ICC/Programming — Week 12 Exercises

11 May 2026

Goal. In these exercises you will practise the fundamentals of multithreaded programming in Python introduced in today's lecture: creating and starting threads, identifying and fixing race conditions with locks, and reasoning about deadlock. Along the way you will also discover **`daemon`** threads — a useful variant introduced through a short hands-on experiment.

These exercises are self-contained. No additional libraries are required beyond the Python standard library.

Setup. Create a new file **`threads.py`** and work through each exercise by adding code to the bottom of the file. Run it at any point with:

```
python3 threads.py
```

Exercise 1. Creating and Starting Threads

A **`Thread`** is created by passing a target function to its constructor. Calling **`.start()`** launches the thread; the next line of your program executes immediately without waiting for the thread to finish.

(a) **Hello from a thread.** Run the following starter code and make sure you understand each line:

```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")
```

Is **`"Main thread: both threads started"`** always the first line printed? Why or why not?

(b) **Joining threads.** Calling **`t.join()`** on a thread blocks the caller until that thread has finished — it is how the main thread can wait for a worker to complete before moving on. Rewrite the launch sequence below so that **`"Main thread: all done"`** is guaranteed to be the last line printed. Add the missing **`join`** calls in the correct places:

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

(c) **Passing arguments to a thread.** Complete both the function body and the thread creation so that each thread prints a personalised greeting the given number of times. Expected output: "**`Hi, I am Alice`**" three times interleaved with "**`Hi, I am Bob`**" five times.

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

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

(d) **Daemon threads.** By default, the Python process waits for all threads to finish before exiting. Setting **`daemon=True`** changes this: the thread is silently killed when the main thread exits. Run the two versions below and observe the difference in behaviour:

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

# Version A -- normal thread: process waits for tick() to finish
t = Thread(target=tick)
t.start()
print("Main thread done")

# Version B -- daemon thread: tick() is cut short when main exits
t = Thread(target=tick, daemon=True)
t.start()
print("Main thread done")
```

Now write a daemon thread of your own that prints **`"heartbeat"`** every 0.2 s indefinitely. Start it, then make the main thread sleep for 1 s and exit. How many heartbeats are printed? What would happen if the thread were not a daemon?

(e) **A thread that returns a value.** Threads cannot return values directly. A common workaround is to store the result in a shared list. Complete the function body and the thread launch below. **`total`** should hold the sum of all integers from **`start`** (inclusive) to **`stop`** (exclusive) — use Python's built-in **`sum(range(start, stop))`** to compute it. Each thread appends its **`total`** to **`results`**; the main thread then prints their combined sum:

```python
results = []

def sum_range(start: int, stop: int) -> None:
    total = ...              # sum of integers from start up to (not including) stop
    results.append(total)   # store result so the main thread can read it

t1 = Thread(target=sum_range, args=(0, 500_000))
t2 = Thread(target=sum_range, args=(500_000, 1_000_000))
# start both threads, then join both threads

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

Exercise 2. Race Conditions

When multiple threads read and write a shared variable without coordination, the result can depend on the exact order in which instructions are interleaved — a race condition.

(a) **Observing the race.** Copy the counter code below into **`threads.py`** and run it several times:

```python
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}")
```

Is the result always **`200 000`**? Draw a short interleaving table (like the one from the lecture) showing how the final value can be less than expected.

Exercise 3. Fixing Race Conditions with Locks

A **`Lock`** (also called a mutex) ensures that only one thread at a time can execute the code inside a **`with lock:`** block. All other threads block until the lock is released.

(a) **Thread-safe counter.** Rewrite the counter from Exercise 2 using a **`Lock`** so that the result is always **`200 000`**. Add the lock in the correct place inside **`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):
        ...   # acquire the lock and increment counter here

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}")  # must be 200 000
```

(b) **Thread-safe bank account.** Complete the **`BankAccount`** class so that concurrent deposits and withdrawals always leave the balance correct. The test at the bottom should always print **`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:
            ...   # fill this in

    def withdraw(self, amount: int) -> None:
        with self.lock:
            ...   # fill this in

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}")  # must be 1000
```

(c) **Lock granularity — measure the trade-off.** In Exercise 3(a), the lock is acquired once per iteration (fine-grained). Implement a second version **`safe_increment_coarse`** that acquires the lock once outside the loop, then use **`time.perf_counter`** to measure and compare the runtime of both:

```python
from time import perf_counter

def safe_increment_coarse(n: int) -> None:
    global counter
    ...   # acquire lock once, then loop inside

def run(fn) -> float:
    """Reset counter, run fn in two threads, return elapsed seconds."""
    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}")
```

Which version is faster? Is the coarse-grained version still correct here?

Exercise 4. Putting It All Together

In this final exercise you will build a small concurrent simulation: a simplified ticket-booking system where multiple customers try to reserve seats at the same time.

(a) **Setup.** Copy the skeleton below into **`threads.py`**:

```python
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:
        """Try to reserve n seats for customer."""
        ...  # implement in 4(b)

office = TicketOffice(total_seats=20)
```

(b) **Implement booking.** Fill in the **`book`** method. It should acquire the lock, check whether enough seats remain, subtract the requested seats if possible, and print either
`" {customer} booked {n} seat(s). Remaining: {self.seats}"`
or
`"× {customer} failed --- only {self.seats} left"`.

(c) **Launch customer threads.** Add the following launch sequence directly after the **`office`** definition from 4(a). The list comprehension is already filled in; your task is to add the start and join loops:

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

# start all threads here (one line)

# join all threads here (one line)

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

Run the simulation several times. Confirm that the seat count never goes negative and is consistent with the printed booking messages. What would happen if you removed the lock from **`book`**?

(d) **Bonus — fix the deadlock.** The function **`transfer`** below contains a deadlock bug. **Step 1:** copy and run the buggy version and explain exactly when the deadlock can occur (under what interleaving of the two threads). *Note: you may need to run it a few times; the deadlock is not guaranteed to trigger every run.*

```python
# --- BUGGY version (may hang) ---
def transfer(src: TicketOffice, dst: TicketOffice, n: int) -> None:
    with src.lock:
        sleep(0.01)        # simulate work; makes the race window wider
        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()      # this line may never return
```

Step 2: implement **`transfer_fixed`** so that it is both correct and deadlock-free. *Hint:* always acquire the two locks in the same order, regardless of which office is **`src`** and which is **`dst`**.

```python
# --- YOUR fixed version ---
def transfer_fixed(src: TicketOffice, dst: TicketOffice, n: int) -> None:
    ...   # fill this in

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)  # must both be 50
```