# Concurrency

WEEK 12

# Let’s take a break from Patterns

We still have 3 Patterns to learn:

- Builder
- Lazy Initialization
- Mediator

But let’s step away from this for a bit.

Let’s add concurrency and multiprocessing to your toolbelt instead!

A super interesting, fun, yet complex topic.

# What is Concurrency?

Concurrency in Computer Science refers to the act or the “perceived act” of processing multiple events in parallel.

Most (if not all) modern computers are concurrent systems. They can run multiple of the following at the same time:

- Threads
- Tasks
- Process

Although, only Process’s actually work in parallel (that is simultaneously), Threads and Tasks give the illusion of parallel processing.

We write concurrent systems to speed up the time it takes to run an application and make it more responsive.

**Definition:**

*noun*

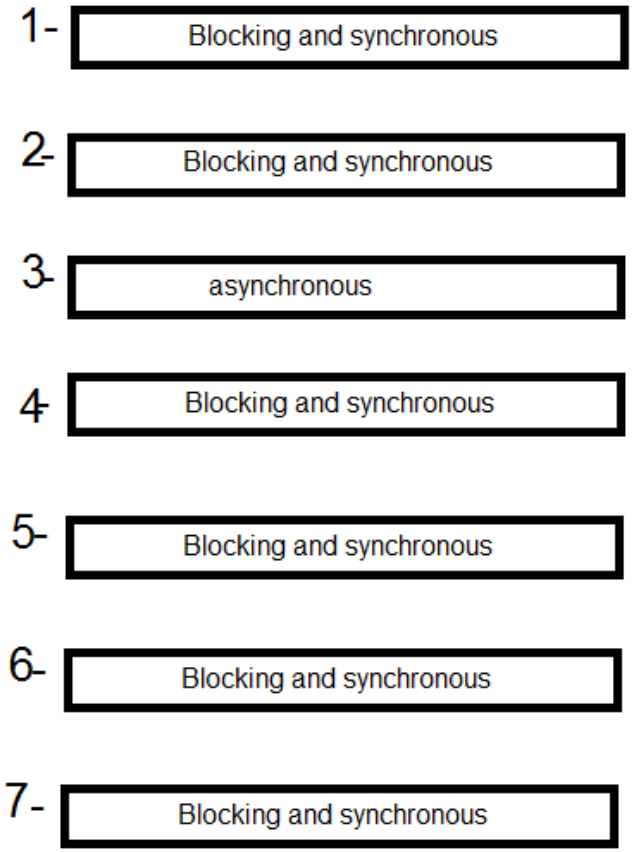
1. the act of concurring.
2. accordance in opinion; agreement: *With the concurrence of several specialists, our doctor recommended surgery.*
3. cooperation, as of agents or causes; combined action or effort.
4. **simultaneous occurrence;**
   *Geometry*. a point that is in three or more lines simultaneously.
5. *Law*. a power equally held or a claim shared equally.

# Asynchronous Programming

**Synchronous code** is code **that runs in a set sequence**. This is the kind of code we have been writing so far.

**Asynchronous code** is what happens when code:

- Defines a discrete unit of work
- This unit of work **runs in "parallel"** from the main thread.
- When this unit of work is complete, **it notifies the main thread** about the completion or failure of its task. (usually done through a **callback**).



# Concurrency in Python

In python we can write asynchronous code in 3 ways:

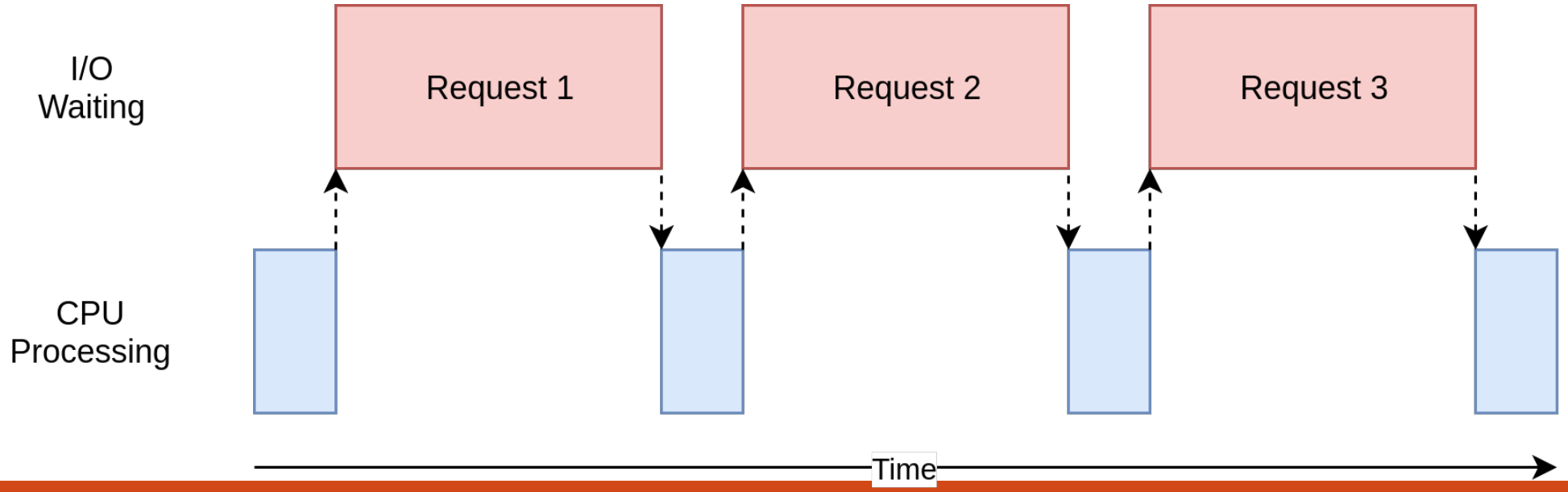
- **Threading**
  - Creating different threads that execute parts of our code in parallel.
  - Runs on the same core.
  - Good for I/O Bound problems
- **Tasks**
  - Units of work defined using AsyncIO and Coroutines (More on this later).
  - Good for I/O Bound problems
- **Multiprocessing**
  - Creates a whole new process with its own address space and python interpreter.
  - Runs on different CPU cores.
  - Good for CPU Bound problems

# I/O vs CPU bound

# I/O Bound

An I/O Bound problem causes your program to run slowly because **it spends the bulk of its time waiting for Input/Output operations**. This usually happens when your program is working with things that are slower than the CPU, such as:

- sending/receiving data across a network connections
- writing/reading from a file, etc.

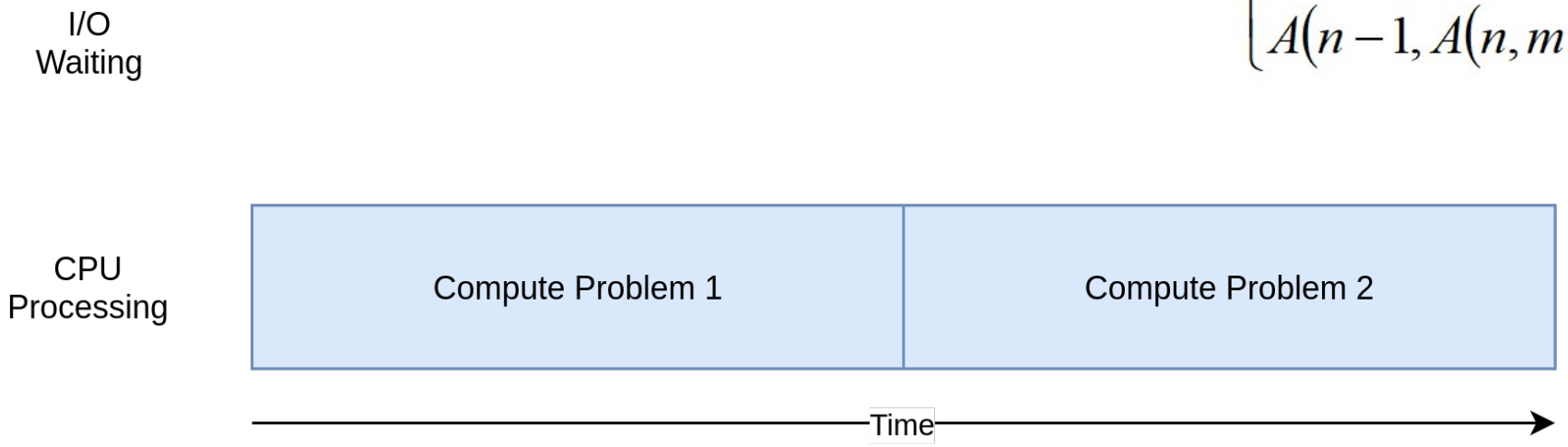


# CPU Bound

A CPU Bound **problem spends the bulk of it's time processing data** instead of waiting on Input/Output.

Remember Ackermann's Algorithm? We looked at it while learning about profiling. That is a CPU Bound problem.

$$A(n,m) = \begin{cases} m+1, if\ n = 0; \\ A(n-1,1), if\ n \neq 0, m = 0; \\ A(n-1, A(n, m-1)), if\ n > 0, m > 0. \end{cases}$$



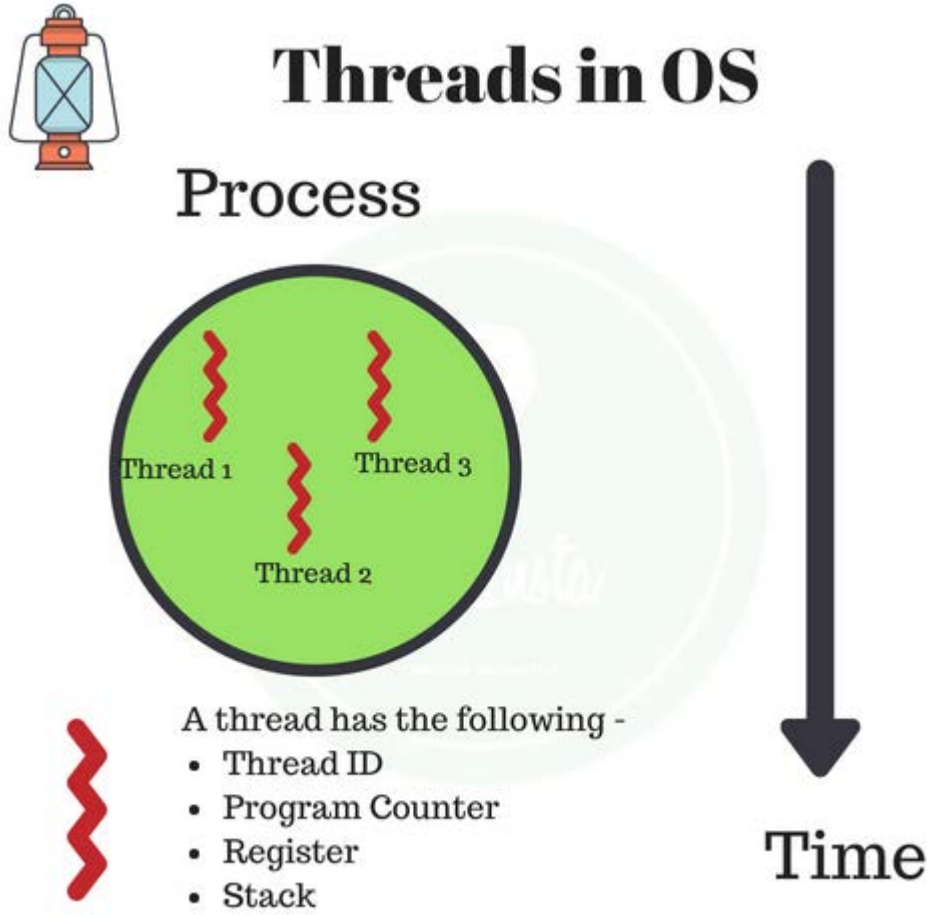
# What is a Thread?

A thread is a **sequence of instructions** within a process that can be **executed independently**.

A process is a program that has been loaded into memory along with all the resources it needs to operate

A **process can run multiple threads**, each of which executes a stream of instructions concurrently and independently.

Threads operate within the **same address space** and as such, share the same resources and data.



# Pre-emptive Multitasking

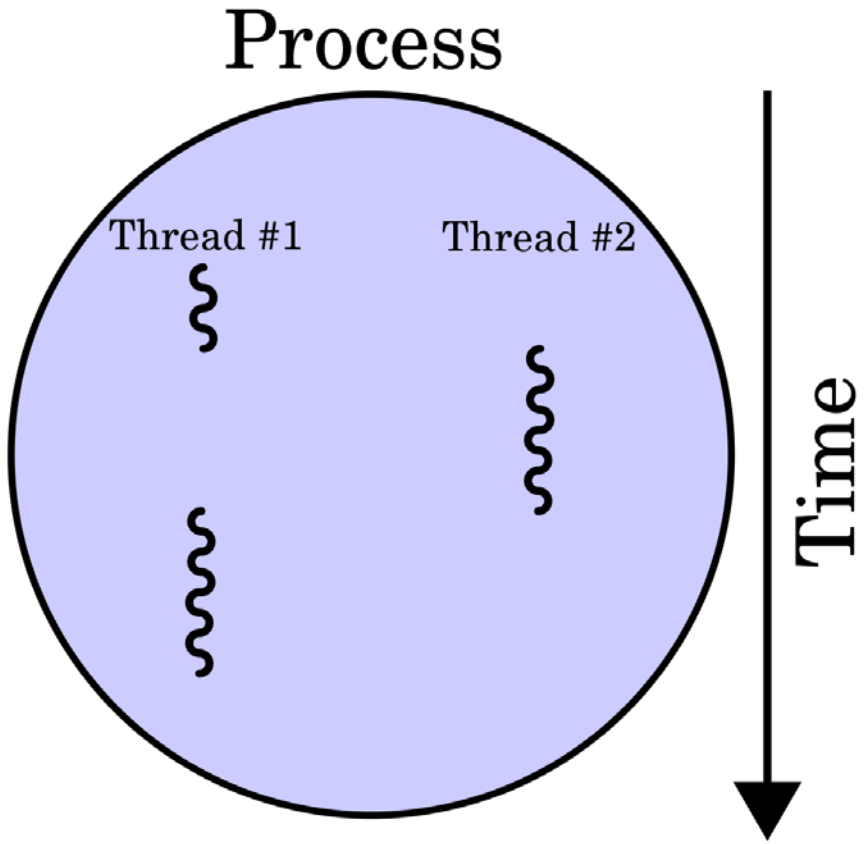
All threads that belong to a process run on the single processor (or CPU Core).

That is, only a single thread can be active at any given time.

So threads are being processes sequentially and not parallel

The Operating System (OS) is responsible for interrupting a thread and switching to a different thread to give the illusion of concurrency. This can happen in the middle of an instruction.

In other words, the OS can pre-empt your thread to make a switch. This is known as **Pre-emptive multitasking.**



# Cooperative Multitasking

Using coroutines and asyncio (we will look at this next class) we can emulate tasks. These are independent units of works that can run concurrently.

These are similar to threads in the sense that only one task at a time get's executed. They create the illusion of concurrency.

The biggest difference here is that instead of the OS pre-empting a switch, we as developers do this via code. That is, we write code that can announce a switch.

Each task must cooperate by announcing when they are ready to be switched out. This allows the developer to be in control and avoid bugs that occur when switching occurs say, in the middle of a statement. This is known as **Cooperative Multitasking.**

# Setting up a Thread

The first step is to import the threading module.

There are three ways to create threads:

- Create an object of type Thread and assign a target function to it.
- Inherit from Thread and create your own thread class (thread_download_site_class.py***)
- Create a Thread Pool Executor (Used to execute multiple threads)

If we inherit and create our own Thread subtype then we need to override the **`run()`** method with the code that will execute on the thread.

We can call the **`start()`** method to start a thread.

# Right, let’s check out some code

Let’s check out the long way of writing multi-threaded code

multi_thread.py # manually creating Thread objects

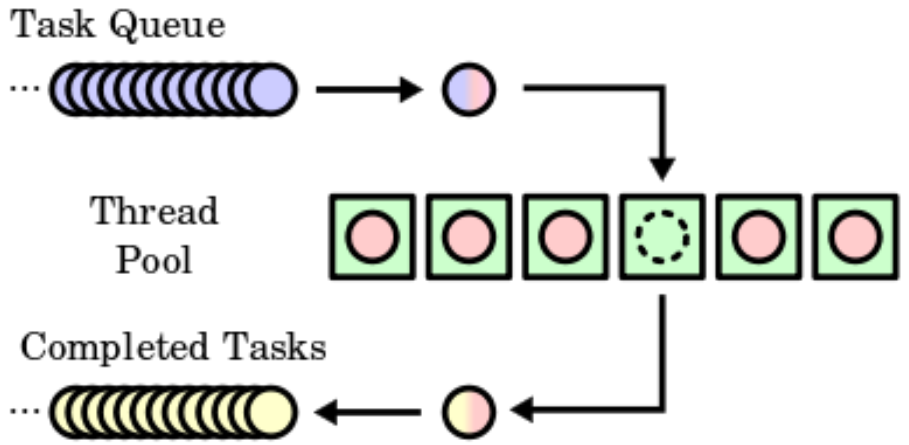
# ThreadPoolExecutor

After a certain number of threads (dependent on the system) the gains in speed have diminishing returns.

There is overhead in creating and running threads.

The thread pool executor implements the **object pool pattern** and creates a pool of threads.

It hides the complexity of creating and maintaining a queue of threads.



# ThreadPoolExecutor

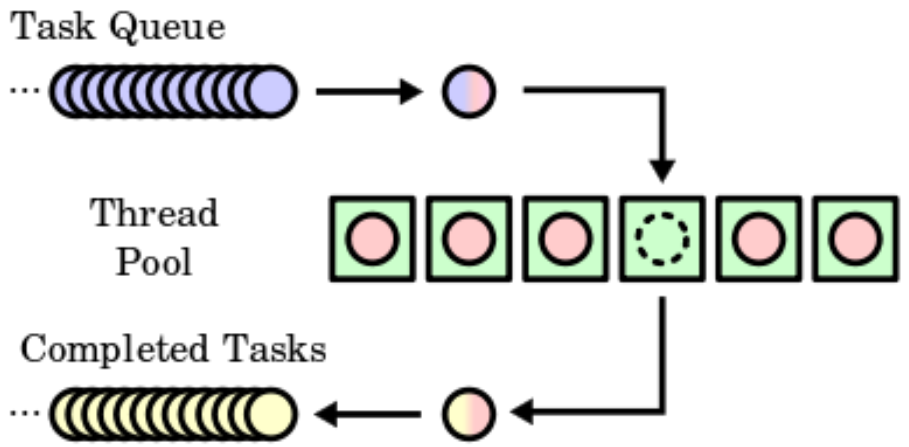
In this pattern, there is a max number of objects maintained in a pool.

These objects are usually expensive to execute.

Say there are 6 threads in a pool, and 100 tasks.

The Object pool pattern (in this case the ThreadExecutorPool) ensures that only a maximum of 6 are running at a given time.

It recycles the objects (threads) in the pool and assigns them to new tasks from a queue




thread_pool_executor.py

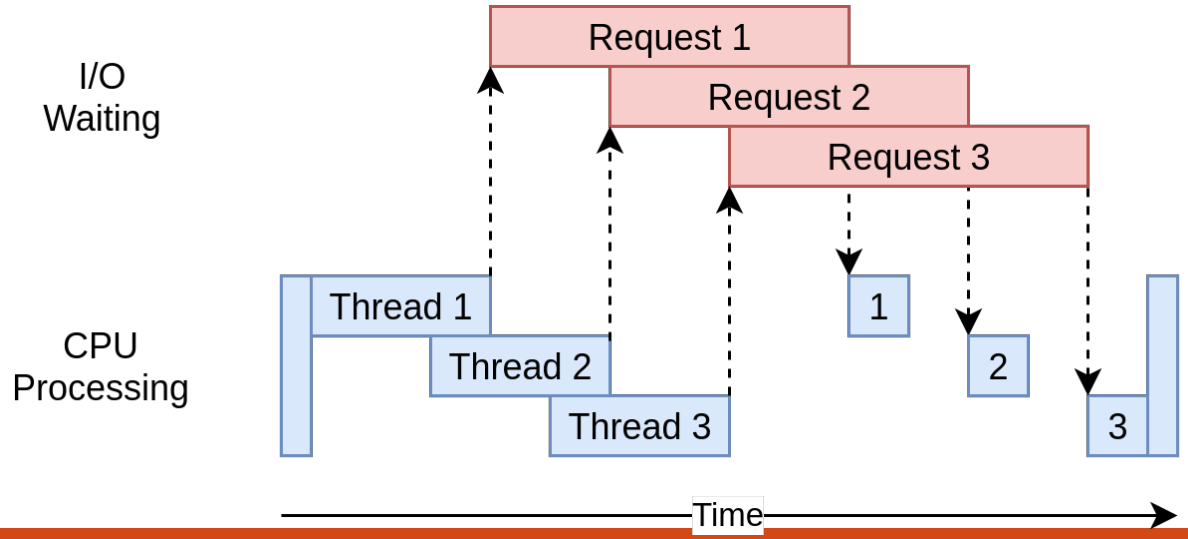
# Threads - Advantages

Code runs faster!

Great for I/O Bound situations.

Most languages have support for threading in some form or the other.

They share the same memory space so it’s easy to share data.



# Threads - Disadvantages

Not so good for CPU Bound situations

Race conditions (next topic)

Deadlocks

More code to write and can be hard to debug.

# Threads – Race Conditions

A race condition occurs when two or more concurrent streams of executions (threads, tasks, processes) try to access the same resource.

If the sequencing of the access is not controlled, then the shared resource may be altered out of order and this can lead to abnormal behaviour.

# Threads – Race Conditions

For example, say we have 2 threads responsible for accessing a data service and saving data to different files.

I want to save data to separate files, using two separate threads

If we don't control access, this may happen:

- Thread 1 sets dest_file to `"output_1.txt"`
- Thread 2 sets dest_file to `"output_2.txt"`
- Thread 1 writes `"Thread 1 was here"`
- Thread 2 writes `"Thread 2 was here"`

All of a sudden we only have 1 resultant file (output_2.txt) with the following 2 lines :

```
Thread 1 was here
Thread 2 was here
```

This is clearly not the intended result.

# Race Condition

Let’s look at an example

thread_race_condition.py

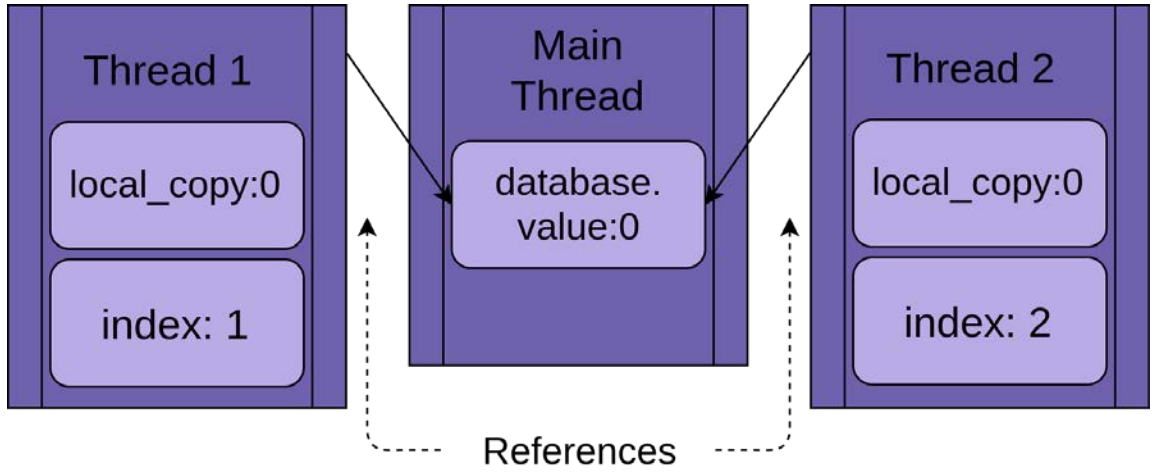
# Race Conditions – Whats happening?

Both threads are accessing the same database and its store value.

Let's break this down.

First, let's see what happens when only 1 thread accesses the database.

Then, let's see what happens when 2 threads access the database.

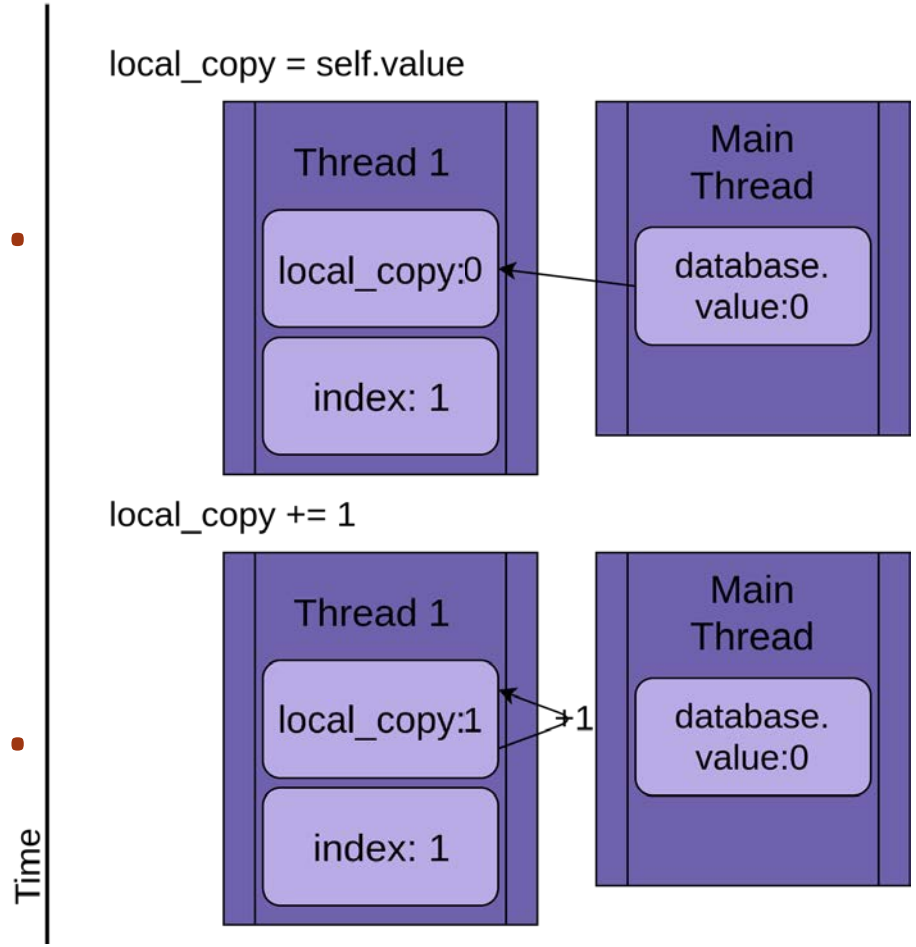


# Race Conditions – Single Thread

1. Thread one access value and stores it in **`local_copy`**.

Important to note, each thread maintains its own store of local variables.

2. Thread one increments its copy of **`local_copy`**.

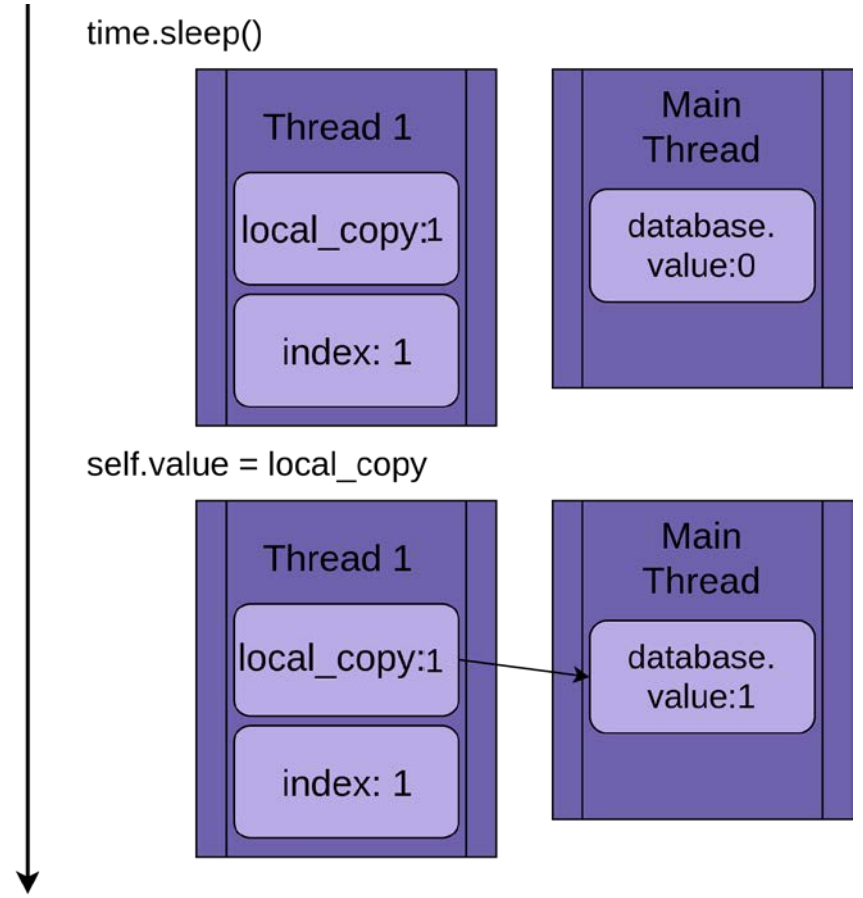


# Race Conditions – Single Thread

3. Thread 1 sleeps for 0.1 seconds

4. Thread one overrides the **`value`** attribute with the value stored in it's **`local_copy`**.

It makes sense that value in database has value of 1 at the end



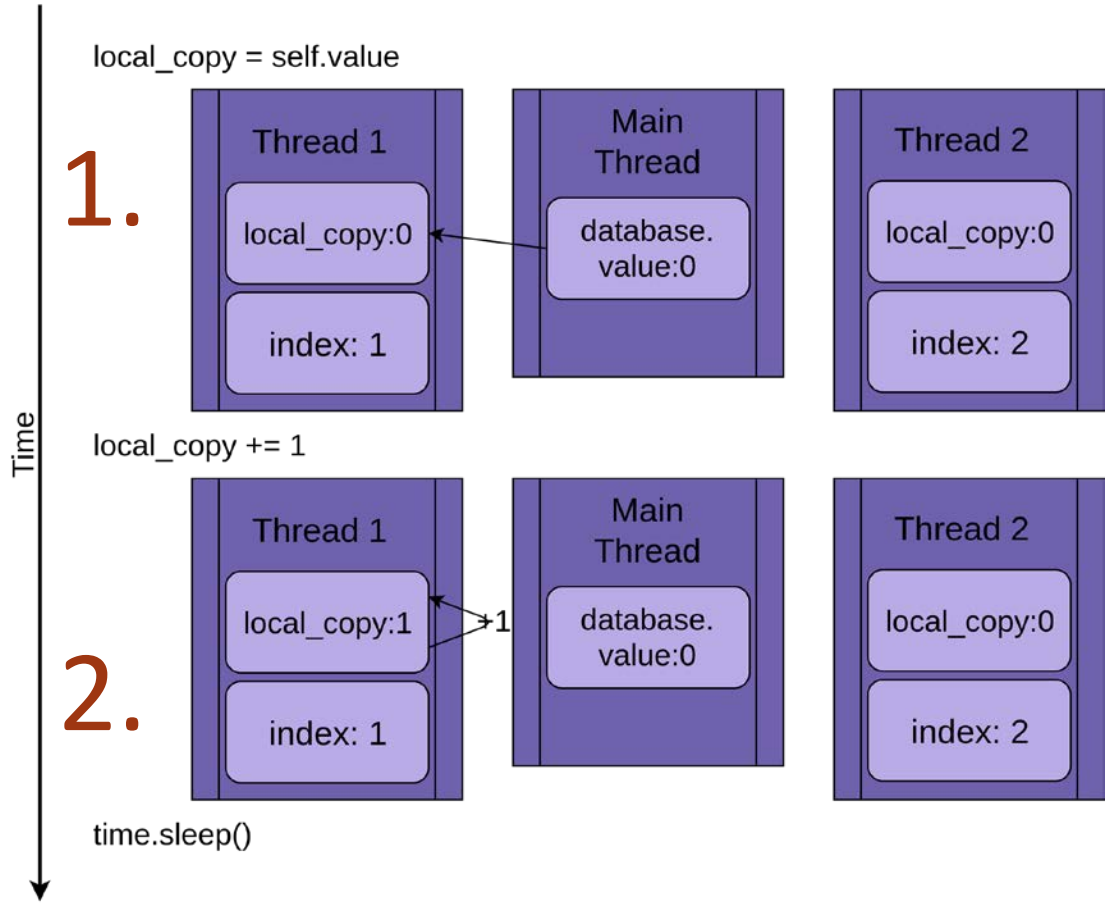
# Race Conditions – Two Threads, Step 1

We probably expect value to be 2 with 2 threads.

Let's see why that isn't the case with two threads with a race condition

1. Thread one access value and stores it in **`local_copy`**.

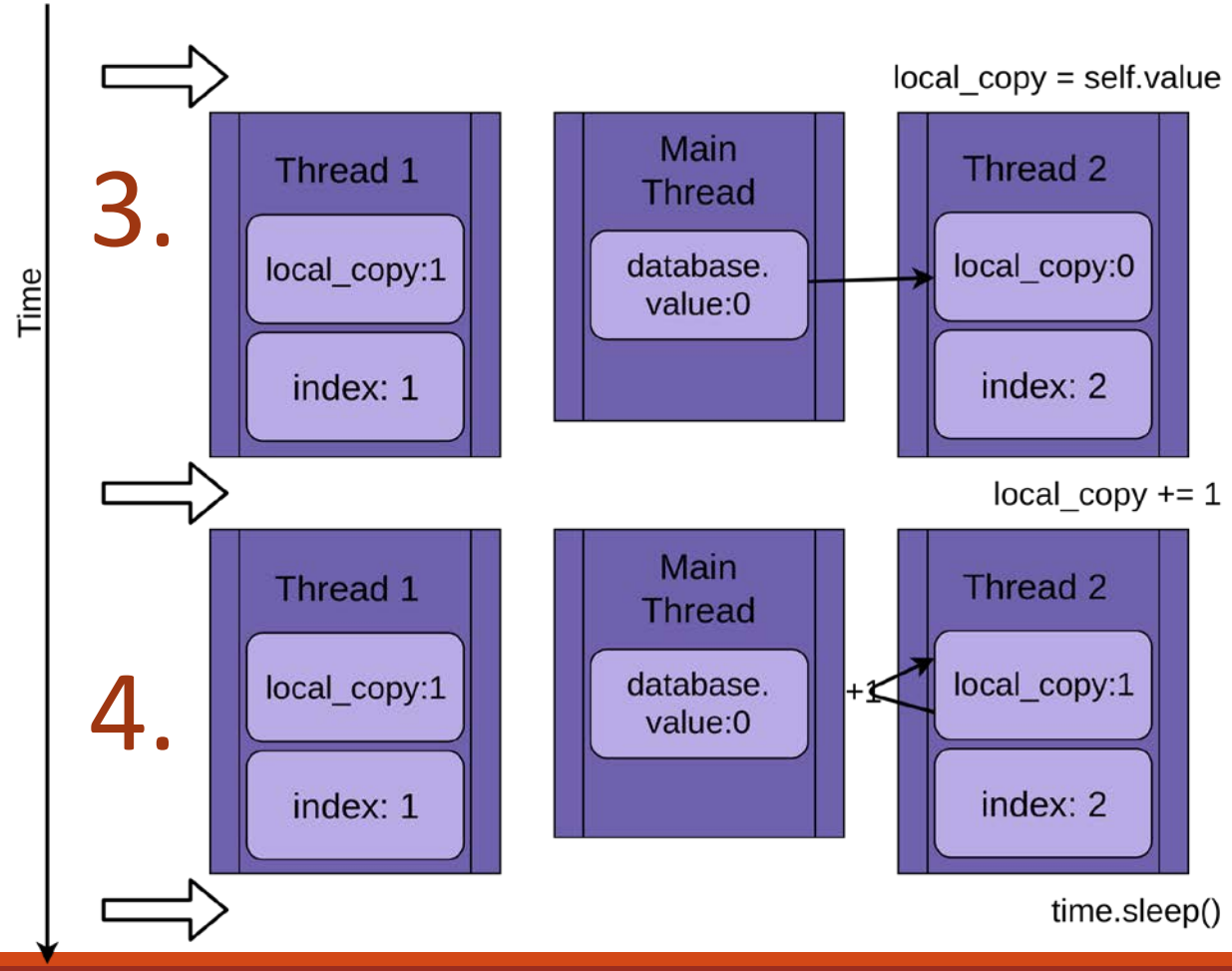
2. Thread one increments its copy of **`local_copy`** and goes to sleep



# Race Conditions – Two Threads, Step 2

3. Thread two access value and stores it in **`local_copy`**.

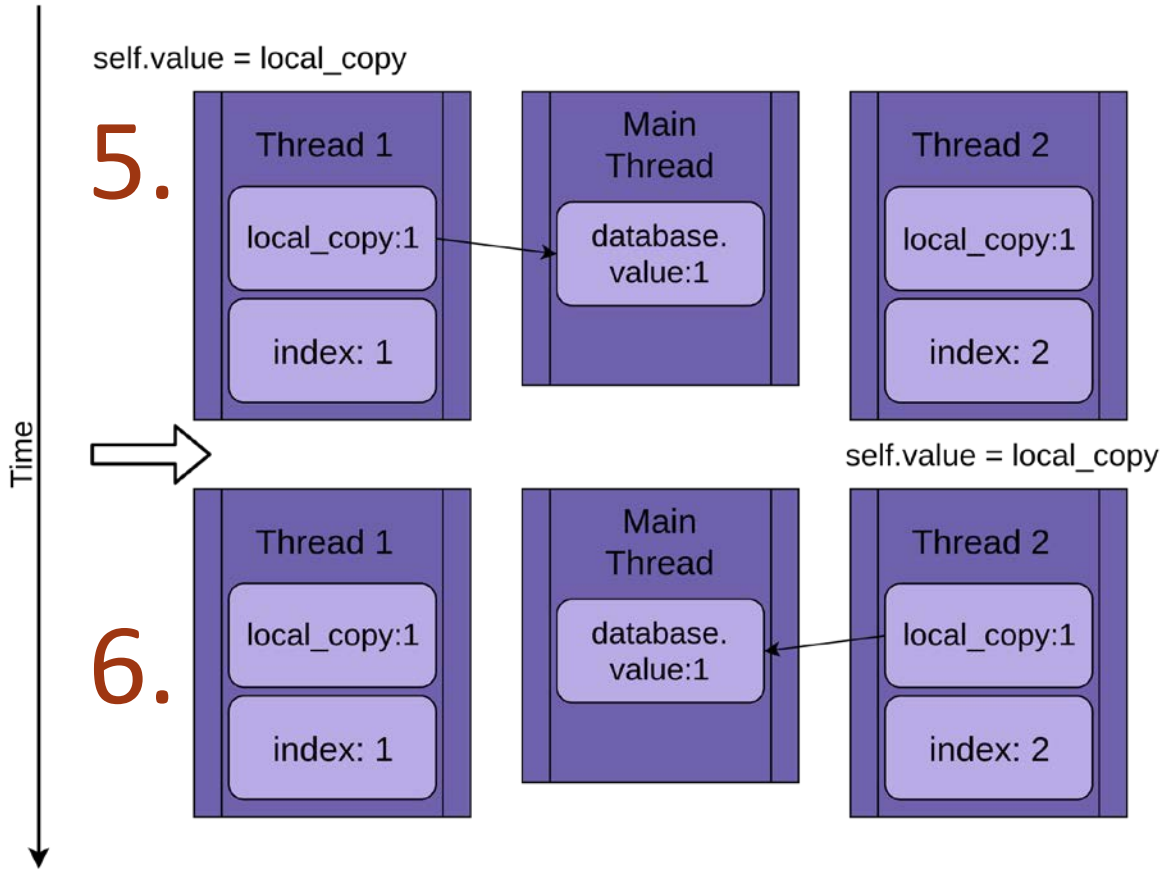
4. Thread two increments its copy of **`local_copy`** and goes to sleep.



# Race Conditions – Two Threads, Step 3

5. Thread one wakes up and overrides the **`value`** attribute with the value stored in it's **`local_copy`**.

6. Thread two wakes up and overrides the **`value`** attribute with the value stored in it's **`local_copy`**.



# Controlling access to shared resources.

These requirements must be met for thread safe access:

- **Mutual Exclusion of Critical Section**
  Only one thread is allowed to execute the code that accesses the shared data (known as the **"critical section"**) at a time.
- **Local copy of variables.**
  Each thread must maintain its own copy of local variables where it stores partly calculated results.
- **Atomic instructions**
  Shared data should be accessed using atomic operations which cannot be interrupted by other threads. Since the operations are atomic the shared data is always kept in a valid state.

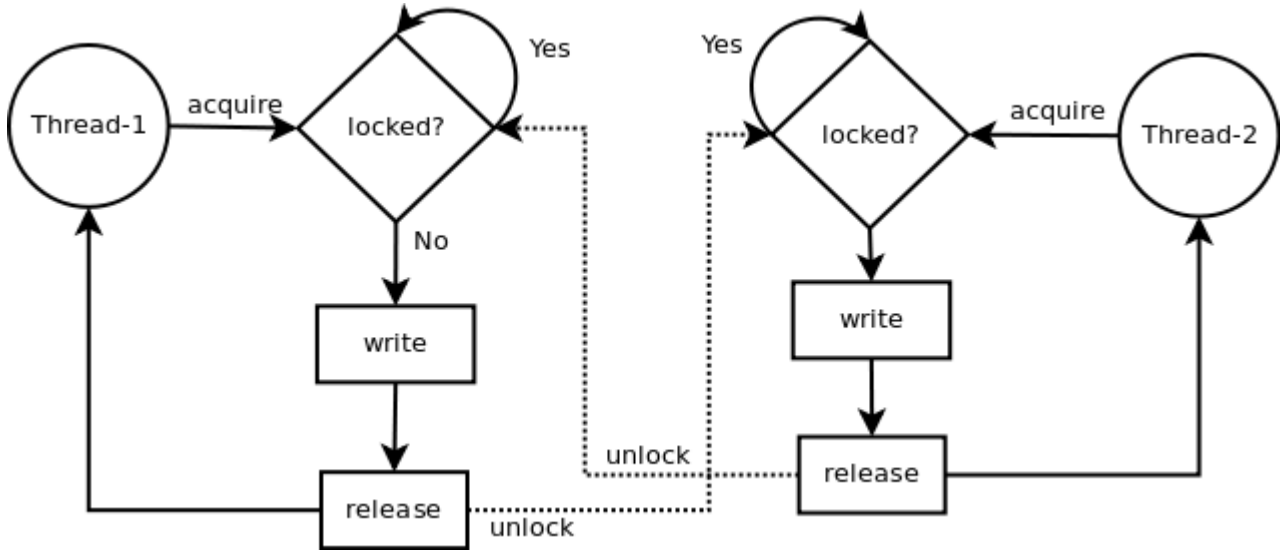
# Threads – Locks

Locks are a special object provided by the threading module. In other languages and in OS terminology it is known as a Mutex. A thread acquires a lock

A lock ensures that only one thread can access data at a time.

Going back to our data service example, locks can be used to control flow of execution like this:

- Thread 1 executes `save_data(dest_file)`
- Thread 1 **acquires lock** (lock was available since no one else was using it)/.
- Thread 1 sets dest_file to `"output_1.txt"`
- *Thread 2 executes `save_data(dest_file)`*
- *Thread 2 attempts to acquire a lock but must wait.*
- Thread 1 writes `"Thread 1 was here"`
- Thread 1 closes the file **and releases the lock**.
- *Thread 2 **acquires the lock.***
- *Thread 2 sets dest_file to `"output_2.txt"`*
- *Thread 2 writes `"Thread 2 was here"`*
- *Thread 2 closes the file **and releases the lock**.*



# Threads – Locks (and context managers)

The threading module has a Lock class.

Once you have created a lock object, you can acquire a lock by calling the **`acquire()`** method.

All locks should be released once they are no longer needed. This is done using the **`release()`** method.

Locks are context managers.

- That is they have the **`__enter__()`** and **`__exit__()`** dunders implemented.
- You can avoid calls to **`acquire()`** and **`release()`** by using the **`with`** statement.
- Files are context managers too.

thread_race_condition_fixed.py

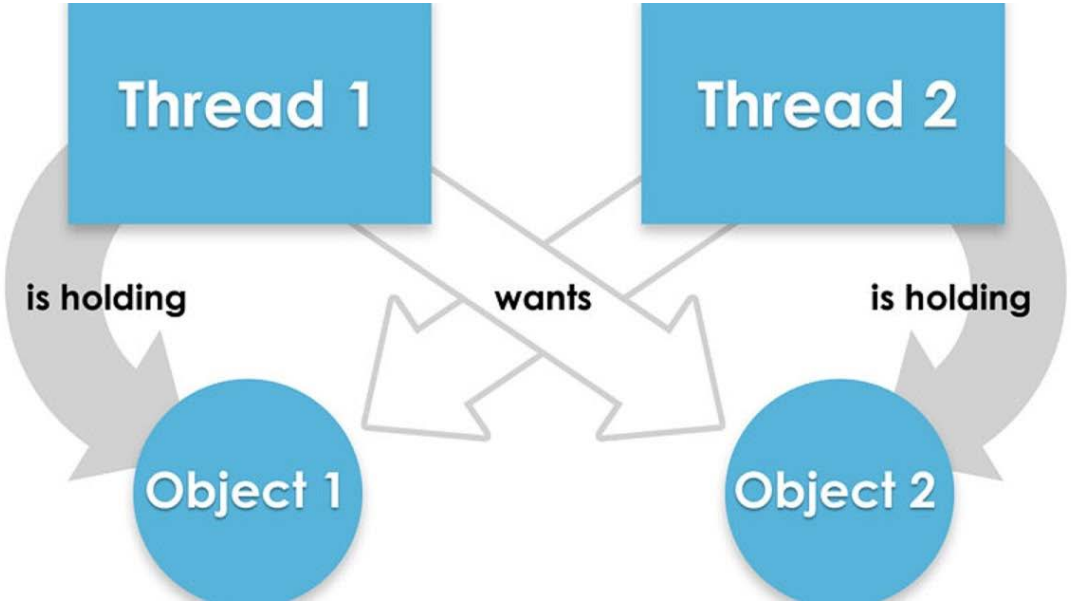
# Deadlock

A deadlock occurs when two or more threads are waiting on locks held by each other.

Neither can release their locks and are stuck in a limbo, waiting for each other.

This causes our code to become unresponsive and crash.

Deadlocks are bad.



# When to use concurrency

If there is no need to speed up our code we should avoid it. Concurrency is complex and hard to debug.

Remember, premature optimization is the enemy of good code. (It's a balancing act).

If there is a known performance issue, then determine what type of concurrency you need.

First Identify any CPU or I/O bound sections of your code, then add concurrency to them.

# Recap

That’s about the extent to which we will cover multi threading in python. Let’s do a quick recap

- Python provides support for concurrency via threads, tasks and processes.
- Threads and tasks provide the illusion of concurrency.
- ThreadExecutorPool hides the complexity of creating and executing multiple threads at a time.
- Threads are great for I/O bound operations, increase the speed and responsiveness of our program.
- Threads can be hard to manage. We need to be aware of race conditions and deadlocks.
- Use locks to access shared resources.

# That’s it for today!

This week’s lab is up on The Learning Hub

I’ll be on Slack during your regular lab times. Message me with any questions then