**CSC 338 Parallel and Distributed Computing**

**Exercise No. 1, January 25, 2017**

**Introduction to the Python threads**

**Goal**

Learn about Python threads and compare serial and threaded programs for downloading web pages in Python.

**Background**

There are several ways of writing concurrent code in Python, including some third-party libraries. We will look at the threading and multiprocessing modules from the standard library. Threads are attractive because they are *lightweight—*they exist as subsets of a process and share memory space and executable code. Processes must be created by the operating system and have separate memory spaces, code, and system resources (such as file buffers).

**Procedure**

Copy first\_thread.py from the Exercise01 folder on *trace.* first\_thread.py starts a thread that executes the function fun(), which does nothing. Do the following. It’s best to run these programs from the command line because, while threaded programs will probably run successfully in an IDE, we’ll be writing programs soon that will not.

1. Print “hello from a thread!” in fun1().
2. Put the thread to sleep for 1 second by using the time.sleep() method (argument is number of seconds to sleep.
3. After the sleep, print “Goodbye from a thread!” in fun1().
4. What is the name of the main program (print \_\_name\_\_); what is the name of the thread (print threading.currentThread().getName()).
5. Give the thread a name—in the statement that creates the Thread, add *name = “fun1.”*
6. Modify the program to give the thread an argument—after *target = fun1,* add *args = [42]* to the thread creation statement and add a parameter to the fun1(). definition. Note that *args* requires a sequence but the sequence is unpacked when passed to fun1(). In other words, if you send a list of two arguments to fun1(), it will receive two separate arguments—not a list with two elements. Try adding another argument after 42 and see what happens. Print the argument(s) in fun1().
7. Now add two more functions—fun2() and fun3(). Create threads t2 and t3 to run them, give them appropriate names, and print appropriate salutations from them (“hello from”, threading.currentThread().getName(), for example). Don’t forget to start them and join them.
8. Try sleeping the threads for different lengths of time. Can you control the order in which they exit?
9. Close first\_thread.py and open web\_fetch\_serial.py (get it from *trace).* This program downloads a number of web pages using the urllib module. Execute the program several times and note how long it usually takes.
10. Save web\_fetch\_serial.py as web\_fetch\_threads.py and modify it to use a separate thread for each download. This will require that threads coordinate so different threads don’t download the same web page. An easy way to do this is to use a Queue from the queue module: que = queue.Queue(). I suggest you make the queue global, although you could define it in main and pass it to get\_page().

In main():

* 1. If you want the queue to be local, create it here. Either way, put the hosts into the queue (use the queue’s .put() method).
  2. In a for loop, spawn a thread for each host (use for i in range(len(hosts))). I suggest you use t for each thread variable and give the thread a name such as “gp” + str(i). The target function is get\_page(); if your queue is local to main(), pass it to the thread using *args.*
  3. Because we’re turning the threads loose and not waiting on them, make the threads “daemon” threads (t.setDaemon(True)).
  4. Start the thread but don’t join it.
  5. Because the threads are daemons, the main thread can exit while the threads continue on their merry way. So, after the loop that creates and starts the threads, wait on the queue to be fully processed (use the queue’s .join() method).

In get\_page():

1. If the queue is local, get\_page() takes no arguments. If you’re passing the queue, then make it an argument to get\_page().
2. Get a host from the queue (use the queue’s .get() method).
3. Open the url, as in web\_fetch\_serial.py, and read the html.
4. Print the name of the thread and the first 50 characters from the html.
5. Tell the queue the task is done (use the queue’s task\_done() method).

Execute web\_fetch\_threads.py several times and note how long it usually takes. Is it faster than web\_fetch\_serial?