Национальный исследовательский ядерный университет «МИФИ» (Московский Инженерно–Физический Институт)

Кафедра №42 «Криптология и кибербезопасность»

**Лабораторная работа №2:**

**«Компьютер и время»**

Антон Гатченко Б22-525

2025 г.

*Используемая рабочая среда:*

* Процессор - AMD Ryzen 5 5600H (laptop), 6c/12t
* Оперативная память – DDR4 16 ГБ
* ОС - Windows 10 Pro 22H2 19045.4780, 64 bit
* IDE – GCC/G++ 13.1, OpenMP 201511

*Теоретическая часть:*

1. std::chrono::time\_point - конкретный момент времени, привязанный к определённым часам.

Внутри std::chrono::time\_point хранится значение типа Duration, которое представляет собой количество тиков, прошедших с эпохи (начала отсчёта) часов. Эпоха зависит от типа часов.

1. В Linux RTC представлены через устройство /dev/rtc или /dev/rtc0.

Команды для работы с RTC:

* hwclock --show для вывода времени;
* hwclock --set --date=”…” для установки времени;
* hwclock --hctosys для синхронизации системного времени с RTC;
* hwclock --systohc для синхронизации RTC с системным временем.

1. (и 4) Типы часов, предоставляемые операционной системой (в C++):

* std::chrono::system\_clock - системные часы реального времени, время может быть изменено, например синхронизацией времени;
* std::chrono::steady\_clock - монотонные часы, которые не могут быть отрегулированы назад во времени;
* std::chrono::high\_resolution\_clock - часы с максимально возможным разрешением, зачастую это псевдоним для system\_clock или steady\_clock.

1. Следующей високосной секунды, видимо, не будет.

Последняя была добавлена в ночь с 31 декабря 2016 г. на 1 января 2017 года. Согласно графику на рис. 1, в последние годы величина UT1 – UTC растет значительно медленнее, чем раньше, так что пока нет необходимости в вводе дополнительной секунды.

«В связи с уменьшением величины положительного хода UT1 - UTC, возможно, что "високосная" секунда так и не будет введена до 2035 года, до которого, согласно готовящемуся решению Всемирной конференции радиосвязи (WRC-23), должна применяться действующая шкала UTC, определённая в соответствии с Рекомендацией МСЭ-R TF.460-6.» - [glonass-iac](https://glonass-iac.ru/news/news_gnss/4155/)

«В ноябре 2022 года на очередной Генеральной конференции по мерам и весам (ГКВМ) было предложено отказаться от дополнительных секунд к 2035 году.» - [wikipedia](https://ru.wikipedia.org/wiki/%D0%94%D0%BE%D0%BF%D0%BE%D0%BB%D0%BD%D0%B8%D1%82%D0%B5%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F_%D1%81%D0%B5%D0%BA%D1%83%D0%BD%D0%B4%D0%B0)

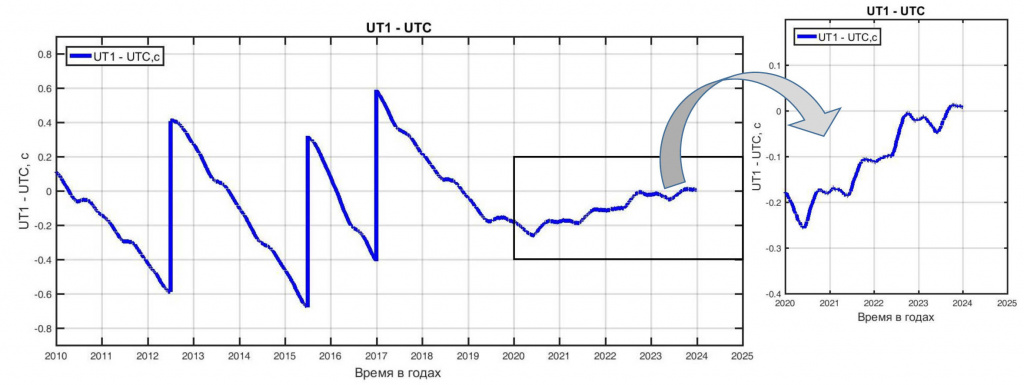


Рисунок 1 - Изменение разницы UT1 и UTC с 2010 по 2024 год (Источник - <https://glonass-iac.ru/news/news_gnss/4155>)

1. Инструкция RDTSC для использования в C++ доступна в компиляторах MSVC и GCC/Clang с помощью функции \_\_rdtsc(). Пример использования \_\_rdtsc() находится в Приложении 7.
2. Чтобы перевести значение TSC (Time-Stamp Counter) в секунды, нужно знать частоту TSC (на современных системах не зависит от текущей частоты процессора). Для моего ПК это значение составило ГГц, что совпадает с базовой частотой процессора. Пример вычисления коэффициента и замеров с его помощью находится в Приложении 8.

Нативно на Windows можно использовать QPC (QueryPerformanceCounter), который официально рекомендуют использовать (и не рекомендуют использовать rdtsc)

«We strongly discourage using the RDTSC or RDTSCP processor instruction to directly query the TSC because you won't get reliable results on some versions of Windows, across live migrations of virtual machines, and on hardware systems without invariant or tightly synchronized TSCs. Instead, we encourage you to use QPC to leverage the abstraction, consistency, and portability that it offers.» - [microsoft](https://learn.microsoft.com/en-us/windows/win32/sysinfo/acquiring-high-resolution-time-stamps#direct-tsc-usage)

Пример использования QPC находится в Приложении 9.

*Практическая часть:*

Так как на Windows нельзя получить доступ к RTC, измерения, связанные с ними, проводились в WSL (дистрибутив Ubuntu 22.04 LTS).

Доступные типы часов операционной системы:

В C: CLOCK\_REALTIME, CLOCK\_MONOTONIC, CLOCK\_THREAD\_CPUTIME\_ID, CLOCK\_PROCESS\_CPUTIME\_ID, CLOCK\_REALTIME\_COARSE.

CLOCK\_REALTIME\_COARSE — это оптимизированный тип часов, предназначенный для быстрого получения грубого значения текущего времени. Он особенно полезен в высокопроизводительных приложениях, где точность времени не является критической. Значения времени кэшируются в памяти процессора и не требуют обращения к аппаратным часам или системным вызовам. Возможно, должно работать только на Linux? Однако при выполнении кода на Windows ошибок не возникало.

В C++: std::chrono::system\_clock, std::chrono::steady\_clock, std::chrono::high\_resolution\_clock.

К сожалению, chrono не предоставляет возможности узнать разрешение через какие-либо методы(.

Таблица 1 - Измерение разрешения и погрешностей таймеров

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Имя таймера** | **Точность, нс (по системному вызову)** | **Разрешение, нс (экспериментально)** | **Погрешность разрешения, нс (средняя абсолютная)** | **Погрешность разрешения, нс (среднеквадратичная)** |
| Часы реального времени (RTC) | - | 1e9 | 0 | 0 |
| CLOCK\_REALTIME | 100 | 100.4 | 0.85 | 102.6 |
| CLOCK\_MONOTONIC | 100 | 100.5 | 0.97 | 114.8 |
| CLOCK\_THREAD\_CPUTIME\_ID | 15625000 | 15625000 | 0 | 0 |
| CLOCK\_PROCESS\_CPUTIME\_ID | 15625000 | 15625000 | 0 | 0 |
| CLOCK\_REALTIME\_COARSE | 15625000 | 992192 | 15391.65 | 81929.12 |
| std::chrono::system\_clock | - | 100.4 | 0.38 | 77.9 |
| std::chrono::steady\_clock | - | 100.4 | 0.39 | 91.4 |
| std::chrono::high\_resolution\_clock | - | 100.5 | 0.51 | 134.9 |
| RDTSC | - | 9.7 | 1.02 | 28.6 |
| QPC | - | 100 | 0.97 | 100.2 |

Таблица 2 - Измерение времени инициализации и возврата ответа

| **Имя таймера** | **Время инициализации, нс** | **Время возврата ответа, нс** |
| --- | --- | --- |
| Часы реального времени (RTC) | - | |
| CLOCK\_REALTIME | 32 | 0.1 |
| CLOCK\_MONOTONIC | 45 | 0.1 |
| CLOCK\_THREAD\_CPUTIME\_ID | 60 | 51 |
| CLOCK\_PROCESS\_CPUTIME\_ID | 83 | 47 |
| CLOCK\_REALTIME\_COARSE | 0.5 | |
| RDTSC | 8.98 | |
| QPC | 20.39 | |

Из-за низкого разрешения значения для RTC не получилось измерить. Значения для RDTSC и QPC получены только как сумма времени инициализации и возврата ответа, так как их начальные значения отличаются от всех остальных, из-за чего таким методом нельзя получить верные данные по этим временам отдельно.

Замеры в таблице 2 могут быть (и скорее всего так оно и есть) довольно неточными, поскольку погрешности не дают найти конкретное решение системы (из-за них выходит, что система не имеет решения). Также некоторые часы отличаются по своему начальному значению, приходится считать это отличие экспериментально, что влияет на точность измерений. С оригинальным выводом программы по замеру этих времен можно ознакомиться в Приложении 6.

*Диаграммы по таблице 2:*

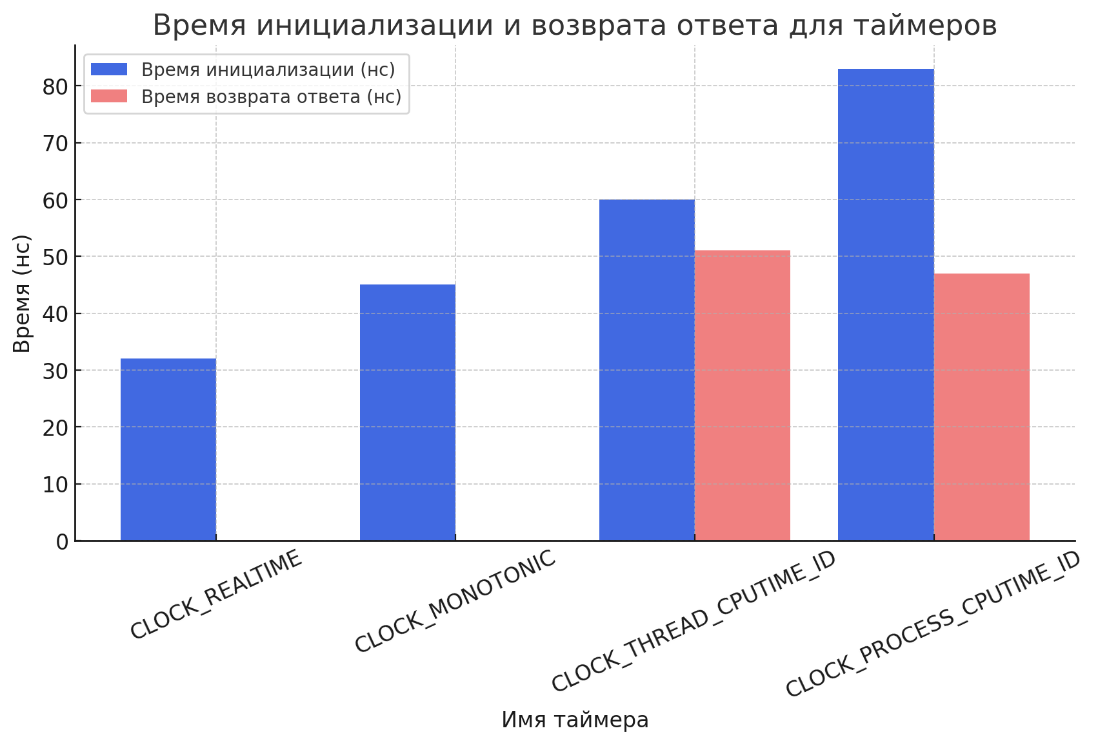


Рисунок 2 - время инициализации и возврата ответа (раздельные)

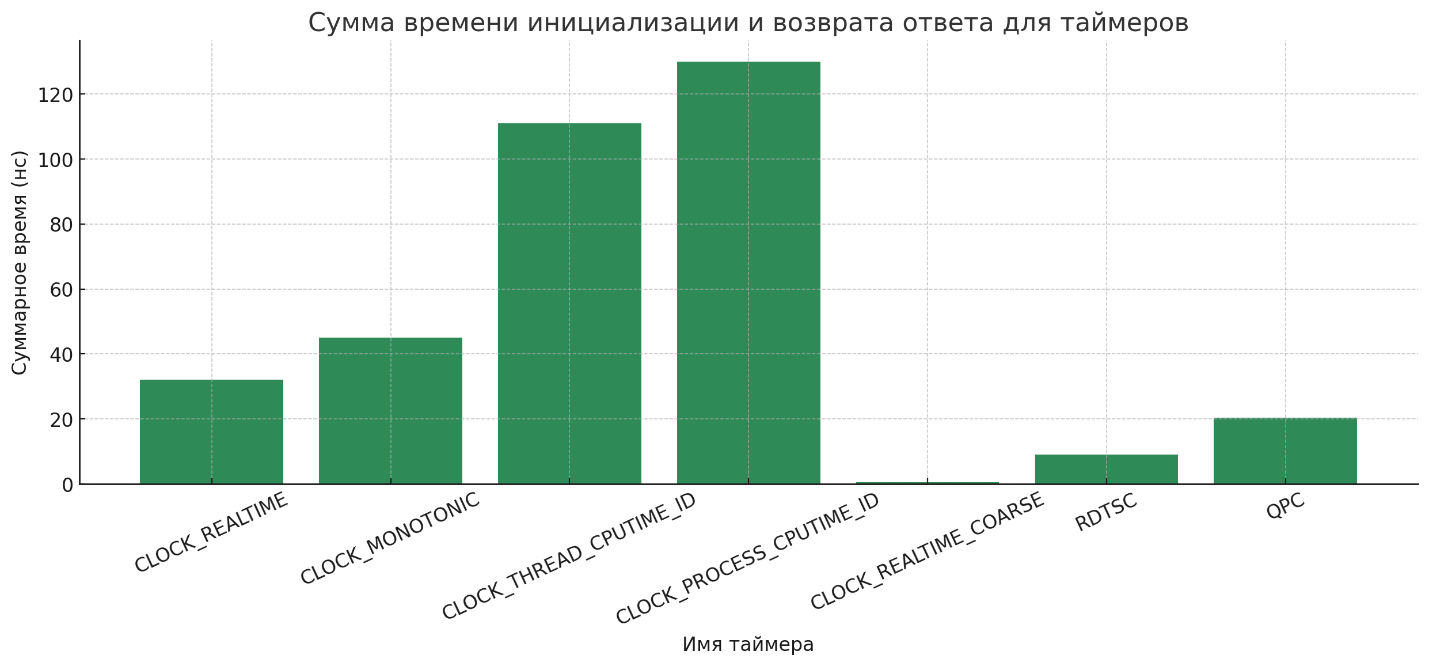


Рисунок 3 - время инициализации и возврата ответа (суммарные)

*Заключение:*

В ходе данной лабораторной работы были изучены различные типы часов, доступные в операционных системах Windows и Linux, а также проведены измерения их характеристик, таких как разрешение, погрешности и время инициализации и возврата ответа. По результатам этих измерений были построены таблицы 1 и 2 и диаграммы.

На Windows доступны CLOCK\_REALTIME, CLOCK\_MONOTONIC, CLOCK\_THREAD\_CPUTIME\_ID, CLOCK\_PROCESS\_CPUTIME\_ID, CLOCK\_REALTIME\_COARSE , std::chrono::system\_clock, std::chrono::steady\_clock, std::chrono::high\_resolution\_clock, RDTSC, QPC.

На Linux также доступны эти часы за исключением QPC, и есть доступ к RTC.

RTC обладает крайне низким разрешением по сравнению с остальными часами (1 секунда, что в раз больше чем, например, у CLOCK\_REALTIME), поэтому для точных измерений он совершенно не подходит.

CLOCK\_REALTIME и CLOCK\_MONOTONIC показали хорошее разрешение около нс с малой погрешностью нс, а также приемлемыми значениями времени инициализации около нс, и временем возврата нс. Их вполне можно использовать для замеров выполнения производительных программ. Лучше отдать предпочтение CLOCK\_MONOTONIC из-за того, что он гарантирует неизменение назад во времени.

Если требуется измерить чистое процессорное время выполнения кода, CLOCK\_THREAD\_CPUTIME\_ID и CLOCK\_PROCESS\_CPUTIME\_ID могут быть предпочтительнее остальных вариантов, однако нужно учитывать их разрешение в мс, что накладывает ограничения на измерение коротких частей программы. Их время инициализации примерно в 2 раза выше, чем у CLOCK\_REALTIME и CLOCK\_MONOTONIC ( и нс соответственно), время возврата около нс.

CLOCK\_REALTIME\_COARSE показал разрешение в мс, и самую высокую погрешность измерения, мкс и мкс средняя абсолютная / среднеквадратичная погрешности соответственно, однако относительно разрешения это приемлемый результат. Время инициализации и ответа в сумме – нс, крайне быстро из-за кэширования на стороне процессора. Может быть отличным вариантом для измерения не самых быстрых частей программы.

Системные часы из C++, std::chrono::system\_clock, std::chrono::steady\_clock, std::chrono::high\_resolution\_clock, показали разрешение около нс с низкой погрешностью, нс нс. Поскольку они основаны на CLOCK\_REALTIME и CLOCK\_MONOTONIC, это ожидаемый результат, и к ним применяется все вышесказанное об этих часах.

RDTSC показал отличные результаты, самое высокое разрешение около нс с низкими погрешностями и нс и хорошим временем инициализации и возврата ответа, суммарно нс. Это предпочтительный вариант для измерения высокопроизводительных и крайне быстрых частей программы. Однако он может давать некорректные результаты на многоядерных системах.

QPC показал результаты, схожие с CLOCK\_REALTIME и CLOCK\_MONOTONIC – разрешение нс, погрешности нс, и обошел их по времени инициализации и возврата ответа – нс в сумме против у них. Несколько разочаровывающий результат, учитывая, что Microsoft рекомендует использовать QPC вместо RDTSC, что не представляется возможным из-за серьезных отличий в разрешении – RDTSC на порядок точнее. Для приложений под Windows QPC может быть предпочтительнее CLOCK\_REALTIME и CLOCK\_MONOTONIC, но если необходима кроссплатформенность, то их можно использовать, в общем-то ничего не теряя.

Подытоживая данные замеры, я бы использовал RDTSC в случае замеров для коротких частей однопоточной программы, где нужна максимальная точность. Для многопоточных программ стоит присмотреться к CLOCK\_THREAD\_CPUTIME\_ID и CLOCK\_PROCESS\_CPUTIME\_ID. В остальных случаях можно использовать CLOCK\_MONOTONIC. RTC я бы не использовал.

*Приложение:*

1. Исходный код программы с измерением параметров системных таймеров в C++:

#include <iostream>  
#include <algorithm>  
#include <chrono>  
#include <vector>  
#include <cmath>  
  
#define MAX\_ITERATIONS (int) 2e8  
  
using namespace std::chrono;  
using std::vector;  
  
vector<double> measure\_time(auto clock\_func){  
 vector<double> intervals(MAX\_ITERATIONS);  
 int count = 0;  
 auto previous\_time = clock\_func();  
  
 for (int i = 0; i < MAX\_ITERATIONS; ++i){  
 auto current\_time = clock\_func();  
  
 if (current\_time != previous\_time){  
 auto delta = duration\_cast<nanoseconds>(current\_time - previous\_time).count();  
 intervals[count++] = static\_cast<double>(delta);  
 previous\_time = current\_time;  
 }  
 }  
 intervals.resize(count);  
 return intervals;  
}  
  
void measure\_deviations(const vector<double> &intervals){  
 double sum = 0.0, mean = 0.0, variance = 0.0, mean\_absolute\_deviation = 0.0;  
  
 // std::cout << \*std::ranges::max\_element(intervals) << std::endl;  
  
 for (double interval: intervals){  
 sum += interval;  
 }  
 mean = sum / intervals.size();  
  
 for (double interval: intervals){  
 mean\_absolute\_deviation += abs(interval - mean);  
 }  
 mean\_absolute\_deviation /= intervals.size();  
  
 for (double interval: intervals){  
 variance += pow(interval - mean, 2);  
 }  
 variance /= intervals.size();  
 double standart\_deviation = std::sqrt(variance);  
  
 std::cout << "Resolution: " << mean << " ns\n";  
 std::cout << "mean\_absolute\_deviation: " << mean\_absolute\_deviation << " ns\n";  
 std::cout << "standart\_deviation: " << standart\_deviation << " ns\n";  
}  
  
int main(){  
 vector<double> intervals = measure\_time(system\_clock::*now*);  
 measure\_deviations(intervals);  
  
 intervals = measure\_time(steady\_clock::*now*);  
 measure\_deviations(intervals);  
  
 intervals = measure\_time(high\_resolution\_clock::*now*);  
 measure\_deviations(intervals);  
 return 0;  
}

1. Исходный код программы с измерением параметров системных таймеров в C:

#include <algorithm>  
#include <iostream>  
#include <cstdio>  
#include <ctime>  
#include <cmath>  
#include <vector>  
  
#define MAX\_ITERATIONS (int) 1e7  
  
using std::vector;  
  
vector<double> measure\_time(const int &clock){  
 struct timespec previous\_time, current\_time;  
 vector<double> intervals(MAX\_ITERATIONS);  
 int count = 0;  
  
 clock\_gettime(clock, &previous\_time);  
  
 for (int i = 0; i < MAX\_ITERATIONS; ++i){  
 clock\_gettime(clock, &current\_time);  
  
 if (current\_time.tv\_sec != previous\_time.tv\_sec || current\_time.tv\_nsec != previous\_time.tv\_nsec){  
 double delta = (current\_time.tv\_sec - previous\_time.tv\_sec) \* 1e9 +  
 (current\_time.tv\_nsec - previous\_time.tv\_nsec);  
 intervals[count++] = delta;  
 previous\_time = current\_time;  
 }  
 }  
 intervals.resize(count);  
 return intervals;  
}  
  
void measure\_deviations(const vector<double> &intervals){  
 double sum = 0.0, mean = 0.0, variance = 0.0, mean\_absolute\_deviation = 0.0;  
  
 for (const double &interval: intervals){  
 sum += interval;  
 }  
 mean = sum / intervals.size();  
  
 for (const double &interval: intervals){  
 mean\_absolute\_deviation += fabs(interval - mean);  
 }  
 mean\_absolute\_deviation /= intervals.size();  
  
 for (const double &interval: intervals){  
 variance += pow(interval - mean, 2);  
 }  
 variance /= intervals.size();  
 double standard\_deviation = sqrt(variance);  
  
 printf("Resolution: %.2f ns\n", mean);  
 printf("Mean Absolute Deviation: %.2f ns\n", mean\_absolute\_deviation);  
 printf("Standard Deviation: %.2f ns\n", standard\_deviation);  
}  
  
void get\_clock\_resolution(const int &clock){  
 struct timespec resolution;  
  
 if (clock\_getres(clock, &resolution) == 0){  
 std::cout << "Resolution of " << clock << ": "  
 << resolution.tv\_sec << " seconds, "  
 << resolution.tv\_nsec << " nanoseconds\n";  
 }  
}  
  
void measure\_initialization\_and\_return(const int &clock1, const int &clock2){  
 struct timespec start\_time, end\_time;  
 double total\_time11 = 0.0, total\_time22 = 0.0, total\_time12 = 0.0, total\_time21 = 0.0;  
 double base\_diff = 0.0;  
  
 if (clock1 == 0 and clock2 == 1){  
 struct timespec clock1\_base, clock2\_base;  
 clock\_gettime(clock1, &clock1\_base);  
 clock\_gettime(clock2, &clock2\_base);  
 base\_diff = (clock2\_base.tv\_sec - clock1\_base.tv\_sec) \* 1e9 + (clock2\_base.tv\_nsec - clock1\_base.tv\_nsec);  
 }  
  
 for (int i = 0; i < MAX\_ITERATIONS; ++i){  
 clock\_gettime(clock1, &start\_time);  
 clock\_gettime(clock1, &end\_time);  
 total\_time11 += (end\_time.tv\_sec - start\_time.tv\_sec) \* 1e9 + (end\_time.tv\_nsec - start\_time.tv\_nsec);  
  
 clock\_gettime(clock2, &start\_time);  
 clock\_gettime(clock2, &end\_time);  
 total\_time22 += (end\_time.tv\_sec - start\_time.tv\_sec) \* 1e9 + (end\_time.tv\_nsec - start\_time.tv\_nsec);  
  
 clock\_gettime(clock1, &start\_time);  
 clock\_gettime(clock2, &end\_time);  
 total\_time12 += (end\_time.tv\_sec - start\_time.tv\_sec) \* 1e9 + (end\_time.tv\_nsec - start\_time.tv\_nsec) -  
 base\_diff;  
  
 clock\_gettime(clock2, &start\_time);  
 clock\_gettime(clock1, &end\_time);  
 total\_time21 += (end\_time.tv\_sec - start\_time.tv\_sec) \* 1e9 + (end\_time.tv\_nsec - start\_time.tv\_nsec) +  
 base\_diff;  
 }  
  
 double avg\_time11 = fabs(total\_time11 / MAX\_ITERATIONS);  
 double avg\_time22 = fabs(total\_time22 / MAX\_ITERATIONS);  
 double avg\_time12 = fabs(total\_time12 / MAX\_ITERATIONS);  
 double avg\_time21 = fabs(total\_time21 / MAX\_ITERATIONS);  
  
 std::cout << "Clock " << clock1 << " and " << clock2 << ":\n";  
  
 std::cout << "A1 + A2 = " << avg\_time11 << " ns\n";  
 std::cout << "B1 + B2 = " << avg\_time22 << " ns\n";  
 std::cout << "A1 + B2 = " << avg\_time21 << " ns\n";  
 std::cout << "B1 + A2 = " << avg\_time12 << " ns\n";  
}  
  
void measure\_all\_clocks\_res\_and\_deviations(){  
 for (int clock: {  
 CLOCK\_REALTIME, CLOCK\_MONOTONIC, CLOCK\_THREAD\_CPUTIME\_ID, CLOCK\_PROCESS\_CPUTIME\_ID, CLOCK\_REALTIME\_COARSE  
 }){  
 const vector<double> intervals = measure\_time(clock);  
 measure\_deviations(intervals);  
 get\_clock\_resolution(clock);  
 }  
}  
  
void measure\_all\_clocks\_initialization\_and\_return(){  
 int CLOCKS\_EVEN\_NUMBER = 6;  
 int clocks[CLOCKS\_EVEN\_NUMBER] = {  
 CLOCK\_REALTIME, CLOCK\_MONOTONIC, CLOCK\_THREAD\_CPUTIME\_ID, CLOCK\_PROCESS\_CPUTIME\_ID, CLOCK\_REALTIME\_COARSE,  
 CLOCK\_REALTIME  
 };  
  
 for (int i = 0; i < CLOCKS\_EVEN\_NUMBER; i += 2){  
 measure\_initialization\_and\_return(clocks[i], clocks[i + 1]);  
 }  
}  
  
int main(){  
 // measure\_all\_clocks\_res\_and\_deviations();  
 measure\_all\_clocks\_initialization\_and\_return();  
 return 0;  
}

1. Исходный код программы с измерением параметров RTC:

#include <iostream>  
#include <fstream>  
#include <fcntl.h>  
#include <unistd.h>  
#include <linux/rtc.h>  
#include <sys/ioctl.h>  
#include <cmath>  
#include <vector>  
  
#define MAX\_ITERATIONS (int) 1e5  
  
using std::vector;  
  
void measure\_rtc\_resolution(){  
 int fd = open("/dev/rtc", O\_RDONLY);  
 if (fd == -1){  
 std::cerr << "Failed to open /dev/rtc: " << std::endl;  
 return;  
 }  
  
 struct rtc\_time previous\_time, current\_time;  
 vector<double> intervals;  
  
  
 if (ioctl(fd, RTC\_RD\_TIME, &previous\_time) == -1){  
 std::cerr << "Failed to read initial RTC time: " << std::endl;  
 close(fd);  
 return;  
 }  
  
 for (int i = 0; i < MAX\_ITERATIONS; ++i){  
 if (ioctl(fd, RTC\_RD\_TIME, &current\_time) == -1){  
 std::cerr << "Failed to read RTC time: " << std::endl;  
 close(fd);  
 return;  
 }  
  
  
 if (current\_time.tm\_sec != previous\_time.tm\_sec ||  
 current\_time.tm\_min != previous\_time.tm\_min ||  
 current\_time.tm\_hour != previous\_time.tm\_hour){  
 double delta = (current\_time.tm\_hour - previous\_time.tm\_hour) \* 3600 +  
 (current\_time.tm\_min - previous\_time.tm\_min) \* 60 +  
 (current\_time.tm\_sec - previous\_time.tm\_sec);  
  
 intervals.push\_back(delta);  
 previous\_time = current\_time;  
 }  
 }  
  
 close(fd);  
  
  
 double sum = 0.0, mean = 0.0, variance = 0.0, mean\_absolute\_deviation = 0.0;  
  
 for (double interval: intervals){  
 sum += interval;  
 }  
 mean = sum / intervals.size();  
  
 for (double interval: intervals){  
 mean\_absolute\_deviation += std::fabs(interval - mean);  
 variance += std::pow(interval - mean, 2);  
 }  
 mean\_absolute\_deviation /= intervals.size();  
 variance /= intervals.size();  
 double standard\_deviation = std::sqrt(variance);  
  
  
 std::cout << "Resolution (average interval): " << mean << " seconds\n";  
 std::cout << "Mean Absolute Deviation: " << mean\_absolute\_deviation << " seconds\n";  
 std::cout << "Standard Deviation: " << standard\_deviation << " seconds\n";  
}  
  
int main(){  
 measure\_rtc\_resolution();  
 return 0;  
}

1. Исходный код программы с измерением параметров \_\_rdtsc():

#include <iostream>  
#include <vector>  
#include <cmath>  
#include <intrin.h>  
#include <thread>  
#include <chrono>  
  
#define MAX\_ITERATIONS (int) 1e8  
  
using std::vector;  
  
double get\_tsc\_frequency(){  
 unsigned long long start\_tsc = \_\_rdtsc();  
 auto start = std::chrono::high\_resolution\_clock::*now*();  
 std::this\_thread::sleep\_for(std::chrono::milliseconds(100));  
 unsigned long long end\_tsc = \_\_rdtsc();  
 auto end = std::chrono::high\_resolution\_clock::*now*();  
 std::chrono::duration<double> elapsed = end - start;  
 return (end\_tsc - start\_tsc) / elapsed.count(); // TSC в Гц  
}  
  
void measure\_rdtsc\_resolution(){  
 vector<double> intervals(MAX\_ITERATIONS);  
 int count = 0;  
 unsigned long long previous\_tsc = \_\_rdtsc();  
  
 for (int i = 0; i < MAX\_ITERATIONS; ++i){  
 unsigned long long current\_tsc = \_\_rdtsc();  
  
 if (current\_tsc != previous\_tsc){  
 double delta = static\_cast<double>(current\_tsc - previous\_tsc);  
 intervals[count++] = delta;  
 previous\_tsc = current\_tsc;  
 }  
 }  
  
 intervals.resize(count);  
  
 double sum = 0.0, mean = 0.0, variance = 0.0, mean\_absolute\_deviation = 0.0;  
  
 for (double interval: intervals){  
 sum += interval;  
 }  
 mean = sum / intervals.size();  
  
 for (double interval: intervals){  
 mean\_absolute\_deviation += std::fabs(interval - mean);  
 variance += std::pow(interval - mean, 2);  
 }  
 mean\_absolute\_deviation /= intervals.size();  
 variance /= intervals.size();  
 double standard\_deviation = std::sqrt(variance);  
 double tsc\_frequency = get\_tsc\_frequency();  
  
 std::cout << "Resolution (average interval): " << mean / tsc\_frequency \* 1e9 << " ns\n";  
 std::cout << "Mean Absolute Deviation: " << mean\_absolute\_deviation / tsc\_frequency \* 1e9 << " ns\n";  
 std::cout << "Standard Deviation: " << standard\_deviation / tsc\_frequency \* 1e9 << " ns\n";  
 std::cout << "tsc\_frequency: " << tsc\_frequency / 1e9 << " GHz\n";  
}  
  
int main(){  
 measure\_rdtsc\_resolution();  
 return 0;  
}

1. Исходный код программы с измерением параметров QPC:

#include <iostream>  
#include <vector>  
#include <cmath>  
#include <windows.h>  
  
#define MAX\_ITERATIONS (int) 1e8  
  
void measure\_qpc\_resolution(){  
 LARGE\_INTEGER frequency;  
 QueryPerformanceFrequency(&frequency);  
  
 LARGE\_INTEGER previous\_time, current\_time;  
 std::vector<double> intervals(MAX\_ITERATIONS);  
 int count = 0;  
  
 QueryPerformanceCounter(&previous\_time);  
  
 for (int i = 0; i < MAX\_ITERATIONS; ++i){  
 QueryPerformanceCounter(&current\_time);  
  
 if (current\_time.QuadPart != previous\_time.QuadPart){  
 double delta = static\_cast<double>(current\_time.QuadPart - previous\_time.QuadPart);  
 intervals[count++] = delta;  
 previous\_time = current\_time;  
 }  
 }  
  
 intervals.resize(count);  
  
 double sum = 0.0, mean = 0.0, variance = 0.0, mean\_absolute\_deviation = 0.0;  
  
 for (double interval: intervals){  
 sum += interval;  
 }  
 mean = sum / intervals.size();  
  
 for (double interval: intervals){  
 mean\_absolute\_deviation += std::fabs(interval - mean);  
 variance += std::pow(interval - mean, 2);  
 }  
 mean\_absolute\_deviation /= intervals.size();  
 variance /= intervals.size();  
 double standard\_deviation = std::sqrt(variance);  
  
 std::cout << "Resolution (average interval): " << mean / frequency.QuadPart \* 1e9 << " ns\n";  
 std::cout << "Mean Absolute Deviation: " << mean\_absolute\_deviation / frequency.QuadPart \* 1e9 << " ns\n";  
 std::cout << "Standard Deviation: " << standard\_deviation / frequency.QuadPart \* 1e9 << " ns\n";  
}  
  
int main(){  
 measure\_qpc\_resolution();  
 return 0;  
}

1. Вывод измерения времени инициализации и возврата ответа:

Clock 0 and 1:  
A1 + A2 = 32.7302 ns  
B1 + B2 = 29.3478 ns  
A1 + B2 = 332.348 ns  
B1 + A2 = 255.815 ns  
Clock 3 and 2:  
A1 + A2 = 106.25 ns  
B1 + B2 = 143.75 ns  
A1 + B2 = 134.375 ns  
B1 + A2 = 125 ns  
Clock 4 and 0:  
A1 + A2 = 0.20001 ns  
B1 + B2 = 33.0242 ns  
A1 + B2 = 500292 ns  
B1 + A2 = 500333 ns

1. Пример использования \_\_rdtsc():

#include <iostream>  
#include <intrin.h> // Для \_\_rdtsc()  
int main() {  
 unsigned long long timestamp = \_\_rdtsc();  
 std::cout << "Timestamp: " << timestamp << "\n";  
 return 0;  
}

1. Пример вычисления коэффициента TSC и замеров времени выполнения с помощью RDTSC:

#include <iostream>  
#include <intrin.h>  
#include <thread>  
#include <chrono>  
double get\_tsc\_frequency(){  
 unsigned long long start\_tsc = \_\_rdtsc();  
 auto start = std::chrono::high\_resolution\_clock::*now*();  
 std::this\_thread::sleep\_for(std::chrono::milliseconds(100));  
 unsigned long long end\_tsc = \_\_rdtsc();  
 auto end = std::chrono::high\_resolution\_clock::*now*();  
 std::chrono::duration<double> elapsed = end - start;  
 return (end\_tsc - start\_tsc) / elapsed.count(); // TSC в Гц  
}  
void measure\_time(double tsc\_freq){  
 unsigned long long start\_tsc = \_\_rdtsc();  
 auto start = std::chrono::high\_resolution\_clock::*now*();  
 std::this\_thread::sleep\_for(std::chrono::milliseconds(100));  
 unsigned long long end\_tsc = \_\_rdtsc();  
 auto end = std::chrono::high\_resolution\_clock::*now*();  
 std::chrono::duration<double> elapsed = end - start;  
 unsigned long long elapsed\_tsc = end\_tsc - start\_tsc;  
 double elapsed\_seconds = elapsed\_tsc / tsc\_freq;  
 std::cout << "Measured Sleep Time: " << elapsed.count() << " seconds (chrono)" << std::endl;  
 std::cout << "Measured Sleep Time: " << elapsed\_seconds << " seconds (TSC)" << std::endl;  
}  
int main(){  
 double tsc\_freq = get\_tsc\_frequency();  
 std::cout << "TSC Frequency: " << tsc\_freq / 1e9 << " GHz" << std::endl;  
 measure\_time(tsc\_freq);  
 return 0;  
}

1. Пример использования QPC:

#include <chrono>  
#include <iostream>  
#include <thread>  
#include <windows.h>  
int main(){  
 LARGE\_INTEGER StartingTime, EndingTime, ElapsedMicroseconds;  
 LARGE\_INTEGER Frequency;  
 QueryPerformanceFrequency(&Frequency);  
 QueryPerformanceCounter(&StartingTime);  
 std::this\_thread::sleep\_for(std::chrono::milliseconds(100));  
 QueryPerformanceCounter(&EndingTime);  
 ElapsedMicroseconds.QuadPart = EndingTime.QuadPart - StartingTime.QuadPart;  
 double elapsed\_time = ElapsedMicroseconds.QuadPart;  
 elapsed\_time /= Frequency.QuadPart;  
 std::cout << elapsed\_time << std::endl;  
 return 0;  
}