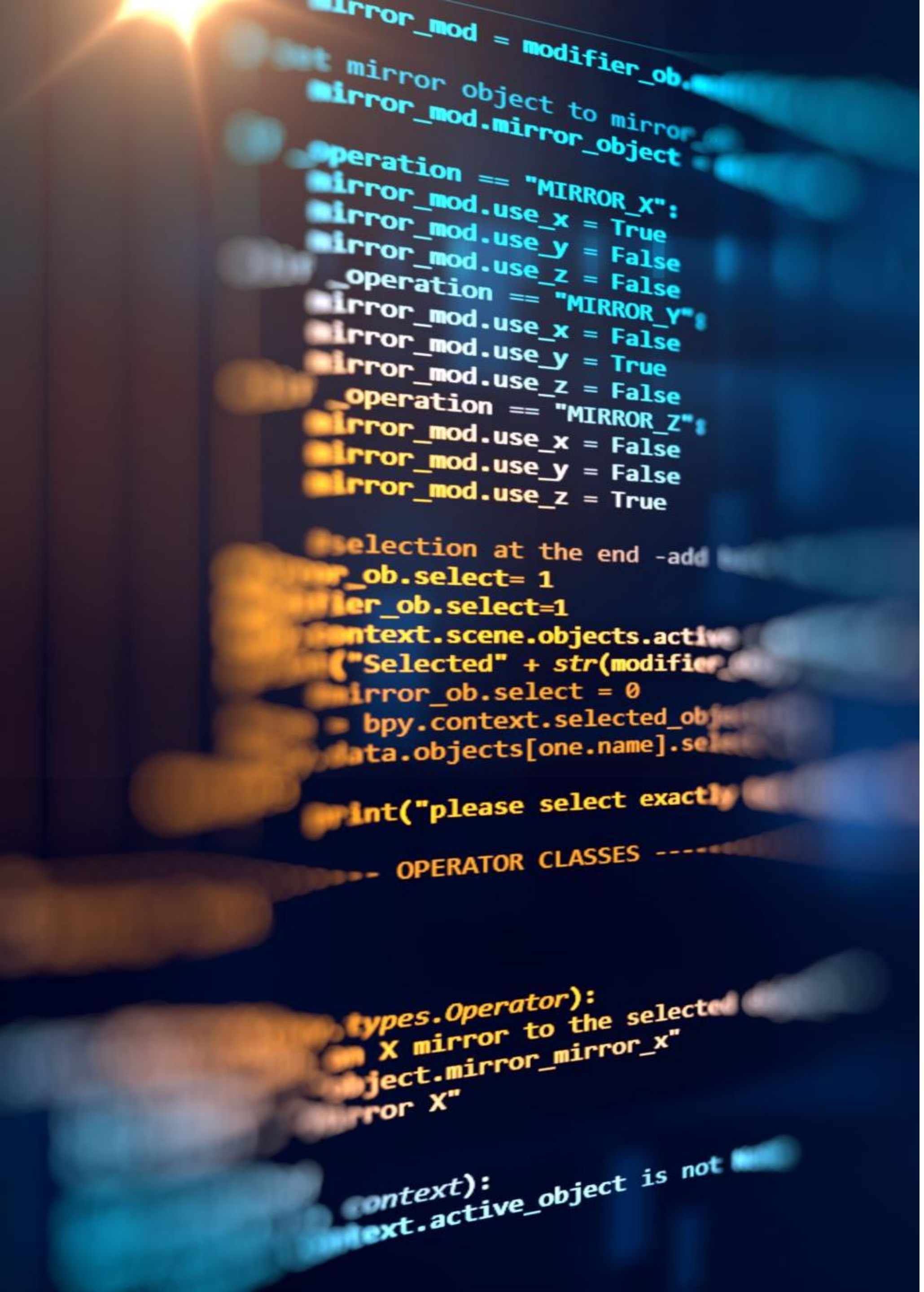
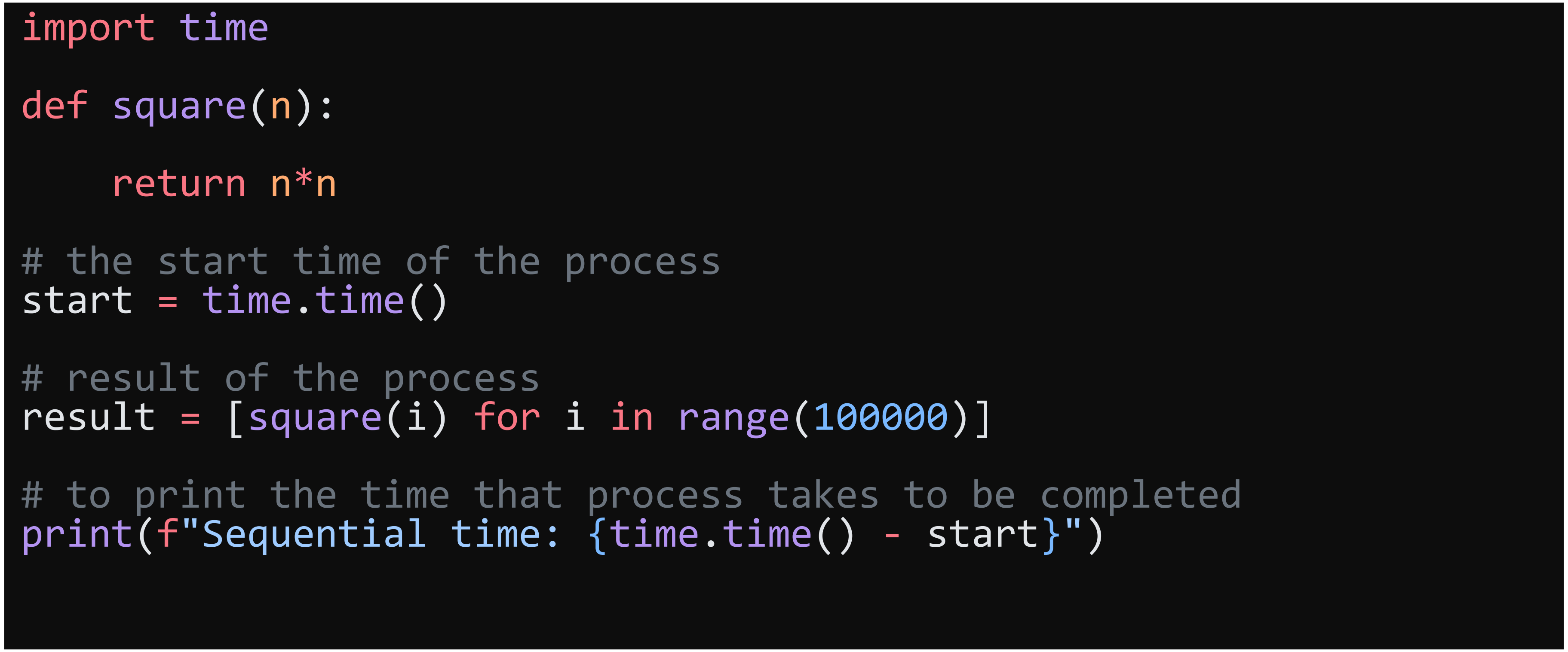


PARALLEL PROGRAMMING

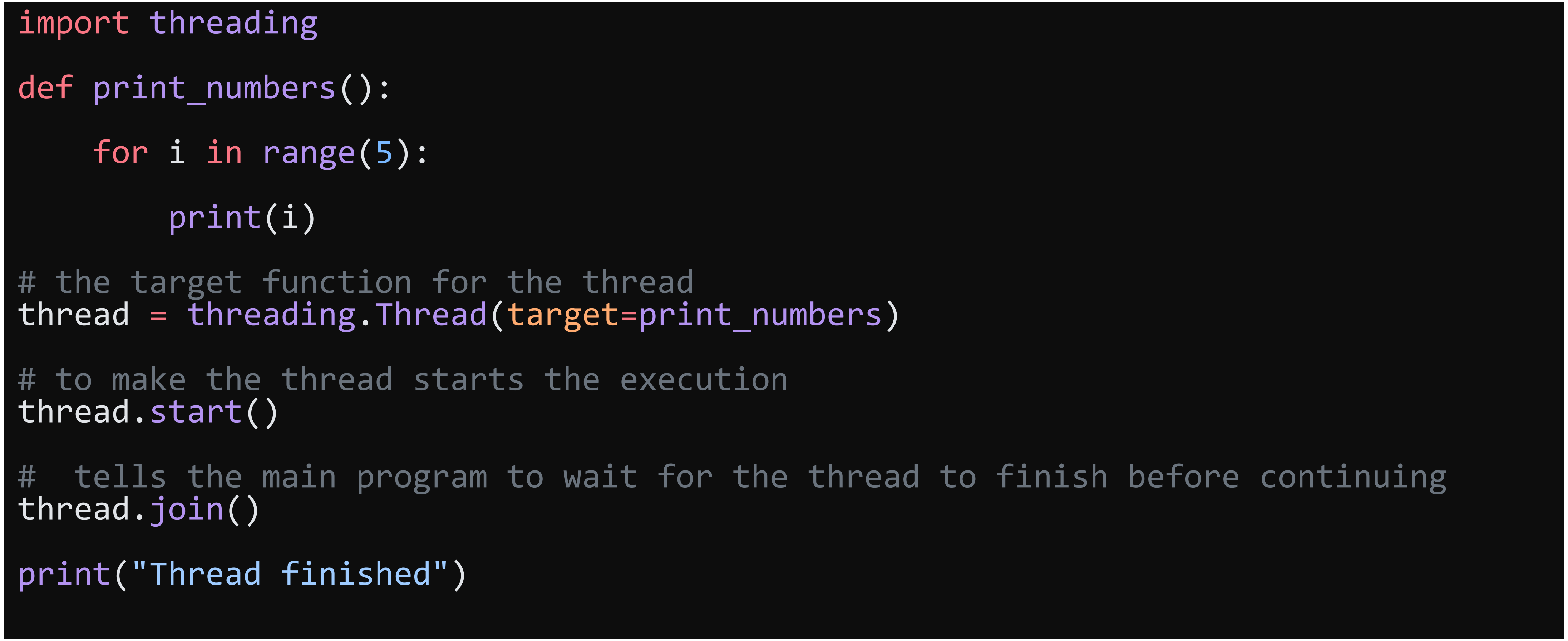
CHAPTER 1

SEQUENTIAL CODE



```python
import time

def square(n):

    return n*n

# the start time of the process
start = time.time()

# result of the process
result = [square(i) for i in range(100000)]

# to print the time that process takes to be completed
print(f"Sequential time: {time.time() - start}")
```

THREADING CODE

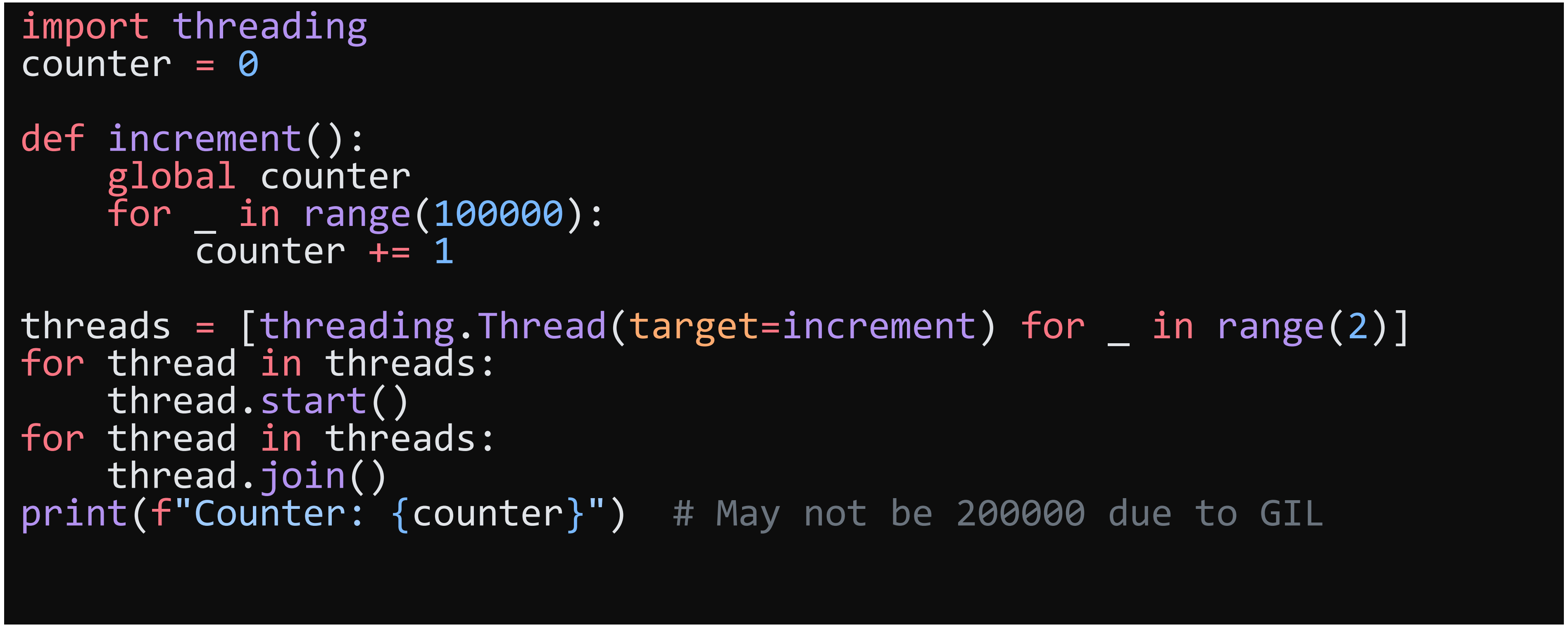


```python
import threading

def print_numbers():

    for i in range(5):

        print(i)

# the target function for the thread
thread = threading.Thread(target=print_numbers)

# to make the thread starts the execution
thread.start()

#  tells the main program to wait for the thread to finish before continuing
thread.join()

print("Thread finished")
```

MULTI-PROCESSING CODE

```
import multiprocessing

def print_numbers():

    for i in range(5):

        print(i)

process = multiprocessing.Process(target=print_numbers)

process.start()

process.join()
```

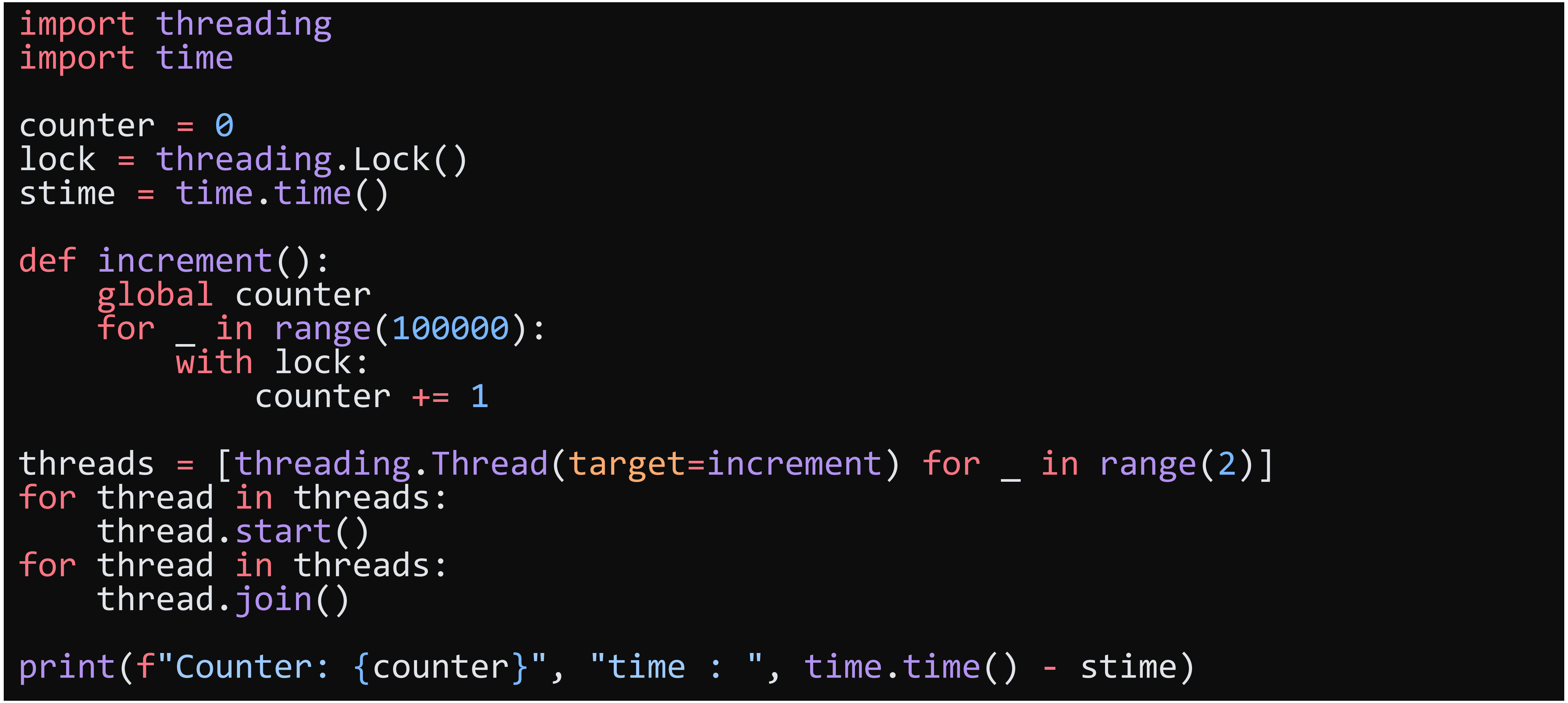
GIL DEMONSTRATION CODE



```python
import threading
counter = 0

def increment():
    global counter
    for _ in range(100000):
        counter += 1

threads = [threading.Thread(target=increment) for _ in range(2)]
for thread in threads:
    thread.start()
for thread in threads:
    thread.join()
print(f"Counter: {counter}")  # May not be 200000 due to GIL
```

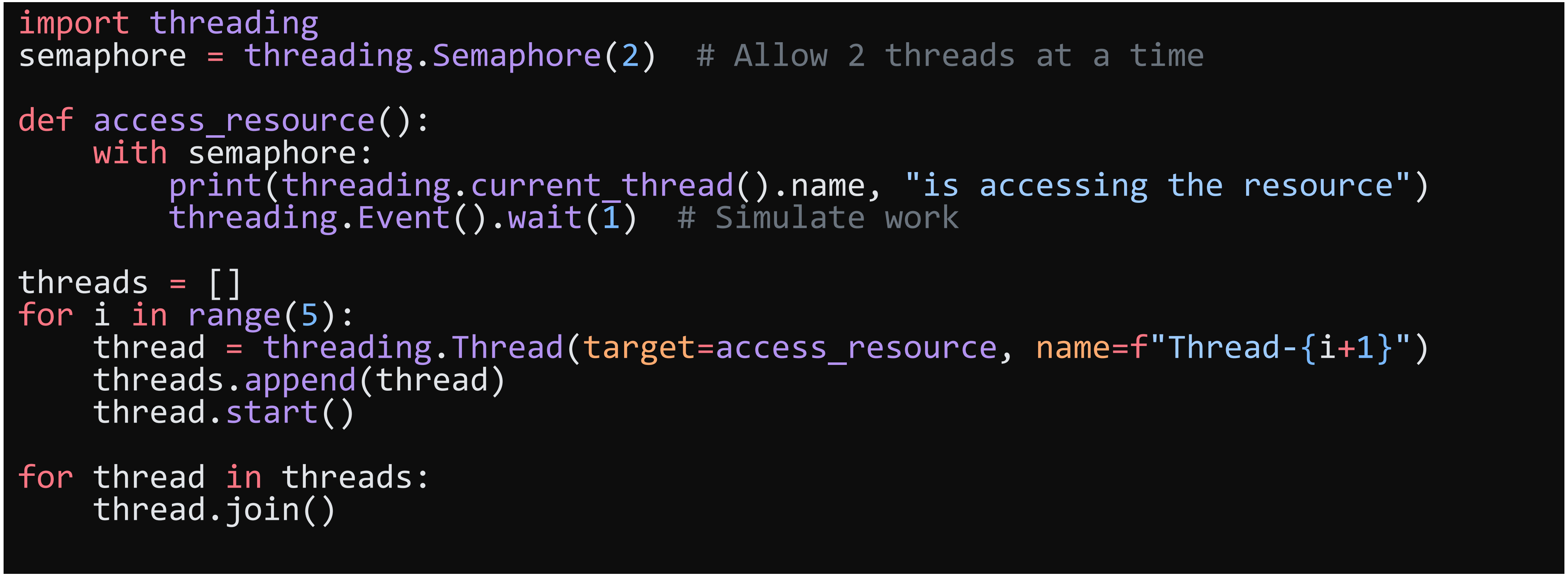
CHAPTER 2

THREAD SYNCHRONIZATION WITH LOCKS



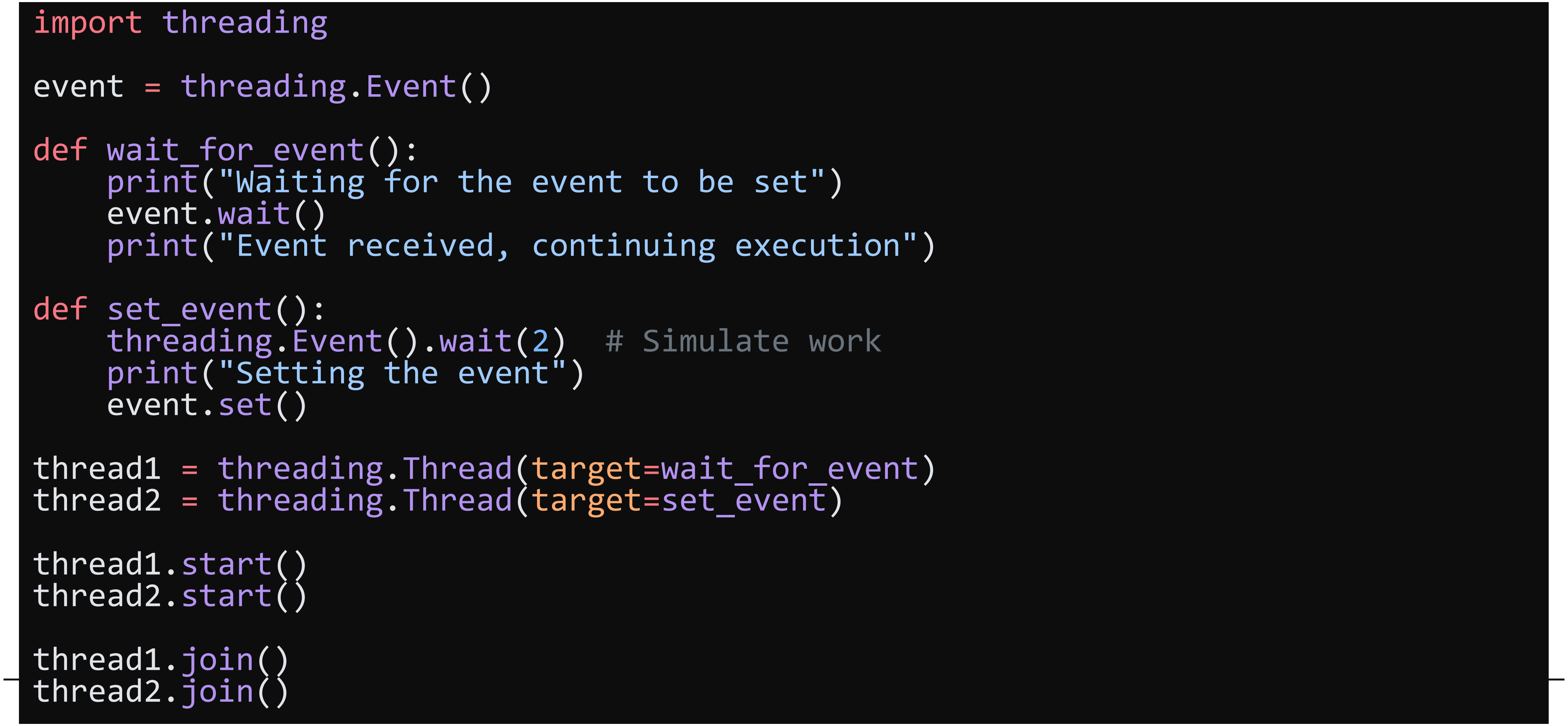
```python
import threading
import time

counter = 0
lock = threading.Lock()
stime = time.time()

def increment():
    global counter
    for _ in range(100000):
        with lock:
            counter += 1

threads = [threading.Thread(target=increment) for _ in range(2)]
for thread in threads:
    thread.start()
for thread in threads:
    thread.join()

print(f"Counter: {counter}", "time : ", time.time() - stime)
```

SEMAPHORE



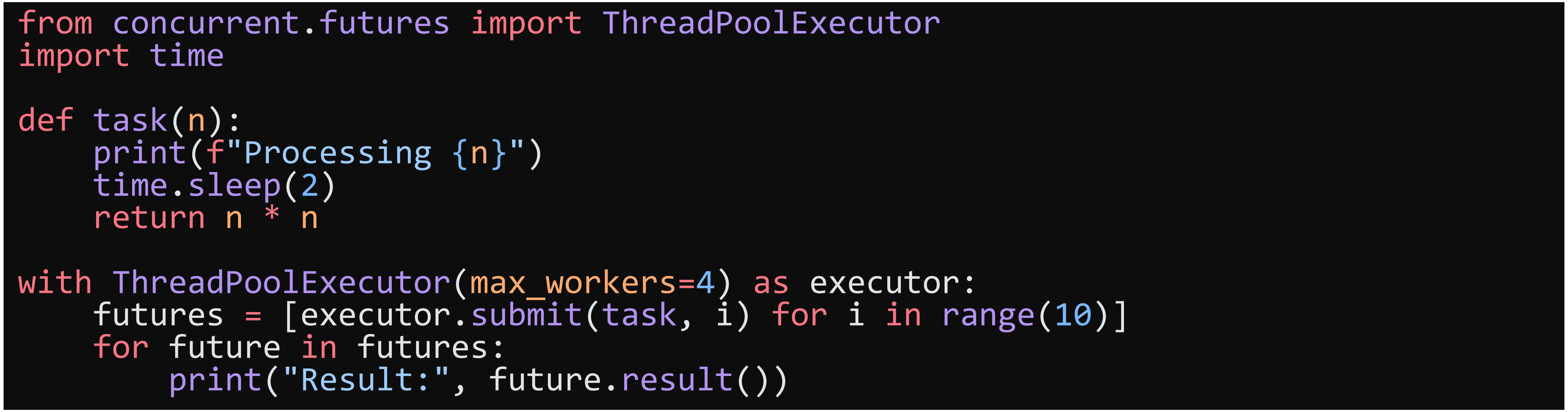
```python
import threading
semaphore = threading.Semaphore(2)  # Allow 2 threads at a time

def access_resource():
    with semaphore:
        print(threading.current_thread().name, "is accessing the resource")
        threading.Event().wait(1)  # Simulate work

threads = []
for i in range(5):
    thread = threading.Thread(target=access_resource, name=f"Thread-{i+1}")
    threads.append(thread)
    thread.start()

for thread in threads:
    thread.join()
```

THREAD COMMUNICATION WITH EVENTS



```python
import threading

event = threading.Event()

def wait_for_event():
    print("Waiting for the event to be set")
    event.wait()
    print("Event received, continuing execution")

def set_event():
    threading.Event().wait(2)  # Simulate work
    print("Setting the event")
    event.set()

thread1 = threading.Thread(target=wait_for_event)
thread2 = threading.Thread(target=set_event)

thread1.start()
thread2.start()

thread1.join()
thread2.join()
```

THREAD POOLS WITH THREADPOOLEXECUTOR



```python
from concurrent.futures import ThreadPoolExecutor
import time

def task(n):
    print(f"Processing {n}")
    time.sleep(2)
    return n * n

with ThreadPoolExecutor(max_workers=4) as executor:
    futures = [executor.submit(task, i) for i in range(10)]
    for future in futures:
        print("Result:", future.result())
```

CHAPTER 3

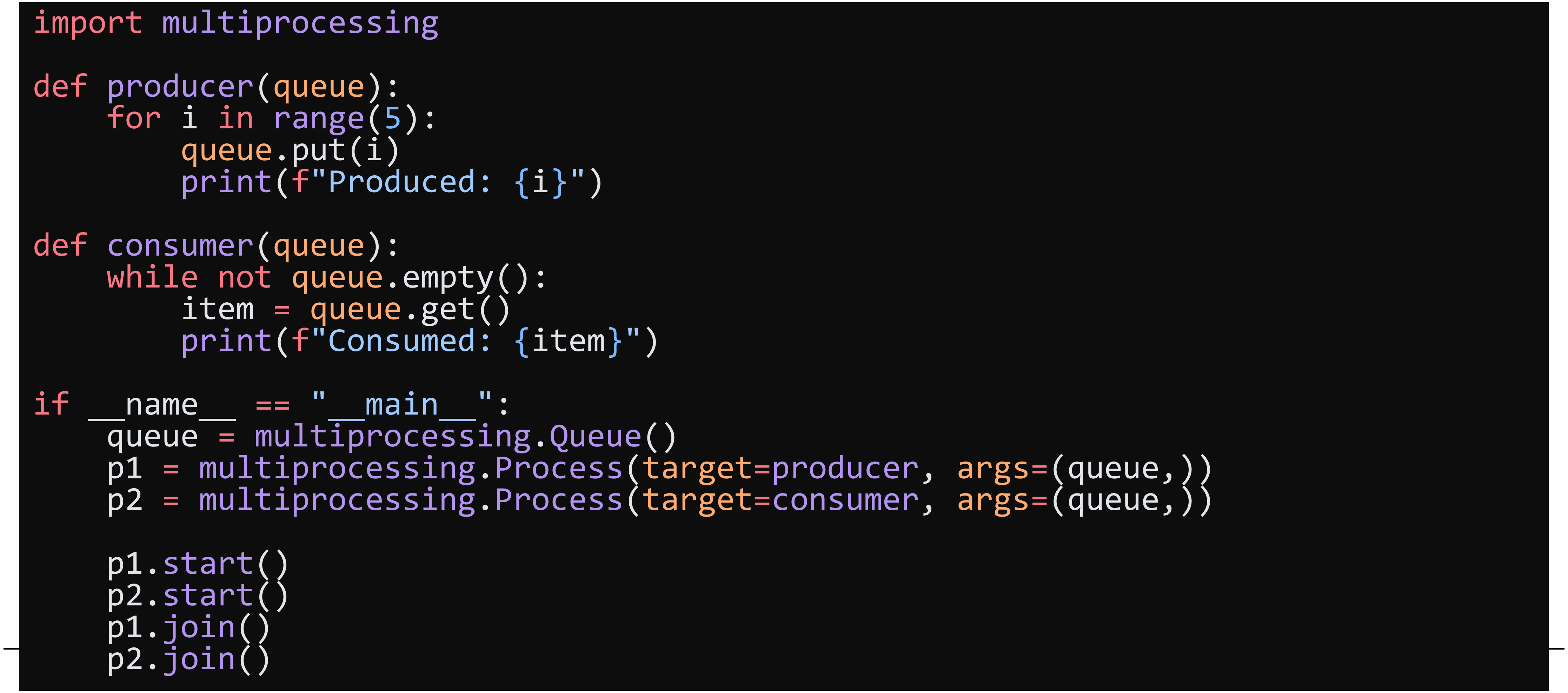
NUM OF CPUS

```
import multiprocessing

print("Number of cpu : ", multiprocessing.cpu_count())
```

PARALLEL SQUARE FUNCTION USING POOL

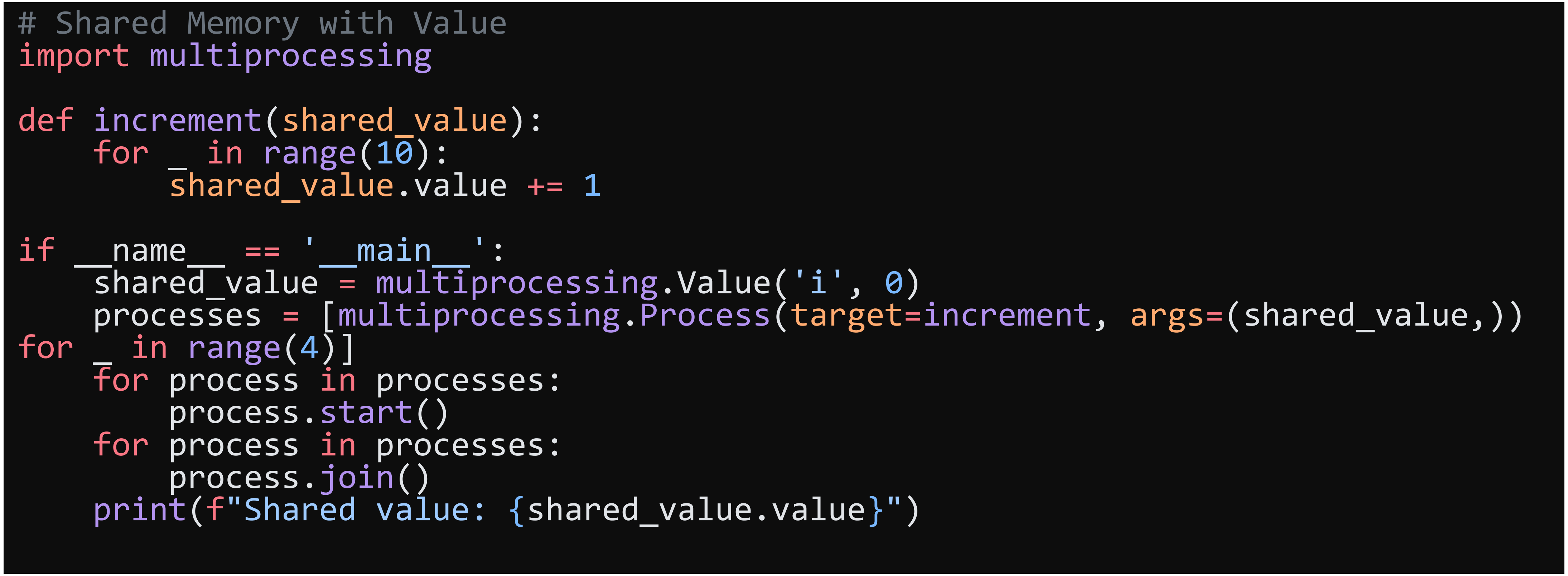
```
import multiprocessing

def square(n):
    return n * n

if __name__ == "__main__":
    with multiprocessing.Pool(4) as pool:
        results = pool.map(square, range(10))
    print(results)
```

QUEUE FOR INTER-PROCESS COMMUNICATION



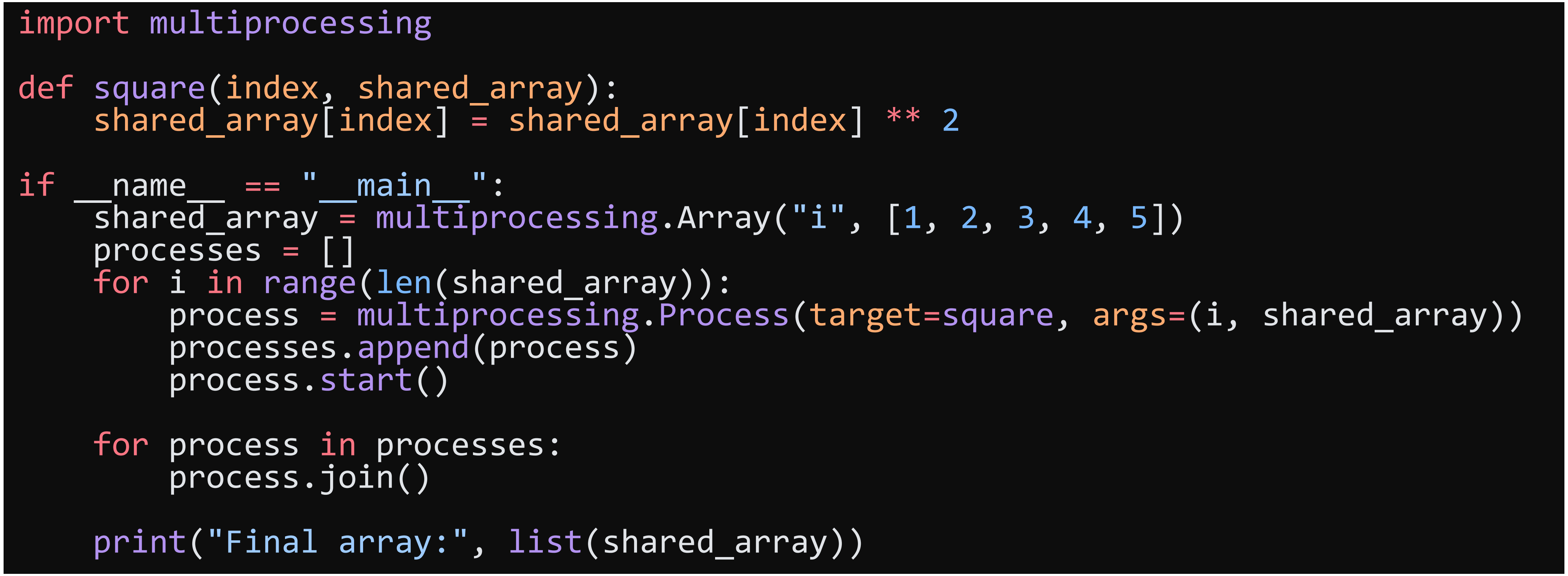
```python
import multiprocessing

def producer(queue):
    for i in range(5):
        queue.put(i)
        print(f"Produced: {i}")

def consumer(queue):
    while not queue.empty():
        item = queue.get()
        print(f"Consumed: {item}")

if __name__ == "__main__":
    queue = multiprocessing.Queue()
    p1 = multiprocessing.Process(target=producer, args=(queue,))
    p2 = multiprocessing.Process(target=consumer, args=(queue,))

    p1.start()
    p2.start()
    p1.join()
    p2.join()
```

SHARED MEMORY WITH VALUE



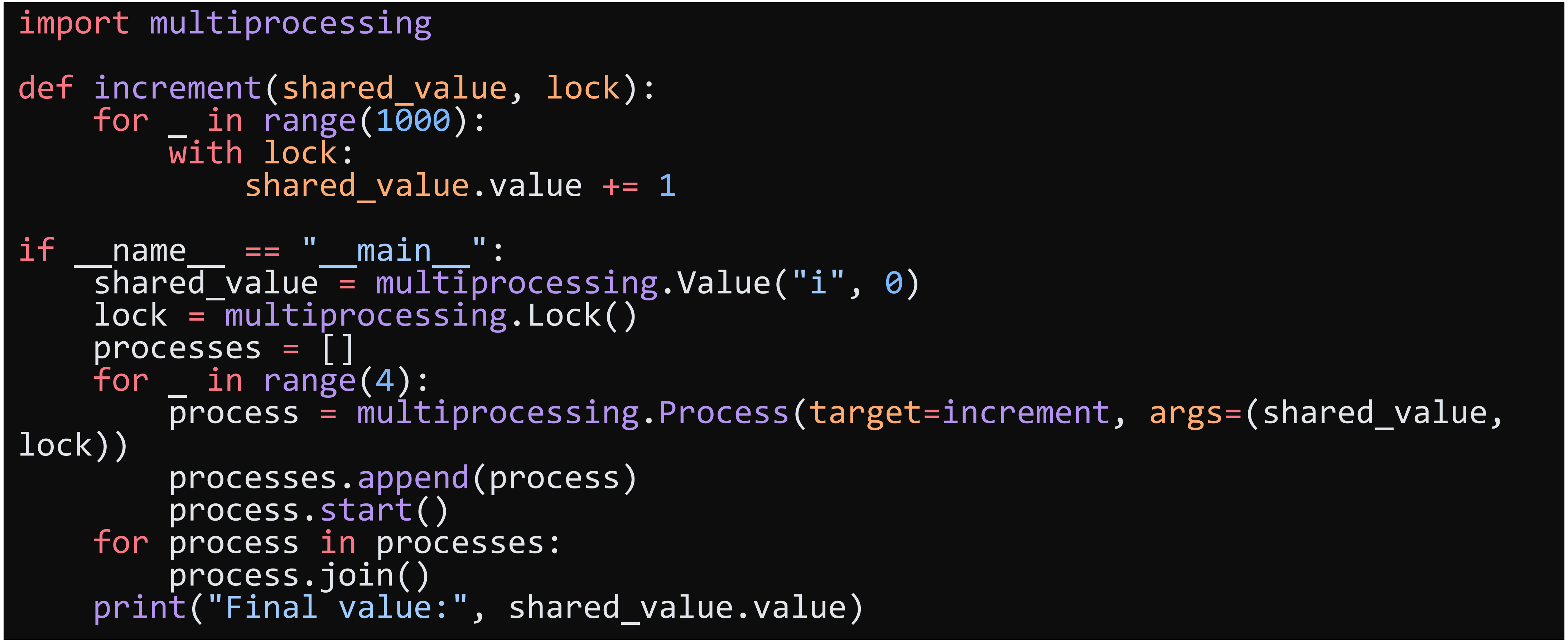
```
# Shared Memory with Value
import multiprocessing

def increment(shared_value):
    for _ in range(10):
        shared_value.value += 1

if __name__ == '__main__':
    shared_value = multiprocessing.Value('i', 0)
    processes = [multiprocessing.Process(target=increment, args=(shared_value,))
for _ in range(4)]
    for process in processes:
        process.start()
    for process in processes:
        process.join()
    print(f"Shared value: {shared_value.value}")
```

SHARED MEMORY WITH ARRAY



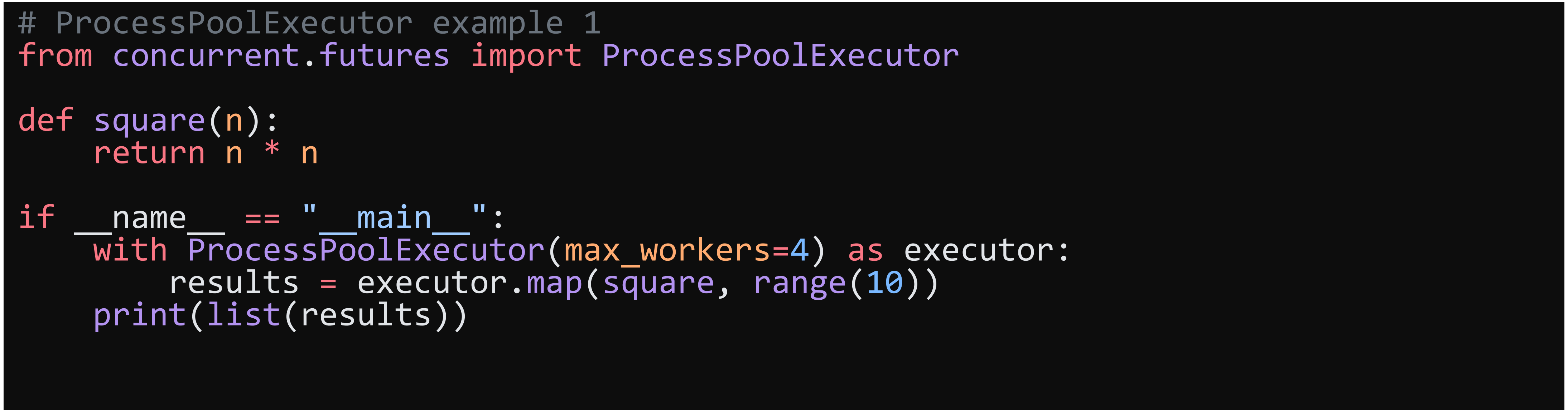
```
import multiprocessing

def square(index, shared_array):
    shared_array[index] = shared_array[index] ** 2

if __name__ == "__main__":
    shared_array = multiprocessing.Array("i", [1, 2, 3, 4, 5])
    processes = []
    for i in range(len(shared_array)):
        process = multiprocessing.Process(target=square, args=(i, shared_array))
        processes.append(process)
        process.start()

    for process in processes:
        process.join()

    print("Final array:", list(shared_array))
```

SYNCHRONIZATION WITH LOCK



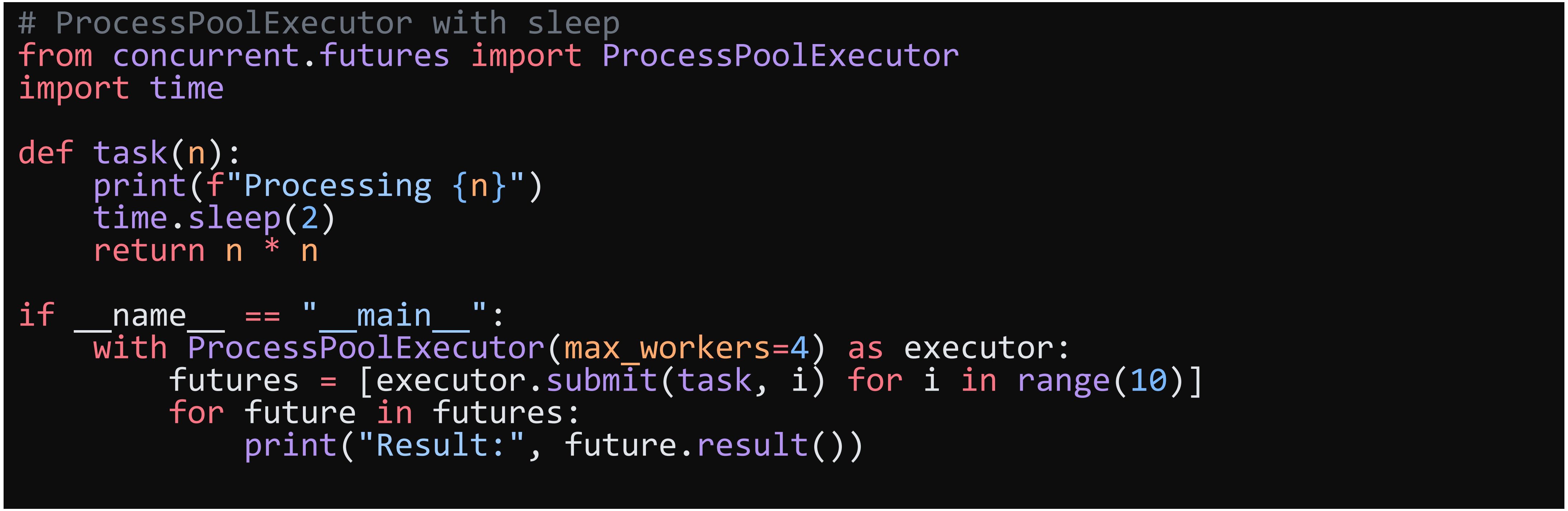
```python
import multiprocessing

def increment(shared_value, lock):
    for _ in range(1000):
        with lock:
            shared_value.value += 1

if __name__ == "__main__":
    shared_value = multiprocessing.Value("i", 0)
    lock = multiprocessing.Lock()
    processes = []
    for _ in range(4):
        process = multiprocessing.Process(target=increment, args=(shared_value,
lock))
        processes.append(process)
        process.start()
    for process in processes:
        process.join()
    print("Final value:", shared_value.value)
```

PROCESS POOL EXECUTOR



```
# ProcessPoolExecutor example 1
from concurrent.futures import ProcessPoolExecutor

def square(n):
    return n * n

if __name__ == "__main__":
    with ProcessPoolExecutor(max_workers=4) as executor:
        results = executor.map(square, range(10))
    print(list(results))
```

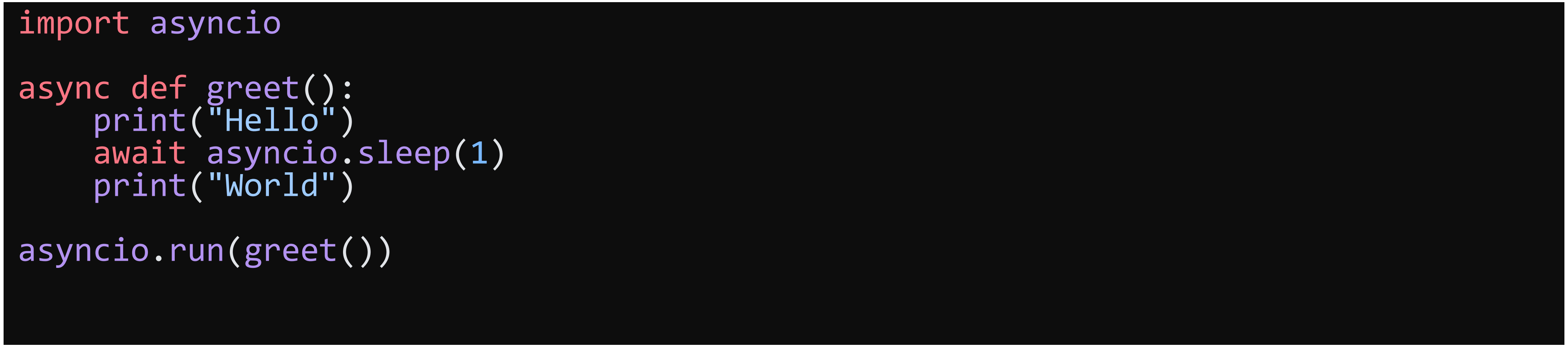
PROCESS POOL EXECUTOR WITH SLEEP



```python
# ProcessPoolExecutor with sleep
from concurrent.futures import ProcessPoolExecutor
import time

def task(n):
    print(f"Processing {n}")
    time.sleep(2)
    return n * n

if __name__ == "__main__":
    with ProcessPoolExecutor(max_workers=4) as executor:
        futures = [executor.submit(task, i) for i in range(10)]
        for future in futures:
            print("Result:", future.result())
```

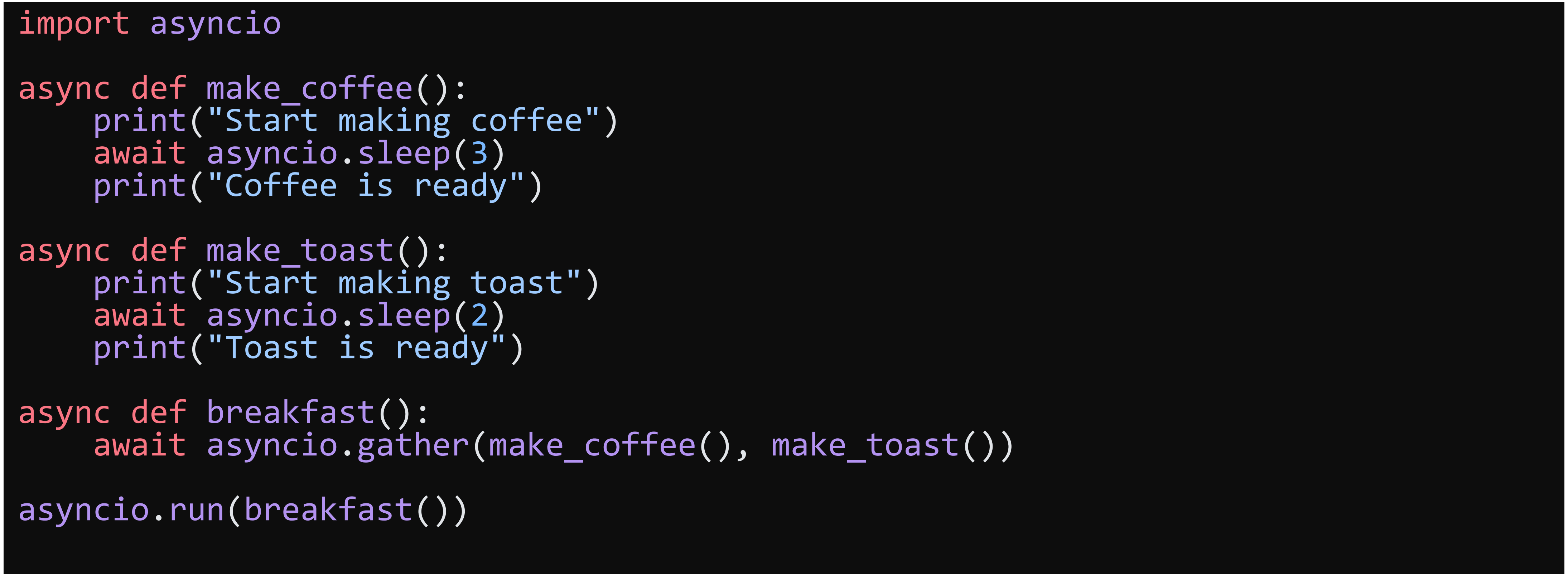
CHAPTER 4

ASYNCIO BASIC USAGE



```python
import asyncio

async def greet():
    print("Hello")
    await asyncio.sleep(1)
    print("World")

asyncio.run(greet())
```

RUNNING MULTIPLE COROUTINES

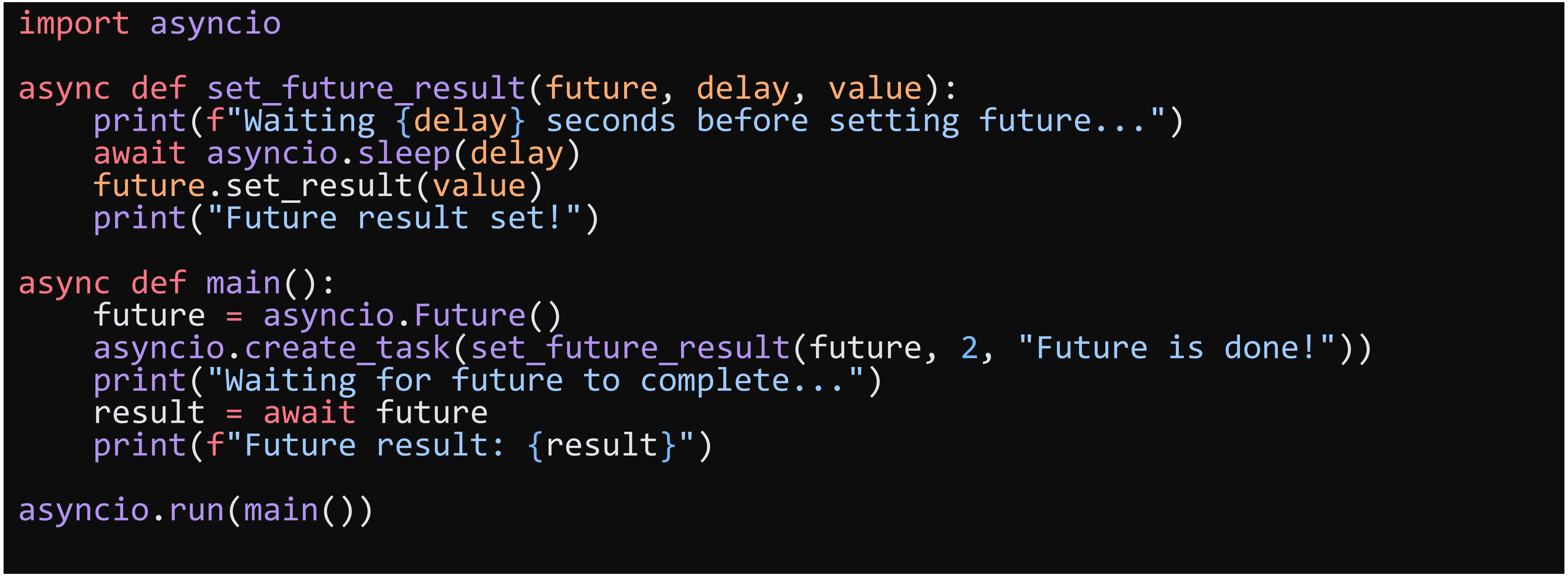


```python
import asyncio

async def make_coffee():
    print("Start making coffee")
    await asyncio.sleep(3)
    print("Coffee is ready")

async def make_toast():
    print("Start making toast")
    await asyncio.sleep(2)
    print("Toast is ready")

async def breakfast():
    await asyncio.gather(make_coffee(), make_toast())

asyncio.run(breakfast())
```

CREATING TASKS

```python
import asyncio

async def count_down(name, seconds):
    for i in range(seconds, 0, -1):
        print(f"{name}: {i}")
        await asyncio.sleep(1)
    print(f"{name}: Done!")

async def main():
    task1 = asyncio.create_task(count_down("Task A", 3))
    task2 = asyncio.create_task(count_down("Task B", 5))
    await task1
    await task2

asyncio.run(main())
```

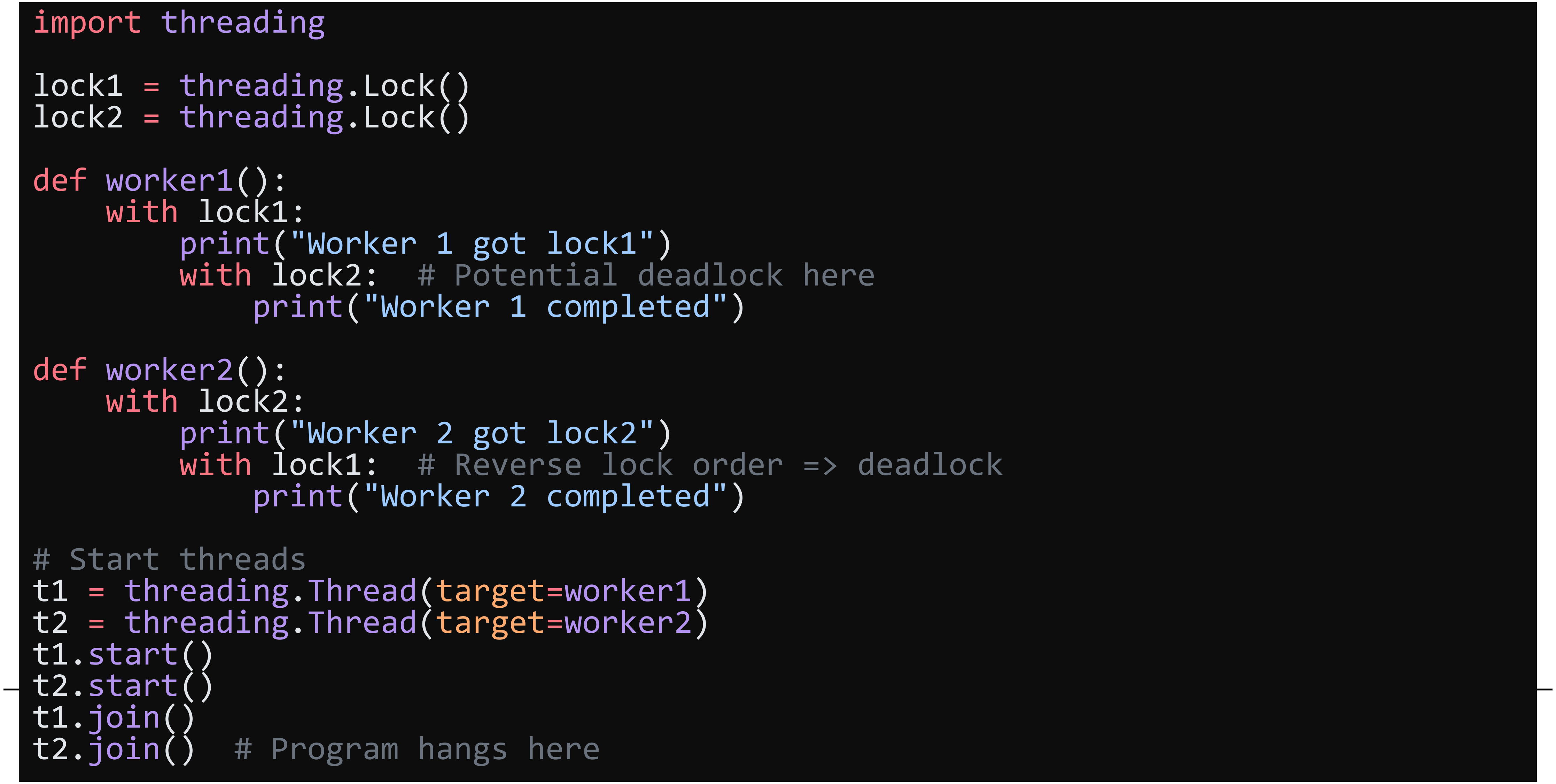
USING ASYNCIO FUTURE



```python
import asyncio

async def set_future_result(future, delay, value):
    print(f"Waiting {delay} seconds before setting future...")
    await asyncio.sleep(delay)
    future.set_result(value)
    print("Future result set!")

async def main():
    future = asyncio.Future()
    asyncio.create_task(set_future_result(future, 2, "Future is done!"))
    print("Waiting for future to complete...")
    result = await future
    print(f"Future result: {result}")

asyncio.run(main())
```

CHAPTER 5

DEADLOCK



```python
import threading

lock1 = threading.Lock()
lock2 = threading.Lock()

def worker1():
    with lock1:
        print("Worker 1 got lock1")
        with lock2:  # Potential deadlock here
            print("Worker 1 completed")

def worker2():
    with lock2:
        print("Worker 2 got lock2")
        with lock1:  # Reverse lock order => deadlock
            print("Worker 2 completed")

# Start threads
t1 = threading.Thread(target=worker1)
t2 = threading.Thread(target=worker2)
t1.start()
t2.start()
t1.join()
t2.join()  # Program hangs here
```

PROFILING

```python
import cProfile

def square(n):
    return n * n

cProfile.run('[square(i) for i in range(1000000)]')
```

سُبْحَانَكَ اللَّهُمَّ رَبَّنَا وَبِحَمْدِكَ، نَشْهَدُ أَنْ لَا إِلَٰهَ إِلَّا أَنْتَ، نَسْتَغْفِرُكَ وَنَتُوبُ إِلَيْكَ.

لا تنسونا من صالح الدعاء

THANKS