CSCI 471

Parallel and Distributed Programming

Spring 2025

Asynchronous Programming

Instructor:

Dr. Muhammad Haris Rais

Office Hours:

Mon, Tues, Wed: 10:00 AM – 12:00 PM

By appointment: Thurs: 10:00 AM – 12:00 PM

Email: mrais@vsu.edu

Office Phone: 804-524-5415

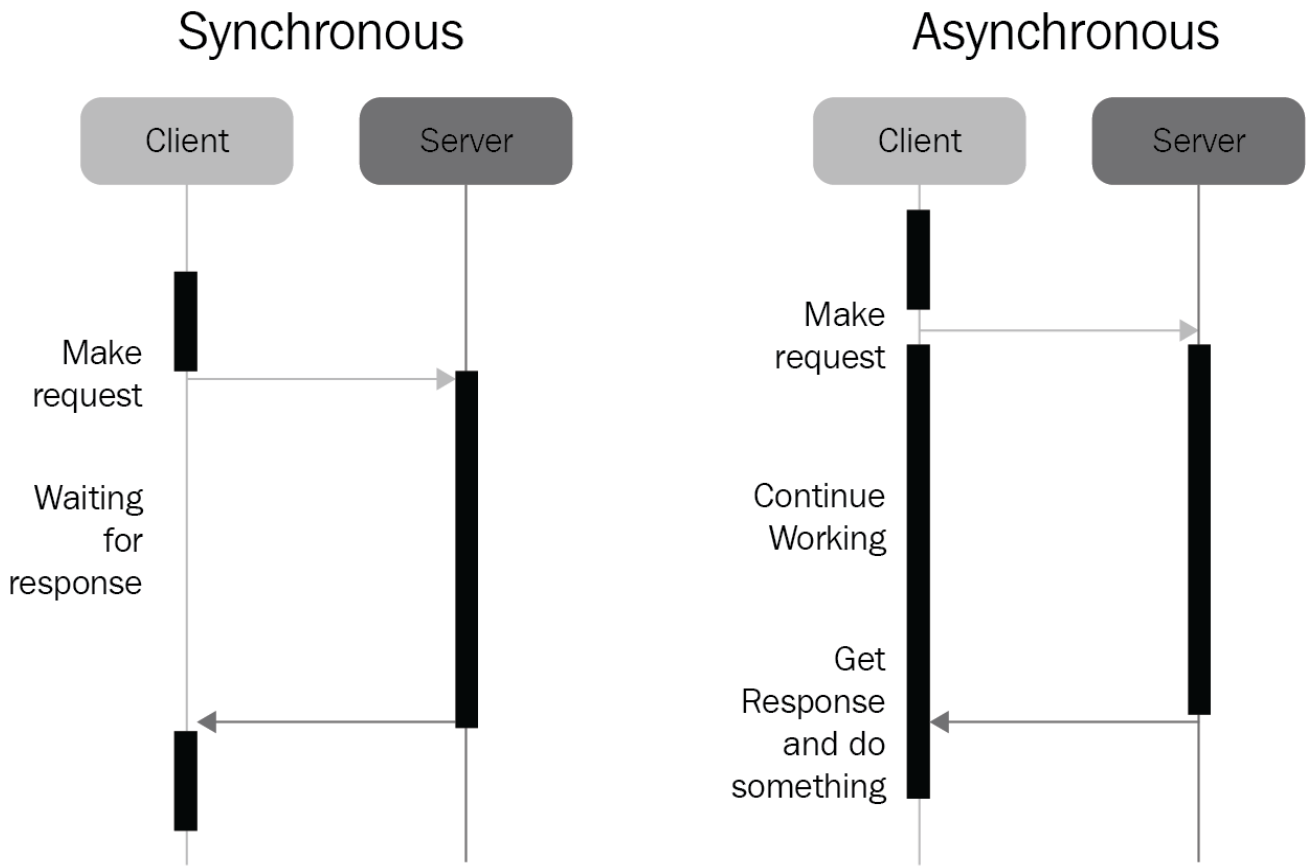
Synchronous Programming Problem

- Synchronous programming is the sequential execution of code
- Once a function/method is called
  - The flow of the program shifts to the called method
  - When all the instructions in the called method are done, the flow is returned to the calling method
- What if the called method has to wait?
  - Sleep
  - Network-bound activities
  - User input
- The program will WAIT and wont do anything else!!!
- Is it efficient?

An analogy from kitchen

- you are cooking a three-course meal that contains the following:
  - An appetizer that will take 2 minutes of preparation and 3 minutes of cooking/waiting
  - A main course that will take 5 minutes of preparation and 10 minutes of cooking/waiting
  - A dessert that will take 3 minutes of preparation and 5 minutes of cooking/waiting
- In a sequential way, the items will be ready in
- What if we overlap preparation and cooking?
- You should not wait while an item is being cooked; prepare appetizer, put it for cooking, and start preparing next; this will take
- What if you start with preparing the main course first?
- Finding the best order to execute and switch between tasks in a program is the main idea behind asynchronous programming

Asynchronous versus Synchronous programming – HTTP request example



Asynchronous versus Threading and Multiprocessing

- In multiprocessing, multiple copies of the program, together with its instructions and variables, are created and executed independently across different cores
- In multithreading, independent portions of the code that are executed in separate threads (multiple executors) do not interact with one another either.
- Asynchronous programming, on the other hand, keeps all of the instructions of a program in the same thread and process.
- The main idea behind asynchronous programming is to have a single executor to switch from one task to another if it is more efficient

Implementing Asynchronous Programming in Python

Coroutines, event loops, and futures

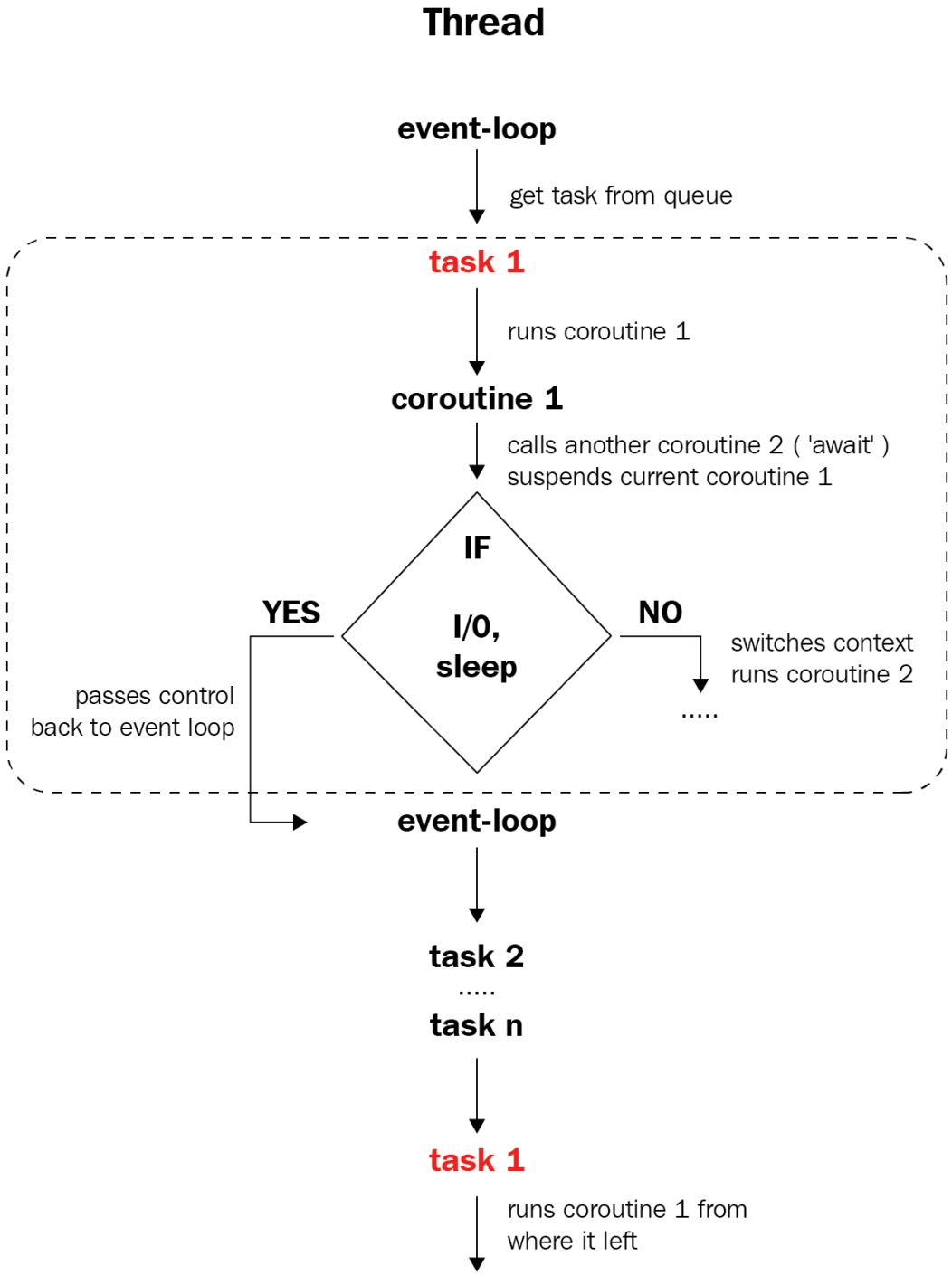
- **Event loops** are the main coordinators of tasks in an asynchronous program.
- An event loop keeps track of all of the tasks that are to be run asynchronously, and decides which of those tasks should be executed at a given moment.
- Event loops handle the task-switching aspect (or the execution flow) of asynchronous programming.
- **Coroutines** are a special type of function that wrap around specific tasks so that they can be executed asynchronously.
- A coroutine is required to specify where in the function the task switching should take place
- The tasks for coroutines are typically either stored in a task queue or created inside the event loop.

Coroutines, event loops, and futures

- **Futures** are placeholders for the results returned from coroutines.
- These future objects are created as soon as coroutines are initiated in the event loop so that futures can represent actual results, or pending results (if the coroutine is not finished)

A sample flow diagram

- First, an event loop is started
- When an async function (coroutine) is called
  - A task is created
  - Added to the event loop
- Event loop picks a task from the task queue
- Coroutine in the task and the corresponding future are created
- When task switching takes place inside the coroutine, that coroutine suspends and the next one is called
- Data and context of the suspended coroutine is saved
- If a coroutine blocks (e.g. Sleeping / I/O operation), the flow is released back to the event loop, which calls the next item in the queue
- If a task is finished, it is de-queued from the task queue



Asyncio API

- The foundation for this API is the async and await keywords added to Python 3.5.
- async is put before the def keyword when a function is declared.
- Python interprets that function as a coroutine
- await keyword is then used to specify where and when, exactly, to give back the flow of execution to the event loop
- Both async and await are provided by Python and not the asyncio API
- We can implement asynchronous programming without asyncio API
- asyncio provides a framework and streamlines the process

Important methods in Asyncio

- asyncio.get_event_loop(): This method returns the event loop for the current context, which is an AbstractEventLoop object. Most of the time, we do not need to worry about this class, as the asyncio module already provides a high-level API to manage our event loops.
- AbstractEventLoop.create_task(): This method is to be called by an event loop. It adds its input to the current task queue of the calling event loop; the input is typically a coroutine (that is, a function with the async keyword).

Important methods in Asyncio

- AbstractEventLoop.run_until_complete(): This method is also to be called by an event loop. It takes in the main coroutine of an asynchronous program and executes it until the corresponding future of the coroutine is returned. While the method initiates the event loop execution, it also blocks all subsequent code following it, until all futures are complete.
- AbstractEventLoop.run_forever(): This method is somewhat similar to AbstractEventLoop.run_until_complete(), except for the fact that, as suggested by the method name, the calling event loop will run forever, unless the AbstractEventLoop.stop() method is called. So, instead of exiting, the loop will continue to run, even upon obtaining the returned futures.
- AbstractEventLoop.stop(): This method causes the calling event loop to stop executing and exit at the nearest appropriate opportunity without causing the whole program to crash.

Important methods in Asyncio

- asyncio.sleep(): While in itself a coroutine, this function creates an additional coroutine that completes after a given time (specified by the input, in seconds). It is typically used as asyncio.sleep(0), to cause an immediate task-switching event.
- asyncio.wait(): This function is also a coroutine, and hence, it can be used to switch tasks. It takes in a sequence (usually a list) of futures and waits for them to complete their execution.

Demonstrating Asynchrony

- Sequential Version

```
import time
def count_down(name, delay):
    indents = (ord(name) - ord('A')) * '\t'
    n = 3
    while n:
        time.sleep(delay)
        duration = time.perf_counter() - start
        print('-' * 40)
        print('%.4f \t%s%s = %i' % (duration, indents, name, n))
        n -= 1
start = time.perf_counter()

count_down('A', 1)
count_down('B', 0.8)
count_down('C', 0.5)
print('-' * 40)
print('Done.')
```

```
----------------------------------------
1.0008  A = 3
----------------------------------------
2.0011  A = 2
----------------------------------------
3.0020  A = 1
----------------------------------------
3.8029      B = 3
----------------------------------------
4.6033      B = 2
----------------------------------------
5.4035      B = 1
----------------------------------------
5.9047          C = 3
----------------------------------------
6.4051          C = 2
----------------------------------------
6.9054          C = 1
----------------------------------------
Done.
```

Demonstrating Asynchrony-Asynchronous Version

```
import time
import asyncio
async def count_down(name, delay):
    indents = (ord(name) - ord('A')) * '\t'
    n = 3
    while n:
        await asyncio.sleep(delay)
        duration = time.perf_counter() - start
        print('-' * 40)
        print('%.4f \t%s%s = %i' % (duration, indents, name, n))
        n -= 1
loop = asyncio.get_event_loop()
tasks = [
                loop.create_task(count_down('A', 1)),
                loop.create_task(count_down('B', 0.8)),
                 loop.create_task(count_down('C', 0.5))  ]
start = time.perf_counter()
loop.run_until_complete(asyncio.wait(tasks))
print('-' * 40)
print('Done.')
```

```
----------------------------------------
0.5058          C = 3
----------------------------------------
0.8168      B = 3
----------------------------------------
1.0198  A = 3
----------------------------------------
1.0200          C = 2
----------------------------------------
1.5338          C = 1
----------------------------------------
1.6287      B = 2
----------------------------------------
2.0317  A = 2
----------------------------------------
2.4353      B = 1
----------------------------------------
3.0308  A = 1
----------------------------------------
Done.
```

Blocking function's impact

- We used asyncio.sleep(delay) instead of time.sleep() as time.sleep is a blocking function
- A blocking function cannot be used to trigger task switching
- Try changing asyncio.sleep(delay) with time.sleep
  - The program practically becomes a synchronous program
- If a heavy, long-running task is blocking, then it is literally impossible to implement asynchronous programming with that task as a coroutine.
- We need to develop a coroutine version of that function

Primality testing- asynchronous program

- Synchronous version

```
from math import sqrt

def is_prime(x):
    print('Processing %i...' % x)
    if x < 2:
        print('%i is not a prime number.' % x)
    elif x == 2:
        print('%i is a prime number.' % x)
    elif x % 2 == 0:
        print('%i is not a prime number.' % x)
    else:
        limit = int(sqrt(x)) + 1
        for i in range(3, limit, 2):
            if x % i == 0:
                print('%i is not a prime number.' % x)
                return
        print('%i is a prime number.' % x)
```

```
if __name__ == '__main__':
    is_prime(9637529763296797)
    is_prime(427920331)
    is_prime(157)
```

- The tasks are asymmetric: large, medium, and small
- Switching between them may help avoid users waiting for the program to respond.
- The challenge in making it asynchronous is to decide when to release the execution flow back to the event loop.
- In this example, we take a value 100,000 in the main loop "for i in range (3, limit,2)" and introduce an opportunity to switch tasks

Prime checking - asynchronous program

```
from math import sqrt
import asyncio

async def is_prime(x):
    print('Processing %i...' % x)
    if x < 2:
        print('%i is not a prime number.' % x)
    elif x == 2:
        print('%i is a prime number.' % x)
    elif x % 2 == 0:
        print('%i is not a prime number.' % x)
    else:
        limit = int(sqrt(x)) + 1
        for i in range(3, limit, 2):
            if x % i == 0:
                print('%i is not a prime number.' % x)
                return
            elif i % 100000 == 1:
                await asyncio.sleep(0)

        print('%i is a prime number.' % x)
```

```
async def main():

    task1 = loop.create_task(is_prime(9637529763296797))
    task2 = loop.create_task(is_prime(427920331))
    task3 = loop.create_task(is_prime(157))

    await asyncio.wait([task1, task2, task3])

if __name__ == '__main__':
    try:
        loop = asyncio.get_event_loop()
        loop.run_until_complete(main())
    except Exception as e:
        print('There was a problem:')
        print(str(e))
    finally:
        loop.close()
```

Synchronous vs asynchronous program outputs

- Synchronous version starts processing a number and finishes it

```
Processing 9637529763296797...
9637529763296797 is a prime number.
Processing 427920331...
427920331 is a prime number.
Processing 157...
157 is a prime number.

Process finished with exit code 0
```

```
Processing 9637529763296797...
Processing 427920331...
427920331 is a prime number.
Processing 157...
157 is a prime number.
9637529763296797 is a prime number.

Process finished with exit code 0
```

concurrent.futures as a solution for blocking tasks

- concurrent.futures module is a high-level interface for implementing asynchronous tasks
- It works seamlessly with asyncio module
- It provides an abstract class called Executor
- Executor contains the skeleton of 2 main classes used in asynchronous threading and multiprocessing
  - ThreadPoolExecutor
  - ProcessPoolExecutor

Changes in the framework

- Previously, we needed 3 components
  - the event loop,
  - the coroutines,
  - and their corresponding futures.
- Now
  - We still need the event loop while utilizing threading/ multiprocessing, to coordinate the tasks and handle their returned results (futures)
  - But for coroutines, we utilize multithreading and multiprocessing and standard Python functions
  - A new component required is PoolExecutor (either thread or process)

Example using ThreadPoolExecutor

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

def count_down(name, delay):
    indents = (ord(name) - ord('A')) * '\t'

    n = 3
    while n:
        time.sleep(delay)

        duration = time.perf_counter() - start
        print('-' * 40)
        print('%.4f \t%s%s = %i' % (duration, indents, name, n))

        n -= 1
```

```
async def main():
    futures = [loop.run_in_executor(
        executor,
        count_down,
        *args
    ) for args in [('A', 1), ('B', 0.8), ('C', 0.5)]]

    await asyncio.gather(*futures)

    print('-' * 40)
    print('Done.')

start = time.perf_counter()
executor = ThreadPoolExecutor(max_workers=3)
loop = asyncio.get_event_loop()
loop.run_until_complete(main())
```

Example using ProcessPoolExecutor

```
from math import sqrt
import asyncio
from concurrent.futures import ProcessPoolExecutor
from timeit import default_timer as timer

#async def is_prime(x):
def is_prime(x):
    print('Processing %i...' % x)

    if x < 2:
        print('%i is not a prime number.' % x)

    elif x == 2:
        print('%i is a prime number.' % x)

    elif x % 2 == 0:
        print('%i is not a prime number.' % x)

    else:
        limit = int(sqrt(x)) + 1
        for i in range(3, limit, 2):
            if x % i == 0:
                print('%i is not a prime number.' % x)
                return

        print('%i is a prime number.' % x)
```

```
async def main():

    task1 = loop.run_in_executor(executor, is_prime, 9637529763296797)
    task2 = loop.run_in_executor(executor, is_prime, 427920331)
    task3 = loop.run_in_executor(executor, is_prime, 157)

    await asyncio.gather(*[task1, task2, task3])

if __name__ == '__main__':
    try:
        start = timer()
        executor = ProcessPoolExecutor(max_workers=3)
        loop = asyncio.get_event_loop()
        loop.run_until_complete(main())
        print('Took %.2f seconds.' % (timer() - start))

    except Exception as e:
        print('There was a problem:')
        print(str(e))

    finally:
        loop.close()
```