**CSC 338 Parallel and Distributed Programming**

**Assignment 1: Threads and processes** **Due September 27, 2016**

**Background**

risk\_serial.py simulates a number of Risk battles. Risk® is a strategy board game in which players try to conquer the world by controlling territory groupings roughly corresponding to continents. A territory is conquered when an attacking player's armies defeat the armies currently occupying the territory.

While it's not necessary to understand the rules of the game to carry out this assignment, it may be useful to know that battles are conducted by rolling dice. An attacking player holding four or more armies may choose to roll up to three dice; a defending player holding two or more armies may choose to roll one or two dice. The number of allowed dice decreases for both players as the number of armies fall below these thresholds. Again, the details are unimportant for the purpose of this assignment but the rules are implemented in risk\_serial.py. The outcome of a battle is determined by comparing rolls of the dice—the attacker’s highest roll vs the defender’s highest roll, then the attacker’s second highest roll vs the defender’s second highest roll; the defender wins ties. For example, if the attacker rolls [5, 3, 2] and the defender rolls [5, 2], then each player will lose one army (5 == 5, won by the defender, and 3 > 2, won by the attacker). Battles may continue until the defender has 0 armies, in which case the attacker has conquered the territory, or until the attacker has 1 army, in which case the attack has failed.

risk\_serial.py takes a command line argument to determine the number of simulations to carry out. Each simulation starts with both players holding 10,000 armies and proceeds until one player is unable to continue. Because the program takes a command line argument, you must invoke it from the command line. For example, *python risk\_serial.py 50* carries out 50 simulations then prints the mean, standard deviation and median of the number of attacking armies remaining when the defender has none. It also prints the time taken by the simulations.

**Procedure**

Your task is to parallelize risk\_serial.py. Create two programs, one using threads (risk\_thr.py) and one using multiple processes (risk\_mp.py). For the threads program, fork a separate thread for each simulation. Rather than returning the result from run\_simulations() you should put it in a queue. You will have to join the threads—queue.task\_done(), needed for queue.join(), is used by consumers to indicate they have finished processing items taken from the queue. run\_simulation(), though, is a producer that puts an item in the queue.

One problem you will face is that the statistics functions don't work on a queue. So you have to get() the items from the queue and put them in a list in order to print the simulation statistics.

Using multiple processes is a bit more involved because you don't want to start a separate process for each simulation—in fact, you should use the number of processors available to the program. You can find the number of processors using multiprocessing.cpu\_count(). The number of simulated battles is not likely to be evenly divisible by the number of processors—so one processor may have to do some additional simulations. For example, if you have 4 processors to run 50 simulations, three processors will do 12 simulations while one will do 14 simulations. You can determine how to apportion simulations among processors using integer division (num\_simulations // num\_processors) and modulo (num\_simulations % num\_processors).

Make a list of processes then start them in one loop and join them in another loop. Once again, you'll have to move the results from the queue into a list before printing the statistics (by the way, be sure your queue for this program is a multiprocessing queue). NOTE: I used a global queue on Linux but I had to make it local to main(), and pass it to run\_simulations(), on Windows.

Try your programs, as well as risk\_serial.py, with 25, 50, 75, and 100 simulations. Include a short report (text or Word document) giving the results of these experiments. Explain your results—which program is fastest, and why? Which program is slowest, and why? Also, in your report, document the number of cores on your test system.

**Deliverables**

Put your programs (risk.thr.py and risk\_mp.py) and report in a folder named Asn1 in your upload folder on trace by five minutes after class time on the due date.