Using Threads and Processes in Pyton

About me

Trayan Iliev

– CEO of **IPT – Intellectual Products & Technologies**
http://www.iproduct.org

– Oracle® certified programmer 15+ Y

– end-to-end reactive fullstack apps with Java, ES6+, TypeScript, Angular, React and Vue.js

– 12+ years IT trainer: Spring, Java EE, Node.js, Express, GraphQL, SOA, REST, DDD & Reactive Microservices

– Voxxed Days, jPrime, Java2Days, jProfessionals, BGOUG, BGJUG, DEV.BG speaker

– Organizer RoboLearn hackathons and IoT enthusiast

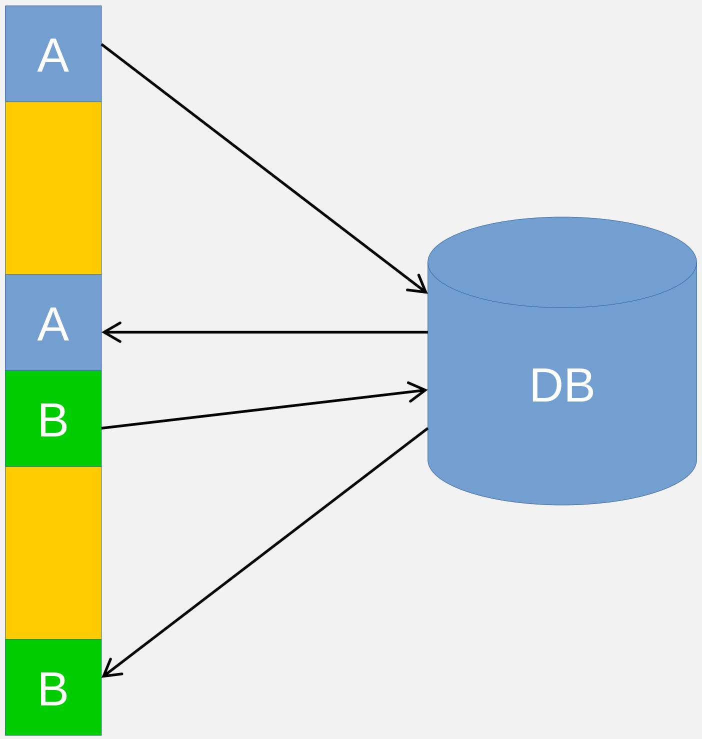
Where to Find The Code and Materials?

https://github.com/iproduct/intro-python

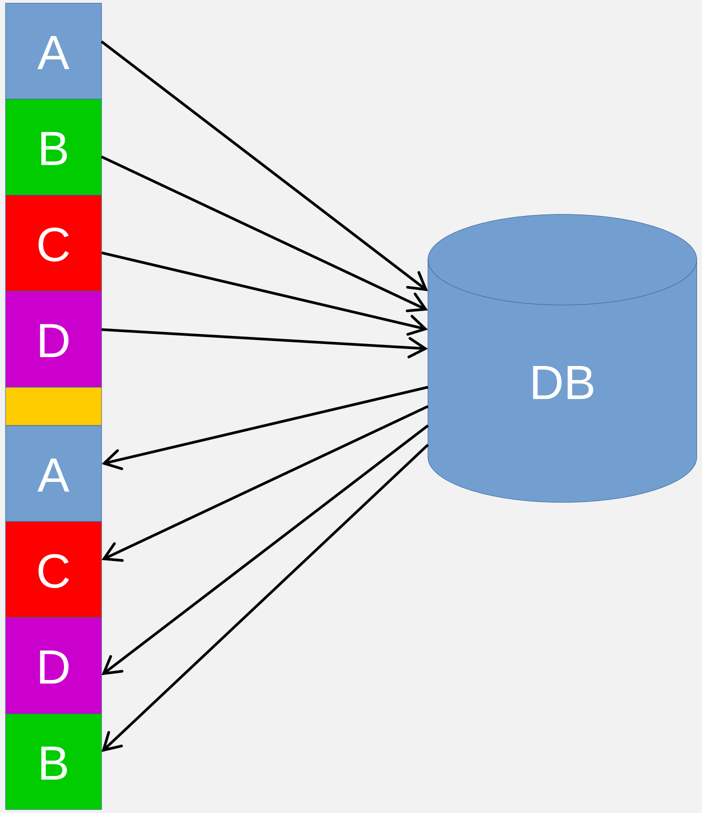
Thread. Processes. Concurrency. Parrallelism

Synchronous vs. Asynchronous IO

Synchronous



Asynchronous



5

Blocking vs. Non-blocking

- Blocking concurrency – uses **Mut**ual **Ex**clusion primitives (aka **Locks**) to prevent threads from simultaneously accessing/modifying the same resource

- Non-blocking concurrency does not make use of locks.

- One of the most advantageous feature of non-blocking vs. blocking is that, threads does not have to be suspended/waken up by the OS. Such overhead can amount to 1ms to a few 10ms, so removing this can be a big performance gain.

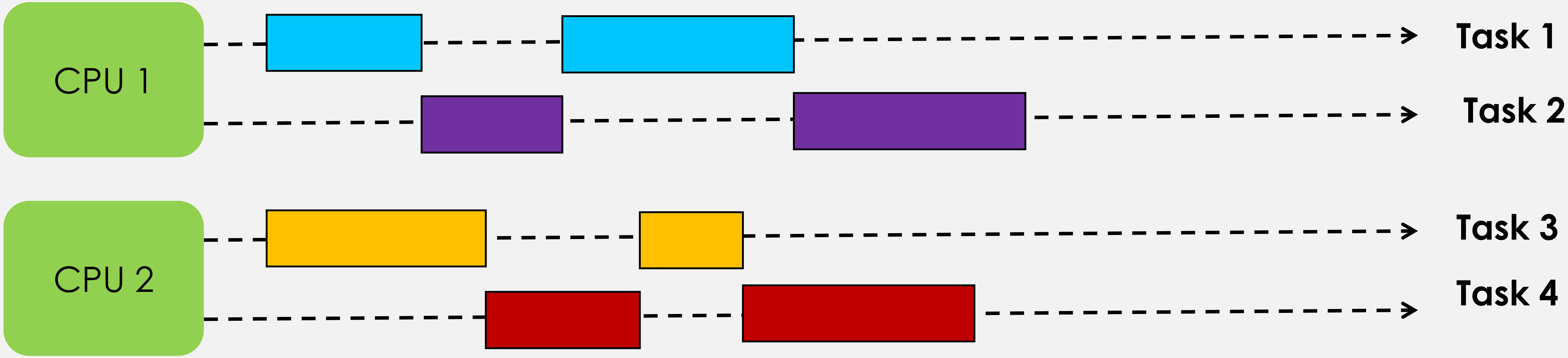
Non-blocking Concurrency

- In computer science, an algorithm is called **non-blocking** if failure or suspension of any thread cannot cause failure or suspension of another thread;[1] for some operations, these algorithms provide a useful alternative to traditional blocking implementations. A non-blocking algorithm is **lock-free** if there is guaranteed system-wide progress, and **wait-free** if there is also guaranteed per-thread progress. "Non-blocking" was used as a synonym for "lock-free" in the literature until the introduction of obstruction-freedom in 2003.
- It has been shown that widely available atomic *conditional* primitives, CAS and LL/SC, cannot provide starvation-free implementations of many common data structures without memory costs growing linearly in the number of threads.

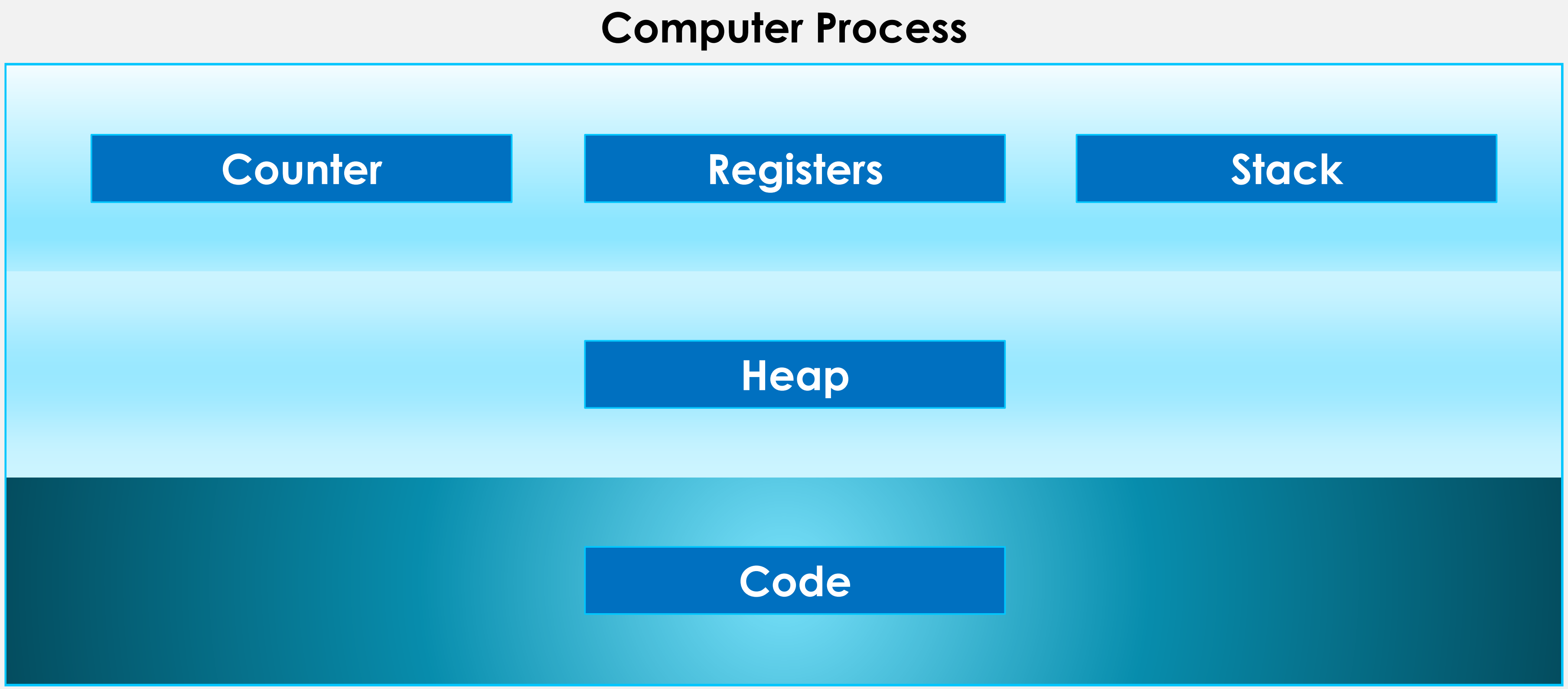
[Wikipedia]

Concurrency vs. Parallelism

- Concurrency refers to how a single CPU can make progress on multiple tasks seemingly at the same time (AKA concurrently).
- Parallelism allows an application to parallelize the execution of a single task - typically by splitting the task up into subtasks which can be completed in parallel.

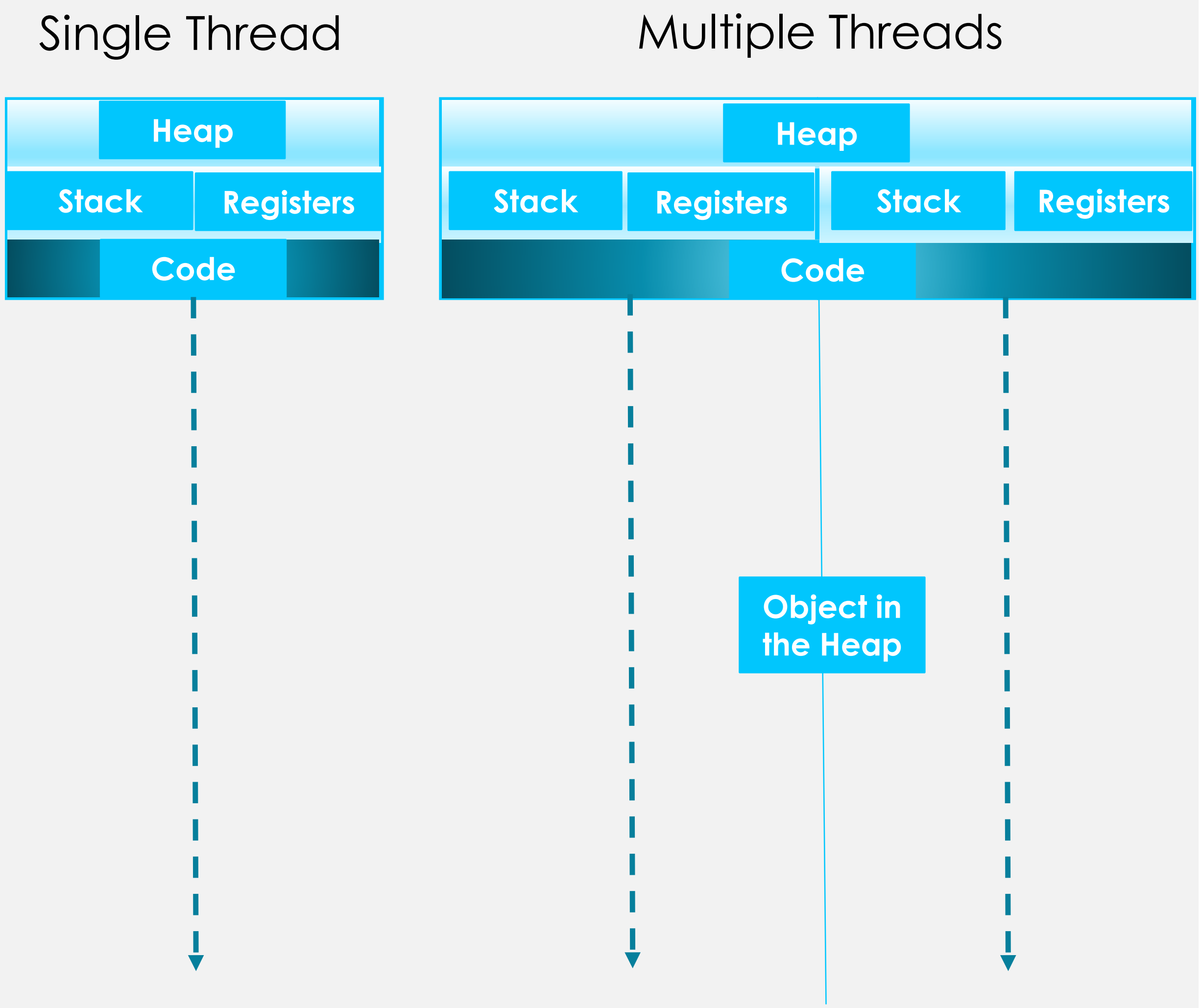


Concurrency Approaches: Processes



Threads

- There can be many threads in the same process
- The threads can access the shared memory
- This means that the global objects can be accessed by all threads
- Provided by the OS
- Cheaper to create than processes
- Some languages expose them directly other hide them behind a level of abstraction



But why should we bother: concurrency problems

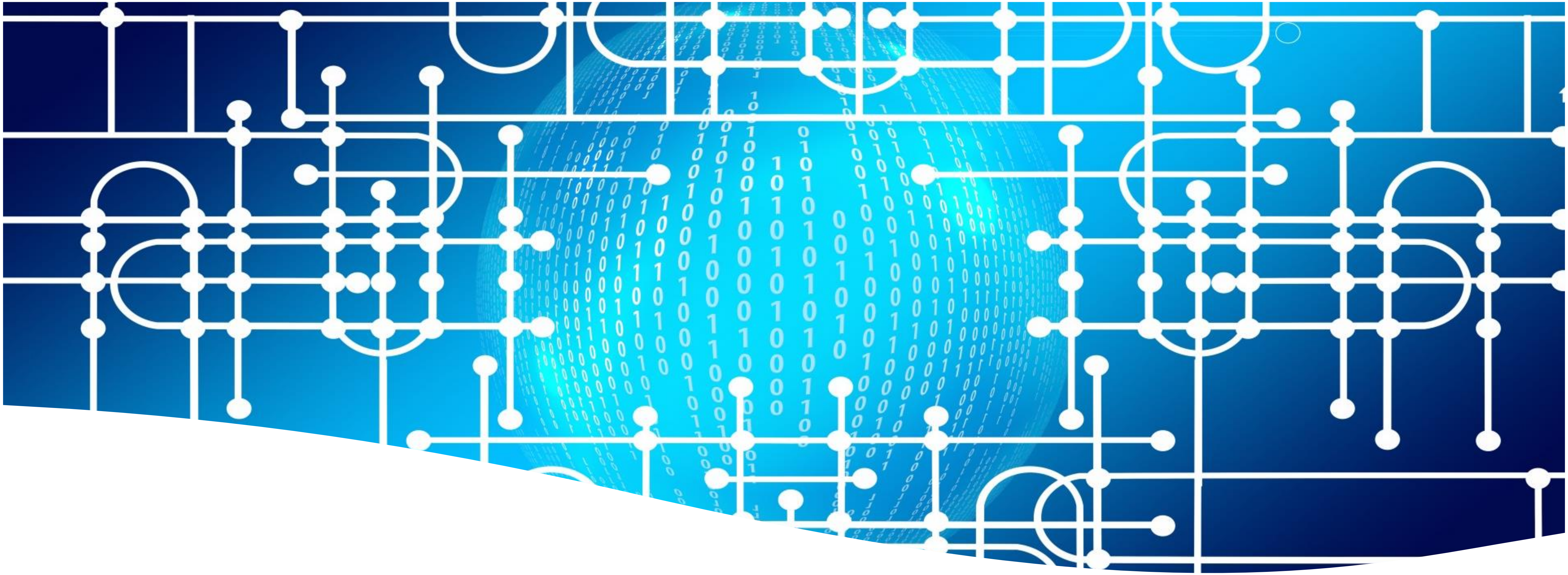
If we access the same memory from two threads/ goroutines, than we have a race! Lets see this pseudo-code:

```
int i = 0

thread1 { i++ }
thread2 { i++ }

wait { thread1 } { thread2 }
print i
```

What will be the result of the computation?

Critical Sections

- We provide Mutual Exclusion between different threads accessing the same resource concurrently
- There are many ways to implement Mutual Exclusion
- In Python there are threading.Lock, threading.RLock, atomic operations, semaphores, barriers etc.



Threads in Pyton

Basic Threads

```
import logging
import threading
import time

def thread_function(name):
logging.info("Thread %s: starting", name)
time.sleep(2)
logging.info("Thread %s: finishing [ID: %s]", name)

if __name__ == "__main__":
    format = "%(asctime)s: %(message)s"
    logging.basicConfig(format=format, level=logging.INFO, datefmt="%H:%M:%S")
    logging.info("Main    : before creating thread")
    x = threading.Thread(target=thread_function, args=(1,))
    logging.info("Main    : before running thread")
    x.start()
    logging.info("Main    : wait for the thread to finish")
    # x.join()
    logging.info("Main    : all done")
```

```
21:52:16: Main    : before creating thread

21:52:16: Main    : before running thread

21:52:16: Thread 1: starting, [ID: 17916]

21:52:16: Main    : wait for the thread to finish

21:52:16: Main    : all done

21:52:18: Thread 1: finishing [ID: 17916]
```

Daemon Threads

```python
import logging
import threading
import time

def thread_function(name):
logging.info("Thread %s: starting", name)
time.sleep(2)
logging.info("Thread %s: finishing", name)

if __name__ == "__main__":
    format = "%(asctime)s: %(message)s"
    logging.basicConfig(format=format, level=logging.INFO, datefmt="%H:%M:%S")
    logging.info("Main    : before creating thread")
    x = threading.Thread(target=thread_function, args=(1,), daemon=True)

    logging.info("Main    : before running thread")
    x.start()
    logging.info("Main    : wait for the thread to finish")
    logging.info("Main    : all done")
```

```
21:48:57: Main    : before creating thread

21:48:57: Main    : before running thread

21:48:57: Thread 1: starting, [ID: 12704]

21:48:57: Main    : wait for the thread to finish

21:48:57: Main    : all done
```

Working With Many Threads

```
import logging
import threading
import time

def thread_function(name):
logging.info("Thread %s: starting", name)
time.sleep(2)
logging.info("Thread %s: finishing", name)

if __name__ == "__main__":
    format = "%(asctime)s: %(message)s"
    logging.basicConfig(format=format, level=logging.INFO, datefmt="%H:%M:%S")
    logging.info("Main    : before creating thread")
    x = threading.Thread(target=thread_function, args=(1,), daemon=True)

    logging.info("Main    : before running thread")
    x.start()
    logging.info("Main    : wait for the thread to finish")
    logging.info("Main    : all done")
```

```
21:48:57: Main    : before creating thread
21:48:57: Main    : before running thread
21:48:57: Thread 1: starting, [ID: 12704]
21:48:57: Main    : wait for the thread to finish
21:48:57: Main    : all done
```

Using a ThreadPoolExecutor

```
from concurrent.futures import ThreadPoolExecutor

if __name__ == "__main__":
    with ThreadPoolExecutor(max_workers=1) as executor:
        future = executor.submit(pow, 323, 1235)
        print(future.result())
```

Using a ThreadPoolExecutor

```
import logging
import threading
import time
import concurrent.futures

def thread_function(name):
    logging.info("Thread %s: starting, [ID: %s]", name, threading.get_ident())
    time.sleep(2)
    logging.info("Thread %s: finishing [ID: %s]", name, threading.get_ident())
    return f"Result {name}"

if __name__ == "__main__":
    format = "%(asctime)s: %(message)s"
    logging.basicConfig(format=format, level=logging.INFO, datefmt="%H:%M:%S")

    with concurrent.futures.ThreadPoolExecutor(max_workers=3) as executor:
        results = executor.map(thread_function, range(3))

    for index, result in enumerate(results):
        logging.info("Main    : thread %d done with result: %s", index, result)
```

```
22:23:24: Thread 0: starting, [ID: 25912]
22:23:24: Thread 1: starting, [ID: 20784]
22:23:24: Thread 2: starting, [ID: 30208]
22:23:26: Thread 2: finishing [ID: 30208]
22:23:26: Thread 0: finishing [ID: 25912]
22:23:26: Thread 1: finishing [ID: 20784]
22:23:26: Main    : thread 0 done with result: Result 0
22:23:26: Main    : thread 1 done with result: Result 1
22:23:26: Main    : thread 2 done with result: Result 2
```

Futures [https://docs.python.org/3.10/library/concurrent.futures.html#future-objects]

- The concurrent.futures.Future class encapsulates the asynchronous execution of a callable. Future instances are created by Executor.submit().
- cancel() - attempt to cancel the call. If the call is currently being executed or finished running and cannot be cancelled then the method will return False, otherwise the call will be cancelled and the method will return True.
- cancelled() - returns True if the call was successfully cancelled.
- running() - returns True if the call is currently being executed and cannot be cancelled.
- done() - returns True if the call was successfully cancelled or finished running.

Futures [https://docs.python.org/3.10/library/concurrent.futures.html#future-objects]

- result(timeout=None) - returns the value returned by the call. If the call hasn’t yet completed then this method will wait up to timeout seconds. If the call hasn’t completed in timeout seconds, then a concurrent.futures.TimeoutError will be raised. timeout can be an int or float. If timeout is not specified or None, there is no limit to the wait time. If the future is cancelled before completing then CancelledError will be raised. If the call raised an exception, this method will raise the same exception.

- exception(timeout=None) - returns the exception raised by the call. If the call hasn’t yet completed then this method will wait up to timeout seconds. If the call hasn’t completed in timeout seconds, a concurrent.futures.TimeoutError will be raised. timeout can be an int or float. If timeout is not specified or None, there is no limit to the wait time. If the call completed without raising, None is returned.

Futures [https://docs.python.org/3.10/library/concurrent.futures.html#future-objects]

- add_done_callback(fn) - attaches the callable fn to the future. fn will be called, with the future as its only argument, when the future is cancelled or finishes running.
  - Added callables are called in the order that they were added and are always called in a thread belonging to the process that added them. If the callable raises an Exception subclass, it will be logged and ignored. If the callable raises a BaseException subclass, the behavior is undefined.
  - If the future has already completed or been cancelled, fn will be called immediately.

Race Conditions

```
class FakeDatabase:
    def __init__(self):
        self.value = 0
    def update(self, name):
        logging.info("Thread %s: starting update", name)
        local_copy = self.value
        local_copy += 1
        time.sleep(0.1)
        self.value = local_copy
        logging.info("Thread %s: finishing update", name)

if __name__ == "__main__":
    format = "%(asctime)s: %(message)s"
    logging.basicConfig(format=format, level=logging.INFO, datefmt="%H:%M:%S")
    database = FakeDatabase()
    logging.info("Testing update. Starting value is %d.", database.value)
    with concurrent.futures.ThreadPoolExecutor(max_workers=2) as executor:
        for index in range(2):
            executor.submit(database.update, index)
    logging.info("Testing update. Ending value is %d.", database.value)
```

```
08:32:00: Testing update. Starting value is 0.
08:32:00: Thread 0: starting update
08:32:00: Thread 1: starting update
08:32:00: Thread 1: finishing update
08:32:00: Thread 0: finishing update
08:32:00: Testing update. Ending value is 1.
```

Basic Synchronization Using Lock

```
class FakeDatabase:
    def __init__(self):
        self.value = 0
        self._lock = threading.Lock()

    def locked_update(self, name):
        logging.info("Thread %s: starting update", name)
        logging.debug("Thread %s about to lock", name)
        with self._lock:
            logging.debug("Thread %s has lock", name)
            local_copy = self.value
            local_copy += 1
            time.sleep(0.1)
            self.value = local_copy
            logging.debug("Thread %s about to release lock", name)
        logging.debug("Thread %s after release", name)
        logging.info("Thread %s: finishing update", name)
```

```
08:36:34: Testing update. Starting value is 0.

08:36:34: Thread 0: starting update

08:36:34: Thread 1: starting update

08:36:34: Thread 0: finishing update

08:36:34: Thread 1: finishing update

08:36:34: Testing update. Ending value is 2.
```

Deadlock

```
import threading

if __name__ == "__main__":
    l = threading.Lock()
    print("before first acquire")
    l.acquire()
    print("before second acquire")
    l.acquire()
    print("acquired lock twice")
```

```
before first acquire

before second acquire
```

Deadlock

```
import threading

if __name__ == "__main__":
    l = threading.Lock()
    print("before first acquire")
    l.acquire()
    print("before second acquire")
    l.acquire()
    print("acquired lock twice")
```

```
before first acquire

before second acquire
```

Example reasons for deadlock:

- An implementation bug where a Lock is not released properly - it is recommended to write code whenever possible to make use of context managers, as they help to avoid situations where an exception skips you over the .release() call.
- A design issue where a utility function needs to be called by functions that might or might not already have the Lock – use RLock, that is designed for just this situation. It allows a thread to .acquire() an RLock multiple times before it calls .release(). That thread is still required to call .release() the same number of times it called .acquire(), but it should be doing that anyway.

Producer-Consumer I

[https://en.wikipedia.org/wiki/Producer%E2%80%93consumer_problem]

```
def producer(pipeline, event):
    """Pretend we're getting a number from the network."""
    while not event.is_set():
        message = random.randint(1, 101)
        logging.info("Producer got message: %s", message)
        pipeline.set_message(message, "Producer")

    logging.info("Producer received EXIT event. Exiting")

def consumer(pipeline, event):
    """Pretend we're saving a number in the database."""
    while not event.is_set() or not pipeline.empty():
        message = pipeline.get_message("Consumer")
        logging.info(
            "Consumer storing message: %s  (queue size=%s)",
            message,
            pipeline.qsize(),
        )
    logging.info("Consumer received EXIT event. Exiting")
```

Producer-Consumer II

[https://en.wikipedia.org/wiki/Producer%E2%80%93consumer_problem]

```
class Pipeline(queue.Queue):
    def __init__(self):
        super().__init__(maxsize=10)

    def get_message(self, name):
        logging.debug("%s:about to get from queue", name)
        value = self.get()
        logging.debug("%s:got %d from queue", name, value)
        return value

    def set_message(self, value, name):
        logging.debug("%s:about to add %d to queue", name, value)
        self.put(value)
        logging.debug("%s:added %d to queue", name, value)
```

Other Threading Objects

- Semaphore - semaphore manages an internal counter which is decremented by each acquire() call and incremented by each release() call. The counter can never go below zero; when acquire() finds that it is zero, it blocks, waiting until some other thread calls release(). Semaphores support the context management protocol.

- Timer - This class represents an action that should be run only after a certain amount of time has passed — a timer. Timer is a subclass of Thread and as such also functions as an example of creating custom threads.

- Barrier - This class provides a simple synchronization primitive for use by a fixed number of threads that need to wait for each other. Each of the threads tries to pass the barrier by calling the wait() method and will block until all of the threads have made their wait() calls. At this point, the threads are released simultaneously.

Problem: Text Search in Multiple Files using Threads

If you have:

1) directory
2) file extension (e.g. txt)
3) text to search (can be regular expression),

provided to the program as command line arguments,

Find all occurrences of searched text (or regex), in the files with given extension, in provided directory recursively.

The program should print out the names of the files, and the exact locations of the matches found (line and column in text file).

The search should be implemented using multiple threads / processes for better performance.

Thank's for Your Attention!

Trayan Iliev

IPT – Intellectual Products & Technologies

http://iproduct.org/

https://github.com/iproduct

https://twitter.com/trayaniliev

https://www.facebook.com/IPT.EACAD