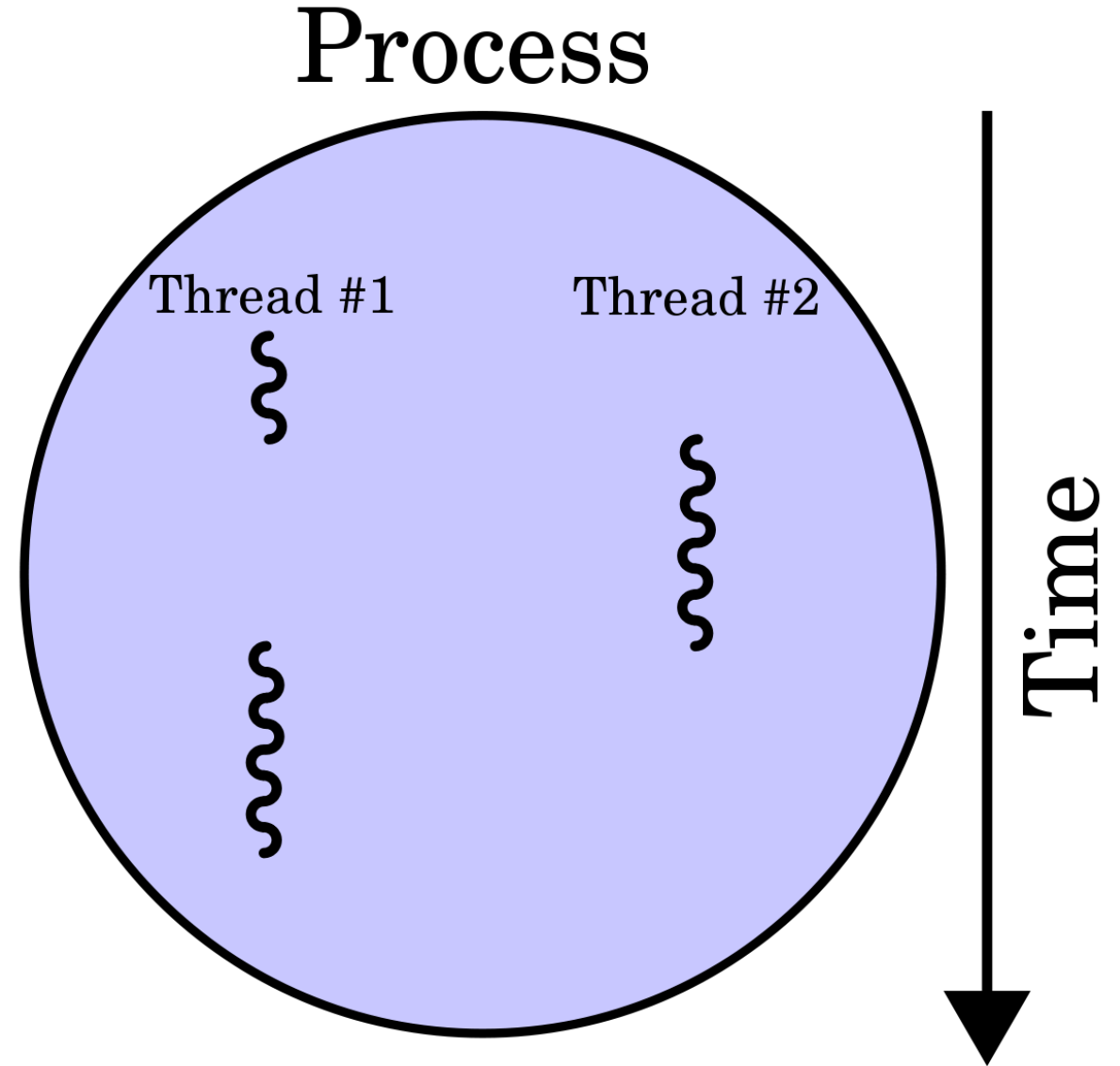


python

python

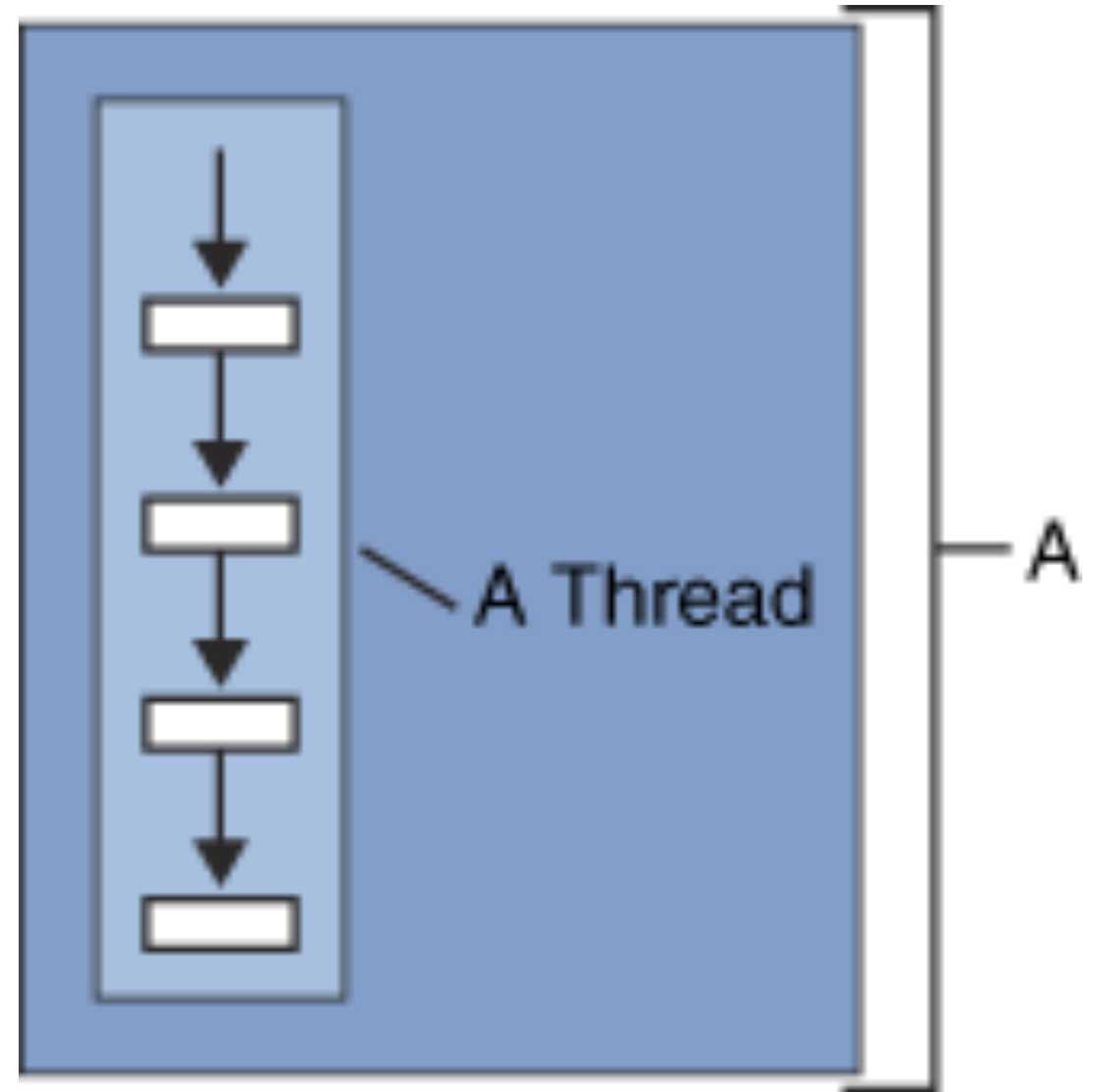
PROCESS

- A process is the instance of a computer program that is being executed by one or many threads. It contains the program code and its activity.*




THREAD

- A *thread* is a single sequential flow of control within a program.



What is GIL?

- "The Python Interpreter Lock is **mutex** that allows only one thread to hold the control of the Python interpreter. This means that only thread can be in a state of execution at any point in time".



What
problem did
the GIL solve
for Python :

Python has something that no other language has that is a reference counter. With the help of the reference counter, we can count the total number of references that are made internally in python to assign a value to a data object.

Due to this counter, we can count the references and when this count reaches to zero the variable or data object will be released automatically.

Example:

- `import sys`
- `check_var = "Check"`
- `print(sys.getrefcount(check_var))` #Output: 4
-
- `string_gfg = check_var`
- `print(sys.getrefcount(string_gfg))` #Output:5

Why was the GIL chosen as the solution

- Python provides a better way to deal with thread-safe memory management.
- Global Interpreter Lock is easy to implement in python as it only needs to provide a single lock to a thread for processing in python.

Impact on multi-threaded Python programs :

- Python programs or any computer programs then there's a difference between those that are CPU-bound in their performance and those that are I/O-bound.
- CPU push the program to its limits by performing many operations simultaneously whereas I/O program had to spend time waiting for Input/Output.

Example :CPU bound program that perform simple countdown

- `import time`
- `from threading import Thread`
-
- `COUNT = 50000000`
- `def countdown(n):`
- `while n>0:`
- `n -= 1`
- `start = time.time()`
- `countdown(COUNT)`
- `end = time.time()`
- `print('Time taken in seconds -', end - start)` **#Output : 2.5236213207244873**

Example: Two threads running parallel

- `import time`
- `from threading import Thread`
- `COUNT = 50000000`
- `def countdown(n):`
- `while n>0:`
- `n -= 1`
- `t1 = Thread(target = countdown, args =(COUNT//2,))`
- `t2 = Thread(target = countdown, args =(COUNT//2,))`
- `start = time.time()`
- `t1.start()`
- `t2.start()`
- `t1.join()`
- `t2.join()`
- `end = time.time()`
- `print('Time taken in seconds -', end - start)` **#Output: 2.183610439300537**

How to deal with Python's GIL :

- GIL is not improved as of now because python 2 having GIL implementation and if we change this in python 3 then it will create a problem for us. So instead of removing GIL, we improve the concept of GIL.
- It's one of the reasons to not remove the GIL at yet is python heavily depends on C in the backend and C extension heavily depends on the implementation methods of GIL.
- Although there are many more methods to solve the problems that GIL solve most of them are difficult to implement and can slow down the system.

How to deal with Python's GIL :

- In this implementation, python provide a different interpreter to each process to run so in this case the single thread is provided to each process in multi-processing.

```
# multiprocessing

import multiprocessing
import time

COUNT = 50000000

def countdown(n):
    while n>0:
        n -= 1

if __name__ == "__main__":
    # creating processes
    start = time.time()
    p1 = multiprocessing.Process(target = countdown, args =(COUNT//2, ))
    p2 = multiprocessing.Process(target = countdown, args =(COUNT//2, ))

    # starting process 1
    p1.start()
    # starting process 2
    p2.start()

    # wait until process 1 is finished
    p1.join()
    # wait until process 2 is finished
    p2.join()
    end = time.time()
    print('Time taken in seconds -', end - start)    #Output : 2.5148496627807617
```



Example :