**WES 237A: Introduction to Embedded System Design (Winter 2025)**

**Lab 2: Process and Thread**

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. **Please make sure your responses are readable.**
* 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;

}

1. **Following the function above, write another function to multiply two integers together. Copy your code below.**
2. Save main.c
3. In Jupyter, open a terminal window (New -> Terminal) and *change directories* (cd) to ‘Lab2’ directory.

$ cd Lab2

1. Compile your ‘main.c’ code as a shared library.

$ gcc -c -Wall -Werror -fpic main.c

$ gcc -shared -o libMyLib.so main.o

1. Download ‘ctypes\_example.ipynb’ from [here](https://drive.google.com/file/d/1P9Xm8ES5mBxnLg1SV4wTsOVzGbBCukD7/view?usp=share_link) and upload it to the Lab2 directory.
2. Go through each of the code cells to understand how we interface between Python and our C code
3. **Write another Python function to wrap your multiplication function written above in step 3. Copy your code below.**

To summarize, we created a C shared library and then called the C function from Python

**Multiprocessing**

1. Download ‘multiprocess\_example.ipynb’ from [here](https://drive.google.com/file/d/1KhVRWTj1VJKfrB2jJD0A62XivSW3PYlZ/view?usp=share_link) and upload it into your ‘Lab2’ directory.
2. Go through the documentation (and comments) and answer the following question
   1. **Why does the ‘Process-#’ keep incrementing as you run the code cell over and over?**

**Each time a new Process is created, the default name is an index that increments by one for every time the constructor creates a new Process. Names do not need to be unique across processes, but the PID will always be unique to a single Process.**

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

os.system("taskset -p -c {} {}".format(0, p1.pid)) # taskset is an os command to pin the process to a specific CPU

**The “taskset” command assigns the processes to run on a CPU core whose index corresponds to the -c argument. In the above example, p1 will be assigned to CPU core #0.**

1. 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. Try sleeping the functions for various, different times (or the same).
   1. **Explain the difference between the results of the ‘Add’ and ‘Multiply’ functions and when the processes are finished.**

The Add and Multiply functions will have different completion times depending on the value of sleep that we defined within those functions. When the processes are finished, the logs reflect how long we have slept inside those functions.

1. Continue to the lab work section. Here we are 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.
      1. Process 1 (add) is stored in index 0 of the array (array[0])
      2. Process 2 (mult) is stored in index 1 of the array (array[1])
   4. Print the results from the array.
   5. **There are 4 TODO comments that must be completed**
2. Answer the following question
   1. **Explain, in your own words, what shared memory is relating to the code in this exercise.**

The shared memory in between processes for this exercise is the returnValues array that we defined. Both processes are accessing the returnValues array and writing back to the array before finishing.

**Threading**

1. Download ‘threading\_example.ipynb’ from [here](https://drive.google.com/file/d/1Cv0Z3YmeREeKa-zHcfQ8q01o3EzYTZYn/view?usp=share_link) and upload it into your ‘Lab2’ directory.
2. Go through the documentation and code for ‘Two threads, single resource’ and answer the following questions
   1. **What line launches a thread and what function is the thread executing?**

t = threading.Thread(target=worker\_t, args=(fork, i))

t.start()

The first line declares the thread and the function to execute (worker\_t) and the t.start() call launches the thread and begins execution.

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

fork = threading.Lock()

This line defines the mutual lock and is passed into the arguments for worker\_t

t = threading.Thread(target=worker\_t, args=(fork, i))

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

This code enters a deadlock when Worker 1 holds lock 0 and Worker 0 holds lock 1. Since neither of them release it before attempting to acquire another lock, the two threads are stuck waiting on each other to release their locks and enter a deadlock.

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

Blocking functions halt thread execution until the function finishes. An example of this is when we pass (True) to the acquire function. Non-blocking functions do not halt thread execution and continues. An example of this is when we pass (False) to the acquire function.