ICC/Programmation — Exercices semaine 12

11 mai 2026

Objectif. Dans ces exercices, vous allez pratiquer les bases de la programmation multithread en Python, introduites lors du cours d'aujourd'hui : créer et démarrer des threads, identifier et corriger des race conditions à l'aide de locks, et raisonner sur les deadlocks. Vous découvrirez également les **`daemon`** threads — une variante utile introduite à travers une courte expérience pratique.

Ces exercices sont autonomes. Aucune bibliothèque supplémentaire n'est nécessaire au-delà de la bibliothèque standard Python.

Mise en place. Créez un nouveau fichier **`threads.py`** et travaillez chaque exercice en ajoutant du code à la fin du fichier. Exécutez-le à tout moment avec :

```
python3 threads.py
```

Exercice 1. Créer et démarrer des threads

Un **`Thread`** est créé en passant une fonction target à son constructeur. Appeler **`.start()`** lance le thread ; la ligne suivante de votre programme s'exécute immédiatement sans attendre que le thread se termine.

(a) **Bonjour depuis un thread.** Exécutez le code ci-dessous et assurez-vous de comprendre chaque ligne :

```python
from threading import Thread, current_thread
from time import sleep

def greet() -> None:
    name = current_thread().name
    for i in range(4):
        print(f"Hello from {name} (iteration {i})")
        sleep(0.5)

t1 = Thread(target=greet, name="Alpha")
t2 = Thread(target=greet, name="Beta")
t1.start()
t2.start()
print("Main thread: both threads started")
```

Est-ce que **`"Main thread: both threads started"`** est toujours la première ligne affichée ? Pourquoi ou pourquoi pas ?

(b) **Joindre des threads.** Appeler **`t.join()`** sur un thread bloque l'appelant jusqu'à ce que ce thread soit terminé — c'est ainsi que le thread principal peut attendre qu'un thread worker ait fini avant de continuer. Réécrivez la séquence de lancement ci-dessous pour que **`"Main thread: all done"`** soit garanti d'être la dernière ligne affichée. Ajoutez les appels **`join`** manquants aux bons endroits :

```python
t1 = Thread(target=greet, name="Alpha")
t2 = Thread(target=greet, name="Beta")
t1.start()
t2.start()
# appeler t1.join() et t2.join() ici
print("Main thread: all done")
```

(c) **Passer des arguments à un thread.** Complétez à la fois le corps de la fonction et la création des threads pour que chaque thread affiche un message de salutation le nombre de fois indiqué. Résultat attendu : **`"Hi, I am Alice"`** trois fois, intercalé avec **`"Hi, I am Bob"`** cinq fois.

```python
def greet_n(name: str, n: int) -> None:
    for _ in range(...):        # à compléter
        print(...)              # à compléter
        sleep(0.3)

t1 = Thread(target=greet_n, args=(..., ...))   # Alice, 3 fois
t2 = Thread(target=greet_n, args=(..., ...))   # Bob,   5 fois
t1.start(); t2.start()
t1.join();  t2.join()
```

(d) **Daemon threads.** Par défaut, le processus Python attend que tous les threads se terminent avant de quitter. Mettre **`daemon=True`** change ce comportement : le thread est automatiquement tué lorsque le thread principal se termine. Exécutez les deux versions ci-dessous et observez la différence de comportement :

```
def tick() -> None:
    for i in range(10):
        print(f"tick {i}")
        sleep(0.4)

# Version A -- thread normal : le processus attend la fin de tick()
t = Thread(target=tick)
t.start()
print("Main thread done")

# Version B -- daemon thread : tick() est interrompu quand le main quitte
t = Thread(target=tick, daemon=True)
t.start()
print("Main thread done")
```

Écrivez maintenant un daemon thread qui affiche **`"heartbeat"`** toutes les 0,2 s indéfiniment. Démarrez-le, puis faites dormir le thread principal pendant 1 s avant qu'il se termine. Combien de **`"heartbeat"`** sont affichés ? Que se passerait-il si le thread n'était pas un daemon ?

(e) **Un thread qui retourne une valeur.** Les threads ne peuvent pas retourner de valeurs directement. Une solution courante est de stocker le résultat dans une liste partagée. Complétez le corps de la fonction et le lancement des threads ci-dessous. **`total`** doit contenir la somme de tous les entiers de **`start`** (inclus) à **`stop`** (exclu) — utilisez **`sum(range(start, stop))`** pour le calculer. Chaque thread ajoute son **`total`** à **`results`** ; le thread principal affiche ensuite leur somme combinée :

```
results = []

def sum_range(start: int, stop: int) -> None:
    total = ...              # somme des entiers de start jusqu'a stop (exclu)
    results.append(total)   # stocker le resultat pour que le main puisse le lire

t1 = Thread(target=sum_range, args=(0, 500_000))
t2 = Thread(target=sum_range, args=(500_000, 1_000_000))
# demarrer les deux threads, puis les joindre

print(sum(results))  # attendu : 499999500000
```

Exercice 2. Race Conditions

Lorsque plusieurs threads lisent et écrivent une variable partagée sans coordination, le résultat peut dépendre de l'ordre exact dans lequel les instructions sont intercalées — c'est une race condition.

(a) **Observer la race condition.** Copiez le code du compteur ci-dessous dans **`threads.py`** et exécutez-le :

```
from threading import Thread

counter: int = 0

def increment(n: int) -> None:
    global counter
    for _ in range(n):
        counter += 1

N = 100_000
for _ in range(50):
    counter = 0
    t1 = Thread(target=increment, args=(N,))
    t2 = Thread(target=increment, args=(N,))
    t1.start(); t2.start()
    t1.join();  t2.join()
    print(f"Expected: {2 * N}, Got: {counter}")
```

Le résultat est-il toujours **`200 000`** ? Dessinez un court tableau d'interleaving (comme celui du cours) montrant comment la valeur finale peut être inférieure à ce qui est attendu.

Exercice 3. Corriger les race conditions avec des locks

Un **`Lock`** (aussi appelé mutex) garantit qu'un seul thread à la fois peut exécuter le code à l'intérieur d'un bloc **`with lock:`**. Tous les autres threads se bloquent jusqu'à ce que le lock soit libéré.

(a) **Compteur thread-safe.** Réécrivez le compteur de l'exercice 2 en utilisant un **`Lock`** pour que le résultat soit toujours **`200 000`**. Ajoutez le lock au bon endroit dans **`safe_increment`** :

```python
from threading import Thread, Lock

counter: int = 0
lock = Lock()

def safe_increment(n: int) -> None:
    global counter
    for _ in range(n):
        ...   # acquerir le lock et incrementer counter ici

N = 100_000
t1 = Thread(target=safe_increment, args=(N,))
t2 = Thread(target=safe_increment, args=(N,))
t1.start(); t2.start()
t1.join();  t2.join()
print(f"Expected: {2 * N}, Got: {counter}")  # doit etre 200 000
```

(b) **Compte bancaire thread-safe.** Complétez la classe **`BankAccount`** pour que les dépôts et retraits concurrents laissent toujours le solde correct. Le test en bas doit toujours afficher **`Final balance: 1000`** :

```python
from threading import Thread, Lock

class BankAccount:
    def __init__(self, initial: int) -> None:
        self.balance: int = initial
        self.lock = Lock()

    def deposit(self, amount: int) -> None:
        with self.lock:
            ...   # a completer

    def withdraw(self, amount: int) -> None:
        with self.lock:
            ...   # a completer

account = BankAccount(1000)

def do_deposits():
    for _ in range(1000):
        account.deposit(10)

def do_withdrawals():
    for _ in range(1000):
        account.withdraw(10)

t1 = Thread(target=do_deposits)
t2 = Thread(target=do_withdrawals)
t1.start(); t2.start()
t1.join();  t2.join()
print(f"Final balance: {account.balance}")  # doit etre 1000
```

(c) **Granularité du lock — mesurer le compromis.** Dans l'exercice 3(a), le lock est acquis une fois par itération (fine-grained). Implémentez une deuxième version **`safe_increment_coarse`** qui acquiert le lock une seule fois en dehors de la boucle, puis utilisez **`time.perf_counter`** pour mesurer et comparer le temps d'exécution des deux versions :

```python
from time import perf_counter

def safe_increment_coarse(n: int) -> None:
    global counter
    ...   # acquerir le lock une fois, puis boucler a l'interieur

def run(fn) -> float:
    """Reinitialise counter, lance fn dans deux threads, retourne le temps ecoule."""
    global counter
    counter = 0
    t1 = Thread(target=fn, args=(N,))
```

```
    t2 = Thread(target=fn, args=(N,))
    start = perf_counter()
    t1.start(); t2.start()
    t1.join();  t2.join()
    return perf_counter() - start

print(f"Fine-grained:   {run(safe_increment):.3f}s  result={counter}")
print(f"Coarse-grained: {run(safe_increment_coarse):.3f}s  result={counter}")
```

Quelle version est la plus rapide ? La version coarse-grained est-elle toujours correcte ici ?

Exercice 4. Tout assembler

Dans ce dernier exercice, vous allez construire une petite simulation concurrente : un système simplifié de réservation de billets où plusieurs clients tentent de réserver des places simultanément.

(a) **Mise en place.** Copiez le squelette ci-dessous dans **`threads.py`** :

```
from threading import Thread, Lock
from time import sleep
import random

class TicketOffice:
    def __init__(self, total_seats: int) -> None:
        self.seats: int = total_seats
        self.lock = Lock()

    def book(self, customer: str, n: int) -> None:
        """Tente de reserver n places pour customer."""
        ...  # a implementer en 4(b)

office = TicketOffice(total_seats=20)
```

(b) **Implémenter la réservation.** Complétez la méthode **`book`**. Elle doit acquérir le lock, vérifier s'il reste suffisamment de places, soustraire les places demandées si possible, et afficher soit :
`" {customer} booked {n} seat(s). Remaining: {self.seats}"`
soit :
`"× {customer} failed --- only {self.seats} left"`.

(c) **Lancer les threads clients.** Ajoutez la séquence de lancement suivante directement après la définition de **`office`** en 4(a). La list comprehension est déjà complète ; votre tâche est d'ajouter les boucles de démarrage et de jointure :

```
customers = [
    Thread(target=office.book, args=(f"Customer-{i}", random.randint(1, 4)))
    for i in range(10)
]

# demarrer tous les threads ici (une ligne)

# joindre tous les threads ici (une ligne)

print(f"\nSeats remaining: {office.seats}")
```

Lancez la simulation plusieurs fois. Confirmez que le nombre de places ne devient jamais négatif et qu'il est cohérent avec les messages affichés. Que se passerait-il si vous supprimiez le lock de **`book`** ?

(d) **Bonus — corriger le deadlock.** La fonction **`transfer`** ci-dessous contient un bug de deadlock. **Étape 1 :** copiez et exécutez la version buggée, et expliquez précisément dans quel cas le deadlock peut se produire (sous quel interleaving des deux threads). *Note : vous devrez peut-être l'exécuter plusieurs fois ; le deadlock n'est pas garanti de se produire à chaque exécution.*

```
# --- Version BUGGEE (peut se bloquer indefiniment) ---
def transfer(src: TicketOffice, dst: TicketOffice, n: int) -> None:
    with src.lock:
        sleep(0.01)        # simule du travail ; agrandit la fenetre de race
        with dst.lock:
            src.seats -= n
            dst.seats += n

office_a = TicketOffice(50)
```

```
office_b = TicketOffice(50)

t1 = Thread(target=transfer, args=(office_a, office_b, 5))
t2 = Thread(target=transfer, args=(office_b, office_a, 5))
t1.start(); t2.start()
t1.join(); t2.join()      # cette ligne peut ne jamais retourner
```

Étape 2 : implémentez **`transfer_fixed`** de façon à ce qu'elle soit à la fois correcte et sans deadlock. *Indice :* acquérez toujours les deux locks dans le même ordre, quel que soit le rôle de chaque office (**`src`** ou **`dst`**).

```
# --- VOTRE version corrigee ---
def transfer_fixed(src: TicketOffice, dst: TicketOffice, n: int) -> None:
    ...   # a completer

office_a = TicketOffice(50)
office_b = TicketOffice(50)

t1 = Thread(target=transfer_fixed, args=(office_a, office_b, 5))
t2 = Thread(target=transfer_fixed, args=(office_b, office_a, 5))
t1.start(); t2.start()
t1.join();  t2.join()
print(office_a.seats, office_b.seats)  # les deux doivent valoir 50
```