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.**

```
def MultiplyC(a,b):

    return _libInC.myMultiply(a,b)

print(MultiplyC(2,5))
```

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?**

Because of the procs.append(p1) and procs.append(p2), each time you run the cell it continuously increments the process number.

    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.**

When the processes finish is determined by how long the sleep is set to. If I set the multiply function to sleep for 2 seconds while addition is set for 4, even though if you go through the code sequentially, the addition code runs first, the multiply function will complete first.

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 shared memory in relation to the code is the created multiprocessing array where we have the return values of the add and multiply functions that are running on different CPUs. It then stores the return values of the addition and multiplication functions into that same array, which then creates the shared memory space.

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))

The function the thread is executing is worker_t(_l, num)

    b. **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(), as it initializes the class instance that allows the user to lock and unlock a resource. It is accessed by the thread function with this line t = threading.Thread(target=worker_t, args=(fork, i)) as it sends the “fork” variable created to define a mutual resource as an argument for a function.

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

This code enters a deadlock because the threads dont properly release the resources they acquired before asking for the next resource, leading to the threads holding a resource that the other thread needs, which leads to a deadlock.

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

Blocking functions stop the execution of code, waiting for an event to occur. Non-blocking functions do not wait for an event to occur for code execution; everything continues to run.

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 any thread is still using any of the resources, it would release them.