“When you come to a fork in the road, take it”

Yogi Bera, 1925

Hands-on H.1:
Python Multiprocessing

CS205: Computing Foundations for Computational Science
Dr. David Sondak
Spring Term 2021

INSTITUTE FOR APPLIED COMPUTATIONAL SCIENCE AT HARVARD UNIVERSITY

HARVARD School of Engineering and Applied Sciences

Lectures developed by Dr. Ignacio Illorente

Before We Start

Where We Are

Computing Foundations for Computational and Data Science

How to use modern computing platforms in solving scientific problems

Intro: Large-Scale Computational and Data Science

A. Parallel Processing Fundamentals

B. Parallel Computing

- B.1. Foundations of Parallel Computing
- B.2. Performance Optimization
- B.3. Accelerated Computing
- B.4. Shared-memory Parallel Processing
- B.5. Distributed-memory Parallel Processing

C. Parallel Data Processing

Wrap-Up: Advanced Topics

CS205: Contents

APPLICATION SOFTWARE

A.3 APPLICATION PARALLELISM

A.4. PARALLEL PROGRAM DESIGN

PROGRAMMING MODEL

Optimization

OpenACC

Python MP

OpenMP

MPI

Spark

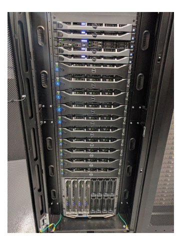
Map-Reduce

B. BIG COMPUTE

PLATFORM

C. BIG DATA

amazon web services



Open Nebula

FAS RC

FAS RC CANNON HARVARD'S LARGEST CLUSTER

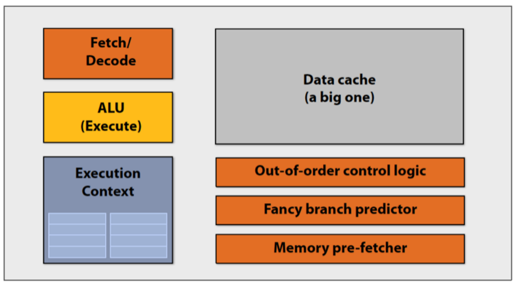
A.2. LARGE-SCALE PROCESSING ON CLOUD

A.1 PARALLEL ARCHITECTURES

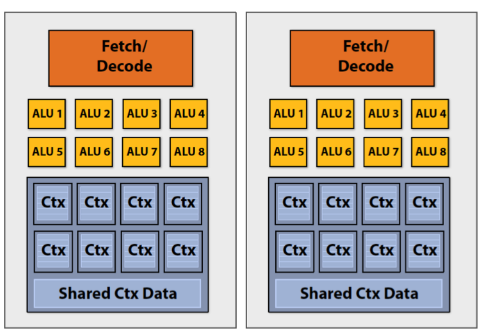
Context

Python Multiprocessing

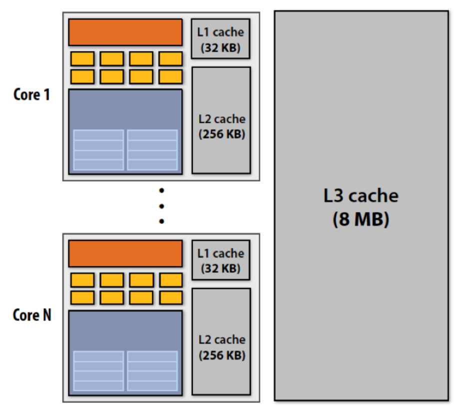
How to develop code that can make effective use of existing parallelism at different levels?



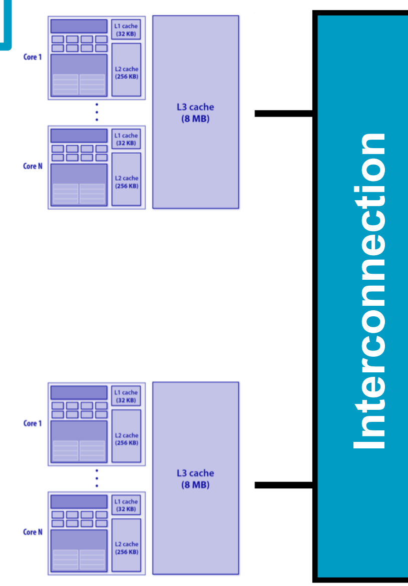
ILP/Data



Many-core



Multi-core



Multi-node

Hands-on Examples

Requirements

1. Unix-like shell (Linux, Mac OS or AWS VM)
2. Python installed

Roadmap

Python Multiprocessing

Multi-processing Basics

Process Creation and Synchronization

Process Communication

Process Synchronization

Work Distribution

Examples

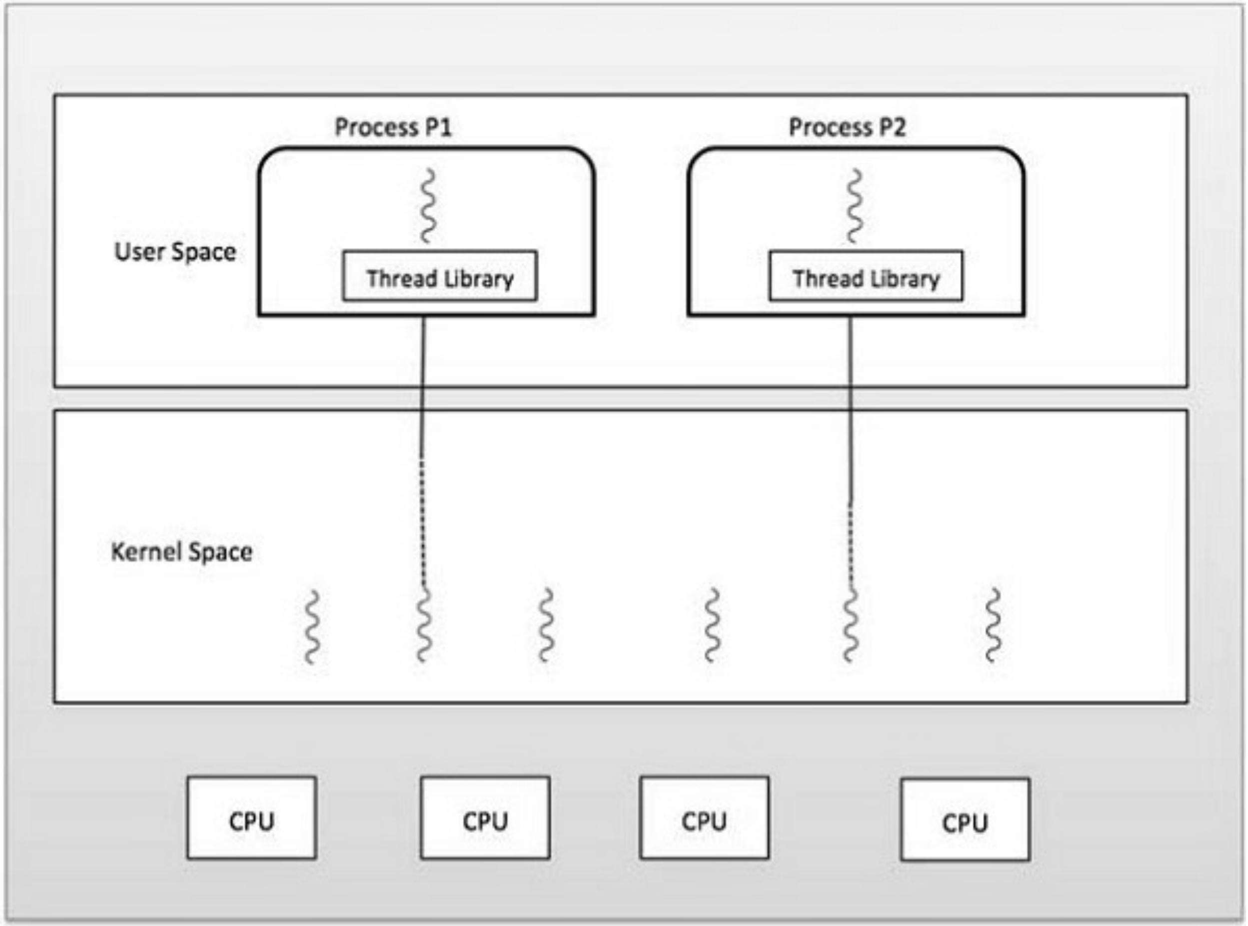
Multi-processing Basics

What is a Process?

- A process is an instance of a computer program that is being executed

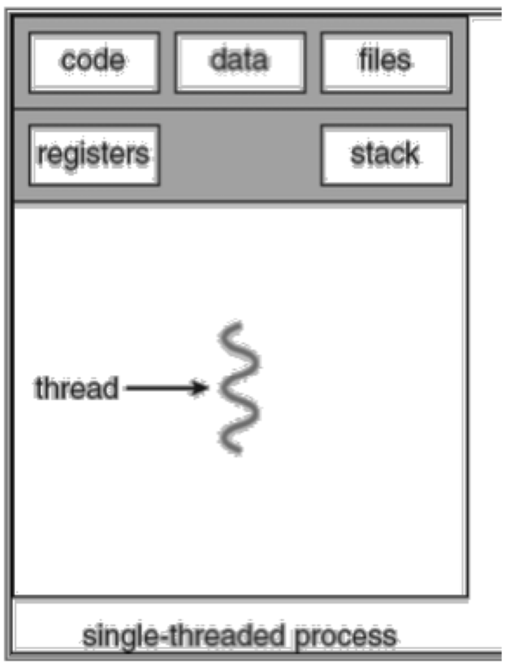
- A process can have 1 or several threads (1 in this hands-on)

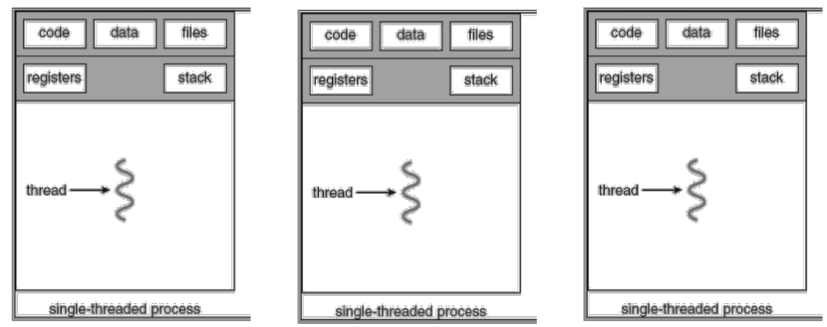
- The kernel of the OS schedules threads to multiple cores



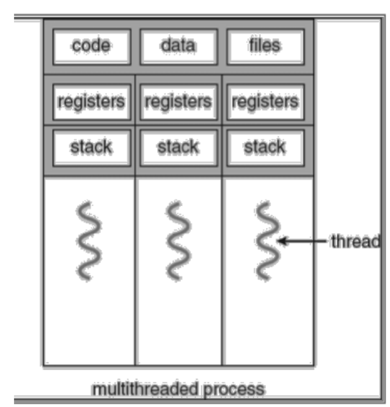
Multi-processing Basics

Multi-Processing vs Multi-Threading





MULTI-PROCESSING



MULTI-THREADING

Multi-processing Basics

Multi-Processing vs Multi-Threading in Python

The CPython implementation (called CPI, C Python Interpreter) is not thread-safe and only permits a single thread to run at a time

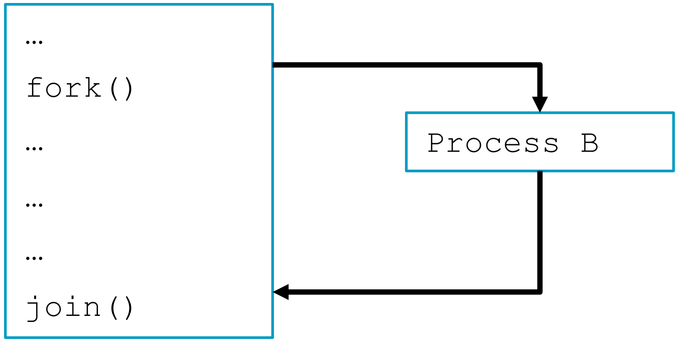
The standard Python library has two main modules for parallel computing:

- `threading`
  - Good for I/O bound tasks
  - Subject to the Global Interpreter Lock (GIL)
- `multiprocessing`
  - Circumvents the GIL
  - Useful for parallel computation
- We will focus on the multiprocessing module

Multi-processing Basics

Elements of Programming

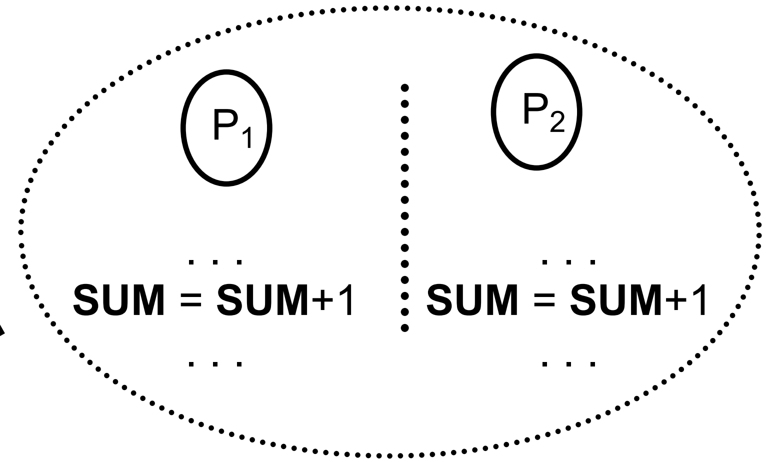
- Memory Isolation
  - ✓ Processes do NOT share memory address space
- Fork/Join Execution Model
  - ✓ Fundamental way of expressing concurrency within a computation
  - ✓ Fork creates a new child process
  - ✓ Parent continues after the **Fork** operation
  - ✓ Child begins operation separate from the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

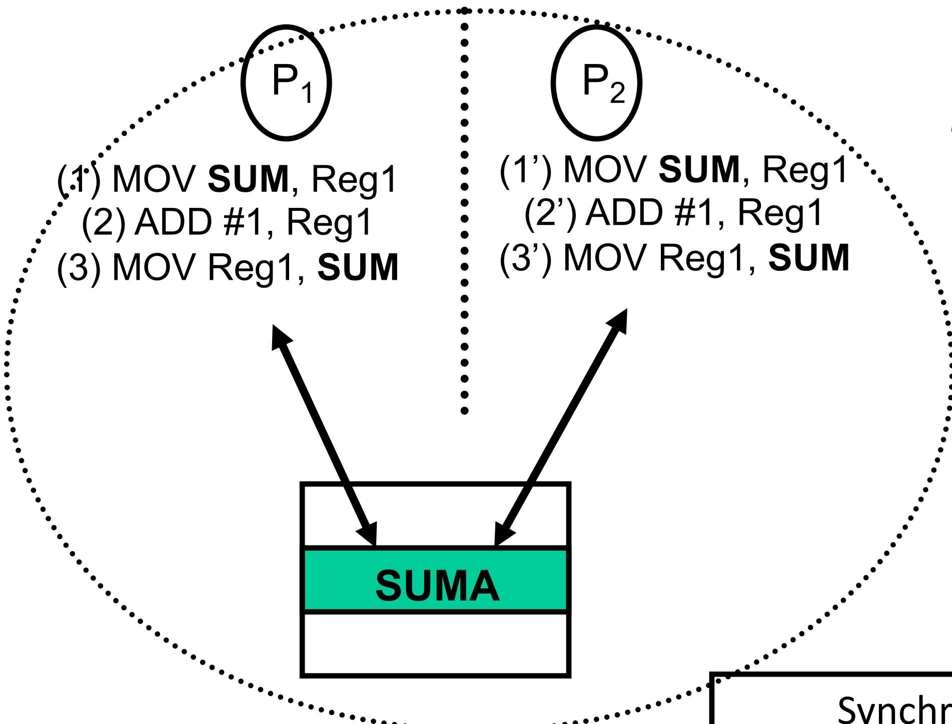


Multi-processing Basics

Race Conditions

A Race Condition occurs, if
- Two or more processes manipulate a shared resource concurrently, and
- The outcome of the execution depends on the particular order in which the access takes place





(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

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 to *S* > 0
  If *S* > 1 then *S* = *S - 1*
Unlock(S):
  S = *S* + *1*

P1: Lock(S)
Critical Section
Unlock(S)

P2: Lock(S)
Critical Section
Unlock(S)

P_1

. . .
Lock(S)
SUM = SUM+1
Unlock(S)
. . .

P_2

. . .
Lock(S)
SUM = SUM+1
Unlock(S)
. . .

(1)(2)(3)(1')(2')(3') => **SUM** = **SUM** +2

(1')(2')(3')(1)(2)(3) => **SUM** = **SUM** +2

Breakout Room

20 minutes

Work on **`example1.py`**, **`example2.py`**, **`example3.py`**, and **`example4.py`**.

If you finish early, feel free to work on the next set of examples with your group.

Process Creation and Synchronization

Join/Fork Model

- The `multiprocessing` module includes a very simple and intuitive API for dividing work between multiple processes

example1.py

```
import multiprocessing
def print_cube(num):
    …
def print_square(num):
    …
if __name__ == "__main__":
    # creating processes
    p1 = multiprocessing.Process(target=print_square, args=(10, ))
    p2 = multiprocessing.Process(target=print_cube, args=(10, ))
    # starting process 1 and 2
    p1.start()
    p2.start()
    # wait until process 1 and 2 are finished
    p1.join()
    p2.join()
    # both processes finished
    print("Done!")
```

Creates a process structure to execute a `target` function with args

Start a process

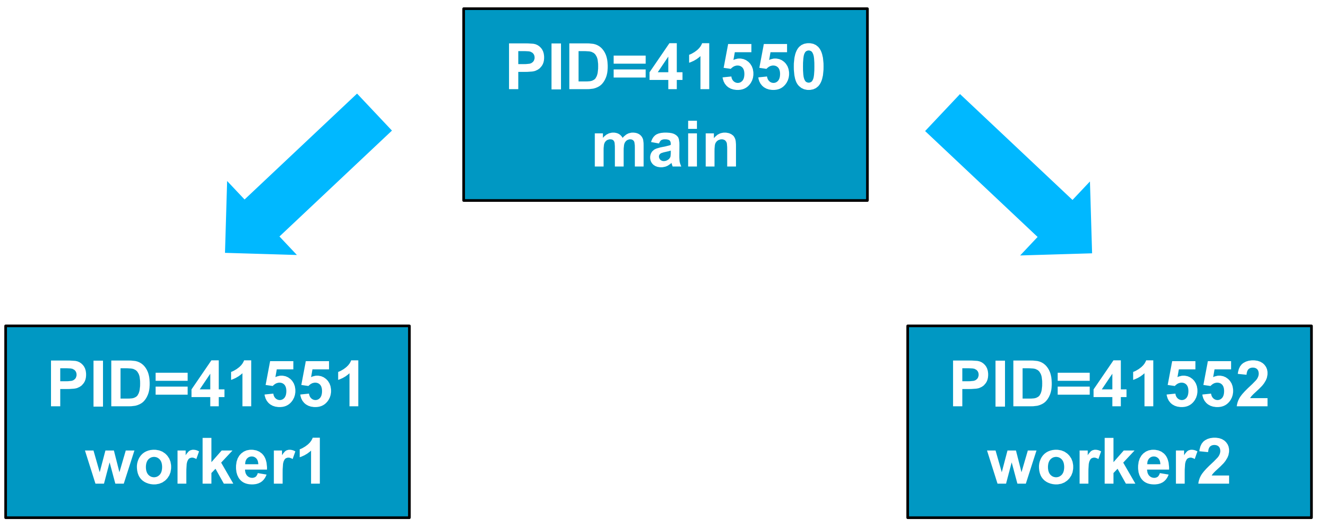
Wait for process termination

Process Creation and Synchronization

Join/Fork Model

- Each process is completely independent

example2.py



Process Communication

Shared Memory

- Each process runs independently and has its own memory space

example3.py

```
import multiprocessing
# empty list with global scope
result = []
def square_list(mylist):
    global result
    # append squares of mylist to global list result
    for num in mylist:
        result.append(num * num)
    # print global list result
    print("Result(in process p1): {}".format(result))
if __name__ == "__main__":
    # input list
    mylist = [1,2,3,4]
    p1 = multiprocessing.Process(target=square_list, args=(mylist,))
    p1.start()
    p1.join()
    # print global result list
    print("Result(in main program): {}".format(result))
```

Process Communication

Shared Memory

`multiprocessing` module provides **`Array`** and **`Value`** objects to share data between processes.

- `Array`: a `ctypes` array allocated from shared memory.
- `Value`: a `ctypes` object allocated from shared memory.

`example4.py`

```
import multiprocessing

def square_list(mylist, result, square_sum):
    …
if __name__ == "__main__":
    …

    # creating Array of int data type with space for 4 integers
    result = multiprocessing.Array('i', 4)
    # creating Value of int data type
    square_sum = multiprocessing.Value('i')
    # creating new process
    p1 = multiprocessing.Process(target=square_list, args=(mylist, result, square_sum))
```

Shared variables passed as arguments

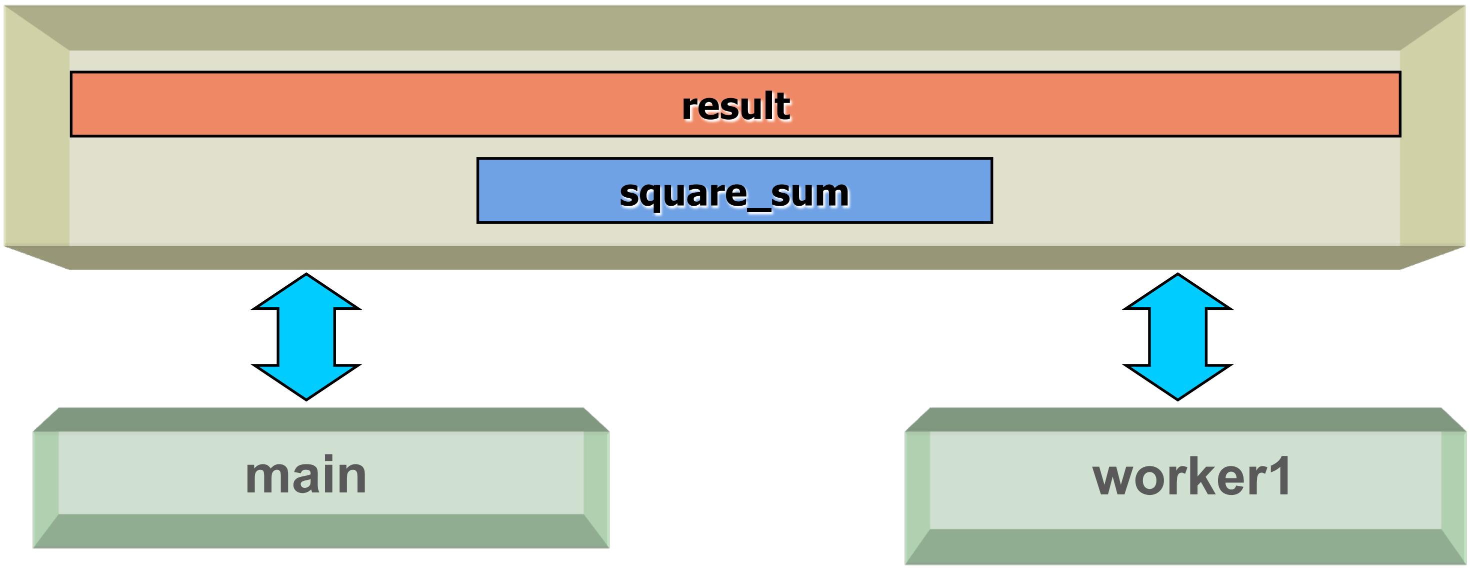
Array `i`ntegers of size `4`

Single variable type `i`ntegers

Process Communication

Shared Memory

Most efficient way to share memory across processes



`multiprocessing` module provides `manager` class (**Advanced!**) that

- Shares arbitrary object types like lists, dictionaries, Queue, Array, etc.
- A single manager can be shared by processes on different computers
- However, they are slower than using shared memory.

Breakout Room

30 minutes

Work on **`example5.py`**, **`example6.py`**, **`example7.py`**, and **`example8.py`**.

Process Communication

Queues

- A simple way to communicate between processes with multiprocessing is to use a Queue to pass messages back and forth. Any Python object can pass through a Queue.

example5.py

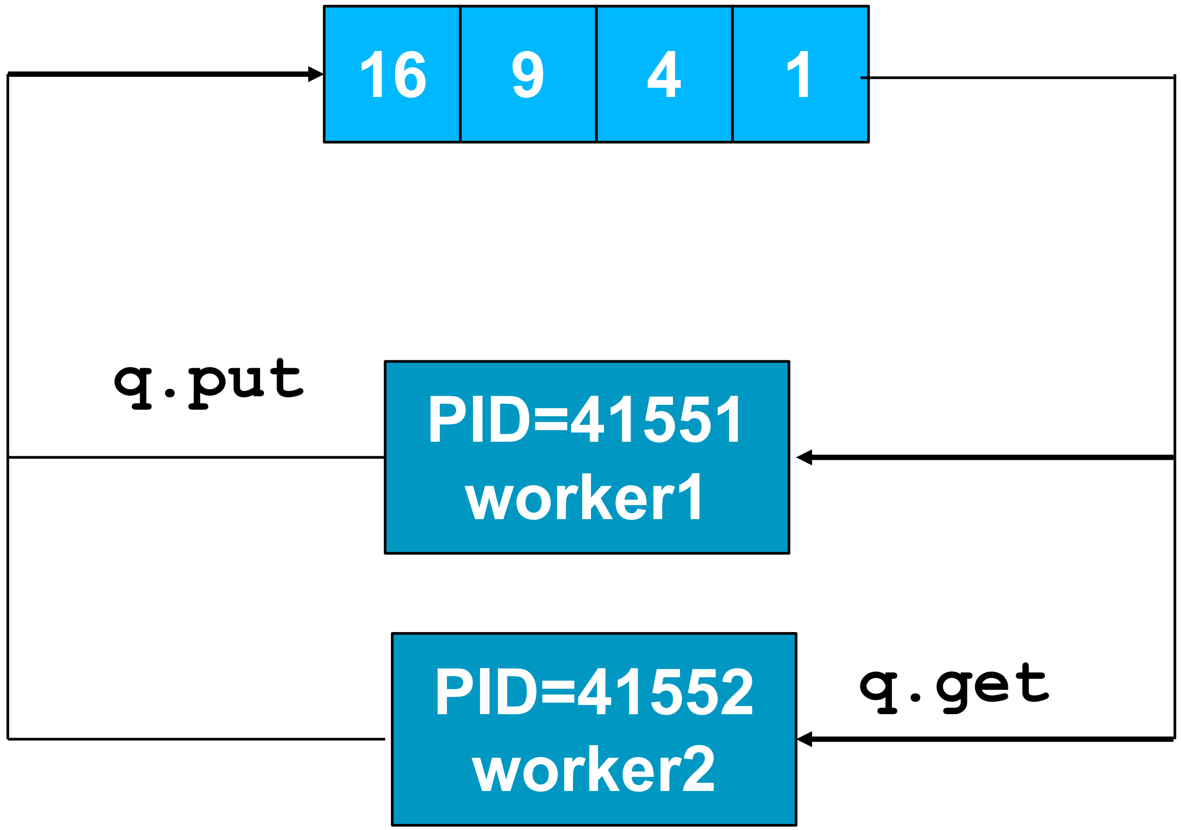
```
def square_list(mylist, q):
    # append squares of mylist to queue
    for num in mylist:
        q.put(num * num)
def print_queue(q):
    print("Queue elements:")
    while not q.empty():
        print(q.get())
    print("Queue is now empty!")
if __name__ == "__main__":
    …
    # creating multiprocessing Queue
    q = multiprocessing.Queue()
    # creating new processes
    p1 = multiprocessing.Process(target=square_list, args=(mylist, q))
    p2 = multiprocessing.Process(target=print_queue, args=(q,))
```

Put value in queue

Read values from queue

Process Communication

Queues



A more efficient two-way communication can be performed with the `pipe` class (**Advanced!**)

Process Synchronization

Race Conditions

- Simultaneous access to shared variable: Why is 100 not the final value?

example6.py

```
# function to withdraw from account
def withdraw(balance):
    for _ in range(10000):
        balance.value = balance.value - 1
# function to deposit to account
def deposit(balance):
    for _ in range(10000):
        balance.value = balance.value + 1
def perform_transactions():
    # initial balance (in shared memory)
    balance = multiprocessing.Value('i', 100)
    # creating new processes
    p1 = multiprocessing.Process(target=withdraw, args=(balance,))
    p2 = multiprocessing.Process(target=deposit, args=(balance,))
```

Process Synchronization

Locks

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

`example7.py`

```
# function to withdraw from account
def withdraw(balance, lock):
    for _ in range(10000):
        lock.acquire()
        balance.value = balance.value - 1
        lock.release()
…
…
    # initial balance (in shared memory)
    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))
```

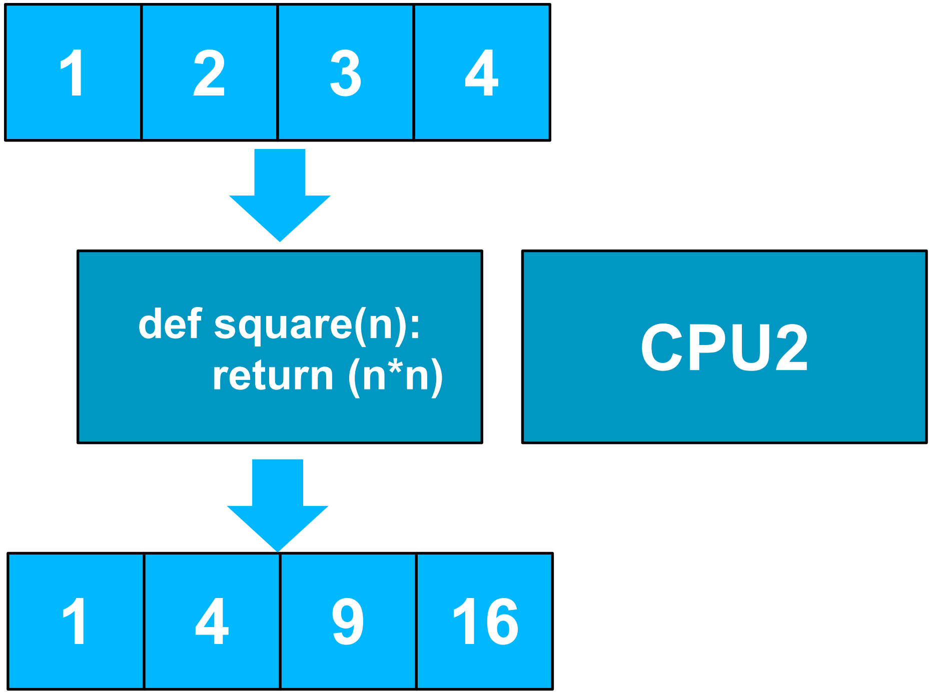
Work Distribution

Pools

- Simple serial program to calculate squares of elements of a given list

example8.py

```
# Python program to find  squares of numbers in a given list
def square(n):
     return (n*n)
if __name__ == "__main__":
     # input list
     mylist = [1,2,3,4,5]
     # empty list to store result
     result = []
     for num in mylist:
          result.append(square(num))
     print(result)
```



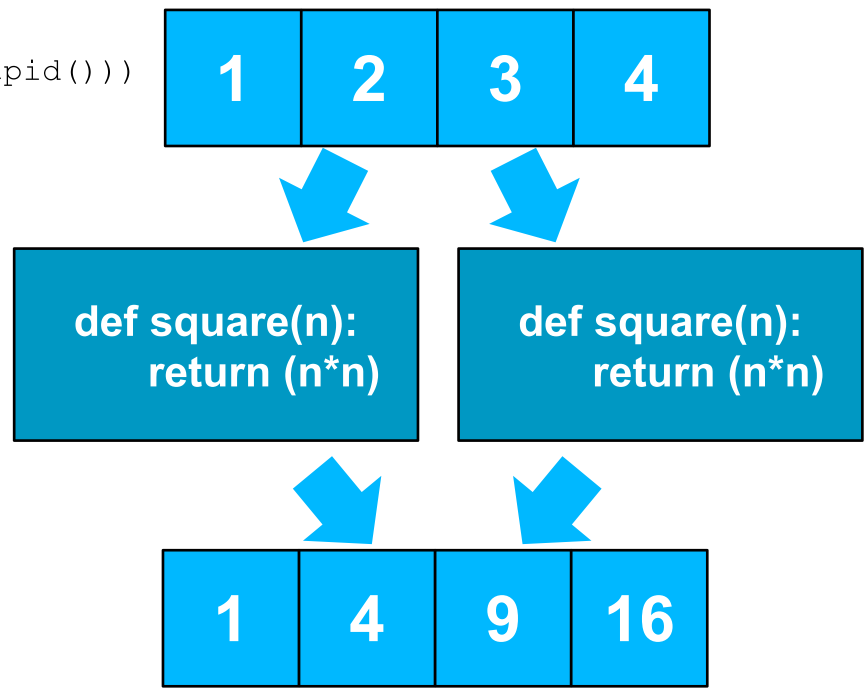
Work Distribution

Pools

- `multiprocessing` module provides a **`Pool`** class that represents a pool of worker processes. It has methods which allows tasks to be offloaded to the worker processes in a few different ways

example9.py

```
# Python program to find  squares of numbers in a given list
import multiprocessing
import os
def square(n):
    print(”Id for {0}: {1}".format(n, os.getpid()))
    return (n*n)
if __name__ == "__main__":
    # input list
    mylist = [1,2,3,4,5]
    # creating a pool object
    p = multiprocessing.Pool()
    # map list to target function
    result = p.map(square, mylist)
```



Breakout Room

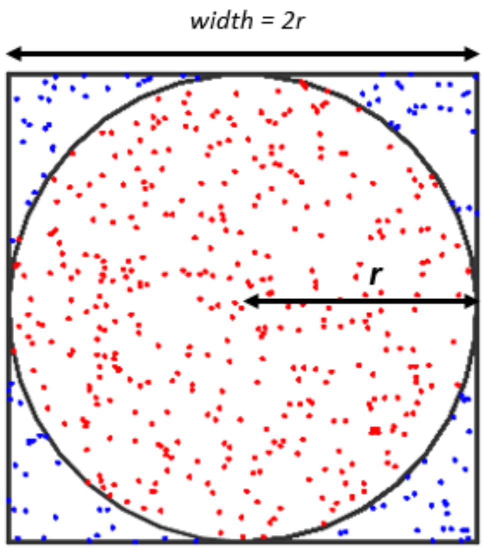
Work Until Called Back

Work on **`pi1.py`**.

Bonus: Have a look at **`pi2.py`**. It is current written as a serial code. How would you go about parallelizing it with the tools you learned today?

Examples

Example: pi using the Monte Carlo Method



pi1.py

Area of the circle is πr^2
The area of the square is $4r^2$
If we divide the area of circle by the area of square we get $\pi/4$
Same ratio can be used between the number of points within the square and the number of points within the circle.
Hence we can use the following formula to estimate Pi:

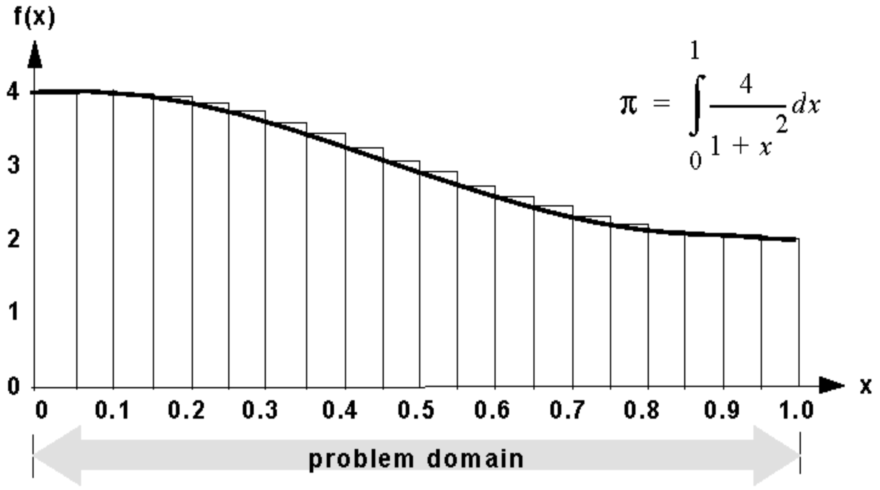
π ≈ 4 x (number of points in the circle / total number of points)

What is the speed-up in your system?

Example

Exercise: pi Using Numerical Approximation to Integral

pi2.py



$$\pi = \int_0^1 \frac{4}{1 + x^2} dx$$

Questions

Python Multiprocessing

Some of the examples have been downloaded from https://www.geeksforgeeks.org/ (Nikhil Kumar)