WES 237A: Introduction to Embedded System Design (Winter 2024)
Lab 2: Process and Thread, Due: 1/22/2024 11:59pm

Ricardo Lizarraga, rlizarraga0@gmail.com , *619.252.4157*	**PID: A69028483** **https://github.com/RiLizarraga**
Demo videos files at: https://drive.google.com/drive/folders/1jYQ4IE-Q7_aljBgwKIEFN_wpze6EYsMB?usp=sharing ctypes_example_RLS.ipynb multiprocess_example_RLS_v2.ipynb threading_example__RLS	

https://www.bogotobogo.com/python/Multithread/python_multithreading_Synchronization_Semaphore_Objects_Thread_Pool.php

https://docs.python.org/3/library/threading.html
https://docs.python.org/3/library/ctypes.html
https://medium.com/@mliuzzolino/wrapping-c-with-python-in-5-minutes-cdd1124f5c01
In order to report and reflect on your WES 237A labs, please complete this Post-Lab report by the end of the weekend by submitting the following 2 parts:

- < > Upload your lab 2 report composed by a **single PDF that includes your in-lab answers** to the bolded questions in the Google Doc Lab and your **Jupyter Notebook code**. You could either scan your written copy, or simply type your answer in this Google Doc. **However, please make sure your responses are readable.**
- < > Answer two short essay-like questions on your Lab experience.

All responses should be submitted to Canvas. Please also be sure to push your code to your git repo as well.

Create Lab2 Folder

1. Create a new folder on your PYNQ jupyter home and rename it 'Lab2'

Shared C++ Library

1. In 'Lab2', create a new text file (New -> Text File) and rename it to 'main.c'
2. Add the following code to 'main.c':

```
#include <unistd.h>

int myAdd(int a, int b){
    sleep(1);
    return a+b;
}
```

```
#include<unistd.h>

int myAdd(int a, int b){
        sleep(1);
        return a+b;
}
```

3. **Following the function above, write another function to multiply two integers together. Copy your code below.**

```
int myMult(int a, int b){
        sleep(3);
        return a*b;
}
```

4. Save main.c
5. In Jupyter, open a terminal window (New -> Terminal) and *change directories* (cd) to 'Lab2' directory.

$ cd Lab2

6. Compile your 'main.c' code as a shared library.

$ gcc -c -Wall -Werror -fpic main.c
$ gcc -shared -o libMyLib.so main.o

7. Download 'ctypes_example.ipynb' from here and upload it to the Lab2 directory.
8. Go through each of the code cells to understand how we interface between Python and our C code
9. **Write another Python function to wrap your multiplication function written above in step 3. Copy your code below.**

ctypes

The following imports ctypes interface for Python

In [1]:
```
import ctypes
```

Now we can import our shared library

In [2]:
```
_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 [12]:
```
import ctypes
_libInC = ctypes.CDLL('./libMyLib.so')

```

In [13]:
```
_libInC.myMult(3, 5)
def MultC(a,b):
    return _libInC.myMult(a,b)
```

In [14]:
```
MultC(3,5)
```

Out[14]: 15

To summarize, we created a C shared library and then called the C function from Python

Multiprocessing

1. Download ‘multiprocess_example.ipynb’ from [here]() and upload it into your ‘Lab2’ directory.
2. Go through the documentation (and comments) and answer the following question
    a. **Why does the ‘Process-#’ keep incrementing as you run the code cell over and over?**

The process number will keep incrementing, regardless the status of older processes, the reason of this is because the operating system on the PID management portion; the PID number will always increment 1 (+=1) to the next value, until reaches the MAX_VALUE, then goes back to initiate counting again.

    b. **Which line assigns the processes to run on a specific CPU?**

p1 = multiprocessing.Process(target=addC_print, args=(0, 3, 5, p1_start)) # the first arg defines which CPU to run the 'target' on
p2 = multiprocessing.Process(target=multC_print, args=(1, 3, 5, p2_start)) # the first arg defines which CPU to run the 'target' on

3. In ‘main.c’ change the ‘sleep()’ command and recompile the library with the commands above. Also reload the jupyter notebook with the □ symbol and re-run all cells. Try sleeping the functions for various, different times (or the same).

```
#include<unistd.h>

int myAdd(int a, int b){
        sleep(1);
        return a+b;
}

int myMult(int a, int b){
        sleep(3);
        return a*b;
}
```

    a. **Explain the difference between the results of the ‘Add’ and ‘Multiply’ functions and when the processes are finished.**

Execution time for myMult() function changed to 3 seconds
CPU_0 Add: 8 in 1.0728306770324707
Process 1 with name, Process-1, is finished
CPU_1 Multiply: 15 in 3.0643014907836914
Process 2 with name, Process-2, is finished

4. Continue to the lab work section. Here we are going to do the following
    a. Create <u>a multiprocessing array object</u> with <u>2 entries of integer type</u>.
    b. Launch 1 process to compute addition and 1 process to compute multiplication.
    c. Assign the results to separate positions in the array.
        i. Process 1 (add) is stored in index 0 of the array (array[0])
        ii. Process 2 (mult) is stored in index 1 of the array (array[1])
    d. Print the results from the array.
    e. **There are 4 TODO comments that must be completed**
5. Answer the following question
    a. **Explain, in your own words, what shared memory is relating to the code in this exercise.**

A single array of multiprocessing processes is created and initialized (and started) with data and reference of 2 processes; all this happened with a simple for-loop, this simplifies the coding.
Shared memory between the processes are input variable as arguments and returned values organized in array format,
In this example a number array of ctypes objects are modified by a child process
For the return values, we had to define and declare an array of double numbers

```
returnValues = Array(c_double,[0, 0], lock=False)
```

And passed on when instanced the each of the new processes

```
for i in range(2):#0,1
    process_start = time.time()
    p = multiprocessing.Process(target=functionsAry[i], args=(cpusAry[i], a, b, returnValues))
```

In [10]:

```python
## My Test code: Multiprocessing Array object
import ctypes
import multiprocessing
import os
import time
from multiprocessing import Process, Lock
from multiprocessing.sharedctypes import Value, Array
from ctypes import Structure, c_double
''' COMPILING and LINKING the code
$ gcc -c -Wall -Werror -fpic main.c
$ gcc -shared -o libMyLib.so main.o
'''
_libInC = ctypes.CDLL('./libMyLib.so')#libMyLib.so this file is generate
def addC_no_print(_i, a, b, returnValue):
    '''
    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 re
    '''
    val = ctypes.c_int32(_libInC.myAdd(a, b)).value
    # TODO: add code here to pass val to correct position returnValues
    returnValues[_i] = val

def multC_no_print(_i, a, b, returnValue):
    '''
    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 re
    '''
    val = ctypes.c_int32(_libInC.myMult(a, b)).value
    print("value = "+str(val))
    # TODO: add code here to pass val to correct position of returnValue
    returnValues[_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 = Array(c_double,[0, 0], lock=False)

cpusAry=[0, 1];a = 3;b = 5;functionsAry = [addC_no_print, multC_no_print
for i in range(2):#0,1
    process_start = time.time()
    p = multiprocessing.Process(target=functionsAry[i], args=(cpusAry[i]
    os.system("taskset -p -c {} {}".format(cpusAry[i], p.pid)) # taskset
    p.start() # start the process
    procs.append(p)
    print('Process: {}, PID: {} Started'.format(p.name, p.pid))
# 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
for i in range(2):
    print('Return Value {} = {}'.format(i, returnValues[i]))
```

```
taskset: invalid PID argument: 'None'
```

```
Process: Process-17, PID: 4961 Started
```

```
taskset: invalid PID argument: 'None'
```

```
Process: Process-18, PID: 4964 Started
Process-17 is finished
value = 15
Process-18 is finished
Return Value 0 = 8.0
Return Value 1 = 15.0
```

Threading

1. Download 'threading_example.ipynb' from [here] and upload it into your 'Lab2' directory.
2. Go through the documentation and code for 'Two threads, single resource' and answer the following questions
    a. **What line launches a thread and what function is the thread executing?**

Line launches the thread:
t = threading.Thread(target=worker_t, args=(fork, i))
Function being executed is:
worker_t

    b. **What line defines a mutual resource? How is it accessed by the thread function?**

Mutual resource:
def blink(t, d, n):
Then the mutual resource is accessed by locking it, then Release it to make it available for other thread or process:

```
using_resource = _l.acquire(True)
print("Worker {} has the lock".format(num))
blink(50, 0.02, num)
_l.release()
```

3. Answer the following question about the 'Two threads, two resources' section.
    a. **Explain how this code enters a deadlock.**

I added trace mssges to see timeline of events, and is obvious that Worker 1 doesn't release and Worker 0 needs the resourse and still waiting
Worker 0 has lock0
*Worker 0 has Released lock0
Worker 1 has lock0
Worker 0 has lock1

4. Complete the code using the non-blocking acquire function.
    a. **What is the difference between 'blocking' and 'non-blocking' functions?**

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.

In [1]:

```python
import threading
import time
from pynq.overlays.base import BaseOverlay
base = BaseOverlay("base.bit")
```

In [14]:

```python
 '''
    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)
 '''
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 ac
        if resource_available:
            # write code to:
            # print message for having the key
            # blink for a while
            # release the key
            # give enough time to the other thread to grab the key
            #_l.acquire(True)
            print("Worker {} has lock".format(num))
            blink(5, 0.02, num)
            _l.release()
            print("Worker {} has Released lock".format(num))
            time.sleep(0.15) # yield
        else:
            # write code to:
            # print message for waiting for the key
            # the timing between having the key + yield and waiting for
            print("Worker {} waiting for resource to be available".forma
            time.sleep(0.2) # yield
    print('worker {} is done.'.format(num))

threads = []
fork = threading.Lock()
for i in range(2):
    t = threading.Thread(target=worker_t, args=(fork, i))
    print('Worker {}, {} Started.'.format(i,t.name))
    threads.append(t)
    t.start()

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

```
Worker 0, Thread-29 (worker_t) Started.
Worker 0 has lock
Worker 1, Thread-30 (worker_t) Started.
Worker 1 waiting for resource to be available
Worker 0 has Released lock
Worker 1 has lock
Worker 0 waiting for resource to be available
Worker 1 has Released lock
Worker 0 has lock
Worker 1 waiting for resource to be available
Worker 0 has Released lock
Worker 1 has lock
Worker 0 waiting for resource to be available
Worker 1 has Released lock
Worker 0 has lock
Worker 1 waiting for resource to be available
Worker 0 has Released lock
Worker 1 has lock
Worker 0 waiting for resource to be available
Worker 1 has Released lock
Worker 0 has lock
Worker 1 waiting for resource to be available
Worker 0 has Released lock
Worker 1 has lock
Worker 0 waiting for resource to be available
Worker 1 has Released lock
Worker 0 has lock
Worker 1 waiting for resource to be available
Worker 0 has Released lock
Worker 1 has lock
Worker 0 waiting for resource to be available
Worker 1 has Released lock
worker 0 is done.
Thread-29 (worker_t) joined
worker 1 is done.
Thread-30 (worker_t) joined
```

ctypes

The following imports ctypes interface for Python

In [1]:
```
import ctypes
```

Now we can import our shared library

In [2]:
```
_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 [12]:
```
import ctypes
_libInC = ctypes.CDLL('./libMyLib.so')
```

In [13]:
```
_libInC.myMult(3, 5)
def MultC(a,b):
    return _libInC.myMult(a,b)
```

In [14]:
```
MultC(3,5)
```

Out[14]:
```
15
```

multiprocessing

importing required libraries and our shared library

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

In [ ]:
```
''' COMPILING and LINKING the code
$ gcc -c -Wall -Werror -fpic main.c
$ gcc -shared -o libMyLib.so main.o
'''
_libInC = ctypes.CDLL('./libMyLib.so')#libMyLib.so this file is generated during LINKI
```

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 [ ]:
```
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 [ ]:
```
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 first
os.system("taskset -p -c {} {}".format(0, p1.pid)) # taskset is an os command to pin 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)) # the first
```

```
os.system("taskset -p -c {} {}".format(1, p2.pid)) # taskset is an os command to pin 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))
```

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.

In [ ]:

RLS notes

```
                taskset is an os command to pin the process to a
specific CPU
```

root@pynq:/home/xilinx/jupyter_notebooks/RLS/Lab2# taskset -h Usage: taskset [options] [mask | cpu-list] [pid|cmd [args...]] Show or change the CPU affinity of a process.

Options: -a, --all-tasks operate on all the tasks (threads) for a given pid -p, --pid operate on existing given pid -c, --cpu-list display and specify cpus in list format -h, --help display this help -V, --version display version

The default behavior is to run a new command: taskset 03 sshd -b 1024 You can retrieve the mask of an existing task: taskset -p 700 Or set it: taskset -p 03 700 List format uses a comma-separated list instead of a mask: taskset -pc 0,3,7-11 700 Ranges in list format can take a stride argument: e.g. 0-31:2 is equivalent to mask 0x55555555

For more details see taskset(1).

In [ ]:

```
## My Test code: Multiprocessing, linear programming
import ctypes
import multiprocessing
import os
import time

''' COMPILING and LINKING the code
$ gcc -c -Wall -Werror -fpic main.c
$ gcc -shared -o libMyLib.so main.o
'''
_libInC = ctypes.CDLL('./libMyLib.so')#libMyLib.so this file is generated during LINKI
```

```
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('.', end='')
    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('.', end='')
    print('CPU_{} Multiply: {} in {}'.format(_i, val, end_time - time_started))

procs = [] # a future list of all our processes

# Launch process1 on CPU0
cpu1 = 0;
a = 3;b = 5
p1_start = time.time()
p1 = multiprocessing.Process(target=addC_print, args=(cpu1, a, b, p1_start)) # the fir
os.system("taskset -p -c {} {}".format(0, p1.pid)) # taskset is an os command to pin t
p1.start() # start the process
procs.append(p1)

# Launch process2 on CPU1
cpu2 = 1;
a = 3;b = 5
p2_start = time.time()
p2 = multiprocessing.Process(target=multC_print, args=(cpu2, a, b, p2_start)) # the fi
os.system("taskset -p -c {} {}".format(1, p2.pid)) # taskset is an os command to pin t
p2.start() # start the process
procs.append(p2)

p1Name = p1.name # get process1 name
p2Name = p2.name
p1PID = p1.pid # get process1 PID
p2PID = p2.pid

# Here we wait for process1 to finish then wait for process2 to finish
p1.join() # wait for process1 to finish, join(): Block until all items in the queue ha
print('Process 1 with name: {}, PID: {} is finished'.format(p1Name, p1PID))#;print('P1

p2.join() # wait for process2 to finish
print('Process 2 with name: {}, PID: {} is finished'.format(p2Name, p2PID))#;print('P1
```

In [ ]:

```
## My Test code: Multiprocessing Array object
import ctypes
import multiprocessing
import os
import time
''' COMPILING and LINKING the code
$ gcc -c -Wall -Werror -fpic main.c
$ gcc -shared -o libMyLib.so main.o
'''
_libInC = ctypes.CDLL('./libMyLib.so')#libMyLib.so this file is generated during LINK1

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))

a = 3;b = 5
cpusAry=[0, 1]
functionsAry = [addC_print, multC_print]
procs = []
for i in range(2):#0,1
    process_start = time.time()
    p = multiprocessing.Process(target=functionsAry[i], args=(cpusAry[i], a, b, proces
    os.system("taskset -p -c {} {}".format(cpusAry[i], p.pid)) # taskset is an os comm
    p.start() # start the process
    procs.append(p)
    print('Process: {}, PID: {} Started'.format(p.name, p.pid))

procs[0].join() # wait for process1 to finish, join(): Block until all items in the qu
print('Process 1 with name: {}, PID: {} is finished'.format(procs[0].name, procs[0].pi

procs[1].join() # wait for process2 to finish
print('Process 2 with name: {}, PID: {} is finished'.format(procs[0].name, procs[0].pi
```

In [ ]:

```
cpusAry=[0, 1]
functionsAry = [addC_print, multC_print]
for i in range(2):
    print(i)
    print(functionsAry[i])
    print(cpusAry[i])
    process_start = time.time()
    print(process_start)
```

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 [10]:

```
## My Test code: Multiprocessing Array object
import ctypes
import multiprocessing
import os
import time
from multiprocessing import Process, Lock
from multiprocessing.sharedctypes import Value, Array
from ctypes import Structure, c_double
''' COMPILING and LINKING the code
$ gcc -c -Wall -Werror -fpic main.c
$ gcc -shared -o libMyLib.so main.o
'''
_libInC = ctypes.CDLL('./libMyLib.so')#libMyLib.so this file is generated during LINKI
def addC_no_print(_i, a, b, returnValue):
    '''
    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
    # TODO: add code here to pass val to correct position returnValues
    returnValues[_i] = val

def multC_no_print(_i, a, b, returnValue):
    '''
```

```
    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
    print("value = "+str(val))
    # TODO: add code here to pass val to correct position of returnValues
    returnValues[_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 = Array(c_double,[0, 0], lock=False)

cpusAry=[0, 1];a = 3;b = 5;functionsAry = [addC_no_print, multC_no_print]
for i in range(2):#0,1
    process_start = time.time()
    p = multiprocessing.Process(target=functionsAry[i], args=(cpusAry[i], a, b, returr
    os.system("taskset -p -c {} {}".format(cpusAry[i], p.pid)) # taskset is an os comm
    p.start() # start the process
    procs.append(p)
    print('Process: {}, PID: {} Started'.format(p.name, p.pid))
# 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
for i in range(2):
    print('Return Value {} = {}'.format(i, returnValues[i]))
```

```
taskset: invalid PID argument: 'None'
Process: Process-17, PID: 4961 Started
taskset: invalid PID argument: 'None'
Process: Process-18, PID: 4964 Started
Process-17 is finished
value = 15
Process-18 is finished
Return Value 0 = 8.0
Return Value 1 = 15.0
```

threading

importing required libraries and programing our board

In [ ]:
```
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 [ ]:
```
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))
```

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 [ ]:

```
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 Released lock1\n".format(num))
        print("Worker {} has lock0\n".format(num))
        blink(50, 0.02, num)

        using_resource1 = _l1.acquire(True)
        if using_resource0:
            _l0.release()
            print("*Worker {} has Released lock0\n".format(num))
        print("Worker {} has lock1\n".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))
```

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

In [14]:
```
 '''
    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)
 '''
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, True=wi
        if resource_available:
            # write code to:
```

```
            # print message for having the key
            # blink for a while
            # release the key
            # give enough time to the other thread to grab the key
            #_l.acquire(True)
            print("Worker {} has lock".format(num))
            blink(5, 0.02, num)
            _l.release()
            print("Worker {} has Released lock".format(num))
            time.sleep(0.15) # yield
        else:
            # write code to:
            # print message for waiting for the key
            # the timing between having the key + yield and waiting for the key should
            print("Worker {} waiting for resource to be available".format(num))
            time.sleep(0.2) # yield
    print('worker {} is done.'.format(num))

threads = []
fork = threading.Lock()
for i in range(2):
    t = threading.Thread(target=worker_t, args=(fork, i))
    print('Worker {}, {} Started.'.format(i,t.name))
    threads.append(t)
    t.start()

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

```
Worker 0, Thread-29 (worker_t) Started.
Worker 0 has lock
Worker 1, Thread-30 (worker_t) Started.
Worker 1 waiting for resource to be available
Worker 0 has Released lock
Worker 1 has lock
Worker 0 waiting for resource to be available
Worker 1 has Released lock
Worker 0 has lock
Worker 1 waiting for resource to be available
Worker 0 has Released lock
Worker 1 has lock
Worker 0 waiting for resource to be available
Worker 1 has Released lock
Worker 0 has lock
Worker 1 waiting for resource to be available
Worker 0 has Released lock
Worker 1 has lock
Worker 0 waiting for resource to be available
Worker 1 has Released lock
Worker 0 has lock
Worker 1 waiting for resource to be available
Worker 0 has Released lock
Worker 1 has lock
Worker 0 waiting for resource to be available
Worker 1 has Released lock
Worker 0 has lock
Worker 1 waiting for resource to be available
Worker 0 has Released lock
Worker 1 has lock
Worker 0 waiting for resource to be available
Worker 1 has Released lock
worker 0 is done.
Thread-29 (worker_t) joined
worker 1 is done.
Thread-30 (worker_t) joined
```

In [ ]:

In [ ]:

In [ ]: