Multiprocessing in Python

In multiprocessing, any newly created process will do following:

- run independently
- have their own memory space.

Demos: multiprocessing/session1.py

```
import multiprocessing

# empty list with global scope
result = []

def square_list(mylist):
    """
    function to square a given list
    """
    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]

    # creating new process
    p1 = multiprocessing.Process(target=square_list, args=(mylist,))
    # starting process
    p1.start()
    # wait until process is finished
    p1.join()

    # print global result list
    print("Result(in main program): {}".format(result))
```

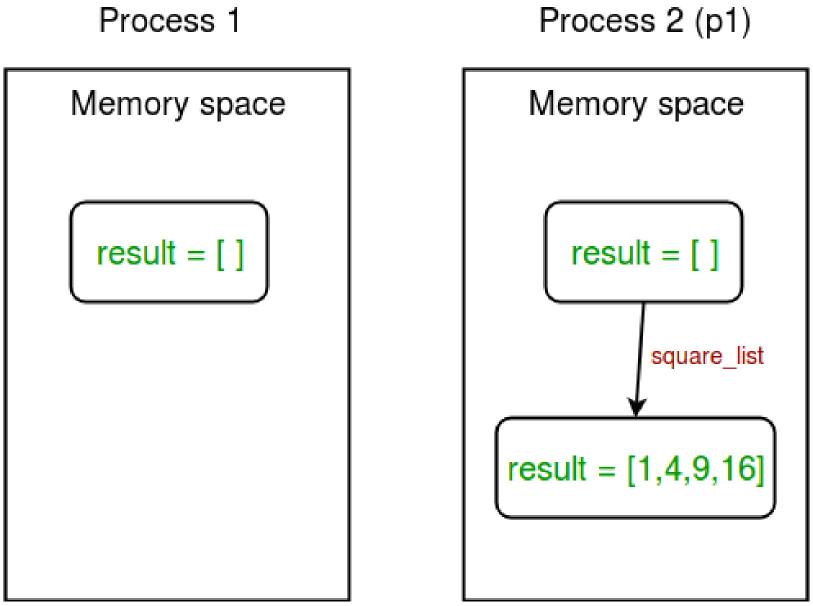
```
Result(in process p1): [1, 4, 9, 16]

Result(in main program): []
```

In above example, we print contents of global list **result** at two places:

- In **square_list** function. Since, this function is called by process **p1**, **result** list is changed in memory space of process **p1** only.

- After the completion of process **p1** in main program. Since main program is run by a different process, its memory space still contains the empty **result** list.

Diagram shown below clears this concept:



Sharing data between processes

1. **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**.

    Use of **Array** and **Value** for sharing data between processes.

    Demos: Multiprocessing/session2.py

```
import multiprocessing

def square_list(mylist, result, square_sum):
    """
    function to square a given list
    """
    # append squares of mylist to result array
```

```
    for idx, num in enumerate(mylist):
        result[idx] = num * num

    # square_sum value
    square_sum.value = sum(result)

    # print result Array
    print("Result(in process p1): {}".format(result[:]))

    # print square_sum Value
    print("Sum of squares(in process p1): {}".format(square_sum.value))

if __name__ == "__main__":
    # input list
    mylist = [1,2,3,4]

    # 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))

    # starting process
    p1.start()

    # wait until process is finished
    p1.join()

    # print result array
    print("Result(in main program): {}".format(result[:]))

    # print square_sum Value
    print("Sum of squares(in main program): {}".format(square_sum.value))
```

```
Result(in process p1): [1, 4, 9, 16]

Sum of squares(in process p1): 30

Result(in main program): [1, 4, 9, 16]

Sum of squares(in main program): 30
```

- First of all, we create an Array **result** like this:
- `result = multiprocessing.Array('i', 4)`

- First argument is the **data type**. ‘i’ stands for integer whereas ‘d’ stands for float data type.
- Second argument is the **size** of array. Here, we create an array of 4 elements.
-

Similarly, we create a Value **square_sum** like this:

```
square_sum = multiprocessing.Value('i')
```

Here, we only need to specify data type. The value can be given an initial value(say 10) like this:

```
square_sum = multiprocessing.Value('i', 10)
```

- Secondly, we pass **result** and **square_sum** as arguments while creating **Process** object.

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

- **result** array elements are given a value by specifying index of array element.

```
for idx, num in enumerate(mylist):
    result[idx] = num * num
```

square_sum is given a value by using its **value** attribute:

```
square_sum.value = sum(result)
```

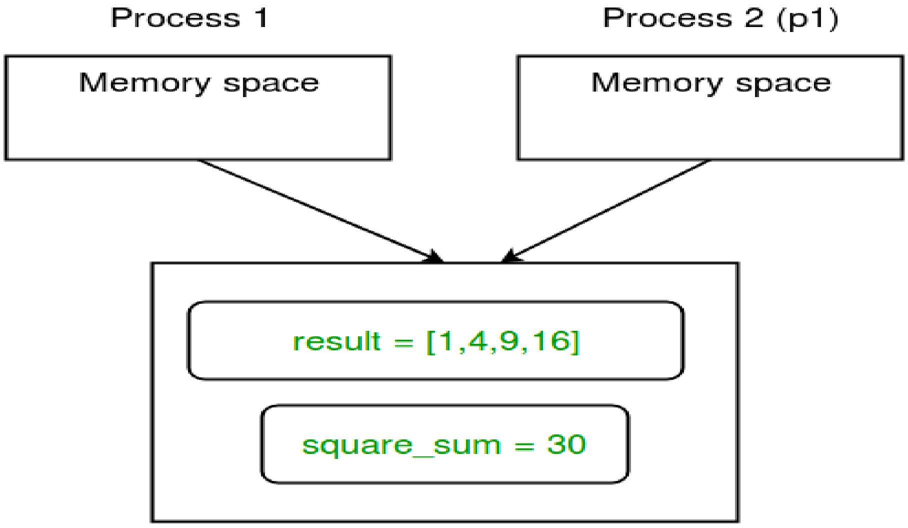
- In order to print **result** array elements, we use **result[:]** to print complete array.

```
print("Result(in process p1): {}".format(result[:]))
```

Value of **square_sum** is simply printed as:

```
print("Sum of squares(in process p1): {}".format(square_sum.value))
```

Diagram depicting how processes share **Array** and **Value** object:



Shared Memory

2. **Server process :** Whenever a python program starts, a **server process** is also started. From there on, whenever a new process is needed, the parent process connects to the server and requests it to fork a new process.

   A **server process** can hold Python objects and allows other processes to manipulate them using proxies.

   multiprocessing module provides a **Manager** class which controls a server process.

   Hence, managers provide a way to create data which can be shared between different processes.

   *Server process managers are more flexible than using **shared memory** objects because they can be made to support arbitrary object types like lists, dictionaries, Queue, Value, Array, etc. Also, a single manager can be shared by processes on different computers over a network. They are, however, slower than using shared memory.*

   Demos: multiprocessing/session3.py

```
import multiprocessing

def print_records(records):
    """
    function to print record(tuples) in records(list)
    """
    for record in records:
        print("Name: {0}\nScore: {1}\n".format(record[0], record[1]))
```

```python
def insert_record(record, records):
    """
    function to add a new record to records(list)
    """
    records.append(record)
    print("New record added!\n")

if __name__ == '__main__':
    with multiprocessing.Manager() as manager:
        # creating a list in server process memory
        records = manager.list([('Sam', 10), ('Adam', 9), ('Kevin',9)])
        # new record to be inserted in records
        new_record = ('Jeff', 8)

        # creating new processes
        p1 = multiprocessing.Process(target=insert_record, args=(new_record, records))
        p2 = multiprocessing.Process(target=print_records, args=(records,))

        # running process p1 to insert new record
        p1.start()
        p1.join()

        # running process p2 to print records
        p2.start()
        p2.join()
```

```
New record added!


Name: Sam

Score: 10


Name: Adam

Score: 9


Name: Kevin

Score: 9


Name: Mallika

Score: 8
```

- We create a **manager** object using:
- with multiprocessing.Manager() as manager:

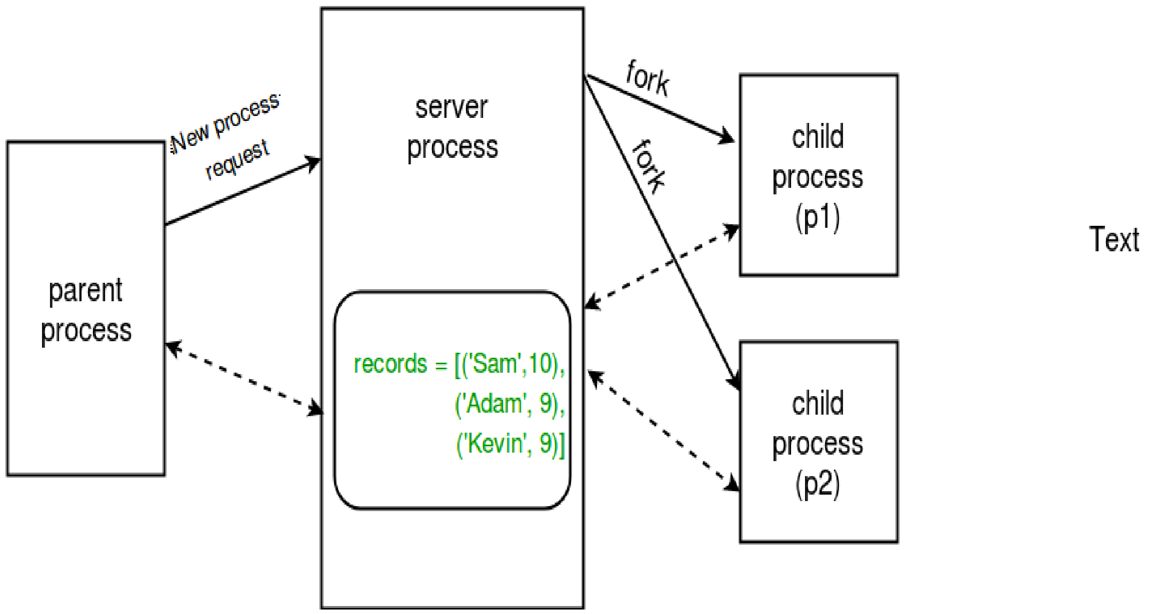
All the lines under **with** statement block are under the scope of **manager** object.

- Then, we create a list **records** in **server process** memory using:
- records = manager.list([('Sam', 10), ('Adam', 9), ('Kevin',9)])

Similarly, you can create a dictionary as **manager.dict** method.

- Finally, we create to processes **p1** (to insert a new record in **records** list) and **p2** (to print **records**) and run them while passing **records** as one of the arguments.
-

The concept of **server process** is depicted in the diagram:



Communication between processes

Effective use of multiple processes usually requires some communication between them, so that work can be divided and results can be aggregated.

multiprocessing supports two types of communication channel between processes:

- **Queue**
- **Pipe**

1. **Queue :** A simple way to communicate between process with multiprocessing is to use a Queue to pass messages back and forth. Any Python object can pass through a Queue.

   Note: The **multiprocessing.Queue** class is a near clone of **queue.Queue**.

   Demos: multiprocessing/session4.py

```
import multiprocessing

def square_list(mylist, q):
    """
    function to square a given list
    """
    # append squares of mylist to queue
    for num in mylist:
        q.put(num * num)

def print_queue(q):
    """
    function to print queue elements
    """
    print("Queue elements:")
    while not q.empty():
        print(q.get())
    print("Queue is now empty!")

if __name__ == "__main__":
    # input list
    mylist = [1,2,3,4]

    # 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,))
```

```
    # running process p1 to square list
    p1.start()
    p1.join()

    # running process p2 to get queue elements
    p2.start()
    p2.join()
```

```
Queue elements:

1

4

9

16

Queue is now empty!
```

- Firstly, we create a **multiprocessing Queue** using:

```
q = multiprocessing.Queue()
```

- Then we pass empty queue **q** to **square_list** function through process **p1**.

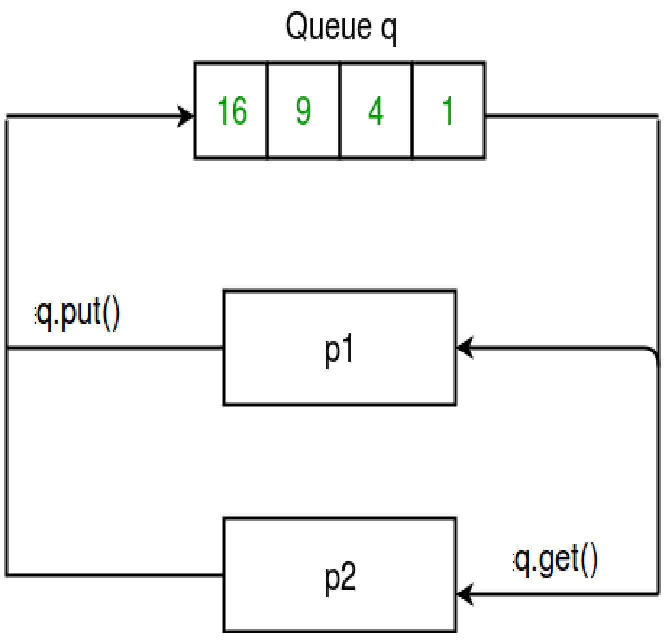
  Elements are inserted to queue using **put** method.

```
q.put(num * num)
```

- In order to print queue elements, we use **get** method until queue is not empty.

```
while not q.empty():
    print(q.get())
```

Daigram depicting the operations on queue:



2. **Pipes :** A pipe can have only two endpoints. Hence, it is preferred over queue when only two-way communication is required.

   multiprocessing module provides **Pipe()** function which returns a pair of connection objects connected by a pipe. The two connection objects returned by **Pipe()** represent the two ends of the pipe. Each connection object has **send()** and **recv()** methods (among others).

```
import multiprocessing

def sender(conn, msgs):
    """
    function to send messages to other end of pipe
    """
    for msg in msgs:
        conn.send(msg)
        print("Sent the message: {}".format(msg))
    conn.close()

def receiver(conn):
    """
    function to print the messages received from other
    end of pipe
    """
    while 1:
        msg = conn.recv()
        if msg == "END":
            break
        print("Received the message: {}".format(msg))

if __name__ == "__main__":
    # messages to be sent
    msgs = ["hello", "hey", "hru?", "END"]

    # creating a pipe
    parent_conn, child_conn = multiprocessing.Pipe()

    # creating new processes
    p1 = multiprocessing.Process(target=sender, args=(parent_conn,msgs))
    p2 = multiprocessing.Process(target=receiver, args=(child_conn,))

    # running processes
    p1.start()
    p2.start()

    # wait until processes finish
```

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

```
Sent the message: hello
Sent the message: hey
Sent the message: hru?
Received the message: hello
Sent the message: END
Received the message: hey
Received the message: hru?
```

- A pipe was created simply using:

```
parent_conn, child_conn = multiprocessing.Pipe()
```

  The function returned two connection objects for the two ends of the pipe.

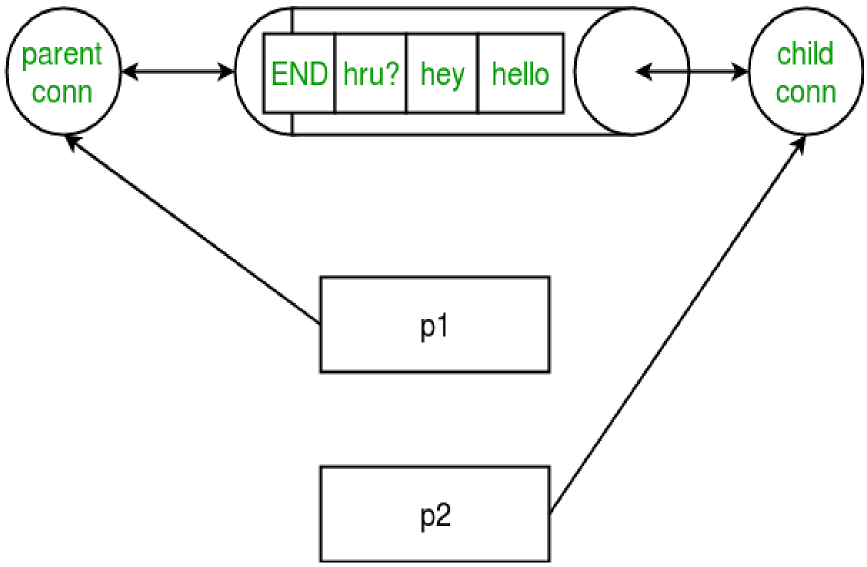
- Message is sent from one end of pipe to another using **send** method.

```
conn.send(msg)
```

- To receive any messages at one end of a pipe, we use **recv** method.

```
msg = conn.recv()
```

- In above program, we send a list of messages from one end to another. At the other end, we read messages until we receive "END" message.

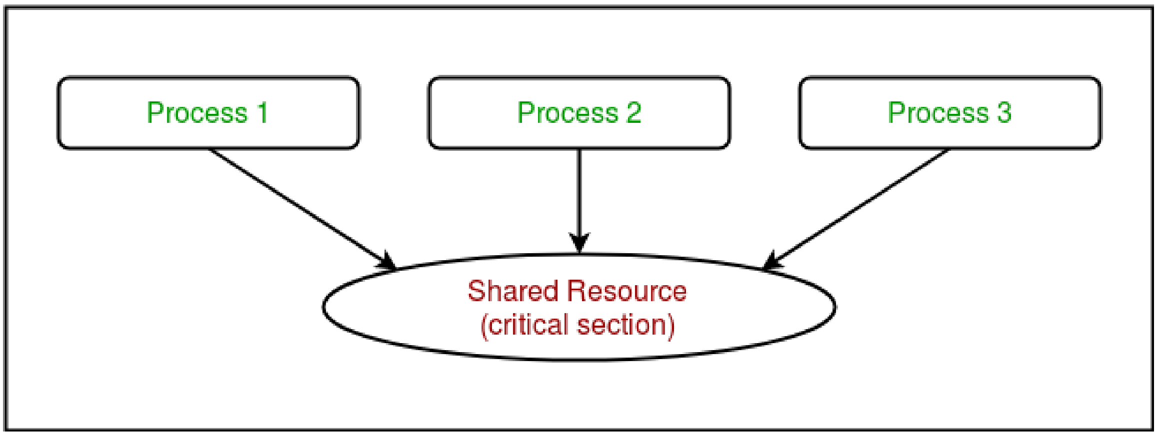


Synchronization between processes

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

Critical section refers to the parts of the program where the shared resource is accessed.

3 processes try to access shared resource or critical section at the same time.



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

```
# Python program to illustrate
# the concept of race condition
# in multiprocessing
import multiprocessing

# 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,))

    # starting processes
    p1.start()
    p2.start()

    # wait until processes are finished
    p1.join()
    p2.join()

    # print final balance
    print("Final balance = {}".format(balance.value))

if __name__ == "__main__":
    for _ in range(10):

        # perform same transaction process 10 times
        perform_transactions()
```

If you run above program, you will get some unexpected values like this:

```
Final balance = 1311

Final balance = 199

Final balance = 558

Final balance = -2265

Final balance = 1371

Final balance = 1158

Final balance = -577
```

```
Final balance = -1300

Final balance = -341

Final balance = 157
```

In above program, 10000 withdraw and 10000 deposit transactions are carried out with initial balance as 100. The expected final balance is 100 but what we get in 10 iterations of **perform_transactions** function is some different values.

This happens due to concurrent access of processes to the shared data **balance**. This unpredictability in balance value is nothing but **race condition**.

- This is a possible sequence which gives wrong answer as both processes read the same value and write it back accordingly.

P1	P2	BALANCE
read(balance) current=100		100
	read(balance) current=100	100
balance=current-1=99 **write(balance)**		99
	balance=current+1=101 **write(balance)**	101

- These are 2 possible sequences which are desired in above scenario.

P1	P2	BALANCE
read(balance) current=100		100
balance=current-1=99 **write(balance)**		99
	read(balance) current=99	99
	balance=current+1=100 **write(balance)**	100

P1	P2	BALANCE
	read(balance) current=100	100
	balance=current+1=101 **write(balance)**	101
read(balance) current=101		101
balance=current-1=100 **write(balance)**		100

Using 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.

A semaphore is a synchronization object that controls access by multiple processes 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 can check and then change. Depending on the value that is found, the process 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 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.*

Demos: multiprocessing/session7.py

```
# Python program to illustrate
# the concept of locks
# in multiprocessing
import multiprocessing

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

# function to deposit to account
def deposit(balance, lock):
    for _ in range(10000):
        lock.acquire()
        balance.value = balance.value + 1
        lock.release()

def perform_transactions():

    # 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))

    # starting processes
    p1.start()
    p2.start()

    # wait until processes are finished
    p1.join()
    p2.join()

    # print final balance
    print("Final balance = {}".format(balance.value))

if __name__ == "__main__":
    for _ in range(10):

        # perform same transaction process 10 times
        perform_transactions()
```

Output:

```
Final balance = 100

Final balance = 100

Final balance = 100

Final balance = 100

Final balance = 100

Final balance = 100

Final balance = 100

Final balance = 100

Final balance = 100

Final balance = 100
```

- Firstly, a **Lock** object is created using:
- lock = multiprocessing.Lock()
- Then, **lock** is passed as target function argument:
- p1 = multiprocessing.Process(target=withdraw, args=(balance,lock))
- p2 = multiprocessing.Process(target=deposit, args=(balance,lock))
- In the critical section of target function, we apply lock using **lock.acquire()** method. As soon as a lock is acquired, no other process can access its critical section until the lock is released using **lock.release()** method.
- lock.acquire()
- balance.value = balance.value - 1
- lock.release()

As you can see in the results, the final balance comes out to be 100 every time (which is the expected final result).

Pooling between processes

Let us consider a simple program to find squares of numbers in a given list.

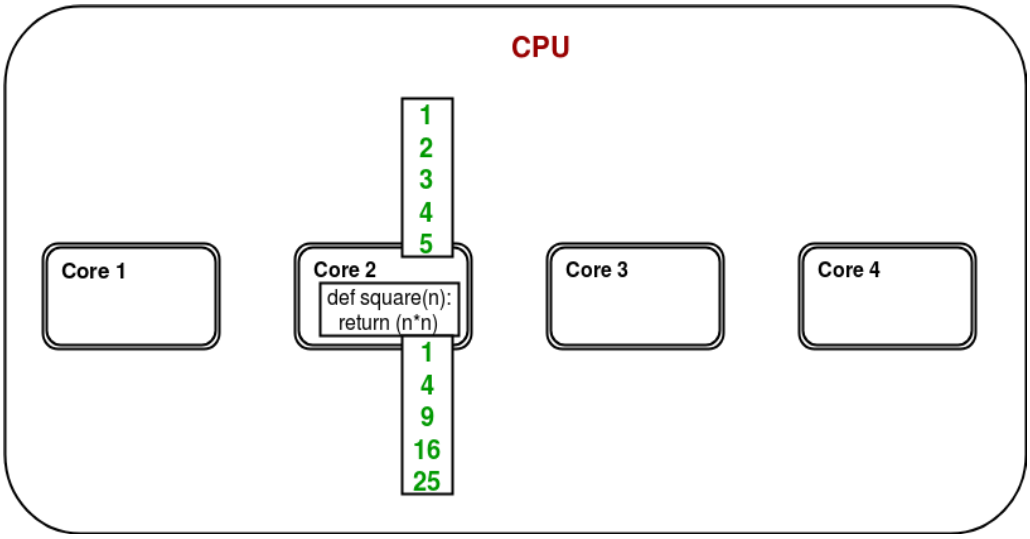
```
# Python program to find
# squares of numbers in a given list
def square(n):
    return (n*n)

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)
```

Output:

```
[1, 4, 9, 16, 25]
```

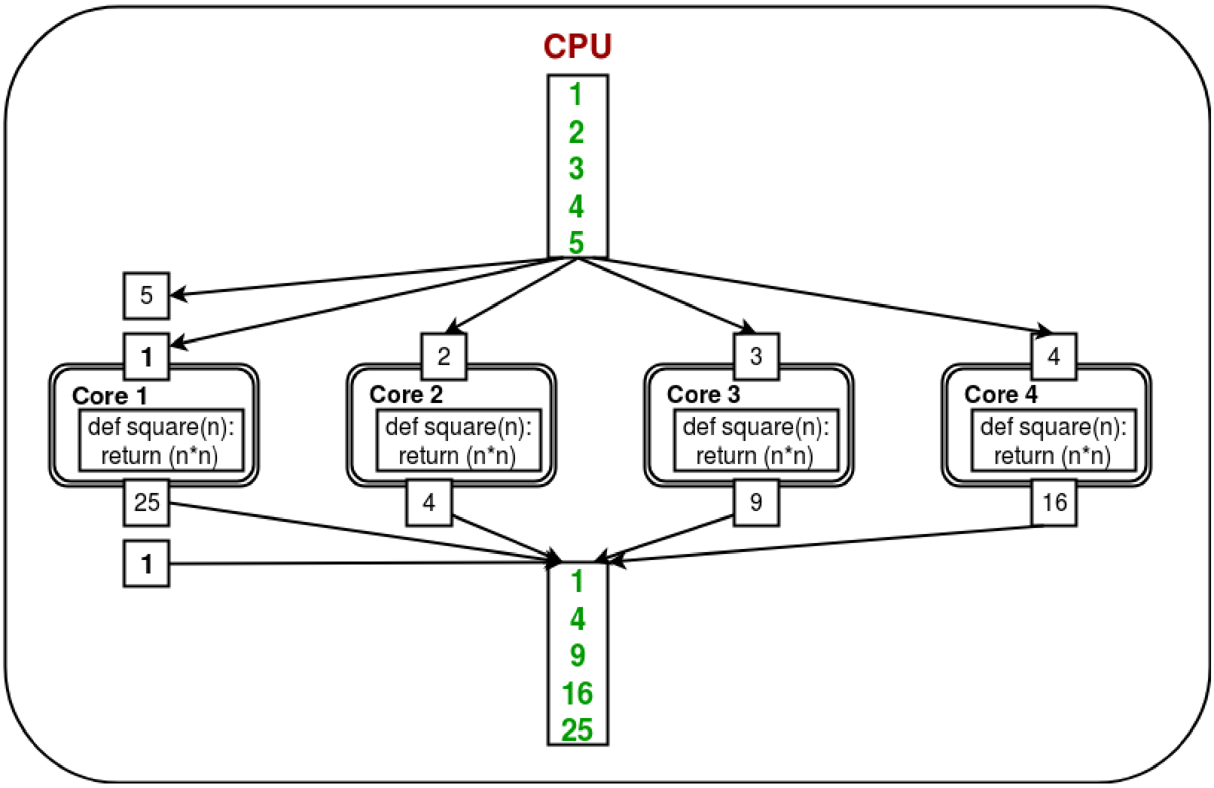
It is a simple program to calculate squares of elements of a given list. In a multi-core/multi-processor system, consider the diagram below to understand how above program will work:



Only one of the cores is used for program execution and it's quite possible that other cores remain idle.

In order to utilize all the cores, **multiprocessing** module provides a **Pool** class.

The **Pool** class represents a pool of worker processes. It has methods which allows tasks to be offloaded to the worker processes in a few different ways. Consider the diagram below:



Here, the task is offloaded/distributed among the cores/processes automatically by **Pool** object. User doesn't need to worry about creating processes explicitly.

Demos: multiprocessing/session

```
# Python program to understand
# the concept of pool
import multiprocessing
import os

def square(n):
    print("Worker process 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)

print(result)
```

Output:

```
Worker process id for 2: 4152

Worker process id for 1: 4151

Worker process id for 4: 4151

Worker process id for 3: 4153

Worker process id for 5: 4152

[1, 4, 9, 16, 25]
```

Let us try to understand above code step by step:

- We create a **Pool** object using:
- ```
  p = multiprocessing.Pool()
  ```

  There are a few arguments for gaining more control over offloading of task. These are:

  - **processes:** specify the number of worker processes.
  - **maxtasksperchild:** specify the maximum number of task to be assigned per child.

  All the processes in a pool can be made to perform some initialization using these arguments:

  - **initializer:** specify an initialization function for worker processes.
  - **initargs:** arguments to be passed to initializer.
- Now, in order to perform some task, we have to map it to some function. In the example above, we map **mylist** to **square** function. As a result, the contents of **mylist** and definition of **square** will be distributed among the cores.
- ```
  result = p.map(square, mylist)
  ```
- Once all the worker processes finish their task, a list is returned with the final result

END