

Lab #2 Report and Reflections

ABDULLAH AJLAN

A69028719

WES 237A: Introduction to Embedded System Design (Winter 2024)

Lab 2: Process and Thread

Due: 1/22/2024 11:59pm

Create Lab2 Folder

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

1/21/24, 8:04 PM

Lab2/

Files Running Clusters

Select items to perform actions on them.

Upload New ↕

<input type="checkbox"/> 0	/tree / Lab2 (/tree/Lab2)	Name ↓	Last Modified	File size
	.. (/tree)		seconds ago	
<input type="checkbox"/>	ctypes_example.ipynb (/notebooks/Lab2/ctypes_example.ipynb)	Running	a year ago	3.61 kB
<input type="checkbox"/>	multiprocess_example.ipynb (/notebooks/Lab2/multiprocess_example.ipynb)	Running	a year ago	8.47 kB
<input type="checkbox"/>	threading_example2.ipynb (/notebooks/Lab2/threading_example2.ipynb)	Running	a year ago	13.6 kB
<input type="checkbox"/>	main.c (/edit/Lab2/main.c)		a year ago	159 B
<input type="checkbox"/>	main.o (/edit/Lab2/main.o)		a year ago	976 B

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;  
}
```

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

1/21/24, 8:17 PM

main.c - Jupyter Text Editor

```
1 #include<unistd.h>  
2 int myadd(int a, int b){  
3     sleep(100);  
4     return a+b;  
5 }  
6 #include<unistd.h>  
7 int mymult(int a, int b){  
8     sleep(100);  
9     return a*b;  
10 }  
11
```

Lab #2 Report and Reflections

ABDULLAH AJLAN

A69028719

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.

1/21/24, 8:03 PM

ctypes_example - Jupyter Notebook

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 call our C function, myAdd(a, b).

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

```
Out[5]: 15
```

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

```
In [6]: def addC(a,b):  
        return _libInC.mymult(a,b)
```

Usage example:

```
In [7]: addC(10, 202)
```

```
Out[7]: 2020
```

Multiply

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

```
In [11]: import ctypes
```

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

```
In [13]: _libInC.mymult(3, 5)
```

```
Out[13]: 15
```

```
In [14]: def addC(a,b):  
        return _libInC.mymult(a,b)
```

```
In [15]: addC(10, 202)
```

```
Out[15]: 2020
```

```
In [ ]:
```

Lab #2 Report and Reflections

ABDULLAH AJLAN

A69028719

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?**

each time we create a new multiprocessing.Process instance, it gets assigned a unique name by default. The default naming convention for processes in Python's multiprocessing module is "Process-1", "Process-2", and so on.

In our code, you are creating new process instances in each run of the code cell:

```
p1 = multiprocessing.Process(target=addC_no_print, args=(0, 3, 5, returnValues))  
p1.start()
```


```
p2 = multiprocessing.Process(target=multC_no_print, args=(1, 3, 5, returnValues))  
p2.start()
```

Each time we run this code, new instances of Process are created, and they are given names like "Process-3", "Process-4", and so forth. This is why you see the 'Process-#' incrementing.

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

The lines that assign the processes to run on specific CPUs using the taskset command are as follows:

```
For Process 1 (p1):  
os.system("taskset -p -c {} {}".format(0, p1.pid))  
For Process 2 (p2):  
os.system("taskset -p -c {} {}".format(1, p2.pid))
```

-
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).
 - a. **Explain the difference between the results of the 'Add' and 'Multiply' functions and when the processes are finished.**

Changing the sleep duration in the 'main.c' file and recompiling the library can impact the execution time of the functions and subsequently affect the timing of the multiprocessing code in the Jupyter notebook. The 'sleep()' command introduces a delay in the execution of each function, allowing us to observe how multiprocessing handles concurrent tasks.

the difference in sleep durations affects the execution time of each function, influencing when the processes finish. Short durations result in close completion times, while longer durations lead to more noticeable time differences. The order of completion is influenced by the actual execution time of the functions.

Lab #2 Report and Reflections

ABDULLAH AJLAN

A69028719

4. Continue to the lab work section. Here we are going to do the following
 - a. Create a multiprocessing array object with 2 entries of integer type.
 - 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**

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

Lab #2 Report and Reflections

ABDULLAH AJLAN

A69028719

1/21/24, 8:02 PM

multiprocess_example - Jupyter Notebook

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

Lab #2 Report and Reflections

ABDULLAH AJLAN

A69028719

1/21/24, 8:02 PM

multiprocess_example - Jupyter Notebook

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 resources to find an example for how to use multiprocessing Array

Lab #2 Report and Reflections

ABDULLAH AJLAN

A69028719

1/21/24, 8:02 PM

multiprocess_example - Jupyter Notebook

```
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 multiplication
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, returnValues))
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 []:

Lab #2 Report and Reflections

ABDULLAH AJLAN

A69028719

5. Answer the following question
- Explain, in your own words, what shared memory is relating to the code in this exercise.

Shared memory, in the context of the code in this exercise, refers to a region of memory that is accessible by multiple processes running concurrently. In multiprocessing programming, each process typically has its own memory space, which is isolated from other processes. However, shared memory allows multiple processes to communicate and exchange data by accessing the same region of memory.

In this exercise, a multiprocessing array (returnValues) is used to create shared memory. This array is accessible by both the 'addC_no_print' and 'multC_no_print' processes. When these processes compute the addition and multiplication results, respectively, they store these results in the shared array at specific positions (indexes 0 and 1). As a result, both processes can read and write to the same memory locations in the array.

Threading

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

The line that launches a thread and specifies the function it is executing is:

```
t = threading.Thread(target=worker_t, args=(fork, i))
```

- What line defines a mutual resource? How is it accessed by the thread function?

The line that defines a mutual resource is

```
fork = threading.Lock():
```

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

This code has the potential to enter a deadlock due to the following scenario:

- Both threads start by acquiring one lock each (_l0 and _l1) at the beginning of their execution.
- Let's assume that Thread 0 acquires _l0 and Thread 1 acquires _l1 initially.
- Thread 0 executes the first part of the loop, releases _l1 (if it was acquired), and prints "Worker 0 has lock0."
- Thread 1 executes the first part of the loop, releases _l0 (if it was acquired), and prints "Worker 1 has lock1."
- Now, both threads attempt to acquire the lock that the other thread just released. Thread 0 wants _l1, and Thread 1 wants _l0.
- Both threads are blocked waiting for the other to release the lock they need. This situation results in a deadlock because neither thread can make progress.

Lab #2 Report and Reflections

ABDULLAH AJLAN

A69028719

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

1. Blocking (`_l.acquire(True)`)

- In the `worker_t` function, when `_l.acquire(True)` is used, it is a blocking acquire.
- If the lock is not available (i.e., another thread holds the lock), the thread will block (wait) until it can acquire the lock.
- This behavior ensures that the thread has exclusive access to the critical section protected by the lock.

2. Non-blocking (`_l.acquire(False)`)

- In the `worker_t` function, when `_l.acquire(False)` is used, it is a non-blocking acquire.
- If the lock is not available, the function immediately returns `False` without blocking the thread.
- This allows the thread to continue its execution even if it couldn't acquire the lock.
- The thread can then perform other tasks or take alternative actions instead of waiting for the lock.

Files

Running

Clusters

Select items to perform actions on them.

UploadNewRefresh

☐ 0

(/tree) / Lab2 (/tree/Lab2)

NameLast ModifiedFile size

.. (/tree)

seconds ago

☐ ctypes_example.ipynb (/notebooks/Lab2/ctypes_example.ipynb)

Runninga year ago3.61 kB

☐ multiprocessing_example.ipynb (/notebooks/Lab2/multiprocessing_example.ipynb)

Runninga year ago8.47 kB

☐ threading_example2.ipynb (/notebooks/Lab2/threading_example2.ipynb)

Runninga year ago13.6 kB

☐ main.c (/edit/Lab2/main.c)

a year ago159 B

☐ main.o (/edit/Lab2/main.o)

a year ago976 B

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 call our C function, myAdd(a, b).

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

```
Out[5]: 15
```

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

```
In [6]: def addC(a,b):  
        return _libInC.mymult(a,b)
```

Usage example:

```
In [7]: addC(10, 202)
```

```
Out[7]: 2020
```

Multiply

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

```
In [11]: import ctypes
```

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

```
In [13]: _libInC.mymult(3, 5)
```

```
Out[13]: 15
```

```
In [14]: def addC(a,b):  
        return _libInC.mymult(a,b)
```

In [15]: `addC(10, 202)`

Out[15]: 2020

In []:

```
1  #include<unistd.h>
2  int myadd(int a, int b){
3      sleep(100);
4      return a+b;
5  }
6  #include<unistd.h>
7  int mymult(int a, int b){
8      sleep(100);
9      return a*b;
10 }
11
12
13
```

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 l
    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
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)) # t
os.system("taskset -p -c {} {}".format(1, p2.pid)) # taskset is an os command
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 resources 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, returnValues))
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 []:

threading

importing required libraries and programing our board

```
In [1]: 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 50×0.02 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 [3]: 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
Worker 1 has the lock
Worker 0 has the lock
Worker 1 has the lock
Worker 0 has the lock
Worker 1 has the lock
Worker 0 has the lock
Worker 0 is done.Worker 1 has the lock
Thread-6 joined

```

```

Worker 1 is done.
Thread-7 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 attempts to get resource1. If the thread is successful in acquiring resource1, it releases resource0 and proceeds 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 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

Worker 0 has lock1Worker 1 has lock0

In []:

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.

deadlock Adjustment

In [19]:

```

def worker_t(_l0, _l1, num):
    using_resource0 = False
    using_resource1 = False

    for i in range(4):
        # Acquire Locks in the same order for both threads
        using_resource0 = _l0.acquire(True)
        using_resource1 = _l1.acquire(True)

        if using_resource0:
            print("Worker {} has lock0".format(num))
            blink(50, 0.02, num)
            _l0.release()

        if using_resource1:
            print("Worker {} has lock1".format(num))
            blink(5, 0.1, num)
            _l1.release()

        time.sleep(0) # yield

    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
Worker 0 has lock1
Worker 1 has lock0
Worker 1 has lock1
Worker 0 has lock0
Worker 0 has lock1
Worker 1 has lock0
Worker 1 has lock1
Worker 0 has lock0
Worker 0 has lock1
Worker 1 has lock0
Worker 1 has lock1
Worker 0 has lock0
Worker 0 has lock1
Worker 0 is done.Worker 1 has lock0
Thread-18 joined
```

```
Worker 1 has lock1
Worker 1 is done.
Thread-19 joined
```

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, `_resourceavailable` 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 [9]:

```
import threading
import time
from pynq import Overlay

# Load the overlay
import threading
import time
```

```

from pynq.overlays.base import BaseOverlay
base = BaseOverlay("base.bit")

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) # Non-blocking acquire
        if resource_available:
            print('Worker {} has the key'.format(num))

            # Blink with the key
            blink(5, 0.1, num)

            # Release the key
            _l.release()

            # Yield to give enough time for the other thread to grab the key
            time.sleep(0)
        else:
            print('Worker {} is waiting for the key'.format(num))

            # Blink without the key at a different rate
            blink(5, 0.2, num)

            # Adjust timing between having the key + yield and waiting for the key
            time.sleep(0.1)

    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.getName()
    t.join()
    print('{} joined'.format(name))

```

Worker 0 has the keyWorker 1 is waiting for the key

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

```
Worker 0 has the key  
Worker 0 is done.  
Thread-12 joined  
Worker 1 has the key  
Worker 1 has the key  
Worker 1 has the key  
Worker 1 has the key  
Worker 1 has the key  
Worker 1 is done.  
Thread-13 joined
```

In []: