WES 237A: Introduction to Embedded System Design (Winter 2026)
Lab 2: Process and Thread
Due: 1/19/2026 11:59pm

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.
- 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;
}
```

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

```
int myMultiply(int a, int b){
      sleep(1);
      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.**

```
_libInC = ctypes.CDLL('./libMyLib.so')

def multiply(a,b):

   return _libInC.myMultiply(a,b)
```

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 to 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?**

Python creates a new OS process assigns each one a monotonically. The counter is global to the Python interpreter session, not reset per cell execution.

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

os.system("taskset -p -c {} {}".format(1, p2.pid)) # taskset is an os command to pin the process to a specific CPU

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. Play around with different sleep times for both functions.
    a. **Explain the difference between the results of the 'Add' and 'Multiply' functions and when the processes are finished.**

`sleep()` only guarantees a minimum delay. The OS decides when to return this causes the. Main source of different time.

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

Two independent threads running two functions writing to the same shared memory variable returnValues.

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?**

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

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

fork = threading.Lock()

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

thread0 has resource1 and thread1 has resource0 and each is waiting for the other to release their resource in order to continue. Ex, thread0 locks l0 so thread1 is locked. Thread0 blinks when its done, it consumes l1. So now thread one is using both locks. Thread1 is still locked. Thread0 releases the l0 so now thread1 can acquire l0. Thread 0 blinks 5 times. Then releases l1. It is still waiting for thread1 but now acquires l0. Eventually, each is waiting for the other to release their resource. Deadlock.

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

Blocking physically idles the thread at the bottlenecked lock. Whereas, non blocking allows for code to continue to execute.

5. BONUS:
   Can you explain why this is used in the 'Two threads, two resources' section:
   if using_resource0:
   _l0.release()
   if using_resource1:
   _l1.release()

This guarantees:

1. Only acquired locks are released
2. Partial acquisition is handled safely
3. Deadlocks are avoided
4. Cleanup always works even if something fails mid execution

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 [ ]: