Programming technologies

Sync, Async, Concurrency, Parallelism

Definitions

Sync vs Async

In Syncrhonous operations, the tasks are executed in sync, one after one. In asynchronous operations, tasks may start and complete independent of each other. One async task may start and continue running while the execution moves on to a new task. Async tasks don’t block (make the execution wait for it’s completion) operations and usually run in the background.

Concurrency and Parallelism

Concurrency implies that two tasks make progress together. Parallelism is in fact a form of concurrency. But parallelism is hardware dependent. For example if there’s only one core in the CPU, two operations can’t really run in parallel. They just share time slices from the same core. This is concurrency but not parallelism. But when we have multiple cores, we can actually run two or more operations (depending on the number of cores) in parallel.

Synchronous execution

There is actually nothing special in synchronous execution. It just means we run the code as usual: one thing happens and then another thing happens. Only one function can run at a time and only when it is finished, something else is allowed to happen.

```
import time

def step_1():
    print("Start of step 1")
    time.sleep(3)
    print("End of step 1")

def step_2():
    print("Start of step 2")
    time.sleep(5)
    print("End of step 2")

def step_3():
    print("Start of step 3")
    time.sleep(1)
    print("End of step 3")

def main():
    step_1()
    step_2()
    step_3()

if __name__ == "__main__":
    start = time.time()
    main()
    print(f"We spent {time.time()-start} s")
```

Asynchronous execution (async)

In async we run one block of code at a time but we cycle which block of code is running. Your program needs to be built around async though you can call normal (synchronous) functions from async program.

Here is a list of what you need in order to make your program async:

- Add ***async*** keyword in front of your function declarations to make them awaitable.
- Add ***await*** keyword when you call your async functions (without it they won’t run).
- Create tasks from the async functions you wish to start asynchronously. Also wait for their finish.
- Call ***asyncio.run*** to start the asynchronous section of your program.

Asynchronous execution (async)

As we can see, we run the first block of code in the *step_1* function, then we give the execution back to the async engine which then ran the first block of code in the *step_2* function (while the *step_1* function was unfinished), then we give the execution back to the async engine which then ran the first block of code in the *step_3* function (while the *step_1* and *step_2* functions were unfinished), then we again released the execution and then the final blocks of codes from each functions were run.

```python
import time
import asyncio

async def step_1():
    print("Start of step 1")
    await asyncio.sleep(3)
    print("End of step 1")

async def step_2():
    print("Start of step 2")
    await asyncio.sleep(5)
    print("End of step 2")

async def step_3():
    print("Start of step 3")
    await asyncio.sleep(1)
    print("End of step 3")

async def main():
    task_1 = asyncio.create_task(step_1())
    task_2 = asyncio.create_task(step_2())
    task_3 = asyncio.create_task(step_3())
    await asyncio.wait([task_1, task_2, task_3])

if __name__ == "__main__":
    start = time.time()
    asyncio.run(main())
    print(f"We spent {time.time()-start} s")
```

Concurrent execution (threading)

In threading, we execute one line of code at a time but we constantly change which line is run. This is done using threading library: we first create some threads, start them and then wait them to finish (using join, for example).

Threading

```
import time
import threading

def step_1():
    print("Start of step 1")
    time.sleep(3)
    print("End of step 1")

def step_2():
    print("Start of step 2")
    time.sleep(5)
    print("End of step 2")

def step_3():
    print("Start of step 3")
    time.sleep(1)
    print("End of step 3")

def main():
    t1 = threading.Thread(target=step_1)
    t2 = threading.Thread(target=step_2)
    t3 = threading.Thread(target=step_3)

    t1.start(), t2.start(), t3.start()

    t1.join(), t2.join(), t3.join()

if __name__ == "__main__":
    start = time.time()
    main()
    print(f"We spent {time.time()-start} s")
```

Parallel execution (multiprocessing)

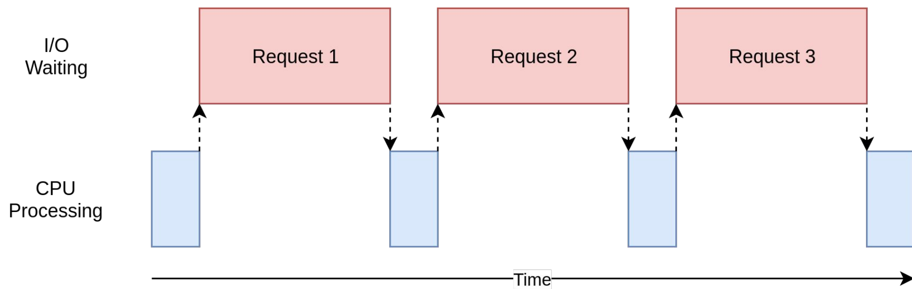
In multiprocessing we actually run multiple lines of Python code at one time. We use multiple processes to achieve this. In order to use multiprocessing, you need to: create processes, set them running and wait for them to finish (using join, for example).

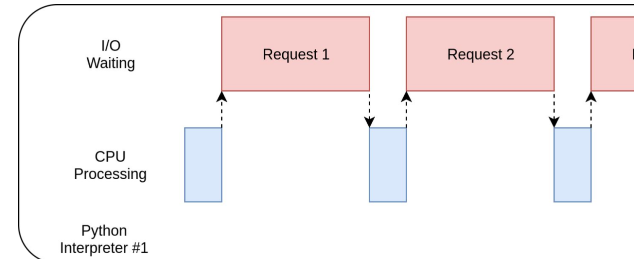
Multiprocessing

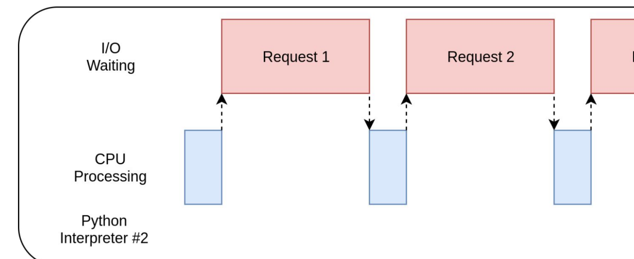
```python
import time
import multiprocessing

def step_1():
    print("Start of step 1")
    time.sleep(3)
    print("End of step 1")

def step_2():
    print("Start of step 2")
    time.sleep(5)
    print("End of step 2")

def step_3():
    print("Start of step 3")
    time.sleep(1)
    print("End of step 3")

def main():
    t1 = multiprocessing.Process(target=step_1)
    t2 = multiprocessing.Process(target=step_2)
    t3 = multiprocessing.Process(target=step_3)

    t1.start(), t2.start(), t3.start()
    t1.join(), t2.join(), t3.join()

if __name__ == "__main__":
    start = time.time()
    main()
    print(f"We spent {time.time()-start} s")
```

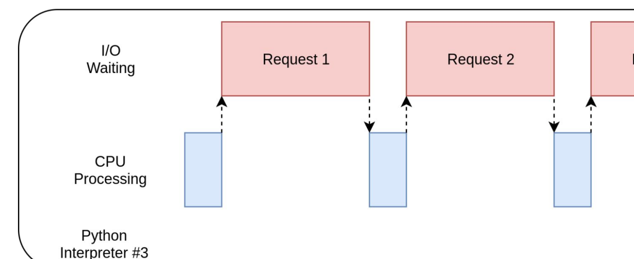
Global Interpreter Lock (GIL)

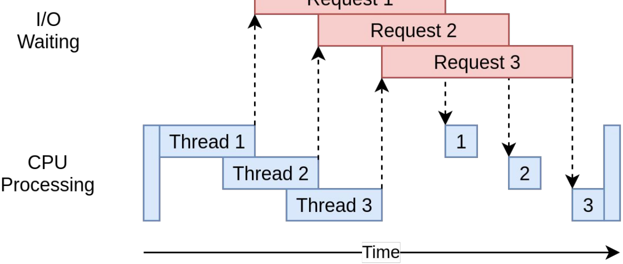
The Global Interpreter Lock aka GIL was introduced to make CPython’s memory handling easier and to allow better integrations with C (for example the extensions). The GIL is a locking mechanism that the Python interpreter runs only one thread at a time. That is only one thread can execute Python byte code at any given time. This GIL makes sure that multiple threads DO NOT run in parallel.

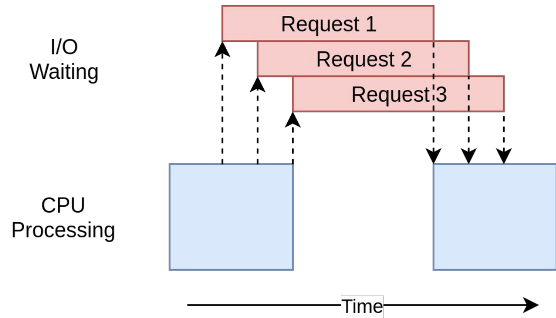

I/O
Waiting
CPU
Processing
Request 1
Request 2
Request 3
Time


I/O
Waiting
CPU
Processing
Python
Interpreter #1
Request 1
Request 2


I/O
Waiting
CPU
Processing
Python
Interpreter #2
Request 1
Request 2


I/O
Waiting
CPU
Processing
Python
Interpreter #3
Request 1
Request 2
Time


I/O
Waiting
CPU
Processing
Request 1
Request 2
Request 3
Thread 1
Thread 2
Thread 3
1
2
3
Time


I/O
Waiting
CPU
Processing
Request 1
Request 2
Request 3
Time

Global Interpreter Lock (GIL)

Quick facts about the GIL:

- One thread can run at a time.
- The Python Interpreter switches between threads to allow concurrency.
- The GIL is only applicable to CPython (the defacto implementation). Other implementations like Jython, IronPython don’t have GIL.
- GIL makes single threaded programs fast.
- For I/O bound operations, GIL usually doesn’t harm much.
- GIL makes it easy to integrate non thread safe C libraries, thansk to the GIL, we have many high performance extensions/modules written in C.
- For CPU bound tasks, the interpreter checks between N ticks and switches threads. So one thread does not block others.
- Many people see the GIL as a weakness. I see it as a blessing since it has made libraries like NumPy, SciPy possible which have taken Python an unique position in the scientific communities.

Multithreading concept

Running several threads is similar to running several different programs concurrently, but with the following benefits:

- Multiple threads within a process share the same data space with the main thread and can therefore share information or communicate with each other more easily than if they were separate processes.
- Threads sometimes called light-weight processes and they do not require much memory overhead; they are cheaper than processes.

Multithreading is a way of achieving multitasking.

Thread VS Process

In computing, a **process** is an instance of a computer program that is being executed. Any process has 3 basic components:

- An executable program.
- The associated data needed by the program (variables, work space, buffers, etc.)
- The execution context of the program (State of process)

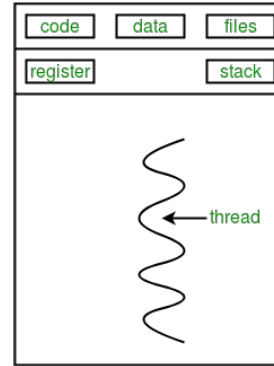
A **thread** is an entity within a process that can be scheduled for execution. Also, it is the smallest unit of processing that can be performed in an OS (Operating System). In simple words, a **thread** is a sequence of such instructions within a program that can be executed independently of other code.

Singlethreading VS Multithreading

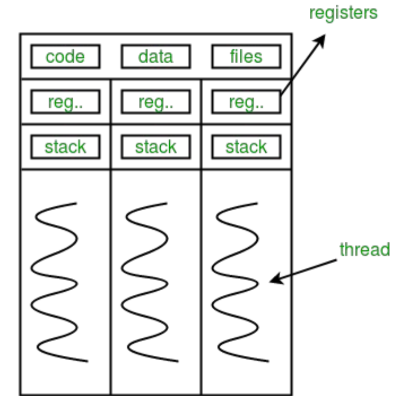
Multithreading: Multiple threads can exist within one process where:

- Each thread contains its own register set and local variables (stored in stack).
- All threads of a process share global variables (stored in heap) and the program code.

Multithreading is defined as the ability of a processor to execute multiple threads concurrently.



single-threaded process



multithreaded process

Multithreading in Python

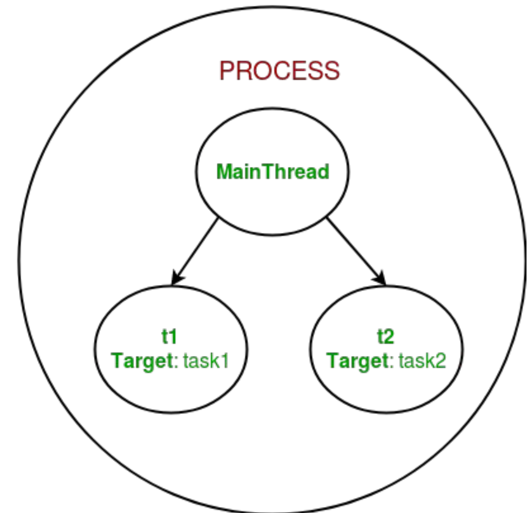
```python
import threading
def print_cube (num):
    # function to print cube of given num
    print ("Cube: {}" .format (num * num * num))
def print_square (num):
    # function to print square of given num
    print ("Square: {}" .format (num * num))
if __name__ =="__main__":
    # creating thread
    t1 = threading.Thread(target=print_square, args=( 10,))
    t2 = threading.Thread(target=print_cube, args=( 10,))
    # starting thread 1
    t1.start()
    # starting thread 2
    t2.start()
    # wait until thread 1 is completely executed
    t1.join()
    # wait until thread 2 is completely executed
    t2.join()
    # both threads completely executed
    print ("Done!")
```

Multithreading in Python

```
import threading
import os
def task1():
    print("Task 1 assigned to thread: {}" .format(threading.current_thread().name))
    print("ID of process running task 1: {}" .format(os.getpid()))
def task2():
    print("Task 2 assigned to thread: {}" .format(threading.current_thread().name))
    print("ID of process running task 2: {}" .format(os.getpid()))
if __name__ == "__main__":
    # print ID of current process
    print("ID of process running main program: {}" .format(os.getpid()))
    # print name of main thread
    print("Main thread name: {}" .format(threading.current_thread().name))
    # creating threads
    t1 = threading.Thread(target=task1, name= 't1')
    t2 = threading.Thread(target=task2, name= 't2')
    # starting threads
    t1.start()
    t2.start()
    # wait until all threads finish
    t1.join()
    t2.join()
```

threading.current_thread()

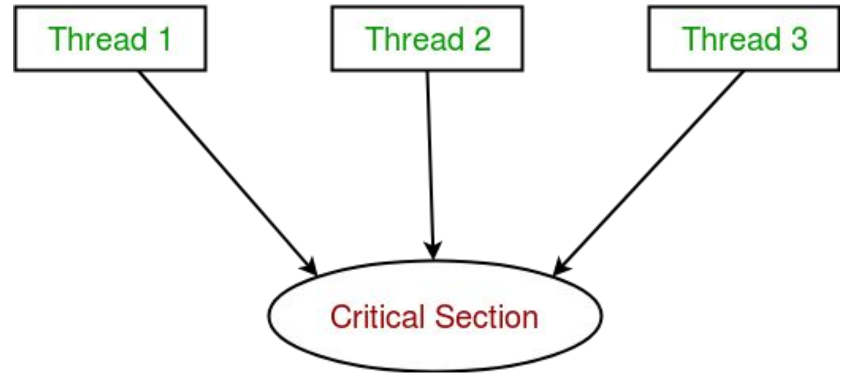
threading.main_thread()



Synchronization

Multithreading can help you make your programs more efficient and responsive. However, it's important to be careful when working with threads to avoid issues such as race conditions and deadlocks.

Thread synchronization is defined as a mechanism which ensures that two or more concurrent threads do not simultaneously execute some particular program segment known as critical section.

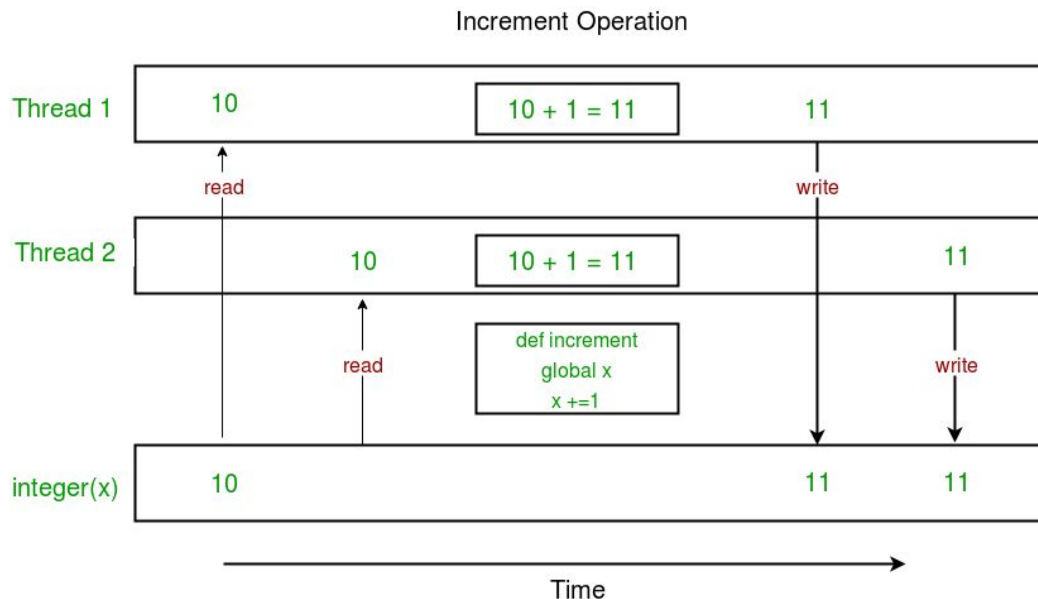


Concurrent accesses to shared resource can lead to **race condition**.

A **race condition** occurs when two or more threads can access shared data and they try to change it at the same time. As a result, the values of variables may be unpredictable and vary depending on the timings of context switches of the processes.

Race condition example

```
import threading
# global variable x
x = 0
def increment():
    """
    function to increment global variable x
    """
    global x
    x += 1
def thread_task():
    """
    task for thread
    calls increment function 100000 times.
    """
    for _ in range(100000):
        increment()
def main_task():
    global x
    # setting global variable x as 0
    x = 0
    # creating threads
    t1 = threading.Thread(target=thread_task)
    t2 = threading.Thread(target=thread_task)
    # start threads
    t1.start()
    t2.start()
    # wait until threads finish their job
    t1.join()
    t2.join()
if __name__ == "__main__":
    for i in range(10):
        main_task()
        print("Iteration {0}: x = {1}".format(i,x))
```



Using Locks

threading module provides a Lock class to deal with the race conditions. Lock is implemented using a Semaphore object provided by the Operating System.

A **semaphore** is a synchronization object that controls access by multiple processes/threads to a common resource in a parallel programming environment. It is simply a value in a designated place in operating system (or kernel) storage that each process/thread can check and then change. Depending on the value that is found, the process/thread can use the resource or will find that it is already in use and must wait for some period before trying again. Semaphores can be binary (0 or 1) or can have additional values. Typically, a process/thread using semaphores checks the value and then, if it using the resource, changes the value to reflect this so that subsequent semaphore users will know to wait.

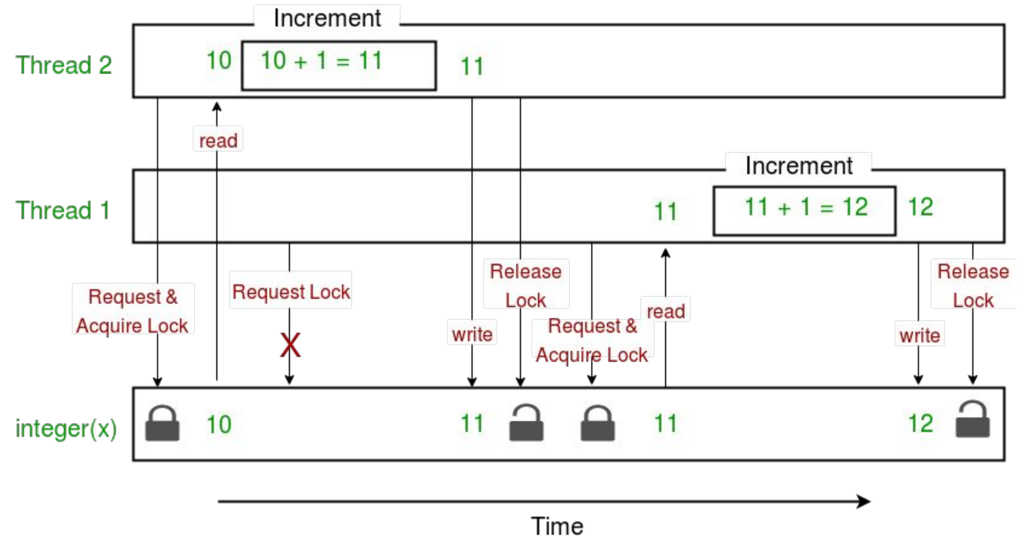
Using Locks

Lock class provides following methods:

- acquire([blocking]) : To acquire a lock. A lock can be blocking or non-blocking.
    - When invoked with the blocking argument set to True (the default), thread execution is blocked until the lock is unlocked, then lock is set to locked and return True.
    - When invoked with the blocking argument set to False, thread execution is not blocked. If lock is unlocked, then set it to locked and return True else return False immediately.
- release() : To release a lock.
    - When the lock is locked, reset it to unlocked, and return. If any other threads are blocked waiting for the lock to become unlocked, allow exactly one of them to proceed.
    - If lock is already unlocked, a ThreadError is raised.

```
import threading
# global variable x
x = 0
def increment():
    """
    function to increment global variable x
    """
    global x
    x += 1
def thread_task(lock):
    """
    task for thread
    calls increment function 100000 times.
    """
    for  in range(100000):
        lock.acquire()
        increment()
        lock.release()
def main task():
    global x
    # setting global variable x as 0
    x = 0
    # creating a lock
    lock = threading.Lock()
    # creating threads
    t1 = threading.Thread(target=thread task, args=(lock,))
    t2 = threading.Thread(target=thread_task, args=(lock,))
    # start threads
    t1.start()
    t2.start()
    # wait until threads finish their job
    t1.join()
    t2.join()
if  name  == "  main _":
    for i in range(10):
        main task()
        print("Iteration {0}: x = {1}".format(i,x))
```

Lock in action



Thread Pool

A **thread pool** is a collection of threads that are created in advance and can be reused to execute multiple tasks. The concurrent.futures module in Python provides a ThreadPoolExecutor class that makes it easy to create and manage a thread pool.

Thread Pool Example

```
import concurrent.futures

def worker():
    print("Worker thread running")

# create a thread pool with 2 threads
pool = concurrent.futures.ThreadPoolExecutor(max_workers=2)

# submit tasks to the pool
pool.submit(worker)
pool.submit(worker)

# wait for all tasks to complete
pool.shutdown(wait=True)

print("Main thread continuing to run")
```

In this example, we define a function worker that will run in a thread. We create a ThreadPoolExecutor with a maximum of 2 worker threads. We then submit two tasks to the pool using the submit method. The pool manages the execution of the tasks in its worker threads. We use the shutdown method to wait for all tasks to complete before the main thread continues.

Conclusion

Advantages:

- It doesn’t block the user. This is because threads are independent of each other.
- Better use of system resources is possible since threads execute tasks parallely.
- Enhanced performance on multi-processor machines.
- Multi-threaded servers and interactive GUIs use multithreading exclusively.

Disadvantages:

- As number of threads increase, complexity increases.
- Synchronization of shared resources (objects, data) is necessary.
- It is difficult to debug, result is sometimes unpredictable.
- Potential deadlocks which leads to starvation, i.e. some threads may not be served with a bad design
- Constructing and synchronizing threads is CPU/memory intensive.

Additional example

Cafe simulation

```
import time

menu = {'1':{'name':'Americano', 'duration':5}, '2': {'name':'Cappuchino', 'duration':10}, '3': {'name':'Cheeseburger', 'duration':20}}

auto_increment = 0

def print_menu():
    print('Наше меню:')
    for item in menu.keys():
        print(item, '->', menu[item]['name'])

def cashier():
    while True:
        print('-'*40, '\nВільна каса')
        print_menu()
        order = input('Введіть номера замовлень через пробіл: ').split()
        global auto_increment
        auto_increment += 1
        order_id = auto_increment
        start = time.time()
        print('Ви замовили:')
        for item in order:
            print('-', menu[item]['name'])
        print('Номер вашого замовлення - ', order_id)
        print('Очикуйте...')
        for item in order:
            worker(menu[item])
        print('Ваше замовлення готове')
        print('Ви очикували: ', time.time() - start, ' секунд. Гарного дня')

def worker(option):
    print('Starting to do', option['name'])
    time.sleep(option['duration'])
    print('Finished doing')

def main():
    print('Mac is open')
    cashier()

if __name__ == '__main__':
    main()
```

```python
import time
import asyncio

menu = {'1':{'name':'Americano', 'duration':5}, '2': {'name':'Cappuchino', 'duration':10}, '3': {'name':'Cheeseburger', 'duration':20}}

auto_increment = 0

def print_menu():
    print('Наше меню:')
    for item in menu.keys():
        print(item, '->', menu[item]['name'])

async def cashier():
    while True:
        print('-'*40, '\nВільна каса')
        print_menu()
        order = input('Введіть номера замовлень через пробіл: ').split()
        global auto_increment
        auto_increment += 1
        order_id = auto_increment
        start = time.time()
        print('Ви замовили:')
        for item in order:
            print('-', menu[item]['name'])
        print('Номер вашого замовлення - ', order_id)
        print('Очикуйте...')
        tasks = []
        for item in order:
            tasks.append(asyncio.create_task(worker(menu[item])))
        await asyncio.wait(tasks)
        print('Ваше замовлення готове')
        print('Ви очикували: ', time.time() - start, ' секунд. Гарного дня')

async def worker(option):
    print('Starting to do', option['name'])
    await asyncio.sleep(option['duration'])
    print('Finished doing', option['name'])

async def main():
    print('Mac is open')
    await cashier()

if __name__ == '__main__':
    asyncio.run(main())
```

```python
import time
import threading

menu = {'1':{'name':'Americano', 'duration':5}, '2': {'name':'Cappuchino', 'duration':10}, '3': {'name':'Cheeseburger', 'duration':20}}

auto_increment = 0

def print_menu():
    print('Наше меню:')
    for item in menu.keys():
        print(item, '->', menu[item]['name'])

def cashier():
    while True:
        print('-'*40, '\nВільна каса')
        print_menu()
        order = input('Введіть номера замовлень через пробіл: ').split()
        global auto_increment
        auto_increment += 1
        order_id = auto_increment
        start = time.time()
        print('Ви замовили:')
        for item in order:
            print('-', menu[item]['name'])
        print('Номер вашого замовлення - ', order_id)
        print('Очикуйте...')
        tasks = []
        for item in order:
            tasks.append(threading.Thread(target=worker, args=(menu[item],)))
        for task in tasks:
            task.start()
        for task in tasks:
            task.join()
        print('Ваше замовлення готове')
        print('Ви очикували: ', time.time() - start, ' секунд. Гарного дня')

def worker(option):
    print('Starting to do', option['name'])
    time.sleep(option['duration'])
    print('Finished doing', option['name'])

def main():
    print('Mac is open')
    cashier()

if __name__ == '__main__':
    main()
```

```python
import time
import threading

menu = {'1':{'name':'Americano', 'duration':5}, '2': {'name':'Cappuchino', 'duration':10}, '3': {'name':'Cheeseburger', 'duration':20}}

auto_increment = 0

def print_menu():
    print('Наше меню:')
    for item in menu.keys():
        print(item, '->', menu[item]['name'])

def cashier():
    while True:
        print('-'*40, '\nВільна каса')
        print_menu()
        order = input('Введіть номера замовлень через пробіл: ').split()
        global auto_increment
        auto_increment += 1
        order_id = auto_increment
        start = time.time()
        print('Ви замовили:')
        for item in order:
            print('-', menu[item]['name'])
        print('Номер вашого замовлення - ', order_id)
        print('Очикуйте...')
        tasks = []
        lock = threading.Lock()
        for item in order:
            tasks.append(threading.Thread(target=worker, args=(lock, menu[item],)))
        for task in tasks:
            task.start()
        for task in tasks:
            task.join()
        print('Ваше замовлення готове')
        print('Ви очикували: ', time.time() - start, ' секунд. Гарного дня')

def worker(lock, option):
    if option['name'] in ('Americano', 'Cappuchino'):
        lock.acquire()
        print('Starting to do', option['name'])
        time.sleep(option['duration'])
        print('Finished doing', option['name'])
        lock.release()
    else:
        print('Starting to do', option['name'])
        time.sleep(option['duration'])
        print('Finished doing', option['name'])

def main():
    print('Mac is open')
    cashier()

if __name__ == '__main__':
    main()
```

Useful links

https://codeguida.com/post/591

https://realpython.com/async-io-python/

https://realpython.com/python-async-features/

https://realpython.com/intro-to-python-threading/

https://realpython.com/python-concurrency/

https://realpython.com/courses/threading-python/

https://realpython.com/learning-paths/python-concurrency-parallel-programming/