# 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))<br><br>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.<br><br>Ex, thread0 locks l0 so thread1 is locked.<br><br>Thread0 blinks when its done, it consumes l1. So now thread one is using both locks.<br><br>Thread1 is still locked.<br><br>Thread0 releases the l0 so now thread1 can acquire l0. Thread 0 blinks 5 times. Then releases l1.<br><br>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