



Universidad de Guadalajara  
Centro Universitario de Ciencias Exactas e Ingenierías  
División de Tecnologías para la Integración Ciber-Humana  
Departamento de Ciencias Computacionales  
Ingeniería en Computación



## Ejercicio 07.

**Programa que usa hilos, procesos, demonios y concurrencia.**

### **Alumno**

Hernández Cortez Kevin Uriel.

### **Materia**

Computación Tolerante a Fallas.

I7036, D06, 2024A

### **Profesor**

Lopez Franco Michel Emanuel.

## INTRODUCCIÓN

En esta tarea se explora el uso de técnicas de concurrencia en Python para gestionar tareas simultáneas de manera eficiente. Se comenzará la codificación importando los módulos necesarios y definiendo funciones que representan las diferentes tareas que se quieren ejecutar. Luego, en el código principal, se crearán y ejecutarán hilos, procesos y tareas concurrentes. Además, se introduce la noción de un "demonio", un proceso en segundo plano que puede ejecutar tareas continuamente sin interferir con la ejecución principal del programa.

## DESARROLLO

### Código

```
#Hernández Cortez Kevin Uriel
#Programa que usa hilos, procesos, demonios y concurrencia.
import threading
import multiprocessing
import concurrent.futures
import time

#-----
#funcion para ejecutar en un hilo
def funcionHilo(name):
    print(f"Hilo {name} iniciado")
    time.sleep(2)
    print(f"Hilo {name} terminado")
#funcion para ejecutar en un proceso
def funcionProceso(name):
    print(f"Proceso {name} iniciado")
    time.sleep(2)
    print(f"Proceso {name} terminado")
#funcion para ejecutar en una tarea concurrente
def funcionConcurrente(name):
    print(f"Tarea concurrente {name} iniciada")
    time.sleep(2)
    print(f"Tarea concurrente {name} terminada")
#funcion para ejecutar como demonio
def funcionDemonio():
    while True:
        #tareas que podría hacer el demonio
        print("Demonio ejecutandose...")
        time.sleep(1)

#-----
if __name__ == "__main__":
    #creamos y ejecutamos un proceso "demonio"
    daemon_process = multiprocessing.Process(target=funcionDemonio)
    daemon_process.daemon = True #aqui se indica que es un demonio
    daemon_process.start()
    #ejecutamos hilos
    for i in range(3):
        thread = threading.Thread(target=funcionHilo, args=(i,))
        thread.start()
    #ejecutamos procesos
```

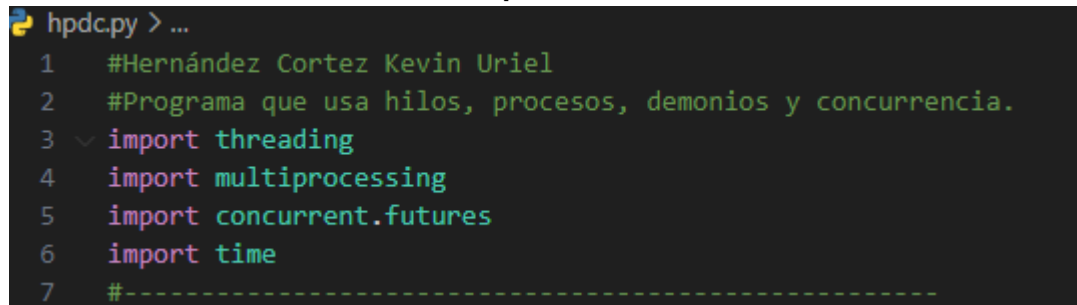
```

    for i in range(3):
        process = multiprocessing.Process(target=funcionProceso,
args=(i,))
        process.start()
        #ejecutamos tareas concurrentes
    with concurrent.futures.ThreadPoolExecutor() as executor:
        futures = [executor.submit(funcionConcurrente, i) for i in
range(3)]
        concurrent.futures.wait(futures)

    print("Exitó!")
#-----

```

### Explicación



```

hpdc.py > ...
1  #Hernández Cortez Kevin Uriel
2  #Programa que usa hilos, procesos, demonios y concurrencia.
3  import threading
4  import multiprocessing
5  import concurrent.futures
6  import time
7  #-----

```

Primeramente, importo los módulos necesarios para trabajar con hilos (threading), procesos (multiprocessing), concurrencia (concurrent.futures) y para gestionar el tiempo (time).

```

#-----
#funcion para ejecutar en un hilo
def funcionHilo(name):
    print(f"Hilo {name} iniciado")
    time.sleep(2)
    print(f"Hilo {name} terminado")
#funcion para ejecutar en un proceso
def funcionProceso(name):
    print(f"Proceso {name} iniciado")
    time.sleep(2)
    print(f"Proceso {name} terminado")
#funcion para ejecutar en una tarea concurrente
def funcionConcurrente(name):
    print(f"Tarea concurrente {name} iniciada")
    time.sleep(2)
    print(f"Tarea concurrente {name} terminada")
#funcion para ejecutar como demonio
def funcionDemonio():
    while True:
        #tareas que podría hacer el demonio
        print("Demonio ejecutandose...")
        time.sleep(1)
#-----

```

Aquí defino varias funciones:

funciónHilo, funcionProceso y funcionConcurrente son funciones que se ejecutarán en hilos, procesos y tareas concurrentes. Cada una imprime un mensaje de inicio, espera 2 segundos y luego imprime un mensaje de finalización. funcionDemonio representa las tareas que realizará el demonio. En este caso, solo imprime un mensaje indicando que el demonio está en funcionamiento y espera 1 segundo antes de repetir.

```

#-----
if __name__ == "__main__":
    #creamos y ejecutamos un proceso "demonio"
    daemon_process = multiprocessing.Process(target=funcionDemonio)
    daemon_process.daemon = True #aquí se indica que es un demonio
    daemon_process.start()
#-----

```

Aquí creo un proceso utilizando la función funcionDemonio como objetivo y luego establezco el proceso como demonio con `daemon_process.daemon = True`. Finalmente, inicio el proceso demonio con `daemon_process.start()`

```

#ejecutamos hilos
for i in range(3):
    thread = threading.Thread(target=funcionHilo, args=(i,))
    thread.start()
#ejecutamos procesos
for i in range(3):
    process = multiprocessing.Process(target=funcionProceso, args=(i,))
    process.start()
#ejecutamos tareas concurrentes
with concurrent.futures.ThreadPoolExecutor() as executor:
    futures = [executor.submit(funcionConcurrente, i) for i in range(3)]
    concurrent.futures.wait(futures)

print("Exito!")
#-----

```

En estas secciones (y las últimas) se ejecutan hilos, procesos y tareas concurrentes. Utilizo bucles for para crear múltiples instancias de cada uno y luego iniciarlas con el método start(). En el caso de las tareas concurrentes, utilizo un ThreadPoolExecutor para ejecutarlas en un conjunto de hilos.

## CONCLUSIÓN

Yo concluyo que cada una de estas técnicas tiene sus propias características y casos de uso. Los hilos son muy útiles para ejecutar tareas de forma concurrente dentro de un mismo proceso, mientras que los procesos nos permiten la ejecución paralela de tareas independientes, aprovechando los recursos del sistema de manera eficiente. Las tareas concurrentes, a través de ThreadPoolExecutor, nos regalan un enfoque flexible para ejecutar múltiples tareas al mismo tiempo. Además, hemos visto el concepto de procesos demonio, que pueden ejecutar tareas en segundo plano de forma continua. Al comprender y aplicar estas técnicas, podemos diseñar aplicaciones más escalables y eficientes que satisfagan las necesidades de concurrencia en distintos entornos distribuidos.