficiencia de los programas.

Se puede paralelizar en función de los datos, dividir un conjunto grande de datos entre varios procesos o hilos para ser procesados paralelamente y así terminar el trabajo más rápido.

Se puede paralelizar por tareas, cuando existe cierta independencia de subtareas. Por ejemplo un servidor manejando diferentes solicitudes de clientes.

Import threading

```python
import threading

def imprimir_numeros():
    for i in range(1, 100):
        print(f'Número {i}')
    print("termina ejecución de imprimir_numeros")

def imprimir_letras():
    letras = 'ABCDEEFGHIJKLMNÑOPQRSTUVWXYZ'
    for letra in letras*5:
        print(f'Letra {letra}')
    print("termina ejecución de imprimir_letras")

thread1 = threading.Thread(target=imprimir_numeros)
thread2 = threading.Thread(target=imprimir_letras)

thread1.start()
thread2.start()

thread1.join()
thread2.join()
print("termina ejecución de programa")
```

```
Letra Ñ
Número 96
Letra O
Número 97
Letra P
Número 98
Letra Q
Número 99
Letra R
Letra S
Letra T
Letra U
termina ejecución de imprimir_numeros
Letra V
Letra W
Letra X
Letra Y
Letra Z
termina ejecución de imprimir_letras
termina ejecución de programa
```

Enfoque OO

```python
from threading import Thread

class Letras(Thread):

    def run(self):
        letras = 'ABCDEEFGHIJKLMNÑOPQRSTUVWXYZ'
        for letra in letras:
            print(self.name+" Letra: "+letra)
        print(self.name+" termina ejecución de imprimir_letras")

class Numeros(Thread):

    def run(self):
        for i in range(1, 10):
            print(self.name+" Número: "+str(i))
        print(self.name+" termina ejecución de imprimir_numeros")

for i in range(0,5):
    Letras().start()
    Numeros().start()

# que sucede si saco los join?
print("termina ejecución de programa")
```

Particularidades de Python

Compartiendo memoria

Global Interpreter Lock (GIL) : es un bloqueo que garantiza que solo un hilo de Python ejecute código de Python puro (es decir, código no compilado) a la vez en un proceso. Esto significa que, en sistemas con múltiples núcleos de CPU, Python **NO** puede aprovechar completamente esos núcleos para ejecutar hilos de Python en paralelo.

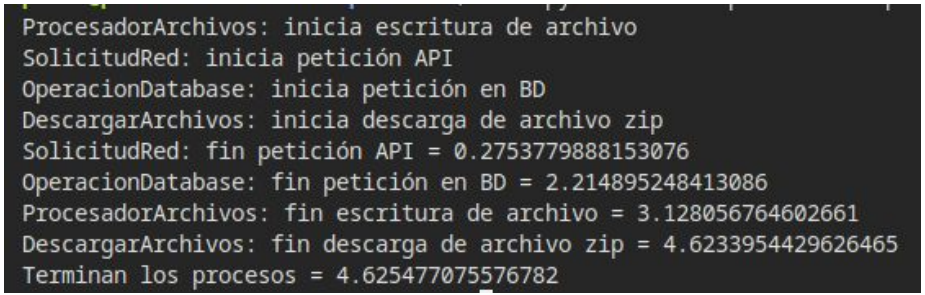
```python
from threading import Thread

contador = 0
class Numeros(Thread):

    def run(self):
        global contador
        for i in range(10000000):
            contador += 1

        print(self.name+" termina ejecución ")

hilos = []
for i in range(5):
    un_hilo = Numeros()
    hilos.append(un_hilo)
    un_hilo.start()

for hilo in hilos:
    hilo.join()
print("termina ejecución de programa")
print("Resultado contador:", contador)
```

Entonces.. ¿Para qué puedo usar Threading en CPython?

El GIL si puede liberar el CPU cuando un hilo está esperando por una operación de E/S, permitiendo que otros hilos tomen el control y continúan su ejecución. Las E/S intensivas pueden ser:

- **Lectura y escritura de archivos grandes.**
- **Solicitudes a través de la red.**
- **Operaciones de Base de Datos.**
- **Descarga de archivos grandes.**

ver ejemplo operaciones_es_intensivas.py



import multiprocessing ...

Sintaxis de instrucciones muy similar a threading.

Esta librería crea **procesos** independientes por lo cual el GIL no causa problemas y si se logra un verdadero paralelismo.

```
Hilo main
Nombre de modulo: __main__
Proceso padre: 6128
ID de proceso: 6252
Hilo MiProceso
Nombre de modulo: __mp_main__
Proceso padre: 6252
ID de proceso: 12720
Duermo el hilo
termina hilo: 0.5087063000537455
```

```python
from multiprocessing import Process
import time
import os

class MiProceso(Process):

    def run(self):
        inicio = time.perf_counter()
        info('Hilo MiProceso')
        print("Duermo el hilo")
        time.sleep(0.5)
        fin= time.perf_counter()
        print("termina hilo: "+ str(fin-inicio))


def info(title):
    print(title)
    print('Nombre de modulo:', __name__)
    print('Proceso padre:', os.getppid())
    print('ID de proceso:', os.getpid())

if __name__ == '__main__':
    info('Hilo main')
    p = MiProceso()
    p.start()
    p.join()
```

Compartir datos entre procesos Queue

Para compartir datos de forma controlada entre procesos se usa la estructura de datos Queue, dentro de la librería multiprocessing.

El funcionamiento es el mismo que Queue normal.

```
Proceso Consumidor recibe: Dato 1
Proceso Consumidor recibe: Dato 2
Proceso Consumidor recibe: Dato 3
Proceso Consumidor recibe: Dato 4
Proceso Consumidor recibe: Dato 5
fin de los procesos
```

```python
from multiprocessing import Process, Queue

class ProcesoProductor(Process):
    def __init__(self, cola):
        super().__init__()
        self.cola = cola

    def run(self):
        for item in range(1, 6):
            self.cola.put(f'Dato {item}')

class ProcesoConsumidor(Process):
    def __init__(self, cola):
        #super(ProcesoConsumidor,self).__init__()
        #Process.__init__(self)
        super().__init__()
        self.cola = cola

    def run(self):
        while not self.cola.empty():
            dato = self.cola.get()
            print(f'Proceso Consumidor recibe: {dato}')

if __name__ == '__main__':
    cola_compartida = Queue()

    proceso_productor = ProcesoProductor(cola_compartida)
    proceso_consumidor = ProcesoConsumidor(cola_compartida)

    proceso_productor.start()
    proceso_consumidor.start()

    proceso_productor.join()
    proceso_consumidor.join()

    print("fin de los procesos")
```

Compartir datos entre procesos Pipe

Pipe genera un canal por el cual los procesos se pueden comunicar.

El objeto Pipe tiene los métodos send() y recv().

Cuando se llama a Pipe(), devuelve dos objetos, uno va al emisor y el otro al receptor.

```
Enviando: Hola
Enviando: Alumnos
Enviando: de
Enviando: POO
Recibido: Hola
Recibido: Alumnos
Recibido: de
Recibido: POO
fin de la ejecución
```

```python
from  multiprocessing import Process, Pipe

class ProcesoEnvio(Process):
    def __init__(self, canal):
        super(ProcesoEnvio, self).__init__()
        self.canal = canal

    def run(self):
        mensajes = ["Hola", "Alumnos","de", "POO"]
        for mensaje in mensajes:
            print(f"Enviando: {mensaje}")
            self.canal.send(mensaje)
        self.canal.send(None)

class ProcesoRecepcion(Process):
    def __init__(self, canal):
        super(ProcesoRecepcion, self).__init__()
        self.canal = canal

    def run(self):
        mensaje = ""
        while mensaje is not None:
            mensaje = self.canal.recv()
            if mensaje is not None:
                print(f"Recibido: {mensaje}")

if __name__ == '__main__':
    pipe_conn1, pipe_conn2 = Pipe() #devuelve dos pipes siempre

    proceso_envio = ProcesoEnvio(pipe_conn1)
    proceso_recepcion = ProcesoRecepcion(pipe_conn2)

    proceso_envio.start()
    proceso_recepcion.start()

    proceso_envio.join()
    proceso_recepcion.join()
    print("fin de la ejecución")
```

Locks

Los locks son bloqueos para asegurar que el hilo termine la ejecución de una porcion de codigo de forma atómica. Esto es importante para para asegurar la exclusión mutua entre los procesos al acceso a un recurso compartido.

Ver código de ejemplo

Pool

Se utilizan para crear grupos de procesos que pueden ejecutar una función en paralelo.

Por lo general se hace sobre una función.

```python
import multiprocessing

def calcular_cuadrado(numero):
    return numero ** 2

if __name__ == '__main__':
    num_procesadores = multiprocessing.cpu_count()
    print("numero de procesadores disponibles:", num_procesadores)
    datos = []
    for i in range(num_procesadores):
        datos.append(i)

    with multiprocessing.Pool(processes=num_procesadores) as pool:
        resultados = pool.map(calcular_cuadrado, datos)

    print("Resultados finales:", resultados)
```

```
numero de procesadores disponibles: 20
Resultados finales: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256, 289, 324, 361]
```

Daemon Process

Un Daemon Thread o Daemon Process se ejecutan en background. Dan soporte a los procesos que no son daemons, sirven para tareas del tipo:

Esporádico: tarea ejecutada bajo ciertas condiciones (ej: log a archivo o base de datos)

Periodico: tarea ejecutada cada cierto tiempo (ej: guardar datos cada x cantidad de minutos)

Larga ejecución: tarea que se ejecuta durante la vida del programa (ej: monitoreo de un recurso)

Daemon Process

Un Daemon no tiene control de cuando finaliza su ejecución, cuando el padre finaliza, todos los deamon hijos terminan. En cambio si tuviera hijos no daemon, el proceso main no podría finalizar.

Esto quiere decir que las tareas que hace un daemon tiene que ser no cruciales para la ejecución, tienen que poder finalizar de forma robusta (pj: cerrar todos los recursos como archivos y conexiones a BD)

Para iniciar un proceso o hilo como demonio usar -->

```
Proceso(daemon=True).start()
Proceso().start()
```

Problemas con la sincronización (mal realizada)

Race conditions (Condiciones de carrera): cuando múltiples hilos o procesos intentan acceder y modificar datos compartidos al mismo tiempo. Esto lleva a problemas de lectura y escritura e inconsistencia de datos.

Starvation (Inanición): cuando un hilo o proceso queda bloqueado de forma permanente.

Deadlock (Interbloqueo): cuando dos o más hilos/procesos se bloquean entre si por la espera de recursos retenidos por otros hilos/procesos.

Fuentes útiles:

Dusty Phillips - Python 3 Object-oriented Programming-Packt Publishing (2015) capitulo 13

https://superfastpython.com/learning-paths/