Universidad de Guadalajara

Centro Universitario de Ciencias Exactas e Ingenierías

Computación Tolerante a Fallas

Maestro: Michel Emanuel Lopez Franco

Juan Antonio Perez Juarez
Carrera: INCO
Código: 215660996

An Introduction to Scaling Distributed Python Applications

Introducción

En el ámbito del desarrollo de software, el escalamiento de aplicaciones es un aspecto crucial para garantizar que los sistemas puedan manejar cargas de trabajo crecientes de manera eficiente. Una de las estrategias más comunes para lograr este escalamiento es mediante el uso de hilos, multihilos, concurrencia y daemons en Python. Estas técnicas permiten optimizar el rendimiento de las aplicaciones, especialmente en tareas que involucran operaciones de entrada/salida (I/O) o procesos que pueden ejecutarse de manera paralela.

Desarrollo

Los hilos son unidades básicas de ejecución dentro de un proceso, permitiendo que múltiples tareas se ejecuten aparentemente al mismo tiempo. En Python, el módulo threading facilita la creación y gestión de hilos, lo que es útil para tareas que no requieren un uso intensivo de la CPU, como la gestión de múltiples conexiones en una aplicación de red.

El multihilo lleva este concepto un paso más allá, permitiendo que varios hilos se ejecuten simultáneamente en un entorno de múltiples núcleos. Sin embargo, debido al Global Interpreter Lock (GIL) en Python, el multihilo no siempre es la mejor opción para tareas intensivas en CPU, pero es altamente efectivo para operaciones de I/O.

La concurrencia es un concepto más amplio que engloba la ejecución de múltiples tareas de manera superpuesta en el tiempo. En Python, esto se puede lograr no solo con hilos, sino también con el módulo asyncio, que permite la programación asíncrona, ideal para aplicaciones que necesitan manejar muchas operaciones de I/O de manera eficiente.

Finalmente, los daemons son procesos que se ejecutan en segundo plano, generalmente sin interacción directa con el usuario. En el contexto de los hilos, un hilo daemon en Python es aquel que no impedirá que el programa principal termine su ejecución, incluso si el hilo daemon aún está en funcionamiento. Esto es útil para tareas de mantenimiento o monitoreo que no deben bloquear la finalización del programa.

Con ayuda de la IA, programé un monitor de Tares dentro del sistema, como si fuera un task manager de Linux, que revisa el estatus de uso del procesador, la memoria, la temperatura, etc.

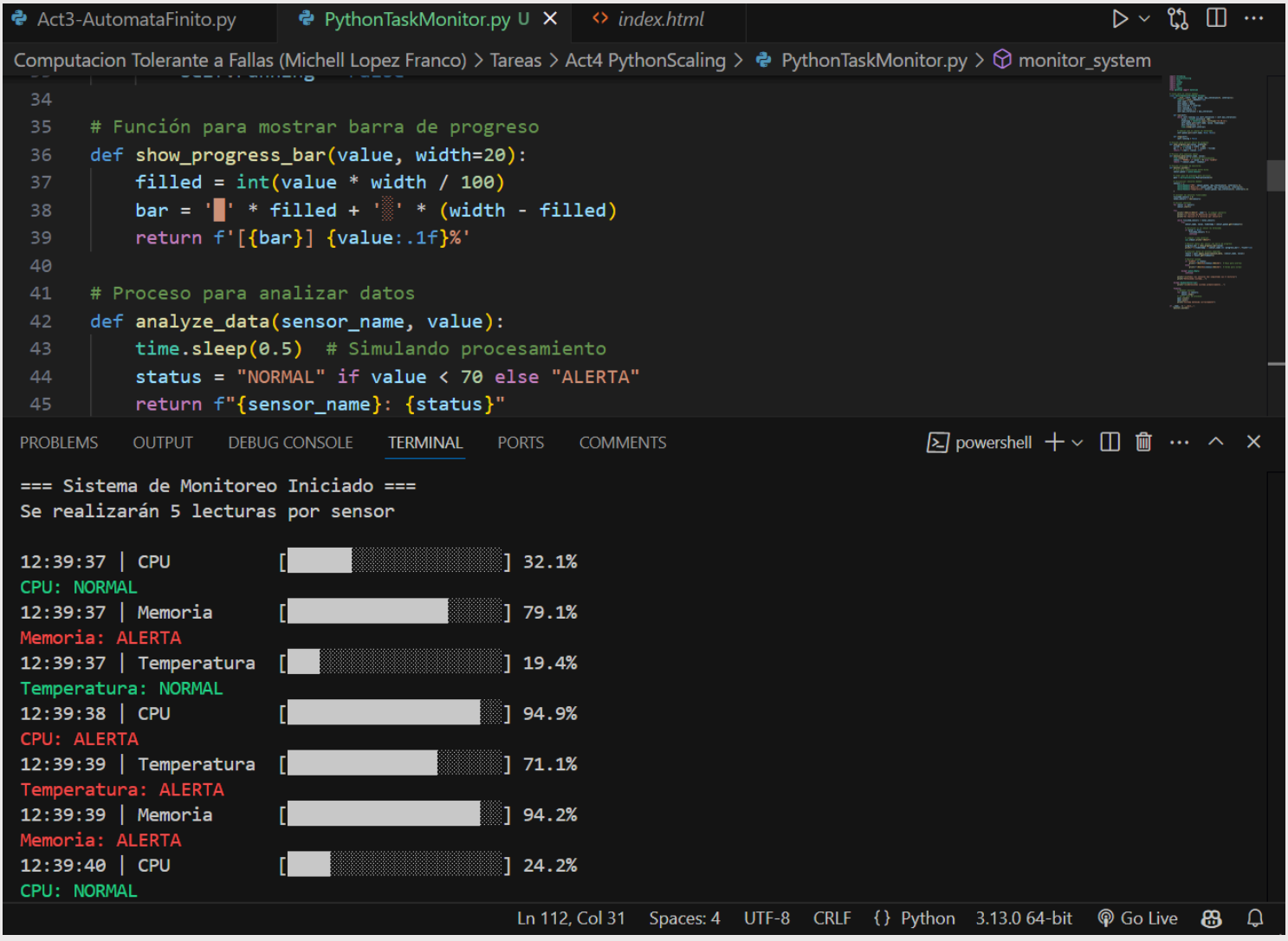
```python
import threading
import multiprocessing
import time
import random
import queue
import sys
```

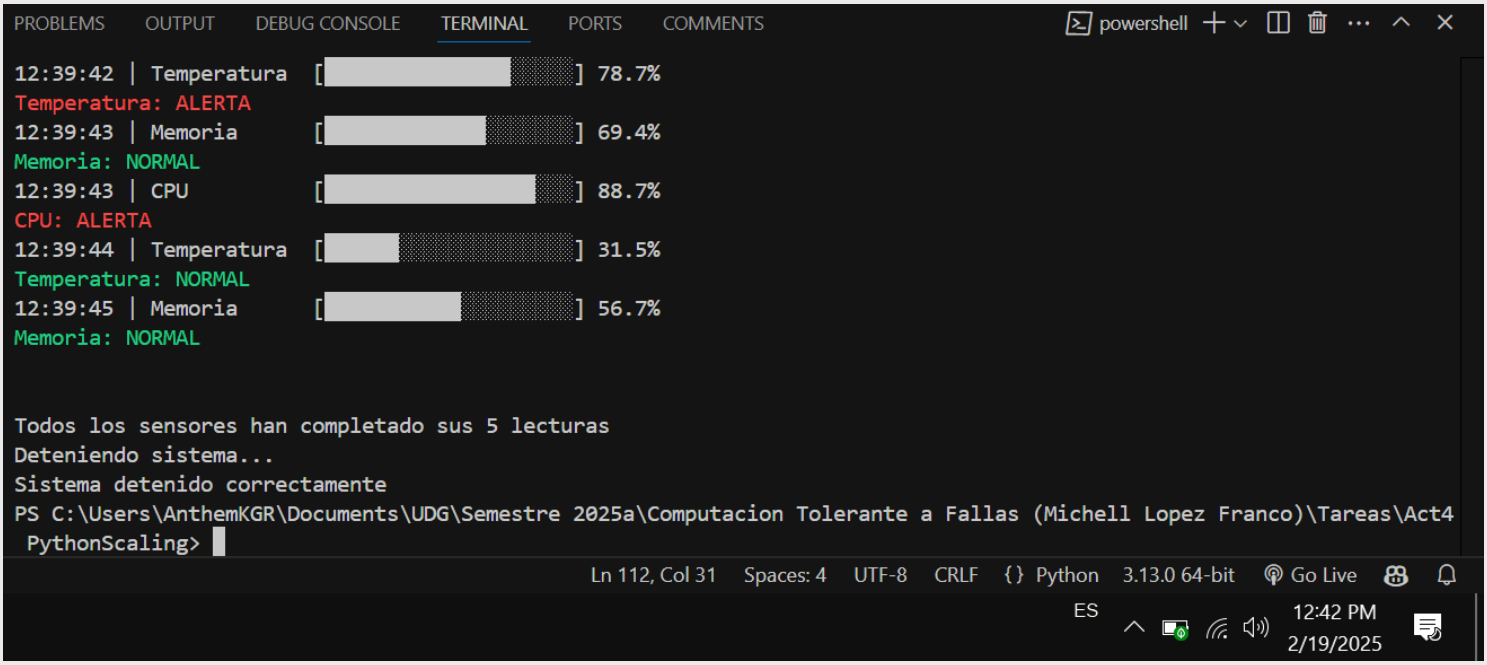
```
import signal
from datetime import datetime

# Clase para el sensor daemon
class SensorDaemon(threading.Thread):
    def __init__(self, name, queue, max_iterations=5, interval=1):
        super().__init__(daemon=True)
        self.name = name
        self.queue = queue
        self.interval = interval
        self.running = True
        self.iterations = 0
        self.max_iterations = max_iterations

    def run(self):
        while self.running and self.iterations < self.max_iterations:
            value = random.uniform(0, 100)
            timestamp = datetime.now().strftime("%H:%M:%S")
            self.queue.put((self.name, value, timestamp))
            self.iterations += 1
            time.sleep(self.interval)

        # Indicar que el sensor ha terminado
        self.queue.put((self.name, None, None))

    def stop(self):
        self.running = False

# Función para mostrar barra de progreso
def show_progress_bar(value, width=20):
    filled = int(value * width / 100)
    bar = '█' * filled + '░' * (width - filled)
    return f'[{bar}] {value:.1f}%'

# Proceso para analizar datos
def analyze_data(sensor_name, value):
    time.sleep(0.5)  # Simulando procesamiento
    status = "NORMAL" if value < 70 else "ALERTA"
    return f"{sensor_name}: {status}"

# Función principal de monitoreo
def monitor_system():
    # Cola para comunicación entre hilos
    sensor_queue = queue.Queue()

    # Crear pool de procesos para análisis
    pool = multiprocessing.Pool(processes=2)

    # Inicializar sensores daemon
    sensors = [
```

```
    SensorDaemon("CPU", sensor_queue, max_iterations=5, interval=1.5),
    SensorDaemon("Memoria", sensor_queue, max_iterations=5, interval=2),
    SensorDaemon("Temperatura", sensor_queue, max_iterations=5, interval=1.8)
]

# Contador de sensores finalizados
finished_sensors = 0
total_sensors = len(sensors)

# Iniciar sensores
for sensor in sensors:
    sensor.start()

try:
    print("\033[2J\033[H", end="")  # Limpiar pantalla
    print("=== Sistema de Monitoreo Iniciado ===")
    print("Se realizarán 5 lecturas por sensor\n")

    while finished_sensors < total_sensors:
        try:
            sensor_name, value, timestamp = sensor_queue.get(timeout=1)

            # Verificar si el sensor ha terminado
            if value is None:
                finished_sensors += 1
                continue

            # Limpiar línea anterior
            sys.stdout.write("\033[K")

            # Mostrar datos del sensor con barra de progreso
            progress_bar = show_progress_bar(value)
            print(f"\r{timestamp} | {sensor_name:12} {progress_bar}", flush=True)

            # Analizar datos en proceso separado
            result = pool.apply_async(analyze_data, (sensor_name, value))
            status = result.get(timeout=1)

            # Mostrar estado
            if "ALERTA" in status:
                print(f"\033[91m{status}\033[0m")  # Rojo para alertas
            else:
                print(f"\033[92m{status}\033[0m")  # Verde para normal

        except queue.Empty:
            continue

    print("\n\nTodos los sensores han completado sus 5 lecturas")
    print("Deteniendo sistema...")
```

```
    except KeyboardInterrupt:
        print("\n\nDeteniendo sistema prematuramente...")

    finally:
        # Detener sensores
        for sensor in sensors:
            sensor.stop()
        # Cerrar pool de procesos
        pool.close()
        pool.join()
        print("Sistema detenido correctamente")

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

Pruebas del Funcionamiento:





Conclusión

Que ganas tenía de hacer algo parecido a Linux, la verdad es que soy un Linux head and fanboy, así que el tener la motivación de hacer esto como si fuera un linux me hizo sentir muy bien.

Aunque siendo sincero sin la ayuda de la IA para las barras de carga esto no se hubiera visto así de bien, pero en general esta actividad me ayudó a entender que todo lo que te propongas en cuanto a programación, lo puedes hacer con python, ya veo por qué es un lenguaje de programación tan usado y tan poderoso.

Referencias

Fawcett, A. (n.d.). An introduction to scaling distributed Python applications. Educative. https://www.educative.io/blog/scaling-in-python

Dixit, P., & Dixit, P. (2023, May 3). Concurrency in Python: threading, processes, and asyncio - StatusNeo. StatusNeo - Cloud Native Technology Services & Consulting. https://statusneo.com/concurrency-in-python-threading-processes-and-asyncio/