Actividad 7:

Ejercicios sobre comunicación entre procesos

Cómputo de Alto Rendimiento

**Luis Fernando Izquierdo Berdugo**

# Análisis del Código 1

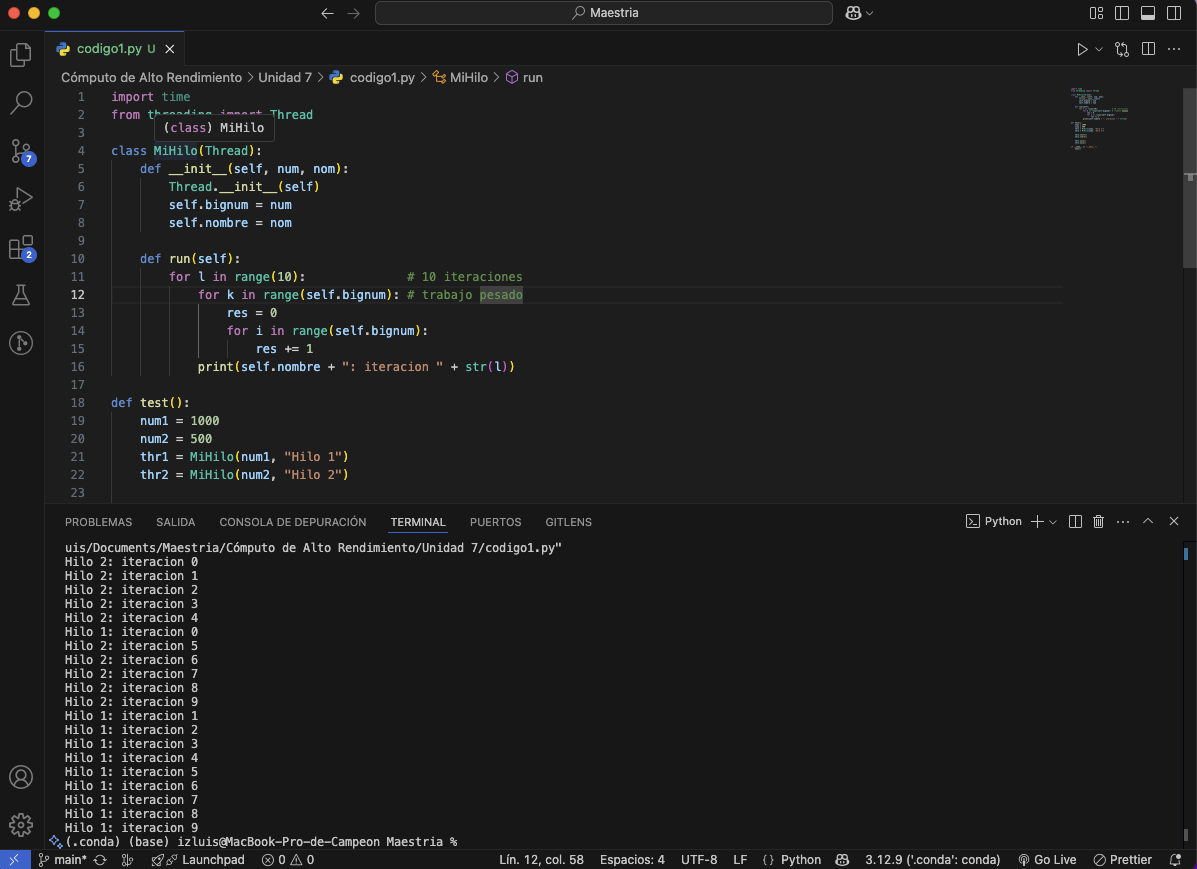
Este código crea dos hilos concurrentes, donde cada uno ejecuta una carga de trabajo definida por un número “bignum” y muestra mensajes durante 10 iteraciones. La clase “MiHilo”:

* hereda de “Thread”,
* tiene la función “\_\_init\_\_” donde se inicializa la clase base, el número de operaciones a hacer y el nombre del hilo,
* tiene la función “run”, donde se hacen 10 iteraciones de impresión del nombre del Hilo y el número de iteración.

Al ejecutar la función “test”:

* Se definen la variables “num1” y “num2”
* Se crean los hilos “thr1” y “thr2” llamando a la clase “MiHilo” con su número de operaciones “num1” y “num2” respectivamente, así como sus nombres “Hilo 1” e “Hilo 2”.
* Se inician ambos hilos “.start()” (ejecutan su función “run”)
* Se espera a que terminen “.join()”

La salida será algo como en la imagen:



Si se ejecutara el código con ciclos “for”, se tendría una ejecución secuencial totalmente, lo cual aumentaría el tiempo total de ejecución. Se ejecutaría el primer conjunto de iteraciones, luego el segundo, etc. Teniendo una salida como:

Hilo 1 iteracion 0

Hilo 1 iteracion 1

Hilo 1 iteracion 2

Hilo 1 iteracion 3

Hilo 1 iteracion 4

Hilo 1 iteracion 5

Hilo 1 iteracion 6

Hilo 1 iteracion 7

Hilo 1 iteracion 8

Hilo 1 iteracion 9

Hilo 2 iteracion 0

Hilo 2 iteracion 1

Hilo 2 iteracion 2

Hilo 2 iteracion 3

Hilo 2 iteracion 4

Hilo 2 iteracion 5

Hilo 2 iteracion 6

Hilo 2 iteracion 7

Hilo 2 iteracion 8

Hilo 2 iteracion 9

# Análisis del Código 2

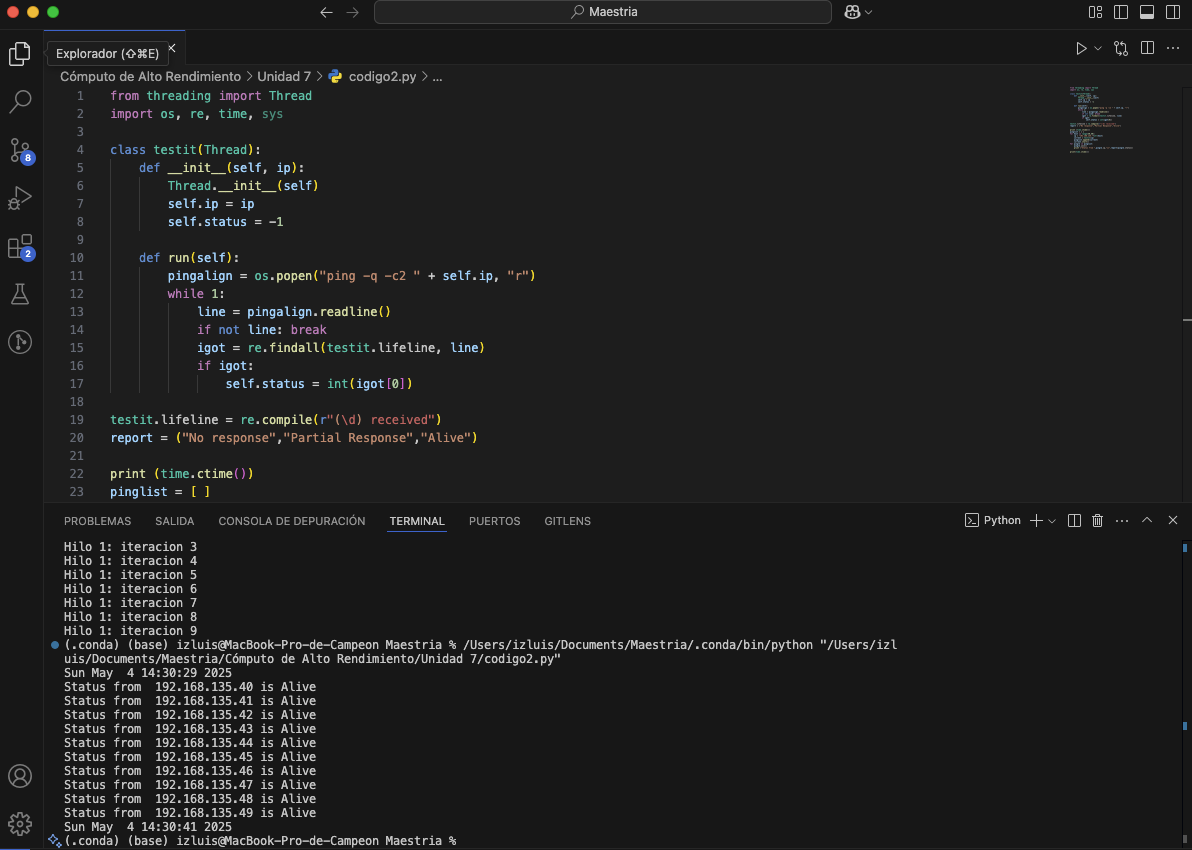
Este código crea una serie de hilos que se encargan de hacer ping a una dirección IP local distinta, al final se reporta si hubo una respuesta o no.

La clase “testit”:

* hereda de “Thread”,
* tiene la función “\_\_init\_\_” donde se guarda la dirección IP que se va a probar y pone el status en -1
* tiene la función “run”, donde se ejecuta el comando ping a la IP (con “os.popen”), lee la respuesta y usa una expresión regular ( r"(\d) received" ) para detectar cuantos paquetes se recibieron, finalmente guarda el resultado en “self.status” donde 0 es que no se recibió nada, 1 es que hubo respuesta parcial y 2 que hubo respuesta exitosa.

En la parte principal del programa se crea “current = testit(ip)” y lanzan (“current.start()”) 10 hilos (uno por cada IP), después espera a que cada hilo termine (“pingle.join()”) e imprime su resultado.

El resultado sería algo como en la imagen:



Si se ejecutara este código con ciclos for, el ping a cada IP se realizaría uno tras otro, esperando que termine cada uno de ellos, esto aumentaría el tiempo de espera. Al ejecutarlo con varios hilos, se tomó 12 segundos en total, si tomamos un tiempo de 2 segundos para cada proceso, tendríamos 20 segundos en total de manera secuencial y esto siendo positivos con el tiempo que tarda en hacerse el ping.

# Hilos literarios

El objetivo de este ejercicio es crear tres hilos que escriban el mismo archivo, sin embargo, debe ser de forma ordenada y sincronizada. Es necesario asegurar que:

1. Primero escriba Quijote
2. Romeo escriba de segundo
3. Julieta escriba de último

Es necesaria la sincronización porque, si los tres hilos escriben al mismo tiempo, el resultados podría tener frases mezcladas de cada escritor, ya que comparten el mismo recurso que es el archivo.

El código en python utiliza las librerías “threading” y “time” para ejecución.

from threading import Thread, Lock

import time

Lo primero será crear variables con los textos que escribirá cada escritor, con el fin de que el código general sea más simple de leer.

# Textos que escribirá cada hilo

quijote\_texto = """...En un lugar de la Mancha de cuyo nombre no

quiero acordarme, no ha mucho tiempo que vivía

un hidalgo de los de lanza en astillero, adarga

antigua, rocín flaco y galgo corredor.

Una olla de algo más vaca que carnero, salpicón

las más noches, duelos y quebrantos los sábados,

lentejas los viernes...

"""

romeo\_texto = """- Habla. ¡Oh! ¡Habla otra vez ángel resplan-

deciente!. . . Porque esta noche apareces tan

esplendorosa sobre mi cabeza como un ala-

do mensajero celeste ante los ojos estáticos y

maravillados de los mortales, que se inclinan

hacia atrás para verle, cuando él cabalga so-

bre las tardas perezosas nubes y navega en

el seno del aire.

"""

julieta\_texto = """¡Oh Romeo, Romeo! ¿Por qué eres tú

Romeo? Niega a tu padre y rehusa tu nom-

bre; o, si no quieres, júrame tan sólo que

me amas, y dejaré yo de ser una Capuleto.

"""

Lo siguiente será crear las variables compartidas, que serán el lock para el acceso al archivo y el turno de cada uno, así como el archivo de salida.

# Variables compartidas

lock = Lock()

turno = 0 # 0 = Quijote, 1 = Romeo, 2 = Julieta

# Ruta del archivo a escribir

archivo\_salida = "literatura.txt"

A continuación se creará la clase Escritor, esta hereda de Thread y en su inicialización se guardará su nombre, el texto y su turno

class Escritor(Thread):

def \_\_init\_\_(self, nombre, texto, mi\_turno):

super().\_\_init\_\_()

self.nombre = nombre

self.texto = texto

self.mi\_turno = mi\_turno

Dentro de la misma clase Escritor, se crea su método “run” que usa el turno global para compararlo con el del escritor, si no son iguales, espera 0.1 segundos para volver a revisar. Si son iguales, usa el lock para acceder al archvio y en este escribe el texto, finalmente le añade 1 a la variable global turno y pasar al siguiente escritor.

def run(self):

global turno

while True:

if turno == self.mi\_turno:

with lock:

with open(archivo\_salida, "a", encoding="utf-8") as f:

f.write(f"\n[{self.nombre}]\n")

f.write(self.texto)

turno += 1 # pasa al siguiente escritor

break

else:

time.sleep(0.1) # Espera un poco antes de volver a revisar

Finalmente, en la parte principal del programa se crean los hilos de cada escritor, se lanzan y se espera a que terminen, después de esto, se imprime “Archivo escrito correctamente” en la consola.

# Crear hilos

quijote = Escritor("Quijote", quijote\_texto, 0)

romeo = Escritor("Romeo", romeo\_texto, 1)

julieta = Escritor("Julieta", julieta\_texto, 2)

# Lanzar hilos

quijote.start()

romeo.start()

julieta.start()

# Esperar a que terminen

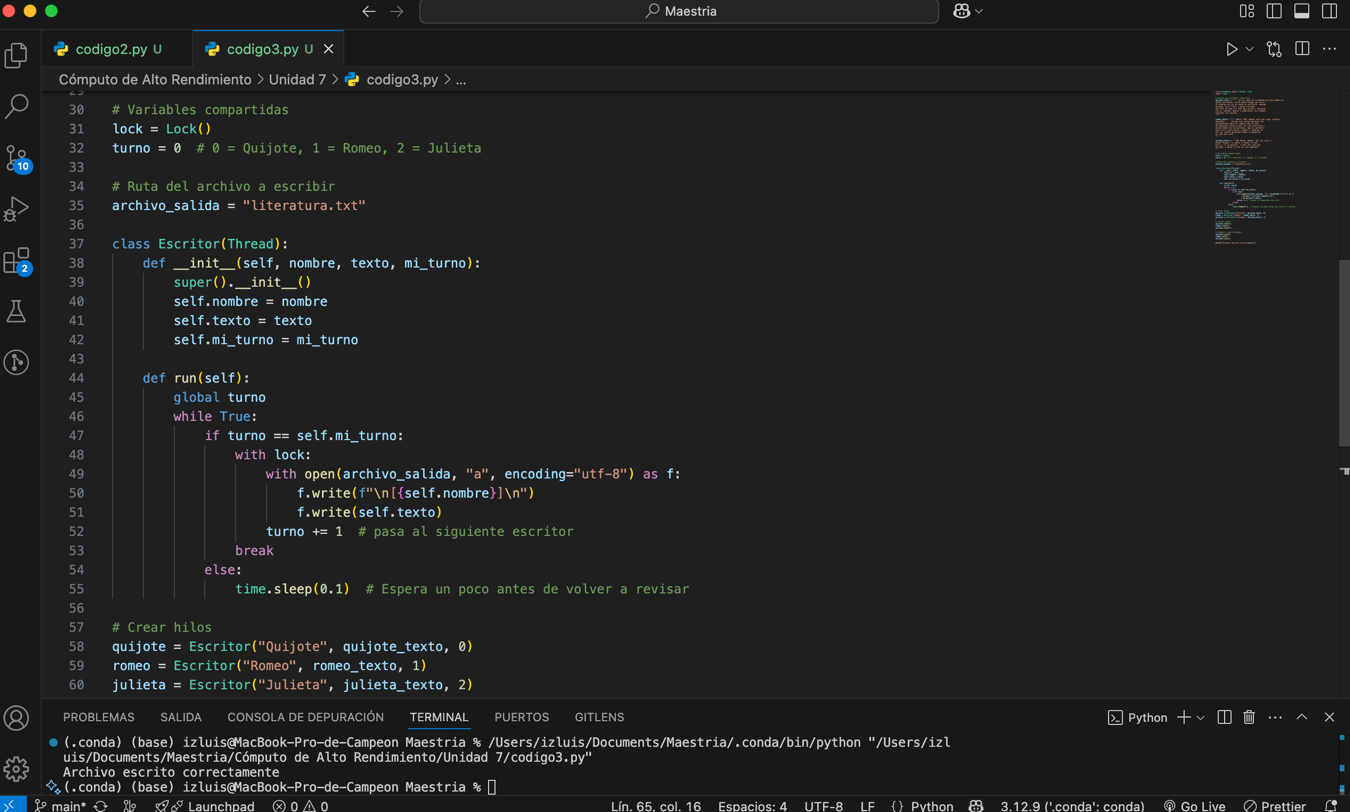
quijote.join()

romeo.join()

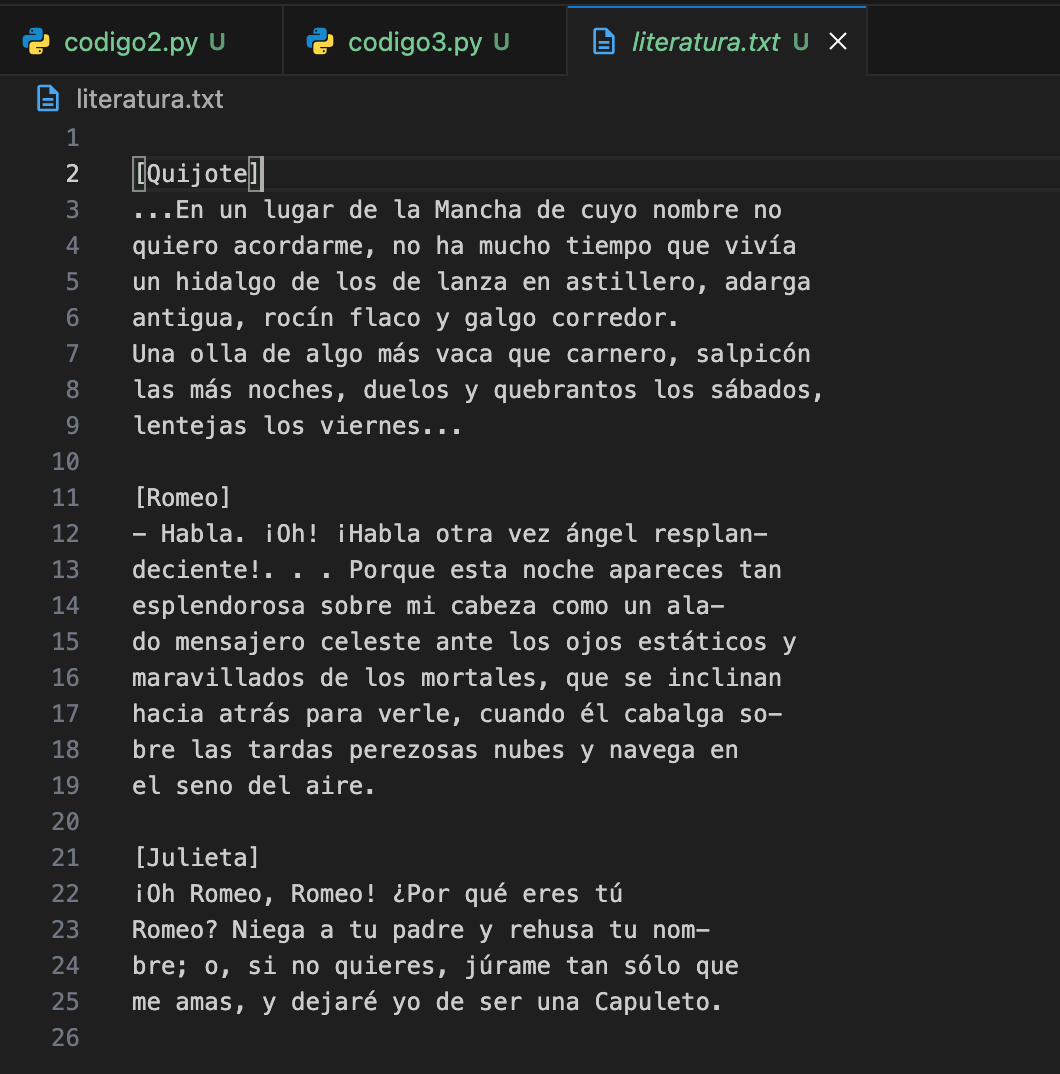
julieta.join()

print("Archivo escrito correctamente")

A continuación se ve la ejecución correcta del programa con el mensaje “Archivo escrito correctamente” en la consola.



Y al revisar el archivo generado, notamos que se generó correctamente:



# Cuenta Palabras

El objetivo es generar un programa que reciba varios archivos de texto como argumentos, cree un hilo por cada uno de ellos y cuente las palabras en paralelo. El output deberá contener el número de palabras por archivo, el total acumulado y debe contener el mismo orden en que se enviaron los archivos.

El código de Python usa las librerías threading y sys, esta última para abrir los archivos.

from threading import Thread

import sys

Primero se creará la clase “ContadorPalabras”, en su inicialización, se guarda el nombre del archivo a procesar, la lista donde se guardan los resultados y un índice de su posición en la lista anterior.

class ContadorPalabras(Thread):

def \_\_init\_\_(self, archivo, resultados, indice):

super().\_\_init\_\_()

self.archivo = archivo # Nombre del archivo a procesar

self.resultados = resultados # Lista compartida para guardar los resultados

self.indice = indice # Posición que le corresponde en esa lista

Lo siguiente es la función run, usa un try-except para ver si el archivo existe, en caso de que no exista imprime en la consola “Archivo no encontrado”. Si el archivo existe, lo abre en modo lectura, lee todo su contenido, guarda cada palabra en la variabla “palabras” y la cantidad de palabras en la variable “cantidad”, finalmente guarda el resultado en la lista.

def run(self):

try:

# Abre el archivo en modo lectura

with open(self.archivo, 'r', encoding='utf-8') as f:

texto = f.read() # Lee todo el contenido

palabras = texto.split() # Separa por espacios (simplificado)

cantidad = len(palabras) # Cuenta las palabras

self.resultados[self.indice] = (self.archivo, cantidad) # Guarda el resultado

except FileNotFoundError:

# Si el archivo no existe, guarda un mensaje de error en su lugar

self.resultados[self.indice] = (self.archivo, "Archivo no encontrado")

Dentro del main del programa, se valida que haya más de dos argumentos (para asegurarnos que mínimo haya un archivo a leer), en caso de que no haya, se imprimen instrucciones de uso del programa.

# Punto de entrada principal

if \_\_name\_\_ == "\_\_main\_\_":

# Validación de argumentos

if len(sys.argv) < 2:

print("Uso: python cuenta\_palabras.py archivo1 archivo2 ...")

sys.exit(1)

Guarda variables con la lista de nombres que se pasaron en la ventana de comandos, una lista vacía con la misma cantidad de entradas y los hilos a utilizar

archivos = sys.argv[1:] # Lista de nombres de archivos desde la línea de comandos

resultados = [None] \* len(archivos) #Lista vacía con la misma cantidad de entradas

hilos = []

Se crea un hilo por archivo y se inicializa, también se añade a la lista “hilos”.

for i, archivo in enumerate(archivos):

hilo = ContadorPalabras(archivo, resultados, i)

hilo.start()

hilos.append(hilo)

Se espera a que todos los hilos terminen y se inicializa la variable “total\_palabras” en cero.

for hilo in hilos:

hilo.join()

total\_palabras = 0

Se imprimen los resultados y se calcula el total

for archivo, cantidad in resultados:

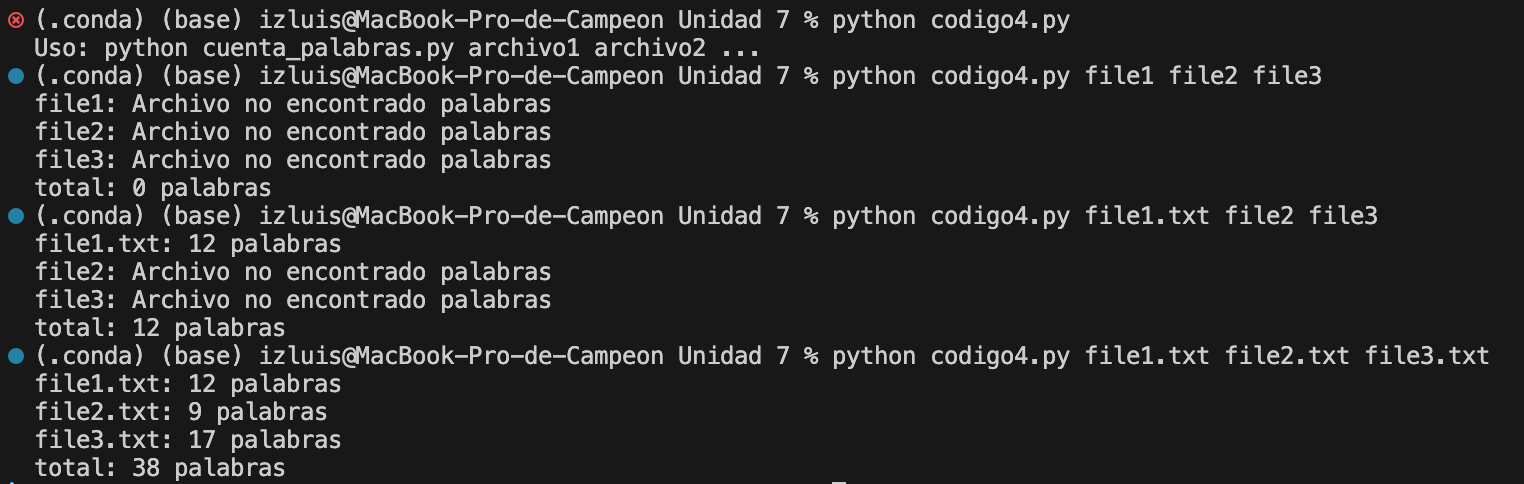
print(f"{archivo}: {cantidad} palabras")

if isinstance(cantidad, int): # Suma solo si no hubo error

total\_palabras += cantidad

print(f"total: {total\_palabras} palabras")

A continuación se muestra la terminal con la correcta ejecución del archivo con distintos casos. El primer caso es cuando no tiene argumentos para archivos, el segundo caso cuando no encuentra los archivos, el tercer caso cuando solo encuentra un archivo y el último caso es el caso ideal.



# Cliente-Servidor a chat

El objetivo de este programa es modificar los archivos “cliente.py” y “server.py” para que actúen como un chat bidireccional y ya no como un cliente y un servidor.

Este es el código original del “server.py”

# -\*- coding: utf-8 -\*-

"""

Created on Thu Apr 16 18:29:55 2015

@author: mag

"""

import socket

puerto = 4545;

miSocket = socket.socket( socket.AF\_INET, socket.SOCK\_STREAM )

miSocket.bind( ( socket.gethostname(), puerto ) )

miSocket.listen( 1 )

while True:

channel, details = miSocket.accept()

channel.send( 'Hola Mundo!' )

channel.close()

* Crea un socket TCP y lo vincula al puerto 4545
* Escucha una conexión
* Cuando un cliente se conecta, le envía el mensaje “Hola Mundo!” y cierra la conexión.

Este es el código original de “cliente.py”

# -\*- coding: utf-8 -\*-

"""

Created on Thu Apr 16 18:28:33 2015

@author: mag

"""

import socket

miSocket = socket.socket( socket.AF\_INET, socket.SOCK\_STREAM )

miSocket.connect( (socket.gethostname(), 4545 ) )

data, server = miSocket.recvfrom( 100 )

print data

miSocket.close()

* El cliente se conecta al mismo host y puerto que el server
* Espera hasta recibir 100 bytes
* Imprime el mensaje recibido
* Cierra la conexión

Primero se cambiará el código de server.py, este ahora también usará la librería “threading”.

import socket

import threading

Lo primero será crear la función para recibir mensajes del cliente. Esta recibe y decodifica hasta 1024 bytes, revisa si el mensaje dice “bye”, lo cual terminaría la conversación e imprimiría un mensaje diciendo lo mismo, en caso contarrio, imprimiría “Cliente: “ y el mensaje recibido.

def recibir\_mensajes(conexion):

while True:

try:

mensaje = conexion.recv(1024).decode('utf-8') # Recibe hasta 1024 bytes y decodifica

if mensaje.lower() == "bye": # Si el cliente dice "bye", termina el chat

print("El cliente terminó la conversación.")

break

print(f"Cliente: {mensaje}") # Muestra el mensaje recibido

except:

break # Si ocurre un error (por ejemplo, desconexión), termina

conexion.close()

Igualmente se creará la función para enviar mensajes al cliente, en esta se pide el mensaje a enviar, lo envía y termina el chat si se escribe “bye”.

def enviar\_mensajes(conexion):

while True:

mensaje = input("Tú: ") # Pide entrada del servidor

conexion.send(mensaje.encode('utf-8')) # Envía el mensaje codificado

if mensaje.lower() == "bye": # Si se escribe "bye", se termina el chat

break

conexion.close()

Se creará la función principal “main”, en esta:

* Se crea una variable para el puerto
* Se crea el socket del servidor
* Vincula el socket a la IP local y al puerto especificado
* Escucha una conexión entrante
* Imprime “Servidor esperando conexión”
* Acepta la conexión del cliente
* Imprime “Cliente conectado desde” y la dirección
* Crea dos hilos para recibir y enviar mensajes, de igual manera los inicia y espera a que terminen
* Cierra el socket e imprime “Chat finalizado”

def main():

puerto = 4545

# Crea el socket del servidor

mi\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

mi\_socket.bind((“127.0.0.1”,puerto)) # Lo vincula a la IP local y puerto 4545

mi\_socket.listen(1) # Escucha una conexión entrante

print("Servidor esperando conexión...")

conexion, direccion = mi\_socket.accept() # Acepta conexión del cliente

print(f"Cliente conectado desde {direccion}")

# Crea dos hilos: uno para recibir y otro para enviar mensajes

hilo\_receptor = threading.Thread(target=recibir\_mensajes, args=(conexion,))

hilo\_emisor = threading.Thread(target=enviar\_mensajes, args=(conexion,))

# Inicia los hilos

hilo\_receptor.start()

hilo\_emisor.start()

# Espera a que ambos hilos terminen

hilo\_receptor.join()

hilo\_emisor.join()

mi\_socket.close()

print("Chat finalizado.")

Finalmente se ejecuta la función principal

# Ejecuta la función principal

if \_\_name\_\_ == "\_\_main\_\_":

main()

Ahora, para modificar el archivo de cliente.py, lo primero será también añadirle la librería “threading”

import socket

import threading

Se definirá la función para recibir mensajes del servidor, esta es una copia de la que se generó en el servidor

def recibir\_mensajes(socket\_cliente):

while True:

try:

mensaje = socket\_cliente.recv(1024).decode('utf-8') # Recibe y decodifica mensaje

if mensaje.lower() == "bye": # Si recibe "bye", termina el chat

print("El servidor terminó la conversación.")

break

print(f"Servidor: {mensaje}") # Muestra mensaje recibido

except:

break

socket\_cliente.close()

Se define la función para enviar mensajes al servidor, que igual será una copia de la del servidor.

def enviar\_mensajes(socket\_cliente):

while True:

mensaje = input("Tú: ") # Captura mensaje desde terminal

socket\_cliente.send(mensaje.encode('utf-8')) # Lo envía al servidor

if mensaje.lower() == "bye": # Si escribes "bye", termina el chat

break

socket\_cliente.close()

La principal diferencia está en la conexión que se genera en la función “main”. En el caso del cliente:

* Se guarda el nombre del host local en una variable
* Se guarda el puerto del servidor en una variable
* Se crea el socket del cliente
* Se conecta al servidor e imprime “Conectado al servidor. Puedes comenzar a chatear”.

Todo lo demás es igual al servidor (creación, inicialización y finalización de hilos).

def main():

host = socket.gethostname() # Usa el nombre del host local

puerto = 4545 # Mismo puerto que el servidor

# Crea el socket del cliente

socket\_cliente = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

socket\_cliente.connect((host, puerto)) # Conecta al servidor

print("Conectado al servidor. Puedes comenzar a chatear.")

# Crea dos hilos: uno para recibir, otro para enviar

hilo\_receptor = threading.Thread(target=recibir\_mensajes, args=(socket\_cliente,))

hilo\_emisor = threading.Thread(target=enviar\_mensajes, args=(socket\_cliente,))

# Inicia ambos hilos

hilo\_receptor.start()

hilo\_emisor.start()

# Espera a que ambos terminen

hilo\_receptor.join()

hilo\_emisor.join()

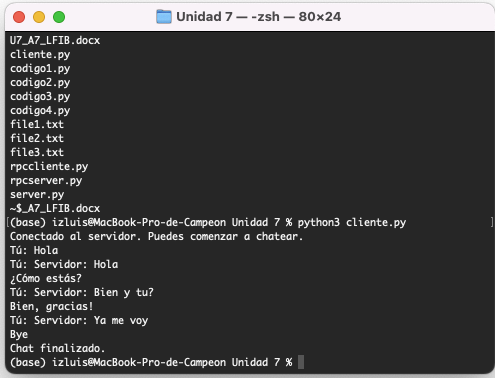
print("Chat finalizado.")

Finalmente se ejecuta la función main.

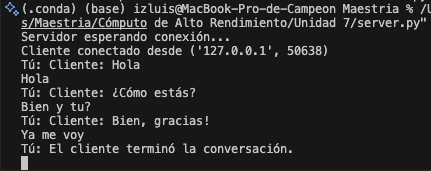
if \_\_name\_\_ == "\_\_main\_\_":

main()

A continuación se observa el correcto funcionamiento del lado del cliente



Y el correcto funcionamiento del lado del servidor



Podría mejorarse el display de los textos, pero en rasgos generales, funcionan correctamente.

# Ampliación código RPC

El objetivo de este ejercicio es tomar los códigos de “rpccliente.py” y “rpcserver.py” para añadir 5 nuevas funciones al servidor que se puedan llamar desde el cliente. El servidor deberá estar en una maquina distinta.

El código de “rpcserver.py” original solamente:Define una función suma, que toma x e y para sumarlos.

* Crea un servidor XML-RPC en el puerto 4242
* Registra la función con el nombre de “suma”
* Escucha peticiones indefinidamente

from SimpleXMLRPCServer import SimpleXMLRPCServer

def suma(x,y):

return x+y

servidor = SimpleXMLRPCServer(("", 4242))

print "Escuchando por el puerto 4242"

servidor.register\_function(suma, 'suma')

servidor.serve\_forever()

El código “rpccliente.py” original:

* Se conecta al servidor RPC en localhost:4242
* Llama la función remota suma con los valores 2 y 6 e imprime el resultado

from xmlrpclib import ServerProxy

servidor = ServerProxy('http://localhost:4242')

print servidor.suma(2,6)

Para la actualización de “rpcserver.py” se cambiaron las librerías para Python 3 y también se importó “math” para hacer las nuevas funciones.

from xmlrpc.server import SimpleXMLRPCServer

import math

Se crearon las nuevas funciones que estarán disponibles remotamente

def suma(x, y):

return x + y

def resta(x, y):

return x - y

def multiplica(x, y):

return x \* y

def divide(x, y):

if y == 0:

return "Error: División entre cero"

return x / y

def potencia(x, y):

return x \*\* y

def factorial(n):

if n < 0:

return "Error: Factorial no definido para negativos"

return math.factorial(n)

Se crea el servidor en localhost y en el puerto 8000

servidor = SimpleXMLRPCServer(("localhost", 8000), allow\_none=True)

print("Servidor RPC escuchando en el puerto 8000...")

Se registran todas las funciones

servidor.register\_function(suma, 'suma')

servidor.register\_function(resta, 'resta')

servidor.register\_function(multiplica, 'multiplica')

servidor.register\_function(divide, 'divide')

servidor.register\_function(potencia, 'potencia')

servidor.register\_function(factorial, 'factorial')

Se inicia el servidor indefinidamente

servidor.serve\_forever()

Para mejorar el código de “rpccliente.py” solamente se usa una librería compatible con Python 3

from xmlrpc.client import ServerProxy

Se establece la conexión al servidor, en mi caso, utilizaré una laptop que se que está en la IP local 192.168.1.67 y el puerto será el 8000 (como se vió en el código del servidor)

servidor = ServerProxy('http://192.168.1.67:8000/')

Se prueban las funciones remotas

print("2 + 3 =", servidor.suma(2, 3))

print("10 - 4 =", servidor.resta(10, 4))

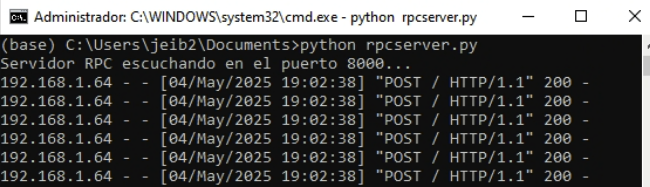
print("6 \* 7 =", servidor.multiplica(6, 7))

print("20 / 5 =", servidor.divide(20, 5))

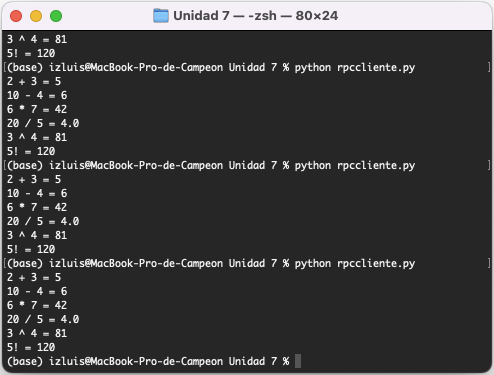
print("3 ^ 4 =", servidor.potencia(3, 4))

print("5! =", servidor.factorial(5))

En una computadora con Windows, en un ambiente de anaconda, se ejecutó el código “rpcserver.py”



En una computadora con MacOS se ejecutó el código “rpccliente.py”



Como se puede observar, la comunicación fue exitosa y se ejecutaron todas las operaciones añadidas.

# Algoritmo de los filósofos comensales

El objetivo es diseñar un algoritmo en pseudocódigo para resolver el problema de los filósofos comensales. Tratando sobretodo evitar starvation y deadlock.

El problema de los filósofos comensales dice que 5 filósofos están sentados alrededor de una mesa circular y frente a cada uno de ellos se encuentra un plato de arroz y un palillo a la izquierda. Cada uno de los filósofos tiene que tomar dos palillos para poder comer y solamente hay 5 palillos en total (uno por cada uno de ellos), por lo cual solamente podrán comer 2 a la vez y los otros 3 deberán esperar.

Los problemas a evitar son:

* Deadlock: este ocurre si todos los filósofos toman el palillo de su izquierda al mismo tiempo y esperan para poder tomar el de la derecha, en este caso, ninguno puede continuar.
* Starvation: Ocurre cuando un filósofo no puede obtener nunca ambos palillos porque siempre se le adelantan los demás.

La solución a implementar es una cola First In, First Out con dos lock, uno serán los dos palillos a tomar y el otro se utiliza para acceder a la cola, igual se implementará un semáforo para que solamente 2 filósofos puedan comer simultáneamente. Los filósofos entran en una cola por orden de llegada, solamente pueden comer cuando son los primeros en la cola y hay permisos disponibles para comer (máximo 2 permisos simultáneos), después de esto toman dos palillos y comen. Con esto se evita el deadlock porque aseguramos que solamente dos comen a la vez y también starvation porque todos terminan comiendo eventualmente.

El código se implemetó en Python para corroborar que funcione, se utilizaron las librerías de “threading”, “time” y “random”.

import threading

import time

import random

Se añade el número de filósofos y palillos

N = 5

Cada palillo es un Lock entre filósofos

palillos = [threading.Lock() for \_ in range(N)]

Se genera un lock para acceder a la cola de forma segura

lock\_cola = threading.Lock()

Se genera el semáforo que actuará como permiso para comer y solo permitirá dos filósofos comiendo simultáneamente.

permiso\_para\_comer = threading.Semaphore(2)

En una función de filósofo tenemos varios pasos:

def filosofo(i):

while True:

El primero será anadirse a la cola de espera

with lock\_cola:

cola.append(i)

print(f"Filósofo {i} se unió a la cola: {cola}")

Después espera a que sea su turno y haya permiso para comer, entonces avanza, caso contrario, vuelve a revisar en 0.1 segundos.

while True:

with lock\_cola:

es\_mi\_turno = (cola[0] == i)

if es\_mi\_turno and permiso\_para\_comer.acquire(blocking=False):

break

time.sleep(0.1)

El tercer paso es tomar dos palillos y comer (se le puso un sleep con tiempo aleatorio para simular comer)

palillos[i].acquire()

palillos[(i + 1) % N].acquire()

print(f"Filósofo {i} está comiendo...")

time.sleep(random.uniform(1, 2)) # Simula tiempo comiendo

Lo siguiente será liberar los palillos y el permiso para comer

palillos[i].release()

palillos[(i + 1) % N].release()

permiso\_para\_comer.release()

Finalmente termina de comer.

with lock\_cola:

print(f"Filósofo {i} terminó de comer.")

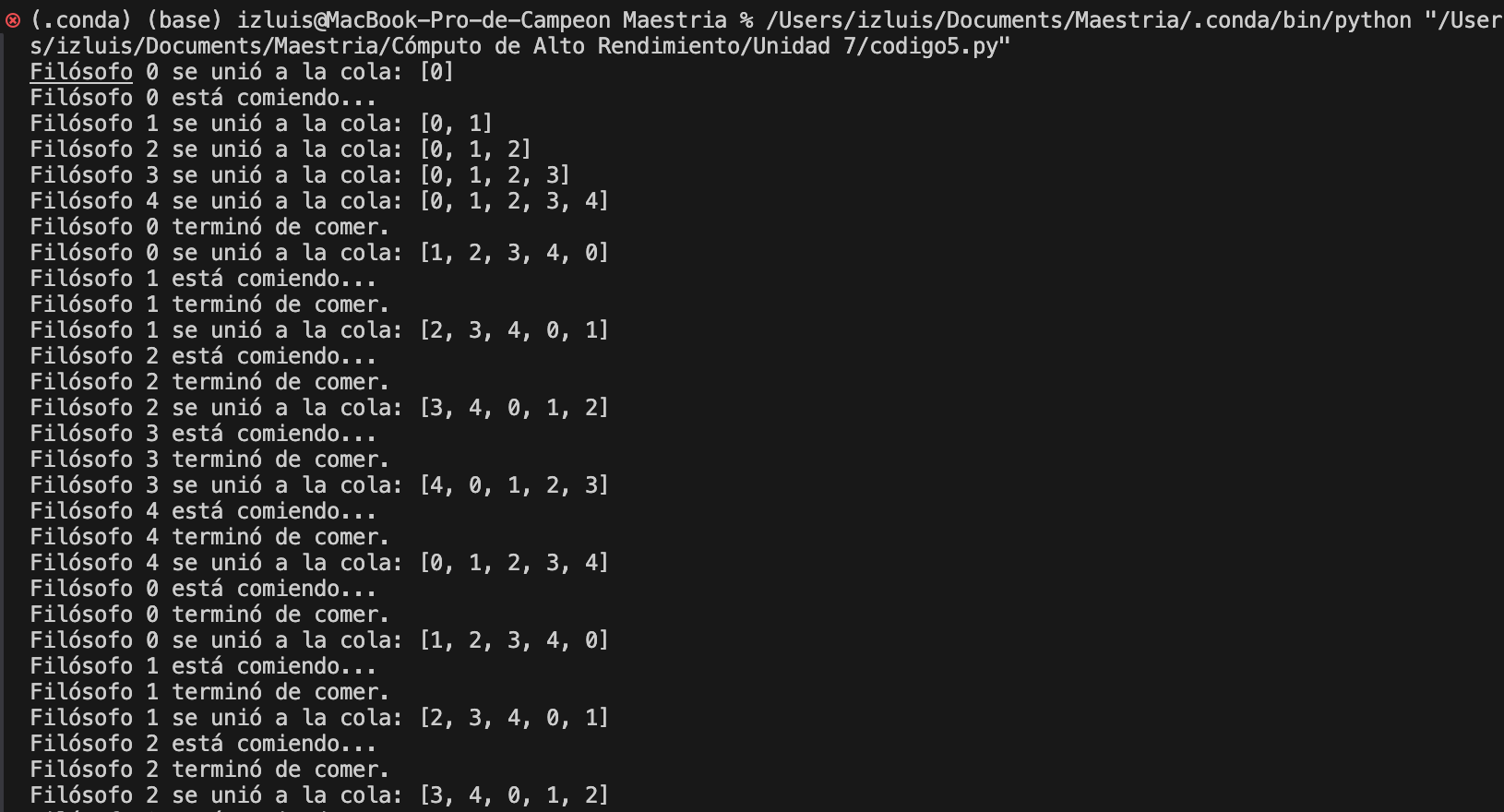
cola.pop(0)

Lo siguiente será crear e iniciar los hilos

for i in range(N):

threading.Thread(target=filosofo, args=(i,), daemon=True).start()

Al ejecutar el Código, se observa que todos los filósofos entran a la cola y ninguno se queda sin comer.



Al implementar esta solución, parecía ser una opción bastante correcta y, al observar los resultados se ve que no hay deadlock ni starvation, sin embargo, analizándolos detenidamente, pareceria que es un proceso un poco más secuencial que paralelo. Estaría interesante probar otra soluciones para lograr que sea más paralelo de lo que se observa actualmente.

# Investigación sobre algoritmos de exclusión mutua.

## Algoritmo de Dekker

Este algoritmo fue una de las primeras soluciones al problema de exclusión mutua (mutex) entre dos procesos concurrentes. Este algoritmo permite que solamente un proceso entre a su sección crítica y el otro se queda esperando.

El algoritmo usa banderas booleanas para indicar si un proceso entró a la sección crítica, usa una variable de turno y decide quien tiene la prioridad, en el caso de que ambos procesos intenten entrar, el turno se respeta hasta que el otro proceso lo ceda.

## Algoritmo de Peterson

Este algoritmo, como el de Dekker, también resuelve la exclusión mutua entre dos procesos, sin embargo, lo hace de una manera más sencilla.

Cada proceso indica que quiere entrar con una bandera booleana, al terminar se cede el turno al otro proceso, el cual solo entra si el otro no quiere o le ceden el turno.

# Bibliografía

Unidad 4: Exclusión mutua, condiciones de carrera y sincronización. (2017). [Presentación en PDF]. Benemérita Universidad Autónoma de Puebla, Facultad de Ciencias de la Computación. <https://www.cs.buap.mx/~hilario_sm/slide/pcp2016/unidad%204%20pcp-2017.pdf>

Arellano Vazquez, M. (s.f.). Procesos. INFOTEC.

Arellano Vazquez, M. (s.f.). Comunicación entre procesos. INFOTEC.