**Университет ИТМО**

Факультет программной инженерии и компьютерной техники

**Лабораторная работа №1**

По дисциплине “Операционные системы”

Выполнил:

Студент группы P3332:

**Терновский** И.Е.

Вариант: windows\CreateProcess\io-thpt-read\linreg\block size: 8K

Санкт-Петербург

2024

Оглавление

[**Задание Часть 1. Запуск программ** 2](#_Toc182948934)

[**Задание Часть 2. Мониторинг и профилирование** 2](#_Toc182948935)

[**Выполнение:** 3](#_Toc182948936)

[Часть 1: 3](#_Toc182948937)

[Часть 2: 5](#_Toc182948938)

[Предположения: 6](#_Toc182948939)

[Пункт 1.1 6](#_Toc182948940)

[Пункт 1.2 10](#_Toc182948941)

[Пункт 2 13](#_Toc182948942)

[Пункт 3 14](#_Toc182948943)

[Пункт 4 14](#_Toc182948944)

[Пункт 5 15](#_Toc182948945)

[**Вывод** 16](#_Toc182948946)

# **Задание Часть 1. Запуск программ**

Необходимо реализовать собственную оболочку командной строки - shell. Выбор ОС для реализации производится на усмотрение студента. Shell должен предоставлять пользователю возможность запускать программы на компьютере с переданными аргументами командной строки и после завершения программы показывать реальное время ее работы (подсчитать самостоятельно как «время завершения» – «время запуска»).

# **Задание Часть 2. Мониторинг и профилирование**

Разработать комплекс программ-нагрузчиков по варианту, заданному преподавателем. Каждый нагрузчик должен, как минимум, принимать параметр, который определяет количество повторений для алгоритма, указанного в задании. Программы должны нагружать вычислительную систему, дисковую подсистему или обе подсистемы сразу. Необходимо скомпилировать их без опций оптимизации компилятора.

Перед запуском нагрузчика, попробуйте оценить время работы вашей программы или ее результаты (если по варианту вам досталось измерение чего-либо). Постарайтесь обосновать свои предположения. Предположение можно сделать, основываясь на свой опыт, знания ОС и характеристики используемого аппаратного обеспечения.

1. Запустите программу-нагрузчик и зафиксируйте метрики ее работы с помощью инструментов для профилирования. Сравните полученные результаты с ожидаемыми. Постарайтесь найти объяснение наблюдаемому.
2. Определите количество нагрузчиков, которое эффективно нагружает все ядра процессора на вашей системе. Как распределяются времена USER%, SYS%, WAIT%, а также реальное время выполнения нагрузчика, какое количество переключений контекста (вынужденных и невынужденных) происходит при этом?
3. Увеличьте количество нагрузчиков вдвое, втрое, вчетверо. Как изменились времена, указанные на предыдущем шаге? Как ведет себя ваша система?
4. Объедините программы-нагрузчики в одну, реализованную при помощи потоков выполнения, чтобы один нагрузчик эффективно нагружал все ядра вашей системы. Как изменились времена для того же объема вычислений? Запустите одну, две, три таких программы.
5. Добавьте опции агрессивной оптимизации для компилятора. Как изменились времена? На сколько сократилось реальное время исполнения программы нагрузчика?

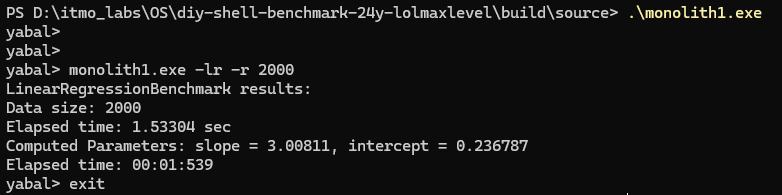
# **Выполнение:**

## Часть 1:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125 | vector<string> split(**const** string& str, **char** delimiter) {  vector<string> tokens;  string token;  istringstream **tokenStream**(str);  **while** (getline(tokenStream, token, delimiter)) {  tokens.push\_back(token);  }  **return** tokens;  }  string formatElapsedTime(**double** elapsedTime) {  **int** minutes = **static\_cast**<**int**>(elapsedTime) / **60**;  **int** seconds = **static\_cast**<**int**>(elapsedTime) % **60**;  **int** milliseconds = **static\_cast**<**int**>((elapsedTime - **static\_cast**<**int**>(elapsedTime)) \* **1000**);  **char** buffer[**50**];  snprintf(buffer, **sizeof**(buffer), "%02d:%02d:%03d", minutes, seconds, milliseconds);  **return** **string**(buffer);  }  **void** RunShell(**bool** isTestMode) {  vector<string> history;  **int** historyIndex = -**1**;  string input;  **while** (true) {  **if** (!isTestMode) {  cout << "yabal> ";  }  input.clear();  **int** ch;  **while** (true) {  ch = \_getch();  **if** (ch == **3**) {  cout << "**\n** Exiting shell**\n**";  exit(**0**);  } **else** **if** (ch == **13**) { // Enter key  cout << endl;  **break**;  } **else** **if** (ch == **8**) { // Backspace key  **if** (!input.empty()) {  input.pop\_back();  cout << "**\b** **\b**";  }  } **else** **if** (ch == **224**) { // First byte of arrow key  ch = \_getch();  **if** (ch == **72**) { // Up arrow  **if** (historyIndex > **0**) {  historyIndex--;  input = history[historyIndex];  // Weird hack to clear the line and print the new input  cout << "**\r**" << "yabal> " << input << string(**50**, ' ') << "**\r**" << "yabal> " << input;  }  } **else** **if** (ch == **80**) { // Down arrow  **if** (historyIndex < **static\_cast**<**int**>(history.size()) - **1**) {  historyIndex++;  input = history[historyIndex];  // Weird hack to clear the line and print the new input  cout << "**\r**" << "yabal> " << input << string(**50**, ' ') << "**\r**" << "yabal> " << input;  } **else** {  historyIndex = history.size();  input.clear();  // Weird hack to clear the line and print the new input  cout << "**\r**" << "yabal> " << string(**50**, ' ') << "**\r**" << "yabal> ";  }  }  } **else** {  input += **static\_cast**<**char**>(ch);  cout << **static\_cast**<**char**>(ch);  }  }  **if** (input == "exit") {  **break**;  }  **if** (!input.empty()) {  history.push\_back(input);  historyIndex = history.size();  }  vector<string> args = split(input, ' ');  **if** (args.empty()) {  **continue**;  }  string command = args[**0**];  string arguments;  **for** (**size\_t** i = **1**; i < args.size(); ++i) {  arguments += args[i] + " ";  }  STARTUPINFOA si;  PROCESS\_INFORMATION pi;  ZeroMemory(&si, **sizeof**(si));  si.cb = **sizeof**(si);  ZeroMemory(&pi, **sizeof**(pi));  string cmdLine = command + " " + arguments;  chrono::high\_resolution\_clock::time\_point start = chrono::high\_resolution\_clock::now();  **if** (!CreateProcess(  nullptr,  **const\_cast**<**char**\*>(cmdLine.c\_str()),  nullptr,  nullptr,  FALSE,  **0**,  nullptr,  nullptr,  &si,  &pi  )) {  cerr << "CreateProcess failed (" << GetLastError() << ").**\n**";  **continue**;  }  WaitForSingleObject(pi.hProcess, INFINITE);  chrono::high\_resolution\_clock::time\_point end = chrono::high\_resolution\_clock::now();  chrono::duration<**double**> elapsedTime = end - start;  cout << "Elapsed time: " << formatElapsedTime(elapsedTime.count()) << "**\n**";  CloseHandle(pi.hProcess);  CloseHandle(pi.hThread);  }  } |

Листинг Исходный код модуля shell

**Пример работы:**



## Часть 2:

### Предположения:

Предполагаю, что нагрузчик проверяющий io будет выдавать скорость чтения примерно равную заявленной производителем диска, на котором будут проводится тесты. При этом скорее всего она будет ниже нежели заявленная, связано это с тем, что читаем мы неоптимизированными способами, нестандартными блоками и без буфера.

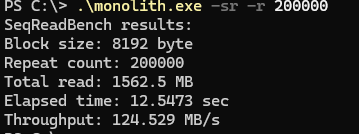
Для нагрузчика с линейной регрессией строить предположения касательно скорости выполнения сложно, скорее всего запуск с входными данными меньше 1 000 будет выполняться меньше секунды, далее с добавлением каждого нуля время работы будет расти чуть быстрее чем линейно, так как чем больше чисел для построения регрессии, тем чаще они будут вытесняться и тем медленнее будет происходить переключение контекста.

Пункт 1.1

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78 | **void** **SequentialReadBenchmark**(**size\_t** repeatCount) {  **const** **size\_t** blockSize = **8192**; // 8KB block size  **const** **size\_t** fileSize = blockSize \* repeatCount; // Total file size  **const** **size\_t** alignment = **4096**; // Alignment for FILE\_FLAG\_NO\_BUFFERING  **void**\* buffer;  // Allocate aligned buffer  buffer = \_aligned\_malloc(blockSize, alignment);  **if** (!buffer) {  std::cerr << "Failed to allocate aligned buffer.**\n**";  **return**;  }  // Write the file  HANDLE hFile = CreateFile(  "benchmark\_output.dat", GENERIC\_WRITE, **0**, NULL, CREATE\_ALWAYS, FILE\_FLAG\_NO\_BUFFERING, NULL  );  **if** (hFile == INVALID\_HANDLE\_VALUE) {  std::cerr << "Can't open file to write.**\n**";  \_aligned\_free(buffer);  **return**;  }  **for** (**size\_t** i = **0**; i < repeatCount; ++i) {  // Fill buffer with random data for each block  **for** (**size\_t** j = **0**; j < blockSize; ++j) {  **static\_cast**<**char**\*>(buffer)[j] = **static\_cast**<**char**>(std::rand() % **256**);  }  DWORD written;  **if** (!WriteFile(hFile, buffer, blockSize, &written, NULL) || written != blockSize) {  std::cerr << "Error writing to file.**\n**";  CloseHandle(hFile);  \_aligned\_free(buffer);  **return**;  }  }  CloseHandle(hFile);  // Read the file  hFile = CreateFile(  "benchmark\_output.dat", GENERIC\_READ, **0**, NULL, OPEN\_EXISTING, FILE\_FLAG\_NO\_BUFFERING, NULL  );  **if** (hFile == INVALID\_HANDLE\_VALUE) {  std::cerr << "Can't open file to read.**\n**";  \_aligned\_free(buffer);  **return**;  }  **auto** start = std::chrono::high\_resolution\_clock::now();  **for** (**size\_t** i = **0**; i < repeatCount; ++i) {  DWORD readBytes;  **if** (!ReadFile(hFile, buffer, blockSize, &readBytes, NULL) || readBytes != blockSize) {  std::cerr << "Error reading from file.**\n**";  CloseHandle(hFile);  \_aligned\_free(buffer);  **return**;  }  }  CloseHandle(hFile);  **auto** end = std::chrono::high\_resolution\_clock::now();  std::chrono::duration<**double**> elapsed = end - start;  **double** totalBytes = **static\_cast**<**double**>(fileSize);  **double** throughput = totalBytes / elapsed.count(); // Bytes per second  std::cout << "SeqReadBench results:**\n**";  std::cout << "Block size: " << blockSize << " byte**\n**";  std::cout << "Repeat count: " << repeatCount << "**\n**";  std::cout << "Total read: " << totalBytes / (**1024** \* **1024**) << " MB**\n**";  std::cout << "Elapsed time: " << elapsed.count() << " sec**\n**";  std::cout << "Throughput: " << throughput / (**1024** \* **1024**) << " MB/s**\n**";  \_aligned\_free(buffer);  } |

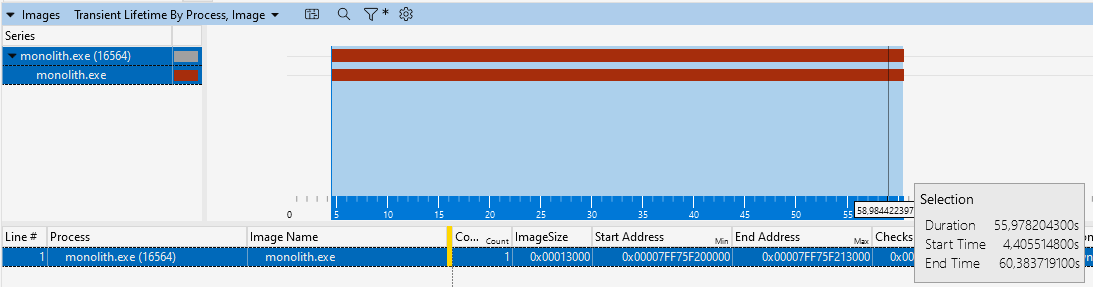
Листинг Код нагрузчика 1

Для тестирования выбрал 200000 блоков, чтобы диск успел прогреться и скорость чтения была больше похожа на реальную.

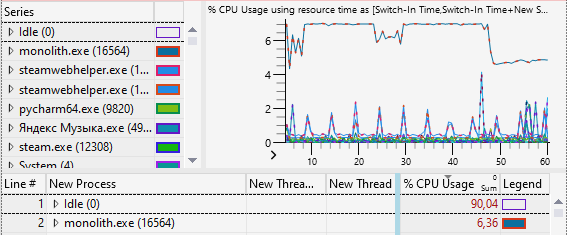


Изображение Вывод нагрузчика

Полученный результат сильно меньше заявленной скорости диска от производителя(3500МБ/с), как и было описано в предположениях скорее всего это связано с тем, что мы читаем специфическим образом со специфическим размером блока. Так же не стоит забывать, что сбор различных статистик так же влияет на скорость работы с диском (так как диск системный). К тому же, мы запускаем процесс в одном потоке, поэтому вполне возможно узкое горлышко в производительности одного потока.

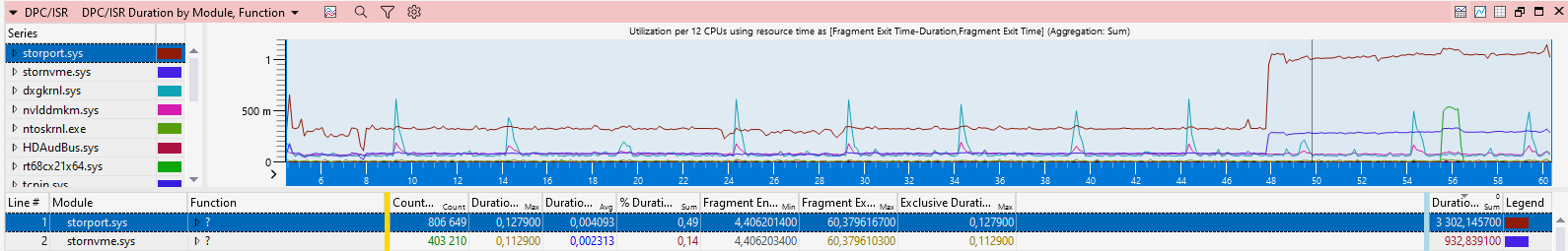


Изображение Полное время работы из профилировщика



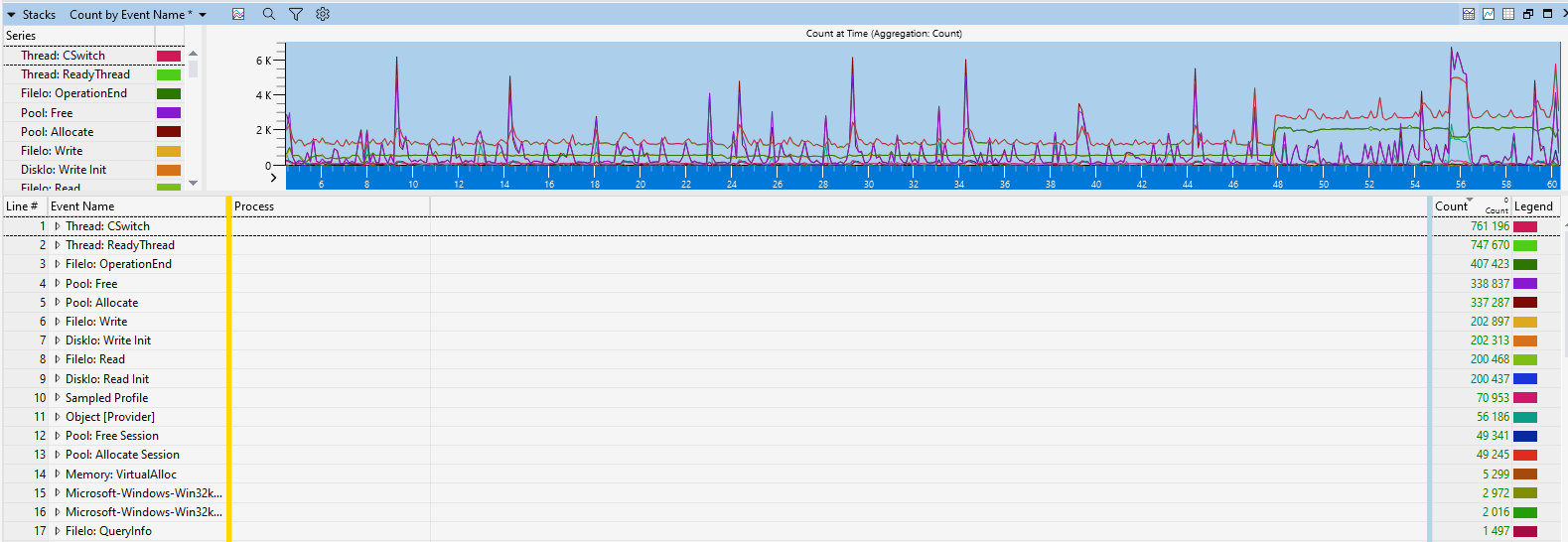
Изображение Использование ЦП и Idle

Так как процесс запускается в одном потоке, то он чисто физически не может забрать больше ресурсов процессора



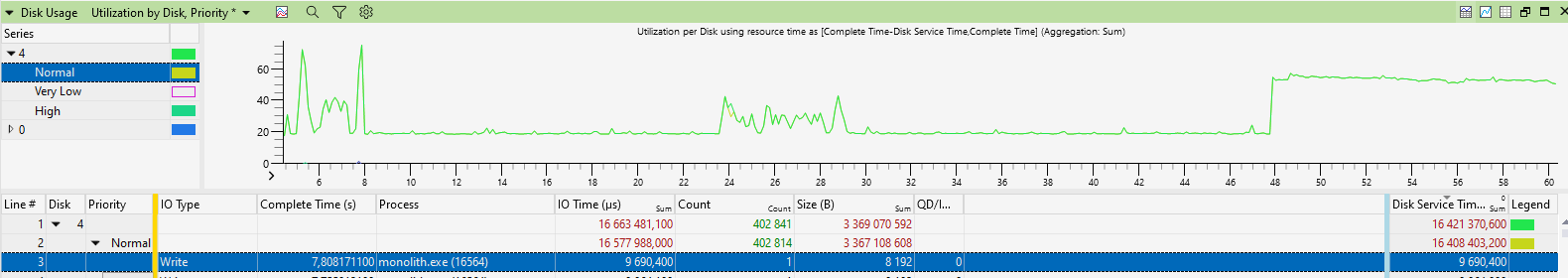
Изображение График использования различных системных модулей

Можно заметить, что самые используемые модули это storport и stornvme, оба модуля используются для записи и чтения файлов по высокоскоростным шинам (nvme в данном случае).



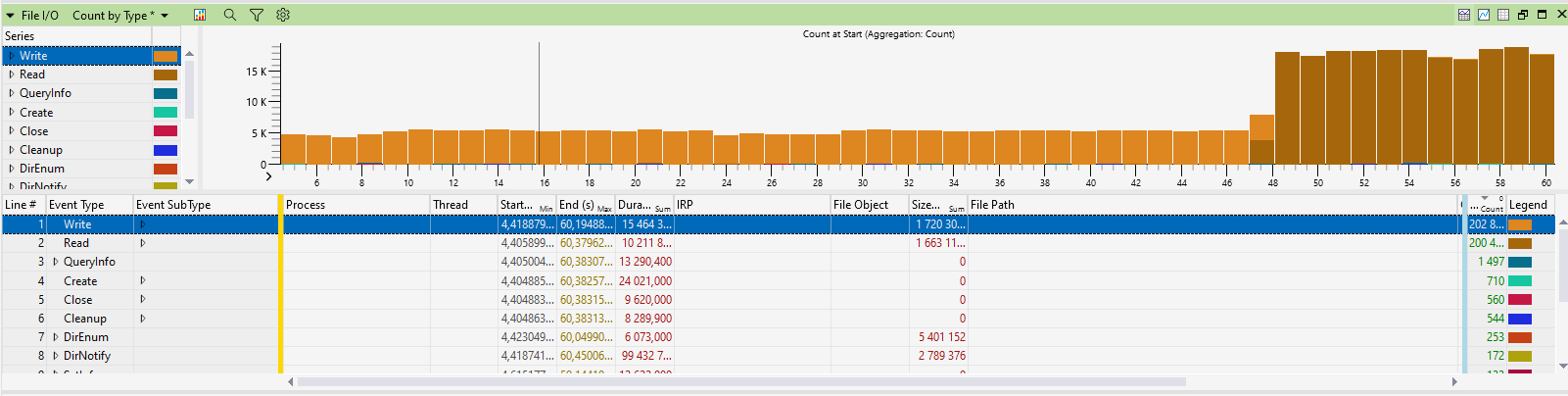
Изображение График событий

Из графика событий можно увидеть сколько переключений контекста произошло, а также по операциям FileIo\DiskIo можно понять, что процесс, который мы профилируем работает с файлами.



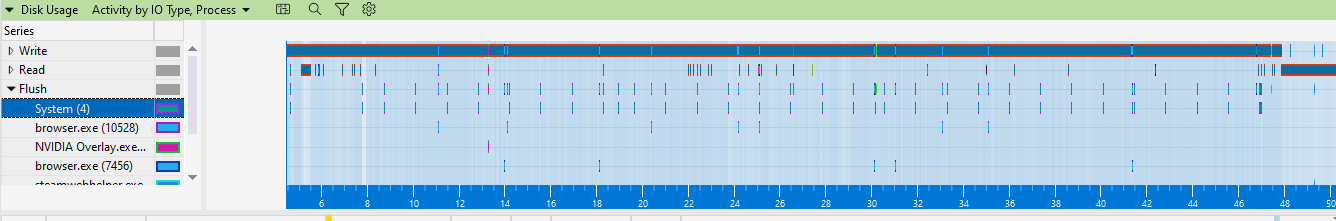
Изображение График использования диска

Можно заметить, что наши операции чтения\записи имеют нормальный приоритет, можно предположить, что если выставить приоритет процессу выше, то результат будет лучше.



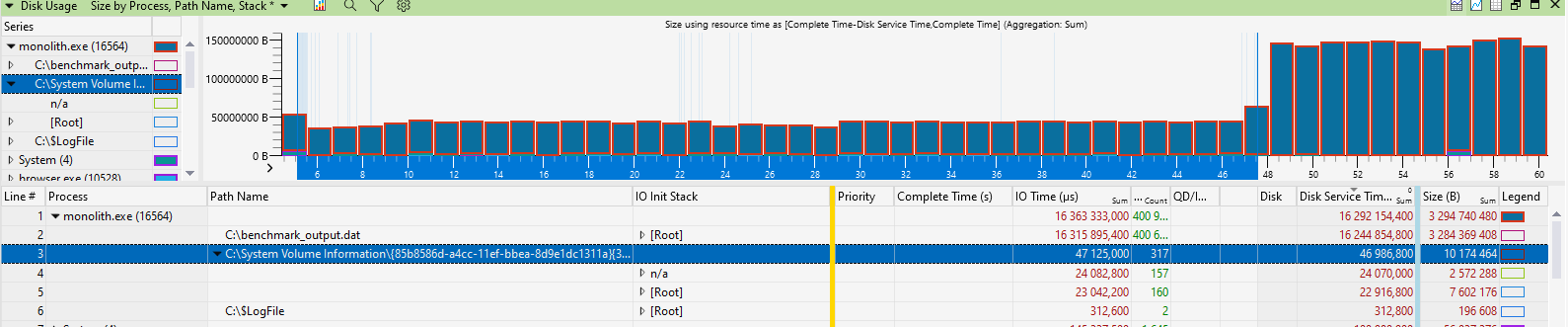
Изображение График IO

Из данного графика видно, что происходит ровно то, что и ожидалось, а именно, сначала файл долго записывается блоками по 8кб, а затем сильно быстрее читается такими же блоками, так же можно проследить и другие вызовы, по типу Create, Close, Cleanup.



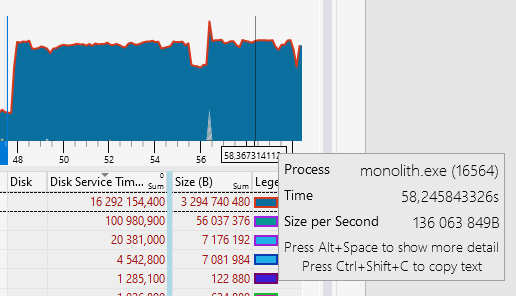
Изображение использование диска по операциям

Интересно что диск не выполняет flush для нагрузчика, связано это скорее всего с тем что мы не используем буфер.



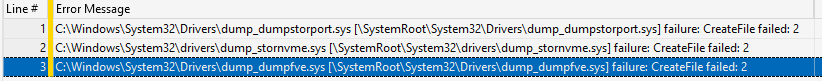
Изображение дополнительные записи

Так же интересно что помимо основного файла, процесс также записывает данные в файлы логов и системные файлы, которые хранят историю файлов, данные индексации и т.д.



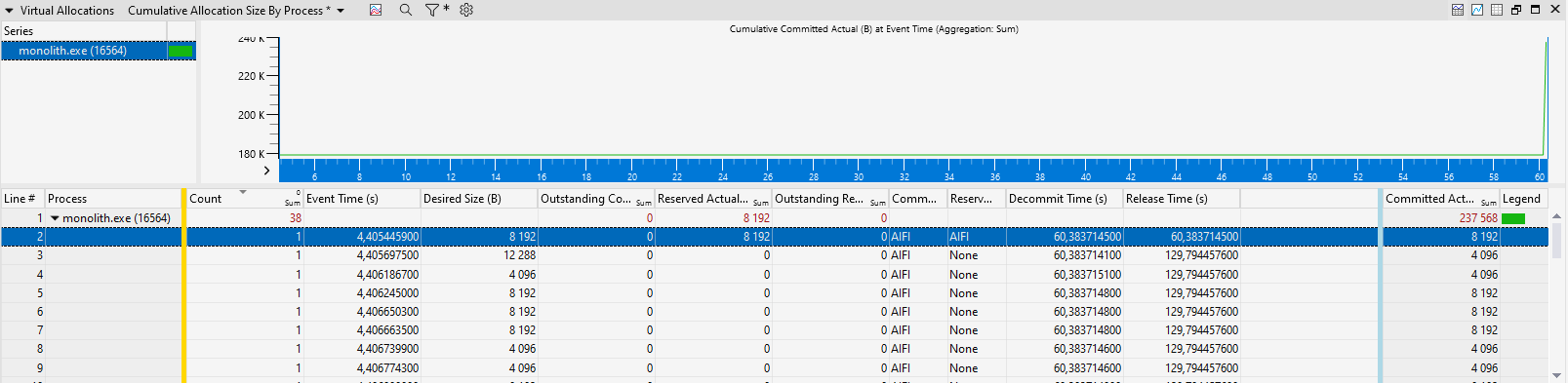
Изображение скорость чтения из профилировщика

Здесь видно, что скорость, которую выводит нагрузчик очень близка к реальной скорости, с которой работает наш диск.



Изображение Ошибочки

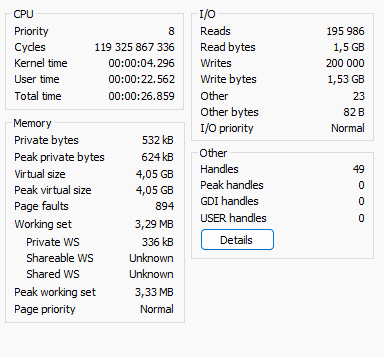
Во время выполнения бенчмарка случилось 3 ошибки создания файла)



Изображение График виртуального выделения памяти

Из данного графика видно, что нагрузчик просит выделения памяти по 8кб, при этом, что интересно, выделяет память он по 8кб, но при этом почти сразу всю???

Статистика из другого мониторинга (Process hacker он же Process explorer)



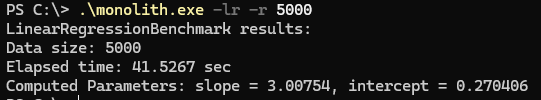
\* немного не хватает операций чтения т.к. метрики собираются раз в 0.5с.

### Пункт 1.2

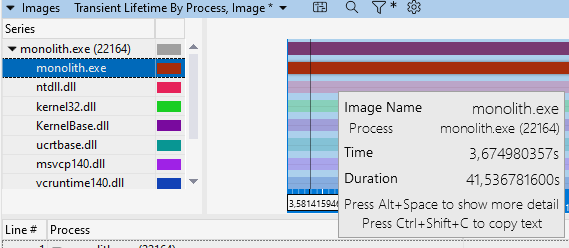
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42 | **void** **LinearRegressionBenchmark**(**size\_t** dataSize) {  // Generate random data  std::vector<**double**> x(dataSize);  std::vector<**double**> y(dataSize);  **for** (**size\_t** i = **0**; i < dataSize; ++i) {  x[i] = **static\_cast**<**double**>(std::rand()) / RAND\_MAX \* **1000.0**;  y[i] = **3.0** \* x[i] + **5.0** +  ((**static\_cast**<**double**>(std::rand()) / RAND\_MAX) - **0.5**) \* **50.0**; // y = 3x + 5 + noise  }  **auto** start = std::chrono::high\_resolution\_clock::now();  // Perform linear regression using gradient descent  **double** slope = **0.0**;  **double** intercept = **0.0**;  **const** **double** learning\_rate = **0.0000001**;  **const** **size\_t** iterations = **1000000**;  **for** (**size\_t** iter = **0**; iter < iterations; ++iter) {  **double** slope\_gradient = **0.0**;  **double** intercept\_gradient = **0.0**;  **for** (**size\_t** i = **0**; i < dataSize; ++i) {  **double** prediction = slope \* x[i] + intercept;  **double** error = prediction - y[i];  slope\_gradient += (**2.0** / dataSize) \* x[i] \* error;  intercept\_gradient += (**2.0** / dataSize) \* error;  }  slope -= learning\_rate \* slope\_gradient;  intercept -= learning\_rate \* intercept\_gradient;  }  **auto** end = std::chrono::high\_resolution\_clock::now();  std::chrono::duration<**double**> elapsed = end - start;  std::cout << "LinearRegressionBenchmark results:**\n**";  std::cout << "Data size: " << dataSize << "**\n**";  std::cout << "Elapsed time: " << elapsed.count() << " sec**\n**";  std::cout << "Computed Parameters: slope = " << slope << ", intercept = " << intercept << "**\n**";  } |

Листинг Код нагрузчика 2

Выбрал 5000 повторений алгоритма, так как это дает запас по времени выполнения для дальнейших тестов в мультипотоке.

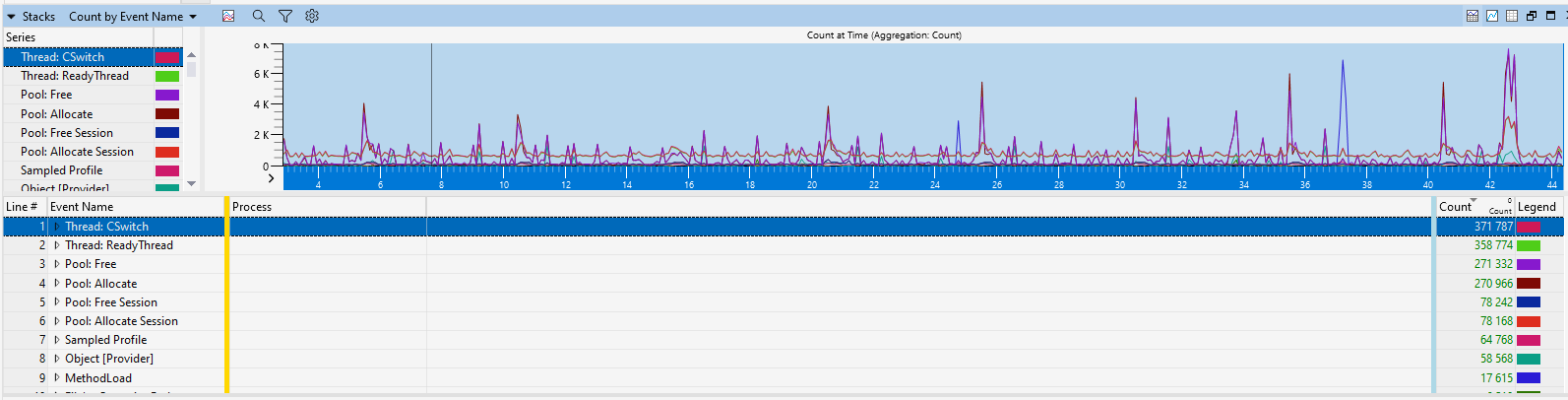


Изображение Результат 2 нагрузчика



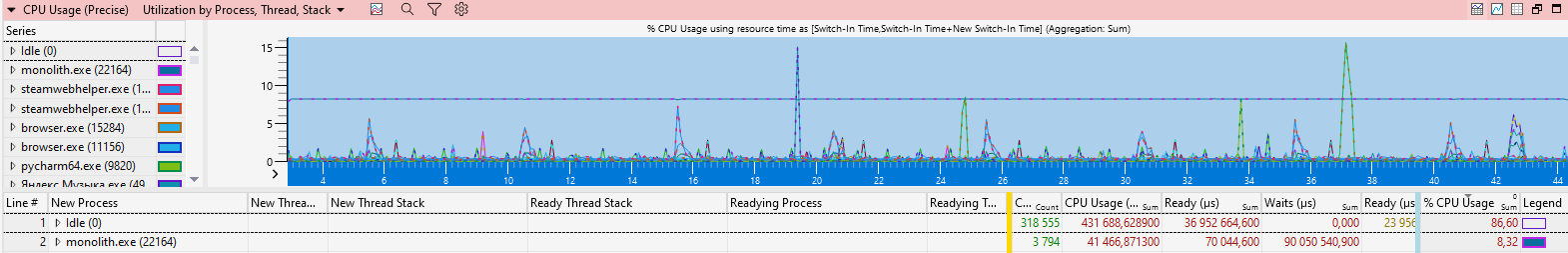
Изображение Время работы нагрузчика

Интересно заметить, что время работы всего процесса и время, замеренное в нагрузчике, отличаются всего на 0.01с



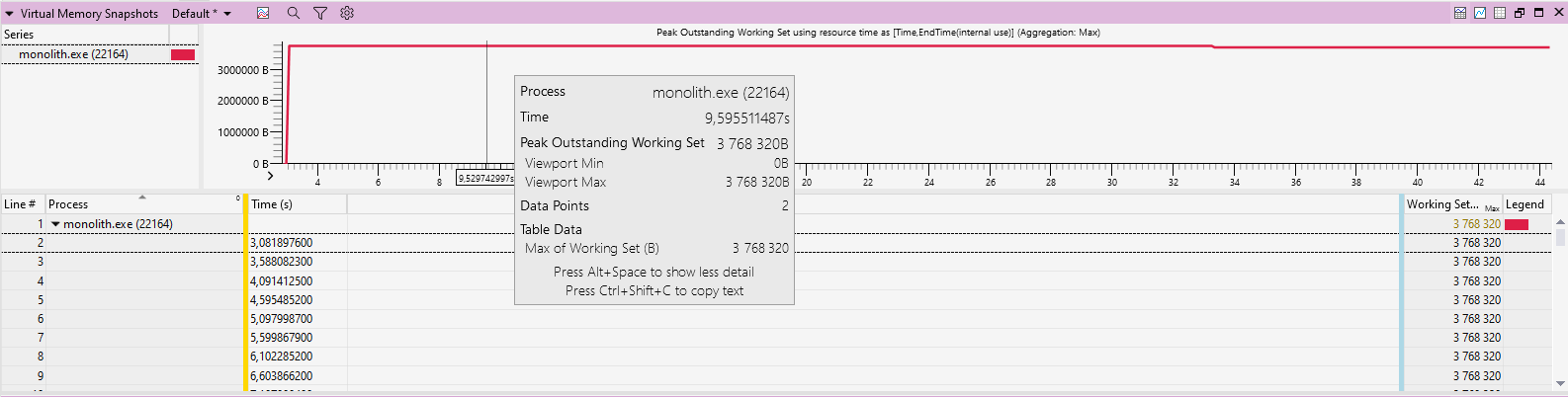
Изображение График системных вызовов

Можно заметить, что тут уже отсутствуют вызовы IO, так же почти в 2 раза уменьшилось количество переключений контекста в общем по системе, связано это с тем, что система не переключает контекст при ожидании ответа от диска.



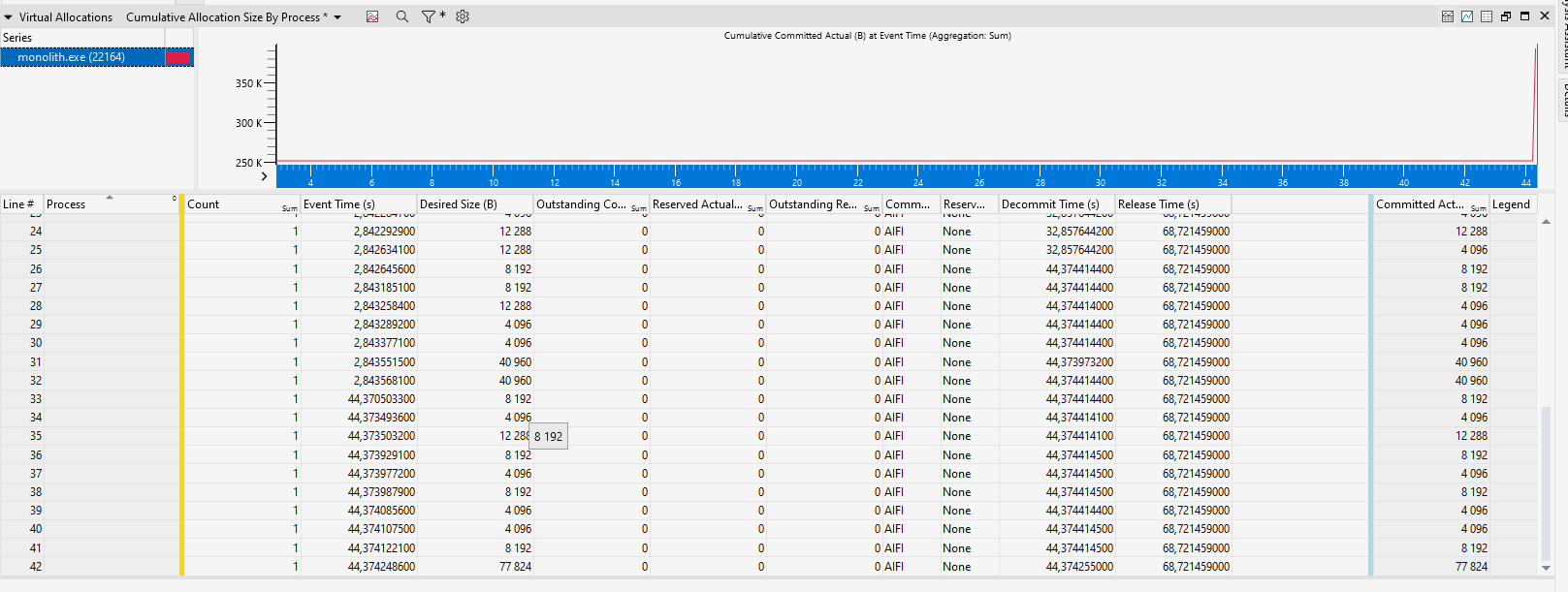
Изображение График использования процессора

Из графика видно, что использование процессора практически равно тому, что было в нагрузчике 1, можно предположить, что связано это с тем, что система имеет 12 потоков и при создании процесса он занимает 1 поток, что примерно равно 8%.



Изображение График занятия виртуальной памяти

Здесь видно, что нагрузчик выделяет себе определенную память, под кучу.



Изображение График выделения виртуальной памяти

Мое предположение, что сначала процесс выделяет себе память под кучу, а прямо перед смертью освобождает???



Изображение Количество context switch

Очень небольшое количество переключений контекста говорит нам о том, что программа выполняется без блокировок, то есть мы не ждем IO как в нагрузчике 1, к тому же на такое небольшое количество переключений влияет то, что система фактически простаивает из за чего ей не требуется частое переключение контекста, она может отдать другую работу на другие потоки.

### Пункт 2

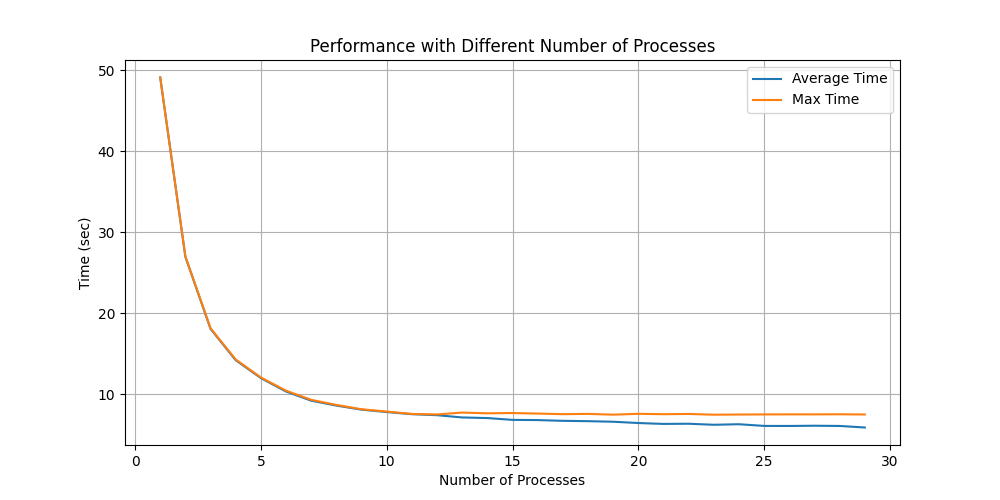
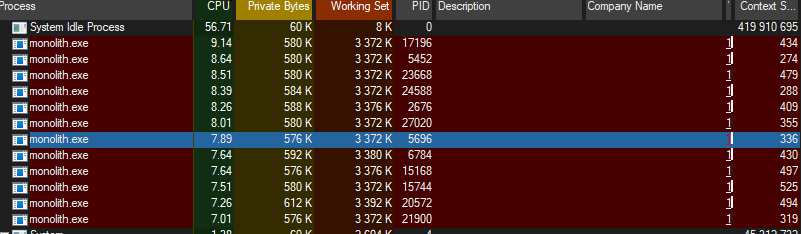


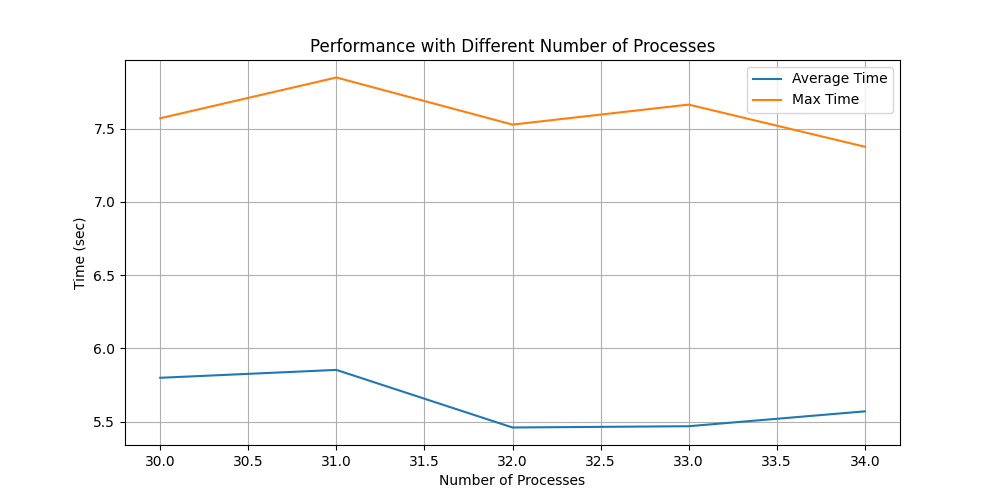
График время вычисления от количества процессов

Из графика можно заметить, что до 12 процессов скорость вычисления растет, после получается примерное плато, связано это с тем, что система имеет 6 ядер и 12 потоков, соответственно, после 12 процессов каждый из которых занимает один поток эффективность не растет и начинается просто борьба за ресурсы.



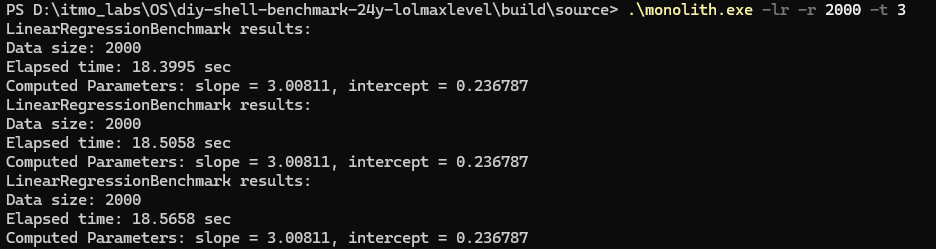
Видно, что количество CS с 12 процессами сильно увеличилось по сравнению с 1 процессом.

### Пункт 3



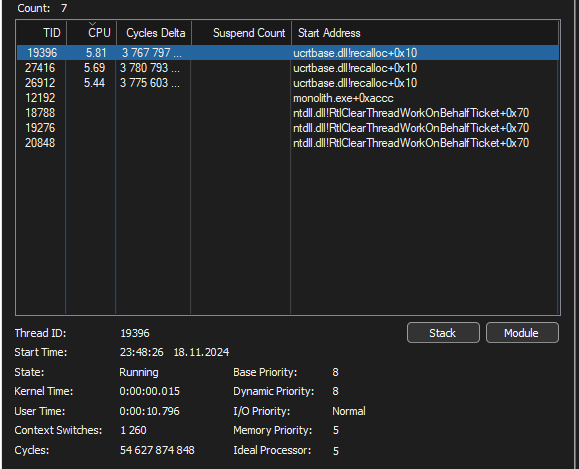
При увеличении количества процессов ситуация не сильно меняется, единственное что стоит заметить это падает количество CS, так как процессы просто на просто быстрее выполняют отведенную им работу.

### Пункт 4



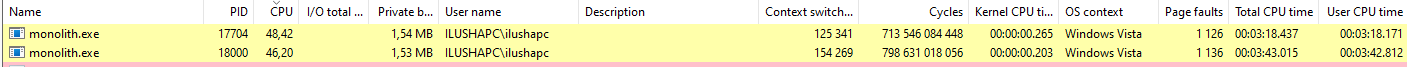
Изображение Работа нагрузчика при выполнении в потоках

Здесь видно, что время выполнения в 3 потоках примерно равно времени выполнения в 3 процессах, нагрузка остается примерно такой же…



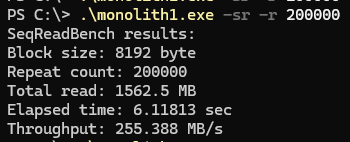
Изображение Все 3 потока в одном процессе

При запуске двух нагрузчиков на 12 потоков (критичное количество). Система начинает работать нестабильно, появляются “статтеры”, при этом процессорное время распределяется примерно 50\50 между двух нагрузчиков, но стоит заметить, что крайне сильно увеличивается количество CS, что логично.



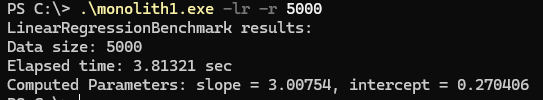
### Пункт 5

При компиляции программы с использованием агрессивной оптимизации (флаг O3)



Мы видим, что время работы нагрузчика уменьшилось почти в 2 раза, что логично, так как скорее всего компилятор понял, что данные не нужны и скорее всего просто меньше их перебрасывал по памяти, единственное что интересно это то, что он вообще послушно читал файл, хотя данные из него нигде не используются и он мог спокойно убрать весь код нагрузчика)

Для второго нагрузчика ситуация обстоит сильно интересней



Здесь агрессивная оптимизация отыгралась на полную и сократила время выполнения в 10 раз. Скорее всего оптимизатор каким-то образом развернул мои бесконечные циклы за счет чего ему удалось так сильно сократить время работы.

# **Вывод**

По итогу выполнения работы, я узнал о таких полезных утилитах WPA и WPR, которые, скорее всего буду использовать в дальнейшем для профилирования своей системы. Так же познакомился поближе с набором утилит SysInternals. Наглядно пронаблюдал как работают флаги оптимизации в компиляторах. Получил опыт разработки нагрузочных бенчмарков в C++. А самое главное, интересно провел время и лучше разобрался в своей любимой и прекрасной системе как Windows.