threading

importing required libraries and programing our board

In [10]:
```
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 [14]:
```
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) # yield
    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.name
    t.join()
    print('{} joined'.format(name))
```

```
Worker 0 has the lock
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-25 (worker_t) joined
Worker 1 has the lock
Worker 1 has the lock
Worker 1 has the lock
Worker 1 is done.
Thread-26 (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 = 0.5 second.

In [11]:
```
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

    if using_resource0:
        _l0.release()
    if using_resource1:
        _l1.release()

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

```
Worker 0 has lock0
Worker 0 has lock1Worker 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 to prevent a deadlock. Write your code below:

In [13]:
```
#code fix deadlock

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):
        # Always acquire in the same order to prevent deadlock
        _l0.acquire()
        _l1.acquire()
        print("Worker {} has lock0 and lock1".format(num))

        # Do work that needs both resources
        blink(50, 0.02, num)
        blink(5, 0.1, num)

        _l1.release()
        _l0.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.name
    t.join()
    print('{} joined'.format(name))
```

```
Worker 0 has lock0 and lock1
Worker 0 has lock0 and lock1
Worker 0 has lock0 and lock1
Worker 0 has lock0 and lock1
Worker 0 is done.Worker 1 has lock0 and lock1

Thread-23 (worker_t) joined
Worker 1 has lock0 and lock1
Worker 1 has lock0 and lock1
Worker 1 has lock0 and lock1
Worker 1 is done.
Thread-24 (worker_t) joined
```

Also, write an explanation for what you did above to solve the deadlock problem.

Your answer: If the goal is to have thread0 complete its blinking 50 and 5 times for 4 loops then have thread1 do its blinking 50 times then 5 times for 4 loops then the fix is to acquire both locks in order and do bith blink methods then when done release the locks.

Bonus: Can you explain why this is used in the worker_t routine?

```
if using_resource0:
    _l0.release()
if using_resource1:
    _l1.release()
```

Hint: Try commenting it out and running the cell, what do you observe?

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 [18]:
```
import random
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, tid):
    for i in range(10):
        resource_available = _l.acquire(False) # this is non-blocking acquir
        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
            print('worker {} has key.'.format(tid))
            blink(i,.2,tid)
            _l.release()
            # give the other thread a clear chance
            time.sleep(0.4)

        else:
            # write code to:
            # print message for waiting for the key
            # blink for a while with a different rate
            # the timing between having the key + yield and waiting for the
            print('worker {} is waiting.'.format(tid))
            blink(i,.5,tid)
            # backoff + jitter so we do not collide forever
            time.sleep(0.1 + random.random() * 0.3)
```

```
    print('worker {} is done.'.format(tid))

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

```
worker 0 has key.
worker 1 has key.
worker 0 has key.
worker 1 is waiting.
worker 0 has key.
worker 1 is waiting.
worker 0 has key.
worker 1 has key.
worker 0 is waiting.
worker 1 has key.
worker 1 has key.
worker 0 is waiting.
worker 1 has key.
worker 1 has key.
worker 0 is waiting.
worker 1 has key.
worker 0 has key.
worker 1 is waiting.
worker 0 has key.
worker 0 has key.
worker 1 is done.
worker 0 is done.
Thread-33 (worker_t) joined
Thread-34 (worker_t) joined
```

In [ ]:

In [ ]: