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.

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 32
    end_time = time.time()
    print('CPU_{} Multiply: {} in {}'.format(_i, val, end_time - time_started
```

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 difference CPUs

In [4]:
```
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)) # t
os.system("taskset -p -c {} {}".format(0, p1.pid)) # taskset is an os command
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 command
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))
```

```
CPU_1 Multiply: 15 in 3.0425331592559814
CPU_0 Add: 8 in 5.047112703323364
Process 1 with name, Process-1, is finished
Process 2 with name, Process-2, 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 [5]:

```
def addC_no_print(_i, a, b, returnValus):
    '''
    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
    returnValus[0] = val;
def multC_no_print(_i, a, b, returnValus):
    '''
    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
    returnValus[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', [0,0])


p1 = multiprocessing.Process(target=addC_no_print, args=(0, 3, 5, returnValue
os.system("taskset -p -c {} {}".format(0, p1.pid)) # taskset is an os command
p1.start() # start the process
procs.append(p1)

p2 = multiprocessing.Process(target=multC_no_print, args=(1, 3, 5, returnValu
os.system("taskset -p -c {} {}".format(1, p2.pid)) # taskset is an os command
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('Add value = {}'.format(returnValues[0]))
print('Multiply value = {}'.format(returnValues[1]))
```

```
Process-3 is finished
Process-4 is finished
Add value = 8
Multiply value = 15
```

```
In [ ]:
```

```
In [ ]:
```

```
In [ ]:
```

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]:
```
_libInC.myAdd(3, 5)
```

Out[3]:
```
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]:
```
addC(10, 202)
```

Out[5]:
```
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(3,4)
```

Out[7]:
```
12
```

```
In [ ]: multC(3,4)
```

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]:

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

```
Worker 0 has the lock
Worker 1 has the lock
Worker 0 has the lock
Worker 1 has the lock
Worker 0 has the lock
Worker 1 has the lock
Worker 0 has the lock
Worker 0 is done.Worker 1 has the lock

Thread-4 joined
Worker 1 is done.
Thread-5 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 = 1 second.

In [ ]:

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

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

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 above to prevent a deadlock.

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 [3]:

```
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 acquire
        if resource_available:

            # write code to:
            # print message for having the key
            # blink for a while
            # release the key
            # give enough time to the other thread to grab the key
            print("Worker {} has the lock!".format(num))
            blink(50, 0.02, num)
            _l.release()
            time.sleep(1)

        else:
            # write code to:
            # print message for waiting for the key
            # blink for a while with a different rate
            # the timing between having the key + yield and waiting for the k
            print("Worker {} is waiting for the lock!".format(num))
            blink(25, 0.04, num)
            time.sleep(2)

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

```
Worker 0 has the lock!
Worker 1 is waiting for the lock!
Worker 0 has the lock!
Worker 1 is waiting for the lock!
Worker 0 has the lock!
Worker 1 has the lock!
Worker 0 is waiting for the lock!
Worker 1 has the lock!
Worker 0 has the lock!
Worker 1 is waiting for the lock!
Worker 0 has the lock!
Worker 1 has the lock!
Worker 0 is waiting for the lock!
Worker 1 has the lock!
Worker 0 has the lock!
Worker 1 is waiting for the lock!
Worker 0 has the lock!
Worker 1 has the lock!
Worker 0 is waiting for the lock!
Worker 1 has the lock!
worker 0 is done.
Thread-6 joined
worker 1 is done.
Thread-7 joined
```

In [ ]: