Ben King

Corey Clark

CS 5393

23 October 2024

Lab 2 Report

The design of my profiler is similar to the example one made in class. I have a Time class that only serves to get the current time in seconds. The time class is included in my profiler class. In my Profiler class there are 3 other sub class inside it called TimeRecordStart and TimeRecordStop and ProfilerStats. Time record start serves to only hold the name of the section the user wants to profile and the time at star, the TimeRecordStop class holds the elapsed time, section name, line number, file name, and function name. it gets the elapsed time by subtracting the current time when its created and the start time from the time record start object. The profiler stats class holds the the same data as time record start, but it also holds min, max, and avg time along with the call count.

The actual Profiler class has methods to enter and exit sections and method to print stats, either to terminal, csv, or json. It also holds a vector of TimeRecordStart objects and TimeRecordStop objects and a map that has a key of the section name that points to a profilerStats object. This is where the data is stored about each section the user profiles. The profiler constructor creates a static global profiler and also initializes the vector for start and elapsed times, start times is 100 elapsed times is 1000000. The enter section method is very simple, takes in a name for a section and then it grabs the current time from whenever it is called and throws that time into a TimeRecordStart object and places that object at the back of the start times vector. The exit section method takes in the section name, file name, function name, and line number. I solved the issue of the user having to add file, function, and line number by using macros to call the enter and exit section, and in those macros, it grabs the needed data from wherever the macro is called. In the method it grabs the current time. It then goes into a loop that starts at the back of the start times vector and searches for the corresponding section name. once it finds it is grabs the start time from the timeRecordStart object and subtracts that from the current time to get the elapsed time. It then removes that object from the start times vector. Then the method creates a timeRecordStop object and stores it in the elapsed time vector. Then it hits an if statement to check to see if a profiler with that section name is already stored in the stats map. If it isn't then it adds it to the map and creates a new profilerStats object that

corresponds to the section name. if it does exist already, it will just update the stats in the object (min/max/count/total time). The printStats method all can be called with macros, and they just output the stats to either the terminal, a csv, or a json. The output to terminal does not include file name, function name, or line number because I thought it was unnecessary clutter on the screen. I also formatted it so that it was easy to read. The .csv and .json files are outputted to the Data folder in the repo.

The reason I used macros to call the profiler is because it is much more user friendly, instead of having to type gProfiler->enterSection("..."); the user only has to type enterSection("...") and exitSection("..."). One of my worries performance wise was how to handle the issue of interweaving. I solved this by instead of popping off the back of the start times, which works in hierarchical, I would loop through starting from back to front. I figured that as long as there aren't thousands of start times that got added to the vector then it would still be efficient. And it still works perfectly in hierarchical as well because it will just find what it is looking for in the first check of the loop.

The visualization tool I used was Pandas matplot library in python. I chose this because I had heard good things about it, and I have no prior experience using it in any class so I figured why not try it. I was able to use the csv file the profiler created to make bar charts and graphs to visualize the data that was outputted by my tests. Bubble sort 1 is an efficient sorting algorithm whereas bubble sort 2 is an inefficient one. You can see that the second bubble sort takes about twice as long as the first one. This is because there is no swapped flag to check if the vector is already sorted, so it always completes the iteration regardless of if it is sorted or not. The sleep test I ran was to see if the timing was right and if interweaving worked. I profiled a "sleepTestShort" and a "SleepTestLong" which both profile the same test. The difference is that I have it interweaved to where the short section is supposed to be half the time of the long. I also created an efficient and inefficient selection sort algorithm and profiled both of those. The difference between them is that on the inefficient algorithm it always performs a swap, even when it isn't necessary, it iterates over the entire array in the inner loop, and it checks each element in each iteration to see if its sorted. The impact this had it that is about doubled the time it took to execute. I also made

a graph that shows the standard deviation of each profile test. Shown below