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 myMult(int a, int b) {
    /* Skipping sanity check(s) and integer overflows */
    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.myMult(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?**

The initial name is set by the Process class constructor. If no explicit name is provided to the constructor, a name of the form ‘Process-N1:N2:…:Nk’ is constructed, where each Nk is the N-th child of its parent.

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

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

The “-c” argument to taskset os command binds the process to run on the specified 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.**

```
CPU_0 Add: 8 in 1.069350242614746
CPU_1 Multiply: 15 in 1.060805320739746
Process 1 with name, Process-1, is finished
Process 2 with name, Process-2, is finished

CPU_0 Add: 8 in 2.0674753189086914
CPU_1 Multiply: 15 in 2.0609095096588135
Process 1 with name, Process-1, is finished
Process 2 with name, Process-2, is finished

CPU_0 Add: 8 in 2.063145399093628
CPU_1 Multiply: 15 in 2.0636985301971436
Process 1 with name, Process-3, is finished
Process 2 with name, Process-4, is finished
```

Sometimes, the add process takes longer than the multiply, and sometimes, it is the other way around.
This difference may be due to system interrupts being handled in either of the CPU cores.

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.**

The multiprocess.Array instance (“returnValues” variable) created is the shared memory in the context of this exercise.

This shared memory (an array containing 2 integer elements) was written to by each of the processes.

The add process wrote into the 0th element and the multiply process wrote into the 1st element.

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))` creates the thread.
`t.start()` starts/launches the thread.

The thread executes the function specified in the “target” parameter. In this code, that is `worker_t`.

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

`fork = threading.Lock()`

The fork variable (or the lock) is the mutual resource.
It is accessed by the thread function with `acquire()` method.

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

The code enters a deadlock in the second iteration when thread0 holds lock1 and attempts to acquire lock0, but thread1 has lock0 and is waiting on lock1 to become available.

Essentially, thread0 needs lock1 to be available in order to release lock0.
Thread1 needs lock0 to be available in order to release lock1.

The 4 conditions are all met in this case:

a) Mutual exclusion - the locks provide mutual exclusion. Each thread can only acquire one lock at a time.

b) Hold and wait - each thread holds a lock and waits for the other lock to be available

c) No preemption - each thread runs in a critical section, the held lock is not released until the other lock is available

d) Circular wait - thread0 holds lock1 and waits on thread1 to release lock0; thread1 holds lock0 and waits on thread0 to release lock1.

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

A blocking function does not let the program execution continue until the requested operation is completed.

A non-blocking function lets the program execution continue even if the requested operation is not completed.

In this example, the operation is acquisition of a lock.
A blocking call will wait for the lock to be acquired (lock is free/available) before letting the program execution continue.
A non-blocking call will attempt to acquire the lock. If the lock is available, it would acquire it. If it is not available, then it will return a false (as per the implementation of acquire() method) and move forward.

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 is done to leave both the locks freed/available after the worker threads are done.

```
#include <unistd.h>

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

int myMult(int a, int b) {
    /* Skipping sanity check(s) and integer overflows */
    sleep(2);
    return (a * b);
}
```

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]:
```
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 [7]:
```
def multC(a, b):
    return _libInC.myMult(a, b)
```

In [8]:
```
print(multC(1331, 11))
```

```
14641
```

In [9]:
```
# Expect this to fail because of int overflow in C code
print(multC(900000000000000000, 2))
```

```
---------------------------------------------------------------------------
ArgumentError                             Traceback (most recent call last)
Input In [9], in <cell line: 2>()
      1 # Expect this to fail because of int overflow in C code
----> 2 print(multC(9000000000000000000, 2))

Input In [7], in multC(a, b)
      1 def multC(a, b):
----> 2     return _libInC.myMult(a, b)

ArgumentError: argument 1: <class 'OverflowError'>: int too long to convert
```

In [10]:

```
# Expect this to fail because of int overflow in C code (the result)
print(multC(2147483647, 4))
```

```
-4
```

In [ ]:

multiprocessing

importing required libraries and our shared library

In [1]:
```
import ctypes
import multiprocessing
import os
import time
```

In [2]:
```
_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 [3]:
```
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.myMult(a, b)).value #cast the result to a 3
    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 [5]:
```
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)) #
os.system("taskset -p -c {} {}".format(0, p1.pid)) # taskset is an os comman
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)) #
os.system("taskset -p -c {} {}".format(1, p2.pid)) # taskset is an os comman
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))
```

```
taskset: invalid PID argument: 'None'
taskset: invalid PID argument: 'None'
CPU_0 Add: 8 in 2.063145399093628
CPU_1 Multiply: 15 in 2.0636985301971436
Process 1 with name, Process-3, is finished
Process 2 with name, Process-4, 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 [7]:
```
def addC_no_print(_i, a, b, returnValues):
    '''
    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[0] = val

def multC_no_print(_i, a, b, returnValues):
    '''
    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.myMult(a, b)).value
    # TODO: add code here to pass val to correct position of returnValues
    returnValues[1] = 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
os.system("taskset -p -c {} {}".format(0, p1.pid)) # taskset is an os comman
p1.start() # start the process
procs.append(p1)

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

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

```
Process-7 is finished
Process-8 is finished
[8, 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 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 [2]:
```
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()
    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-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 [3]:
```python
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) # yield

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

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

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
Input In [3], in <cell line: 43>()
     43 for t in threads:
     44     #name = t.getName()
     45     name = t.name
---> 46     t.join()
     47     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 [22]:
```python
# TODO: Write your code here
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)
        print("Worker {} has lock0".format(num))
        blink(50, 0.02, num)
        _l0.release()

        using_resource1 = _l1.acquire(True)
        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()
    name = t.name
    t.join()
    print('{} joined'.format(name))
```

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

Worker 1 has lock1
Worker 0 has lock0
Worker 0 has lock1Worker 1 has lock0

Worker 1 has lock1Worker 0 has lock0

Worker 0 has lock1Worker 1 has lock0

Worker 1 has lock1Worker 0 has lock0

Worker 0 has lock1Worker 1 has lock0

Worker 0 is done.
Thread-45 (worker_t) joined
Worker 1 has lock1
Worker 1 is done.
Thread-46 (worker_t) joined
```

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

Your answer:

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 [32]:
```
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
            print(f"Worker %d has the lock" % (num))
            # blink for a while
            blink(50, 0.02, num);
            # release the key
            _l.release()
            # give enough time to the other thread to grab the key
            time.sleep(0.5) # yield

        else:
            # write code to:
            # print message for waiting for the key
            print(f"Worker %d is waiting for the lock" % (num))
            # blink for a while with a different rate
            blink(5, 0.1, num);
            # the timing between having the key + yield and waiting for the
            time.sleep(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()
    name = t.name
    t.join()
    print('{} joined'.format(name))
```

```
Worker 0 has the lock
Worker 1 is waiting for the lock
Worker 1 has the lock
Worker 0 is waiting for the lock
Worker 0 has the lock
Worker 1 is waiting for the lock
Worker 1 has the lock
Worker 0 is waiting for the lock
Worker 0 has the lock
Worker 1 is waiting for the lock
Worker 1 has the lock
Worker 0 is waiting for the lock
Worker 0 has the lock
Worker 1 is waiting for the lock
Worker 1 has the lock
Worker 0 is waiting for the lock
Worker 0 has the lock
Worker 1 is waiting for the lock
Worker 1 has the lock
Worker 0 is waiting for the lock
worker 0 is done.
Thread-63 (worker_t) joined
worker 1 is done.
Thread-64 (worker_t) joined
```

In [ ]: