CSV Analyzer - Petal Coding Challenge

This is the semiformal program spec for a CSV analyzer.

Usage

Please run the code as follows:

```
./csv_analyzer.py -f file_1.csv file_2.csv
```

This command will output files of the format `file_1_output.csv` and `file_2_output.csv`. You can specify as many files as you want, but all have to have the `.csv` extension.

You can also specify the number of processes you would like it to use (by default, it uses a single process) as follows:

```
./csv_analyzer.py -f file.csv -p NUM_PROCESSES
```

`NUM_PROCESSES` must be replaced by an int. Keep in mind, however, that the program will use at most the number of CPU cores as the number of processes. It will also only use one process per file (if you ask for 8 processes, but only input 3 files, it will only use 3 processes).

Design Decisions

The code is written entirely in Python, using nothing but the standard library. It reflects my own personal design philosophies; a featureful and reliable standard library should provide developers with most of what they need in most situations, and third-party tools should be used only in the case that the standard library does not have a module that satisfies the use case.

The CSV documents are parsed using the Python `csv` module; theoretically, this will deal with most of the edge cases regarding the "desc" and "misc" columns, which are arbitrary and possibly hard-to-parse strings.

Discussion on Parallelism

The spec for a parallelized CSV analyzer is as follows.

Python runs under a Global Interpreter Lock (GIL), which is essentially a *mutex* on the interpreter; that is, Python program memory cannot be accessed by more than one context at one time. There is a `threading` module supplied in the standard library, but it is of no use in this case, since it would not enable CPU-level parallelism because of the GIL; its purpose is instead to facilitate I/O-heavy workloads in which multiple contexts are necessary, but not necessarily CPU-level parallelism.

Instead, to enable CPU-level parallelism, I use the `multiprocessing` module to enable process-level CPU parallelism. This will ultimately result in better CPU utilization, unlike in the case with threads via the `threading` module, because multiprocessing involves copying the full address-space of a program in memory; this effectively circumvents the issue presented by the GIL.

Possibility for Future Improvements

This code was designed to be as well-designed and similar to production-ready software as possible. As such, I would have implemented mechanisms to address the following considerations.

More Extensive Automated Testing

Currently, I have added unit-tests for the core functionality of the program, which is written inside `utils.py`. You may run `./test.py` to run the two tests I have written. One is meant to test that the program ingests the CSV records properly, and the other is meant to test that the program properly aggregates them upon having read a CSV file to submission.

If I had had more time, I would have written E2E tests, which would have tested every facet of the application from the argument parser to the end-file writer. In doing so, I would have probably used the `pytest` module, a much more featureful testing framework than the standard library's `unittest`.

Further Benchmarks on Parallelism

In general, concurrency, parallelism, and really any kind of asynchrony, require extensive testing and benchmarking to handle all the edge cases and to ensure that the asynchrony does indeed make the program faster at scale. Fortunately, Python provides a very black-box `multiprocessing` module that handles most of the complexity on behalf of the user. Still, it remains to extensively benchmark the code and prove that it does indeed run faster at scale than the non-parallel equivalent does.

I was not able to do this because I was not able to procure a large enough input dataset for this to make a difference. However, I was able to confirm by copying the extsting input files several times that the parallelized version runs about an order of magnitude faster than the non-parallelized version does. To do this, I simply copied the existing input files 20 times each, and used the Unix `time` utility for benchmarking. I was able to confirm with this workload that the parallelized version ran an order of magnitude faster than the non-parallelized one (the former took 8 seconds to run,whereas the latter took 35 seconds).

More interestingly, I did observe that the program's parallelized version spent a significant amount of its time in the system, whereas the the non-parallelized version spent most of its time in userspace. Ostensibly, this is because the time taken to apply the Unix `fork` call for each process consumed a large amount of time for multiple processes. It remains up to further experimentation whether or not multithreading (via the Unix `clone` call) in a language that easily supported it would be faster than the multiprocessing approach.

Logging in Error Cases

The final thing I would add before deploying this program as a service in production would be extensive logging in all error cases.

Multiple approaches could be taken to this. Printing to stdout/stderr is not ideal, because those error traces might not be readily available over all deployment methods. However, appending all output to a designated log file, or perhaps even using a dedicated logging service like LogStash as a logging backend (depending on scale), are good ways to make the logging more robust.