**CSC 338 Parallel and Distributed Computing**

**Exercise No. 2, February 1, 2017**

**Introduction to multiprocessing**

**Goal**

Learn about Python’s multiprocessing module.

**Background**

Threads are attractive but sometimes you need separate processes—especially in languages like Python that don’t allow threads to execute in parallel. The Python interpreter is not “thread-safe” so Python has something called a “global interpreter lock” (GIL) that lets only one thread execute at any given time. Processes, though, have their own memory space and system resources and they can execute in parallel.

**Procedure**

Use the command line to execute programs in the following steps.

1. Start a python shell from the command line. Import multiprocessing as mp and use mp.cpu\_count() to find out how many cores your system has.
2. Copy mp01.py from trace and execute it. Note the name that’s printed by the generated processes. As you can see, there is a lot of similarity between the way the programmer uses multiple processes and multiple threads.
3. Remove the *if* statement (if \_\_name\_\_) and see what happens when you execute the program (be sure to unindent the call to main()).
4. Put the *if* statement back (be sure to indent the call to main()). Inside the *for* loop, add *p.join()* immediately following *p.start().* Run the program several times. What do you notice about the order in which the processes print their messages? Now take *p.join()* out and run the program several times. Does the order change? What does that tell you about *p.join()?*
5. Add a second function, foo2(). foo2() should take the same argument as foo(). foo2() should print something to distinguish it from foo() but should also print the argument and \_\_name\_\_. Modify main() so that it creates a process to call one of the functions when the loop index (i) is odd and the other when the loop index is even (recall that i % 2 can tell you whether i is even or odd). Note the name of the processes. Can you find a way to distinguish them? *Hint:* You can give a process a name, just as you can a thread. There doesn’t seem to be a getName() method but *name* is an instance variable. Import multiprocessing into a Python shell and use the help function to find a method that might be useful in accessing the current process. Of course, in this case, the argument *i* serves to disambiguate the processes.
6. Copy the program spartan\_ideas1.py (from a blog at Michigan State University) from trace and execute it. Look at the code and note that multiple processes can also communicate using a Queue—but this time it’s a queue defined in the multiprocessing module. This is not shared memory—there’s some message passing going on in the background to enable the processes to all put their output in the queue.
7. Note the way spartan\_ideas1.py uses *p.join()—*it makes a list of processes then starts them, one by one. It joins the processes in a separate loop so they will all be started without waiting for the joins. Take out the calls to *join().* Does it make any difference in the way the program works? You can see that the *join()* calls are not needed in this program—the main process will wait until all processes have finished before it exits (we can set processes to daemon if we want the main process to be able to exit while some are still executing). But sometimes we have to let all processes finish before going on to some other operation—then the joins are necessary.
8. It’s impossible to predict what order the strings will be in when they go into (and come out of) the queue. Order probably isn’t very important when you’re looking at random strings but sometimes order is important. Modify spartan\_ideas1.py to print the strings in the order they were created.
   1. Add the loop variable (x) as an argument to rand\_string().
   2. Instead of just putting the string into the queue, put a tuple that has, as its first element, the new loop variable argument and, as its second element, the rand\_str.
   3. In main, sort the results (results is a list; results.sort() will sort on the first element in each tuple).

There may be some Python elements you aren’t used to seeing in this program. In particular, list comprehensions are widely used by people who program a lot in Python (*processes* and *results* are created using list comprehensions and *rand\_str* seems to be using a list comprehension as well, although it looks a bit like magic to me). Dictionaries, tuples, and sets can also be created using comprehensions.