ASSIGNMENT 1 – LAB 3: CONCURRENCY

MAIN TOPIC: MULTIPROCESSING AND MULTITHREADING

Objective:

Write a Python script that spawns `W` workers (processes or threads). Each worker must first generate a sequence of `N` random values obtained from a specified statistical distribution, which should be considered as inter-event time intervals. After each interval, the worker must write the current timestamp (with millisecond precision) to an output file. When the all workers terminate, the script must read the output file and compute the average inter-event time based on the recorded timestamps. The average value must be computed both overall and separately for each worker.

You can use the numpy library to generate random samples from a given distribution. For instance, `np.random.exponential(1 / LAMBA, size=SIZE)` generates a ramdom numpy array of `SIZE` values sampled from an exponential distribution with rate parameter `LAMBDA`.

Command line arguments:

The script must allow these inputs from the command line:

- Number of workers to generate
- Number of inter-event intervals to generate
- Distribution of the inter-event time and related parameter(s), including at least:
    - deterministic (parameter: fixed inter-event time)
    - uniform within an interval `[0:T]` (parameter: size of the interval)
    - exponential (parameter: mean event occurrence rate)
- Name of the file to write the timestamps in
- Choice between sequential, multithreading or multiprocessing

I'll report here only the deterministic function outputs with the difference between Multiprocessing, threading and sequence to discuss on the different results. Further observations to compare the other distributions are reported at the end.

Uniform (I limit the output at 10 values in the list)

1) **MULTIPROCESSING OUTPUT:**

```
python3 ASSIGNMENT1_done.py --workers 4 --intervals 10 --dist u --param 2.0 --file outputUNI.txt --mode processes
```

outputUNI.txt

```
worker_id,timestamp_ms

4,1768816568056

2,1768816568907
```

```
1,1768816569334

4,1768816569528

2,1768816569548

3,1768816569661

2,1768816569844

2,1768816569917

1,1768816570506

3,1768816570561

…
```

TERMINAL

```
Starting multiprocessing mode for worker 4

Worker 4 has generated the following intervals: [0.19166558 1.46650006 1.33993414 1.62374284 1.25677148 0.33721578

 1.92894777 0.65876374 1.29618866 1.03090656]

Worker 1 has generated the following intervals: [1.46953028 1.16800515 0.49011886 0.43756963 0.06246784 0.71648893

 0.75923152 0.31915935 0.96241619 1.45243817]

Worker 2 has generated the following intervals: [1.04271764 0.63507399 0.29102598 0.06778656 1.02229345 1.09180237

 0.13037563 1.24399351 1.051255   0.85469703]

Worker 3 has generated the following intervals: [1.7965352  0.89986148 0.19290923 0.93781163 0.49481843 0.87212149

 0.34439696 1.56470686 1.54104406 0.31103879]

Results:

Worker 1: Inter-event average = 716.56 ms

Worker 2: Inter-event average = 714.22 ms

Worker 3: Inter-event average = 800.00 ms

Worker 4: Inter-event average = 1220.56 ms


Global inter-event average: 862.83 ms
```

2) SEQUENTIAL OUTPUT:

outputUNISeq.txt

```
worker_id,timestamp_ms

1,1768816922093

1,1768816922525

1,1768816922978

1,1768816924751

1,1768816925540

1,1768816927187

1,1768816928631

1,1768816929140

1,1768816930160

1,1768816930956

…
```

TERMINAL

```
python3 ASSIGNMENT1_done.py --workers 4 --intervals 10 --dist u --param 2.0 --file
outputUNISeq.txt --mode seq

Starting sequential mode for worker 4

Worker 1 has generated the following intervals: [0.17172952 0.42642224 0.4464098
1.76985748 0.78249451 1.64189366

 1.43770273 0.50547417 1.01902057 0.79084291]

Worker 2 has generated the following intervals: [1.19763738 0.77661761 1.33582607
1.4599295  0.20488375 1.03433871

 0.58568239 1.04711224 1.00935673 1.86470107]

Worker 3 has generated the following intervals: [0.53163927 1.07657718 0.48838831
1.94249694 1.92971947 1.82848063

 1.11937421 0.222041   1.05880848 1.42358283]

Worker 4 has generated the following intervals: [1.37116438 0.80234255 1.8009963
0.60482525 0.86130562 1.3752063

 1.65592903 1.405909   1.76480451 0.2167883 ]
```

```
Results:

Worker 1: Inter-event average = 984.78 ms

Worker 2: Inter-event average = 1040.44 ms

Worker 3: Inter-event average = 1237.33 ms

Worker 4: Inter-event average = 1169.33 ms


Global inter-event average: 1107.97 ms
```

3) MULTITHREADING OUTPUT

outputUNIThr.txt

TERMINAL

```
Starting multithreading mode for worker 4

Worker 1 has generated the following intervals: [0.36121297 0.4700239  1.72150332
1.16120147 1.54714174 0.14570826

 1.09594233 1.21930417 0.58561586 0.98566249]

Worker 2 has generated the following intervals: [0.61421072 1.63027825 1.58926013
1.58007267 0.14795611 0.91082494

 1.98893485 1.69137978 0.9869538  0.84647748]

Worker 3 has generated the following intervals: [0.7643227  0.85075449 0.63434906
0.69830257 0.10706514 1.63381442

 1.97858079 0.58323153 0.39185109 0.67470897]

Worker 4 has generated the following intervals: [1.95224235 1.58672482 0.28766299
0.09209149 0.62180045 1.44202139

 0.59437464 1.77013702 0.59113466 0.41651401]


Results:

Worker 1: Inter-event average = 996.78 ms

Worker 2: Inter-event average = 1268.22 ms

Worker 3: Inter-event average = 842.22 ms

Worker 4: Inter-event average = 828.11 ms
```

Global inter-event average: 983.83 ms

OBSERVATIONS ON THESE RESULTS

Using the same distribution we notice the main differences between the modes.

In the sequential all the timestamps per workers are grouped at the very starting point, there's no overlapping between the generation of data. When the worker 1 has finished then the subsequent starts. In the multiprocessing and multithreading there's the concurrency: we have an alternation of the execution and the programme writes in the file when the events are generated. With respect to multithreading and multiprogramming in the sequential we have more overhead in computing the sum of all time intervals.

In my code I have used the Global Interpreter Lock that in very complex computations by the workers is necessary, in order to not risk the Race conditions. If here I have the lock, in real conditions there's the need of the queuing system of the processes.

OBSERVATIONS BETWEEN DIFFERENT DISTRIBUTIONS (by using multiprocessing)

python3 ASSIGNMENT1_done.py --workers 4 --intervals 10 --dist *** **--param 2.0** --file output***.txt --mode processes

Uniform:

Global inter-event average: 862.83 ms

Exponential:

Global inter-event average: 417.92 ms

Deterministic:

Global inter-event average: 2004.44 ms

The experiment demonstrates that the choice of distribution drastically impacts event traffic and timing, while the execution mode (multithreading/processing) dictates the total "wall-clock" completion time and data organization. Obviously it depends also on the parameter chosen for each distribution, that are not the same for each of them also in terms of logic.