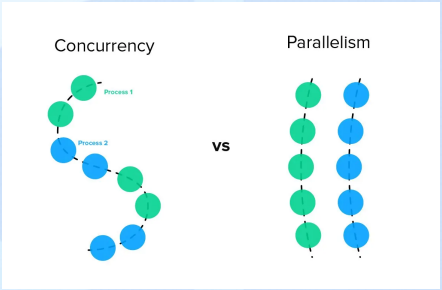
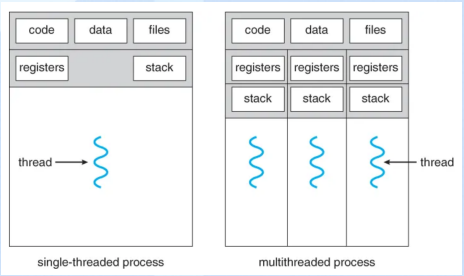
Concurrency vs. Parallelism -- Why It Matters

- Concurrency: multiple tasks make progress by interleaving their execution (not necessarily at the same time).
- Parallelism: tasks execute at the same time on different cores/processing units to increase throughput.



Threads vs. Processes -- Memory Model

- Thread: is a thread of execution in a program. Aka, lightweight process.
- Process: is an instance of a computer program that is being executed.
- Implications:
  - Threads share the memory and state of the parent, process share nothing.
  - Processes use inter-process communication (IPC) to communicate, thread do not.
  - A process can have 1 or several threads.
- Scheduling: the OS kernel schedules threads on CPU cores; a process can host one or multiple threads.



Multi-threading Basics: Creating Threads with `threading.Thread`

- Two threads (`t1`, `t2`) are created to run the `worker` function in parallel, each simulating work with `time.sleep(2)`.
- Because both threads sleep concurrently, the total execution time is about 2 seconds instead of 4.

```
Thread A starting
Thread B starting
Thread A done
Thread B done
Timing: 2.002 sec
```

```python
import threading
import time

def worker(name):
    print(f"Thread {name} starting")
    time.sleep(2)
    print(f"Thread {name} done")

# Create two threads
t1 =threading.Thread(target=worker, args=("A",))
t2 =threading.Thread(target=worker, args=("B",))

start =time.time()
t1.start()
t2.start()
t1.join()
t2.join()
end =time.time()

print("Timing: ", end -start, "sec")
```

Multi-threading Basics: Subclassing `threading.Thread`

- Step 1: Create a class inheriting from `threading.Thread`.
- Step 2: Override the `run()` method with the task logic.
- Step 3: Instantiate the class and call `start()`.
- Step 4: Use `join()` to wait for completion.

```
Thread A starting
Thread B starting
Thread A done
Thread B done
```

```python
import threading
import time

class Worker(threading.Thread):
    def __init__(self, name):
        super().__init__()
        self.name =name

    def run(self):
        print(f"Thread {self.name} starting")
        time.sleep(2)
        print(f"Thread {self.name} done")

# Create and start threads
t1 =Worker("A")
t2 =Worker("B")
t1.start()
t2.start()
t1.join()
t2.join()
```

Shared Memory in Python

Multi-threading Basics: Prime Calculation with single Thread

- Compute the sum of all primes up to 200,000.
- Using 1 thread only.

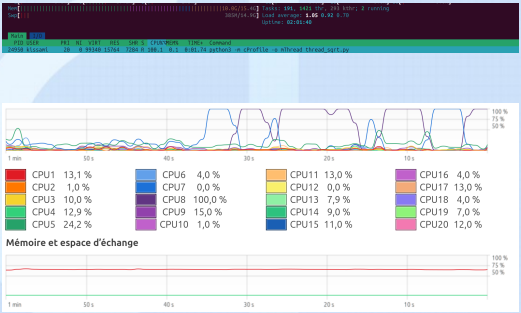
```python
def isPrime(n):
    if n <2:
        return False
    if n ==2:
        return True
    max_val =int(math.ceil(math.sqrt(n)))
    i =2
    while i <=max_val:
        if n % i ==0:
            return False
        i +=1
    return True

def sum_primes(n):
    return sum([x for x in range(2, n) if isPrime(x)])
```

```python
if __name__ =="__main__":

    for i in range(0, 200000, 500):
        sum_primes(i)
```

Shared Memory in Python

Multi-threading Basics: Profiling Single-Thread Prime Calculation

- Profiling with `cProfile` shows most time spent in:
  - `sum_primes`, `isPrime`
- tottime = $percall_{local}$ x ncalls
- $percall_{cumul}$ = cumtime / ncalls




```
 ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      1    0.000    0.000   47.330   47.330 {built-in method builtins.exec}
      1    0.000    0.000   47.330   47.330 isprime.py:1(<module>)
    400    4.139    0.010   47.330    0.118 isprime.py:16(sum_primes)
39899202  38.625    0.000   43.177    0.000 isprime.py:3(isPrime)
39898803   2.470    0.000    2.470    0.000 {built-in method math.sqrt}
39898803   2.081    0.000    2.081    0.000 {built-in method math.ceil}
    400    0.014    0.000    0.014    0.000 {built-in method builtins.sum}
      1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

```
Execution time (1 thread) = 47.330 sec
```

4.139 + 38.625 + 2.470 + 2.081 + 0.014 = 47.33 secondes

Multi-threading Basics: Prime Calculation with Multi Thread (8 threads)

```
def do_work(q):
    while True:
        try:
            x = q.get(block=False) # Get an item from the queue (non-blocking)
            sum_primes(x)         # Compute the sum of primes below x
        except Empty:
            break

if __name__ =="__main__":
    work_queue =Queue()
    for i in range(0, 200000, 500):
        work_queue.put(i)         # inserts the value i into the queue.

    threads =[Thread(target=do_work, args=(work_queue,)) for _ in range(8)]

    for t in threads:
        t.start()
    for t in threads:
        t.join()
```

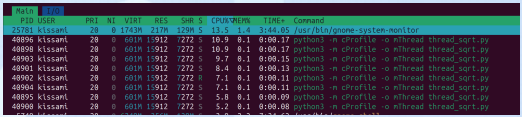
Shared Memory in Python

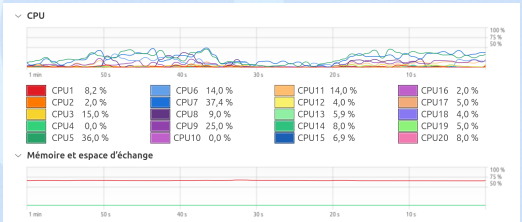
Profiling Multi-Thread Prime Calculation

- Profiling with `cProfile` highlights time spent in:
  - `threading.py:join`
  - `isPrime`
  - `_thread.lock.acquire`

tottime	percall	cumtime	percall	filename:lineno(function)
0.000	0.000	171.816	21.477	threading.py:1115(join)
0.000	0.000	171.811	21.476	threading.py:1153(_wait_for_tstate_lock)
0.001	0.000	114.813	0.261	{method 'acquire' of '_thread.lock' objects}
50.089	0.000	57.532	0.001	thread_sqrt.py:6(isPrime)
0.000	0.000	57.442	57.442	{built-in method builtins.exec}
0.000	0.000	57.442	57.442	thread_sqrt.py:1(<module>)
0.003	0.000	57.440	57.440	threading.py:1016(_bootstrap)
0.008	0.001	57.440	57.440	threading.py:1056(_bootstrap_inner)
2.978	0.000	2.978	0.000	{built-in method math.sqrt}
2.525	0.000	2.525	0.000	{built-in method math.ceil}

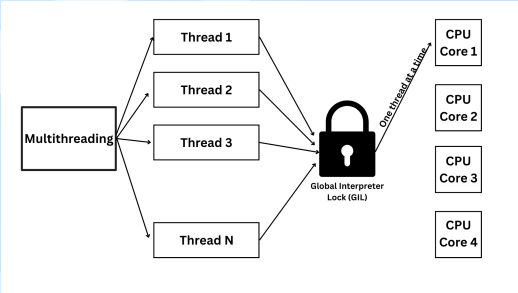
```
Execution time (8 threads) = 57.442 sec
```





Multi-threading Basics: The Global Interpreter Lock (GIL)

- Only one thread can run Python bytecode at a time.
- CPU-bound: No true parallelism ⇒ threads wait.
- I/O-bound: GIL is released during blocking I/O ⇒ overlap possible.




```
def add(a, b):
    return a +b
```

```
0 LOAD_FAST 0 (a)
2 LOAD_FAST 1 (b)
4 BINARY_OP 0 (+)
6 RETURN_VALUE
```

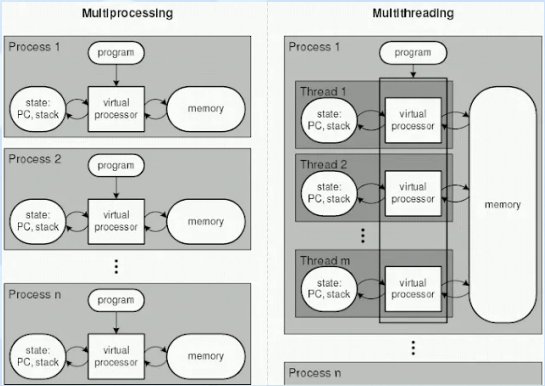
Multi-threading Basics: Why Multi-threading in Python Doesn't Scale for CPU-bound Tasks

- Consequence: Even on multi-core CPUs, threads run in a concurrent but not parallel way.
- In Profiling:
  - Real runtime ≈ 57s.
  - Cumulative time (e.g., `join` = 171s) adds up waiting + scheduling overhead from all threads.
- Takeaway:
  - For I/O tasks → threads can still improve responsiveness.
  - For CPU-heavy work → use `multiprocessing` (separate processes, separate GILs).

Shared Memory in Python

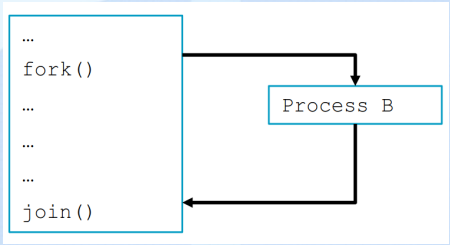
Multithreading vs. Multiprocessing

- Multithreading ⇒ Concurrency
  - Multiple threads share the same memory inside one process.
  - Good for I/O-bound tasks (overlap waiting times).
  - Limited by the GIL: no true parallel CPU execution.
- Multiprocessing ⇒ Parallelism
  - Each process has its own memory and interpreter.
  - Achieves true parallelism across CPU cores.
  - Higher cost: inter-process communication (IPC).
- Key takeaway: `threading` = concurrency (I/O-bound). `multiprocessing` = parallelism (CPU-bound).



Multi-processing Basics: Elements of Programming

- Memory Isolation
  - Processes do NOT share memory address space
- Fork/Join Execution Model
  - Fundamental way of expressing concurrency within a computation
  - `Fork` creates a new child process
  - Parent continues after the `Fork` operation
  - Child begins operation separate from the parent
  - Parent waits until child `joins` (continues afterwards)



Multi-processing Basics: Race Conditions

- A Race Condition occurs if:
  - Two or more processes manipulate a shared resource concurrently
  - The outcome depends on the order of access

Process P1:

(1) MOV `SUM`, Reg1

(2) ADD #1, Reg1

(3) MOV Reg1, `SUM`

Process P2:

(1') MOV `SUM`, Reg1

(2') ADD #1, Reg1

(3') MOV Reg1, `SUM`

Possible interleavings:

- (1')(1)(2)(3)(2')(3') ⇒ `SUM = SUM+1`
- (1)(1')(2')(3')(2)(3) ⇒ `SUM = SUM+1`
- (1)(2)(3)(1')(2')(3') ⇒ `SUM = SUM+2`

Solution

Synchronization needed to prevent race conditions
Mutual Exclusion: prevents simultaneous access to a shared resource.

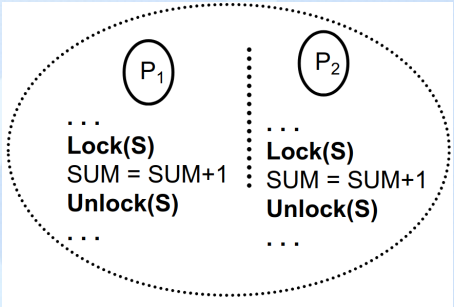
Multi-processing Basics: Synchronization

Variable Mutex: S

- Boolean: 0 / 1
- General: Integer ≥ 0

Functions:

- `Lock(S)`
  - If $S == 0$ then wait until $S > 0$
  - If $S > 0$ then $S = S - 1$
- `Unlock(S)`
  - $S = S + 1$



- (1′)(2′)(3′)(1)(2)(3) $\Rightarrow$ `SUM = SUM+2`
- (1)(2)(3)(1′)(2′)(3′) $\Rightarrow$ `SUM = SUM+2`

Shared Memory in Python

Multi-processing Basics: Join/Fork Model

- The `multiprocessing` module provides an easy API for parallelism.
- Steps:
  1. Create a `Process` structure with target function + args
  2. Start processes with `.start()`
  3. Wait for processes to finish with `.join()`
- This model avoids the GIL by using separate processes.

```python
import multiprocessing

def print_cube(num):
    ...

def print_square(num):
    ...

if __name__ =="__main__":
    p1 =multiprocessing.Process(target=print_square, args=(10,))
    p2 =multiprocessing.Process(target=print_cube, args=(10,))

    p1.start()
    p2.start()

    p1.join()
    p2.join()

    print("Done!")
```

Multi-processing Basics: Shared Memory

- Each process has its own memory space.
- Global variables are not shared between processes.
- Example: modifying a global list in a process does not affect the parent.

```
In process: [1, 4, 9, 16]
In main: []
```

```python
import multiprocessing

result =[]

def square_list(mylist):
    global result
    for num in mylist:
        result.append(num *num)
    print("In process:", result)

if __name__ =="__main__":
    mylist =[1,2,3,4]
    p1 =multiprocessing.Process(target=square_list, args=(mylist,))

    p1.start()
    p1.join()

    print("In main:", result)
```

Shared Memory in Python

Multi-processing Basics: Shared Memory with `Array` and `Value`

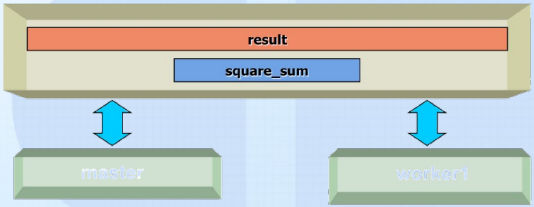
- The `multiprocessing` module provides objects to share data:
  - `Array`: a `ctypes` array allocated in shared memory.
  - `Value`: a `ctypes` variable allocated in shared memory.
- These objects must be passed as arguments to processes.
- Enables efficient communication between processes.

```
Result array: [1, 4, 9, 16]
Sum of squares: 30
```

```python
import multiprocessing

def square_list(mylist, result, square_sum):
    for idx, num in enumerate(mylist):
        result[idx] =num *num
        square_sum.value +=result[idx]

if __name__ =="__main__":
    mylist =[1,2,3,4]
    result =multiprocessing.Array('i', 4)
    square_sum =multiprocessing.Value('i')

    p1 =multiprocessing.Process(target=square_list, args=(mylist, ↩
          result, square_sum))

    p1.start()
    p1.join()

    print("Result array:", result[:])
    print("Sum of squares:", square_sum.value)
```

Multi-processing Basics: Shared Memory with `Manager` (Advanced)

- `multiprocessing.Manager` allows sharing complex objects:
  - `list`, `dict`, `Queue`, `Array`, etc.
- A single `Manager` can be used by multiple processes, even across different machines.
- Slower than direct shared memory (`Array` / `Value`).



Multi-processing Basics: Shared Memory with `multiprocessing.Manager`

```python
from multiprocessing import Manager, Process

def worker(shared_list, idx):
    # each worker squares its index
    shared_list[idx] =shared_list[idx] **2

if __name__ =="__main__":
    with Manager() as manager:
        data =manager.list([i for i in range(10)])

        processes =[Process(target=worker, args=(data, i)) for i in range(len(data))]

        for p in processes:
            p.start()

        for p in processes:
            p.join()

        print("Final result:", list(data))
```

```
Final result: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Shared Memory in Python

Multi-processing Basics: Using `multiprocessing.Queue`

- `Queue` is a simple way to communicate between processes.
- Can pass any Python object.
- Key functions:
  - `put()` : insert a value into the queue.
  - `get()` : read/remove a value from the queue.
  - `empty()` : check if the queue is empty.
- Useful for synchronizing and sharing results between workers.

```python
import multiprocessing

def square_list(mylist, q):
    for num in mylist:
        q.put(num *num)

def print_queue(q):
    print("Queue elements:")
    while not q.empty():
        print(q.get())

if __name__ =="__main__":
    mylist =[1, 2, 3, 4]
    q =multiprocessing.Queue()

    p1 =multiprocessing.Process(target=square_list, args=(mylist, q))
    p2 =multiprocessing.Process(target=print_queue, args=(q,))

    p1.start(); p2.start()
    p1.join(); p2.join()
```

Multi-processing Basics: Prime Calculation Queue (8 processes)

- Overhead comes from process management (fork/join, waiting).
- Different from threads: no `acquire` or GIL contention.

```
if __name__ =="__main__":
    work_queue =Queue()
    for i in range(0, 200000, 500):
        work_queue.put(i)

    processes =[Process(target=do_work, args=(work_queue,))↩
            for _ in range(8)]

    # start_time = time.time()
    for p in processes:
        p.start()
    for p in processes:
        p.join()
```

```
tottime  percall  cumtime  percall filename:lineno(function)
  0.000    0.000   11.893   11.893 {built-in method builtins.exec}
  0.000    0.000   11.893   11.893 process_sqrt.py:1(<module>)
  0.000    0.000   11.879    1.485 process.py:142(join)
  0.000    0.000   11.879    0.330 popen_fork.py:24(poll)
  0.000    0.000   11.879    1.485 popen_fork.py:36(wait)
 11.879    0.330   11.879    0.330 {built-in method posix.waitpid}
  0.000    0.000    0.019    0.005 <frozen importlib._bootstrap>:1349(_find_and_load)
  0.000    0.000    0.019    0.005 <frozen importlib._bootstrap>:1304(_find_and_load_unlocked)
  0.000    0.000    0.018    0.005 <frozen importlib._bootstrap>:911(_load_unlocked)
  0.000    0.000    0.018    0.004 <frozen importlib._bootstrap_external>:989(exec_module)
```

```
Execution time (8 process) = 11.893 sec
```

Multi-processing Basics: Race Conditions

- Race condition occurs when multiple processes access a shared variable concurrently.
- Without synchronization, final result is unpredictable.
- Example: Withdraw and deposit modify the same `Value`.

```
def withdraw(balance):
    for _ in range(10000):
        balance.value -=1

def deposit(balance):
    for _ in range(10000):
        balance.value +=1

if __name__ =="__main__":

    # initial balance (in shared memory)
    balance =multiprocessing.Value('i', 100)

    p1 =multiprocessing.Process(target=withdraw, args=(balance,))
    p2 =multiprocessing.Process(target=deposit, args=(balance,))

    p1.start(); p2.start()

    p1.join(); p2.join()
```

Shared Memory in Python

Multi-processing Basics: Locks to Prevent Race Conditions

- `multiprocessing.Lock()` ensures mutual exclusion.
- Only one process can access the shared resource at a time.
- Prevents data corruption, ensures consistent results.

```
Final balance: 100
```

```python
def withdraw(balance, lock):
    for _ in range(10000):
        with lock:
            balance.value -=1
def deposit(balance, lock):
    for _ in range(10000):
        with lock:
            balance.value +=1

if __name__ =="__main__":
    balance =multiprocessing.Value('i', 100)
    # creating a lock object
    lock =multiprocessing.Lock()

    # creating new processes
    p1 =multiprocessing.Process(target=withdraw, args=(balance,lock))
    p2 =multiprocessing.Process(target=deposit, args=(balance,lock))

    p1.start(); p2.start()

    p1.join(); p2.join()
```

Shared Memory in Python

Multi-processing Basics: Using `multiprocessing.Pool`

- `Pool` represents a pool of worker processes.
- Allows tasks to be distributed automatically.
- Methods:
  - `map(func, iterable)` – apply function to list of inputs.
  - `apply()` – run function once.
  - `apply_async()` – asynchronous execution.
- Efficient for data-parallel computations.

```
import multiprocessing
import os

def square(n):
    print(f"Id for {n}: {os.getpid()}")
        return n *n

if __name__ =="__main__":
    mylist =[1,2,3,4,5]

    # creating a pool object
    p =multiprocessing.Pool()

    # map list to target function
    result =p.map(square, mylist)

    print(result)
```

Multi-processing Basics: Prime Calculation using Pool (8 processes)

```
if __name__ =="__main__":
    LIMIT =200000
    STEP =500
    N_PROCESSES =8

    # list of jobs
    tasks =list(range(0, LIMIT, STEP))

    with multiprocessing.Pool(processes=N_PROCESSES) as pool:
    results =pool.map(sum_primes, tasks) # apply sum_primes for ↩
          each job
```

```
ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.000    0.000   16.164   16.164 connection.py:246(recv)
     1    0.000    0.000   16.163   16.163 connection.py:429(_recv_bytes)
     1    0.000    0.000   16.161   16.161 connection.py:390(_recv)
     1    5.486    5.486   16.161   16.161 {built-in method posix.read}
     1    0.000    0.000   10.704   10.704 {built-in method builtins.exec}
     1    0.000    0.000   10.704   10.704 pool_sqrt.py:1(<module>)
     1    0.000    0.000   10.692   10.692 pool.py:738(__exit__)
     1    0.000    0.000   10.686   10.686 pool.py:654(terminate)
     1    0.000    0.000   10.678   10.678 util.py:208(__call__)
     1    0.000    0.000   10.678   10.678 pool.py:680(_terminate_pool)
:3
```

```
Execution time (8 process) = 10.704 sec
```