ctypes

The following imports ctypes interface for Python

```
In [2]: import ctypes
```

Now we can import our shared library

```
In [3]: _libInC = ctypes.CDLL('./libMyLib.so')
```

Let's calll our C function, myAdd(a, b).

```
In [3]: _libInC.myAdd(3, 5)
```

```
Out[3]: 8
```

This is cumbersome to write, so let's wrap this C function in a Python function for ease of use.

```
In [4]: def addC(a,b):
            return _libInC.myAdd(a,b)
```

Usage example:

```
In [5]: addC(10, 202)
```

```
Out[5]: 212
```

Multiply

Following the code for your add function, write a Python wrapper function to call your C multiply code

```
In [6]: _libInC.myMULt(3,5)
```

```
Out[6]: 15
```

```
In [ ]:
```

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 [6]:
```
def addC_print(_i, a, b, time_started):
    val = ctypes.c_int32(_libInC.myAdd(a, b)).value #cast the result to a 32 bit integ
    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 bit inte
    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 [8]:
```
def addC_print(_i, a, b, time_started):
    val = ctypes.c_int32(_libInC.myAdd(a, b)).value  # cast the result to a 32 bit int
    end_time = time.time()
    print('Process {}, CPU_{} Add: {} in {}'.format(multiprocessing.current_process().

def multC_print(_i, a, b, time_started):
    val = ctypes.c_int32(_libInC.myMULt(a, b)).value  # cast the result to a 32 bit ir
    end_time = time.time()
    print('Process {}, CPU_{} Multiply: {} in {}'.format(multiprocessing.current_proce

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))
p1.start() # start the process first
```

```python
os.system("taskset -p -c {} {}".format(0, p1.pid)) # then set the CPU affinity
procs.append(p1)

# Launch process2 on CPU1
p2_start = time.time()
p2 = multiprocessing.Process(target=multC_print, args=(1, 3, 5, p2_start))
p2.start() # start the process first
os.system("taskset -p -c {} {}".format(1, p2.pid)) # then set the CPU affinity
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))
```

```
pid 22748's current affinity list: 0,1
pid 22748's new affinity list: 0
pid 22752's current affinity list: 0,1
pid 22752's new affinity list: 1
Process Process-5, CPU_0 Add: 8 in 20.110741138458252
Process 1 with name, Process-5, is finished
Process Process-6, CPU_1 Multiply: 15 in 40.12253165245056
Process 2 with name, Process-6, 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 [13]:
```
# Step 3: Define functions to compute addition and multiplication
def addC_no_print(_i, a, b, returnValues):
    '''
    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 index
    '''
    val = ctypes.c_int32(_libInC.myAdd(a, b)).value
    returnValues[_i] = val  # Store the result in the array at the correct index

def multC_no_print(_i, a, b, returnValues):
    '''
    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 index
    '''
    val = ctypes.c_int32(_libInC.myMULt(a, b)).value
    returnValues[_i] = val  # Store the result in the array at the correct index

# Step 1: Define returnValues as a multiprocessing array with 2 entries of integer typ
returnValues = multiprocessing.Array('i', 2)  # 'i' is the typecode for ctypes.c_int

# Create an empty list to hold the processes
```

```
procs = []

# Step 2: Launch process1 (addition) on CPU0
p1 = multiprocessing.Process(target=addC_no_print, args=(0, 3, 5, returnValues))  # th
p1.start()  # Start the process to get a valid PID
os.system("taskset -p -c {} {}".format(0, p1.pid))  # Set the CPU affinity after the p
procs.append(p1)

# Step 2: Launch process2 (multiplication) on CPU1
p2 = multiprocessing.Process(target=multC_no_print, args=(1, 3, 5, returnValues))  # t
p2.start()  # Start the process to get a valid PID
os.system("taskset -p -c {} {}".format(1, p2.pid))  # Set the CPU affinity after the p
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))

# Step 4: Print the results that have been stored in returnValues
print("Addition result (array[0]):", returnValues[0])  # Result from process1 (add)
print("Multiplication result (array[1]):", returnValues[1])  # Result from process2 (m
```

```
pid 25501's current affinity list: 0,1
pid 25501's new affinity list: 0
pid 25504's current affinity list: 0,1
pid 25504's new affinity list: 1
Process-9 is finished
Process-10 is finished
Addition result (array[0]): 8
Multiplication result (array[1]): 15
```

```
In [ ]:
```

threading

importing required libraries and programing our board

In [1]:
```python
import threading
import time
from pynq.overlays.base import BaseOverlay
base = BaseOverlay("base.bit")
```

Two threads, single resource

Here we will define two threads, each responsible for blinking a different LED light. Additionally, we define a single resource to be shared between them.

When thread0 has the resource, led0 will blink for a specified amount of time. Here, the total time is 50 x 0.02 seconds = 1 second. After 1 second, thread0 will release the resource and will proceed to wait for the resource to become available again.

The same scenario happens with thread1 and led1.

In [2]:
```python
def blink(t, d, n):
    '''
    Function to blink the LEDs
    Params:
      t: number of times to blink the LED
      d: duration (in seconds) for the LED to be on/off
      n: index of the LED (0 to 3)
    '''
    for i in range(t):
        base.leds[n].toggle()
        time.sleep(d)
    base.leds[n].off()

def worker_t(_l, num):
    '''
    Worker function to try and acquire resource and blink the LED
    _l: threading lock (resource)
    num: index representing the LED and thread number.
    '''
    for i in range(4):
        using_resource = _l.acquire(True)
        print("Worker {} has the lock".format(num))
        blink(50, 0.02, num)
        _l.release()
        time.sleep(0) # yeild
    print("Worker {} is done.".format(num))

# Initialize and launch the threads
threads = []
```

```
fork = threading.Lock()
for i in range(2):
    t = threading.Thread(target=worker_t, args=(fork, i))
    threads.append(t)
    t.start()

for t in threads:
    name = t.getName()
    t.join()
    print('{} joined'.format(name))
```

```
Worker 0 has the lock
/tmp/ipykernel_1678/1470821917.py:37: DeprecationWarning: getName() is deprecated, ge
t the name attribute instead
  name = t.getName()
Worker 0 has the lock
Worker 0 has the lock
Worker 0 has the lock
Worker 0 is done.
Worker 1 has the lock
Thread-5 (worker_t) joined
Worker 1 has the lock
Worker 1 has the lock
Worker 1 has the lock
Worker 1 is done.
Thread-6 (worker_t) joined
```

Two threads, two resource

Here we examine what happens with two threads and two resources trying to be shared between them.

The order of operations is as follows.

The thread attempts to acquire resource0. If it's successful, it blinks 50 times x 0.02 seconds = 1 second, then attemps to get resource1. If the thread is successful in acquiring resource1, it releases resource0 and procedes to blink 5 times for 0.1 second = 1 second.

In [3]:
```
def worker_t(_l0, _l1, num):
    '''
    Worker function to try and acquire resource and blink the LED
    _l0: threading lock0 (resource0)
    _l1: threading lock1 (resource1)
    num: index representing the LED and thread number.
    init: which resource this thread starts with (0 or 1)
    '''
    using_resource0 = False
    using_resource1 = False

    for i in range(4):
        using_resource0 = _l0.acquire(True)
        if using_resource1:
            _l1.release()
        print("Worker {} has lock0".format(num))
        blink(50, 0.02, num)
        using_resource1 = _l1.acquire(True)
        if using_resource0:
            _l0.release()
        print("Worker {} has lock1".format(num))
        blink(5, 0.1, num)

        time.sleep(0) # yeild
    print("Worker {} is done.".format(num))

# Initialize and launch the threads
threads = []
fork = threading.Lock()
fork1 = threading.Lock()
for i in range(2):
    t = threading.Thread(target=worker_t, args=(fork, fork1, i))
    threads.append(t)
    t.start()

for t in threads:
    name = t.getName()
    t.join()
    print('{} joined'.format(name))
```

```
Worker 0 has lock0
/tmp/ipykernel_1678/2415493498.py:38: DeprecationWarning: getName() is deprecated, ge
t the name attribute instead
  name = t.getName()
Worker 0 has lock1
Worker 1 has lock0
KeyboardInterrupt
```

You may have notied (even before running the code) that there's a problem! What happens when thread0 has resource1 and thread1 has resource0! Each is waiting for the other to release their resource in order to continue.

This is a **deadlock**. Adjust the code above to prevent a deadlock.

Non-blocking Acquire

In the above code, when *l.acquire(True)* was used, the thread stopped executing code and waited for the resource to be acquired. This is called **blocking**: stopping the execution of code and waiting for something to happen. Another example of **blocking** is if you use *input()* in Python. This will stop the code and wait for user input.

What if we don't want to stop the code execution? We can use non-blocking version of the acquire() function. In the code below, _resource*available* will be True if the thread currently has the resource and False if it does not.

Complete the code to and print and toggle LED when lock is not available.

In [ ]:
```
def blink(t, d, n):
    for i in range(t):
        base.leds[n].toggle()
        time.sleep(d)
    base.leds[n].off()

def worker_t(_l, num):
    for i in range(10):
        resource_available = _l.acquire(False)  # this is non-blocking acquire
        if resource_available:
            # Print message for having the key
            print('Worker {} has the key.'.format(num))
            # Blink for a while
            blink(5, 0.5, num)
            # Release the key
            _l.release()
            # Give enough time to the other thread to grab the key
            time.sleep(1)
        else:
            # Print message for waiting for the key
            print('Worker {} is waiting for the key.'.format(num))
            # Blink for a while with a different rate
            blink(5, 1, num)
            # The timing between having the key + yield and waiting for the key shoul

    print('Worker {} is done.'.format(num))

threads = []
fork = threading.Lock()
for i in range(2):
    t = threading.Thread(target=worker_t, args=(fork, i))
    threads.append(t)
    t.start()

for t in threads:
    name = t.name  # Use the `name` attribute instead of `getName()`
    t.join()
    print('{} joined'.format(name))
```

```
Worker 0 has the key.Worker 1 is waiting for the key.

Worker 0 has the key.
Worker 1 is waiting for the key.
Worker 0 has the key.
Worker 1 has the key.
Worker 0 is waiting for the key.
Worker 1 has the key.
Worker 0 is waiting for the key.
Worker 1 has the key.
Worker 0 has the key.
Worker 1 is waiting for the key.
Worker 0 has the key.
Worker 1 is waiting for the key.
Worker 0 has the key.
Worker 1 has the key.
Worker 0 is waiting for the key.
```

```
In [ ]:
```