multiprocessing

importing required libraries and our shared library

In [1]:
```
import ctypes
import multiprocessing
import os
import time
```

In [2]:
```
_libInC = ctypes.CDLL('./libMyLib.so')
```

Here, we slightly adjust our Python wrapper to calculate the results and print it. There is also some additional casting to ensure that the result of the *libInC.myAdd()* is an int32 type.

In [3]:
```
def addC_print(_i, a, b, time_started):
    val = ctypes.c_int32(_libInC.myAdd(a, b)).value #cast the result to a 32 bit
    end_time = time.time()
    print('CPU_{} Add: {} in {}'.format(_i, val, end_time - time_started))

def multC_print(_i, a, b, time_started):
    val = ctypes.c_int32(_libInC.myMult(a, b)).value #cast the result to a 32 bi
    end_time = time.time()
    print('CPU_{} Multiply: {} in {}'.format(_i, val, end_time - time_started))
```

Now for the fun stuff.

The multiprocessing library allows us to run simultaneous code by utilizing multiple processes. These processes are handled in separate memory spaces and are not restricted to the Global Interpreter Lock (GIL).

Here we define two proceses, one to run the _addC*print* and another to run the _multC*print()* wrappers.

Next we assign each process to be run on difference CPUs

In [9]:
```
procs = [] # a future list of all our processes

# Launch process1 on CPU0
p1_start = time.time()
p1 = multiprocessing.Process(target=addC_print, args=(0, 3, 5, p1_start)) # the
os.system("taskset -p -c {} {}".format(0, p1.pid)) # taskset is an os command to
p1.start() # start the process
procs.append(p1)

# Launch process2 on CPU1
p2_start = time.time()
p2 = multiprocessing.Process(target=multC_print, args=(1, 3, 5, p2_start)) # the
os.system("taskset -p -c {} {}".format(1, p2.pid)) # taskset is an os command to
p2.start() # start the process
procs.append(p2)
```

```
p1Name = p1.name # get process1 name
p2Name = p2.name # get process2 name

# Here we wait for process1 to finish then wait for process2 to finish
p1.join() # wait for process1 to finish
print('Process 1 with name, {}, is finished'.format(p1Name))

p2.join() # wait for process2 to finish
print('Process 2 with name, {}, is finished'.format(p2Name))
```

```
CPU_0 Add: 8 in 1.0571329593658447
CPU_1 Multiply: 15 in 1.0447323322296143
Process 1 with name, Process-11, is finished
Process 2 with name, Process-12, is finished
```

Return to 'main.c' and change the amount of sleep time (in seconds) of each function.

For different values of sleep(), explain the difference between the results of the ‘Add’ and ‘Multiply’ functions and when the Processes are finished.

Lab work

One way around the GIL in order to share memory objects is to use multiprocessing objects. Here, we're going to do the following.

1. Create a multiprocessing array object with 2 entries of integer type.
2. Launch 1 process to compute addition and 1 process to compute multiplication.
3. Assign the results to separate positions in the array.
    A. Process 1 (add) is stored in index 0 of the array (array[0])
    B. Process 2 (mult) is stored in index 1 of the array (array[1])
4. Print the results from the array.

Thus, the multiprocessing Array object exists in a *shared memory* space so both processes can access it.

Array documentation:

https://docs.python.org/2/library/multiprocessing.html#multiprocessing.Array

typecodes/types for Array:

'c': ctypes.c_char

'b': ctypes.c_byte

'B': ctypes.c_ubyte

'h': ctypes.c_short

'H': ctypes.c_ushort

'i': ctypes.c_int

'I': ctypes.c_uint

'l': ctypes.c_long

'L': ctypes.c_ulong

'f': ctypes.c_float

'd': ctypes.c_double

Try to find an example

You can use online reources to find an example for how to use multiprocessing Array

In [14]:

```
def addC_no_print(_i, a, b, returnValus):
    '''
    Params:
      _i    : Index of the process being run (0 or 1)
      a, b : Integers to add
      returnValues : Multiprocessing array in which we will store the result at
    '''
    val = ctypes.c_int32(_libInC.myAdd(a, b)).value
    # TODO: add code here to pass val to correct position returnValues
    returnValus[_i] = val

def multC_no_print(_i, a, b, returnValus):
    '''
    Params:
      _i    : Index of the process being run (0 or 1)
      a, b : Integers to multiply
      returnValues : Multiprocessing array in which we will store the result at
    '''
    val = ctypes.c_int32(_libInC.myMult(a, b)).value
    # TODO: add code here to pass val to correct position of returnValues
    returnValus[_i] = val

procs = []

# TODO: define returnValues here. Check the multiprocessing docs to see
# about initializing an array object for 2 processes.
# Note the data type that will be stored in the array

returnValues = multiprocessing.Array('i',2)


p1 = multiprocessing.Process(target=addC_no_print, args=(0, 3, 5, returnValues))
os.system("taskset -p -c {} {}".format(0, p1.pid)) # taskset is an os command to
p1.start() # start the process
procs.append(p1)

p2 = multiprocessing.Process(target=multC_no_print, args=(1, 3, 5, returnValues)
os.system("taskset -p -c {} {}".format(1, p2.pid)) # taskset is an os command to
```

```
p2.start() # start the process
procs.append(p2)

# Wait for the processes to finish
for p in procs:
    pName = p.name # get process name
    p.join() # wait for the process to finish
    print('{} is finished'.format(pName))

# TODO print the results that have been stored in returnValues
print('CPU_0:{}'.format(returnValues[0]))
print('CPU_1:{}'.format(returnValues[1]))
```

```
Process-17 is finished
Process-18 is finished
CPU_0:8
CPU_1:15
```

In [ ]: