Programación Avanzada
IIC2233 2024-2

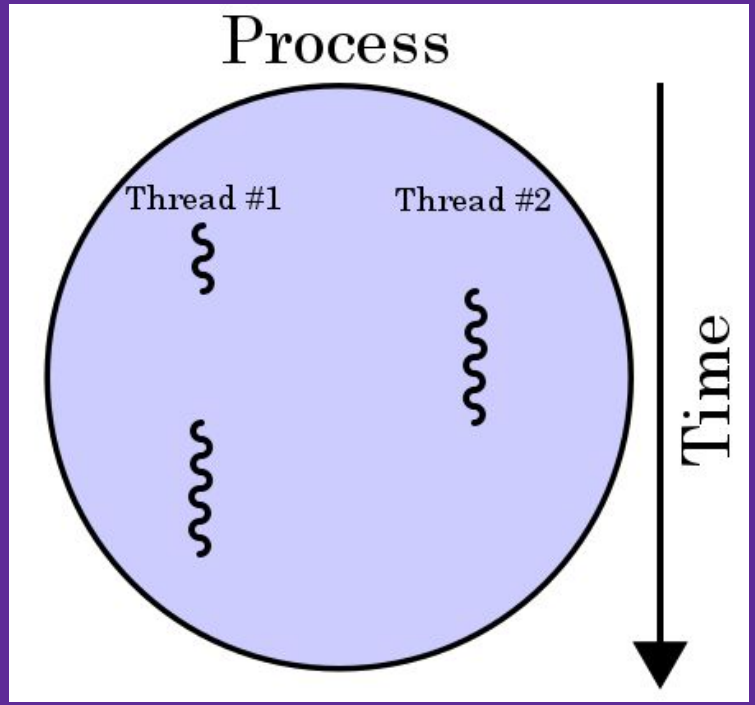
Hernán Valdivieso - Daniela Concha - Francisca Ibarra - Lucas Van Sint Jan - Francisca Cattan

Anuncios

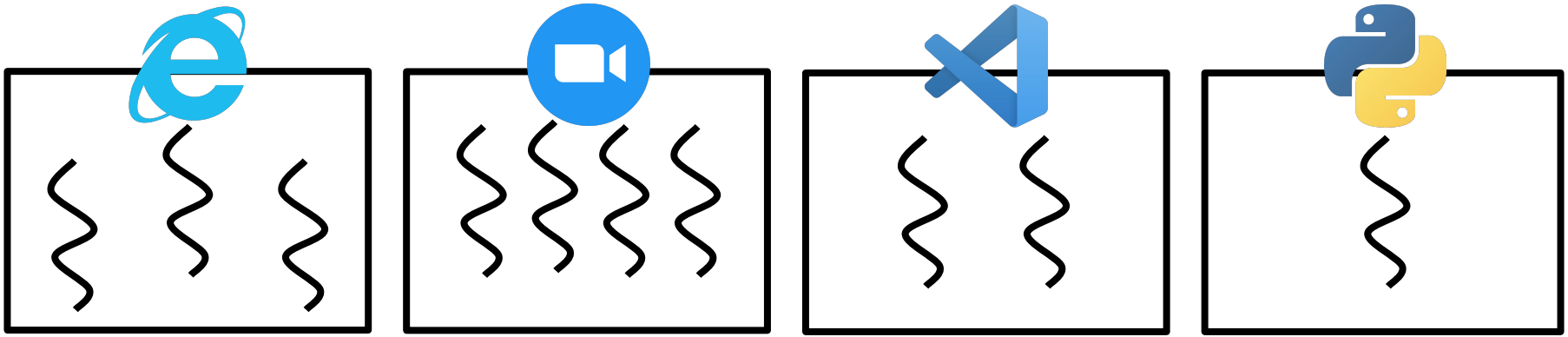
1. Hoy tenemos la cuarta actividad que se entrega mañana a las 23:59.
2. Hoy, 3 de octubre, es el *midterm*. Empezará a las 17:30. Dado lo anterior, la **clase finalizará a las 17:00.**

2. Hoy, 3 de octubre, es el *midterm*. Empezará a las 17:30. Dado lo anterior, la **clase finalizará a las 17:00.**

Paralelismo
(o concurrencia)



Paralelismo: Procesos y *Threads*



DCCommits

Parte A

DCCommits (Parte A)

Necesitamos notificar los *commits* a corregir, pero el código actual solo permite notificar de a uno a la vez.

¿Cómo podemos mejorar esto? Queremos que se publiquen *commits* de las 5 secciones simultáneamente.

```
class Commits:
    publicados = 0

    def __init__(self, commits):
        ...

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

    def publicar(self):
        for commit in self.commits:
           self.notificar_commit(commit)

c = Commits(commits)
c.publicar()
```

DCCommits (Parte A)

Necesitamos notificar los *commits* a corregir, pero el código actual solo permite notificar de a uno a la vez.

¿Cómo po
que se pub
secciones

¡Necesitamos *Threads*!

```
class Commits:
    publicados = 0

    def __init__(self, commits):


                                    commit):

                                    int(1)


    def publicar(self):
        for commit in self.commits:

self.notificar_commit(commit)

c = Commits(commits)
c.publicar()
```

DCCommits (Parte A)

Necesitamos notificar los *commits* a
corregir, pero el código actual
notificar de a uno a la vez.

¿Cómo po
que se pub
secciones

¡Ne ds!


```
class Commits:
    os = 0

    it__(self, commits):


                commit):

    int(1)


    icar(self):
        commit in self.commits:

            self.notificar_commit(commit)

c = Commits(commits)
c.publicar()
```

Threads

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

- *Thread* principal
- Otros *threads*

Threads

```
from threading import Thread


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




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

```
from threading import Thread

# ¡Importante heredar!
class MiThread(Thread):
    def __init__(self, *args, **kwargs):
        # ¡Importante el super!
        super().__init__(*args, **kwargs)

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


t1 = MiThread()
t2 = MiThread()
t1.start()
t2.start()
print(1)
```

Threads

Cada *Thread* se encargará de notificar los *commits* de una sección y de aumentar en 1 el contador global "`Commits.publicados`".

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

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

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

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

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

DCCommits

Parte B

DCCommits (Parte B)

Ahora cada *commit* se sube de manera independiente entre las 5 secciones.

Pero … **la cantidad total de *commits* publicados no calza con la cantidad real de *commits*** 😨

```
- C1 lee 0 de Commits.publicados
- C1 se pausa

…

- C2 lee 5 de Commits.publicados
- C2 suma 1 => 6
- C2 guarda 6 en Commits.publicados
- C2 se pausa


- C1 se reanuda
- C1 suma 1 => 1 (😨)
- C1 guarda 1 en Commits.publicados (😨)
```

DCCommits (Parte B)

Ahora cada *commit* se sube de manera independiente entre las 5 secciones.

Pero … **la**
publicado
de *commi*

¡Necesitamos sincronizar los accesos a este contador!

```
- C1 lee 0 de Commits.publicados
- C1 se pausa
                                    s
                                    ados
- C1 se reanuda
- C1 suma 1 => 1 (😨)
- C1 guarda 1 en Commits.publicados (😨)
```

DCCommits (Parte B)

Ahora cada *commit* se sube de manera independiente entre las 5 secci

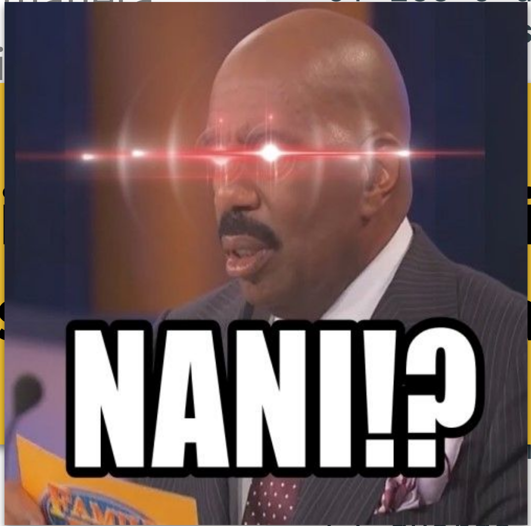
Pero … **la**

publicado

de *commit*

¡Necesi izar los

acces dor!



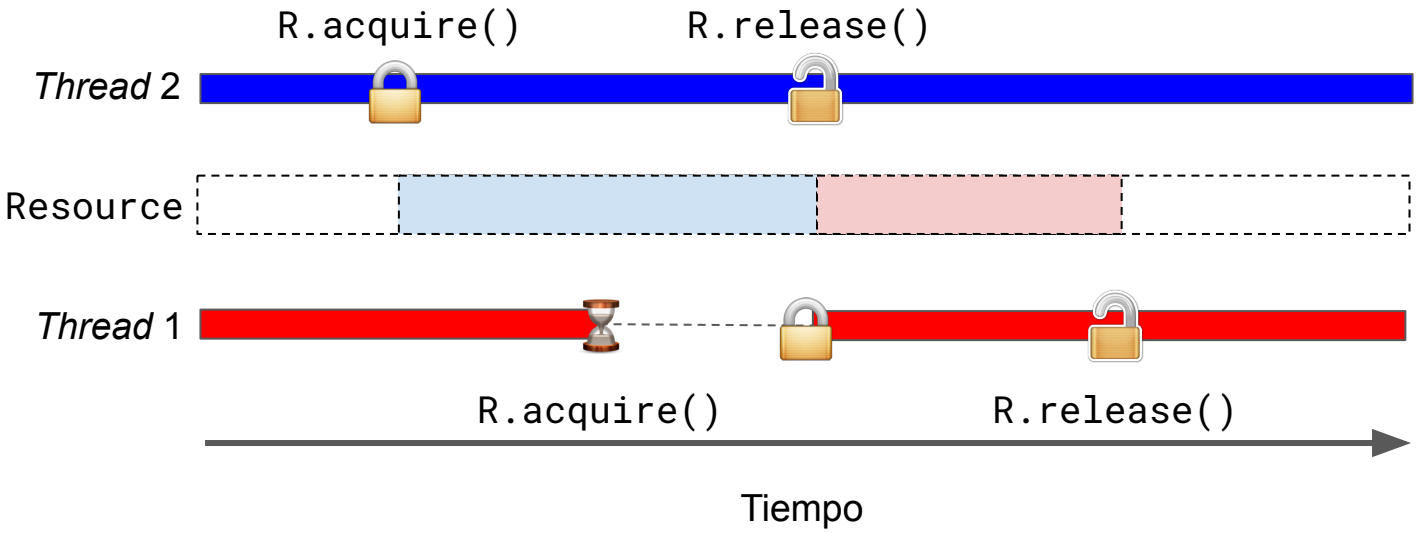

```
- C1 lee 0 de Commits.publicados
                 a
                                                        s
                                                        ados
                 uda
                 => 1 (😨)
- C1 guarda 1 en Commits.publicados (😨)
```

Sincronización de recursos

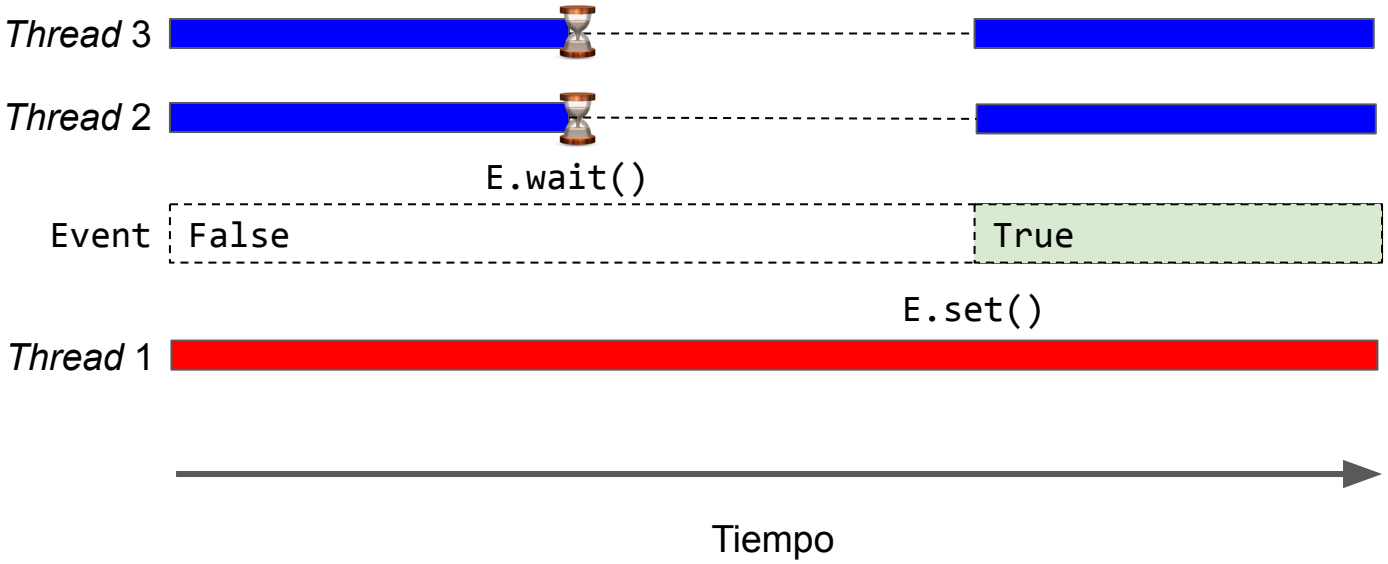
- `lock()`
- `set()`
- `wait()`

- Operación atómica

Lock()


R.acquire()
R.release()
Thread 2
Resource
Thread 1
R.acquire()
R.release()
Tiempo

Event: set() - wait()



Operaciones atómicas

También llamadas operaciones ***thread-safe*** son acciones en Python que no pueden ser interrumpidas a la mitad.

- Asignar valores
  - `x = 1`
- Obtener dato de una lista
  - `lista[2]`
- Agregar dato de una lista
  - `lista.append(2)`

Mientras que hay operaciones que no son ***thread-safe***.

- Realizar más de una acción al mismo tiempo:
  - `lista.append(lista[0])`
  - `x = x + 1`**

**Desde Python 3.10, esta operación fue optimizada, pero no en todos los lenguajes o versiones es así.

Lock()

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

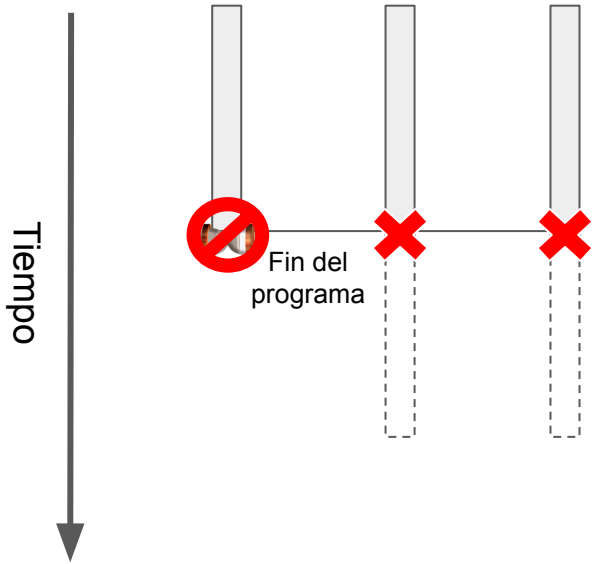
```
class Commits(Thread):
    publicados = 0
    lock = Lock()

    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)
```

DCCommits

Parte C

DCCommits (Parte C)

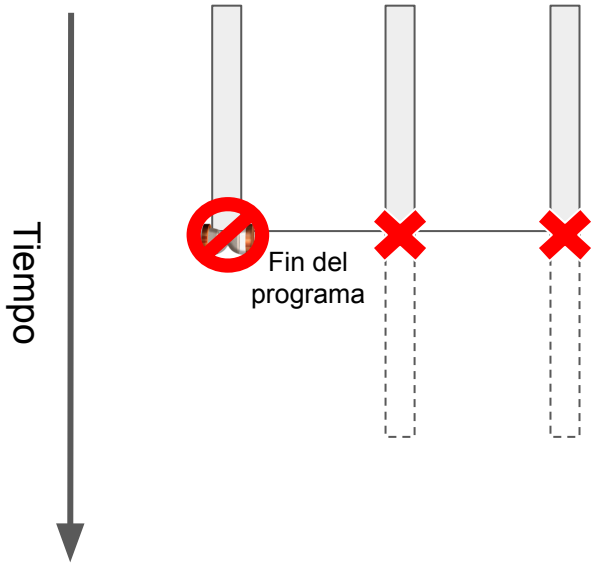
El sindicato nos exige trabajar 1 hora diaria, así que solo publicaremos *commits* durante ese tiempo y luego hay que bajar el sistema.



DCCommits (Parte C)

El sindicato nos exige trabajar 1 hora diaria, así que solo publicaremos *commits* durante ese tiempo y luego hay que bajar el sistema.

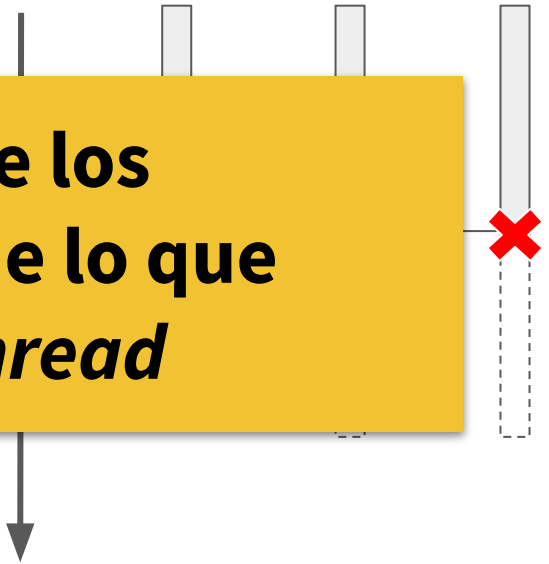
Necesitamos **un mecanismo para detener la publicación de *commits* que no alcanzaron a subirse luego de 1 hora.**



DCCommits (Parte C)

El sindicato nos exige trabajar 1 hora diaria, así que solo publicaremos *commits* durante es
el sistema.

Necesitam
la publica
alcanzaron a subirse luego de 1 hora.



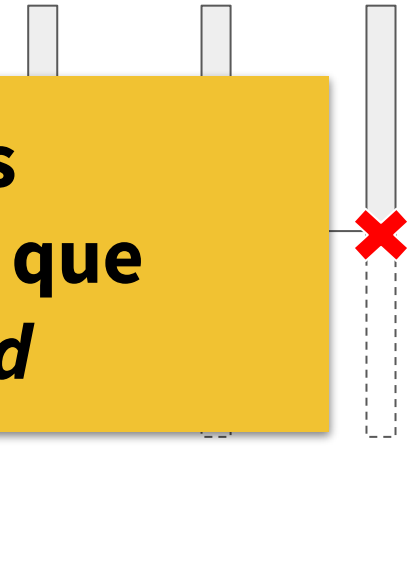
Necesitamos que los *threads* dependan de lo que ocurre en otro *thread*

DCCommits (Parte C)

El sindicato nos exige trabajar 1 hora diaria, así que solo publicaremos durante es el sistema.

Necesitam **la publica alcanzaron a subirse luego de 1**

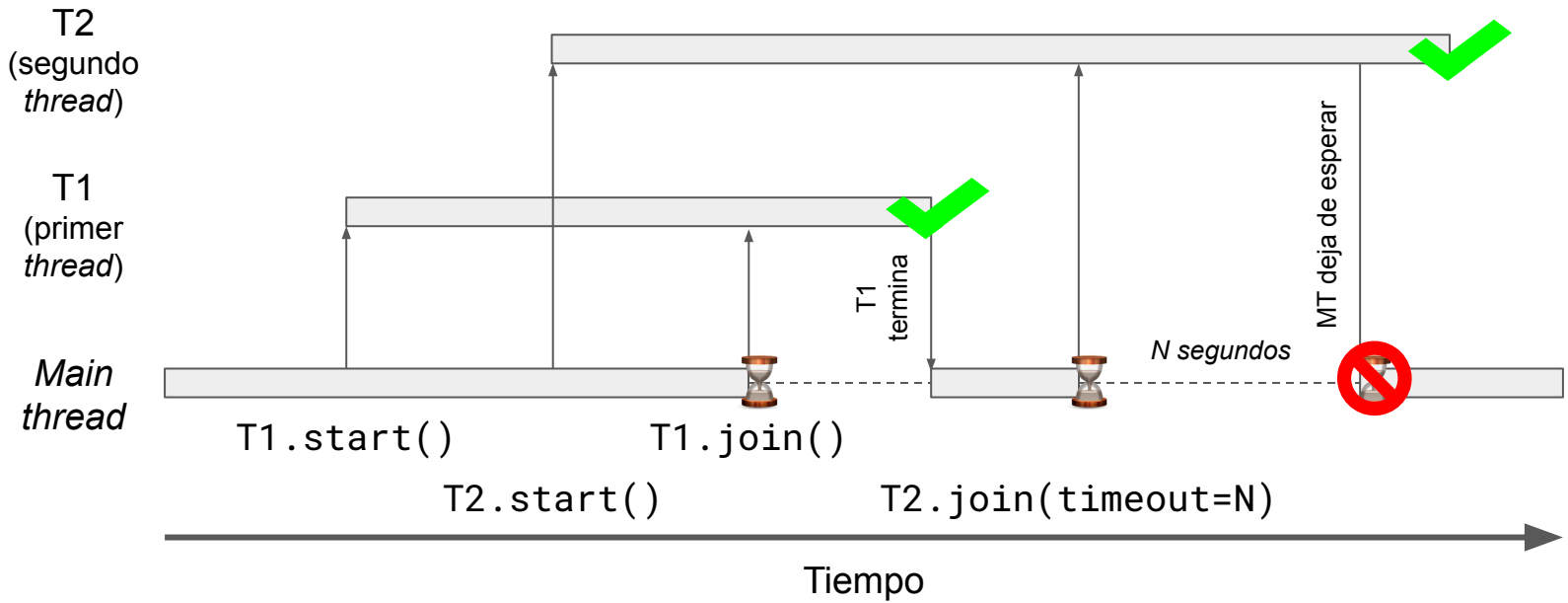
Nec ***threads*** **ocu** **los** **e lo que** ***read***



Dependencia entre *threads*

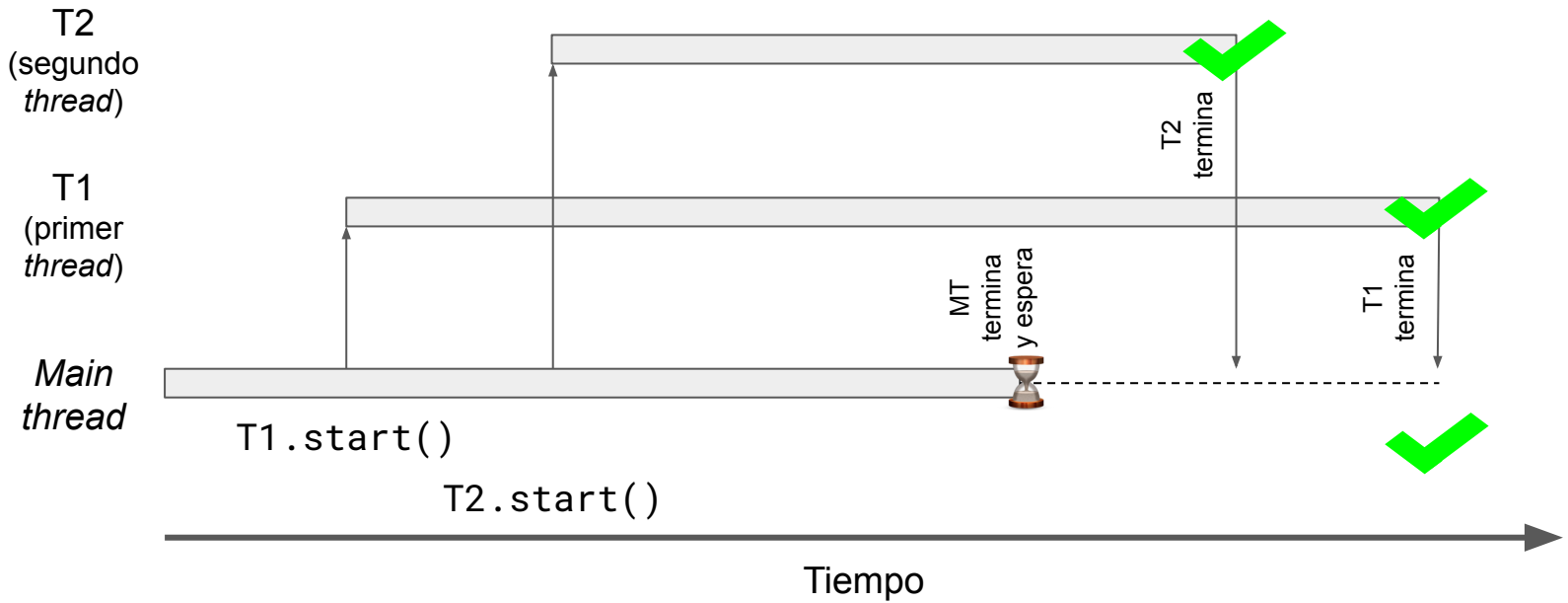
- `join()`
- `daemon`

join()

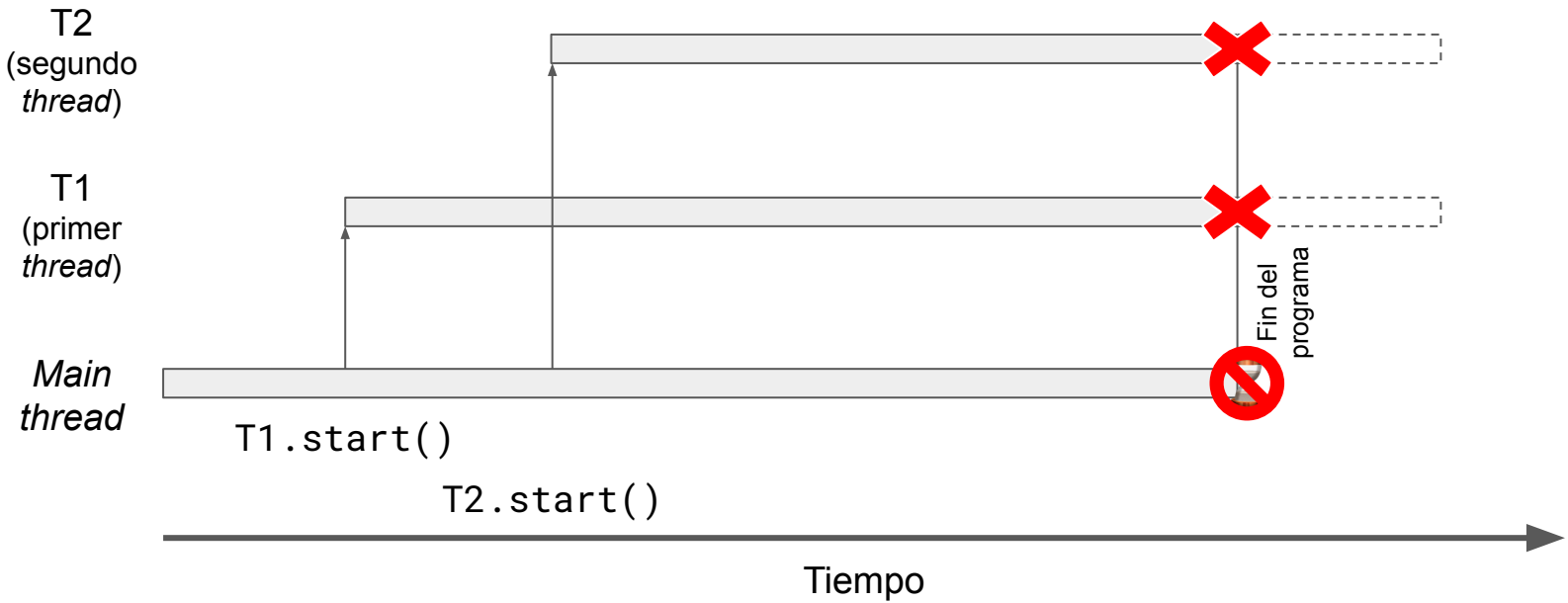

T2
(segundo
thread)
T1
(primer
thread)
Main
thread
T1
termina
MT deja de esperar
N segundos
T1.start()
T1.join()
T2.start()
T2.join(timeout=N)
Tiempo

daemon=False

Comportamiento por **defecto**



daemon=True


T2
(segundo
thread)
T1
(primer
thread)
Main
thread
Fin del
programa
T1.start()
T2.start()
Tiempo

daemon

🤔 ¿Cómo recordar la diferencia entre un thread daemon o no daemon?

1. Daemon es Demonio.
2. Estamos en la Universidad Católica.
3. Debemos **destruir a nuestros demonios**.

Por lo tanto, **un *thread* Daemon debe morir**… si el programa termina, exterminamos a todos los demonios.

daemon

Con el uso de ***daemon=True*** hacemos que su vida acabe cuando el programa principal termin. De este modo, esperamos 1 hora y todos los *threads* dejan de ejecutarse.

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

    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.seccion=seccion, commits)
    thread_commit = Commits(commit_s)
    thread_commit.start()

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

Recapitulación

- Con *Thread* podemos asegurar concurrencia .
- 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

Programación Avanzada
IIC2233 2024-2

Hernán Valdivieso - Daniela Concha - Francisca Ibarra - Lucas Van Sint Jan - Francisca Cattan

Comentarios AC4