Event-Driven Programming & Asyncio

https://youtu.be/PbJWx5o3QC4

Event-driven Programming

So far we’ve been looking at concurrent systems and concurrent solutions to problems.

We’ve looked at using both **threads** and **processes**.

We’ve looked at getting threads and processes to synchronize and communicate.

Our concurrent solutions often exhibited non-deterministic execution, but still generally followed a “flow” of execution. We used synchronization to control the flow.

Event-driven Programming

So, a typical programming solution so far might look like this:

Sequential:

(start) A → B → C → D → E (end)

Concurrent:

(start) A → (B || C || D) → E (end)

Note: in this case B,C,D could be threads or processes.

Event-driven Programming

Event-driven programs usually look quite different.

They usually have an “event loop” that looks for incoming events or event messages.

It is these incoming events that determine what the program will do and when it will do it.

Note that the events are from some outside source and can come in in any order. The program has no way of knowing what even will happen next.

Event-driven Programming

You’ve likely seen this before and used a program that uses event-driven programming.

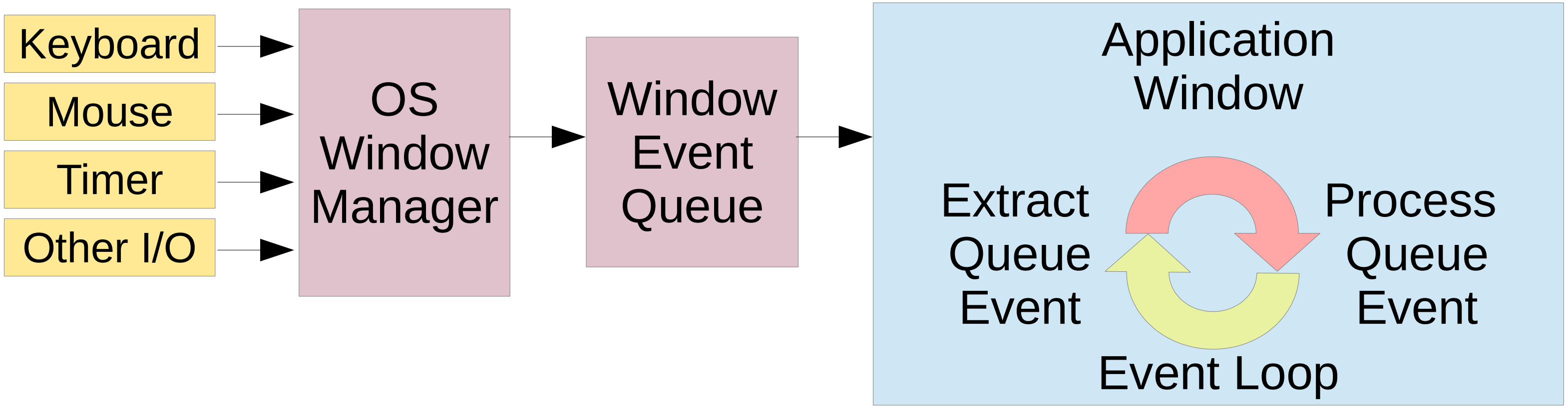
Examples:

pygame programs – usually the main game loop responds to events: keyboard events, mouse events, window events, etc.

Windows GUI programs – also usually have a window that receives and processes incoming events: keyboard, mouse, window, resize, redraw, load, exit, etc.

Event-driven Programming

In a windows program, every window has an associated event queue. It is up the the programmer to extract events from the event queue and then handle those events.



Events will generally be handled by the application one event at a time. The code that handles the events is the event loop.

Event-driven Programming

So the flow of an event driven program looks more like this:

(start) A → { (wait for event) → (handle event) }* → E (end)

Note that the (wait) → (handle) code repeats indefinitely and is usually ended by an event causing the loop to exit.

Also note that the number and type of events can be small or very large.

Event-driven Programming

This method of programming can allow us to think of solutions in terms of a set of possible events that could occur and how we’d deal with each.

This can be ideal for simplifying the design of certain types of systems: operating systems, GUI systems, games, automotive systems, industrial control systems, etc.

To write code this way we simply need to account for all possible events and write handlers for each of those events.

Event-driven Programming

But this type of programming is a different way of thinking and can also create some difficulties.

It is also offer a way of writing and controlling solutions that have concurrent elements of execution but only use a single thread.

The python module we are going to look at is called "`asyncio`".

Event-driven Programming

Here is a summary of what the asyncio module does:

asyncio is a library to write concurrent code using the async/await syntax.

The anyncio module provides infrastructure for writing single-threaded concurrent code using coroutines, multiplexing I/O access over network sockets and other resources, running network clients and servers, and other related primitives.

asyncio is often a perfect fit for IO-bound and high-level structured network code.

Event-driven Programming

Here is a summary of what the asyncio module does:

asyncio is a library to write concurrent code using the async/await syntax.

The anyncio module provides infrastructure for writing single-threaded concurrent code using coroutines, multiplexing I/O access over network sockets and other resources, running network clients and servers, and other related primitives.

asyncio is often a perfect fit for IO-bound and high-level structured network code.

Event-driven Programming

Here is a summary of what the asyncio module does:

asyncio is a library to write concurrent code using the async/await syntax.

The anyncio module provides infrastructure for writing single-threaded concurrent code using coroutines, multiplexing I/O access over network sockets and other resources, running network clients and servers, and other related primitives.

asyncio is often a perfect fit for IO-bound and high-level structured network code.

Event-driven Programming

“asyncio” provides both high-level and low-level APIs.

High-level asyncio APIs:

- run and control Python coroutines concurrently
- perform network IO and IPC
- control subprocesses
- distribute tasks via queues
- synchronize concurrent code

Event-driven Programming

“asyncio” provides both high-level and low-level APIs.

Low-level asyncio APIs:

- create and manage event loops, which provide asynchronous APIs for networking, running subprocesses, handling OS events and signals, etc
- implement efficient protocols using transports
- bridge callback-based libraries and code with async/await syntax

Event-driven Programming

The asyncio event loop:

The core element of all systems using asyncio is the ‘event loop’.

The event loop is a loop that schedules / runs asynchronous tasks, handles asynchronous I/O operations, and runs subprocesses.

The event loop is related to code that uses the async/await feature on things that are “awaitable”.

Event-driven Programming

Awaitable items:

Awaitable items can be used in an await expression.

There are three main types:

- Coroutines
- Tasks
- Futures

Let’s look at each one of these types...

Event-driven Programming

Coroutines:

Coroutines are a special type of function that is “awaitable”.

To create one of these we use “async def” in place of the usual “def” when defining a function.

An “async def” defined function is called a **“coroutine function”**

An object returned from a coroutine function is called a **“coroutine object”**.

Event-driven Programming

Tasks:

Tasks are used to schedule coroutines to execute concurrently.

We will need to make at least one task that will run the event loop.

Thus there is usually at least one “main” task that will be started.

Tasks are started by passing a coroutine to asyncio.run.

Event-driven Programming

Futures:

A future is a low-level object that is awaitable and represents a result of some asynchronous operation that will eventually be produced.

Think of this like an order that has been placed that will eventually be filled.

Like at a restaurant. You place an order that will result in food, but it may take a while. So futures are a way to track and handle expected future results.

Event-driven Programming

Simple Example:

```
import asyncio
import time

async def future_print(message,delay):
    await asyncio.sleep(delay)
    print(message)

asyncio.run(future_print("This is a test!",1))
asyncio.run(future_print("goodbye!",2))
```

Event-driven Programming

That previous example seems trivial because it simply executes one function call then the next.

There is no concurrent execution really happening here.

To run coroutines concurrently and help control and coordinate these, there are some additional functions…

Event-driven Programming

asyncio.gather - takes a sequence of awaitables, returns an aggregate list of successfully awaited values.

asyncio.shield - prevent an awaitable object from being cancelled.

asyncio.wait - wait for a sequence of awaitables, until the given ‘condition’ is met.

asyncio.wait_for - wait for a single awaitable, until the given ‘timeout’ is reached.

asyncio.as_completed - similar to gather but returns Futures that are populated when results are ready.

Event-driven Programming

Concurrent Example:

```
import asyncio
import time

async def future_print(message,delay):
    await asyncio.sleep(delay)
    print(message)

async def main():
    await asyncio.gather(future_print("This is a test!",1),
                         future_print("goodbye!",2))

asyncio.run(main())
```

Event-driven Programming

One thing to keep in mind: **await** can only be used in a function defined with **async**.

Note that in the previous example, both tasks are sheduled to execute in the event loop so the execution is overlapped, but unlike with threading, cooperative multi-tasking is used.

In other words, the gather is simultaneously starting and waiting on multiple tasks.

Let’s look at a use-case that is more practical…

Event-driven Programming

Downloading Multiple Files

Downloading multiple files sequentially might look like this:

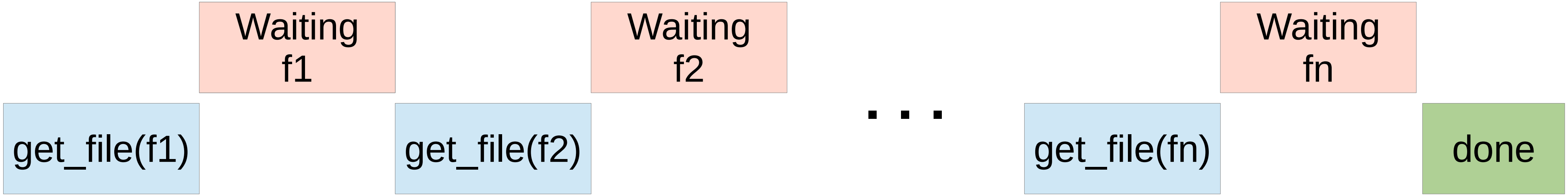
```
get_file(f1)
get_file(f2)
get_file(f3)
    . . .
get_file(fn)
```

This might be something that a web-crawler would need to do.

Event-driven Programming

Downloading Multiple Files

In the sequential version we are waiting for each function call to finish before going to the next one.



This means that each must wait for all previous downloads to complete.

Event-driven Programming

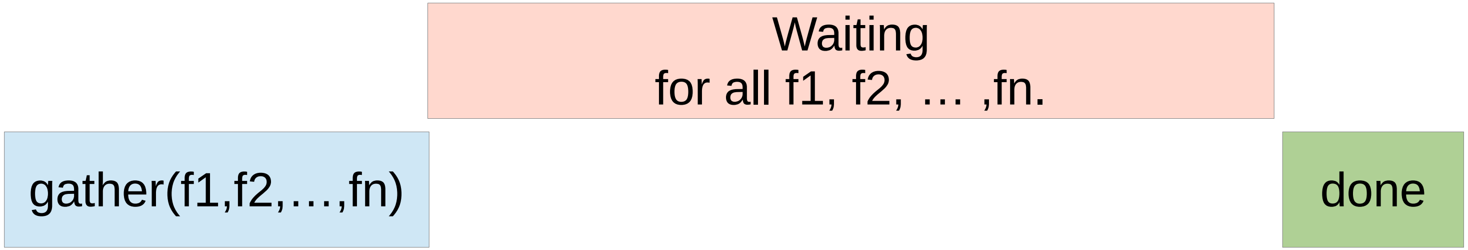
Sequential Implementation:

```
import urllib.request
import time

image_url_list=["images-assets.nasa.gov/image/PIA04921/PIA04921~orig.jpg",
            "images-assets.nasa.gov/image/PIA10600/PIA10600~orig.jpg",
            "images-assets.nasa.gov/image/PIA11999/PIA11999~orig.jpg"]

def download_all_sequential(url_list):
    start_time=time.time()
    print("Starting all downloads.")
    for file in url_list:
        output_fname=file.split('/')[-1]
        file_start_time=time.time()
        download_file("http://"+file,output_fname)
    stop_time=time.time()
    print("All downloads completed in",stop_time-start_time,"seconds.")
    return stop_time-start_time

def download_file(url,outfile):
    file_start_time=time.time()
    urllib.request.urlretrieve(url, outfile)
    print(outfile,"Done in:",time.time()-file_start_time,"seconds.")

ts=download_all_sequential()
```

Event-driven Programming

Downloading Multiple Files

In the asyncio version we are able to start all of the downloads as coroutines. All can be started simultaneously...



And if we start them all within an await asyncio.gather() then we can still wait for all downloads to finish.

Event-driven Programming

Asyncio Implementation:

```
import time
import asyncio
import aiohttp

image_url_list=["images-assets.nasa.gov/image/PIA04921/PIA04921~orig.jpg",
            "images-assets.nasa.gov/image/PIA10600/PIA10600~orig.jpg",
            "images-assets.nasa.gov/image/PIA11999/PIA11999~orig.jpg"]

async def download_all_async():
    start_time=time.time()
    tasks=[]
    print("Starting all downloads.")
    async with aiohttp.ClientSession() as session:
        for file in image_url_list:
            output_fname=file.split('/')[-1]
            tasks.append(asyncio.ensure_future(download_file_async(session,"http://"
                                                                             +file,output_fname)))
        await asyncio.gather(*tasks)
    stop_time=time.time()
    print("All downloads completed in",stop_time-start_time,"seconds.")
    return stop_time-start_time

#continued...
```

Event-driven Programming

Asyncio Implementation:

```
#continued…

async def download_file_async(session,url,outfile):
    file_start_time=time.time()
    async with session.get(url) as response:
        data=await response.read()
        fp=open(outfile,"wb")
        fp.write(data)
        fp.close()
    print(outfile,"Done in:",time.time()-file_start_time,"seconds.")

tc=asyncio.run(download_all_async())
```

Event-driven Programming

Notes:

This implementation uses the aiohttp module. This module has asynchronous versions of http functions. Essentially these allow us to asynchronously get data from multiple sources.

To use the functions from this module we need to use async with and await.

To run the event loop that starts all of the downloads, we need to use asyncio.run() and pass the coroutine download_all_async().

Keep in mind that asyncio programs are a form of cooperative multitasking so we can't simply call urlliburllib.request.urlretrieve() as we did before.   Why not?

Event-driven Programming

More Notes:

Other languages also support asynchronous behavior like this: notably, javascript makes heavy use of this type of behavior.

Asyncio can be a powerful tool, especially for handling numerous IO-bound tasks or maintaining responsiveness of a program (like a GUI) during long operations.

Even though asynchronous programming is a form of cooperative multitasking, the tasks are still being executed concurrently. This means that there is need for synchronization and communication mechanisms. Asyncio has support for: Lock(), Event(), Condition(), Semaphore(), BoundedSemaphore(), Queue(), PriorityQueue(), LifoQueue()

You should investigate event-driven asynchronous programming further.

Here is a link with more details: https://docs.python.org/3.6/library/asyncio.html

Event-driven Programming

That’s all for today!

Stay Safe!