multiprocessing

importing required libraries and our shared library

In [23]:
```
import ctypes
import multiprocessing
import os
import time
```

In [24]:
```
_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 [25]:
```
def addC_print(_i, a, b, time_started):
    val = ctypes.c_int32(_libInC.myadd(a, b)).value #cast the result to a 32 b
    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 b
    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 processes, 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 [26]:
```
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 t
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)) # th
os.system("taskset -p -c {} {}".format(1, p2.pid)) # taskset is an os command t
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 1.064201831817627
CPU_1 Multiply: 15 in 1.045560359954834
Process 1 with name, Process-17, is finished
Process 2 with name, Process-18, 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
(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 [1]:
```
import multiprocessing
import ctypes
import os

# Step a: Create a multiprocessing array object with 2 entries of integer type
# TODO: Define the multiprocessing array with two entries of integer type
returnValues = multiprocessing.Array(ctypes.c_int, 2)

def addC_no_print(_i, a, b, returnValues):
    val = ctypes.c_int32(a + b).value
    # Step c(i): Assign the result to index 0 of the array
    # TODO: Assign 'val' to the correct position in 'returnValues'
    returnValues[_i] = val

def multC_no_print(_i, a, b, returnValues):
    val = ctypes.c_int32(a * b).value
    # Step c(ii): Assign the result to index 1 of the array
    # TODO: Assign 'val' to the correct position in 'returnValues'
    returnValues[_i] = val

# Step b: Launch 1 process to compute addition and 1 process to compute multipl
procs = []

# Launch process for addition (Process-0)
p1 = multiprocessing.Process(target=addC_no_print, args=(0, 3, 5, returnValues)
p1.start()
procs.append(p1)

# Launch process for multiplication (Process-1)
p2 = multiprocessing.Process(target=multC_no_print, args=(1, 3, 5, returnValue
p2.start()
procs.append(p2)

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

# Step e: Print the results from the array
# TODO: Print the results stored in 'returnValues'
print('Result from array index 0:', returnValues[0])
print('Result from array index 1:', returnValues[1])
```

```
Process-1 is finished
Process-2 is finished
Result from array index 0: 8
Result from array index 1: 15
```

In [ ]: