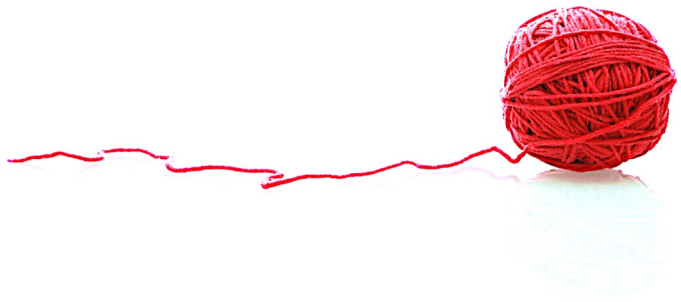
# DISTRIBUTED SYSTEM DESIGN

## Lab 3

## Multithreading & Multiprocessing

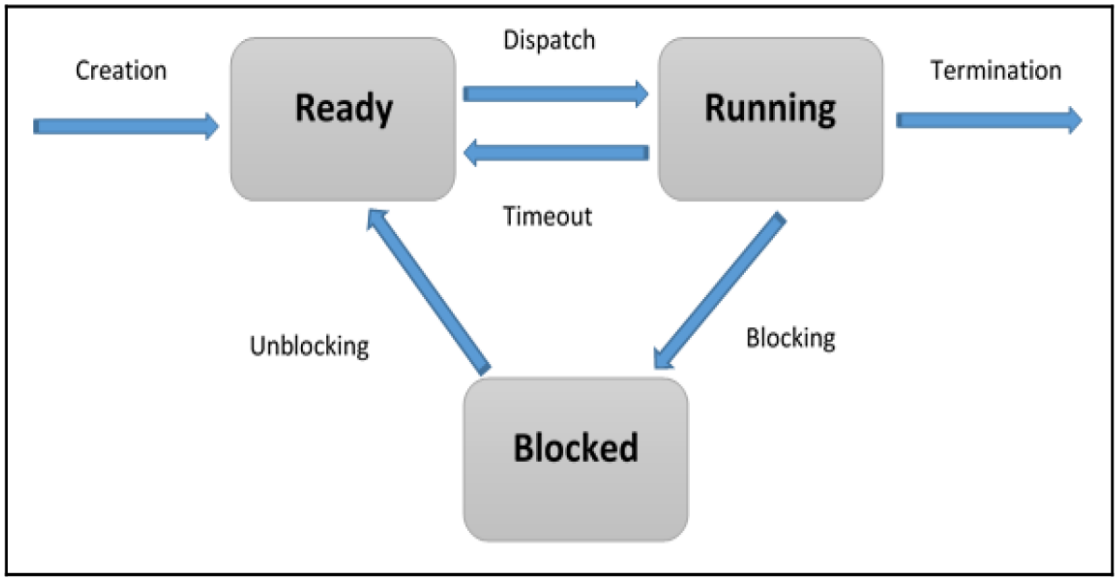
# More on threads …

- A thread is an **independent execution flow** that can be executed in parallel and concurrently with other threads in the system
- Often referred to as a **light weighted** process
- Each gets its **own stack**
- A thread is contained inside a **process** and different threads in the same process conditions **share some resources**
- **Shares** memory, data and code
- When to use? **I/O bound** applications

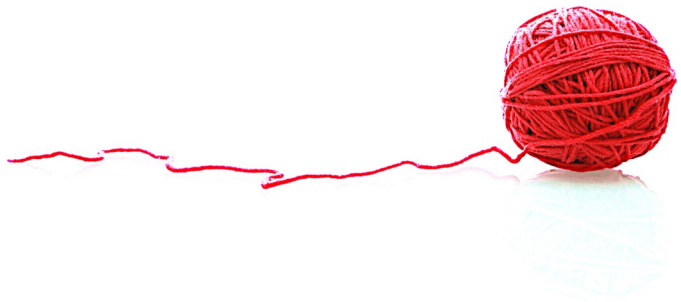


# Thread states

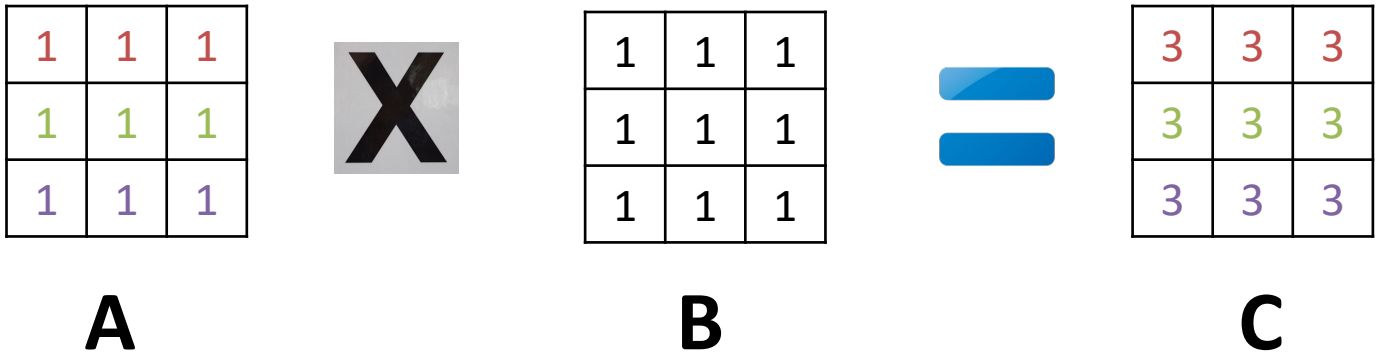
- A thread state can be ready, running, or blocked:
  - When a thread is **created**, it enters the **ready** state.
  - A thread is **scheduled for execution** by the OS and, when its turn arrives, it begins execution by going into the **running** state.
  - The thread can **wait** for a condition to occur, passing from the **running state to the blocked state**. Once the locked condition is terminated, the Blocked thread returns to the Ready state.




Thread life cycle



# Matrix Multiplication (Iterative Approach) with threads

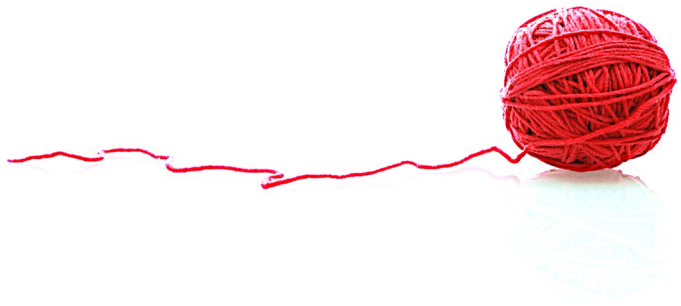



Let's use 3 threads :

- Thread number 1 multiples A:row 1 by B then get C:row 1
- Thread number 2 multiples A:row 2 by B then get C:row 2
- Thread number 3 multiples A:row 3 by B then get C:row 3

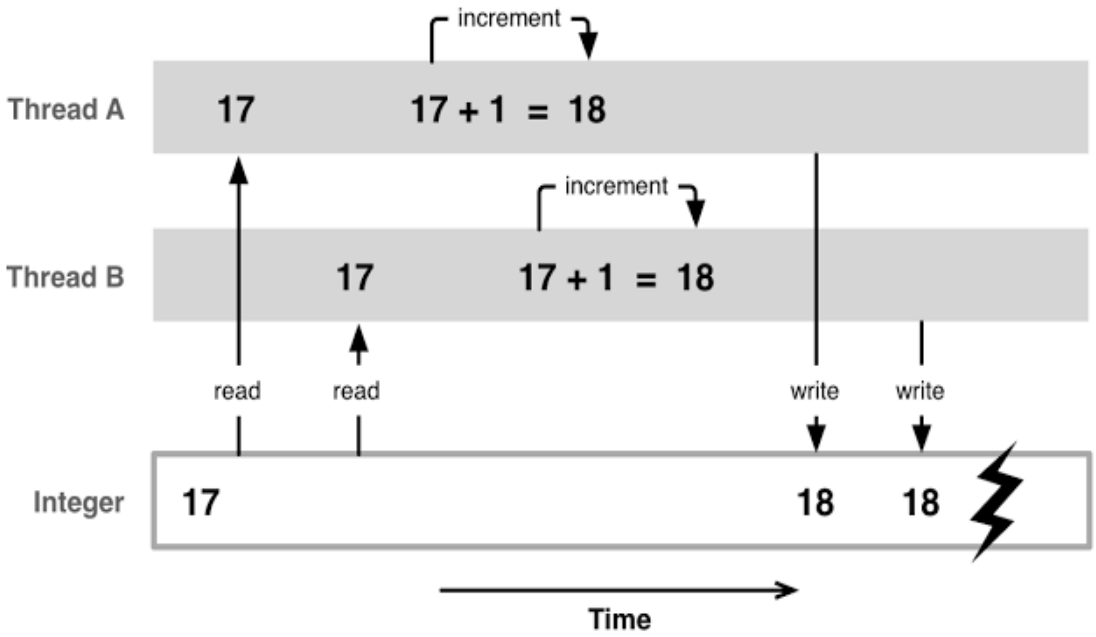
Let's use 10 by 10 matrix and 5 threads :

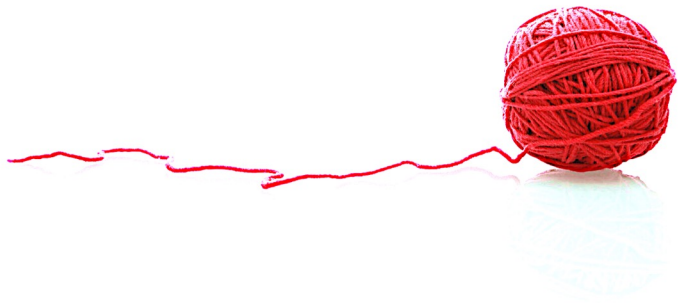
- Thread number 1 multiples A:rows 1,2 by B then get C:rows 1,2
- Thread number 2 multiples A:rows 3,4 by B then get C:rows 3,4
- Thread number 3 multiples A:rows 5,6 by B then get C:rows 5,6
- Thread number 4 multiples A:rows 7,8 by B then get C:rows 7,8
- Thread number 5 multiples A:rows 9,10 by B then get C:rows 9,10



# Threads Race Condition

- A race condition is a concurrency problem that may occur inside a critical section that access a shared data between threads.
- A critical section is a section of code that is executed by multiple threads and where the sequence of execution for the threads makes a difference in the result of the concurrent execution of the critical section.
- What Should be the final result of the following example? 19

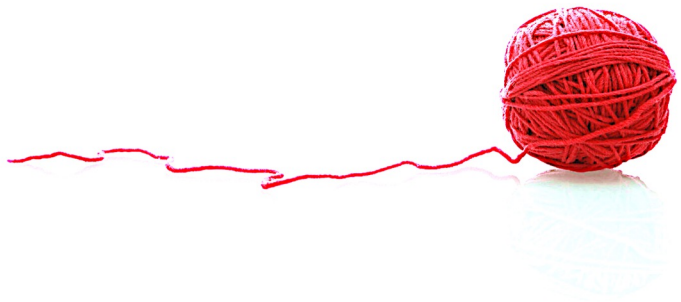


# Thread synchronization with a lock

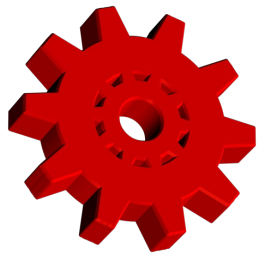
- A lock is nothing more than an object that is typically accessible by multiple threads.
- This lock tells a thread must prossess before it can proceed to the execution of a critical section of a program.
- These locks are created by executing the Lock() method, which is defined in the threading module.

```
threadLock = threading.Lock()
def run(self):
    #Acquire the Lock
    threadLock.acquire()
    print ("---> " + self.name + \ " running, belonging
    to process ID "\ + str(os.getpid()) + "\n")
    time.sleep(1000)
    print ("---> " + self.name + " over\n")
    #Release the Lock
    threadLock.release()
```



# What is a Process?

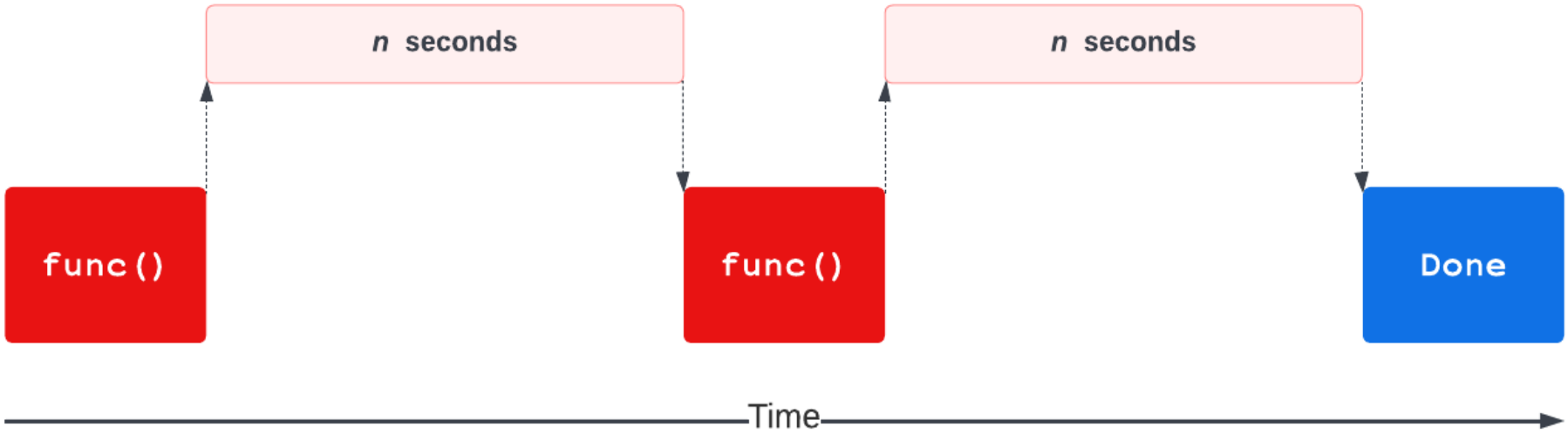
- An **independent** process-of-control
- Process are **Share nothing**
- Processes are **Big**
- Processes run on **multiple cores**
- When to use? **CPU-bound** applications



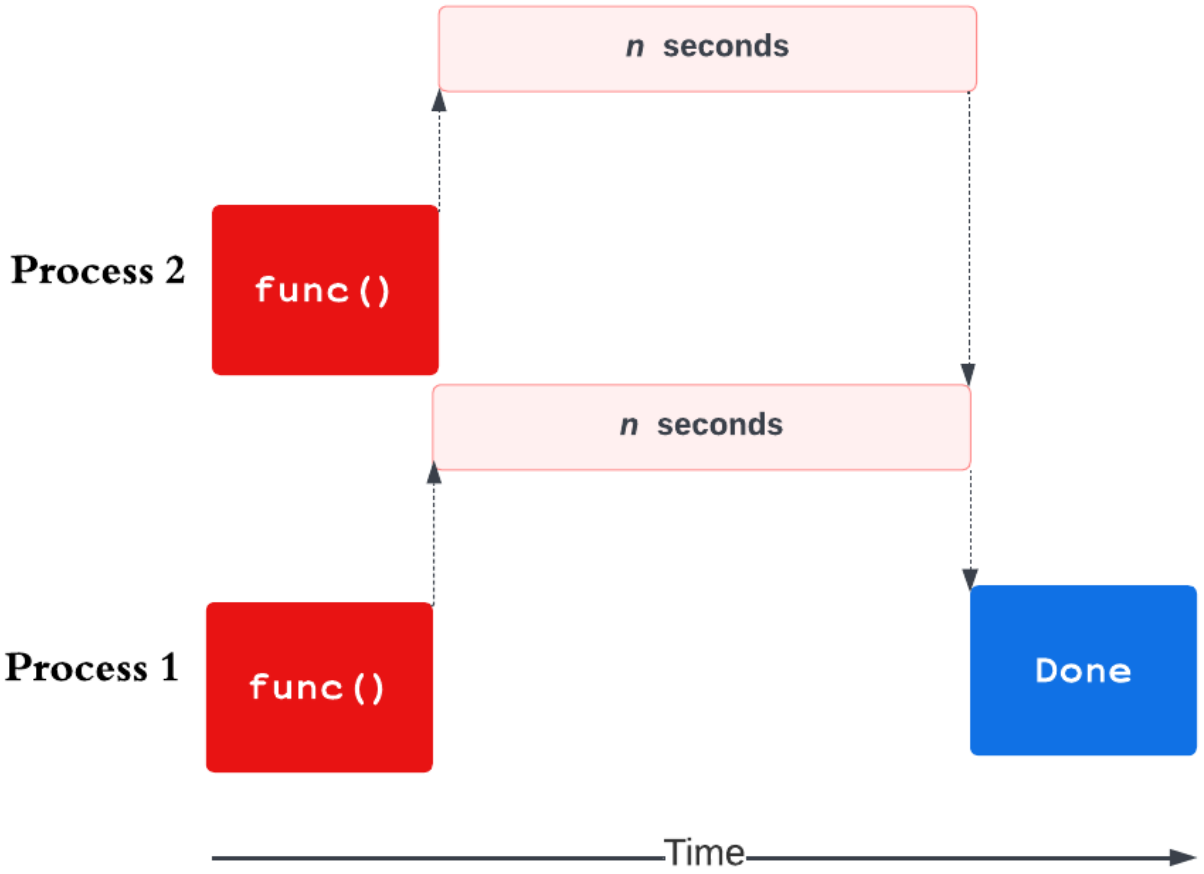
# Threads v/s Processes

| | Threads | Processes |
|---|---|---|
| 1. | System calls are not involved | System calls are involved |
| 2. | Context switching is faster | Context switching is slower |
| 3. | Blocking a thread will block entire process | Blocking a process will not block another process |
| 4. | Threads share same copy of code and data | Different processes have different copies of code and data |
| 5. | Interdependent | Independent |
| 6. | I/O bound | CPU bound |

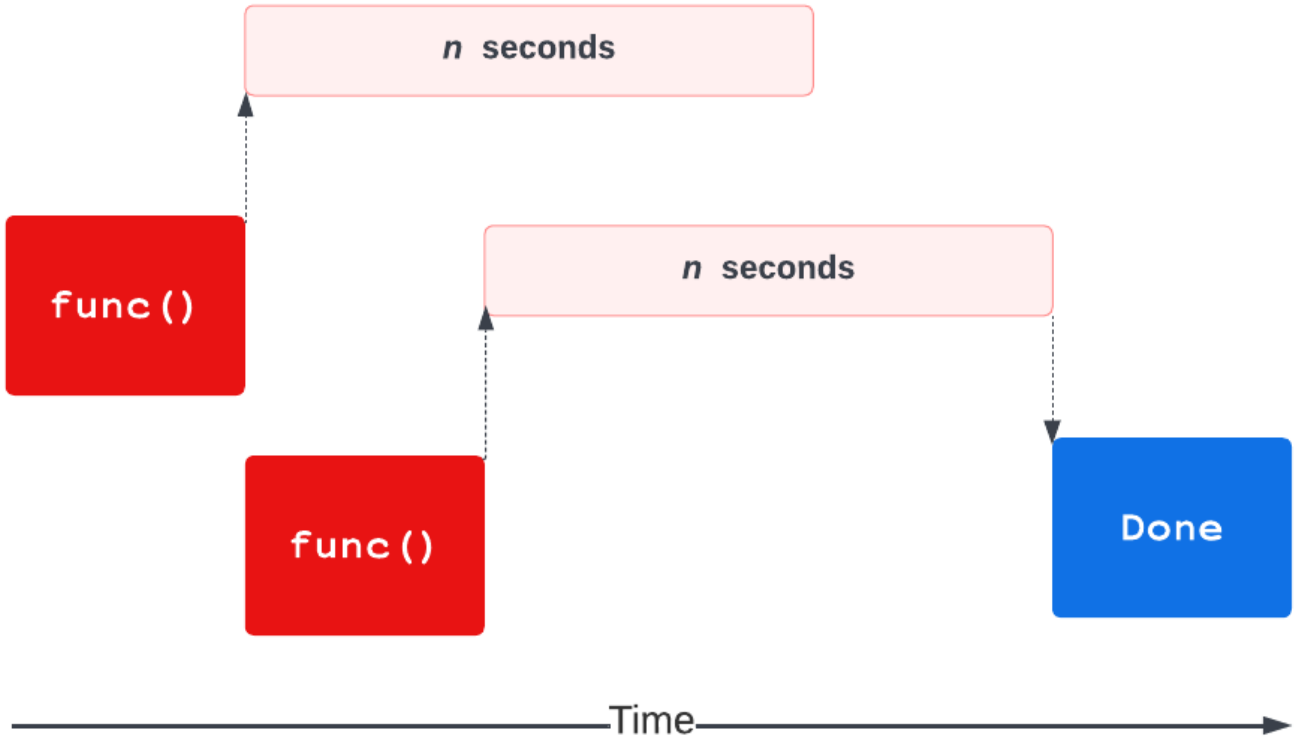
# Visualizing execution: serial



# Visualizing execution: processes



# Visualizing execution: threads



# Hello world in Multiprocessing

```
import os
import multiprocessing
from time import time, sleep

def compute():
    print('computing...')
    sleep(1)

def compute_multi_processing():
    print('using multi-processing\nCPU-core(s) available: ', os.cpu_count())
    start = time()
    p1 = multiprocessing.Process(target=compute)
    p2 = multiprocessing.Process(target=compute)
    p1.start()
    p2.start()
    p1.join()
    p2.join()
    finished = time()
    print(f'time taken (multi-processing): {round(finished - start, 2)}
second(s)')

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

# Handling multiple processes

- Pool class in multiprocessing can handle an enormous number of processes.
- It **allows you to run multiple jobs per process** (due to its ability to queue the jobs)
- The memory is allocated only to the executing processes, unlike the Process class, which allocates memory to all the processes.

```
import time
from multiprocessing import Pool

def sum_square(number):
    s = 0
    for i in range(number):
        s += i * i
    return s

def compute_multiprocessing(numbers):
    print('using multiprocessing')
    start = time.time()
    p = Pool()
    result = p.map(sum_square, numbers)
    p.close()
    p.join()
    finish = time.time()
    print(f'time taken (multiprocessing execution): {round(finish - start, 2)} second(s)')

if __name__ == '__main__':
    n = range(30000)
    compute_multiprocessing(numbers=n)
```

# Exercises

1. Run multithreading examples:
   - Matrix multiplication
   - Thread locking
2. Run hello world pandas.ipynb notebook
3. Run multiprocessing examples:
   - Hello world
   - Pool
   - Pandas with multiprocessing
4. Modify *Pandas with multiprocessing (from exercise)* example to calculate missing years *(NAN)* **serially**
5. Modify *Pandas with multiprocessing (from exercise)* example to calculate missing years *(NAN)* using **multiprocessing**

   you should get **410974** missing values