GENERALITAT VALENCIANA
Conselleria d'Educació, Cultura i Esport

ceedcv CENTRE ESPECÍFIC D'EDUCACIÓ A DISTÀNCIA DE LA COMUNITAT VALENCIANA

Unidad 1.3

Programación multihilo

Índice

Sumario

Licencia

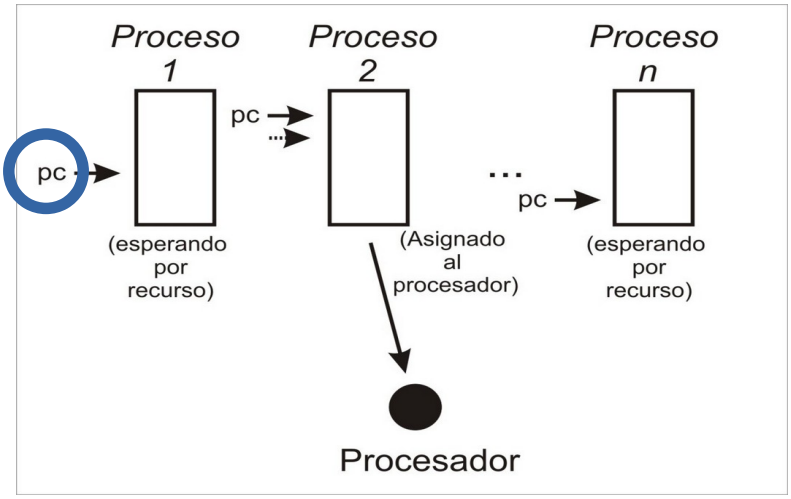
1. INTRODUCCIÓN Y FUNDAMENTOS

En el ámbito de la programación, un proceso se define como un programa en ejecución y constituye la unidad fundamental de procesamiento gestionada por el sistema operativo. Cada proceso conceptualmente posee un hilo de ejecución, también conocido como thread, que se visualiza como una CPU virtual. Este enfoque permite al sistema operativo gestionar eficientemente la ejecución de múltiples tareas de manera simultánea.

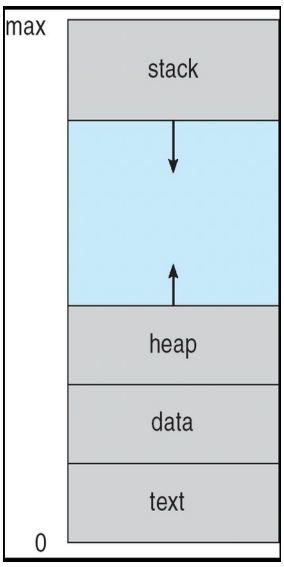
1.1. Elementos de un proceso

Cada proceso se identifica mediante un Identificador Único de Proceso (**PID**), que lo distingue de otros procesos en el sistema. Además, cada proceso cuenta con un Contador de Programa (**PC**) que contiene la dirección de la próxima instrucción a ejecutar y avanza cuando el proceso tiene acceso a la unidad de procesamiento.

```
  PID TTY      STAT   TIME COMMAND
 1664 ?        Ss     0:00 /lib/systemd/systemd --user
 1665 ?        S      0:00 (sd-pam)
 1679 ?        S<sl   0:00 /usr/bin/pipewire
 1680 ?        S<sl   0:02 /usr/bin/pulseaudio --daemonize=no --log-target=jou
 1682 ?        SNsl   0:00 /usr/libexec/tracker-miner-fs
 1686 ?        Ss     0:00 /usr/bin/dbus-daemon --session --address=systemd: -
 1691 ?        Sl     0:00 /usr/bin/gnome-keyring-daemon --daemonize --login
 1706 ?        S<l    0:00 /usr/bin/pipewire-media-session
 1708 ?        Ssl    0:00 /usr/libexec/gvfsd
```



1.2. Memoria de los procesos



Un proceso en memoria se constituye de varias secciones:

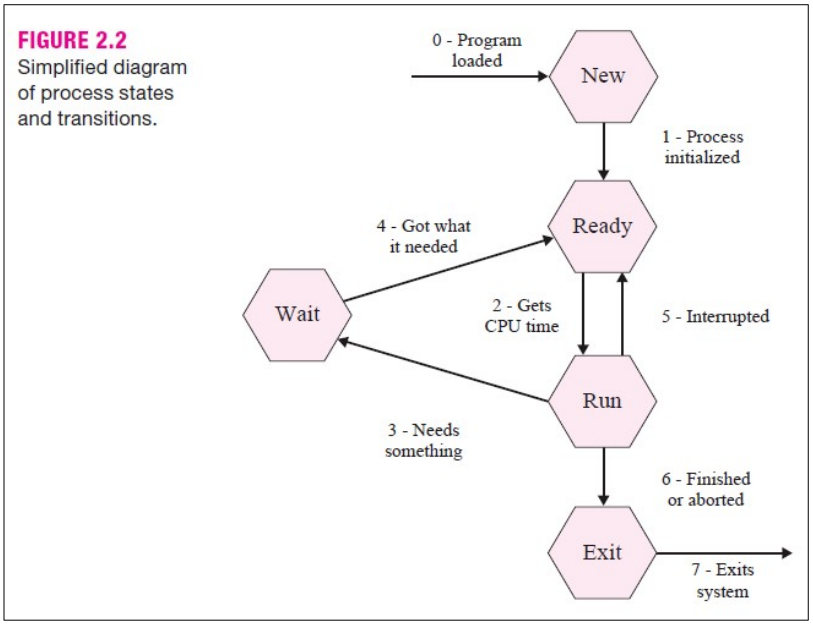
- **Código** (text): Instrucciones del proceso.
- **Datos** (data): Variables globales del proceso.
- **Memoria dinánica** (heap): Memoria dinámica que genera el proceso.
- **Pila** (stack): Utilizada para preservar el estado en la invocación anidada de procedimientos y funciones.

1.3. Estados de un proceso

El estado de un proceso es definido por la actividad corriente en que se encuentra.

Los estados de un proceso son:

- **Nuevo** (New): Cuando el proceso es creado.
- **Ejecutando** (running o Run): El proceso tiene asignado un procesador y está ejecutando sus instrucciones.
- **Bloqueado** (waiting o Wait): El proceso está esperando por un evento (que se complete un pedido de E/S o una señal).
- **Listo** (Ready): El proceso está listo para ejecutar, solo necesita del recurso procesador.
- **Finalizado** (Finished o Exit): El proceso finalizó su ejecución.

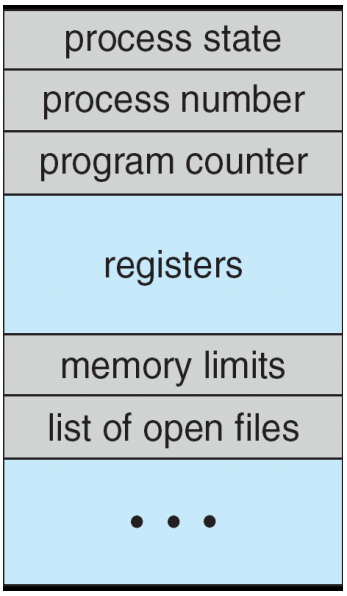


1.4. Bloque descriptor de proceso (PCB)

A nivel del sistema operativo, el Proceso se representa mediante el **Bloque de Control de Proceso (PCB)**, que incluye información como el estado de la CPU, el número de procesador asignado, entro otros.

Para ello se utiliza una estructura de datos que será el operando de las operaciones sobre procesos, recursos y del planificador (scheduler).

Los campos de esta estructura son:



- **Estado CPU**: El contenido de esta estructura estará indefinido toda vez que el proceso está en estado ejecutando (puesto que estará almacenado en la CPU indicada por procesador). Registro de flags.
- **Procesador**: [1..#CPU]: Contendrá el número de CPU que está ejecutando al proceso (si está en estado ejecutando), si no su valor es indefinido.
- **Memoria**: Describe el espacio virtual y/o real de direccionamiento según la arquitectura del sistema. Contendrá las reglas de protección de memoria así como cuál es compartida, etc..
- **Estado del proceso**: ejecutando, listo, bloqueado, etc.
- **Recursos**: Recursos de software (archivos, semáforos, etc.) y hardware (dispositivos, etc.).
- **Planificación**: Tipo de planificador.
- **Prioridad**: Podrá incluir una prioridad externa de largo alcance, o en su defecto una prioridad interna dinámica de alcance reducido.
- **Contabilización**: Información contable como cantidad de E/S, fallos de página (page faults), consumo de procesador, memoria utilizada, etc.
- **Ancestro**: Indica quién creó este proceso.
- **Descendientes**: Lista de punteros a PCBs de los hijos de este proceso

2. Creación de un proceso

Los procesos se crean a partir de otro proceso, estableciendo una relación padre-hijo. Ejemplo: el fork() de Linux.

Al creador se le denomina padre y al nuevo proceso hijo. Esto genera una jerarquía de procesos en el sistema.

En el diseño del sistema operativo, se decide qué recursos compartirán padre e hijo en el momento de la creación del nuevo proceso.

También se debe determinar qué sucede con los hijos cuando muere el padre. Pueden morir también o cambiar de padre.

Una vez creado el nuevo proceso tendrán un hilo (program counter) de ejecución propio. **El sistema genera un nuevo PCB para el proceso creado.**

2.1. Hilos o Threads

Por otro lado, los Threads (hilos) son unidades básicas de utilización de la CPU, también conocidas como procesos ligeros, que consistenten en un juego de registros y un espacio de pila.

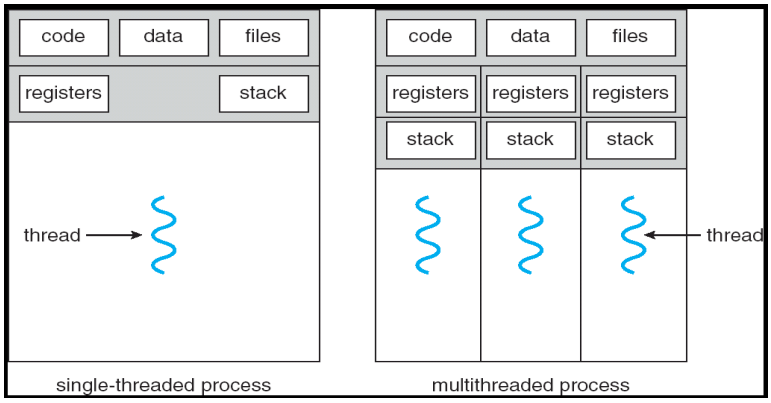
Cada thread contendrá **su propio program counter**, un conjunto de registros, un espacio para el stack y su prioridad. En cambio los threads, a diferencia de los procesos, comparten el código, los datos y los recursos con su hilo (thread) principal. En resumen, un hilo (Thread) comparte con su Hilo Principal **todos los recursos indicados en el PCB.**

Un hilo principal está formada ahora por uno o varios threads (hilos secundarios) y un hilo secundario (thread) pertenece a un solo hilo principal. Destacar que todos los recursos, secciones de código y datos son compartidos entre los distintos threads de un mismo hilo principal. Este enfoque proporciona una manera eficiente de realizar operaciones concurrentes, mejorando la utilización de los recursos de la CPU.

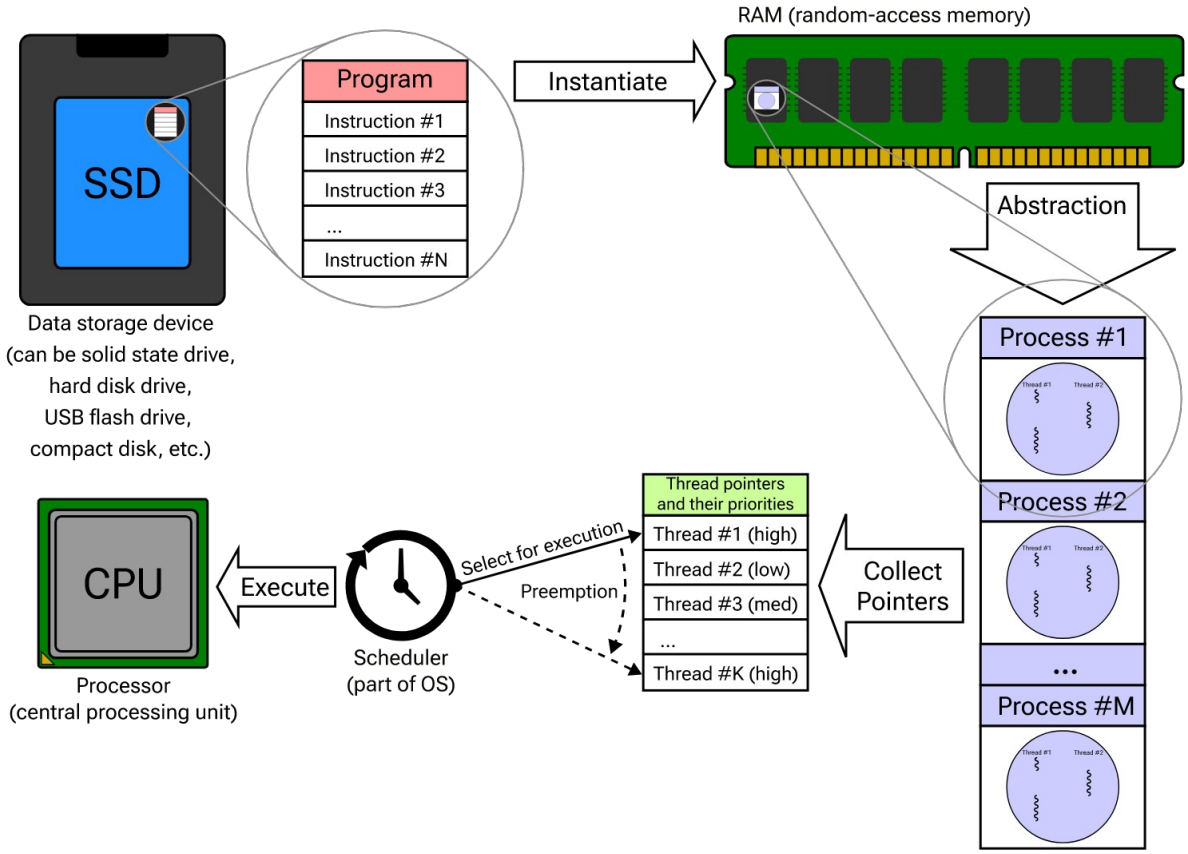
Para resumir:

- **Recursos compartidos entre los hilos**:
  - Código (instrucciones)
  - Variables globales
  - Ficheros y dispositivos abiertos
- **Recursos no compartidos entre los hilos**:
  - Contador del programa (cada hilo puede ejecutar una sección distinta de código)
  - Registros de CPU
  - Pila para las variables locales de los procedimientos a las que se invoca después de crear un hilo
  - Estado: distintos hilos pueden estar en ejecución, listos o bloqueados esperando un evento

En esta imagen se puede ver la diferencia entre un thread y varios thread. Se puede ver qué compartes y qué no (se repite), dado que cada thread al crearse tiene su registro y su stack (pila):



En le tema anterior hemos visto que un programa que se está ejecutando pasa a llamarse proceso. Cada Proceso esta compuesto por un hilo princial y cero o N hilos secundarios. La CPU sirve tanto a los procesos y a los threads según la política del Sheduler (Round Robin, SJR, FCFS, con prioridades, sin prioridades, etc.)



2.2. Para entender los threads con un símil

Imaginemos un camarero que nos prepara un desayuno: Café con leche y tostadas. (Esto es el proceso principal)

El camarero, para optimizar el tiempo y no hacer una tarea detrás de otra, hace varias a la vez.

Particionamos el proceso en hilos de ejecución. Mientras mete las rebanadas de pan en la tostadora, muele el café y sitúa una taza debajo de la cafetera. Mientras el café se deposita en la taza, calienta la leche y prepara un plato con la cuchara y el azucarillo. Pone un plato en la barra con la mantequilla, la mermelada y un cuchillo. Cuando el café está listo pone la taza en el plato y añade la leche. Mientras lleva el café a la barra saca las tostadas de la tostadora y las coloca en el plato de la mantequilla. Cuando todo está listo lleva la bandeja con todo colocado a la mesa.

En este ejemplo el hilo principal ha ejecutado varios hilos secundarios:

- Hilo secundario 1 – La tostadora: Activar la tostadora, pasar por parámetro el pan y esperar que acabe. Return: Tostadas.
- Hilo secundario 1 - Moler café: Paso por parámetro los granos de café y el hilo hace la operación de moler devolviendo café molido.

El hilo principal espera que acabe el molinillo y pone una taza en la cafetera. Ahora lanza otros 2 hilos secundarios y el también se sigue ejecuando:

- Hilo secundario 1: La máquina echa café en la taza.
- Hilo secundario 2: La leche se calienta.
- Hilo principal: prepara un plato con la cuchara y el azucarillo.

  (Los 3 hilos se ejecutan a la vez)

2.3. Ventajas de usar hilos o threads

- **Compartir recursos**: Los threads de un proceso comparten la memoria y los recursos que utilizan.
- **Economía**: Es más fácil un cambio de contexto entre threads ya que no es necesario cambiar el espacio de direccionamiento. A su vez, es más “liviano” para el sistema operativo crear un thread que crear un proceso nuevo.
- **Utilización de arquitecturas con multiprocesadores**: Disponer de una arquitectura con más de un procesador permite que los threads de un mismo proceso ejecuten en forma paralela.
- **Repuesta**: Desarrollar una aplicación con varios hilos de control (threads) permite tener un mejor tiempo de respuesta.

2.4. Desventajas del uso de hilos o threads

- **Dificulta la programación**: Al compartir todo el espacio de direccionamiento un thread mal programado puede romper el funcionamiento del resto de los threads.

3. Threads (hilos) en Python

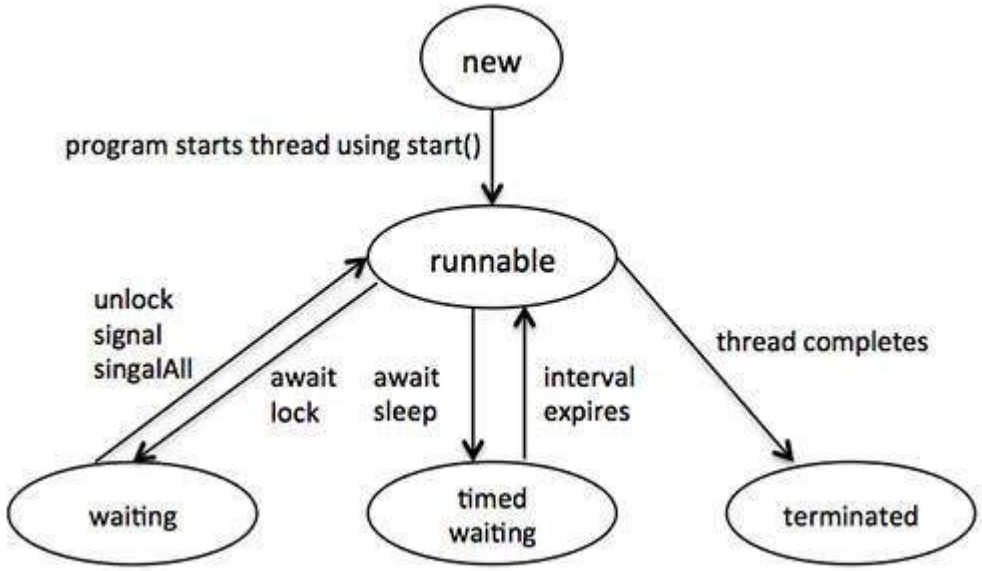
Los Threads (hilos) son una característica esencial en el desarrollo de aplicaciones concurrentes en Python, permitiendo la ejecución simultánea de múltiples tareas. Estos actúan como unidades de ejecución independientes dentro de un programa, compartiendo recursos y facilitando la realización eficiente de operaciones paralelas. En Python, la gestión de hilos se simplifica mediante el módulo `threading`, que proporciona las herramientas necesarias para crear, coordinar y controlar hilos de ejecución.

3.1. Estados de un Thread en Python

Los Threads en Python experimentan distintos estados a lo largo de su ciclo de vida, reflejando su progresión desde la creación hasta la finalización. Los estados más significativos son:

- **New** (Nuevo):
  - Descripción: El hilo se ha creado pero aún no se le han asignado todos los recursos necesarios.
  - Acciones Típicas: Inicialización del hilo y preparativos para su ejecución.
- **Runnable** (Ejecutable):
  - Descripción: El hilo ha obtenido todos los recursos requeridos y está listo para ser ejecutado en el procesador.
  - Acciones Típicas: En espera de su turno para ejecutarse en la CPU.
- **Running** (Ejecutándose):
  - Descripción: El hilo se encuentra activamente en ejecución, utilizando el procesador para llevar a cabo sus tareas asignadas.
  - Acciones Típicas: Procesamiento activo de operaciones y tareas designadas.
- **Non-running/Waiting** (No en Ejecución/Esperando):
  - Descripción: El hilo está en pausa, ya sea porque está esperando una operación de entrada/salida (E/S) o debido a la ejecución de otro hilo.
  - Acciones Típicas: Pausa temporal, esperando la finalización de operaciones o hilos secundarios.
- **Dead** (Finalizado):
  - Descripción: El hilo ha concluido su ejecución de instrucciones o ha sido finalizado explícitamente por otro hilo o el proceso principal (Hilo Principal).
  - Acciones Típicas: Liberación de recursos y cierre de las actividades del hilo.

Resumen gráfico de los estados de un Thread:



3.2. Creación y Puesta en Ejecución de Hilos en Python

La creación y gestión de hilos en Python se lleva a cabo mediante la clase Thread proporcionada por la librería threading. Aquí se describe el proceso básico para crear un nuevo hilo y ponerlo en ejecución:

- **Primero**, importar la Librería:

  Para utilizar funcionalidades relacionadas con hilos, primero, importa la librería threading.

```
import threading
```

- **Segundo**, definir la Función Asociada al Hilo:

  Crea la función que se asociará al nuevo hilo. Esta función contendrá las instrucciones que el hilo ejecutará en paralelo.

```
def mi_funcion():
    # Instrucciones del hilo
    print("¡Hola desde el hilo!")
```

- **Tercero**, crear una Instancia de la Clase Thread:

  Utiliza la clase Thread para crear una instancia del nuevo hilo. Asocia la función definida anteriormente mediante el parámetro **target**.

```
mi_hilo = threading.Thread(target=mi_funcion)
```

- **Finalmente**, iniciar la Ejecución del Hilo:

  Usa el método start() para que el nuevo hilo comience a ejecutarse en paralelo al hilo principal y, si los hubiera, con otros hilos existentes.

  El método `start()` se encarga de realizar las tareas necesarias para poner en marcha el hilo, como asignar recursos y ejecutar la función asociada en paralelo.

```
mi_hilo.start()
```

En resumen, el código completo podría verse así:

```
1  import threading
2
3  def mi_funcion():
4      # Instrucciones del hilo
5      print("¡Hola desde el hilo!")
6
7  # Crear una instancia de la clase Thread
8  mi_hilo = threading.Thread(target=mi_funcion)
9
10 # Iniciar la ejecución del hilo
11 mi_hilo.start()
```

3.3. Paralelismo en Python

Como hemos visto, los hilos (threads) son un mecanismo de programación que permite ejecutar múltiples tareas de manera concurrente dentro del mismo proceso. Cada hilo puede ejecutar una sección de código independientemente de otros hilos, lo que permite una mayor eficiencia en la ejecución de tareas que no necesitan esperar a otras tareas para completarse.

Sin embargo, la utilización de hilos también plantea desafíos, como la coordinación entre hilos, la gestión de recursos y la evitación de conflictos de datos.

Gestión de los Threads debido a los Factores No Deterministas:

Los factores deterministas son aquellos que influyen en la forma en que los hilos se ejecutan y cómo se comparten los recursos. Algunos ejemplos de factores deterministas son la prioridad de los hilos, la disponibilidad de recursos y las limitaciones de memoria. La CPU, el número de núcleos, los algoritmos de planificación (FCFS, SJF, SRT, RR,..), la cantidad de servicios y procesos que se estén ejecutando en ese momento, influyen en el orden en el que se irán atendiendo a los diferentes hilos de nuestro proceso. Estos factores no deterministas hacen que nuestro programa se comporte de forma imprevisible y cada vez nos de un resultado diferente. A este problema se le llama Condición de carrera (race condition).

Condición de carrera: Una condición de carrera es un comportamiento del software en el cual la salida depende de un orden de ejecución de eventos que no se encuentran bajo control y que puede provocar resultados incorrectos.

Consistencia de los datos:

La consistencia de los datos se refiere al problema de garantizar que los datos compartidos entre los hilos sean correctos e invariantes, incluso cuando varios hilos intentan acceder a ellos al mismo tiempo. La inconsistencia de los datos puede ocurrir cuando los hilos cambian los datos de manera simultánea y no se sigue una regla clara para determinar qué hilo debe tener acceso a los datos en primer lugar. Este problema es especialmente problemático en programas concurrentes, ya que los hilos pueden acceder a los mismos datos y alterarlos de manera independiente. Si no se manejan adecuadamente, los conflictos de datos pueden provocar errores y incoherencias en la ejecución del programa.

La Complejidad de la Ejecución Concurrente:

En resumen, mientras que los hilos ofrecen un enfoque poderoso para mejorar la concurrencia en las aplicaciones, también introducen desafíos en términos de gestión y control. Los programadores deben implementar estrategias efectivas para la sincronización y la prevención de condiciones de carrera a fin de garantizar resultados predecibles y consistentes. La comprensión de los factores no deterministas asociados con la gestión de hilos es esencial para desarrollar aplicaciones robustas y eficientes en entornos concurrentes.

En Python hay varias maneras de crear N hilos.

a) Se puede crear cada hilo de manera individual:

```python
import threading
import time

def tarea_hilo(identificador):
    print(f"Hilo {identificador} ejecutando...")
    time.sleep(1)


if __name__ == "__main__":

    # Crear e iniciar 10 hilos sin utilizar append
    hilo1 = threading.Thread(target=tarea_hilo, args=(1,))
    hilo2 = threading.Thread(target=tarea_hilo, args=(2,))
    hilo3 = threading.Thread(target=tarea_hilo, args=(3,))
    hilo4 = threading.Thread(target=tarea_hilo, args=(4,))
    hilo5 = threading.Thread(target=tarea_hilo, args=(5,))
    hilo6 = threading.Thread(target=tarea_hilo, args=(6,))
    hilo7 = threading.Thread(target=tarea_hilo, args=(7,))
    hilo8 = threading.Thread(target=tarea_hilo, args=(8,))
    hilo9 = threading.Thread(target=tarea_hilo, args=(9,))
    hilo10 = threading.Thread(target=tarea_hilo, args=(10,))

    # Iniciar cada hilo individualmente
    hilo1.start()
    hilo2.start()
    hilo3.start()
    hilo4.start()
    hilo5.start()
    hilo6.start()
    hilo7.start()
    hilo8.start()
    hilo9.start()
    hilo10.start()
```

Como puedes ver no es muy escalable. Es por eso que se usa una lista para gestionar los hilos ofrece una manera flexible y dinámica de trabajar con múltiples hilos en Python.

b) Crear hilos usando una lista:

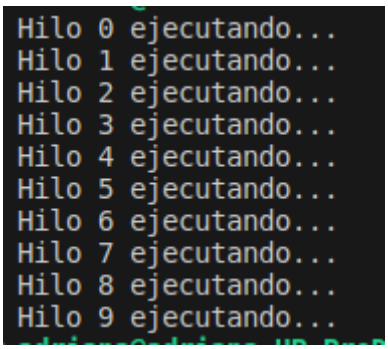
```python
import threading
import time

def tarea_hilo(identificador):
    print(f"Hilo {identificador} ejecutando...")
    time.sleep(1)

if __name__ == "__main__":
    # Lista para almacenar los hilos
    hilos = []

    # Crear e iniciar varios hilos
    for i in range(10):
        nuevo_hilo = threading.Thread(target=tarea_hilo, args=(i,))
        hilos.append(nuevo_hilo)
        nuevo_hilo.start()
```

El resultado de ambas ejecuciones es el mismo:



Más adelante hablaremos de los parámetros del constructor de la clase Thread, pero dado que ya se ve en este ejemplo, os comento el el parámetro 'args'.

Hemos comentado que target sirve para indicar qué función vamos a ejecutar de forma paralela (es la función que ejecutaran los hilos) y args se utiliza para pasar argumentos posicionales a una función. Este parámetro permite que una función acepte un número variable de argumentos, lo que hace que la función sea más flexible y reutilizable.

En este ejemplo:

- target=tarea_hilo: Se especifica la función objetivo del hilo como "tarea_hilo".
- args=("Valor1", "Valor2"): Se pasa una tupla de argumentos ("Valor1" y "Valor2") a la función "tarea_hilo". En este caso sólo un valor que indica el orden en que se va crado 1 (primero), 2 (segundo), 3 (tercero), …..

Cuando el hilo se inicia con hilo.start(), la función "tarea_hilo" se ejecutará en un hilo secundario con los argumentos proporcionados. El hilo principal ejecuta en nuestro caso, todo lo que se encuentra en el main. Todos los hilos ven todo el código, pero sólo ejecutan la parte de código que les toca.

Para resumir, destacar las ventajas al utilizar una lista para crear muchos hilos, en lugar de crear hilos de manera individual:

- **Facilita la Escalabilidad**:

  Al utilizar una lista y un bucle, puedes escalar fácilmente el número de hilos sin tener que duplicar o triplicar líneas de código. Esto hace que el código sea más limpio y más fácil de mantener a medida que aumenta el número de hilos.
- **Dinamismo en la Creación de Hilos**:

  La lista permite que la creación de hilos sea más dinámica. Puedes modificar fácilmente la cantidad de hilos ajustando el tamaño de la lista o generando la lista de manera programática.
- **Facilita la Gestión**:

  Tener hilos almacenados en una lista facilita la gestión y manipulación posterior. Puedes realizar operaciones en todos los hilos de la lista, como iniciarlos, detenerlos o realizar un seguimiento de su estado, de manera más sencilla.

- **Mejora la Legibilidad:**

  Utilizar una lista con append mejora la legibilidad del código. En lugar de repetir líneas de código casi idénticas para cada hilo, puedes ver claramente que estás creando y gestionando una colección de hilos.
- **Facilita la Espera de Hilos** (lo veremos más adelante):

  Almacenar hilos en una lista facilita la espera hasta que todos los hilos hayan completado su ejecución. Puedes utilizar el método join de manera más eficiente en una lista de hilos.

3.4. Hilo principal – Hilo secundario

La ejecución de hilos puede ser no determinista debido a factores como el número de núcleos del ordenador, la versión del sistema operativo y la carga del sistema. Sin mecanismos de control, los hilos pueden ejecutarse en cualquier orden, lo que puede causar resultados inesperados o incluso errores en la lógica del programa.

Ejemplo: (Para simular programas más largos usaremos la función de sleep())

```
import threading
import time
import random

def tarea_hilo(identificador):
    tiempo_espera = random.randint(1, 5)
    time.sleep(tiempo_espera)
    print(f"Soy el hilo secundario con id = {identificador} y he dormido {tiempo_espera} segundos antes de acabar.")


if __name__ == "__main__":
    # Crear una lista para almacenar los hilos
    hilos = []

    # Crear e iniciar 5 hilos utilizando append
    for i in range(1, 6):
        nuevo_hilo = threading.Thread(target=tarea_hilo, args=(i,))
        hilos.append(nuevo_hilo)
        nuevo_hilo.start()

    for j in range(5):
        print("Soy el hilo principal y estoy ejecutadno el for. Estoy en la vuelta: ",j)
        time.sleep(1)
```

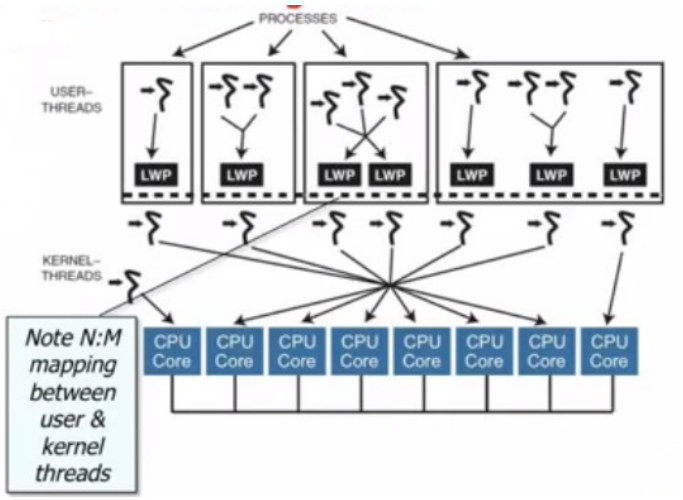
El resultado de este programa es:

```
Soy el hilo principal y estoy ejecutadno el for. Estoy en la vuelta:  0
Soy el hilo principal y estoy ejecutadno el for. Estoy en la vuelta:  1
Soy el hilo principal y estoy ejecutadno el for. Estoy en la vuelta:  2
Soy el hilo secundario con id = 5 y he dormido 3 segundos antes de acabar.
Soy el hilo principal y estoy ejecutadno el for. Estoy en la vuelta:  3
Soy el hilo secundario con id = 1 y he dormido 4 segundos antes de acabar.
Soy el hilo secundario con id = 3 y he dormido 4 segundos antes de acabar.
Soy el hilo secundario con id = 4 y he dormido 4 segundos antes de acabar.
Soy el hilo principal y estoy ejecutadno el for. Estoy en la vuelta:  4
Soy el hilo secundario con id = 2 y he dormido 5 segundos antes de acabar.
```

Como podéis ver la CPU va sirviendo tanto al hilo principal como a todos los hilos secundarios creados. Por eso se mezclan los mensajes.

Veremos más adelante como controlar y separar los mensajes de los hilos secundarios y del hilo principal.

(Os dejo este programa en el aules para que lo probéis. En cada ejecución debería dar resultados diferentes)



Repaso de conceptos:

- Tenemos 1 **hilo principal** que crea 5 threads (5 hijos). El hilo principal imprime el mensaje: "Soy el hilo principal y estoy ejecutadno el for. Estoy en la vuelta: "
- Tenemos 5 Threads (**hilos secundarios**) que tienen un tiempo de espera random y cada uno imprime este mensaje: "Soy el hilo secundario con id = {identificador} y he dormido {tiempo_espera} segundos antes de acabar."
  - Los 5 hilos secundarios **tiene 1 padre** (es el hilo principal que los ha creado)

3.5. El constructor de la clase Thread

El constructor de la clase `Thread` en Python ofrece una serie de argumentos que permiten personalizar y gestionar los hilos de manera eficiente.

Vamos a ver algunos argumentos de la case Thread:

- **group:**debe ser None; reservado para una futura extensión cuando se implemente una clase ThreadGroup.
- **Target:** es el objeto invocable a ser invocado por el método run(). Por defecto es None, lo que significa que nada es llamado.
- **Name:** es el nombre del hilo. De forma predeterminada, se construye un nombre único con el formato «Hilo-N», donde N es un número decimal pequeño, o «Hilo-N (target)» donde «target» es target.__name__ si se especifica el argumento target.
- **args:** es la tupla de argumento para la invocación objetivo. Por defecto es ().
- **kwargs:** es un diccionario de argumentos de palabra clave para la invocación objetivo. Por defecto es {}.

El argumento args:

Vamos a trabajar primero algo más el argumento args:

```python
import threading
import time

def tarea_animal(nombre, patas, vuela):
    vuelo = "vuela" if vuela else "no vuela"
    informacion = f"{nombre} es un animal con {patas} patas y {vuelo}."
    print(informacion)

# Crear e iniciar hilos para diferentes animales con patas y si vuelan
hilo_perro = threading.Thread(target=tarea_animal, args=("Perro", 4, False))
hilo_pajaro = threading.Thread(target=tarea_animal, args=("Pájaro", 2, True))
hilo_araña = threading.Thread(target=tarea_animal, args=("Araña", 8, False))

# Iniciar los hilos
hilo_perro.start()
hilo_pajaro.start()
hilo_araña.start()
```

La salida de este programa sería:

```
Perro es un animal con 4 patas y no vuela.
Pájaro es un animal con 2 patas y vuela.
Araña es un animal con 8 patas y no vuela.
```

El argumento name:

Se puede consultar y modificar le nombre que pone por defecto el constructor. Veremos primero cómo se modifica en la llamada:

```
import threading
import time

def funcion_hilo(color):
    nombre_hilo = threading.current_thread().name
    print(f"Soy el hilo con el nombre {nombre_hilo}. Mi color preferido es el: {color} .")
    time.sleep(2)
    print(f"Hilo {nombre_hilo} completado.")

# Crear e iniciar hilos con nombres personalizados
hilo1 = threading.Thread(target=funcion_hilo, args=("azul",))
hilo2 = threading.Thread(target=funcion_hilo, args=("rojo",), name="Hilo-B")
hilo3 = threading.Thread(target=funcion_hilo, args=("verde",), name="Hilo-C")

# Iniciar los hilos
hilo1.start()
hilo2.start()
hilo3.start()
```

Las salida del programa es:

```
Soy el hilo con el nombre Thread-1. Mi color preferido es el: azul .
Soy el hilo con el nombre Hilo-B. Mi color preferido es el: rojo .
Soy el hilo con el nombre Hilo-C. Mi color preferido es el: verde .
Hilo Thread-1 completado.
Hilo Hilo-B completado.
Hilo Hilo-C completado.
```

El primer thread, al cual no se le ha cambiado el nombre, tiene el que le ha puesto el Sistema: “Thread-1”. Los otros dos threads tienen el nombre que le hemos puesto nosotros.

Otra manera de modificar el nombre es usando la instancia de la clase Thread e ir a su argumento directamente.

```python
import threading
import time

def funcion_hilo():
    nombre_hilo = threading.current_thread().name
    print(f"Soy el hilo con el nombre {nombre_hilo}.")

if __name__ == "__main__":
    # Crear una lista para almacenar los hilos
    hilos = []
    # Crear e iniciar 10 hilos utilizando append
    for i in range(1, 11):
        nuevo_hilo = threading.Thread(target=funcion_hilo)
        if i == 4:
            #ponemos un nombre especial al hilo 4
            nuevo_hilo.name = "Hilo_cuatro"
        elif i != 6:
            #al resto de hilos, a excepción del hilo 6,
            # le pomnemos el nombre Hilo seguido del número de iteracion
            nuevo_hilo.name = "Hilo_"+str(i)
        # Iniciar los hilos
        hilos.append(nuevo_hilo)
        nuevo_hilo.start()
```

La salida del programa es:

```
Soy el hilo con el nombre Hilo_1.
Soy el hilo con el nombre Hilo_2.
Soy el hilo con el nombre Hilo_3.
Soy el hilo con el nombre Hilo_cuatro.
Soy el hilo con el nombre Hilo_5.
Soy el hilo con el nombre Thread-6.
Soy el hilo con el nombre Hilo_7.
Soy el hilo con el nombre Hilo_8.
Soy el hilo con el nombre Hilo_9.
Soy el hilo con el nombre Hilo_10.
```