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 myProduct(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.**

```
def multC(a,b):

    return _libInC.myProduct(a,b)

print(multC(10, 2))
```

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?

The process number keeps climbing because the Jupyter Python kernel stays running in the background, and it uses a simple "running counter" to give every new process a unique name so they don't overlap.

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

```
os.system("taskset -p -c {} {}".format(0, p1.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. 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.**

The functions run on separate physical CPU cores, so they don't have to wait for each other; the multiply function will finish and print its result first just because its sleep() timer is shorter.

A process finishes as soon as its sleep() timer ends and the function returns; however, the Python script only reacts and reports that it is finished when it reaches the p.join() line for that specific process.

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.

Shared memory is a synchronized block of RAM that allows independent processes to bypass their usual isolation and write results into a common space accessible by the main program.

Each process in Python typically lives in its own isolated memory bubble; they normally can't see each other's variables. By using a shared memory object, we create a common "communal table" where Process 1 can drop off the addition result and Process 2 can drop off the multiplication result, allowing the main parent process to collect both values after the work is done.

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.start()) launches the thread. The function it executes is worker_t, which is defined as the target on line 32.

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

(fork = threading.Lock()) defines the mutual resource. It is accessed using the _l.acquire(True) method (Line 21) to grab the lock and _l.release() (Line 24) to give it back.

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

A deadlock occurs because the threads acquire resources in a circular wait: Thread 0 grabs Lock 0 and waits for Lock 1, while Thread 1 grabs Lock 1 and waits for Lock 0. Since neither thread will release the lock they already hold until they get the second one, they both hang forever.

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

Blocking: The code stops and waits indefinitely at that line until the resource is available

Non-Blocking: The code checks if the resource is available; if it's locked, it immediately returns False and continues executing other code instead of waiting.

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()
```

If a thread finishes its loop or crashes while still holding a lock, the resource stays locked forever, preventing other threads from ever starting. These lines ensure that if the thread terminates, it drops any keys it is still holding, so the system doesn't stall.

ctypes

The following imports ctypes interface for Python

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

Now we can import our shared library

In [10]:
```
_libInC = ctypes.CDLL('./libMyLib.so')
```

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

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

```
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]:
```
print(addC(10, 202))
```

```
212
```

Multiply

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

In [12]:
```
def multC(a,b):
    return _libInC.myProduct(a,b)

print(multC(10, 2))
```

```
20
```

In [ ]:
```

```

```c
#include <unistd.h>

int myAdd(int a, int b){
        sleep(2);
        return a+b;
}

int myProduct(int a, int b){
    sleep(5);
    return a*b;
}

```

multiprocessing

importing required libraries and our shared library

In [12]:
```
import ctypes
import multiprocessing
import os
import time
```

In [13]:
```
_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 [19]:
```
def addC_print(_i, a, b, time_started):
    val = ctypes.c_int32(_libInC.myAdd(a, b)).value #cast the result to a 32
    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.myProduct(a, b)).value #cast the result to
    end_time = time.time()
    print('CPU_{} Multiply: {} in {}'.format(_i, val, end_time - time_starte
```

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 different CPUs

In [29]:
```
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
os.system("taskset -p -c {} {}".format(0, p1.pid)) # taskset is an os comman
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
os.system("taskset -p -c {} {}".format(1, p2.pid)) # taskset is an os comman
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 12939's current affinity list: 0,1
pid 12939's new affinity list: 0
pid 12942's current affinity list: 0,1
pid 12942's new affinity list: 1
CPU_0 Add: 8 in 1.0422554016113281
CPU_1 Multiply: 15 in 1.0566394329071045
Process 1 with name, Process-33, is finished
Process 2 with name, Process-34, 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 [30]:
```
def addC_no_print(_i, a, b, returnValus):
    '''
    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
    '''
    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, returnValus):
    '''
    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
    '''
    val = ctypes.c_int32(_libInC.myProduct(a, b)).value
    # 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 = multiprocessing.Array('i', 2)


p1 = multiprocessing.Process(target=addC_no_print, args=(0, 3, 5, returnValu
p1.start() # start the process
os.system("taskset -p -c {} {}".format(0, p1.pid)) # taskset is an os comman
procs.append(p1)

p2 = multiprocessing.Process(target=multC_no_print, args=(1, 3, 5, returnVal
p2.start() # start the process
os.system("taskset -p -c {} {}".format(1, p2.pid)) # taskset is an os comman
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))

# TODO print the results that have been stored in returnValues
print("Addition Result (index 0):", returnValues[0])
print("Multiplication Result (index 1):", returnValues[1])
```

```
pid 12971's current affinity list: 0,1
pid 12971's new affinity list: 0
pid 12974's current affinity list: 0,1
pid 12974's new affinity list: 1
Process-35 is finished
Process-36 is finished
Addition Result (index 0): 8
Multiplication Result (index 1): 15
```

In [ ]:

In [ ]:

In [ ]:

In [ ]:

threading

importing required libraries and programing our board

In [3]:
```
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 [4]:
```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_1285/1470821917.py:37: DeprecationWarning: getName() is depre
cated, get 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 = 0.5 second.

In [5]:
```
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.getName()
    t.join()
    print('{} joined'.format(name))
```

```
Worker 0 has lock0
```

```
/tmp/ipykernel_1285/4236124143.py:44: DeprecationWarning: getName() is depre
cated, get the name attribute instead
  name = t.getName()
```

```
Worker 0 has lock1Worker 1 has lock0
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
Input In [5], in <cell line: 43>()
     43 for t in threads:
     44     name = t.getName()
---> 45     t.join()
     46     print('{} joined'.format(name))

File /usr/lib/python3.10/threading.py:1089, in Thread.join(self, timeout)
   1086     raise RuntimeError("cannot join current thread")
   1088 if timeout is None:
-> 1089     self._wait_for_tstate_lock()
   1090 else:
   1091     # the behavior of a negative timeout isn't documented, but
   1092     # historically .join(timeout=x) for x<0 has acted as if timeout=
0
   1093     self._wait_for_tstate_lock(timeout=max(timeout, 0))

File /usr/lib/python3.10/threading.py:1109, in Thread._wait_for_tstate_lock(
self, block, timeout)
   1106     return
   1108 try:
-> 1109     if lock.acquire(block, timeout):
   1110         lock.release()
   1111         self._stop()

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 [6]:

```
# TODO: Write your code here
def worker_t(_l0, _l1, num):
    using_resource0 = False
    using_resource1 = False

    for i in range(4):
        # always acquire l0 before l1 to prevent circular waiting
        using_resource0 = _l0.acquire(True)
        print("Worker {} has lock0".format(num))
        blink(50, 0.02, num)

        using_resource1 = _l1.acquire(True)
        print("Worker {} has lock1".format(num))

        # release l0 now that, have l1
        if using_resource0:
            _l0.release()
            using_resource0 = False

        blink(5, 0.1, num)

        # release l1 to finish the cycle
        if using_resource1:
            _l1.release()
            using_resource1 = False

        time.sleep(0.01)

    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_1285/1793502687.py:42: DeprecationWarning: getName() is depre
cated, get the name attribute instead
  name = t.getName()
```

```
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 1 has lock0
Worker 0 is done.
Thread-9 (worker_t) joined
Worker 1 has lock1
Worker 1 is done.
Thread-10 (worker_t) joined
```

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

Your answer:

I enforced lock ordering. Deadlocks happen when Thread 0 grabs Lock A and waits for Lock B, while Thread 1 grabs Lock B and waits for Lock A. By forcing both threads to ALWAYS REQUES Lock 0 first, it becomes impossible for one thread to hold the second lock while waiting for the first.

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

```python
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 [7]:
```
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 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 the key".format(num))
            blink(20, 0.05, num)
            _l.release()
            time.sleep(0.2)

        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 {} waiting...".format(num))
            blink(5, 0.1, num)

    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 key
Worker 1 waiting...
```

```
/tmp/ipykernel_1285/2556706393.py:41: DeprecationWarning: getName() is depre
cated, get the name attribute instead
  name = t.getName()
```

```
Worker 1 waiting...
Worker 1 has the key
Worker 0 waiting...
Worker 0 waiting...
Worker 0 has the key
Worker 1 waiting...
Worker 1 waiting...
Worker 1 has the key
Worker 0 waiting...
Worker 0 waiting...
Worker 1 has the key
Worker 0 waiting...
Worker 0 waiting...
Worker 0 has the key
Worker 1 waiting...
Worker 1 waiting...
Worker 1 has the key
Worker 0 waiting...
worker 0 is done.
Thread-11 (worker_t) joined
worker 1 is done.
Thread-12 (worker_t) joined
```

In [ ]:

```
#Note: The DeprecationWarning for getName() was present in the provided lab
```