WES 237A: Introduction to Embedded System Design (Winter 2024)
Lab 2: Process and Thread
Due: 1/22/2024 11:59pm

Github: https://github.com/achari23/lab2
WES237A Lab 2 Writeup - Threads and Processes
Anand Chart (A69028625) and Thomas Offenbecher (A59021337)
1/20/2024

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. You could either scan your written copy, or simply type your answer in this Google Doc. **However, 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;
}
```

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 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 into 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?**

Kernel makes things move faster since it assigns IDs by incrementing. Much slower if you scan for what IDs are available.

    b. **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 -p flag assigns the core. The -c flag assigns the ID of the process p1.

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

When both sleep times are the same, the threads finish almost simultaneously with Add finishing first. When sleep time in the multiply function is increased, add finishes first then multiply finishes later. When the add function has longer sleep than multiply, both will print simultaneously after add has finished sleeping. The join() method of the Python multiprocessing library makes it so that the add function is waited for even though the multiply C function completes first due to smaller sleep time. The multiply function completing first is demonstrated by the multiply print statement appearing before the add. But the process #1 print statement appears first, meaning the process #1 completes first. The multiply C function completing before the add demonstrates parallelism due to there being multiple cores (separate memory spaces used).

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 relating to the code in this exercise.**

The memory is mostly not shared as the processes are running on separate CPUs and will have their own stack trace. Since these variables are not dynamically allocated, there is no heap and no memory shared between the two processes. The only “shared” memory is the output array but I’d argue this memory is not shared as we introduce values to two different indices or offsets to the array.

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, fork1, i)) <- defines the thread and that it will execute function worker_t with arguments as listed. Worker_t accessing the lock to the resource needed to run the blink method for the LED four times for the thread to finish.

t.start() triggers the thread.

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

The lines below define mutual resources:

fork = threading.Lock()

These are accessed by the thread function with the line:

_l.acquire(True);

And released with line:

_l.release();

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 as the first process acquires lock 1 and the second process acquires lock 0 at the same time. Neither process is able to call the blink function because of this as both functions are locked by each other. Process 1 is waiting for lock 0 and Process 2 is waiting for lock 1; this leaves the entire execution in deadlock.

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

Blocking functions will cause other threads to stop because it is running. A non blocking function allows for other threads to continue running regardless of lock state, perhaps a section of code may run or not run based on lock state though.