CONCURRENCY

PYTHON CONCURRENCY CONCEPTS

FUNDAMENTAL CONCURRENCY CONCEPTS

- Concurrency
- Processes
- Threads
- Time-slicing/Interleaving and Parallelism
- OS Thread Scheduler
- The Python Global Interpreter Lock
- CPU vs I/O Bound Workloads
- Preemptive vs Cooperative multitasking
- Async Programming

CONCURRENCY

→ Concurrency is a **concept**

→ how a program, or algorithm, is **structured**

→ not how it is executed

→ there are different ways of executing concurrent code

An algorithm is concurrent when it is **structured** into several sub parts that can be executed **out of order** (or **partially** ordered), without affecting the **outcome**

→ code sections are not necessarily executed in a specific order

→ the final result remains the same

EXAMPLE

Suppose we want to average a list of N numbers $x_0, x_1, x_2, \ldots, x_N$

sequential algorithm

add all numbers → sum

count number of elements → count

$\text{sum} = \text{sum}([x_0, x_1, x_2, \ldots, x_N])$

$\text{count} = \text{len}([x_0, x_1, x_2, \ldots, x_N])$

$$\text{avg} = \frac{\text{sum}}{\text{count}}$$

concurrent algorithm

split list into chunks

$x_0, \ldots, x_j$ | $x_{j+1}, \ldots, x_k$ | $x_{k+1}, \ldots, x_N$

sum_1 count_1 | sum_2 count_2 | sum_3 count_3

concurrent computations, or modules → **fragment**

$$\text{avg} = \frac{\text{sum}_1 + \text{sum}_2 + \text{sum}_3}{\text{count}_1 + \text{count}_2 + \text{count}_3}$$

→ no matter the order of execution of the concurrent fragments, the final result is the same

HOW ARE CONCURRENT FRAGMENTS EXECUTED?

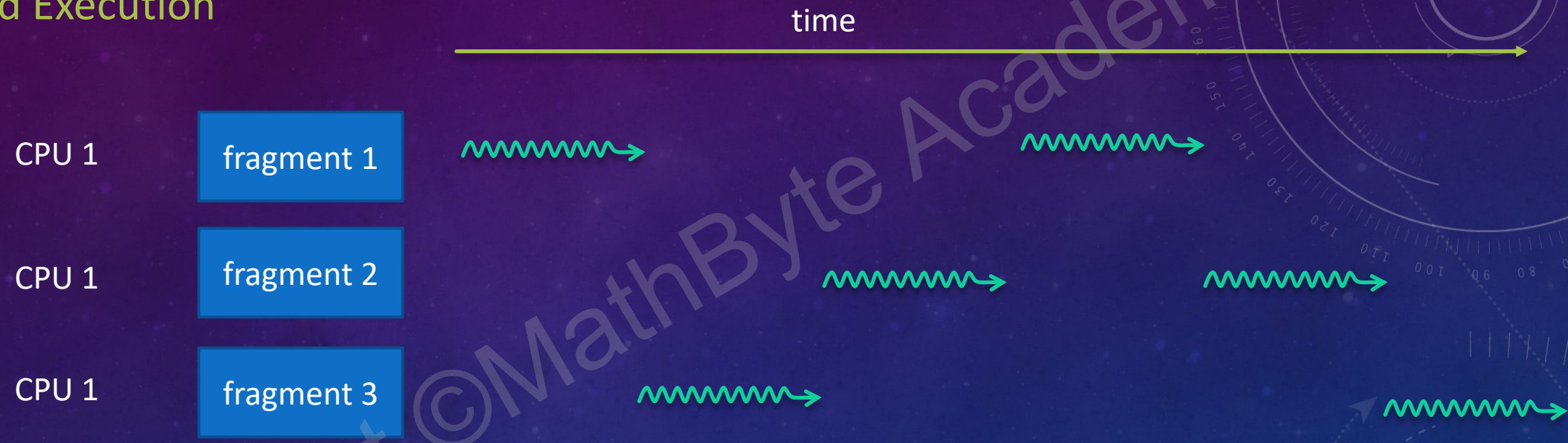
Parallel Execution

→ requires multiple CPUs (or cores)

→ code executor must be able to take advantage of multiple CPUs/cores

HOW ARE CONCURRENT FRAGMENTS EXECUTED?

Time-Sliced Execution



→ only a single CPU/Core is needed

→ every time execution is paused, state must be saved somewhere

→ every time execution is resumed, saved state must be loaded

context switching

→ each context switch has a performance cost

→ also called **interleaving**

PROCESSES

→ think of a process as an application instance executing in your OS Excel Web Browser Python App

→ processes run concurrently

→ may run in parallel on machine with multiple CPUs/cores

→ or using interleaving/time-slicing (even on a multi CPU/core machine)

→ each process runs independently

→ memory is not shared between processes (as well as some other resources)

→ possible states

running → has access to CPU and is currently executing code

ready → could run, if it had access to a CPU

blocked → waiting for something to happen (e.g. I/O op, such as receiving data from a web API)

THREADS

→ each process is executed using one or more threads (in modern operating systems)

→ always starts with at least one thread, often called the main thread

→ as well as "global" data (shared state)

→ concurrent code can be executed using multiple threads

→ multithreaded process → otherwise called a single-threaded process

→ threads have access to shared resources in the process (global data, open files, etc)

→ but threads can also have their own "private" resources (e.g. local variables)

→ sometimes called thread-local data

→ may run in parallel on machine with multiple CPUs/cores

→ and/or using time-slicing/interleaving (even on a multi-CPU/core machine)

SCHEDULING

→ OS has a piece of software called the scheduler

→ it decides when to run threads

(remember, multiple running processes results in multiple threads, even if each process is single-threaded)

→ the scheduler runs, pauses and restarts threads all the time

→ the **scheduler decides when to do that**, not us as the application developer

→ when a thread is paused, it's current state must be saved (in memory) – that's a lot of work!

→ when a thread is resumed, it's original saved state must be restored – that's also a lot of work!

→ this is called context switching, and has a performance penalty

→ we say the scheduler preempts the thread → hence the term preemptive multitasking

→ context switching has a performance penalty **→ scheduler decides when, not us**

MULTITASKING

→ general term used to denote that multiple “things” are running concurrently

→ could be multiple processes

→ could be multiple threads

→ running Excel, Python app and Browser at the same time

→ or any other way of running code fragments concurrently

→ Note how I use the term concurrent, not parallel!

→ multitasking is a concept, running concurrent code, and can be achieved in a variety of ways

THE PYTHON GLOBAL INTERPRETER LOCK

→ also called the **GIL**, for short

→ the way CPython is written, only allows for interleaved/time-sliced running of multiple threads

→ even when we create multiple threads, only one thread is allowed to run at a time

→ no parallelism

→ does not take advantage of multiple CPUs/cores

→ makes writing the CPython interpreter easier

→ actually speeds up single-threaded processes

→ which is how most Python apps are written

→ there have been attempts to remove the GIL (gilectomy!), but none have been successfully accepted yet

→ often causes significant slowdowns in existing single-threaded Python apps

ANOTHER APPROACH TO MULTITASKING

→ we saw that the scheduler implements preemptive multithreading

→ we don't really have much say over when one thread is stopped and another started

→ this can be difficult when using shared resources – a lot of care has to be taken to ensure things run as expected even if the task is interrupted preemptively

→ another approach is to write code where we explicitly say when (or where) a task can be interrupted to allow another task to run concurrently

→ cooperative multitasking

→ this can be really useful in cases where a code fragment is waiting on an external resource to complete and return something

→ waiting for database response

→ waiting for a file to be opened and read by OS

→ waiting for response from web API

→ waiting for another process to return something

} waiting for some I/O operation to complete

WORKLOADS

→ characterize the type of work done by a chunk of code

→ what kind of work is the code dealing with the majority of the time?

→ spends a lot of time running computations

→ would benefit from a more powerful CPU/core → CPU workload

→ spends a lot of time waiting for I/O to complete

→ would not benefit from a more powerful CPU/core → I/O workload

→ most workloads are **mixed**

→ but have a **dominant** bottleneck, CPU or I/O → CPU bound

→ I/O bound

CPU BOUND WORKLOADS IN PYTHON - THREADING

→ assuming the workload can be written into concurrent fragments

→ running fragments using multiple threads would seem like a good solution to increase performance

→ but, the GIL!

→ threads will run interleaved

→ not in parallel across multiple CPUs/cores

→ so, no parallel execution

→ and we have the additional cost of context switches

→ generally means our code will actually run slower than single threaded

CPU BOUND WORKLOADS IN PYTHON - MULTIPROCESSING

→ one way to spread CPU bound loads across multiple CPUs/cores is to start multiple parallel processes

→ Python calls this multiprocessing

→ process state is not shared – they are independent

→ means communicating between main app and other processes is more difficult/costly than with threading

→ main application can spawn multiple processes

→ can pass data to a spawned process

→ can get results back from a process

} data marshalling / unmarshalling

→ not scalable! → limited to the CPUs/cores of a single machine

→ in modern computing, and with ever larger data sets, scaling limited to a single machine is often not sufficient

→ reality is that although you can use multiprocessing to take full advantage of a single machine

→ if you need that level of performance, you will likely need to write more complex code to scale across multiple machines

→ which may mean you don't even need multiprocessing

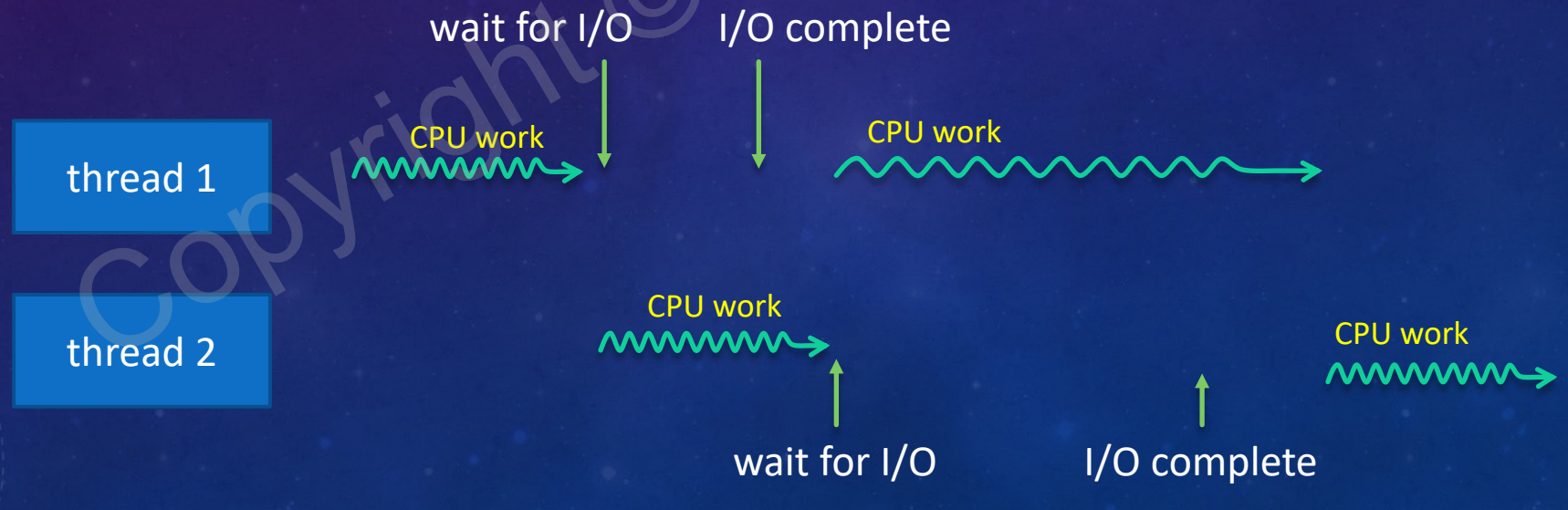
I/O BOUND WORKLOADS IN PYTHON - THREADING

→ CPython is inherently single-threaded

→ but since most of the time is spent waiting for an I/O operation to return something

→ running code concurrently makes sense, even if single-threaded

→ switching to another thread while one is waiting for I/O means other code fragments can be executing code that they would otherwise have been blocked from running until the I/O operation completed



MULTITHREADING DANGERS

→ multithreading concurrent I/O bound fragments should improve performance

→ writing multithreaded code is difficult

→ preemptive – we don't know exactly when a thread will be interrupted

→ have to be careful with shared state – easy to run into problems

→ easy to make mistakes

→ difficult to debug

shared state→ balance

deposit $500

thread 1 read balance compute balance + $500 update balance to new value

withdraw $100

thread 2 read balance compute balance - $100 update balance to new value

SCENARIO 1

shared state → balance → initial balance $500

deposit $500

thread 1

read balance → 500
new_balance = 500 + 500 = 1000
set balance = 1000

withdraw $100

thread 2

read balance → 1000
new_balance = 1000 – 100 = 900
set balance = 900

Ending Balance
$900

this is correct

SCENARIO 2

shared state → balance → initial balance $500

deposit $500

thread 1

read balance → 500

new_balance = 500 + 500 = 1000
set balance = 1000

Ending Balance
$400

withdraw $100

thread 2

read balance → 500
new_balance = 500 – 100 = 400

set balance = 400

this is wrong

SCENARIO 3

shared state → balance → initial balance $500

deposit $500

thread 1

read balance → 500
new_balance = 500 + 500 = 1000

set balance = 1000

withdraw $100

thread 2

read balance → 500
new_balance = 500 – 100 = 400
set balance = 400

Ending Balance
$1000

this is wrong

→ there are other scenarios, some of which will produce the correct result, and some the wrong result

→ possible to guard against such issues (locks, queues, etc)

→ preemptive nature of threads and shared state is causing this

COOPERATIVE MULTITASKING

→ writing multithreaded code can be difficult

→ multithreading in CPython does not help with CPU bound workloads

→ but can help with I/O bound workloads

→ is there a simpler/safer alternative for writing concurrent code for I/O bound workloads?

→ yes!

→ asynchronous programming → uses `asyncio` module and special keywords: `async`, `await`, `yield`

→ it is a form of cooperative multitasking

→ in our code we specify exactly where a task can be paused, and another one allowed to run

→ still single-threaded execution

→ can significantly improve performance for I/O bound workloads

PYTHON ASYNC PROGRAMMING

→ offers an easier/safer alternative to multithreading

→ both async and multithreading are inherently single-threaded (GIL)

→ performance improvements for I/O bound workloads primarily

→ safer than preemptive multitasking, but adds some complexity to our code

→ need to write concurrent code (just like with threading or multiprocessing)

→ but, unlike threading, we have to explicitly let Python know how the different fragments can interrupt each other and work together

→ this adds a little bit of complexity to our code – blocking and non-blocking code

→ generally simpler overall than multithreading

→ easier to debug

→ but does require 3rd party I/O libraries to be async enabled

THE ASYNCIO EVENT LOOP

→ basic idea is we “register” concurrent code fragments that run code, and intermittently indicate that a particular line of code would be a good time to interrupt the function

→ these calls to async enabled functions are called tasks

→ we end up with a collection of these tasks that need to be executed

→ Python creates an event loop – single threaded

→ it runs one task at a time

→ the task indicates that it is ready to be switched (or completes) – usually because it is waiting for I/O response

→ the event loop gets control back, and starts running another task, until that task indicates it is ready to be switched (or completes)

→ continue doing this until all tasks are completed

THE ASYNCIO EVENT LOOP

→ when a task runs, it is executing code, and at some point should yield control back to the event loop

→ functions or tasks that run and do not yield control back before they have completed, are called blocking

→ it is usual for tasks to contain blocking code – after all we do have some computations to do

→ as long we yield control once in a while, especially when we are waiting for I/O that is fine

→ this means that to get the most benefit from async-based concurrency, we need to be able to yield control (not block) when performing I/O operations

→ many libraries available that implement non-blocking I/O

	→ calling a database	→ making an HTTP request	→ reading/writing Python queues
blocking	`psycopg2`	`requests`	`queue.Queue`
non-blocking	`psycopg3`	`aiohttp`	`asyncio.Queue`

RECAP OF CONCURRENT PROGRAMMING OPTIONS

multiprocessing

no shared data

passing data between processes is expensive

useful for CPU bound workloads

multithreading

shared data

care needed for shared data

preemptive

OS decides when to switch

useful for performance improvements in I/O bound workloads

useful for other multitasking use cases that have nothing to do with performance improvement, or when needing concurrency with blocking code

async

shared data

care needed for shared data

cooperative

we decide when we can yield control back to main event loop

useful for performance improvements in I/O bound workloads, but requires I/O code and libraries to be async aware (many useful libraries have async equivalents)