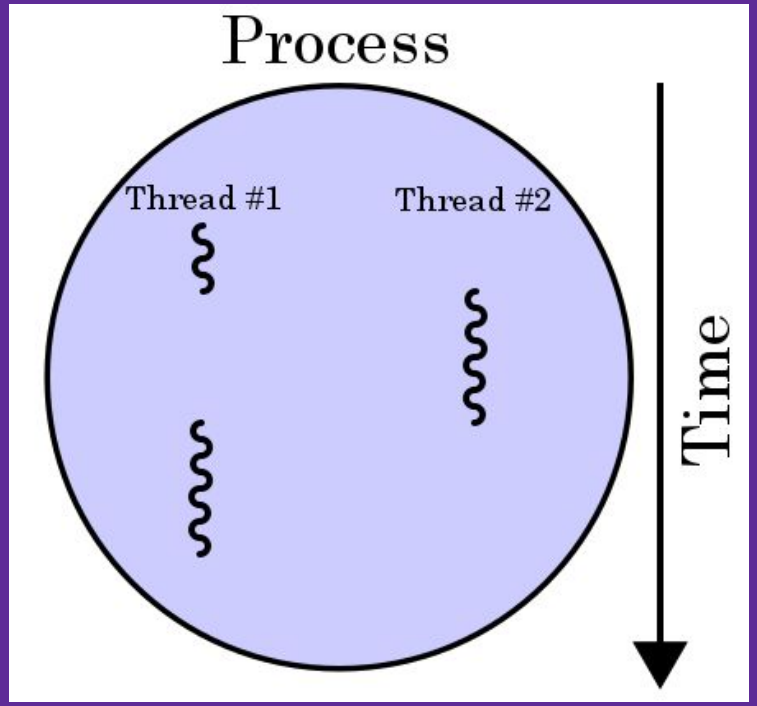
Programación Avanzada
IIC2233 2025-2

Cristian Ruz - Pablo Araneda - Francisca Ibarra - Tamara Vidal - Daniela Concha

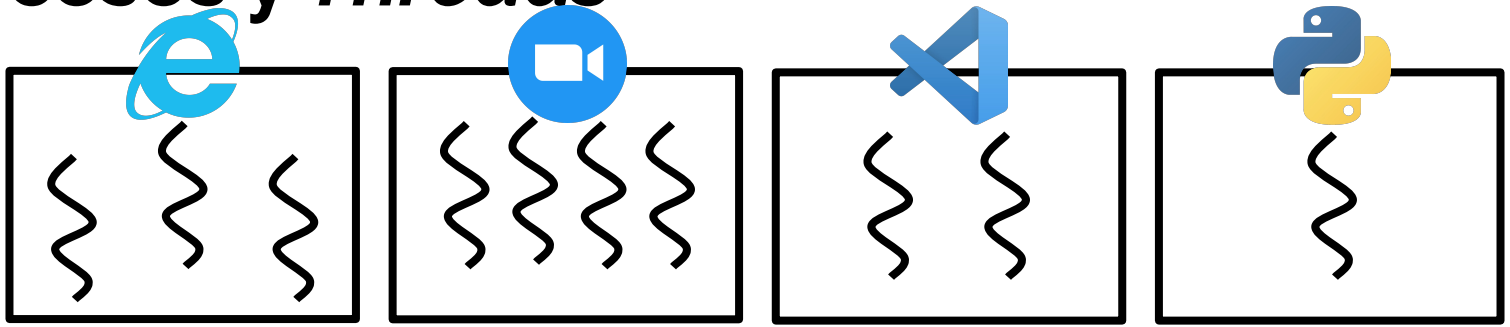
Anuncios

1. Hoy tenemos la Actividad 5.
2. Recuerden que la ECA abre de domingo a martes. Si responden 10 veces tienen un bonus en el promedio final.

Paralelismo
Y
Concurrencia



Paralelismo y Concurrencia: Procesos y *Threads*



- ¿Cual es la principal funcionalidad del Paralelismo y la Concurrencia?
- ¿Qué es la Concurrencia? ¿Qué es el Paralelismo? ¿En qué se diferencian uno del otro?
- La librería *threading*, ¿cuál utiliza?
- ¿Cómo podemos obtener paralelismo real en Python?

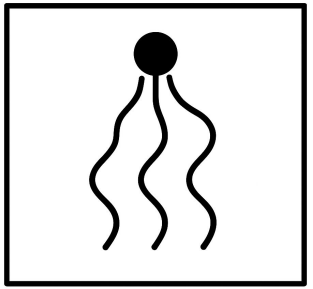
Threads

- `start()`
- `run()`

- *Thread* principal
- Otros *threads*

Threads: Creando un nuevo *Thread*

- Cuando creas un *thread* de manera funcional, ¿qué es lo mínimo que deberías definir para que el *thread* funcione?
- Cuando creas un *thread* heredando de *Thread,* ¿qué método debes sobreescribir para definir el comportamiento principal del *thread*?
- Supongamos que inicias un *thread*, ¿cómo puedes reconocer si ha terminado?



Threads: Definición de *Threads*

Tenemos **2 maneras** de implementar *Threads*. Ya sea **definiendo** la **función objetivo** de un *Thread* y pasándosela, o **creando** nuestra **propia clase** que herede de *Thread*.

```python
from threading import Thread


def funcion():
    # Secuencia de instrucciones
    ...



t = Thread(target=funcion)
t.start()
```

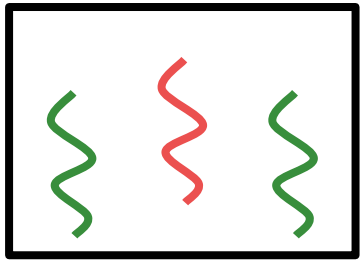
```python
from threading import Thread


class MiThread(Thread):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

    def run(self):
        # Este método inicia el thread
        # cuando ejecutamos el método start()
        print(f"{self.name} partiendo...")

t = MiThread()
t.start()
```

Sincronización de recursos

- `lock()`
- `set()`
- `wait()`

- Operación atómica

Sincronización de recursos: Entendiendo conceptos claves



En un *thread,* ¿qué es la sección crítica?

Si tenemos un cajero automático donde múltiples usuarios (*threads*) intentan retirar dinero de una misma cuenta, ¿qué sucedería si no utilizáramos sincronización?

Al comparar un caso en que se utilice *Context Manager* (`with lock:`) con una implementación donde se maneje el `Lock` manualmente con `acquire()` y `release()`: ¿Qué método posee más ventajas sobre el otro?

¿Qué estrategias existen para evitar *deadlocks* en programas con múltiples *threads*?

Para la situación en que un Estudiante pregunta constantemente a un Profesor cuando estarán la nota de su prueba, pero un Profesor debe esperar a que un Ayudante corrija la prueba. Utilizando Event ¿Quienes de ellos usarían los métodos set, is_set y wait?

Sincronización de recursos: Logrando sincronización con Lock( )

En este caso, con el uso de *locks* podemos asegurar que solo 1 *thread* a la vez modifique el contador global "`Commits.publicados`".

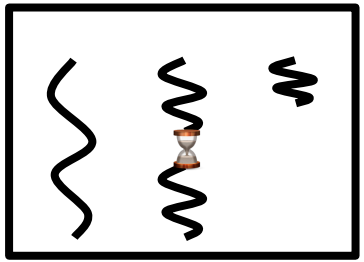
```
class Commits(Thread):
    publicados = 0
    lock = Lock()

    def __init__(self, commits, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.commits = commits

    def notificar_commit(self, commit):
        self.subir(commit)
        with self.lock:
            Commits.publicados += int(1)
        time.sleep(10)

    def run(self):
        for commit in self.commits:
            self.notificar_commit(commit)
```

Dependencia entre *threads*

- `join()`
- `daemon`

Dependencia entre threads: Entendiendo la dependencia

- ¿Qué tipo de *thread* puede esperar a otro *thread*?
- Si tenemos un *Thread daemon* ¿Qué ocurre si el *Main Thread* termina? ¿Y en caso de que no sea *daemon*?



Implementar *daemon*

Con el uso de ***daemon=True*** hacemos que “la vida del *thread*” acabe cuando el programa principal termina.

De este modo, esperamos “1 minuto” y todos los *threads* dejan de ejecutarse.

```
class Commits(Thread):
    def __init__(self, commits, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.commits = commits
        self.daemon = True

    def notificar_commit(self, commit):
        ...

    def run(self):
        for commit in self.commits:
            self.notificar_commit(commit)
```

```
for sección in range(1, 6):
    commit_s = filter(lambda c: c.sec = sección, commits)
    thread_commit = Commits(commit_s)
    thread_commit.start()

time.sleep(60) # Esperamos 60 seg. y el programa termina.
```

En resumen

- Con *Thread* podemos asegurar paralelismo.
- Con *Lock* y eventos, podemos detener los *threads* para que esperen cierta acción o para asegurar que solo 1 *thread* ejecute a la vez cierto código.
- Con *join* y *daemon*, podemos permitir que la ejecución de un *thread* dependa de otro *thread* o del programa principal.

Veamos una pregunta de Evaluación Escrita

Tema: Threading (Midterm 2024-1)

16. Respecto a *threading*, ¿cuál o cuáles afirmaciones son **incorrectas**?

    I. Si dentro de una función se utiliza un `lock.acquire()`, no es necesario hacer `lock.release()` porque al momento de finalizar la función, el *lock* será liberado automáticamente.

    II. Varios *threads* pueden esperar a un mismo `threading.Event`.

    III. El método `.join()` es utilizado por la clase `threading.Event` para indicarle al *thread* que debe esperar hasta que el evento finalice.

    A) Solo I
    B) Solo II
    C) Solo III
    D) I y III
    E) II y III

Veamos una pregunta de Evaluación Escrita

Tema: Threading (Midterm 2024-1)

16. Respecto a *threading*, ¿cuál o cuáles afirmaciones son **incorrectas**?

I. Si dentro de una función se utiliza un `lock.acquire()`, no es necesario hacer `lock.release()` porque al momento de finalizar la función, el lock será liberado automáticamente.

II. Varios *threads* pueden esperar a un mismo `threading.Event`.

III. El método `.join()` es utilizado por la clase `threading.Event` para indicarle al *thread* que debe esperar hasta que el evento finalice.

A) Solo I
B) Solo II
C) Solo III
D) I y III
E) II y III

Actividad

Recuerden que es una evaluación individual y oficial.

Pediremos SILENCIO y se respetará la ética.

Comentarios AC5

NO GIT PUSH NO GAIN!

- En caso de que les salga `TimeoutError`, pero la ejecución del código sea menor a 1 segundo, entonces comenten el `@timeout` del test.

Programación Avanzada
IIC2233 2025-2

Cristian Ruz - Pablo Araneda - Francisca Ibarra - Tamara Vidal - Daniela Concha