experiment/benchmark.cpp

```cpp
/**
 * Benchmarker
 * Author: Arnab Ghosh
 * Date: 11/8/2023
 *
 * Profiles the results of the experiment according to the methodology described in the manuscript
 * Usage: ./benchmark ./eccKEM 1gb_test
*/

// TODO: Add functionality for when a program errors out due to memory constraints.

#include <iostream>
using std::cin;
using std::cout;

#include <string>
using std::string;

#include <vector>
using std::vector;

#include <fstream>
using std::ofstream;
using std::ifstream;

#include <ctime>
using std::time;
using std::time_t;

#include <thread>
using std::thread;

#include <future>
using std::async;

#include <chrono>
using std::chrono::milliseconds;

// Header input
#include "stdlib.h"
#include "stdio.h"
#include "string.h"
#include "sys/times.h"

// Initial time
const time_t INITIAL_TIME = time(NULL);

// TODO: find the include header for C++ asynchronous code

/**
 * Calculates the deltaTime from the initial time defined above.
*/
time_t getDeltaTime() { return time(NULL) - INITIAL_TIME; }
```

```

/**
 * Generates a performance report, and outputs it to a file.
 * TODO: Finish this function.
*/
bool generatePerformanceReport(string filename, int totalTimeIntervals, vector<int> deltaTime, vector<double> memoryUsage, vector<double> cpuUsage) {
    // Create buffer for report
    ofstream report(filename);

    // Create headers
    report << "Record,Time_Interval,Memory_Usage,CPU_Usage,\n";

    // Fill in data
    for(int i = 0; i < totalTimeIntervals; i++) {
        report << i << ","; // Record Number
        report << deltaTime[i] << ","; // Delta Time
        report << memoryUsage[i] << ","; // Memory usage
        report << cpuUsage[i] << ","; // CPU usage
        report << "\n"; // End line
    }

    // Close output stream
    report.close();
    return true;
}

// TODO: we might not want to do this! We need to decide if this is the approach we wish to take.
// /**
//  * Parses specified line in order to find the first number (and this number is usually the number we want to read)
// */
// int parseLine(char* line) {
//     int i = strlen(line);
//     const char* p = line;
//     // Skipping all non-numbers
//     while (*p < '0' || *p > '9') {
//         p++;
//     }

//     line[i - 3] = '\0';
//     i = atoi(p);
//     return i;
// }

// /**
//  * Reads /proc/self/status in order to get total virtual memory usage by this program
// */
// int getVirtualMemory() {
//     // Read the file with current process memory usage
//     FILE* file = fopen("/proc/self/status", "r");
//     int result = -1;
//     char line[128];

//     // Find the VmSize line
//     while (fgets(line, 128, file) != NULL) {
//         if (strncmp(line, "VmSize:", 7) == 0) {
```

```
//             result = parseLine(line);
//             break;
//         }
//     }
// }

/**
 * Returns the current memory usage for the system in megabytes.
 * TODO: replace whatever this is with this more robust approach: https://stackoverflow.com/questions/63166/how-to-determine-cpu-and-memory-consumption-from-inside-a-process
*/
double currentMemoryUsage() {
    // Read from the meminfo file
    ifstream report("/proc/meminfo");

    string scratch;

    // We want to read the first 5 lines; they have following info (in order):
    // MemTotal, MemFree, MemAvailable, Buffers, Cached
    double memTotal;
    double memFree;
    double memAvailable;
    double buffers;
    double cached;

    // Line 1
    report >> scratch;
    report >> memTotal;
    report >> scratch;

    // Line 2
    report >> scratch;
    report >> memFree;
    report >> scratch;

    // Line 3
    report >> scratch;
    report >> memAvailable;
    report >> scratch;

    // Line 4
    report >> scratch;
    report >> buffers;
    report >> scratch;

    // Line 5
    report >> scratch;
    report >> cached;
    report >> scratch;

    // Memory usage is calculated as:
    // memTotal - memFree - buffers + cached
    return memTotal - memFree - buffers + cached;
}

/**
```

```
 * Returns the current memory usage for the system as a percentage.
 * TODO: Finish this function.
*/
double currentCPUUsage() {
    // Helper methods
    struct X {
        /**
         * Returns the deltaClock for user and total CPU cyles of a given process using /proc/stat
        */
        static std::tuple<double, double> deltaClock() {
            // Total clocks
            // Load the memory report
            ifstream report("/proc/stat");

            // Parsing variables
            string scratch;
            double read;

            double user {0};
            double total {0};

            // We have seven numbers to read for total
            // And three numbers to read for user
            report >> scratch;
            report >> read;
            total += read;
            user += read;

            report >> read;
            total += read;
            user += read;

            report >> read;
            total += read;
            user += read;

            // Now we only read into total
            report >> read;
            total += read;

            report >> read;
            total += read;

            report >> read;
            total += read;

            report >> read;
            total += read;

            return std::tuple<double, double> {user, total};
        }
    };
    // Get initial and final
    std::tuple<double, double> initial = X::deltaClock();
    // wait a little bit (so that our values aren't 0)
    std::this_thread::sleep_for(std::chrono::milliseconds(200));
```

```cpp
    std::tuple<double, double> final = X::deltaClock();

    // return Derivative (consider this dU/dX), where U = user and X = total
    // we do (dU/dt) / (dX/dT)
    return (std::get<0>(final) - std::get<0>(initial)) / (std::get<1>(final) - std::get<1>(initial));
}


/**
 * Entry point of program
*/
int main(int argc, char* argv[]) {
    // Parsing arguments
    // We want to join all of the arguments together, and build that as the command
    string command {""};

    // Loop through arguments and append to commandToProfile
    for(int i = 1 /* Skipping the first argument as it will be ./benchmark */; i < argc; i++) {
        command.append(argv[i]);
        command.append(" ");
    }

    // Now we run
    // We need to store some data
    int totalTimeIntervals { 0 };
    vector<int> deltaTime { };
    vector<double> memoryUsage { };
    vector<double> cpuUsage { };

    // 2) PERFORMANCE METRIC 2 - MEMORY
    // TODO: implement performance benchmarking metrics
    // Asynchronously call command
    auto future = async(std::launch::async,
        [command] { return system(command.c_str()); // This is an anonymous lambda
    });

    // Profile future
    while (future.wait_for( milliseconds(500) /* Profile for data every 1/2 second*/)  != std::future_status::ready) {
        // Time to store some data!
        totalTimeIntervals++;
        deltaTime.push_back(getDeltaTime()); // the time interval
        memoryUsage.push_back(currentMemoryUsage()); // the memory usage
        cpuUsage.push_back(currentCPUUsage()); // the CPU usage
    }

    // Generate the performance report and close
    // First, we make a filename signature that joins the arguments together with a dash
    // For example:
    // eccKEM-1gb_test
    string filename {""};
    for (int i = 1; i < argc; i++) {
        filename.append(argv[i]);
        filename.append("-");
    }

    // And we are done
    generatePerformanceReport(filename, totalTimeIntervals, deltaTime, memoryUsage, cpuUsage);
    return 0;
}
```